PIE Tech

POLLACHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE and Affiliated to Anna University) sky is the limit

Department of Artificial Intelligence and Data Science

Regulation 2021

II Year – III Semester

AD3251 - Design Analysis of Algorithm

Unit I Introduction

Notion of an algorithm

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem,i.e.,for obtaining a required output for any legitimate input in a finite amount of time.

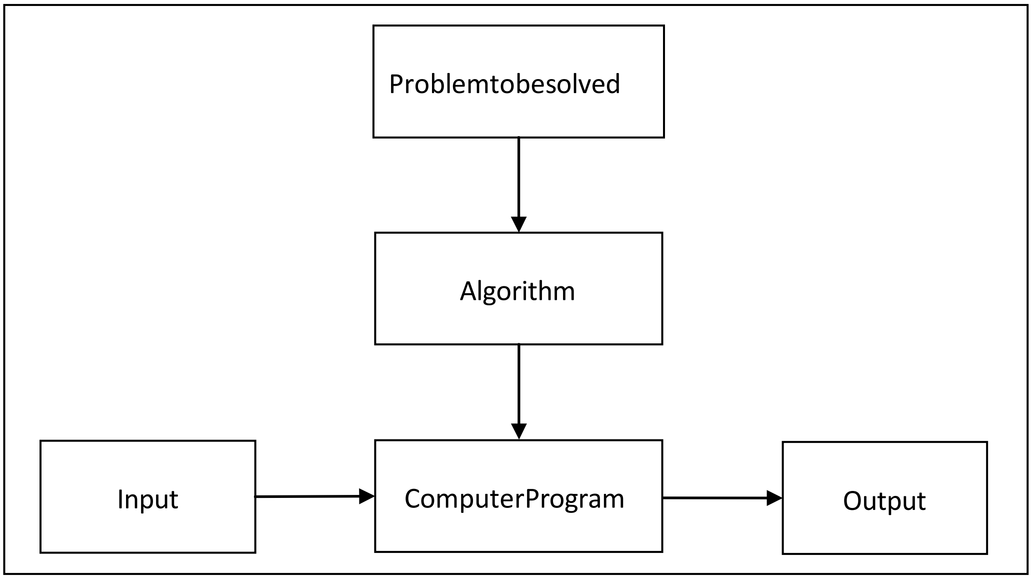


Figure1.1the notion of the algorithm.

It is a step by step procedure with the input to solve the problem in a finite amount of timeto obtain the required output.

Thenotionofthealgorithmillustratessomeimportantpoints:

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms forthesameproblemcanbebasedonverydifferentideasandcansolvethe problem with dramatically different speeds.

Characteristicsofan algorithm:

Input: zero/morequantitiesareexternallysupplied**.**

Output: at least one quantity is produced.

Definiteness: eachinstructionisclearand unambiguous.

Finiteness: if the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.

Efficiency: every instruction must be very basic and runs in short time.

Stepsforwritinganalgorithm:

1. Analgorithmisaprocedure.ithastwoparts;thefirstpartis**head**andthesecondpartis **Body**.
2. Theheadsectionconsistsofkeyword**algorithm**andnameofthealgorithmwith Parameterlist.e.g.algorithmname1(p1,p2,…,p3)

   Theheadsection alsohasthe following:

   //problemdescription:

   //input:

   //output:
3. Inthebody ofanalgorithmvariousprogrammingconstructslike**if,for,while**andsome statements like assignments are used.
4. Thecompoundstatementsmaybeenclosedwith**{**and**}**brackets.**if**,**for**,**while**canbe closed by **endif**, **endfor**, **endwhile** respectively. Proper indention is must for block.
5. Commentsarewrittenusing//atthe beginning.
6. The**identifier**shouldbeginbyaletterandnotbydigit.itcontainsalphanumericletters after first letter. No need to mention data types.
7. Theleftarrow**"**←**"**usedasassignmentoperator.e.g.v←10
8. **Boolean**operators(true,false),**logical**operators(and,or,not)and**relational** Operators(<,<=,>,>=,=,≠,<>)arealsoused.
9. Inputand outputcanbe doneusing **read**and**write**.
10. **Array[]**,**ifthen elsecondition**,**branch**and**loop**canbealsousedinalgorithm.

Example:

The greatest common divisor(gcd) of two nonnegative integers *m* and *n* (not-both-zero), denoted gcd*(m, n),* is defined as the largest integer that divides both *m* and *n* evenly, i.e., with a remainder of zero.

Euclid's algorithm is based on applying repeatedly the equality gcd*(m, n)* = gcd*(n, m* mod *n),* where*m*mod *n* is theremainderofthedivision of *m*by*n,* until *m* mod *n*is equal to 0.sincegcd*(m,* 0*)* = *m*, the last value of *m* is also the greatest common divisor of the initial *m* and *n*.

Gcd*(*60,24*)*can be computedas follows:gcd*(*60,24*)*=gcd*(*24,12*)*=gcd*(*12,0*)*=12.

Euclid'salgorithmforcomputinggcd*(m, n)*insimplesteps

Step1if*n*=0,returnthe valueof*m*astheanswerandstop;otherwise,proceedtostep2.

Step2divide*m*by*n*andassignthevalueofthe remainderto *r*.

Step3assignthevalue of*n*to*m*andthevalueof*r*to*n*.gotostep1.

Euclid'salgorithmforcomputinggcd*(m,n)*expressedinpseudocode

```
Algorithm euclid_gcd(m,n)
    //computes gcd(m,n)byeuclid'salgorithm
    //input:twononnegative,not-both-zerointegers mandn
    //output:greatestcommondivisorofmandn
    While n≠0 do
        R←mmodn
        m←n
        N←r
    Return m
```

Fundamentalsofalgorithmicproblemsolving

A sequence of steps involved in designing and analyzing an algorithm is shown in the figure Below.

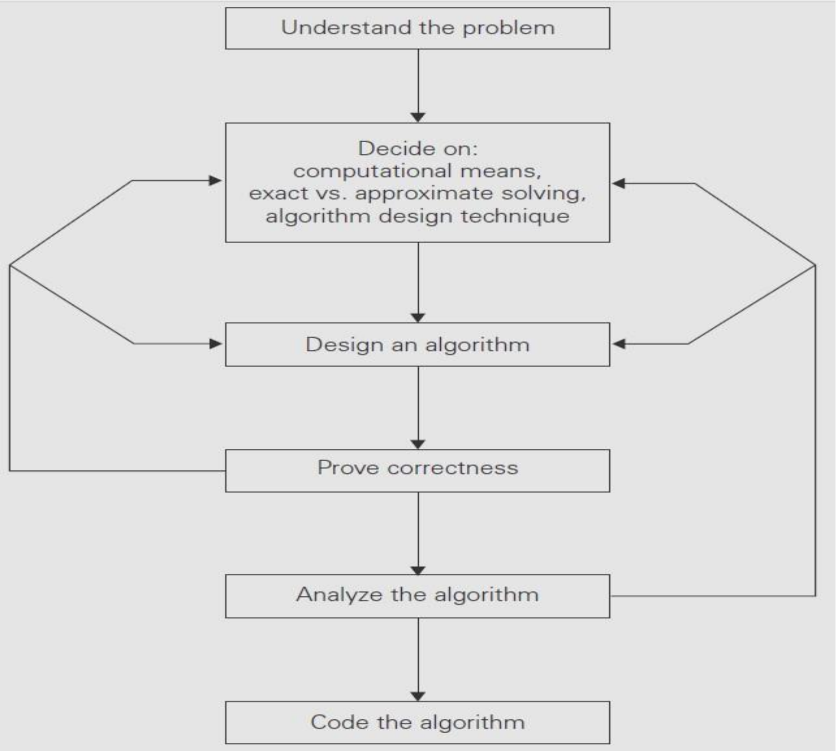


Figure1.2algorithm design and analysis process.

(i) Understandingtheproblem

- This is thefirst stepin designingof algorithm.
- Read the problem's description carefullyto understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problemtypes anduse existingalgorithm tofind solution.
- Input(***instance***)totheproblem andrangeoftheinput getfixed.

(ii) decision making

Thedecision makingisdoneon the following:

(a) Ascertaining the capabilities of the computation al device

- In ***random-access machine*** (***ram***), instructions are executed one after another (the central assumption is that one operation at a time). Accordingly, algorithmsdesigned to be executed on such machines are called ***sequential algorithms***.
- In some newer computers, operations are executed **concurrently**, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***.
- Choiceofcomputationaldeviceslikeprocessorandmemoryismainlybasedon **Spaceandtimeefficiency**

(b) Choosingbetweenexactandapproximateproblemsolving

- Thenextprincipaldecisionistochoosebetweensolvingtheproblemexactlyor solving it approximately.
- An algorithm used to solve the problem exactly and produce correct result is called an **exact algorithm**.
- Iftheproblemissocomplexandnotabletogetexactsolution,thenwehaveto chooseanalgorithmcalledan**approximationalgorithm**.i.e.,producesan

Approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(c) Algorithmdesigntechniques

- An ***algorithm design technique*** (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- **Algorithms+*DataStructures=Programs***
- Though algorithms and data structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- **Implementation** of algorithm is possible only with the help of algorithms and data structures
- **Algorithmic strategy / technique / paradigm** are a general approach by which many problems can be solved algorithmically. E.g., brute force, divide and conquer, dynamic programming, greedy technique and so on.

(iii) Methods of specifying an algorithm

Therearethreewaystospecifyanalgorithm.they are**:**

a. Naturallanguage
b. Pseudocode
c. Flowchart

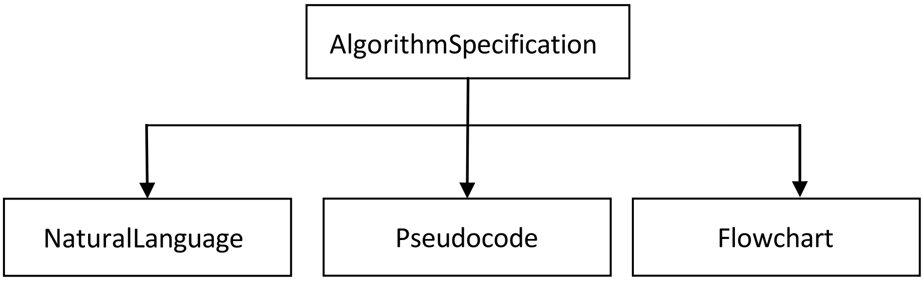


Figure1.3algorithm specifications

Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. Naturallanguage

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example:analgorithmtoperformadditionoftwo numbers.

Step1: Read the first number, say a.
Step2: Read the first number, say b.
Step3: Add the above two numbers and store the result in c.
Step 4: Display the result from c.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of pseudocode.

b. Pseudocode

- Pseudocodeisamixtureofanaturallanguageandprogramminglanguageconstructs. Pseudocode is usually more precise than natural language.
- For assignment operation left arrow "←", for comments two slashes "//",**if** condition**, for, while** loops are used.

```
ALGORITHMSum(a,b)
        //ProblemDescription:Thisalgorithmperformsadditionoftwo numbers
        //Input:Twointegersaandb
        //Output:Additionoftwo integers
        c←a+b
        returnc
```

Thisspecification is moreuseful forimplementationofanylanguage.

c. Flowchart

In the earlier days of computing, the dominant method for specifying algorithms was a ***flowchart***, this representation technique has proved to be inconvenient.

Flowchart is agraphical representation of an algorithm. It is a amethod of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

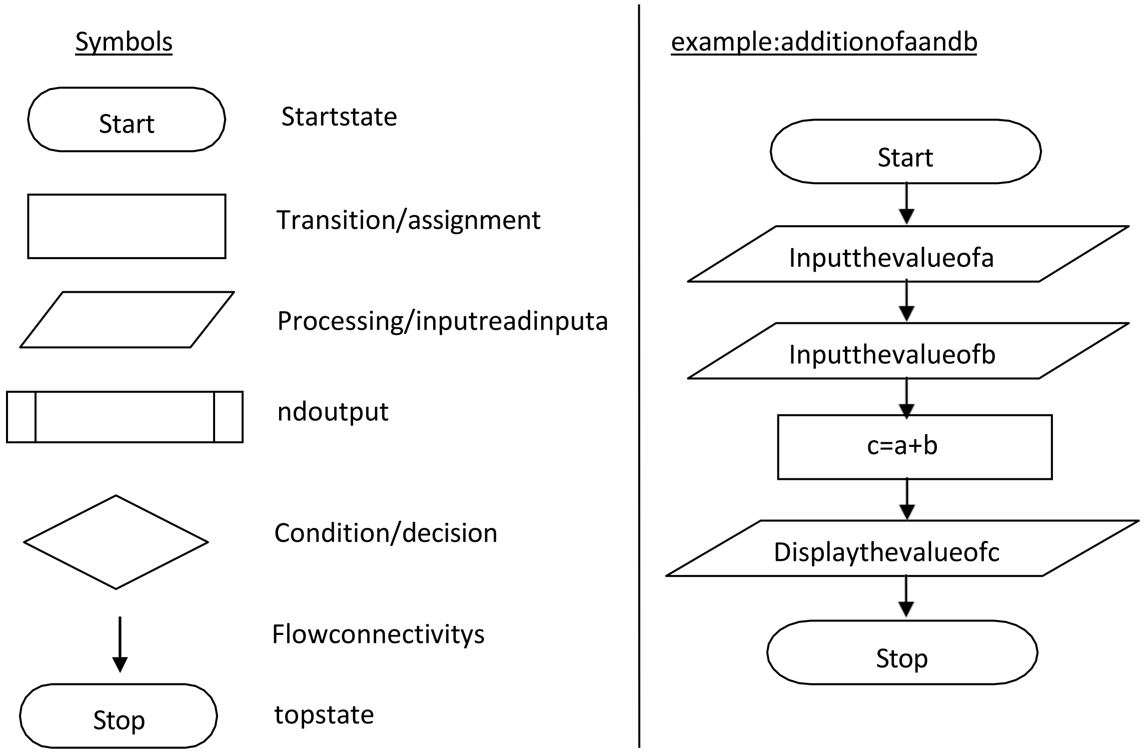


Figure1.4 flowchartsymbolsandexamplefortwointeger addition.

(iv) Provinganalgorithm'scorrectness

- Once an algorithmhasbeenspecifiedthenits ***correctness***mustbe proved.
- An algorithm must yields a required **result** for every legitimate input in a finite amount oftime.

- Forexample,thecorrectnessofeuclid’salgorithmforcomputingthegreatestcommon Divisors tems from the correctness of the equality gcd *(m,n)*= gcd*(n,m*mod*n).*
- Acommon technique for proving correctness is to use **mathematicalinduction**becausean Algorithm’siterationsprovideanaturalsequenceofstepsneeded forsuch proofs.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The **error** produced by the algorithm should not exceed a predefinedlimit.

(v) Analyzingan algorithm

- For an algorithm the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency. They are:
- ***Timeefficiency***,indicating how fast the algorithm runs,and
- ***Spaceefficiency***, indicating how much extra memory it uses.
- The efficiency of an algorithm is determined by measuring both time efficiency and spaceefficiency.
- Sofactorstoanalyzeanalgorithmare:
    - Timeefficiencyof analgorithm
    - Spaceefficiencyofanalgorithm
    - Simplicityof analgorithm
    - Generalityof analgorithm

(vi) Codinganalgorithm

- The coding / implementation of an algorithm is done by a suitable programming languagelike c, c++, java.
- The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The algorithm power should not reduced by inefficient implementation.
- Standard tricks like computing a **loop’s invariant** (an expression that does not change its value) outside the loop, collecting **common subexpressions**, replacing expensiveoperationsbycheap ones,selectionofprogramminglanguageandso onshouldbeknownto the programmer.
- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by **orders of magnitude**. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
- It is very essential to write an **optimized code (efficient code)**to reduce the burden of compiler.

Importantproblemtypes

Themostimportantproblemtypesare:

(i). Sorting
(ii). Searching
(iii). Stringprocessing
(iv). Graphproblems
(v). Combinatorialproblems
(vi). Geometricproblems
(vii). Numericalproblems

(i) Sorting

- The ***sorting problem*** is to rearrange the items of a given list in nondecreasing (ascending) order.
- Sorting can be done on numbers,characters,strings or records.
- To sort student records in alphabetical order of names or by student number or by studentgrade-point average. Such a specially chosen piece of information is called a ***key***.
- Analgorithmissaidto be**in-place**ifit doesnotrequireextramemory,e.g.,quicksort.
- Asortingalgorithmiscalled**stable**ifitpreservestherelativeorderofanytwoequal elements in its input.

(ii) Searching

- The ***searching problem*** deals with finding a given value,called a ***searchkey***,in a givenset.
- E.g.,ordinary linear search and fast binary search.

(iii) Stringprocessing

- A***string***isasequenceofcharacters froman alphabet.
- Strings comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and genesequences, which can be modeled by strings of characters from the four-character alphabet {a, c, g, t}. It is very useful in bioinformatics.
- Searchingfor agivenwordinatextiscalledstringmatching

(iv) Graphproblems

- A***graph***isacollectionofpointscalledvertices, some of which are connected by line segments called edges.
- Some of the graph problems are graph traversal, shortest path algorithm, topological sort, traveling salesman problem and the graph-coloring problem and so on.

(v) Combinatorialproblems

- Theseareproblems that ask, explicitlyorimplicitly, to finda combinatorial object suchas a permutation, a combination, or a subset that satisfies certain constraints.
- A desired combinatorial object may also be required to have some additional property suchs a maximum value or a minimum cost.
- Inpractical,thecombinatorialproblemsarethemostdifficultproblemsincomputing.
- Thetravelingsalesmanproblemandthegraphcoloringproblemareexamplesof ***Combinatorial problems***.

(vi) Geometricproblems

- ***Geometricalgorithms***dealwithgeometricobjectssuchaspoints,lines,and polygons.
- Geometricalgorithmsareusedincomputergraphics,robotics,and tomography.
- The***closest-pairproblemandt***he***convex-hullproblemarecomesunderthiscategory.***

(vii) Numericalproblems

- ***Numericalproblems***areproblemsthatinvolvemathematicalequations,systemsof equations, computing definite integrals, evaluating functions, and so on.
- Themajorityof such mathematical problemscanbesolved only approximately.

Fundamentalsoftheanalysisofalgorithmefficiency

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

a. Analysisframework.
b. Asymptoticnotationsandits properties.
c. Mathematicalanalysisforrecursive algorithms.
d. Mathematicalanalysisfornon-recursivealgorithms.

Analysis framework

Therearetwo kindsofefficienciesto analyzetheefficiencyofanyalgorithm. They are:

- ***Timeefficiency***,indicatinghow fastthealgorithmruns,and
- ***Spaceefficiency***,indicatinghow muchextramemoryit uses.

Thealgorithmanalysisframeworkconsistsofthe following:

- Measuringaninput'ssize
- Units formeasuringrunningtime
- Ordersofgrowth
- Worst-case,best-case,andaverage-caseefficiencies

(i) Measuringaninput'ssize

- An algorithm's efficiency is defined as a function of some parameter *n* indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward.for example, it will be the size of the list for problems of sorting, searching.
- For the problem of evaluating a polynomial$p(x) = a_nx^n + \dots + a_0$ of degree *n,* the size ofthe parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.
- In computing the product of two *n* × *n* matrices, the choice of a parameter indicating an input size does matter.
- Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.
- In measuring input size for algorithms solving problems such as checking primality of a positive integer *n*. The input is just one number.
- Theinputsizebythe number*b* ofbitsin the*n*'sbinaryrepresentationis $b=(\log_2 n)+1$.

(ii) Unitsformeasuringrunning time

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

- Dependenceonthespeedofaparticular computer.
- Dependenceonthequalityof aprogramimplementingthe algorithm.
- Thecompilerusedingeneratingthemachinecode.
- Thedifficultyof clockingtheactual runningtimeof theprogram.

So,weneedmetrictomeasurean*algorithm*'sefficiencythatdoesnotdependonthese Extraneousfactors.

Onepossibleapproachisto***countthenumberoftimeseachofthealgorithm'soperations Isexecuted***.thisapproachisexcessivelydifficult.

Themostimportantoperation(+,-,*,/)ofthealgorithm,calledthe***basicoperation***. Computingthenumberoftimesthebasicoperationisexecutediseasy.thetotalrunningtimeis basicoperations count.

(iii) Ordersof growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For example, the greatest common divisor of two small numbers, it is not immediatelyclear how much more efficient euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only.
- Forlargevaluesof*n*,itisthefunction'sorderofgrowththatcountsjustlikethetable1.1, Whichcontainsvaluesofafewfunctions particularlyimportantforanalysisofalgorithms.

Table1.1values(approximate)ofseveralfunctionsimportantforanalysisofalgorithms

N	$\sqrt{n}$	*Log₂n*	*N*	*N log₂n*	N^2	N^3	2^n	*N!*
1	**1**	**0**	**1**	**0**	**1**	**1**	**2**	**1**
2	**1.4**	**1**	**2**	**2**	**4**	**4**	**4**	**2**
4	**2**	**2**	**4**	**8**	**16**	**64**	**16**	**24**
8	**2.8**	**3**	**8**	$2.4•10^1$	**64**	$5.1•10^2$	$2.6•10^2$	$4.0•10^4$
10	**3.2**	**3.3**	**10**	$3.3•10^1$	10^2	10^3	10^3	$3.6•10^6$
16	**4**	**4**	**16**	$6.4•10^1$	$2.6•10^2$	$4.1•10^3$	$6.5•10^4$	$2.1•10^{13}$
10^2	**10**	**6.6**	10^2	$6.6•10^2$	10^4	10^6	$1.3•10^{30}$	$9.3•10^{157}$
10^3	31	10	10^3	$1.0•10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3•10^5$	10^8	10^{12}		
10^5	$3.2•10^2$	17	10^5	$1.7•10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0•10^7$	10^{12}	10^{18}		

(iv) Worst-case,best-case,andaverage-caseefficiencies

consider sequential search algorithm some search key *k*

algorithm *sequentialsearch(a*[0..*n* - 1]*, k)*

```
    //searchesforagivenvalueinagivenarraybysequentialsearch
    //input:anarraya[0..n-1]andasearchkeyk
    //output:theindex of thefirst element inathatmatches kor-1 iftherearено
    //            matchingelements
    I ←0
    Whilei<nanda[i]≠kdo
            I←i+1
    ifi<nreturni
    else return -1
```

Clearly,therunningtimeofthisalgorithmcanbe quitedifferentforthesamelistsize*n*.

In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

Worst-caseefficiency

- The***worst-caseefficiency*** ofanalgorithmisitsefficiencyfortheworstcaseinputofsize*n*.
- Thealgorithm runs thelongest amongall possibleinputs ofthat size.
- Fortheinputofsize*n*,therunningtimeis$c_{worst}(n)=n$.

Bestcaseefficiency

- The***best-caseefficiency***ofanalgorithm isits efficiencyforthebestcaseinputof size*n*.
- Thealgorithm runs thefastest amongallpossible inputs ofthat sizen.
- In sequential search, if we search a first element in list of size *n*. *(i.e.*first element equal toa search key),then the running time is$c_{best}(n) = 1$

Averagecase efficiency

- Theaveragecaseefficiencyliesbetweenbestcaseandworstcase.
- Toanalyzethealgorithm’saveragecaseefficiency,wemustmakesomeassumptionsabout Possibleinputsofsize*n*.
- Thestandardassumptionsarethat
    - Theprobabilityofasuccessfulsearchis equal to$p$ ($0\leq p\leq 1$)and
    - The probability of the first match occurring in the *i*th position of the list is the same for every *i*.

$$\begin{aligned}C_{avg}(n) &= [1\cdot\frac{p}{n}+2\cdot\frac{p}{n}+\cdots+i\cdot\frac{p}{n}+\cdots+n\cdot\frac{p}{n}]+n\cdot(1-p)\\ &= \frac{p}{n}[1+2+\cdots+i+\cdots+n]+n(1-p)\\ &= \frac{p}{n}\frac{n(n+1)}{2}+n(1-p)=\frac{p(n+1)}{2}+n(1-p).\end{aligned}$$

Yet anothertypeofefficiencyis called ***amortized efficiency***. It applies not to asinglerunof an algorithm but rather to a sequence of operations performed on the same data structure.

Asymptoticnotationsanditsproperties

Asymptoticnotationisanotation,whichisusedtotakemeaningfulstatementaboutthe efficiency of a program.

The efficiency analysisframeworkconcentrateson the order ofgrowthof analgorithm’s basic operation count as the principal indicator of the algorithm’s efficiency.

Tocompareandranksuchordersofgrowth,computerscientistsusethreenotations,they Are:

- O-bigoh notation
- Ω-bigomega notation
- Θ-bigthetanotation

Let*t(n)*and*g(n)*canbeanynonnegativefunctionsdefinedonthesetofnaturalnumbers. Thealgorithm’srunningtime*t(n)*usuallyindicatedbyitsbasicoperationcount*c(n)*,and*g(n),* Somesimplefunction tocomparewiththecount.

Example1:

$n \in O(n^2)$,	$100n+5 \in O(n^2)$,	$\frac{1}{2}n(n-1) \in O(n^2)$.
$n^3 \notin O(n^2)$,	$0.00001n^3 \notin O(n^2)$,	$n^4+n+1 \notin O(n^2)$.
$n^3 \in \Omega(n^2)$,	$\frac{1}{2}n(n-1) \in \Omega(n^2)$,	but $100n+5 \notin \Omega(n^2)$.

Where$g(n)=n^2$.

(i) O-big oh notation

A function t*(n)* is said to be in *o(g(n)),* denoted $t\ (n)\in O(g(n))$, if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n,* i.e., if there exist some positive constant *c* and some nonnegative integer n_0 such that

$$t(n)\leq cg(n) for all n\geq n_0.$$

Where*t(n)*and*g(n)*arenonnegativefunctionsdefinedonthesetofnaturalnumbers. O = asymptotic upper bound = useful for worst case analysis = loose bound

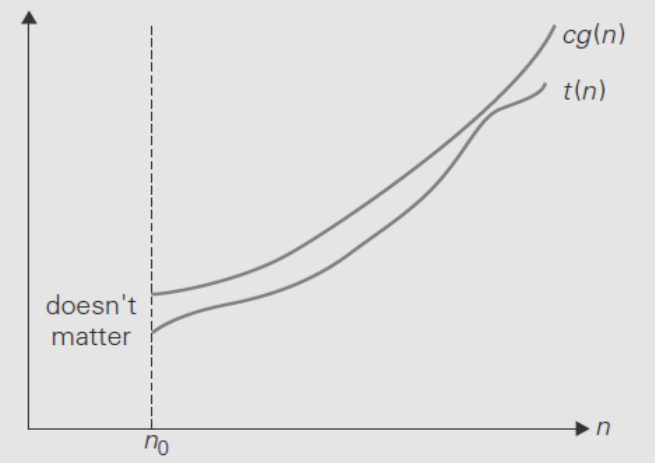


Figure1.5big-ohnotation: $t(n)\in O(g(n))$.

Example2: provetheassertions$100n+5\in O(n^2)$.

Proof: $100n+5\leq 100n+ n$ (forall $n\geq 5$)

$=101n$

$\leq 101n^2$($\ni n\leq n^2$)

Since,thedefinitiongivesusalotoffreedominchoosingspecificvaluesforconstants**c** And**n_0**. Wehavec=101 and n_0=5

Example3: provetheassertions$100n+5\in O(n)$.

Proof: $100n+5\leq 100n+ 5n$ (forall$n\geq 1$)

$=105n$

I.e., $100n+5\leq 105n$

i.e., $t(n)\leq cg(n)$

ѳ$100n+5\in O(n)$withc=105andn_0=1

(ii) Ω-bigomega notation

A function *t(n)* is said to be in Ω*(g(n)),* denoted *t(n)* ∈ Ω*(g(n)),* if *t(n)* is bounded below by some positive constant multiple of *g(n)* for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$T(n)\geq cg(n) \text{ foralln} \geq n_0.$$

Where*t(n)*and*g(n)*arenonnegativefunctionsdefinedonthesetofnaturalnumbers.

Ω=asymptoticlowerbound=usefulforbestcaseanalysis= loose bound

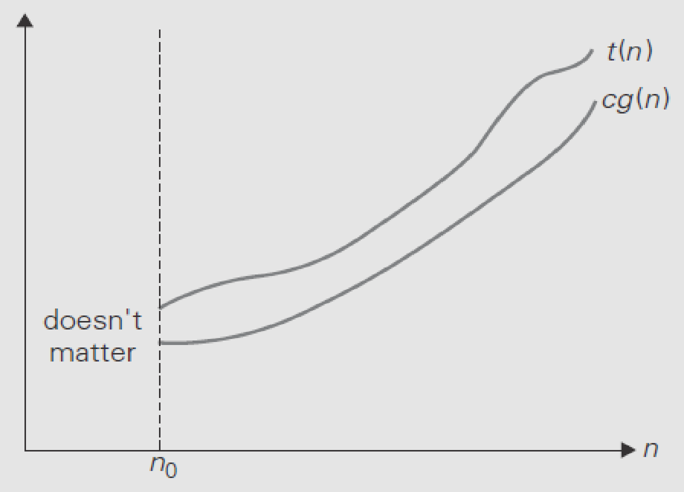


Figure1.6 big-omeganotation: *t(n)∈Ω(g(n))*.

Example4: provetheassertions$n^3+10n^2+4n+2\in\Omega(n^2)$.

Proof:$n^3+10n^2+4n+2\geq n^2$ (forall $n\geq0$)

I.e.,bydefinitiont(n)≥cg(n),wherec=1and$n_0=0$

(iii) Θ-big theta notation

Afunctiont(n)issaid to beinθ*(g(n))*, denoted t(n)∈ θ(g(n)),ift(n)isboundedbothabove and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$C_2g(n)\leq t(n)\leq c_1g(n)\text{forall } n\geq n_0.$$

Where*t(n)*and*g(n)*arenonnegativefunctionsdefinedonthesetofnaturalnumbers.

Θ=asymptotictightbound =usefulforaverage caseanalysis

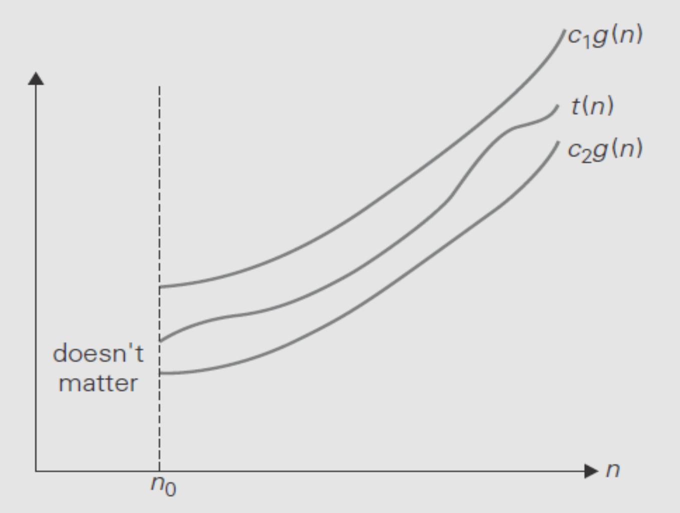


Figure1.7 big-thetanotation:*t(n)*∈θ*(g(n))*.

Example5: provetheassertions$\frac{1}{2}n(n-1)\in\theta(n^2)$.

Proof:firstprovethe rightinequality(theupper bound):

$$\frac{1}{2}n(n-1)=\frac{1}{2}n^2-\frac{1}{2}n\leq\frac{1}{2}n^2\text{forall}n\geq0.$$

Second,weprovetheleft inequality(thelowerbound):

$$\frac{1}{2}n(n-1)=\frac{1}{2}n^2-\frac{1}{2}n\geq\frac{1}{2}n^2-[\frac{1}{2}n][\frac{1}{2}n]\text{forall}n\geq2.$$

ӫ $\frac{1}{2}n(n-1)\geq\frac{1}{4}n^2$

I.e., $\frac{1}{4}n^2\leq\frac{1}{2}n(n-1)\leq\frac{1}{2}n^2$

Hence, $\frac{1}{2}n(n-1)\in\theta(n^2)$, where $c2=\frac{1}{4}$, $c1=\frac{1}{2}$ and $n_0=2$

Note:asymptotic notation can be thought of as "relational operators" for functions similar tothecorresponding relational operators for values.

=⇒θ(), ≤⇒o(), ≥⇒ω(), <⇒o(), >⇒ω()

Usefulpropertyinvolvingtheasymptoticnotations

The following property, in particular, is useful in analyzing algorithms that comprise twoconsecutively executed parts.

Theorem:if$t_1(n)$∈o($g_1(n)$)and$t_2(n)$∈o($g_2(n)$),then$t_1(n)+t_2(n)$∈o(max{$g_1(n)$,$g_2(n)$}). (the analogous assertions are true for the **Ω** and θ notations as well.)

Proof: the proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a_2 \leq 2$ max{b_1, b_2}.

Since $t_1(n)$ ∈o($g_1(n)$), there exist some positive constant c_1 and some nonnegative integern_1 such that

$T_1(n)\leq c_1g_1(n)$forall $n \geq n_1$.

Similarly,since$t_2(n)$∈o($g_2(n)$),

$T_2(n)\leq c_2g_2(n)$forall $n \geq n_2$.

Letusdenotec_3=max{c_1,c_2}andconsidern≥max{n_1,n_2}sothatwecanuse both inequalities. Adding them yields the following:

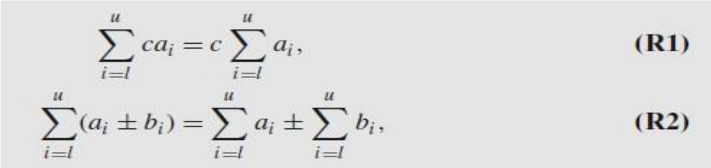
$$\begin{aligned}T_1(n)+t_2(n) &\leq c_1g_1(n)+c_2g_2(n)\\ &\leq c_3g_1(n)+c_3g_2(n)\\ &=c_3[g_1(n)+g_2(n)]\\ &\leq c_3 2\max\{g_1(n), g_2(n)\}.\end{aligned}$$

Hence,$t_1(n)+t_2(n)$∈o(max{$g_1(n)$,$g_2(n)$}),withtheconstantscand$n_0$requiredbythe definition o being $2c_3$ = 2 max{c_1, c_2} and max{n_1, n_2}, respectively.

Thepropertyimpliesthatthealgorithm’soverallefficiencywillbedeterminedbythepart Withahigherorderofgrowth,i.e.,itsleastefficient part.

ӫ$t_1(n)$∈o($g_1(n)$)and$t_2(n)$∈o($g_2(n)$),then$t_1(n)+t_2(n)$∈ o(max{$g_1(n)$,$g_2(n)$}).

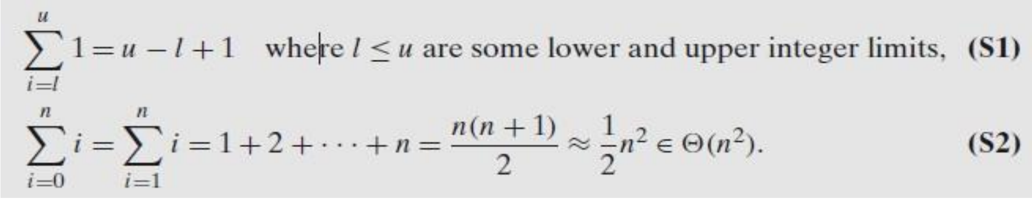
Basicrulesofsummanipulation



$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

Summationformulas



$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \leq u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Mathematicalanalysisforrecursivealgorithms

Generalplanforanalyzingthetimeefficiencyofrecursivealgorithms

1. Decideon a*parameter*(orparameters)indicatinganinput’s size.
2. Identifythealgorithm’s*basic operation.*
3. Check whether the *number of times the basic operation is executed* can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case*efficiencies* must be investigated separately.
4. *Set up a recurrence relation*, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solvetherecurrenceor, atleast,ascertainthe *orderofgrowth*ofitssolution.

Example1:computethefactorialfunctionf(n)=n!Foranarbitrarynonnegativeintegern. Since*n!=1*•....•(*n*− *1*)•*n*=(*n*−*1*)*!*•*n*,forn≥1and0!=1by definition,wecancompute *f*(*n*) = *f*(*n* − *1*) • *n* with the following recursive algorithm. **(nd 2015)**

algorithm *f(n)*

```
    //computesn!Recursively
    //input:anonnegativeintegern
    //output:thevalue ofn!
    If n=0return1
    Elsereturnf(n−1)*n
```

Algorithmanalysis

- Forsimplicity,weconsider*n*itselfasanindicatorofthisalgorithm’sinputsize.i.e.1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote*m(n)*.sincethefunction*f(n)*iscomputed accordingtotheformula *f(n)*=*f*(*n*−1)•*n* for *n* >0.
- Thenumberofmultiplications*m(n)*neededto computeitmust satisfythe equality

$$M(n)=m(n-1)\ \underset{\substack{\uparrow\\ \text{Tocompute}\\ \text{f(n-1)}}}{}\quad + \quad \underset{\substack{\uparrow\\ \text{Tomultiply}\\ \text{f(n-1)byn}}}{1}\quad \text{forn>0}$$

M(n−1*)*multiplicationsarespenttocompute*f(n*−1*)*,andonemoremultiplicationis needed to multiply the result by *n*.

Recurrencerelations

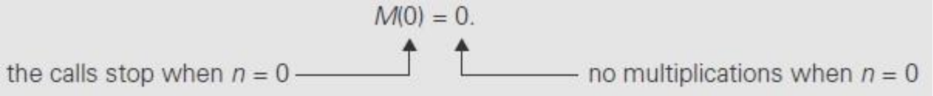
The last equation defines the sequence m(n) that we need to find. This equation defines m(n) not explicitly, i.e., as a function of n, but implicitlyas a function of its value at another point, namely n − 1. Such equations are called *recurrence relations* or *recurrences*.

Solvetherecurrencerelation$M(n)=M(n-1)+1$,i.e.,tofindanexplicitformulafor *M(n)*intermsof*n*only.

To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

If *n*=0**return**1.

This tells us two things. First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence m(n) defined is 0. Second, by inspecting the pseudocode’s exiting line, we can see that when n = 0, the algorithm performs no multiplications.



$M(0) = 0.$

the calls stop when $n = 0$

no multiplications when $n = 0$

Thus,therecurrencerelationandinitialconditionforthealgorithm’snumberofmultiplications *M(n)*:

M(n)=m(n−1)+1forn>0, m(0)
= 0 for n = 0.

Methodofbackwardsubstitutions

M(n) =*m(n*−1*)*+1 substitute*m(n* −1*)*=*m(n* −2*)*+1
=[*m(n*−2*)*+1]+1
=*m(n* −2*)*+2 substitute*m(n* −2*)*=*m(n* −3*)*+1
=[*m(n*−3*)*+1]+2
=*m(n* −3*)*+3
…
=m(n −i)+i
…
=m(n −n)+n
=n.

Therefore*m(n)=n*

Example 2: consider educational workhorse of recursive algorithms: the ***tower of hanoi*** puzzle. We have n disks of different sizes that can slide onto any of three pegs. Considera (source), b (auxiliary), and c (destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.

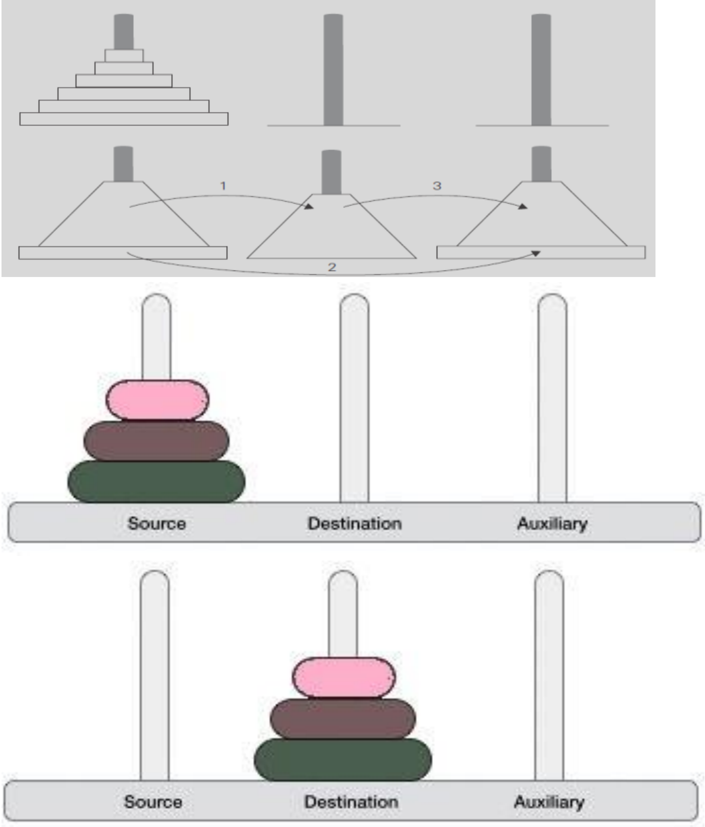


Figure1.8recursivesolutiontothetowerofhanoipuzzle.

```
Algorithmtoh(n,a,c,b)
    //movedisksfromsourcetodestinationrecursively
    //input: ndisksand3pegsa,b,andc
    //output:disksmovedtodestinationasinthesource order.
    Ifn=1
        Movediskfrom atoc
    Else
        Movetopn-1disksfromatobusingc toh(n - 1, a,
        b, c)
        Movetopn-1disksfrombtocusinga toh(n - 1, b,
        c, a)
```

Algorithmanalysis

Thenumberofmoves*m(n)*dependson*n*only,andwegetthefollowingrecurrenceequation for it:

$$m(n) = m(n - 1) + 1 + m(n - 1) \text{ for } n > 1.$$

Withtheobviousinitialcondition$m(1)$=1,wehavethefollowingrecurrencerelationforthe number of moves *m(n)*:

$$M(n)=2m(n-1)+1 \text{for} n>1,\ m(1) = 1.$$

Wesolvethisrecurrence bythesamemethodofbackward substitutions:

$M(n)=2m(n-1)+1$ sub.$m(n-1)=2m(n-2)+1$

$=2[2m(n-2)+1]+1$

$=2^2m(n-2)+2+1$ sub.$m(n-2)=2m(n-3)+1$

$=2^2[2m(n-3)+1]+2+1$

$=2^3m(n-3)+2^2+2+1$ sub.$m(n-3)=2m(n-4)+1$

$=2^4m(n-4)+2^3+2^2+2+1$

…

$=2^i m(n-i)+2^{i-1}+2i^{-2}+. .. +2+1=2^i m(n-i)+2^i-1.$

…

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$,

$m(n)=2^{n-1}m(n-(n-1))+2^{n-1}-1=2^{n-1}m(1)+2^{n-1}-1=2^{n-1}+2^{n-1}-1=2^n-1.$

Thus,wehavean exponentialtimealgorithm

Example 3: an investigation of a recursive version of the algorithm which finds the number of binary digits in the **binary representation** of a positive decimal integer.

```
Algorithmbinrec(n)
    //input:apositivedecimalintegern
    //output:thenumberofbinarydigitsinn'sbinaryrepresentation
    If n=1return1
    Elsereturnbinrec(⌊n/2⌋)+1
```

Algorithmanalysis

The number of additions made in computing *binrec*($\lfloor n/2 \rfloor$) is $a(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence $a(n) = a(\lfloor n/2 \rfloor) + 1$ for $n > 1$.

Then,theinitial conditionis*a(*1*)*=0.

Thestandardapproachto solvingsucharecurrenceistosolveitonlyfor *n* $=2^k a(2^k) =$

$a(2^{k-1}) + 1$ for $k > 0$,

$A(2^0)=0$.

Backwardsubstitutions

$A(2^k)=a(2^{k-1})+1$ substitutea$(2^{k-1})=a(2^{k-2})+1$

$=[a(2^{k-2})+1]+1=a(2^{k-2})+2$ substitutea$(2^{k-2})=a(2^{k-3})+1$

$=[a(2^{k-3})+1]+2=a(2^{k-3})+3$ ...

. . .

$=a(2^{k-i})+i$

. . .

$=a(2^{k-k})+k$.

Thus,weendupwitha$(2^k)=a(1)+k=k$,or,afterreturning totheoriginalvariablen$=2^k$and hence $k = \log_2 n$, $A(n)=\log_2 n \epsilon \theta(\log_2 n)$.

Mathematicalanalysisfornon-recursivealgorithms

Generalplanforanalyzingthetimeefficiencyofnonrecursivealgorithms

1. Decideon a*parameter*(orparameters)indicatinganinput’s size.
2. Identifythealgorithm’s*basicoperation*(intheinnermostloop).
3. Check whetherthe *number of times thebasicoperation is executed* depends onlyon thesize of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case *efficiencies* have to be investigated separately.
4. Setupa*sum*expressingthenumberoftimesthealgorithm’sbasicoperationisexecuted.
5. Using standard formulas and rules of sum manipulation either find a closed form formulafor the count or at the least, establish its *order of growth*.

Example 1: consider the problem of finding the value of **the largest element in a list of n numbers**. Assume that the list is implemented as an array for simplicity.

Algorithmmaxelement(a[0..n−1])

```
    //determinesthevalueofthelargestelementinagivenarray
    //input:anarraya[0..n−1] ofreal numbers
    //output:thevalueofthelargestelementin a
    Maxval ←a[0]
    Fori ←1 to n −1 do
        Ifa[i]>maxval
            Maxval←a[i]
    Returnmaxval
```

Algorithmanalysis

- Themeasureofan input’ssizehereisthenumber ofelementsinthe array,i.e., n.
- Therearetwooperations intheforloop’sbody:
    - Thecomparisona[i]>maxvaland
    - Theassignment maxval←a[i].

- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let c(n) denotes the number of times this comparison is executed. The algorithm makesone comparison on each execution of the loop, which is repeated for each value of theloop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for c(n) is calculated as follows:

$$c(n)=\sum_{i=1}^{n-1} 1$$

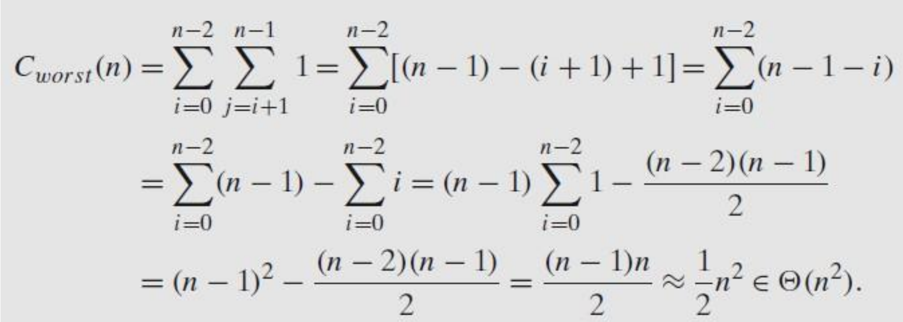
I.e.,sumup1 inrepeated n-1 times

$$c(n)=\sum_{i=1}^{n-1} 1=n-1\in\theta(\mathrm{n})$$

Example2:considerthe**elementuniquenessproblem**:checkwhetheralltheelementsina given array of n elements are distinct.

```
Algorithmuniqueelements(a[0..n−1])
    //determineswhetherall theelementsinagivenarrayaredistinct
    //input:an arraya[0..n−1]
    //output:returns "true"ifallthe elementsinaaredistinct and"false" otherwise
    Fori ←0 to n −2 do
        Forj ←i +1 to n −1 do
            Ifa[i]=a[j]returnfalse
    Returntrue
```

Algorithmanalysis

- Thenatural measureof theinput'ssizehereisagainn (thenumberofelementsin the array).
- Sincetheinnermostloopcontainsasingleoperation(thecomparisonoftwoelements),we Shouldconsideritasthe algorithm'sbasic operation.
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only**.**
- One comparison is made for each repetition of theinnermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n − 2.
- 

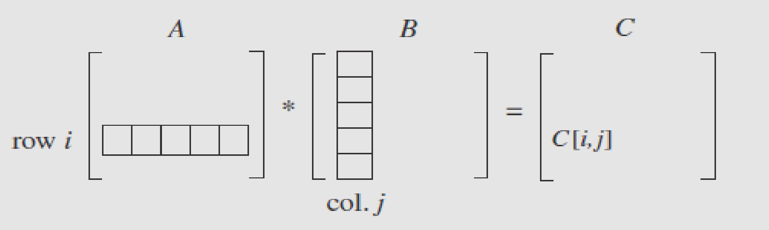
$$C_{worst}(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}[(n-1)-(i+1)+1]=\sum_{i=0}^{n-2}(n-1-i)$$

$$=\sum_{i=0}^{n-2}(n-1)-\sum_{i=0}^{n-2}i=(n-1)\sum_{i=0}^{n-2}1-\frac{(n-2)(n-1)}{2}$$

$$=(n-1)^2-\frac{(n-2)(n-1)}{2}=\frac{(n-1)n}{2}\approx\frac{1}{2}n^2\in\Theta(n^2).$$

Example3:considermatrixmultiplication.giventwon× nmatricesa andb,findthe time efficiencyofthedefinition-basedalgorithmforcomputingtheirproductc=ab.bydefinition,c

Isann×nmatrix whoseelementsarecomputed asthescalar(dot)productsof therowsofmatrix a and the columns of matrix b:



Wherec[i,j]=a[i,0]b[0,j]+...+a[i,k]b[k,j]+...+a[i,n−1]b[n−1,j]forevery pairof indices 0 ≤ i, j ≤ n − 1.

```
Algorithmmatrixmultiplication(a[0..n−1,0..n−1],b[0..n−1,0..n−1])
        //multipliestwosquarematricesofordernbythedefinition-basedalgorithm
        //input:twon×nmatricesaandb
        //output:matrixc=ab
        Fori ←0 to n −1 do
                Forj ←0to n −1 do
                        C[i, j]←0.0
                        Fork←0 to n − 1 do
                                C[i,j]←c[i,j ]+a[i, k]∗b[k, j]
        Returnc
```

Algorithmanalysis

- Aninput'ssizeismatrixordern.
- There are two arithmetical operations (multiplication and addition) in the innermost loop. But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications m(n) executed by the algorithm. Since this count depends only on the size of the input matrices, we do not have toinvestigate the worst-case, average-case, and best-case efficiencies separately.
- There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound n − 1.
- Therefore, the number of multiplications made for every pair of specific values of variablesi and j is

$$\sum_{k=0}^{n-1} 1.$$

Thetotalnumberofmultiplicationsm(n)isexpressedbythefollowingtriplesum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Now,wecan computethis sum byusingformula(s1) andrule (r1)

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Therunningtimeofthealgorithmonaparticularmachinem,wecandoit bytheproduct if we

$$T(n) \approx c_m M(n) = c_m n^3,$$

consider, time spent on the additions too, then the total time on the machine is

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

Example4thefollowingalgorithmfindsthenumberofbinarydigitsinthe**binary representation** of a positive decimal integer.

```
algorithm binary(n)
    //input:apositivedecimalintegern
    //output:thenumberofbinarydigitsinn'sbinaryrepresentation count ←1
    While n>1 do
        Count←count+1 n←$\lfloor n/2 \rfloor$
    Return count
```

Algorithmanalysis

- Aninput'ssizeisn.
- Theloop variabletakes ononlyafew valuesbetween itslower and upperlimits.
- Sincethevalueofnisabouthalvedoneachrepetitionoftheloop,theanswershouldbe about $\log_2 n$.
- Theexact formulafor the numberof times.
- Thecomparison$n>1$willbeexecutedisactually$\lfloor \log_2 n \rfloor+1$.

Unitiibruteforceanddivide-and-conquer

Bruteforce

Brute force is a straightforward approach to solving a problem, usually directly based onthe problem statement and definitions of the concepts involved.

Selectionsort,bubblesort,sequentialsearch,stringmatching,depth-firstsearchand breadth-first search, closest-pair and convex-hull problems can be solved by brute force.

Examples:

1. Computinga^n: a * a* a* … * a(n times)
2. Computingn! : then! Can becomputed as n*(n-1)* … *3*2*1
3. Multiplicationoftwomatrices:c=ab
4. Searchingakeyfromlistofelements(sequentialsearch)

advantages:

1. Bruteforceis applicable to averywidevarietyof problems.
2. Itisveryusefulforsolvingsmallsizeinstancesofaproblem,eventhoughitis inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical valuewith no limitation on instance size for sorting, searching, and string matching.

Selectionsort

- First scan the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.
- Thenscanthelist,startingwiththesecondelement,tofindthesmallestamongthelastn −1 elements and exchange it with the second element, putting the second smallest element inits final position in the sorted list.
- Generally, on the ***i***th pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among thelast n −i elements and swaps it with a_i:

$$\underbrace{A_0 \le a_1 \le . .. \le a_{i-1}}_{\textit{Intheir finalpositions}} | \underbrace{a_i, .. ., a_{min}, . .. , a_{n-1}}_{\textit{thelast } n - i \textit{ elements}}$$

- Aftern −1 passes, the list issorted.

Algorithmselectionsort(a[0..n−1])

```
//sortsagivenarraybyselectionsort
//input:anarraya[0..n−1]oforderableelements
//output:arraya[0..n−1] sortedinnondecreasingorder
For i ← 0 to n −2 do
    Min ← i
    For j ← i +1 to n −1 do
        If a[j]<a[min] min←j
    Swapa[i]anda[min]
```

|89 45 68 90 29 34 **17**

17 | 45 68 90 **29** 34 89

17 29 | 68 90 45 **34** 89

17 29 34 | 90 **45** 68 89

17 29 34 45 | 90 **68** 89

17 29 34 45 68 |90 **89**

17 29 34 45 68 89 |90

Thesortingof list 89, 45,68, 90, 29, 34, 17 isillustrated with theselection sort algorithm.

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison$A[j\ ] < A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n)=\sum_{I=0}^{N-2}\sum_{j=i+1}^{N-1}1=\sum_{I=0}^{N-2}[(n-1)-(i+1)+1]=\sum_{I=0}^{N-2}(n-1-i)=\frac{(n-1)n}{2}$$

Thus,selectionsortisa**θ(n^2)**algorithmonallinputs.

Note:thenumberofkeyswaps is only**θ(n),** or,moreepreciselyn– 1.

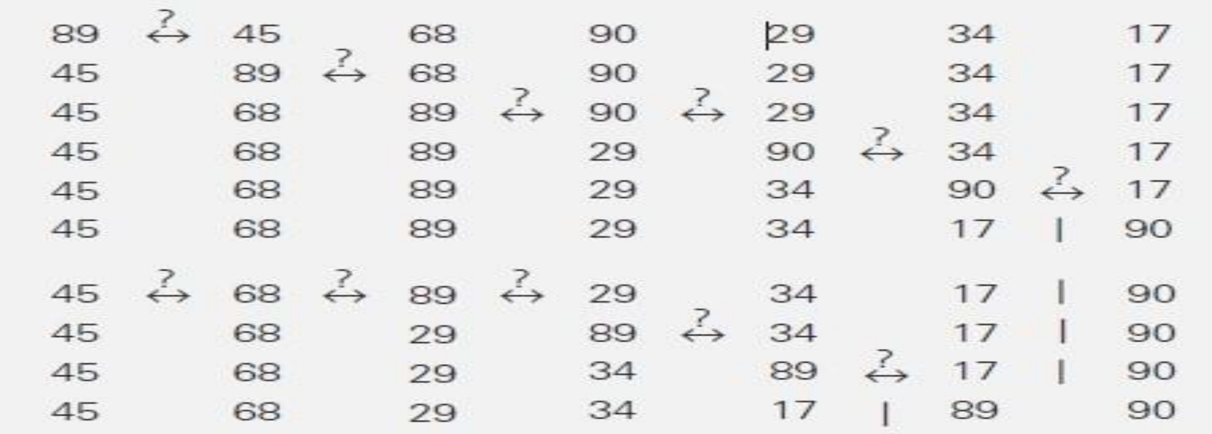
Bubblesort

The bubble sorting algorithm is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until aftern−1passesthelistissorted.passi(0≤i≤n−2)ofbubblesortcanberepresentedbythe

Following:$a_0,\ldots,a_j \overset{?}{\ni\in} a_{j+1},\ldots,a_{n-i-1}|a_{n-i}\le\ldots\le a_{n-1}$

```
Algorithmbubblesort(a[0..n−1])
    //sortsa given arraybybubblesort
    //input:anarraya[0..n−1]oforderableelements
    //output:arraya[0..n−1] sortedinnondecreasingorder
    Fori ← 0 to n −2 do
        Forj ← 0 to n −2 −i do
            Ifa[j+1]<a[j]swapa[j]anda[j+1]
```

Theactionofthe algorithmonthelist 89,45, 68,90, 29,34,17 isillustratedasan example.



89	$\overset{?}{\leftrightarrow}$	45		68		90		|29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17	|	90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17	|	90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17	|	90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17	|	90
45		68		29		34		17	|	89		90

etc.

The number of key comparisons for the bubble-sort version given above is the same for all arraysof size n; it is obtained by a sum that is almost identical to the sum for selection sort:

$$C(n)=\sum_{I=0}^{N-2}\ \sum_{J=i+1}^{N-2-i}1=\sum_{I=0}^{N-2}[(n-2-i)-0+1]=\sum_{I=0}^{N-2}(n-1-i)=\frac{(n-1)n}{2}$$

The number of key swaps, however, depends on the input. In the worst case of decreasingarrays, it is the same as the number of key comparisons.

$C_{\text{worst}}(n)\in\theta(n^2)$

Closest-pairandconvex-hullproblems

Weconsiderastraightforwardapproach(bruteforce)totwowell-known problemsdealing with a finite set of points in the plane. These problems are very useful in important applied areas like computational geometry and operations research.

Closest-pairproblem

Theclosest-pairproblem findsthetwoclosestpointsinasetofnpoints.itisthesimplestof a varietyof problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.

Considerthetwo-dimensional caseof the closest-pairproblem. The points are specified in a standard fashion by their (x, y) cartesian coordinates and that the distance between two points$p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard euclidean distance.

$$d(p_i,p_j)=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$

The following algorithm computes the distance between each pair of distinct points and finds a pair with the smallest distance.

Algorithm*bruteforceclosestpair(p)*

```
//findsdistancebetweentwo closestpoints inthe planebybruteforce
//input:alist pof $n(n\geq 2)$points$p_1(x_1,y_1),\ldots, p_n(x_n,y_n)$
//output:thedistancebetween theclosest pairofpoints
$D\leftarrow\infty$
For $i \leftarrow 1$ to $n-1$ do
    For $j \leftarrow i+1$ to $n$ do
        $D\leftarrow \min(d,sqrt((x_i-x_j)^2+(y_i-y_j)^2))$ //sqrtissquareroot
Return $d$
```

The basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

$$C(n)=\sum_{i=1}^{n-1}. \sum_{J=(i+1)}^{n} 2$$

$$=2\sum_{i=1}^{n-1}(n-i)$$

$$=2[(n-1)+(n-2)+...+1]$$

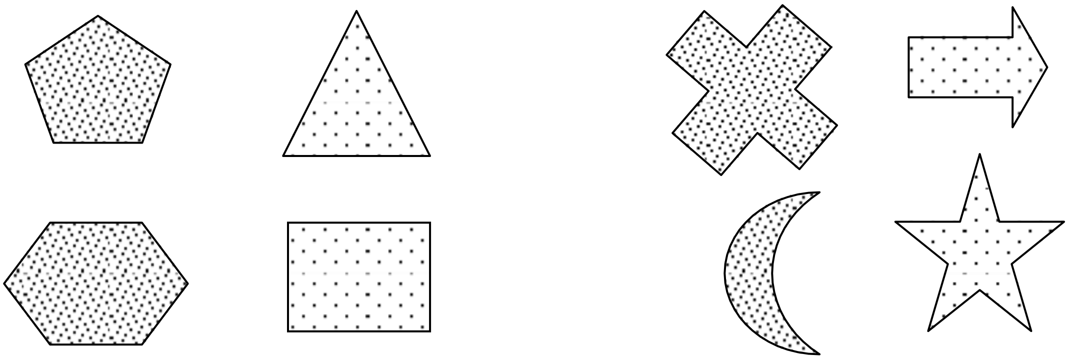
$$=(n-1)n\in\theta(n^2).$$

Ofcourse,speedinguptheinnermostloopofthealgorithmcouldonlydecreasethe Algorithm'srunningtime bya constantfactor, butitcannotimproveitsasymptoticefficiencyclass.

Convex-hullproblem

convex set

Asetofpoints(finiteorinfinite)intheplaneiscalled***convex***ifforanytwopoints*p*and*q* Intheset,the entireline segmentwiththeendpointsat*p*and *q*belongstothe set.



(a) (b)

Figure2.1(a)convexsets.(b)setsthatarenotconvex.

All the sets depicted in figure 2.1 (a) are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon, a circle, and the entire plane.

On the other hand, the sets depicted in figure 2.1 (b), any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.

Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band as shown in figure 2.2

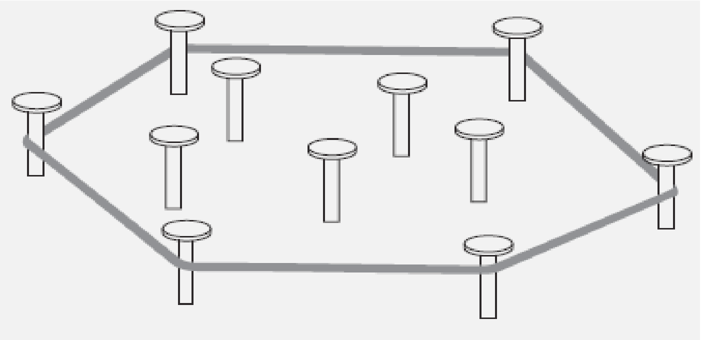


Figure2.2rubber-bandinterpretationoftheconvexhull.

Convexhull

The ***convex hull*** of a set *s* of points is the smallest convex set containing *s*. (the smallest convex hull of *s* must be a subset of any convex set containing *s*.)

If *s* is convex, its convex hull is obviously*s* itself. If *s* is a set of two points, its convex hull is the line segment connecting these points. If *s* is a set of three points not on the same line, its convex hull is thetrianglewith thevertices at thethreepoints given; ifthethreepoints do lieonthe same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see figure 2.3.

Theorem

The convex hull of any set *s* of *n*>2 points not all on the same line is a convex polygon with the vertices at some of the points of *s*. (if all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of *s*.)

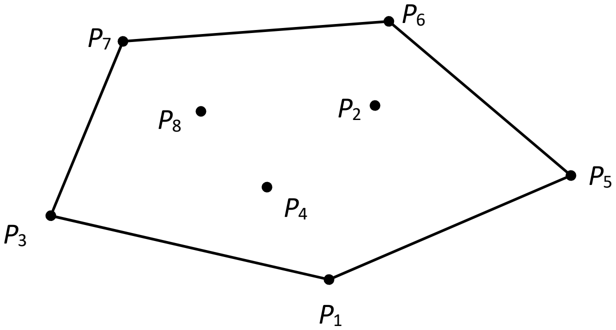


Figure 2.3 the convex hull for this set of eight points is the convex polygon with vertices at p_1, p_5, p_6, p_7, and p_3.

The ***convex-hull problem*** isthe problem of constructingthe convex hull for a given set *s* of *n* points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon “extreme points.” By definition, an ***extreme point*** of a convex set is a point of this set that is *not a middle point of any line segmentwith endpoints in the set*. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in figure 2.3 are p_1, p_5, p_6, p_7, and p_3.

Application

Extremepointshaveseveralspecialpropertiesotherpointsofaconvexsetdonothave.one of them is exploited bythe ***simplexmethod***, thisalgorithm solves ***linear programming* problems.**

We are interested in extreme points because their identification solves the convex-hull problem.actually, tosolvethisproblem completely,weneed toknow abit morethanjustwhich of *n* points of a given set are extreme points of the set’s convex hull. We need to know which pairs of points need to be connected to form the boundary of the convex hull. Note that this issue can alsobe addressed by listing the extreme points in a clockwise or a counterclockwise order.

We can solve the convex-hull problem by brute-force manner. The convex hull problem is onewithnoobvious algorithmicsolution.thereis asimplebutinefficientalgorithmthatisbasedon the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points *pi* and *pj* of a set of *n* points is a part of the convex hull’s boundary ifandonlyif allthe otherpointsofthe setlieon thesamesideof thestraightlinethroughthesetwo points. Repeating this test for every pair of points yields a list of line segments that make up the convex hull’s boundary.

Facts

Afew elementaryfactsfromanalyticalgeometryareneededtoimplementtheabove algorithm.

- First, the straight line through two points (x_1, y_1), (x_2, y_2) in the coordinate plane can be defined by the equation $ax + by = c$,where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$.
- Second, suchalinedivides theplaneintotwo half-planes: forall thepointsin oneofthem, $ax+ by > c$, while for all the points in the other, $ax + by < c$. (for the points on the line itself, of course, $ax + by = c$.) Thus, to check whether certain points lie on the same side of the line, we can simplycheck whether the expression $ax + by - c$ has the same sign for each of these points.

Timeefficiencyofthis algorithm.

Time efficiency of this algorithmis in $o(n^3)$: for each of $n(n - 1)/2$ pairs of distinct points, we may need to find the sign of $ax + by - c$ for each of the other $n - 2$ points.

Exhaustivesearch

Fordiscreteproblems in which no efficient solution method is known,it might benecessary totesteachpossibilitysequentiallyinordertodetermineifitisthesolution. Such***exhaustive***examination of all possibilities is known as ***exhaustivesearch, complete searchor* direct *search.***

Exhaustive search is simply a brute force approach to combinatorial problems (minimization or maximization of optimization problems and constraint satisfaction problems).

Reason to choose brute-force / ***exhaustive search*** approach as an important algorithmdesign strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide varietyofproblems. In fact,itseems tobetheonly**general approach**forwhichitis more difficult to point out problems it *cannot* tackle.
2. Second,forsomeimportant problems,e.g.,sorting,searching,matrix multiplication, stringmatching the brute-force approach yields reasonable algorithms of at least some practical value **with no limitation on instance size**.
3. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with **acceptable speed**.
4. Fourth, even if too **inefficient** in general, a brute-force algorithm can still be **useful for solving small-size instances** of a problem.

Exhaustivesearchisappliedtotheimportantproblemslike

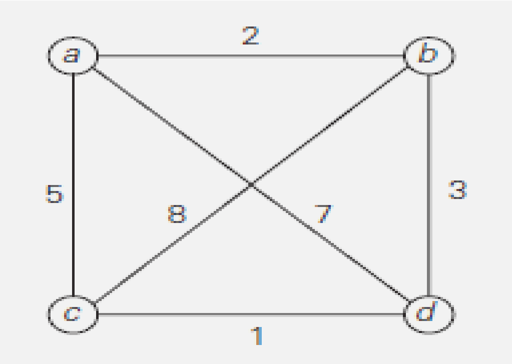
- Travelingsalesman problem
- Knapsackproblem
- Assignmentproblem.

Travelingsalesmanproblem

The ***traveling salesman problem (tsp)*** is one of the combinatorial problems. The problem asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started.

The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***hamiltonian circuit*** of the graph. (a hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once).

Ahamiltoniancircuitcanalsobedefinedasasequenceof$n$+1adjacentvertices vi_0, vi_1, . . . , vi_{n-1}, vi_0, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



Tour	length
A--->b--->c --->d--->a	i=2 +8+1 +7 =18
A--->b--->d--->c--->a	**i= 2 + 3 + 1 + 5 = 11 optimal**
A--->c--->b--->d--->a	i=5 +8+3 +7 =23
A--->c--->d--->b--->a	**i= 5 + 1 + 3 + 2 = 11 optimal**
A--->d--->b--->c--->a	i=7 +3+8 +5 =23
A--->d--->c --->b--->a	i=7 +1+8 +2 =18

Figure2.4 solutionto asmall instanceof thetravelingsalesmanproblem byexhaustive search.

Time efficiency

- Wecangetallthetoursbygeneratingallthepermutationsofn−1intermediatecities Fromaparticularcity..i.e.**(n-1)!**
- Considertwointermediatevertices,say,*b*and*c*,andthenonlypermutationsinwhich*b* Precedes*c*.(thistrickimplicitlydefinesatour's direction.)
- An inspection of figure 2.4reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same.
- The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of *n*.

Knapsackproblem

Given *n* items ofknownweights $w_1, w_2, \ldots, w_n$and values $v_1, v_2, \ldots, v_n$and aknapsack of capacity *w*, find the most valuable subset of the items that fit into the knapsack.

Realtime examples:

- Athiefwho wantstostealthemostvaluableloot thatfitsinto hisknapsack,
- Atransportplanethathastodeliverthemostvaluablesetofitemstoaremotelocation Withoutexceedingtheplane'scapacity.

The exhaustive-search approach to this problem leads to generatingall the subsets of the set of*n*items given,computingthetotalweightofeachsubsetinordertoidentifyfeasiblesubsets(i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

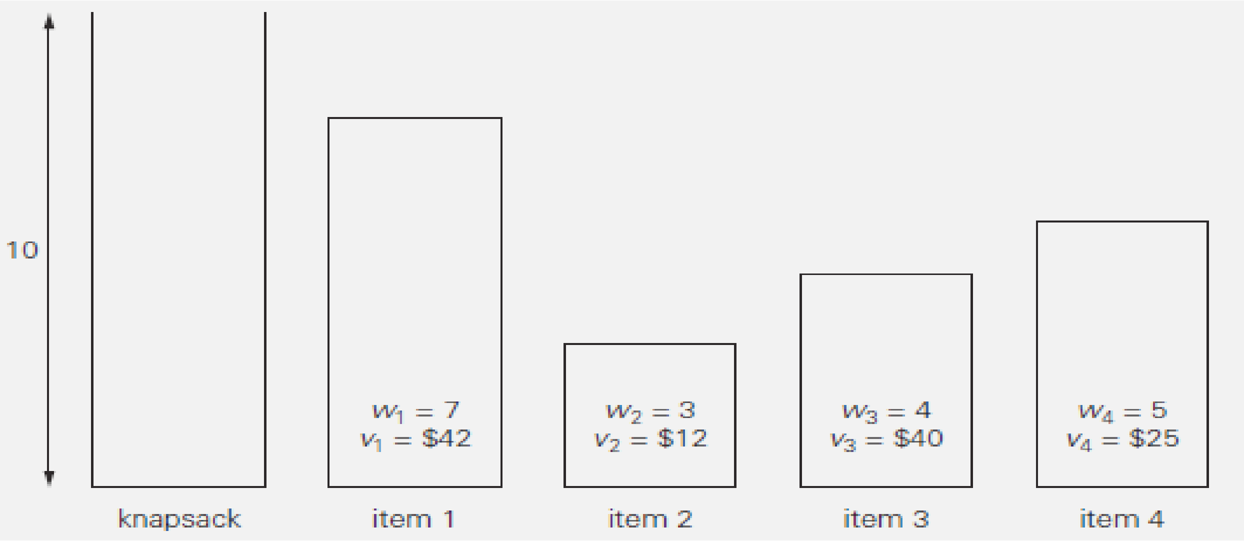


Figure2.5instanceoftheknapsackproblem.

Subset	Totalweight	Total value
Φ	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	Notfeasible
{1, 4}	12	Notfeasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65(maximum-optimum)**
{1, 2, 3}	14	Notfeasible
{1, 2, 4}	15	Notfeasible
{1, 3, 4}	16	Notfeasible
{ 2, 3, 4}	12	Notfeasible
{1, 2, 3, 4}	19	Notfeasible

Figure2.6knapsackproblem'ssolution byexhaustivesearch.the information abouttheoptimal Selectionisinbold.

Time efficiency: as given in the example, the solution to the instance of figure 2.5 is given in figure 2.6. Since the *number of subsets of an n-element set is* ***2^n***, the exhaustive search leads to a ***$\Omega(2^n)$*** algorithm, no matter how efficiently individual subsets are generated.

Note:exhaustive search of both the traveling salesman and knapsack problems leads to extremely inefficient algorithms on every input. In fact, these two problems are the best-known examples of ***np-hard problems***. **No polynomial-time** algorithm is known for any *np*-hard problem. Moreover, most computer scientists believe that such algorithms do not exist. Some sophisticated approaches like **backtracking** and **branch-and-bound** enable us to solve some instances but not all instances of these in less than exponential time. Alternatively, we can use one of many **approximation algorithms.**

Assignmentproblem.

There are n people who need to be assigned to execute n jobs, one person per job. (that is, each person is assigned to exactlyone job and each job is assigned to exactlyone person.) The cost thatwouldaccrueiftheithpersonisassignedtothejthjobisaknownquantity$C[i,$j]foreachpair i,j=1,2,...,n.theproblemistofindanassignmentwiththeminimumtotalcost.

Assignment problem solved byexhaustive search is illustrated with an example as shown in figure 2.8. A small instance of this problem follows, with the table entries representing the assignment costs $c[i, j]$.

	Job1	Job2	Job3	Job4
Person1	9	2	7	8
Person2	6	4	3	7
Person3	5	8	1	8
Person4	7	6	9	4

Figure2.7instanceofanassignmentproblem.

Aninstanceof theassignment problemis completelyspecifiedbyitscostmatrix*c*.

$$C=\begin{bmatrix}9 & 2 & 7 & 8\\ 6 & 4 & 3 & 7\\ 5 & 8 & 1 & 8\\ 7 & 6 & 9 & 4\end{bmatrix}$$

The problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

We can describe feasible solutions to the assignment problem as *n*-tuples <j_1, . . . , j_n> in which the *i*th component, i=1,...,n, indicates the column of the element selected in the *i*th row (i.e.,thejobnumberassignedtothe*i*thperson).forexample,forthecost matrixabove,<2,3,4,1> indicatestheassignmentofperson1tojob2,person2tojob3,person3tojob4,andperson4to Job1. Similarlywecanhave4!=4 ·3·2·1=24,*i.e.*,24permutations.

The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first *n* integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutationsofintegers 1,2,...,n, computingthetotal cost ofeach assignment bysummingup the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum. A few first iterations of applying this algorithm to the instance given above are given below.

<1,2, 3, 4>	Cost= 9 +4 +1+ 4 =**18**	**<2, 1, 3, 4>**	**Cost= 2+ 6 +1 + 4= 13(min)**
<1,2, 4, 3>	Cost= 9 +4 +8+9 =30	<2,1, 4, 3>	Cost= 2 +6 +8+9 =25
<1,3, 2, 4>	Cost= 9 +3 +8+4 =24	<2,3, 1, 4>	Cost= 2 +3 +5+4 =14
<1,3, 4, 2>	Cost= 9 +3 +8+6 =26	<2,3, 4, 1>	Cost= 2 +3 +8+7 =20
<1,4, 2, 3>	Cost= 9 +7 +8+9 =33	<2,4, 1, 3>	Cost= 2 +7 +5+9 =23
<1,4, 3, 2>	Cost= 9 +7 +1+6 =23	<2,4, 3, 1>	Cost= 2 +7 +1+7 =17,etc

Figure 2.8 first few iterations of solving a small instance of the assignment problem by exhaustive search.

Since the number of permutations to be considered for the general case of the assignment problem is ***n*!,**exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the ***hungarian method.***

Divideandconquermethodology

A**divide and conquer algorithm**works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (**divide**), until these become simple enough to be solved directly (**conquer**).

Divide-and-conqueralgorithmsworkaccordingtothefollowinggeneralplan:

1. Aproblemisdividedintoseveralsubproblems ofthesametype,ideallyof about equalsize.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique as shown in figure 2.9, which depicts the case of dividing a problem into two smaller subproblems, then the subproblems solved separately. Finally solution to the original problem is done by combining the solutions of subproblems.

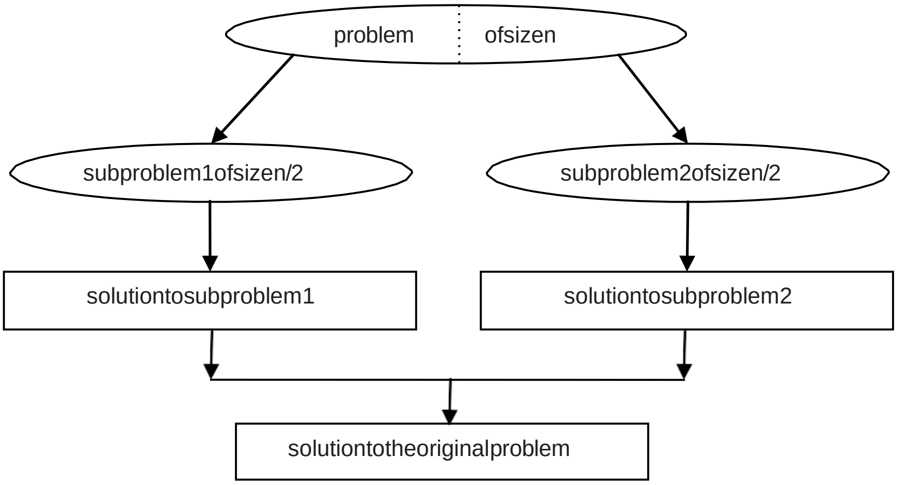


Figure2.9divide-and-conquertechnique.

Divideand conquer methodologycan beeasilyapplied on the following problem.

1. Merge sort
2. Quicksort
3. Binarysearch

Merge sort

Mergesort is based on divide-and-conquer technique. It sorts a given array $a[0..n-1]$ by dividing it into two halves $a[0..\lfloor n/2]-1]$ and $a[\lfloor n/2]..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

```
Algorithmmergesort($a[0..n-1]$)
    //sortsarray$a[0..n-1]$byrecursive mergesort
    //input:anarray$a[0..n-1]$oforderable elements
    //output:array$a[0..n-1]$sortedinnondecreasingorder
    If$n>1$
        Copy$a[0..\lfloor n/2]-1]$to$b[0..\lfloor n/2]-1]$
        copy$a[\lfloor n/2]..n-1]$to$c[0..]n/2]-1]$
        mergesort($b[0..\lfloor n/2]-1]$)
        mergesort($c[0..]n/2]-1]$)
        Merge(b,c,a)//see below
```

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the indexof the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

```
Algorithmmerge($b[0..p-1],c[0..q-1],a[0..p+q-1]$)
    //mergestwosortedarraysintoonesortedarray
    //input:arrays$b[0..p-1]$and$c[0..q-1]$both sorted
    //output:sortedarray$a[0..p+q-1]$oftheelementsof$b$and$c$ $i \leftarrow 0$; $j$
    $\leftarrow 0$; $k \leftarrow 0$
    While$i<p$and$j<q$do
        If$b[i] \le c[j]$
            $A[k] \leftarrow b[i]; i \leftarrow i+1$ else
        $a[k] \leftarrow c[j]; j \leftarrow j+1$ $k \leftarrow k+1$
    If $i=p$
        Copy$c[j..q-1]$to $a[k..p+q-1]$
    Elsecopy$b[i..p-1]$to $a[k..p+q-1]$
```

Theoperation ofthe algorithmon the list8, 3,2, 9, 7,1, 5,4 is illustratedinfigure2.10.

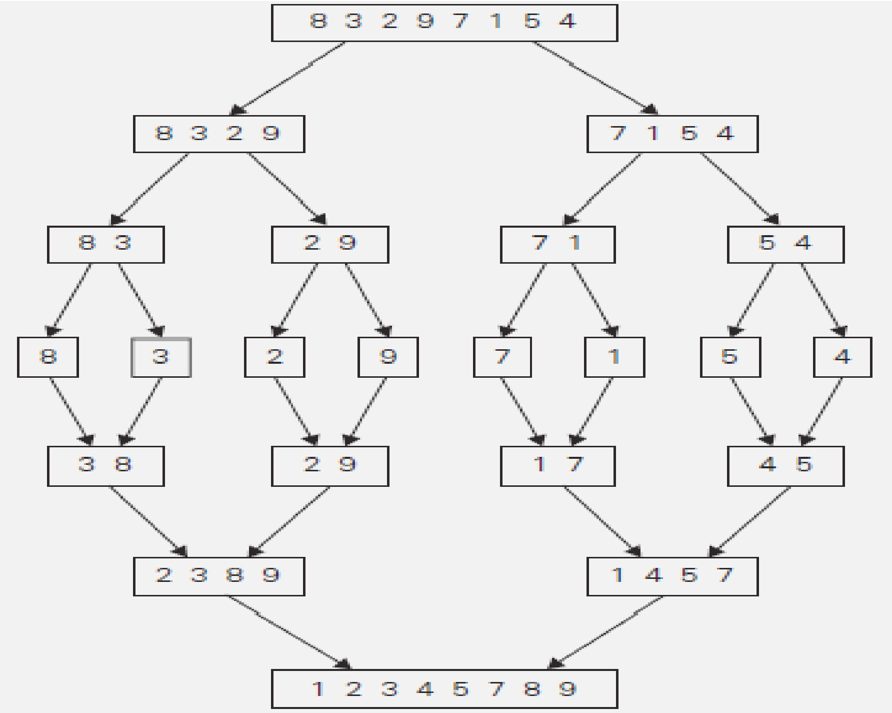


Figure2.10exampleofmergesortoperation.

Therecurrencerelationforthenumberofkeycomparisons$c(n)$is

$C(n)=2c(n/2)+c_{merge}(n)forn>1$, $c(1)=0$.

Intheworstcase,$c_{merge}(n)=n-1$, andwehavetherecurrence

$Cworst(n)=2c_{worst}(n/2)+n-1forn>1, c_{worst}(1)=0$.

Bymastertheorem,$c_{worst}(n)\in\theta(n\log n)$

Theexactsolutiontotheworst-caserecurrencefor$n=2^k$

$C_{worst}(n)=n\log_2 n-n+1$.

Forlarge*n*, thenumberofcomparisonsmadebythis algorithm in theaveragecaseturns out to be about 0.25*n* less and hence is also in**θ *(n* log *n).***

First,thealgorithmcanbeimplementedbottomupbymergingpairsofthearray’selements, then merging the sorted pairs, and so on. This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called ***multiway mergesort***.

Quicksort

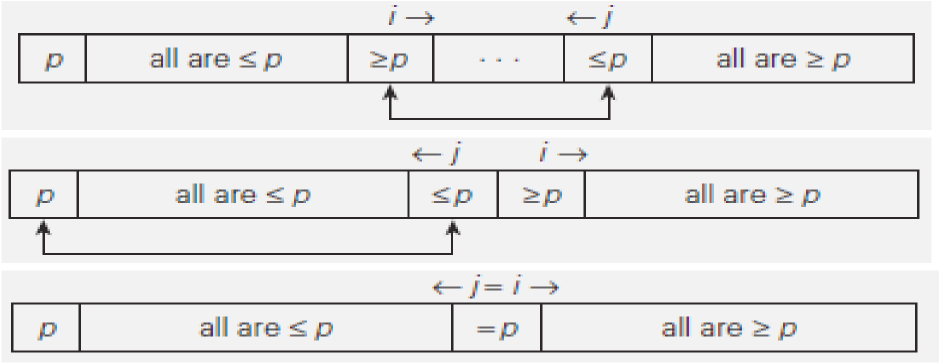
Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Quicksort divides input elements according to their value. A partition is an arrangementof the array’s elements so that all the elements to the left of some element $a[s]$ are less than orequal to $a[s]$, and all the elements to the right of $a[s]$ are greater than or equal to it:

$$\underset{\text{Allare}\le a[s]}{A[0]\ldots a[s-1]} \qquad a[s] \qquad \underset{\text{allare}\ge a[s]}{a[s+1]\ldots a[n-1]}$$

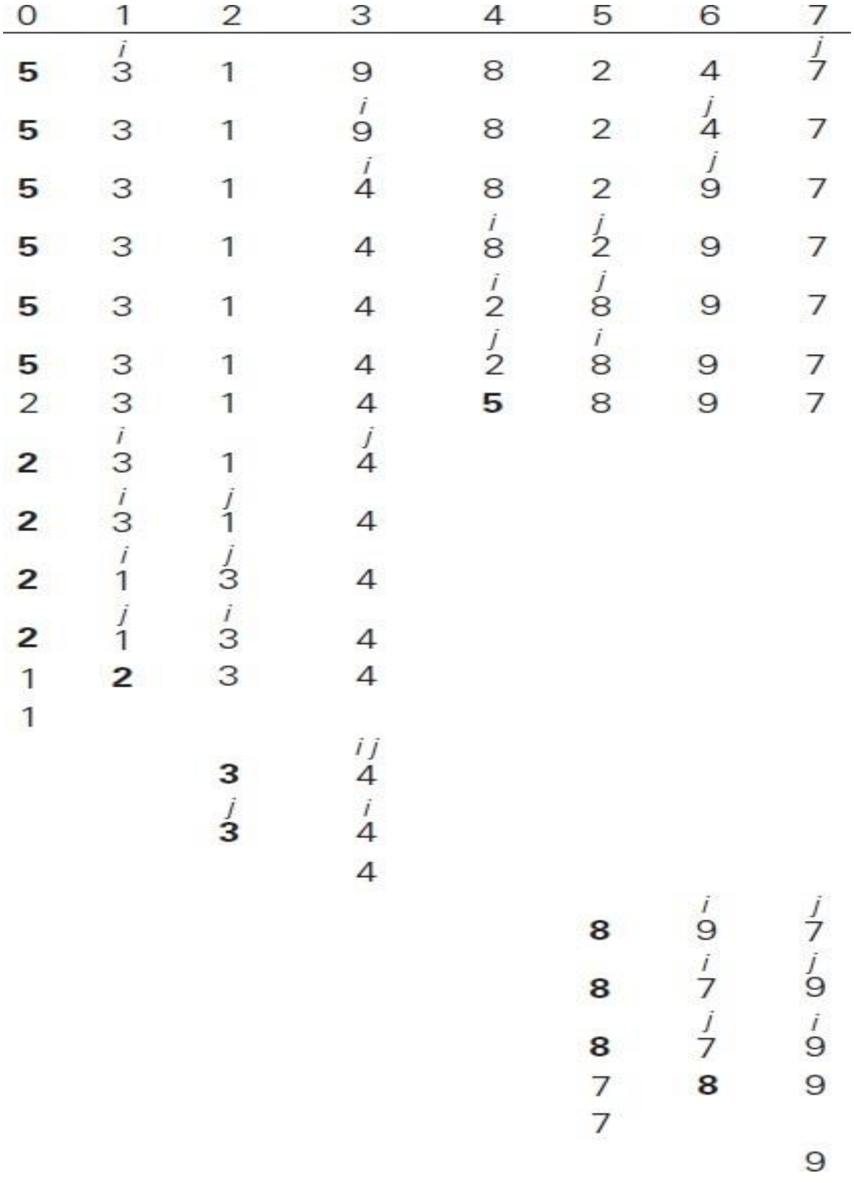
Sortthetwosubarraystotheleftandtotherightof$a[s]$independently.noworkrequiredto combine the solutions to the subproblems.

Hereispseudocodeofquicksort:call*quicksort(a*[0..*n*−1]*)*whereasapartitionalgorithmusethe *Hoarepartition*

```
Algorithmquicksort(a[l..r])
    //sortsa subarraybyquicksort
    //input:subarrayofarraya[0..n−1],definedbyitsleftandrightindices landr
    //output:subarraya[l..r]sortedinnondecreasingorder
    If l <r
        S←hoarepartition(a[l..r])//sisasplit position
        Quicksort(a[l..s−1])
        quicksort(a[s+1..r])
```



```
**Algorithm***hoarepartition(a*[*l..r*]*)*
        //partitionsasubarraybyhoare'salgorithm,usingthefirstelementasapivot
        //input:subarrayofarray*a*[0..*n*−1],definedbyitsleftandrightindices *l*and*r(l<r)*
        //output:partitionof*a*[*l..r*],withthesplitpositionreturned asthisfunction's value
        $P \leftarrow a[l]$
        $I \leftarrow l;\ j \leftarrow r + 1$
        **Repeat**
                **Repeat** $i \leftarrow i + 1$ **until** $a[i] \geq p$
                **repeat**$j \leftarrow j - 1$**until**$a[j] \leq p$
                swap$(a[i], a[j\,])$
        **Until**$i \geq j$
        Swap$(a[i],a[j])$//undolastswapwhen$i \geq j$
        Swap$(a[l],a[j])$
        **Return***j*
```



0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{ij}{4}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9

Figure2.11exampleofquicksortoperationofarraywithpivots shownin bold.

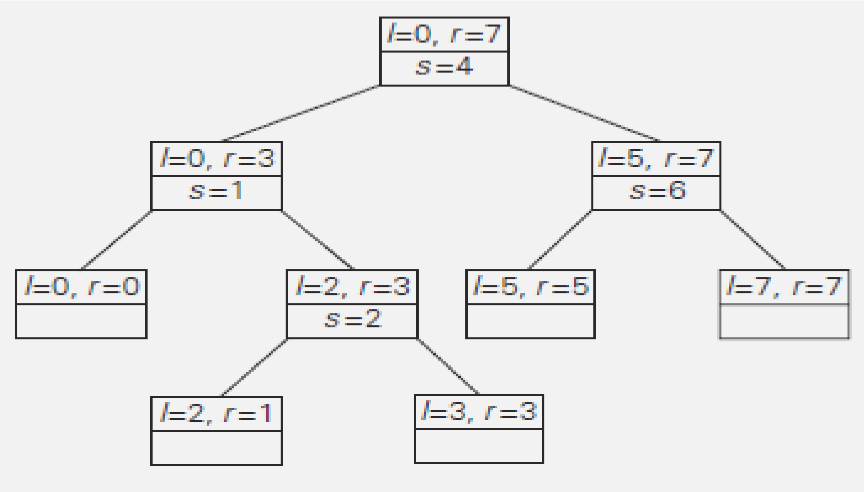


Figure 2.12treeofrecursivecallsto *quicksort* withinputvalues *l*and*r*ofsubarrayboundsand split position *s* of a partition obtained.

Thenumberofkeycomparisonsinthebestcasesatisfies therecurrence

$C_{\text{best}}(\text{n})=2c_{\text{best}}(\text{n}/2)+\text{n}$ forn>1, $c_{\text{best}}(1)=0$.

Bymaster theorem,$c_{\text{best}}(\text{n})\in\theta(\text{nlog}_2\ \text{n})$;solvingitexactlyforn=$2^k$ yields$c_{\text{best}}(\text{n})=\text{nlog}_2\ \text{n}$. The total number of key comparisons made will be equal to

Cworst(n)= *(n*+1)+*n*+ . ..+3=((*n*+1)*(n*+2))/2−3∈θ*(n*^{2}*).*

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

$$C_{avg}(n) \approx 2n \ln n \approx 1.39 n \log_2 n.$$

Binarysearch

A binary search is efficient algorithm to find the position of a target (key) value within a sorted array.

- The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished.
- If the target value is less than the middle element's value, then the search continues on the lower half of the array.
- If the target value is greater than the middle element's value, then the search continueson the upper half of the array.
- This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (position is returned).

Binary search is a remarkably efficient algorithm for searching in a sorted array (say a). It works by comparing a search key k with the array’s middle element a[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the arrayif k <a[m], and for the second half if k >a[m]:

$$\underbrace{A[0]\ldots A[\mathrm{m}-1]}_{\text{Searchhereifk}\le A[\mathrm{m}]} \qquad \overset{\overset{\mathrm{K}}{\Updownarrow}}{A[\mathrm{m}]} \qquad \underbrace{A[\mathrm{m}+1]\ldots A[n-1]}_{\text{Searchhereifk}\ge A[\mathrm{m}]}$$

Though binarysearch is clearlybased on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is pseudocode of this nonrecursive version.

```
Algorithmbinarysearch(a[0..n−1],k)
    //implementsnonrecursivebinarysearch
    //input:anarraya[0..n−1]sortedinascendingorderandasearchkeyk
    //output:anindexofthearray'selementthatisequaltok/or−1ifthereisnosuch element
    L← 0; r ← n −1
    Whilel≤rdo
        M←⌊(l+r)/2]
        Ifk=a[m]returnm
        Else ifk <a[m]
            R← m−1
        Elsel← m+1
    Return−1
```

The standard way to analyze the efficiency of binary search is to count the number of times the search keyis compared with an element of the array (three-way comparisons). One comparison of k with a[m], the algorithm can determine whether k is smaller, equal to, or larger than a[m].

Asanexample,letusapplybinarysearchtosearchingfor *k*=70inthearray. Theiterations of the algorithm are given in the following table:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value	3	14	27	31	39	42	55	70	74	81	85	93	98
Iteration1	*L*						*M*						*R*
Iteration2								*L*		*M*			*r*
Iteration3								*L,* *m*	*R*				

The worst-case inputs include all arrays that do not contain a given search key, as well as somesuccessfulsearches. Sinceafterone comparison thealgorithm faces thesamesituation but for an array half the size,

Thenumberofkeycomparisonsintheworstcase$c_{worst}(n)$byrecurrence relation.

$$C_{worst}(n)=C_{worst}(\lfloor \tfrac{n}{2} \rfloor)+1 for n>1, C_{worst}(1)=1.$$

$\therefore C_{worst}(n)=\lfloor log_2 n\rfloor+1=\lceil log_2(n+1)\rceil$ $\qquad$ $\therefore C_{worst}(2^k)=(k+1)=log_2 k+1 \text{forn}=2^k$

- First,theworst-casetime efficiencyofbinarysearch isin θ(logn).
- Second, the algorithm simply reduces the size of the remaining array by half on each iteration, the number of such iterations needed to reduce the initial size *n* to the final size 1 has to be about $\log_2 n$.

- Third, the logarithmic function grows so slowly that its values remain small even for verylarge values of *n*.

Theaverage caseslightlysmallerthanthatintheworstcase

$C_{avg}(n) \approx \log_2 n$

Theaveragenumberofcomparisonsinasuccessfulis

$\boldsymbol{C_{avg}(n) \approx \log_2 n - 1}$

Theaveragenumberofcomparisonsinanunsuccessfulis

$\boldsymbol{C_{avg}(n) \approx \log_2 (n+1)}$.

Multiplicationoflargeintegers

Some applications like modern cryptography require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment.

In the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications.

The divide-and-conquer method does the above multiplication in less than n^2 digit multiplications.

Example: $23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$

$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0$

$= 2 \cdot 10^2 + 11 \cdot 10^1 + 12 \cdot 10^0$

$= 3 \cdot 10^2 + 2 \cdot 10^1 + 2 \cdot 10^0$

$= 322$

Theterm$(2 * 1 + 3 * 4)$ computedas$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) – (2 * 1) − (3 * 4)$.here $(2 * 1)$ and$(3 * 4)$ are already computed used. So only one multiplication only we have to do.

Foranypair of two-digit numbers $a = a_1a_0$ and $b= b_1b_0$, their product c can be computed by the formula $c = a * b = c_2 10^2 + c_1 10^1 + c_0$,

Where

$C_2 = a_1 * b_1$is theproduct of theirfirst digits,

$C_0 = a_0 * b_0$ is theproductof theirsecond digits,

$C_1 = (a_1 + a_0) * (b_1 + b_0) − (c_2 + c_0)$isthe productofthesum ofthe

A’sdigitsandthe sumof the b’sdigitsminusthesumof c_2 andc_0.

Now we apply this trick to multiplying two *n*-digit integers *a* and *b* where *n* is a positive even number. Let us divide both numbers in the middle to take advantage of the divide-and-conquer technique. We denote the first half of the *a*’s digits by a_1 and the second half by a_0; for *b*, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$and $b = b_1b_0$ impliesthat $b = b_1 10^{n/2} + b_0$.therefore, takingadvantage of the same trick weused for two-digit numbers, we get

$$C = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$

$$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$

$$= c_2 10^n + c_1 10^{n/2} + c_0,$$

Where

$C_2 = a_1 * b_1$ is theproductof theirfirst halves,

$C_0 = a_0 * b_0$ isthe productof theirsecond halves,

$C_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

If$n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1.thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

The multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $m(n)$ is $m(n) = 3m(n/2)$ for $n > 1$, $m(1) = 1$.

Solvingitbybackwardsubstitutionsfor$n=2^k$yields

$$
\begin{aligned}
M(2^k) &= 3m(2^{k-1}) \\
&= 3[3m(2^{k-2})] \\
&= 3^2 m(2^{k-2}) \\
&= \ldots \\
&= 3^i m(2^{k-i}) \\
&= \ldots \\
&= 3^k m(2^{k-k}) \\
&= 3^k.
\end{aligned}
$$

(since$k=\log_2 n$)

$$\boldsymbol{M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585.}}$$

(onthelast step,wetookadvantageof thefollowingpropertyoflogarithms:$a^{\log_b c} = c^{\log_b a}$.)

Let $a(n)$ be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides $3a(n/2)$of these operations needed to compute the three products of $n/2$-digit numbers, the above formulas require five additions and one subtraction. Hence, we have the recurrence

$A(n) = 3 \cdot a(n/2) + cnforn > 1$,$a(1) = 1$.

Byusingmastertheorem, weobtain$\boldsymbol{a(n) \in \theta(n^{\log_2 3}),}$

Which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

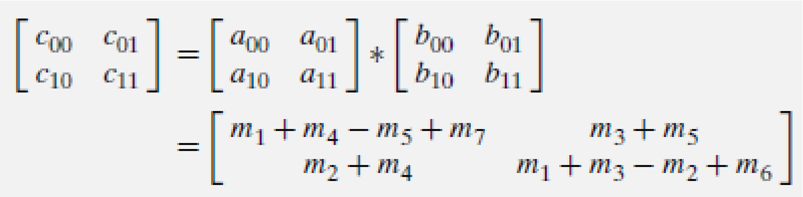
Example:forinstance:a=2345,b=6137,i.e.,n=4. Then$c$

$= a * b = (23*10^2+45)*(61*10^2+37)$

$$
\begin{aligned}
C = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
&= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\
&= (23 * 61)10^4 + (23 * 37 + 45 * 61)10^2 + (45 * 37) \\
&= 1403 \bullet 10^4 + 3596 \bullet 10^2 + 1665 \\
&= 14391265
\end{aligned}
$$

Strassen’smatrixmultiplication

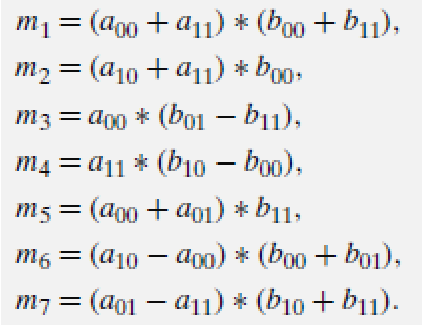
The strassen’s matrix multiplication find the product *c* of two 2 × 2 matrices *a* and *b* with just seven multiplications as opposed to the eight required by the brute-force algorithm.



$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

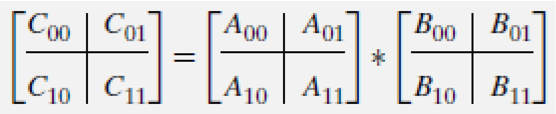
Where



$$\begin{aligned} m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}), \\ m_2 &= (a_{10} + a_{11}) * b_{00}, \\ m_3 &= a_{00} * (b_{01} - b_{11}), \\ m_4 &= a_{11} * (b_{10} - b_{00}), \\ m_5 &= (a_{00} + a_{01}) * b_{11}, \\ m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}), \\ m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}). \end{aligned}$$

Thus, to multiply two 2 × 2 matrices, strassen’s algorithm makes 7 multiplications and 18 additions/subtractions,whereasthebrute-forcealgorithmrequires8multiplicationsand4additions. These numbers should not lead us to multiplying 2 × 2 matrices by strassen’s algorithm. Its importance stems from its *asymptotic* superiority as matrix order *n* goes to infinity.

Let *a* and *b* be two *n* × *n* matrices where *n* is a power of 2. (if *n* is not a power of 2,matrices can be padded with rows and columns of zeros.) We can divide *a*, *b*, and their product *c* into four *n/*2 × *n/*2 submatrices each as follows:



$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

The value c_{00} can be computed either as $a_{00} * b_{00} + a_{01} * b_{10}$ or as $m_1 + m_4 - m_5 + m_7$ where m_1, m_4, m_5, and m_7 are found by strassen’s formulas, with the numbers replaced by the corresponding submatrices. The seven products of *n/*2 × *n/*2 matrices are computed recursively by strassen’s matrix multiplication algorithm.

Theasymptoticefficiencyofstrassen’smatrixmultiplicationalgorithm

If m(n) is the number of multiplications made by strassen’s algorithm in multiplying two n×n matrices, where n is a power of 2, the recurrence relation is m(n) = 7m(n/2) for n > 1,m(1)=1.

Since$n=2^k$,

$$\begin{aligned} M(2^k) &= 7m(2^{k-1}) \\ &= 7[7m(2^{k-2})] \\ &= 7^2 m(2^{k-2}) \\ &= \ldots \end{aligned}$$

$=7^i m(2^{k-i})$

$=. . .$

$=7^k m(2^{k-k})=7^k m(2^0)=7^k m(1)=7^k(1)$ (sincem(1)=1)

$M(2^k)=7^k.$

Since$k$=log$_2 n$,

$$M(n)=7^{\log_2 n}$$
$$=n^{\log_2 7}$$
$$\approx \boldsymbol{n^{2.807}}$$

Whichis smallerthan n^3 required bythe brute-forcealgorithm.

Since this savings in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions $a(n)$ made by strassen’s algorithm. To multiply two matrices of order $n>1$, the algorithm needs to multiply seven matrices of order$n/2$and make18 additions/subtractions ofmatrices of size $n/2$; when $n=1$, noadditions aremadesince two numbers are simply multiplied. These observations yield the following recurrence relation:

$$A(n)=7a(n/2)+18(n/2)^2 \text{for} n>1,\ a(1)=0.$$

Byclosed-formsolutiontothisrecurrenceandthemastertheorem,$a(n)\in\theta(n^{\log_2 7})$.whichisa Betterefficiencyclassthan$\theta(n^3)$ofthebrute-forcemethod.

Example:multiplythe followingtwomatricesbystrassen’smatrix multiplicationalgorithm.

$$\textbf{A}=\begin{bmatrix}1&0&2&1\\4&1&1&0\\0&1&3&0\\5&0&2&1\end{bmatrix} \qquad \textbf{b}=\begin{bmatrix}0&1&0&1\\2&1&0&4\\2&0&1&1\\1&3&5&0\end{bmatrix}$$

Answer:

$$\textbf{C}=\begin{bmatrix}C_{00}&C_{01}\\C_{10}&C_{11}\end{bmatrix}=\begin{bmatrix}A_{00}&A_{01}\\A_{10}&A_{11}\end{bmatrix}\textbf{x}\begin{bmatrix}B_{00}&B_{01}\\B_{10}&B_{11}\end{bmatrix}$$

Where$\mathbf{a_{00}}=\begin{bmatrix}1&0\\4&1\end{bmatrix}$ $\mathbf{a_{01}}=\begin{bmatrix}2&1\\1&0\end{bmatrix}$ $\mathbf{a_{10}}=\begin{bmatrix}0&1\\5&0\end{bmatrix}$ $\mathbf{a_{11}}=\begin{bmatrix}3&0\\2&1\end{bmatrix}$

$\mathbf{B_{00}}=\begin{bmatrix}0&1\\2&1\end{bmatrix}$ $\mathbf{b_{01}}=\begin{bmatrix}0&1\\0&4\end{bmatrix}$ $\mathbf{b_{10}}=\begin{bmatrix}2&0\\1&3\end{bmatrix}$ $\mathbf{b_{11}}=\begin{bmatrix}1&1\\5&0\end{bmatrix}$

$$M_1=(a_{00}+a_{11})*(b_{00}+b_{11})=(\begin{bmatrix}1&0\\4&1\end{bmatrix}+\begin{bmatrix}3&0\\2&1\end{bmatrix})*(\begin{bmatrix}0&1\\2&1\end{bmatrix}+\begin{bmatrix}1&1\\5&0\end{bmatrix})=\begin{bmatrix}4&0\\6&2\end{bmatrix}*\begin{bmatrix}1&2\\7&1\end{bmatrix}=\begin{bmatrix}4&8\\20&14\end{bmatrix}$$

Similarlyapplystrassen’smatrixmultiplicationalgorithmto findthe following.

$M_2=\begin{bmatrix}2&4\\2&8\end{bmatrix}$,$m_3=\begin{bmatrix}-1&0\\-9&4\end{bmatrix}$,$m_4=\begin{bmatrix}6&-3\\3&0\end{bmatrix}$,$m_5=\begin{bmatrix}8&3\\10&5\end{bmatrix}$, $m_6=\begin{bmatrix}2&-3\\-2&-3\end{bmatrix}$,$m_7=\begin{bmatrix}3&2\\-9&-4\end{bmatrix}$

$C_{00}=\begin{bmatrix}5&4\\4&5\end{bmatrix}$,$c_{01}=\begin{bmatrix}-7&3\\1&9\end{bmatrix}$,$c_{10}=\begin{bmatrix}8&1\\5&8\end{bmatrix}$,$c_{11}=\begin{bmatrix}3&7\\7&7\end{bmatrix}$

$$\textbf{C}=\begin{bmatrix}C_{00}&C_{01}\\C_{10}&C_{11}\end{bmatrix}=\begin{bmatrix}5&4&7&3\\4&5&1&9\\8&1&3&7\\5&8&7&7\end{bmatrix}$$

Closest-pairandconvex-hullproblems.

The two-dimensional versions of the closest-pair problem and the convex-hull problem problems can be solved by brute-force algorithms in $\theta(n^2)$and $o(n^3)$time, respectively.thedivide-and-conquer technique provides sophisticated and asymptotically more efficient algorithms to solve these problems.

Theclosest-pair problem

Let *p* be a set of *n* >1 points in the cartesian plane. The points are ordered innondecreasing order of their *x* coordinate. It will also be convenient to have the points sorted (by merge sort) in a separate list in nondecreasingorder ofthe *y* coordinate and denote such a list by*q.*

If $2 \leq n \leq 3$, the problem can be solved by the obvious brute-force algorithm. If *n* >3, we candividethepointsinto twosubsets *pl*and*pr*of]*n*/2] andL*n*/2] points,respectively,bydrawing a vertical line through the median *m* of their *x* coordinates so that]*n*/2]points lie to the left of oron the line itself, and]*n*/2]points lie to the right of or on the line. Then we can solve the closest-pairproblem recursivelyforsubsets $p_l$and p_r . Let $d_l$and $d_r$bethesmallest distances between pairs of points in *pl* and *pr*, respectively, and let $d = \min\{d_l, d_r\}$.

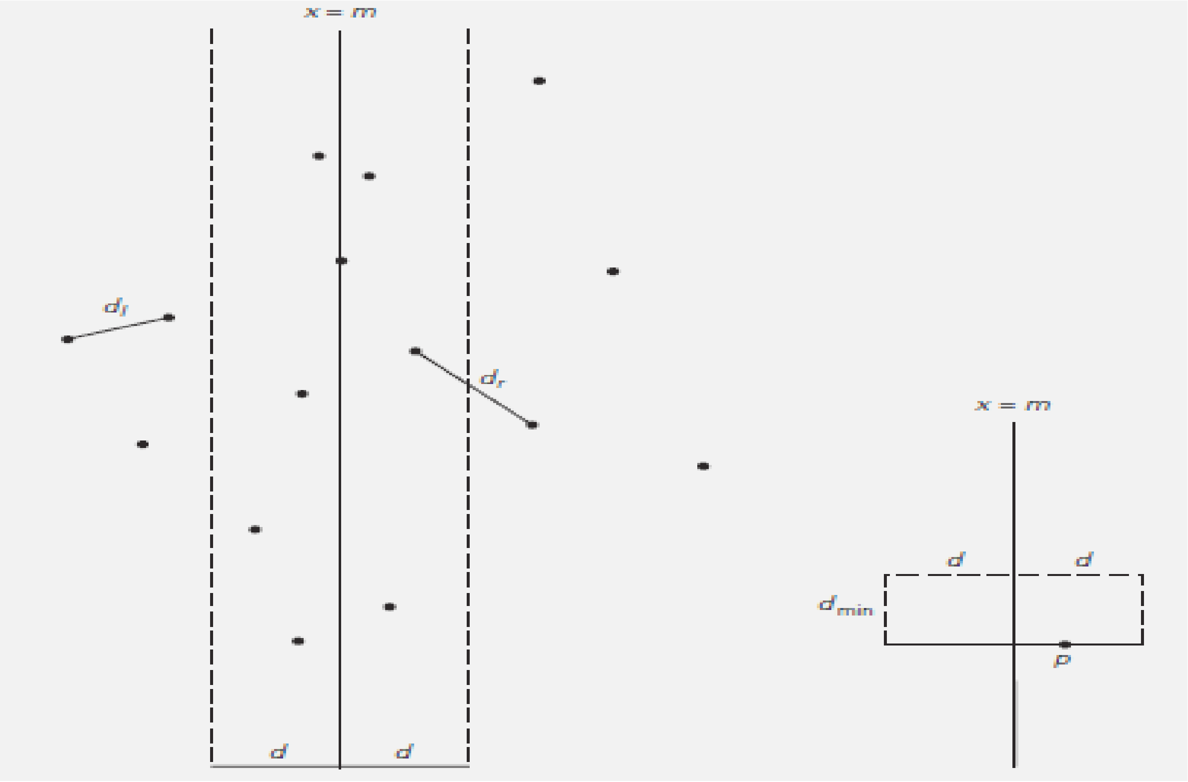


Figure2.13(a)ideaofthedivide-and-conqueralgorithmfortheclosest-pair problem.
(b)rectangle thatmaycontainpoints closerthand_{min} topoint *p*.

Note that *d* is not necessarily the smallest distance between all the point pairs becausepoints of a closer pair can lie on the opposite sides of the separating line. Therefore, as a step combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width 2*d* around the separating line, since the distance between any other pair of points is at least *d* (figure 2.13a).

Let s be the list of points inside the strip of width 2d around the separating line, obtained from q and hence ordered in nondecreasing order of their y coordinate. We will scan this list, updatingtheinformationaboutdmin,theminimumdistanceseensofar,ifweencounteracloser

Pair of points. Initially, d_{min} = d, and subsequently $d_{min} \le$ d. Let p(x, y) be a point on this list. For a point p (x, y) to have a chance to be closer to p than d_{min}, the point must follow p on list s and the difference between their y coordinates must be less than d_{min}.

Geometrically, this means that p must belong to the rectangle shown in figure 2.13b. The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distancedapart. Itiseasytoprovethatthetotalnumberofsuchpointsintherectangle,includingp, does not exceed 8. A more careful analysis reduces this number to 6. Thus, the algorithm can considernomorethan fivenextpoints followingp onthe lists, beforemovingupto thenext point.

Here is pseudocode of the algorithm. We follow the advice given in to avoid computing square roots inside the innermost loop of the algorithm.

```
**Algorithm***efficientclosestpair(p,q)*
    //solvestheclosest-pairproblembydivide-and-conquer
    //input:anarray*p*of*n*≥2points inthecartesianplanesortedinnondecreasing
    //      orderoftheir*x*coordinates andanarray*q*ofthe samepoints sortedin
    //      nondecreasingorder ofthe *y* coordinates
    //output:euclideandistancebetweentheclosestpairofpoints
    **If***n* ≤ 3
            Returnthe minimaldistancefoundbythebrute-forcealgorithm
    **Else**
            Copythefirst]$n/2$]points of *p*to array$p_l$
            Copy the same ]$n/2$] points from *q* to array
            $q_l$copytheremaining⌊$n/2$]pointsof*p*toarray$p_r$copy
            the same ⌊$n/2$] points from *q* to array $q_r$
            $d_l$←efficientclosestpair($p_l$, $q_l$)
            $d_r$←efficientclosestpair($p_r$, $q_r$)
            $D$ ←min{$d_l$, $d_r$}
            $m$←$p$[]$n/2$]−1].$x$
            Copyall thepoints of*q*forwhich|$x$−$m$|<*d*intoarray*s*[0..*num*−1]
            $Dminsq$←$d^2$
            **For***i* ←0 **to***num*−2 **do**
                    $K$←$i$+1
                    **While***k*≤*num*−1**and** ($s[k].y−s[i].y)^2$<*dminsq*
                            $Dminsq$←min(($s[k].x−s[i].x)^2$+($s[k].y−s[i].y)^2$,*dminsq*) *k*←*k* +
                            1
    **Return***sqrt(dminsq)*
```

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that *n* is a power of 2, we have the following recurrence for the running time of the algorithm:

$$T(n)=2t(n/2)+f(n),$$

Where*f(n)*∈θ*(n)*.applyingthemastertheorem(with*a*=2,*b*=2,and*d*=1),weget *t (n)*∈θ *(n* log*n)*. Thenecessityto presort input points does notchangetheoverall efficiencyclass ifsortingisdonebya*o(n*log*n)*algorithmsuchasmergesort.infact,thisisthebestefficiency

Unitiiidynamicprogrammingandgreedytechnique

Computingabinomialcoefficient

Dynamicprogrammingbinomialcoefficients

Dynamicprogrammingwasinventedbyrichardbellman,1950.itisaverygeneraltechnique for solving optimization problems.

Dynamicprogrammingrequires:

1. Problemdividedintooverlappingsub-problems
2. Sub-problemcanberepresentedbyatable
3. Principle of optimality, recursive relation between smaller and larger problems

comparedtoabruteforcerecursivealgorithmthatcouldrunexponential,thedynamic Programmingalgorithm runstypicallyin quadratic time.therecursivealgorithm ran in exponential Timewhiletheiterativealgorithmraninlinear time.

Computingabinomialcoefficient

Computingbinomialcoefficientsisnonoptimizationproblembutcanbesolvedusing dynamic programming.

Binomialcoefficientsarerepresentedby$c(n,k)$=n!/(k!(n-k)!)Or(^{n})andcanbeusedto Representthecoefficients ofabinomial:

$(a+b)^n=c(n, 0)a^nb^0+.. .+c(n,k)a^{n-k}b^k+.. .+c(n, n)a^0b^n$

Therecursiverelation isdefinedbytheprior power

$C(n,k)=c(n-1, k-1)+c(n-1,k)forn>k>0$withinitialcondition $c(n,0)=c(n,n)=1$

Dynamic algorithm constructs anxktable, with the first column and diagonal filled out using theinitial condition. Construct the table:

	K						
N		**0**	**1**	**2**	**...**	***K*-1**	***K***
	0	1					
	1	1	1				
	2	1	2	1			
	. . .						
	K	1					1
	. . .						
	***N*-1**	1				$C(n-1,k-1)$	$C(n-1,k)$
	N	1					$C(n,k)$

Thetableisthenfilledoutiteratively,rowbyrow usingtherecursive relation.

Algorithm*binomial*(n,k)

```
For$i$←0 to$n$do//filloutthetablerowwise
    For$i$=0to min($i,k$)do
        If $j$==0or$j$==$i$then$c[i, j]$ ←1//initial condition
        Else$c[i,j]$←$c[i-1,j-1]+c[i-1,j]$//recursive relation
Return$c[n,k]$
```

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \le n$, the sumneedstobesplitintotwopartsbecauseonlythehalfthetableneedstobefilledout for $i < k$ and remaining part of the table is filled out across the entire row.

$A(n,k)$=sumforuppertriangle+sumforthelower rectangle

$$=\sum_{i=1}^{k}\sum_{j=1}^{i-1}1+\sum_{i=1}^{n}\sum_{j=1}^{k}1$$

$$=\sum_{i=1}^{k}(i-1)+\sum_{i=1}^{n}k$$

$$=(k-1)k/2+k(n-k)\epsilon\theta(nk)$$

Time efficiency: θ(*nk*)

spaceefficiency:θ(*nk*)

Example:relationofbinomialcoefficients andpascal’striangle.

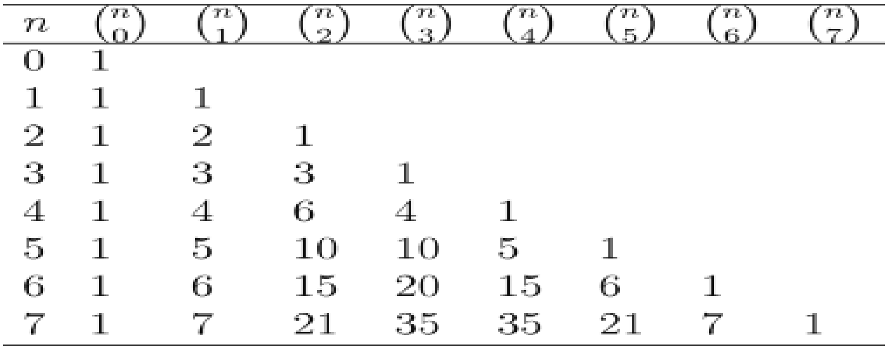
Aformulaforcomputingbinomialcoefficientsis this:

$$\binom{n}{m}=\frac{n!}{(n-m)!m!}$$

Usinganidentitycalledpascal'sformulaarecursiveformulationforitlookslike this:

$$\binom{n}{m}=\begin{cases}1 & \text{if } m=0\\ 1 & \text{if } n=m\\ \binom{n-1}{m}+\binom{n-1}{m-1} & \text{otherwise}\end{cases}$$

This construction forms each number in the triangle is the sum of the two numbers directly aboveit.



n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Findingabinomialcoefficientisassimpleasalookupinpascal's triangle.

Example:$(x+y)^7=1\bullet x^7y^0+7\bullet x^6y^1+21\bullet x^5y^2+35\bullet x^4y^3+35\bullet x^3y^4+21\bullet x^2y^5+7\bullet x^1y^6+1\bullet x^0y^7$

$=x^7+7x^6y+21x^5y^2+35x^4y^3+35x^3y^4+21x^2y^5+7xy^6+y^7$

Warshall'sandfloyd'algorithm

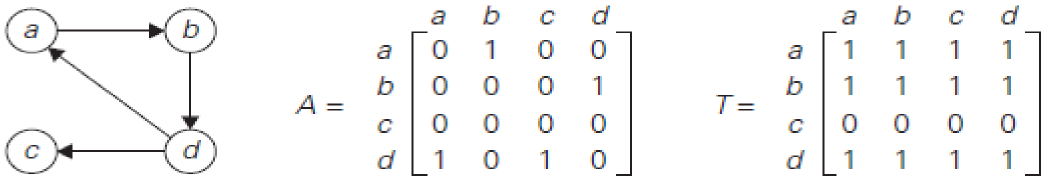
Warshall's and floyd's algorithms: warshall's algorithm for computing the transitive closure (there is a path between any two nodes) of a directed graph and floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on dynamic programming.

Warshall'salgorithm(all-pairspathexistenceproblem)

A **directed graph** (or digraph) is a graph, or set of vertices connected by edges, where the edges have a direction associated with them.

An**adjacency matrix** a={a_{ij}}ofadirectedgraphisthebooleanmatrix thathas1initsith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex.

The **transitive closure** of a directed graph with n vertices can be defined as the n x n booleanmatrix t={t_{ij}},inwhichtheelementintheithrowandthejthcolumnis1ifthereexistsa nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, tij is 0.



$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

$$T = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

(a) (b) (c)

Figure3.1(a)digraph.(b)itsadjacencymatrix.(c)itstransitive closure.

The transitive closure of a digraph can be generated with the help of depth-first search or breadth-first search. Every vertex as a starting point yields the transitive closure for all.

Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices: $r^{(0)}, \ldots, r^{(k-1)}, r^{(k)}, \ldots R^{(n)}$.

Theelement$r_{ij}^{(k)}$inthe*i*throwand*j*thcolumnofmatrix$r^{(k)}(i,j=1,2,\ldots,n,k=0,1,\ldots$, n) is equal to 1 if and onlyif there exists a directed path of a positive length from the *i*th vertex to the *j*th vertex with each intermediate vertex, if any, numbered not higher than *k*.

Stepstocompute$r^{(0)}$,. ..,$r^{(k-1)}$,$r^{(k)}$,...$r^{(n)}$.

- The series starts with $r^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $r^{(0)}$ is nothing other than the adjacency matrix of the digraph.
- $R^{(1)}$containstheinformationaboutpathsthatcanusethefirstvertex asintermediate. It may contain more 1's than $r^{(0)}$.
- The last matrix in the series, r(n), reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.
- In general, each subsequent matrix in series has one more vertex to use as intermediate for its paths than its predecessor.
- The last matrix in the series, $r^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

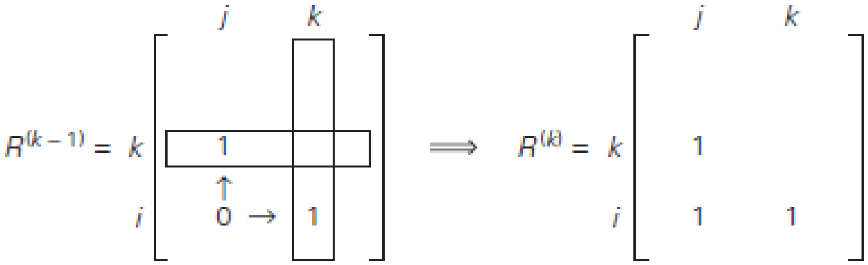


Figure3.2rulefor changingzerosinwarshall'salgorithm.

All the elements of each matrix r$^{(k)}$ is computed from its immediate predecessor r$^{(k−1)}$. L$_{ij}$etr$^{(k)}$, the element in the ith row and jth column of matrix r$^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k.

Thefirstpartofthisrepresentationmeansthatthereexistsapathfrom$v_i$to$v_k$witheach
Intermediatevertexnumberednothigherthan$k$−1(hence,$r^{(k-1)}$=1)$_{ik}$,andthesecondpartmeans
That there exists a path from $v_k$to $v_j$with each intermediate vertex numbered not higher than k − 1(hence, $r_{kj}^{(k-1)}$= 1).

Thusthefollowingformulagenerastheelementsofmatrixr$^{(k)}$fromtheelementsofmatrix
R$^{(k−1)}$:

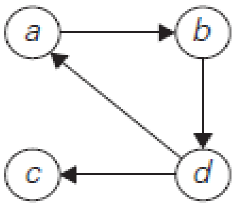
$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right)$$

Applyingwarshall'salgorithmbyhand:

- If anelementr_{ij}is1in$r^{(k-1)}$,it remains1in$r^{(k)}$.
- If an element $r_{ij}$is 0 in $r^{(k-1)}$, it has to be changed to 1 in $r^{(k)}$if and onlyif the element in its row i and column k and the element in its column j and row k are both 1's in$r^{(k-1)}$.

```
Algorithmwarshall(a[1..n,1..n])
    //implementswarshall'salgorithmforcomputingthetransitive closure
    //input:theadjacencymatrixaofadigraphwith nvertices
    //output:thetransitiveclosureofthedigraph r(0) ←a
    Fork←1 to n do
        Fori ←1to n do
            Forj ←1 to n do
                R(k)[i,j]←r(k−1)[i,j]or(r(k−1)[i,k]andr(k−1)[k,j])
    Return r(n)
```

Warshall'salgorithm'stimeefficiencyisonlyθ(n^3).spaceefficiencyisθ(n^2).i.e matrixsize.



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($r^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $r^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex **a** (note a new path from d to b); boxed row and column are used for getting $r^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., **a** and **b** (note two new paths); Boxedrowandcolumnareusedforgetting $r^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., **a**, **b**, and **c** (no new paths); Boxedrowandcolumnareusedforgetting $r^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., **a**, **b**, **c**, and **d** (note five new paths).

Figure3.3applicationofwarshall'salgorithmtothedigraphshown.new1'sarein bold.

Floyd'salgorithm(all-pairsshortest-pathsproblem)

Floyd'slalgorithmisanalgorithmforfindingshortestpathsforallpairsinaweighted Connectedgraph(undirectedordirected)with(+/-)edge weights.

A **distance matrix** is a matrix (two-dimensional array) containing the distances, taken pairwise, between the vertices of graph.

The lengths of shortest paths in an n × n matrix d called the distance matrix: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex.

We can generate the distance matrix with an algorithm that is very similar to warshall's algorithm is called floyd's algorithm.

Floyd'salgorithmcomputesthedistance matrix ofaweighted graph with *n* verticesthrough a series of $n \times n$ matrices:

$D^{(0)},\ldots, d^{(k-1)}, d^{(k)}, \ldots, d^{(n)}$

Theelement$d_{ij}^{(k)}$inthe*i*throw andthe*j*thcolumnof matrix$d^{(k)}$ $(i,j=1,2,\ldots,n,k= 0,1, \ldots,n)$isequaltothelengthoftheshortestpathamongallpathsfromthe*i*thvertextothe*j*th vertex with each intermediate vertex, if any, numbered not higher than *k*.

Stepstocompute$\mathbf{d^{(0)},\ldots,d^{(k-1)},d^{(k)},\ldots,d^{(n)}}$

- The series starts with $d^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $d^{(0)}$is simply the weight matrix of the graph.
- As in warshall's algorithm, we can compute all the elements of each matrix $d^{(k)}$ from its immediate predecessor $d^{(k-1)}$.
- The last matrix in the series, $d^{(n)}$, contains the lengths of the shortest paths amongall paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix.

Let $d_{ij}(^{k)}$ be the element in the ith row and the jth column of matrix $d^{(k)}$. This means that $d_{ij}(^{k)}$ is equal to the length of the shortest path among all paths from the ith vertex v_i to the jthvertex v_j with their intermediate vertices numbered not higher than k.

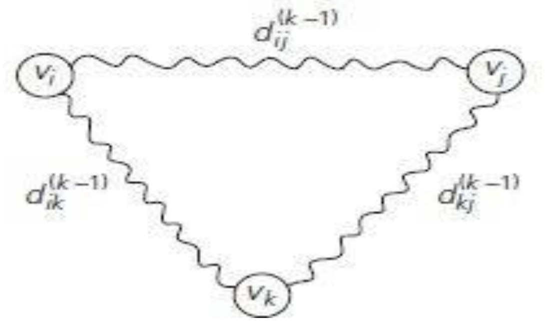


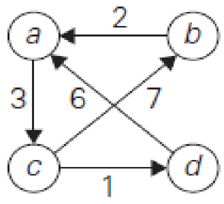
Figure3.4underlyingideaoffloyd'salgorithm.

Thelengthof theshortestpathcanbecomputedbythefollowingrecurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ \ d_{ij}^{(0)} = w_{ij}$$

```
Algorithmfloyd(w[1..n,1..n])
    //implementsfloyd'salgorithmfortheall-pairsshortest-paths problem
    //input:theweightmatrixw ofagraphwithnonegative-lengthcycle
    //output:thedistancematrixoftheshortestpaths'lengths d ←w
    //is not necessary if w can be overwritten
    Fork←1 to n do
        Fori ←1to n do
            Forj ←1 to n do
                D[i,j]←min{d[i, j], d[i, k]+d[k, j]}
    Returnd
```

Floyd'salgorithm'stimeefficiencyisonly$\theta(n^3)$.spaceefficiencyis$\theta(n^2)$.i.ematrixsize.



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($d^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just ***a*** (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., **a** and ***b*** (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., **a**, ***b***, and ***c*** (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., **a**, ***b***, ***c***, and ***d*** (note a new shortest path from *c* to *a*).

Figure 3.5 application of floyd's algorithm to the digraph shown. Updated elements are shown in bold.

Optimalbinarysearchtrees

A binary search tree is one of the most important data structures in computer science. Oneof its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

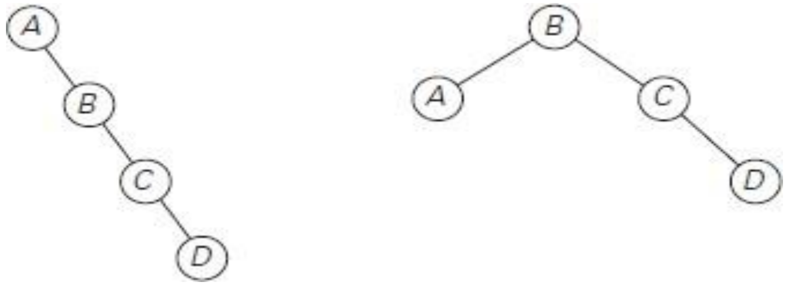


Figure3.6twooutof 14possiblebinarysearchtreeswithkeys*a*,*b*,*c*,and*d*.

Consider four keys a, b, c, and d to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. Figure 3.6 depicts two out of 14 possible binary search trees containing these keys.

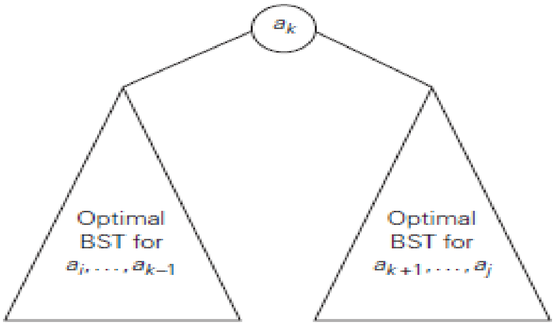
The average number of comparisons in a successful search in the first of these trees is 0.1 . 1+0.2.2+0.4.3+0.3.4=2.9,andfor thesecondoneitis 0.1.2+0.2.1+0.4.2+0.3.3=2.1. Neitherofthesetwotreesisoptimal.

Thetotalnumberofbinarysearchtreeswith*n*keysisequaltothe*n*th**catalan number**,

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1$$

C(n)=(2n)!/(n+1)!N!

Let $a_1, \ldots, a_n$be distinct keys ordered from the smallest to the largest and let $p_1,\ldots,$ $p_n$be the probabilities of searching for them. Let *c(i, j)* be the smallest average number of comparisons made in a successful search in a binary search tree t_i^jmade up of keys $a_i, \ldots, a_j$, where *i, j* are some integer indices, $1\leq i \leq j \leq n$.



a_k
Optimal BST for $a_i, \ldots, a_{k-1}$
Optimal BST for $a_{k+1}, \ldots, a_j$

Figure3.7binarysearchtree (bst)withroot$a_k$andtwooptimalbinarysearchsubtrees T_i^{k-1}andt_{k+1}^j**.**

Consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree (figure 3.7), the root contains key a_k, the left subtree t_i^{k-1} contains keys$a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree t_{k+1}^jcontains keys $a_{k+1}, \ldots, a_j$ also optimallyarranged.

Ifwecounttreelevelsstartingwith1tomakethecomparisonnumbersequalthekeys' Levels,thefollowingrecurrencerelationisobtained:

$$C(i, j) = \min_{i\leq k\leq j}\{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1)$$
$$+ \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\}$$
$$= \min_{i\leq k\leq j}\{\sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s\}$$
$$= \min_{i\leq k\leq j}\{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

$$C(i, j) = \min_{i\leq k\leq j}\{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \leq i \leq j \leq n.$$

Weassumein above formulathat c(i, i −1)=0 for1≤ i ≤ n +1,which can beinterpreted as The number ofcomparisonsintheemptytree.notethatthisformulaimpliesthatc(i,i)=$p_i$for 1≤i ≤n, asitshouldbeforaone-nodebinarysearchtreecontaininga_i.

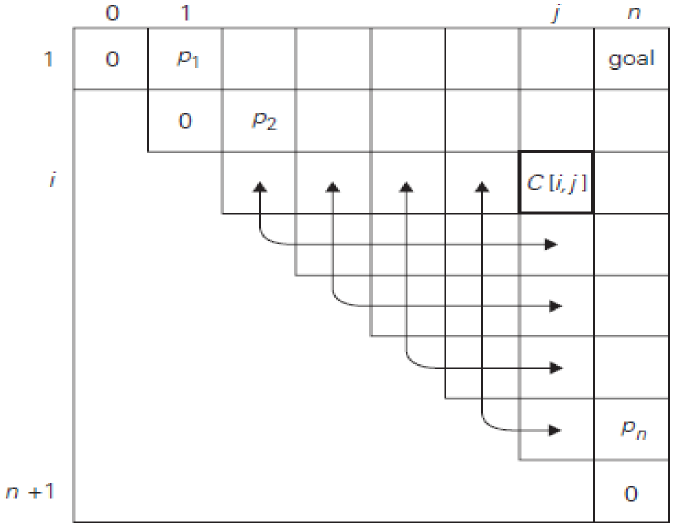


Figure 3.8 table of the dynamic programming algorithm for constructing an optimal binary search tree.

The two-dimensional table in figure 3.8 shows the values needed for computingc(i, j). Theyare in rowi and the columns to the left of column j and in column j and the rows below rowi. The arrows point to the pairs of entries whose sums are computed in order to find the smallest one toberecordedasthevalueofc(i,j).thissuggestsfillingthetablealongitsdiagonals,startingwith all zeros on the main diagonal and given probabilities p_i,$1 \leq i \leq n$, right above it and moving toward the upper right corner.

```
**Algorithm**optimalbst(p[1..n])
    //findsan optimalbinarysearchtreebydynamic programming
    //input:anarrayp[1..n]ofsearchprobabilitiesforasortedlistofnkeys
    //output:averagenumberofcomparisonsinsuccessfulsearchesinthe
    //optimalbstandtabler ofsubtrees'rootsintheoptimal bst
    **For**i ←1 **to** n **do**
        C[i,i−1]←0
        c[i, i]←p[i]
        r[i, i]←i
    C[n+1,n]←0
    **For**d←1**to**n−1**do**//diagonalcount
        **For**i ←1**to** n −d **do**
            J ←i + d
            minval←∞
            **For**k←i **to** j **do**
                **If** c[i, k − 1]+c[k +1, j]< minval
                    Minval←c[i,k − 1]+c[k +1, j]; kmin←k
            R[i,j]←kmin
            sum←p[i];
            **For**s←i +1 **to** j **do**
                Sum←sum+p[s]
            c[i, j ]←minval + sum
```

Returnc[1,n],r

The algorithm’s space efficiencyis clearlyquadratic, ie, **:θ(*n*^{3})**; the time efficiencyof this version of the algorithm is cubic. It is possible to reduce the runningtime of the algorithm to **θ(n^2)** by taking advantage of monotonicity of entries in the root table, i.e., r[i,j] is always in the range between r[i,j-1] and r[i+1,j]

Example: letusillustratethealgorithmbyapplyingittothefour-keysetweusedatthe beginning of this section:

Key

abcdprobability0.10.20.

40.3 the initial tables are:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Letus compute*c(*1,2*)*:

$$C(1,2) = \min \begin{cases} k=1: & C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: & C(1,1) + C(3,2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{cases}$$

$$= 0.4.$$

Thus, out of two possiblebinarytrees containing the first two keys, a and b, the root of the optimal treehas index 2 (i.e., it contains b), and theaveragenumberofcomparisonsin asuccessful search in this tree is 0.4.

Wearriveatthe followingfinaltables:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since r(1, 4) = 3, the root of the optimal tree contains the third key, i.e., c. Its left subtree is made up of keys a and b, and its right subtree contains just key d. To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since r(1, 2) = 2, the root of the optimal tree containing a and b is b, with a being its left child (and the root of the one node tree:r(1,1)=1).sincer(4,4)=4,therootofthisone-nodeoptimaltreeisitsonlykeyd.figure 3.10presentstheoptimaltreeinits entirety.

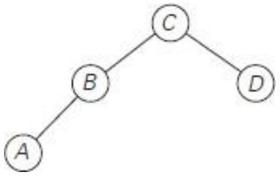


Figure3.10optimalbinarysearch treeforthe above example.

Knapsackproblemandmemoryfunctions

Designingadynamicprogrammingalgorithmfortheknapsackproblem:

Givennitemsofknownweightsw$_1$,...,w$_n$andvaluesv$_1$,...,v$_n$andaknapsackof capacity w, find the most valuable subset of the items that fit into the knapsack.

Assume that all the weights and the knapsack capacityare positive integers; the item values do not have to be integers.

0/1knapsackproblemmeans,thechosenitemshouldbeeithernullor whole.

Recurrencerelationthatexpressesasolutiontoaninstanceoftheknapsackproblem

Let us consider an instance defined by the first *i* items, 1≤ *i* ≤ *n*, with weights $w_1, \dots, w_i$, values $v_1, \dots, v_i$, and knapsack capacity*j*, $1 \le j \le w$. Let *f(i, j)*be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first *i* items that fit into the knapsack of capacity *j*. We can divide all the subsets of the first *i* items that fit the knapsack of capacity *j* into two categories: those that do not include the *i*th item and those that do. Note the following:

1. Among the subsets that do not include the *i*th item, the value of an optimal subset is, by definition, *f(i −* 1*, j)*.
2. Amongthesubsets that do includethe *i*th item (hence, $j - w_i \ge 0$), an optimal subset is made up of this item and an optimal subset of the first *i* − 1 items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is *vi* + *f(i −* 1*, j − w$_i$)*.

Thus, the value of an optimal solution among all feasible subsets of the first *i* items is the maximum of these two values. Of course, if the *i*th item does not fit into the knapsack, the value of an optimal subset selected from the first *i* items is the same as the value of an optimal subset selected from the first *i* − 1 items. These observations lead to the following recurrence:

$$F(i, j) = \begin{cases} \max\{F(i-1, j),\ v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \ge 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

Itisconvenienttodefine theinitialconditionsasfollows:

$F(0,j)=0$for$j\ge0$ and$f(i,0)=0$for$i\ge0$.

Our goal is to find *f(n, w),* the maximal value of a subset of the *n* given items that fit into the knapsack of capacity *w*, and an optimal subset itself.

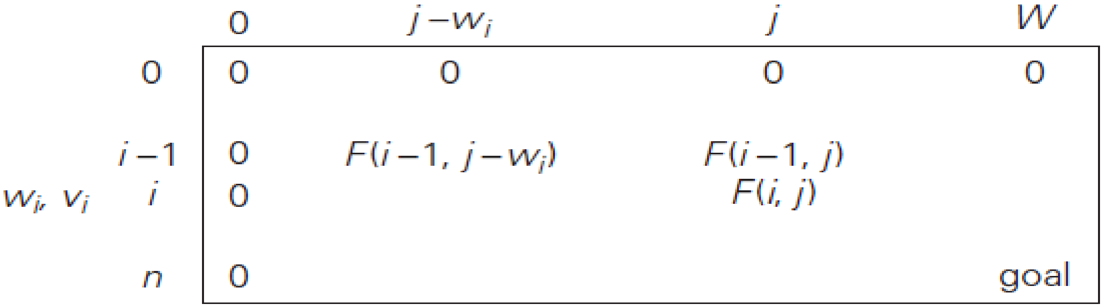
*For f(i, j),*computethe maximum oftheentryin theprevious rowand the samecolumn and the sum of *vi* and the entry in the previous row and *w$_i$*columns to the left. The table can be filled either row by row or column by column.

```
Algorithmdpknapsack(w[1..n],v[1..n],w)
     Varv[0..n,0..w],p[1..n,1..w]:int
     Forj :=0 towdo
           V[0,j]:=0
      Fori :=0 to n do
           V[i,0]:=0
      Fori :=1 to n do
          Forj :=1 towdo
```

```
        Ifw[i]≤jandv[i]+v[i-1,j-w[i]]>v[i-1,j] then
                V[i,j]:=v[i]+v[i-1,j-w[i]];p[i,j]:=j-w[i]
        Else
                V[i,j]:=v[i-1,j];p[i,j]:=j
    Returnv[n,w]andtheoptimalsubsetbybacktracing
```

Note:runningtimeand space: o(nw).

Table3.1forsolvingtheknapsack problembydynamicprogramming.



		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Example1letusconsidertheinstancegivenbythefollowingdata: table 3.2 an instance of the knapsack problem:

item	**weight**	**value**	capacity
1	2	$12	*W*=5
2	1	$10	
3	3	$20	
4	2	$15	

The maximal value is *f*(4, 5) = $37. We can find the composition of an optimal subset by **backtracing** (back tracing finds the actual optimal subset, i.e. Solution), the computations of this entry in the table. Since *f*(4, 5) > *f*(3, 5), item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity. The value of thelatteris *f*(3, 3). Since *f*(3, 3)=*f*(2, 3), item 3 need not bein an optimalsubset. Since *f*(2, 3)> *f*(1, 3), item 2 is a part of an optimal selection, which leaves element *f*(1, 3 − 1) to specify its remaining composition. Similarly, since *f*(1, 2) > *f*(0, 2), item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

Table3.3solvinganinstanceoftheknapsackproblem bythe dynamicprogramming algorithm.

	Capacityj						
	I	0	1	2	3	4	5
	0	0	0	0	0	0	0
*W*1=2, *v*1 =12	1	0	0	12	12	12	12
*W*2=1, *v*2 =10	2	0	10	12	22	22	22
*W*3=3, *v*3 =20	3	0	10	12	22	30	32
*W*4=2, *v*4 =15	4	0	10	15	25	30	**37**

Memoryfunctions

The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient.

The bottom up fills a table with solutions to all smaller subproblems, but each of them is solved onlyonce. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given.

Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using **memory functions.**

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm.

Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with i = n (the number of items) and j = w (the knapsack capacity).

Algorithmmfknapsack(i,j)

```
    //implementsthememoryfunctionmethodforthe knapsackproblem
    //input:anonnegativeintegeri indicatingthenumberofthefirstitemsbeing considered
    //          andanonnegativeintegerj indicatingthe knapsackcapacity
    //output:thevalue ofanoptimal feasiblesubset of thefirst i items
    //note:usesasglobalvariablesinputarrays weights[1..n], values[1..n],
    //          andtablef[0..n,0..w] whoseentriesare initializedwith−1'sexceptfor
    //          row0andcolumn 0initializedwith 0's
    If f[i,j]<0
        Ifj <weights[i]
            Value←mfknapsack(i−1,j)
        Else
        Value←max(mfknapsack(i−1,j),
            Values[i]+mfknapsack(i−1,j−weights[i]))
        F[i,j ]←value
    Returnf[i,j]
```

Example2letus applythememoryfunctionmethodtotheinstanceconsideredin example1.

	Capacityj						
	I	0	1	2	3	4	5
	0	0	0	0	0	0	0
*W*1=2, *v*1 =12	1	0	0	12	12	12	12
*W*2=1, *v*2 =10	2	0	-	12	22	-	22
*W*3=3, *v*3 =20	3	0	-	-	22	-	32
*W*4=2, *v*4 =15	4	0	-	-	-	-	37

Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, v (1, 2), is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger.

Greedytechnique

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step and this is the central point of this technique.

Thechoicemademustbe:

- *Feasible*,i.e.,ithastosatisfytheproblem's constraints
- *Locallyoptimal*,i.e.,ithastobethebestlocalchoiceamongallfeasiblechoicesavailableon that step
- *Irrevocable*,i.e.,oncemade,itcannotbechanged onsubsequentstepsofthe algorithm

Greedytechniquealgorithmsare:

- Prim'salgorithm
- Kruskal'salgorithm
- Dijkstra'salgorithm
- Huffmantrees

Two classic algorithms for the minimum spanning tree problem: prim's algorithm andkruskal's algorithm. They solve the same problem by applying the greedy approach in twodifferent ways, and both of them always yield an optimal solution.

Another classic algorithm nameddijkstra's algorithm used to find the shortest-path in a weighted graph problem solved by greedy technique . Huffman codes is an important data compression method that can be interpreted as an application of the greedy technique.

Thefirstwayisoneofthecommonwaystodotheproofforgreedytechniqueisby **Mathematicalinduction**.

The second way to prove optimality of a greedy algorithm is to show that on each step it does at least as well as any other algorithm could in **advancing** toward the problem's goal.

Example: findthe minimum number of moves needed for a chess knight to go from one corner of a 100 × 100 board to the diagonally opposite corner. (the knight's moves are l-shaped jumps: two squares horizontally or vertically followed by one square in the perpendicular direction.)

A greedysolution is clear here: jump as close to the goal as possible on each move. Thus, if its start and finish squares are (1,1) and (100, 100), respectively, a sequence of 66 moves such as(1, 1) − (3, 2) − (4, 4) − . . . − (97, 97) − (99, 98) − (100, 100) solves the problem(the number k of two-move advances can be obtained from the equation 1+ 3k = 100).

Why is this a minimum-move solution? Because if we measure the distance to the goal by the manhattan distance, which is the sum of the difference between the row numbers and the difference between the column numbers of two squares in question, the greedy algorithm decreases it by 3 on each move.

The third way is simply to show that the final result obtained by a greedy algorithm is optimal based on the **algorithm’s output** rather than the way it operates.

Example:considertheproblemofplacingthemaximumnumberofchips onan8×8boardsothat no two chips are placed on the same or adjacent vertically, horizontally, or diagonally.

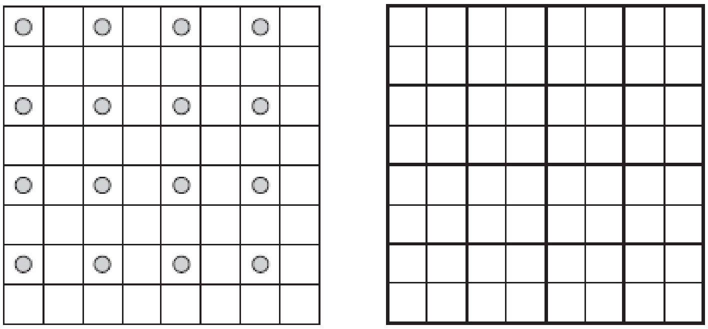


Figure3.12(a)placementof16chipsonnon-adjacentsquares.(b)partitionoftheboard proving impossibility of placing more than 16 chips.

It is impossible to place more than one chip in each of these squares, which implies that the total number of nonadjacent chips on the board cannot exceed 16.

Prim’salgorithm

A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

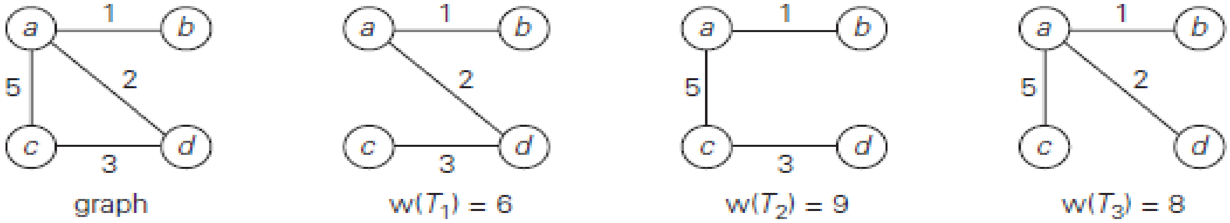


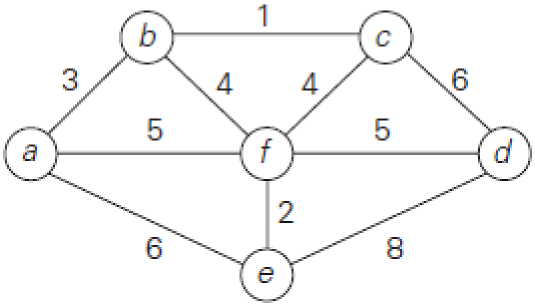
Figure3.13graphanditsspanningtrees,with *t*1beingtheminimumspanningtree.

The minimum spanning treeis illustrated in figure 3. If we were to try constructing a minimum spanning tree by exhaustive search, we would face two serious obstacles. First, the number of spanning trees grows exponentially with the graph size (at least for dense graphs). Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a minimum spanning tree for a weighted graph.

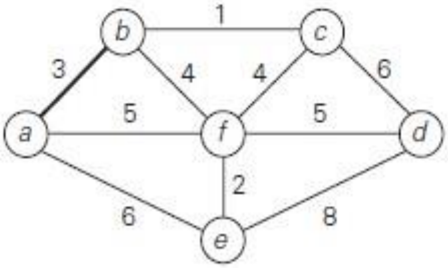
Prim’s algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set v of the graph’s vertices. On each iteration, the algorithm expands the current tree in the greedymannerbysimplyattachingto it the nearest vertex not inthat tree. Thealgorithm stops after all the graph’s vertices have been included in the tree being constructed.

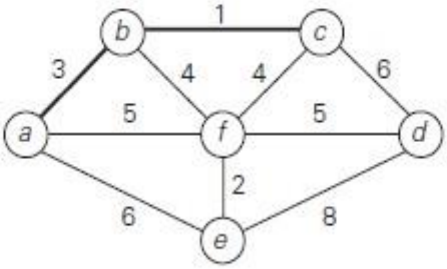
```
Algorithmprim(g)
    //prim’salgorithmforconstructingaminimumspanningtree
    //input:aweightedconnectedgraph $g=\{v,e\}$
    //output:$e_t$,thesetofedgescomposingaminimumspanningtreeofg $v_t \leftarrow \{v_0\}$
    //the set of tree vertices can be initialized with any vertex $e_t \leftarrow \varphi$
    For$i$ $\leftarrow 1$to$|v|-1$do
        Findaminimum-weightedge$e^*=(v^*,u^*)$amongalltheedges(v,u) such that
        $v$ is in $v_t$ and u is in $v - v_t$
        $V_t \leftarrow v_t \cup \{u^*\}$
        $E_t \leftarrow e_t \cup \{e^*\}$
    Return$e_t$
```

Ifagraphisrepresentedby itsadjacency listsandthepriority queueisimplementedasa min-heap,therunningtimeofthealgorithmiso(|e|log|v|)inaconnectedgraph,where|v|−1≤ |e|.



Tree vertices	Remaining vertices	Illustration

a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞)
	e(a, 6) f(a, 5)
		 a
		b
		c
		d
		e
		f
		1
		3
		4
		4
		6
		5
		5
		2
		6
		8

b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6)
	f(b, 4)
		 a
		b
		c
		d
		e
		f
		1
		3
		4
		4
		6
		5
		5
		2
		6
		8

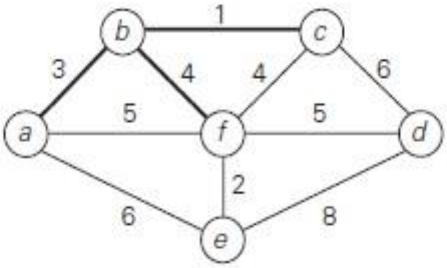
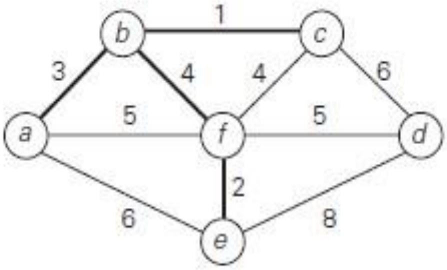
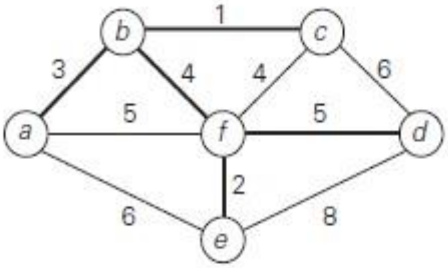
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	 a b c d e f 1 3 4 4 6 5 5 2 6 8
f(b, 4)	d(f, 5) **e(f, 2)**	 a b c d e f 1 3 4 4 6 5 5 2 6 8
e(f, 2)	**d(f, 5)**	 a b c d e f 1 3 4 4 6 5 5 2 6 8
d(f, 5)		

Figure3.14applicationofprim’salgorithm.theparenthesizedlabelsofavertexinthemiddle Columnindicatethenearesttreevertexandedge weight;selectedverticesandedgesareinbold.

Kruskal'salgorithm

Kruskal'salgorithmlooksataminimumspanningtreeofaweightedconnectedgraph g= {v, e} as an acyclic subgraph with |v| − 1 edges for which the sum of the edge weights is the smallest. Thealgorithm constructs aminimum spanningtreeas an expandingsequenceofsubgraphs that are always acyclic but are not necessarily connected on the intermediate stages of thealgorithm.

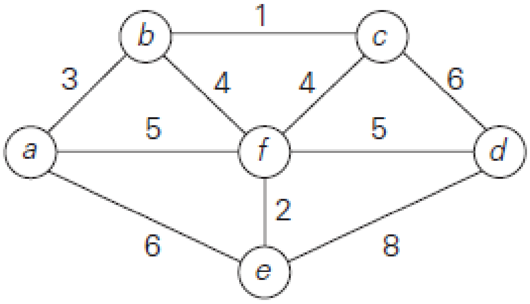
The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Kruskal'salgorithmlooksataminimumspanningtreeofaweightedconnectedgraph g = (v, e) as an acyclic subgraph with |v| − 1 edges for which the sum of the edge weights is the smallest.

```
Algorithmkruskal(g)
    //kruskal'salgorithmforconstructingaminimumspanningtree
    //input:aweighted connectedgraph $g=(v,e)$
    //output:$e_t$,thesetofedgescomposingaminimumspanningtreeof $g$
            Sorteinnondecreasingorderoftheedgeweights $w(e_{i1})\leq\ldots\leq w(e_{i|e|})$ $e_t\leftarrow \varphi$;
    $ecounter \leftarrow 0$                //initialize the set of tree edges and its size
    $K \leftarrow 0$                       //initializethenumberofprocessed edges
    While $ecounter<|v|-1$ do
        $K\leftarrow k+1$
        If $e_t\cup\{e_{ik}\}$ is acyclic
            $E_t\leftarrow e_t\cup\{e_{ik}\}$; $ecounter \leftarrow ecounter+1$
    Return $e_t$
```

The initial forest consists of |v | trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containingthevertices u and v, and, ifthese trees arenot thesame,unites them in alargertree by adding the edge (u, v).

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called union-find algorithms. With an efficient union-find algorithm, the running time of kruskal's algorithm will be $o(|e| \log |e|)$.



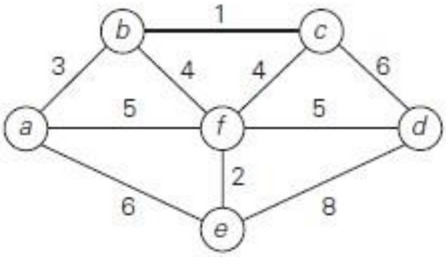
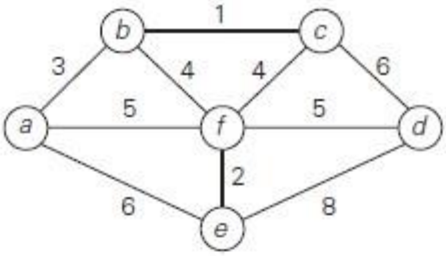
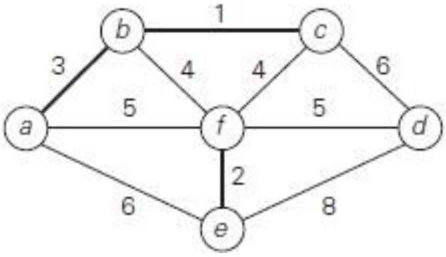
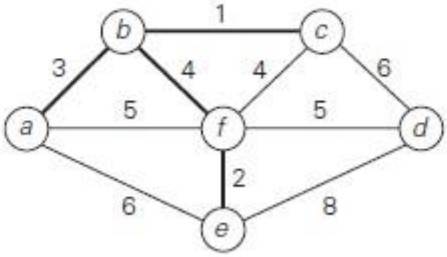
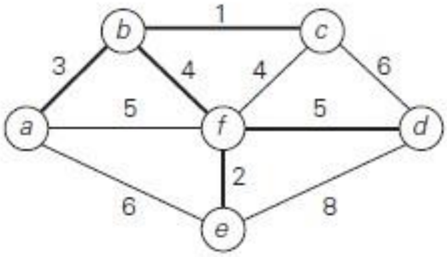
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

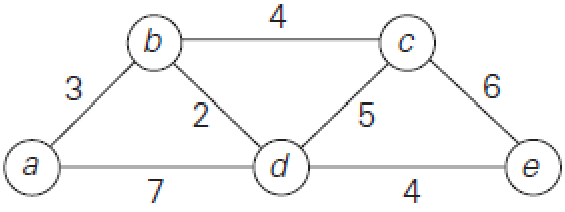
Figure3.15applicationofkruskal’s algorithm.selectededges areshownin bold.

Dijkstra'salgorithm

- Dijkstra’salgorithmsolvesthe**single-sourceshortest-pathsproblem**.
- Foragivenvertexcalled the***source***inaweighted connected graph,findshortestpathstoall its other vertices.
- The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have **edges in common**.
- The most widely used **applications** are transportation planning and packet routing in communication networks including the internet.
- Italsoincludes**finding shortest paths**insocialnetworks,speech recognition,document formatting, robotics, compilers, and airline crew scheduling.
- Intheworldof**entertainment**,onecanmentionpathfindinginvideogamesandfinding best solutions to puzzles using their state-space graphs.
- Dijkstra’salgorithmisthebest-knownalgorithmforthesingle-sourceshortest-paths problem.

```
Algorithmdijkstra(g,s)
    //dijkstra’salgorithmforsingle-sourceshortestpaths
    //input:aweightedconnectedgraph $g=(v,e)$withnonnegativeweightsanditsvertex$s$
    //output:thelength$dv$ofashortestpathfrom$s$to$v$anditspenultimatevertex$pv$forevery
    //          vertex $v$in$v$
    Initialize(q)//initializepriorityqueuetoempty
    Foreveryvertex$v$in $v$
        $Dv \leftarrow \infty$; $pv \leftarrow$ null
        Insert(q,v,dv)//initializevertexpriorityinthepriorityqueue
    $Ds \leftarrow 0$;decrease(q,s,$d_s$)//updatepriorityof$s$with$d_s$ $v_t \leftarrow \varphi$
    For$i \leftarrow 0$to$|v|-1$do
        $U^* \leftarrow$deletemin(q)//deletetheminimumpriorityelement
        $V_t \leftarrow v_t \cup \{u^*\}$
        Foreveryvertex $u$in$v-vt$thatisadjacentto$u^*$do if $d_{u^*}+$
            $w(u^*, u) < d_u$
            $D_u \leftarrow d_{u^*}+w(u^*,u)$;$p_u \leftarrow u^*$
            decrease(q, u, $d_u$)
```

The time efficiency of dijkstra’s algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. It is in θ ($|v|^2$) for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priorityqueue implemented as a min- heap, it is in o(|e| log |v |).



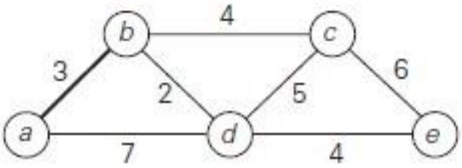
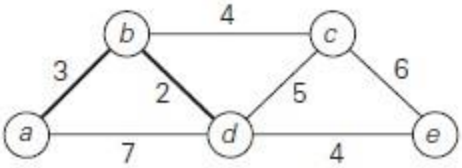
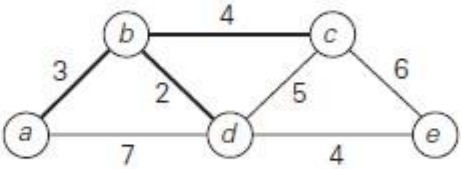
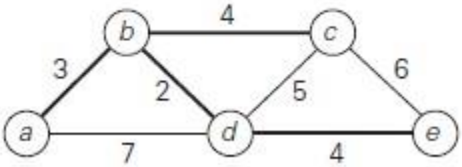
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

Figure3.16applicationofdijkstra’salgorithm.thenextclosestvertexisshowninbold

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

From a to b : a − b of length 3

fromatod:a−b−doflength5 fromato

c: a −b−coflength7

Fromatoe :a−b −d−eof length9

Huffmantrees

To encode atext that comprisessymbols from some *n*-symbol alphabet byassigningto each of the text’s symbols some sequence of bits called the ***codeword***. For example, we can use a ***fixed-length encoding*** that assigns to each symbol a bit string of the same length *m* (*m* ≥ log2 *n*). This is exactly what the standard ascii code does.

Variable-length encoding, which assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the *i*th) symbol? To avoid this complication, we can limit urselvesto the so-called ***prefix-free*** (or simply ***prefix***) ***codes***.

In a prefix code, no codeword is a prefix of a codeword of another symbol. Hence, withsuch an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits bythis symbol, and repeat this operation until the bit string’s end is reached.

Huffman'salgorithm

Step 1initialize *n* one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (more generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2repeat the following operation until a single tree is obtained. Find two trees withthe smallest weight (ties can be broken arbitrarily, but see problem 2 in thissection's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a **huffman tree**. It defines in themanner described above is called a **huffman code**.

Example consider the five-symbol alphabet {a, b, c, d, _} with the following occurrence frequencies in a text made up of these symbols:

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15

Thehuffmantreeconstructionforthisinputisshowninfigure 3.18

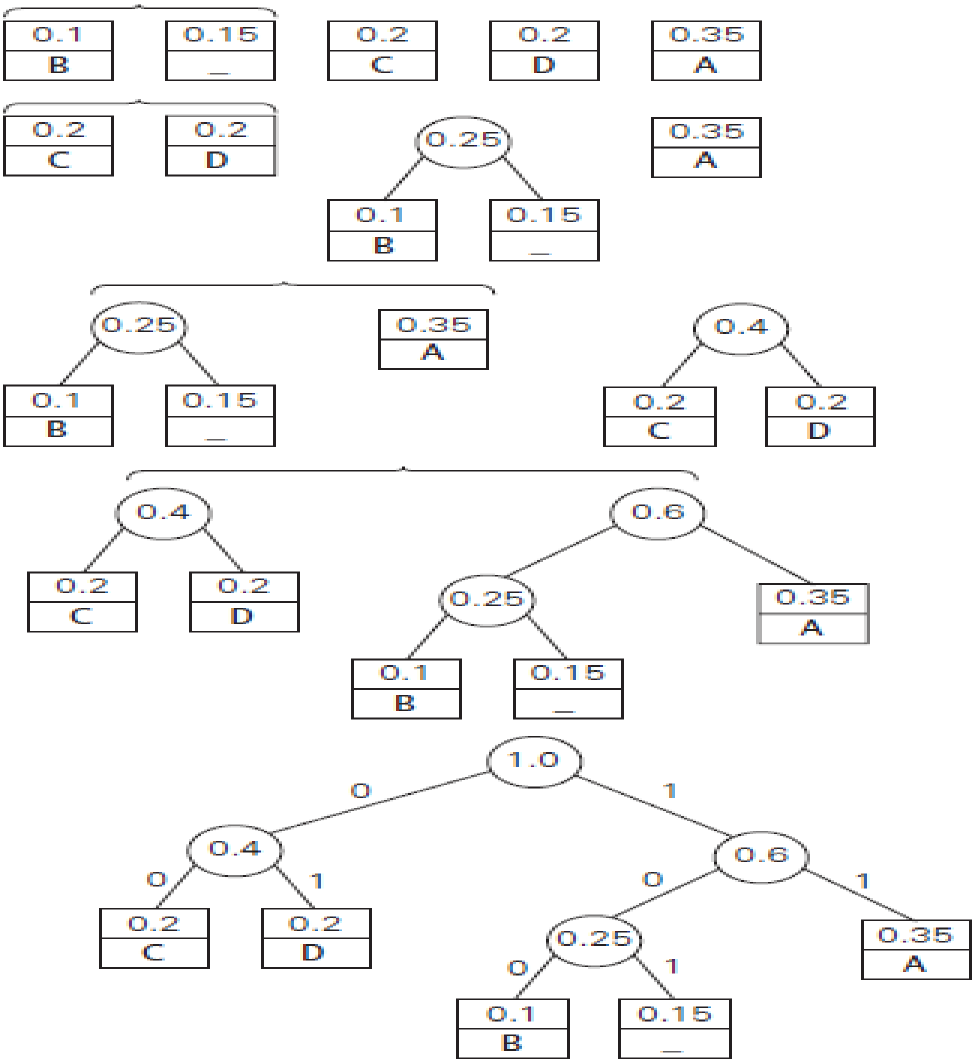


Figure3.18exampleofconstructingahuffmancodingtree.

Theresultingcodewordsareas follows:

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15
Codeword	11	100	00	01	101

Hence, dad is encoded as 011101, and 10011011011101 is decoded as bad_ad. Withthe occurrence frequencies given and the codeword lengths obtained, the average number of bitsper symbol in this code is 2 . 0.35 + 3 . 0.1+ 2 . 0.2 + 2 . 0.2 + 3 . 0.15 = 2.25.

We used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this toy example, huffman's code achieves the ***compression ratio* -** a standard measure of a compression algorithm's effectiveness of (3− 2.25) / 3 · 100% = **25%**. In other words, huffman's encoding of the text will use 25% less memory than its fixed-length encoding.

Runningtimeis o(*n logn*),aseach priorityqueueoperationtakes timeo(*log n*).

Applicationsofhuffman'sencoding

1. Huffman'sencodingisavariablelengthencoding,sothatnumberofbitsusedarelesser than fixed length encoding.
2. Huffman's encodingisveryusefulforfile compression.
3. Huffman's codeisusedintransmissionofdatainanencodedformat.
4. Huffman's encodingisusedindecisiontrees andgame playing.

Unitiviterativeimprovement

Thesimplexmethod

Linearprogramming

Linear programming problem (lpp) is to optimize a linear function of several variables subject to linear constraints:

Maximize(orminimize)$c_1x_1+...+c_nx_n$

Subject to $a_{i1}x_1+ ...+ a_{in}\, x_n \le$ (or $\ge$ or $=$) b_i , $i = 1,...,m$ $x_1\ge0,...,x_n\ge0$ the function $z = c_1x_1+ ...+ c_nx_n$ is called the *objective function*;

Constraints$x_1 \ge 0,..., x_n\ge0$ arecalled *nonnegativity constraints*

Example

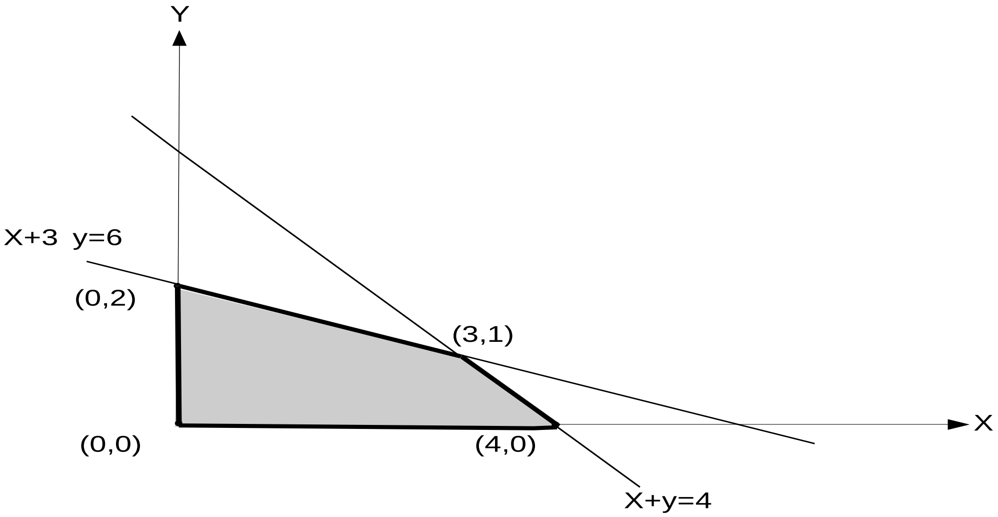
Maximize 3x+5y

Subjectto x +y≤ 4

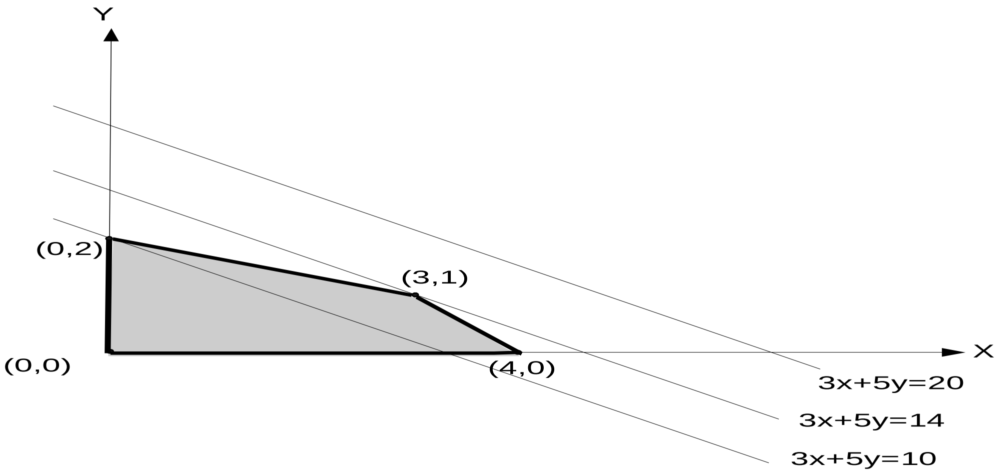
X+ 3y≤6

X≥ 0,y≥0

Feasibleregionisthesetofpointsdefinedbythe constraints



Geometricsolution



Optimalsolution:x=3,y=1

Extreme point theoremany lp problem with a nonempty bounded feasible region has an optimal solution; moreover, an optimal solution can always be found at an *extreme point* of the problem's feasible region.

Threepossibleoutcomesinsolvinganlpproblem

- Has a finite optimal solution, which maynot beunique
- *Unbounded*:theobjectivefunctionofmaximization(minimization)lpproblemisunbounded from above (below) on its feasible region
- *Infeasible*:therearenopointssatisfyingalltheconstraints,i.e.theconstraintsare contradictory

Thesimplexmethod

- The classic method for solving lp problems; one of the most important algorithms ever invented.
- Inventedbygeorgedantzigin1947.
- Basedontheiterativeimprovement idea.
- Generatesasequenceof adjacentpointsoftheproblem’sfeasibleregionwithimproving values of the objective function until no further improvement is possible.

Standardformoflpproblem

- Mustbea**maximization** problem
- Allconstraints(exceptthenonnegativityconstraints)mustbeintheformoflinear equations
- Allthevariablesmustbe requiredtobenonnegative
- Thus, the general linear programming problem in standard form with *m* constraints and *n* unknowns ($n \geq m$) is
- Maximize$c_1x_1+...+c_nx_n$
- Subject to$a_{i1}x_1+...+a_{in}x_n=b_i,i=1,...,m,$ $x_1 \geq 0, ... , x_n\geq 0$

Example

Maximize$3x+5y$

Subjectto $X+y \leq 4$

$X+3y\leq 6$

$X\geq0,y\geq0$

maximize$3x+5y+0u+0v$

Subjectto $X+y+u=4$

$X+3y+v=6$

$X\geq0,y\geq0,\quad U\geq0,v\geq0$

Variables*u*and*v*,transforminginequalityconstraintsintoequalityconstrains,arecalled*slack variables*

Basicfeasible solutions

A *basic solution* to a system of *m* linear equations in *n* unknowns ($n \geq m$) is obtained by setting $n - m$ variables to 0 and solving the resulting system to get the values of the other *m* variables.the variables set to 0 are called *nonbasic*; the variables obtained bysolving the system are called *basic.*

Abasicsolutioniscalled *feasible*ifallits(basic)variablesarenonnegative.

Example$x+y+u=4$

$X+3y+v=6$

(0,0,4,6)isbasicfeasible solution

(x, $y$arenonbasic;u, $v$are basic)

Thereisa1-1correspondencebetweenextremepointsoflp’sfeasibleregionanditsbasicfeasible Solutions.

Simplextableau

Maximize $z = 3x + 5y + 0u + 0v$

subjectto $x+y+u \qquad =4$

$X+3y \qquad +v=6$

$X\geq0,y\geq0,u\geq0,v\geq0$

$$\begin{array}{c|cccc|c|}
 & X & y & u & v & \\ \hline
U & 1 & 1 & 1 & 0 & 4 \\
 & 1 & 3 & 0 & 1 & 6 \\ \hline
v & -3 & -5 & 0 & 0 & 0 \\ \hline
\end{array}$$

Objectiverow

Basicvariables=u,v

Basicfeasiblesolution=(0,0, 4, 6)

Valueofz at (0,0, 4, 6) =0

Outlineofthesimplex method

Step0[initialization]presentagivenlpprobleminstandardformandsetupinitial tableau.

Step 1 [optimality test] if all entries in the objective row are nonnegative then stop: the tableau represents an optimal solution.

Step 2 [find entering variable] select the most negative entry in the objective row.mark its column to indicate the entering variable and the **pivot column**.

Step 3 [find departing (leaving) variable]for each positive entry in the pivot column, calculatethe θ-ratio by dividing that row's entry in the rightmost column (solution) by its entry in the pivot column.(if there are no positive entries in the pivot column then stop: the problem is unbounded.)Find the row with the smallest θ-ratio, mark this row to indicate the departing variable and the **pivot row.**

Step 4 [form the next tableau] divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to step 1.

Exampleofsimplexmethodapplication

$$\begin{array}{c|cccc|c|}
 & x & y & u & v & \\
\hline
u & 1 & 1 & 1 & 0 & 4 \\
\leftarrow v & 1 & 3 & 0 & 1 & 6 \\
\hline
 & 3 & 5 & 0 & 0 & 0 \\
\hline
 & & \uparrow & & &
\end{array}$$

basicfeasiblesol.(0, 0,4, 6)z =0

$$\begin{array}{c|cccc|c|}
 & x & y & u & v & \\
\hline
\leftarrow U & \frac{2}{3} & 0 & 1 & -\frac{1}{3} & 2 \\
 & \frac{1}{3} & 1 & 0 & \frac{1}{3} & 2 \\
\hline
y & -\;- & & & - & \\
\hline
 & 3 & 0 & 0 & 3 & 10 \\
 & \uparrow & & & &
\end{array}$$

Basicfeasiblesol.(0, 2, 2,0)z =10

$$\begin{array}{c|cccc|c|}
 & X & Y & U & V & \\
\hline
X & 1 & 0 & 3/2 & -1/3 & 3 \\
Y & 0 & 1 & -1/2 & 1/2 & 1 \\
\hline
 & 0 & 0 & 2 & 1 & 14 \\
\hline
\end{array}$$

Basicfeasiblesol.(3, 1,0, 0)z =14

Notesonthesimplexmethod

- Findinganinitialbasic feasiblesolutionmaypose a problem.
- Theoreticalpossibilityof cycling.
- Typicalnumberofiterationsisbetweenmand3m,wheremis the number of equality constraints in the standard form.
- Worse-caseefficiencyis **exponentia**l.
- More recent interior-point algorithms such as karmarkar'salgorithm (1984) have polynomial worst-case efficiency andhave performed competitively with the simplex method inempirical tests.

Example1:

Usesimplexmethodtosolvetheformersproblem givenbelow.

A farmer has a 320 acre farm on which she plants two crops: corn and soybeans. For eachacreofcornplanted,herexpensesare$50andforeachacreofsoybeansplanted,herexpensesare $100.eachacreofcornrequires100bushelsofstorageandyieldsaprofitof$60;eachacreof

Soybeans requires 40 bushels of storage and yields a profit of \$90. If the total amount of storage space available is 19,200 bushels and the farmer has only \$20,000 on hand, how many acres ofeach crop should she plant in order to maximize her profit? What will her profit be if she follows this strategy?

Solution

Linearprogrammingproblemformulation

	Corn	Soybean	**Total**
Expenses	\$50	\$100	\$20,000
Storage(bushels)	100	40	19,200
Profit	60	90	Maximizeprofit

Afarmerhasa320acrefarmisunwanteddatabutc+s<=320. C =

corn planted acres and s = soybean planted acres

$50c+100s\leq 20,000$

$100c+40s\leq 19,200$

Maximize:$60c+90s=p$

Canonicalformoflpp

Maximize:$60c +90s$

Subjectto $50c+100s =20000$

$100c+40s=19200$

$C\geq 0, s\geq 0$

Solvingbyalgebra(intersectionof lines)

Maximize: $60c+90s$

Subjectto $50c+100s =20000$ (1)

$100c+40s=19200$ (2)

$(1)/50 \Rightarrow c+2s= 400$

$(2)/20 \Rightarrow 5c+2s=960$

$(2)–(1) \Rightarrow$ $4c= 560$

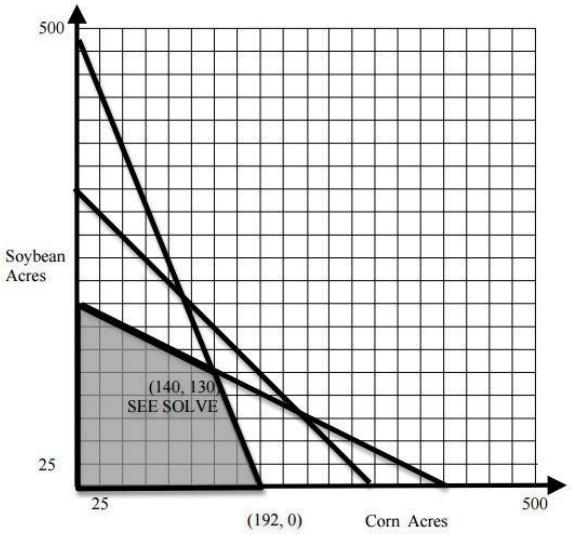
$C= 140$

Substitutec=140 in(1)then s=130

Profit:$p=60c+90s=60(140) +90(130)=\$20,100$

Sheshouldplant140acrescornand130acresof soybeanfor\$20,100.

Solvingbygraphicalmethod



Profitat(0,200)=60c + 90s=60(0)+90(200)= $18,000

Profit at(192,0)=60c + 90s=60(192)+90(0)= $11,520

Profitat(140,130)=60c +90s=60(140) +90(130)= $20,100

Sheshouldplant140acrescornand130acresof soybeanfor$20,100.

Solvingbysimplexmethod

Canonicalformof lpp

Maximize:60x +90y

Subjectto $50x+100y+ s_1=20000$

$100x+40y+s_2=19200$

$X\geq 0,y\geq 0$

Iterationi

CPR

Basic	z	x	y	s_1	s_2	Solution
s_1	0	50	**100**	1	0	**20000**
s_2	0	100	40	0	1	19200
z	1	-60	**-90**	0	0	0

Selectleastratioo

Solution/pivotelements

20000/100 =200 √

Select the most negative value in row z.

Pivot element :intersectionofpivotrowandpivotcolumn:100

basic variables : $s_{1,}$ $s_{2,z}$

Non basic variables :x,y

enter variable : y

Leavevariable : s_1

Initialsolutionat($x,y,s_{1,}s_2$)=(0,0,20000,19200) initial solution z = 0

Pivotrow:

Replacethe**leavingvariable**inbasiccolumnwiththe**enteringvariable**. **New pivot row = current pivot row / pivot element**

Allotherrowsincludingz:

Newrow=currentrow–(itspivotcolumncoefficient)*newpivotrow

Row y

Newpivotrow=currentpivotrow/pivot element

$=(0, 50, 100, 1, 0, 20000) /100$

$=(0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200)$

Rows$_2$

Newrow=currentrow–(itspivotcolumncoefficient)*newpivotrow

$=(0, 100, 40, 0, 1, 19200)-(40)*(0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200)$

$=(0,80,0,\frac{-4}{10},1,96)$

Rowz

Newrow=currentrow–(itspivotcolumncoefficient)*newpivotrow

$=(1, -60, -90, 0,0, 0)-(-90)*(0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200)$

$=(1, -60, -90, 0,0, 0)+(90)*(0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200)$

$=(1,-15, 0,\frac{9}{10},0,18000)$

Iterationii

NPR

Basic	z	x	y	s_1	s_2	Solution
y	0	1/2	1	1/100	0	200
s_2	0	80	0	-4/10	1	96
z	1	-15	0	9/10	0	18000

CPR

Basic	z	x	y	s_1	s_2	Solution
y	0	1/2	1	1/100	0	200
s_2	0	$\boxed{80}$	0	-4/10	1	11200
z	1	-15	0	9/10	0	18000

Selectleastratio

Solution/pivotelements

200/(1/2) =400

Select the most negative value in row z.

Pivot element	:intersectionofpivotrowandpivotcolumn:80
basic variables	: y, $s_{2,z}$
Non basic variables	:x,s_1
enter variable	x
Leavevariable	: s_2

Secondsolutionat(x,y,s_1,s_2)=(0,200,0,11200) second solution z = 18000 (improved solution)

Rowx

Newpivotrow=currentpivotrow/pivot element

$=(0,80,0,\frac{-4}{10},1,11200)/80$

$=(0,1, 0, \frac{-1}{200}, \frac{1}{80},140)$

Newrow=currentrow–(itspivotcolumncoefficient)*newpivotrow

=(0, 1/2, 1, 1/100, 0,200)-($\frac{1}{2}$)*(0,1,0, $\frac{-1}{200}$, $\frac{1}{80}$,140)

=(0, 0, 1, $\frac{1}{80}$, $\frac{-1}{160}$, 130)

<u>Rowz</u>

Newrow=currentrow–(itspivotcolumncoefficient)*newpivotrow

=(1, -15, 0, $\frac{9}{10}$,0, 18000)-(-15)*(0,1,0, $\frac{-1}{200}$, $\frac{1}{80}$, 140)

=(1, -15, 0, $\frac{9}{10}$,0, 18000)+(15)*(0,1,0, $\frac{-1}{200}$, $\frac{1}{80}$,140)

=(1, 0, 0, $\frac{33}{40}$, $\frac{15}{80}$,20100)

Iterationiii

Basic	Z	X	Y	S_1	S_2	Solution
Y	0	0	1	1/80	-1/160	130
X	0	1	0	-1/200	1/80	140
Z	1	0	0	33/40	15/80	20100

Theabovetablehasnonegativevaluesinrowz. Therefore, the above table is optimum table.

Profitat(140,130)=60c +90s=60(140) +90(130)= $20,100

Final solution at (x, y, s_1,s_2) = (130, 140, 0, 0)

finalsolutionz=$20,100(optimizedsolution)

Primaltodualconversion(dualto primal)

[primal=dualofdual]

Primal

Maximize

$$z=\sum_{J=1}^{n} c_j x_j,$$

Subjectto:

$$\sum_{J=1}^{n} a_{ij}x_j \quad \le b_i \qquad (i=1,2,...,m),$$

$$x_j \ge 0 \qquad (j=1,2,...,n).$$

Dual

Minimize

$$z'=\sum_{i=1}^{m} b_i y_i,$$

Subjectto:

$$\sum_{i=1}^{m} a_{ij} y_i \quad \ge c_j \qquad (j=1,2,...,n),$$

$$y_i \ge 0 \qquad (i=1,2,...,m).$$

Theprimalproblem

Minimize $4x_1+2x_2-x_3$

Subjectto $x_1+x_2+ 2x_3\geq 3$

$2x_1- 2x_2+4x_3\leq 5$

$X_1,x_2,x_3\geq 0.$

Thedual problem

Maximize $3y_1 + 5y2$

subjectto $y_1+2y_2 \leq 4$

$Y_1 - 2y_2 \leq 2$

$2y_1+4y_2\leq -1$ y_1

≥ 0, $y_2 \geq 0$

Themaximum-flowproblem

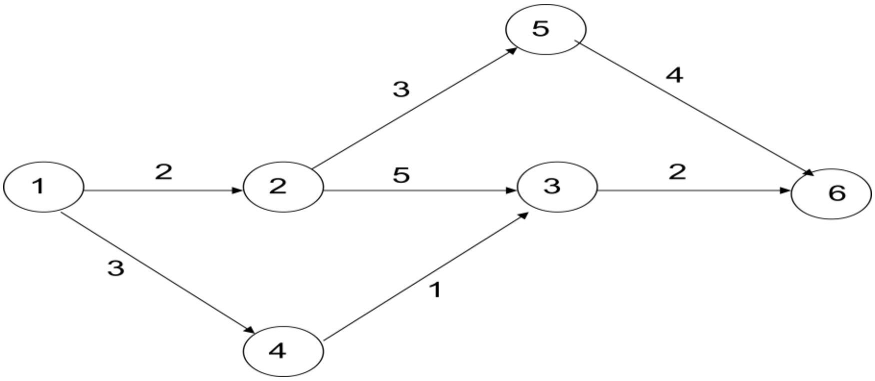
Maximumflowproblem

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formallyrepresentedbyaconnectedweighteddigraphwith*n*verticesnumberedfrom1to*n* Withthefollowingproperties:

- Containsexactlyonevertexwithnoenteringedges,calledthe***source***(numbered 1)
- Containsexactlyonevertexwithnoleavingedges,calledthe***sink***(numbered*n*)
- Has positive integer weight u_{ij}on each directed edge (*i.j*),called the***edge capacity,***indicating the upper bound on the amount of the material that can be sent from *i* to *j* through this edge.
- Adigraphsatisfyingthesepropertiesiscalleda**flownetwork**orsimplya network.

Exampleofflownetwork



Node(1)=source

node(6) = sink

Definitionofa flow

A *flow* is an assignment of real numbers $x_{ij}$to edges (*i,j*) of a given network that satisfy the following:

- *Flow-conservationrequirements*

  Thetotalamountofmaterialenteringanintermediatevertexmustbeequaltothetotal amount of the materialleaving the vertex

- *Capacityconstraints*

  $0\leq x_{ij}\leq u_{ij}$foreveryedge(*i,j*) $\in e$

Flowvalueandmaximumflow problem

Sinceno material can be lost oradded to bygoing through intermediatevertices ofthenetwork, the total amount of the material leaving the source must end up at the sink:

$$\sum x_{1j}=\sum x_{jn}$$

$$J:(1,j)\epsilon e j:(j,n)\epsilon e$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into the sink). The*maximumflow problem* is to findaflowofthelargest value(maximum flow)fora given network.

Maximum-flowproblemaslpproblem

Maximize$v=\sum x_{1j}$

$J:(1,j)\in e$

Subjectto

$\sum x_{ji}$- $\sum x_{ij}$ $=0$ for$i$ $=2, 3,\ldots,n$-1

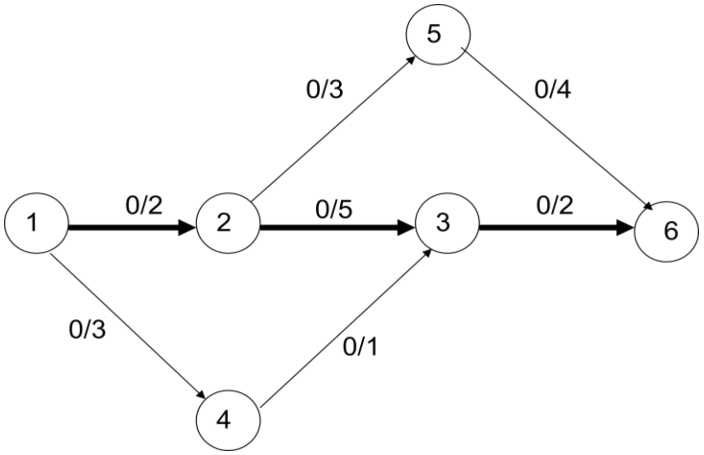
$J:(j,i)\in e$ $j:(i,j)\in e$

$0\leq x_{ij}\leq u_{ij}$foreveryedge $(i,j)\in e$

Augmentingpath(ford-fulkerson)method

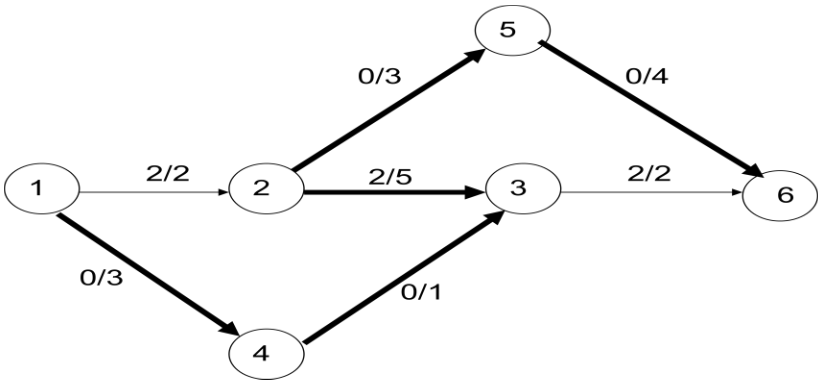
- Startwiththezeroflow(x_{ij}=0foreveryedge).
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent.
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again.
- If noflow-augmentingpathisfound,thecurrent flowismaximum.

Example1



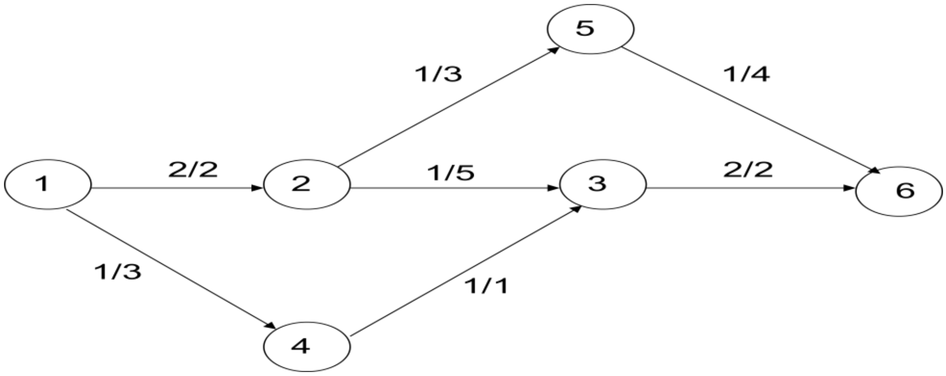
Augmentingpath: 1→2→3 →6

X_{ij}/u_{ij}



Augmentingpath: 1 →4→3←2 →5 →6

Example1(maximumflow)



Findingaflow-augmentingpath

To find a flow-augmenting path for a flow x, consider paths from source to sink in the underlying <u>undirected</u> graph in which any two consecutive vertices *i*,*j* are either:

- Connectedbyadirectededge(*i* to*j*)with somepositiveunused capacityr_{ij}=u_{ij}–x_{ij}
  - Knownas*forwardedge*(→)

Or

- Connectedbyadirected edge (*j*to *i*)with positiveflow x_{ji}
  - Knownas*backwardedge*(←)

Ifaflow-augmentingpathisfound,the currentflow canbe increasedby*r*unitsbyincreasing $x_{ij}$by *R* on each forward edge and decreasing x_{ji} by *r* on each backward edge, where

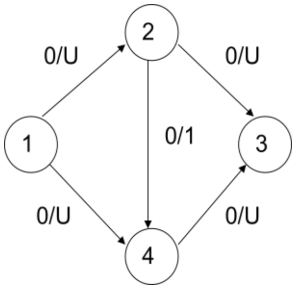
R=min{r_{ij}onallforwardedges,x_{ji}onallbackward edges}

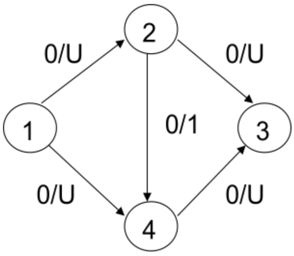
- Assumingtheedgecapacities areintegers,*r*is apositiveinteger
- Oneachiteration,theflowvalueincreasesbyatleast1
- Maximumvalueisboundedbythesumofthecapacitiesoftheedgesleavingthesource; hence the augmenting-path method has to stop after a finite number of iterations
- Thefinalflowisalwaysmaximum,itsvaluedoesn’tdependonasequenceof Augmentingpathsused

Performancedegenerationofthemethod

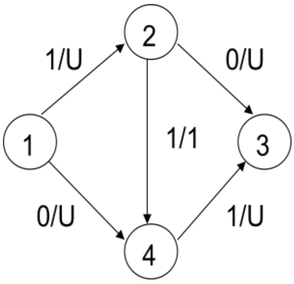
- Theaugmenting-pathmethoddoesn’tprescribeaspecificwayforgeneratingflow-augmenting paths
- Selectingabadsequence ofaugmentingpathscouldimpactthemethod’s efficiency

Example2

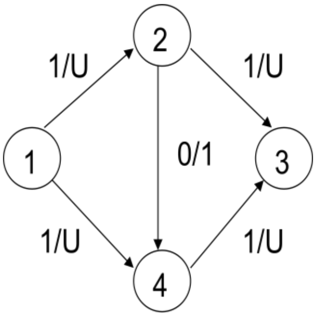




1→2→4→3

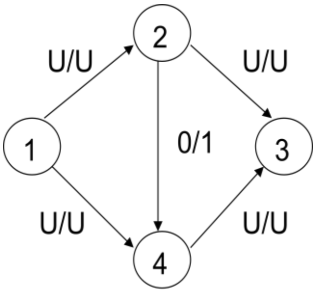


1→4←2→3 **v=1**



V=2

…



V=2u

Requires2uiterationstoreachmaximumflowofvalue 2u

Shortest-augmenting-pathalgorithm

Generateaugmentingpathwiththeleastnumberofedgesbybfsasfollows.

Startingatthesource,performbfstraversalbymarkingnew(unlabeled)verticeswithtwo labels:

- First label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled
- Secondlabel –indicatesthevertex fromwhichthe vertex beinglabeledwas reached, with "+" or"–"added to the second label to indicate whether the vertex was reached via a forward or backward edge

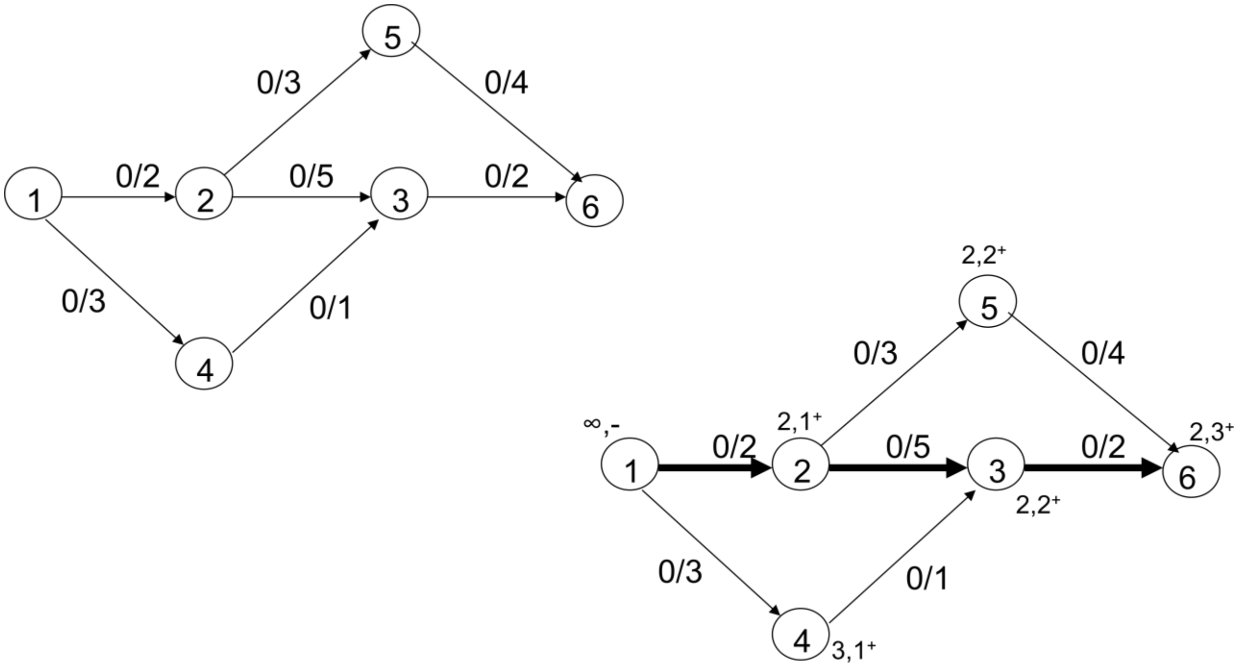
Vertexlabeling

- Thesourceisalwayslabeledwith∞,-
- Allotherverticesarelabeledas follows:
  - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from i to j with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$(forward edge), vertex j is labeled with l_j, i^+, where $l_j = \min\{l_i, r_{ij}\}$

- If unlabeled vertex *j* is connected to the front vertex *i* of the traversal queue by a directed edge from *j* to *i* with positive flow x_{ji}(backward edge), vertex *j* is labeled l_j, i^-, where l_j= min{l_i, x_{ji}}

- Ifthesinkendsupbeinglabeled,thecurrentflowcanbeaugmentedbytheamount Indicatedbythe sink'sfirst label.
- The augmentation of the current flow is performed along the augmenting path traced by following the vertex second labels from sink to source; the current flow quantities are increased on the forward edges and decreased on the backward edges of this path.
- If thesinkremainsunlabeledafterthetraversalqueuebecomesempty,the algorithmreturns the current flow as maximum and stops.

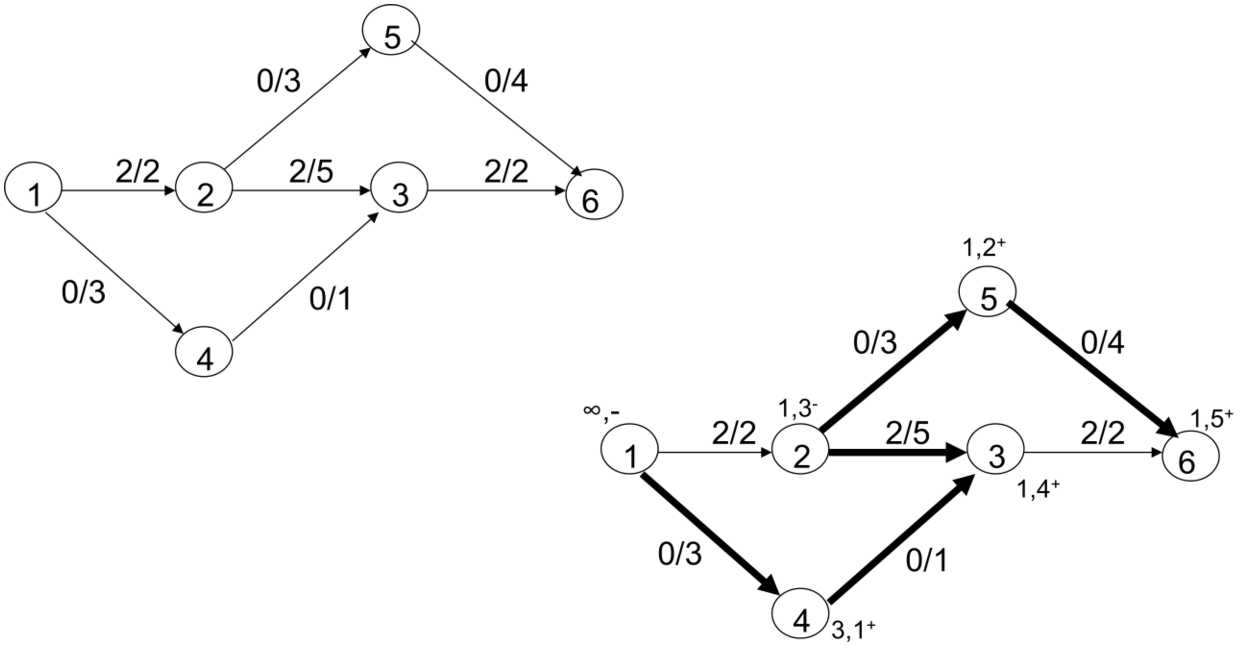
Example:shortest-augmenting-pathalgorithm



Queue:124356

↑↑↑↑

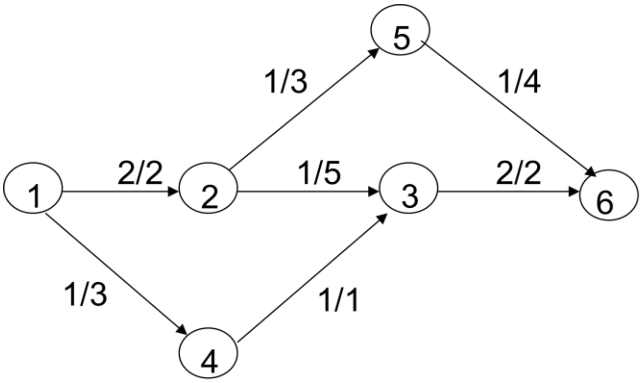
Augmentthe flowby2(thesink'sfirstlabel) alongthepath 1→2→3→6

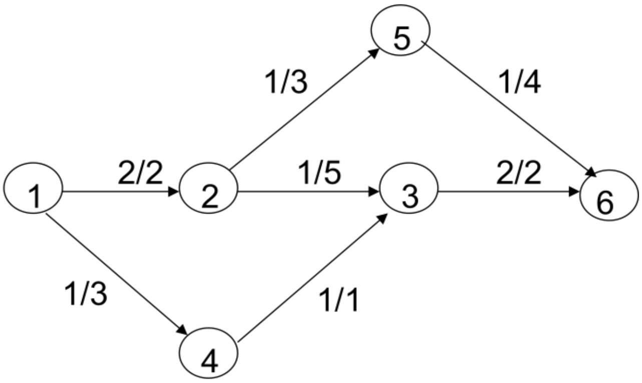


Queue:143256

↑↑↑↑↑

Augmentthe flowby1(thesink’sfirstlabel) alongthepath 1→4→3←2→5→6





Queue:14

↑↑

Noaugmentingpath(the sinkisunlabeled)thecurrentflowismaximum

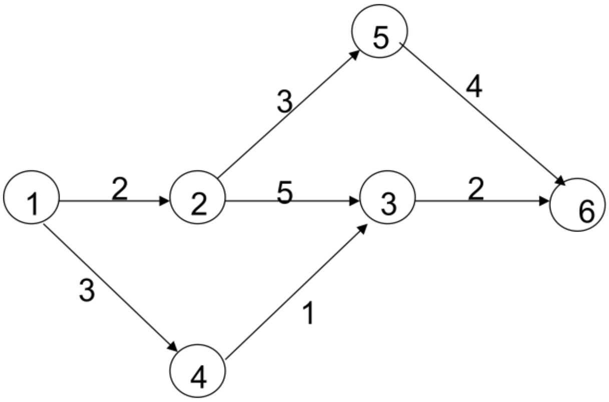
Definitionofacut

Let x be a set of vertices in a network that includes its source but does not include its sink, and let x, the complement of x, be the rest of the vertices including the sink.the *cut* induced by this partition of the vertices is the set of all the edges with a tail in x and a head in x.

*Capacityofa cut*isdefined asthesum ofcapacitiesof theedgesthat composethecut.

- →e’lldenoteacutanditscapacitybyc(x,x)and c(x,x)
- Notethatifalltheedgesofacutweredeletedfromthenetwork,therewouldbeno directed path from source to sink
- *Minimumcut*isacut of thesmallestcapacityin a given network

Examplesofnetwork cuts



Ifx={1}andx={2,3,4,5,6},c(x,x)={(1,2), (1,4)},c=5

Ifx={1,2,3,4,5} andx={6},c(x,x)={(3,6),(5,6)},c=6

Ifx={1,2,4}andx={3,5,6},c(x,x)={(2,3),(2,5),(4,3)},c =9

Max-flowmin-cuttheorem

1. Thevalue ofmaximum flowin anetworkisequal to the capacityof its minimum cut
2. Theshortestaugmentingpathalgorithmyieldsboth amaximum flowand aminimum cut:
    - Maximumflow isthe final flow produced bythealgorithm
    - Minimumcutisformedbyalltheedgesfromthelabeledverticestounlabeled vertices on the last iteration of the algorithm.
    - All the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them.

```
Algorithmshortestaugmentingpath(g)
    //implementstheshortest-augmenting-path algorithm
    //input:anetworkwithsinglesource1,singlesink$n$,andpositiveintegercapacities$uij$on
    //        itsedges $(i,j)$
    //output:amaximumflow$x$
    Assign$xij$=0to everyedge$(i,j)$inthenetwork
    Labelthesourcewith∞,−and add thesourceto theemptyqueue$q$
    Whilenot$empty(q)$do
        $I \leftarrow front(q)$;$dequeue(q)$
        Foreveryedgefrom$i$to$j$do//forwardedges
            If $j$ is unlabeled
                $R_{ij} \leftarrow u_{ij} - x_{ij}$
                If$r_{ij}>0$
                    $Lj \leftarrow \min\{l_i, r_{ij}\}$;label$j$with$l_j, i+$
                    $Enqueue(q,j)$
        Foreveryedgefrom$j$to$i$do//backwardedges
            If $j$ is unlabeled
                 If$x_{ji}>0$
                    $L_j \leftarrow \min\{l_i, x_{ji}\}$;label$j$with$l_j, i-$
                    $enqueue(q, j)$
        Ifthesinkhasbeen labeled
            //augmentalongtheaugmentingpath found
            $J \leftarrow n$//startatthesink andmovebackwardsusingsecond labels
            While$j \neq 1$//thesourcehasn'tbeen reached
                Ifthesecondlabelofvertex$j$is$i+$
                    $x_{ij} \leftarrow x_{ij} + l_n$
                Else//thesecondlabelofvertex$j$is$i-$ $x_{ij} \leftarrow x_{ij}$
                    $-l_n$
                $J \leftarrow i$;$i \leftarrow$thevertexindicatedby$i$'ssecondlabel erase
            all vertex labels except the ones of the source reinitialize
            $q$ with the source
        Return$x$//thecurrentflowismaximum
```

Timeefficiency

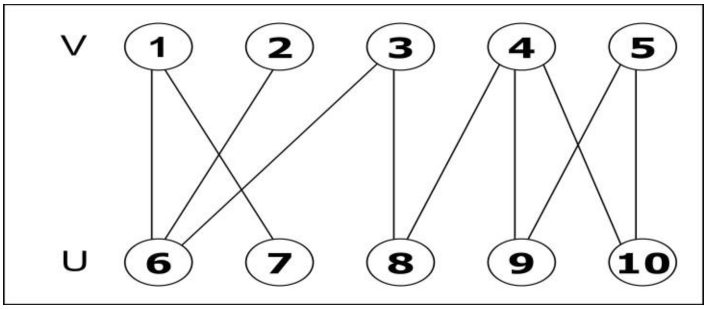
- The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds *nm*/2, where *n* and *m* are the number of vertices and edges, respectively.
- Since the time required to find shortest augmenting path by breadth-first search is in o(*n*+*m*)=o(*m*) for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in o(nm^2) for this representation.
- Moreefficientalgorithmshavebeenfoundthatcanrunincloseto o(*nm*)time,butthese algorithms don't fall into the iterative-improvement paradigm.

Maximummatchinginbipartitegraphs

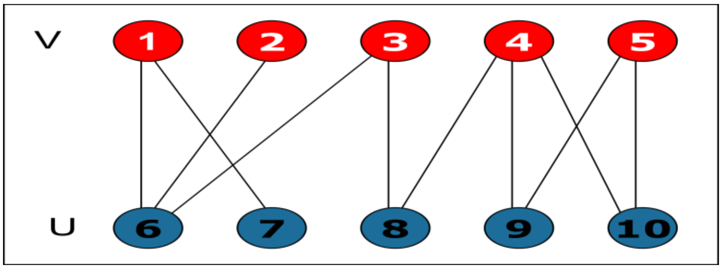
Bipartite graphs

Bipartitegraph:agraphwhoseverticescanbepartitionedintotwodisjointsetsvandu,not necessarily of the same size, so that every edge connects a vertex in v to a vertex in u.

Agraphis bipartiteif andonlyifit doesnot haveacycleof anodd length.



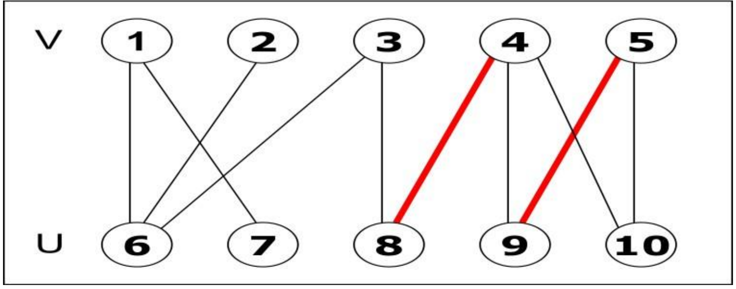
A bipartite graph is *2-colorable*: the vertices can be colored in two colors so that every edgehas its vertices colored differently



Matchinginagraph

A*matching*inagraphisasubsetofitsedgeswiththepropertythatnotwoedgessharea vertex

Amatchinginthis graphm ={(4,8), (5,9)}



A*maximum*(or*maximumcardinality*)*matching*isamatchingwiththelargestnumberofedges

- Alwaysexists
- Notalwaysunique

Freeverticesandmaximummatching

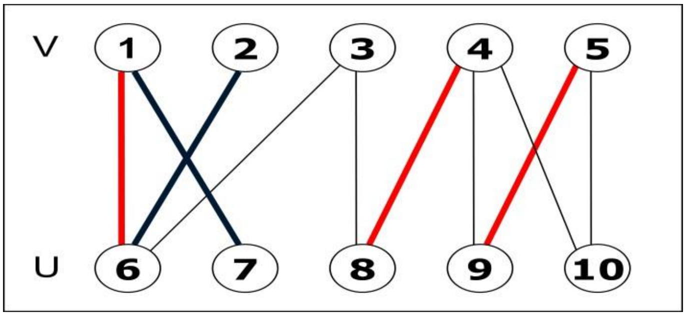
For a given matching m, a vertex is called *free* (or *unmatched*) ifit is not an end point of anyedge in m; otherwise, a vertex is said to be *matched*

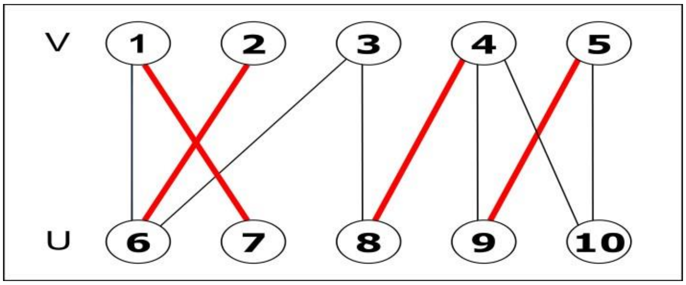
- Ifeveryvertexismatched,thenmisa maximummatching
- Ifthere areunmatched orfreevertices,then mmaybeable to be improved
- Wecanimmediatelyincrease amatchingbyaddinganedge connectingtwo free vertices (e.g., (1,6) above)
- Matchedvertex=4,5, 8,9. Freevertex=1,2, 3,6, 7,10.

Augmentingpathsandaugmentation

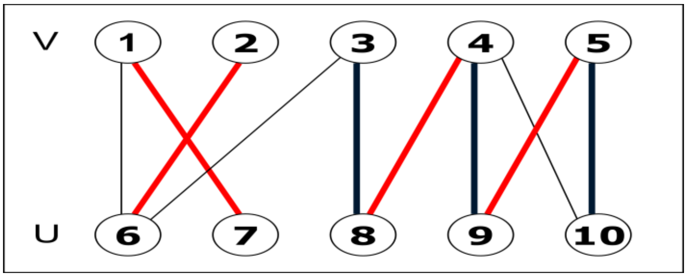
An *augmenting path* for a matching m is a path from a free vertex in v to a free vertex in u whose edges alternate between edges not in m and edges in m

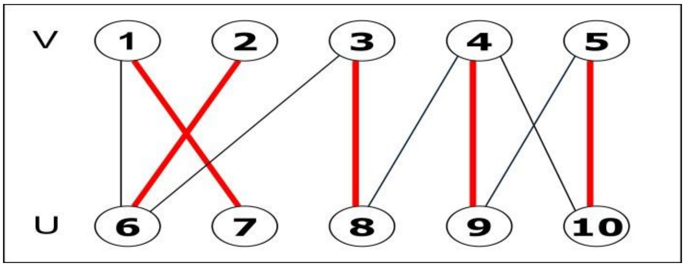
- Thelengthofanaugmentingpathisalwaysodd
- Addingtomtheoddnumberedpathedgesanddeletingfromittheevennumberedpath edges increases the matching size by 1 (*augmentation*)
- One-edgepathbetweentwofreeverticesisspecialcaseofaugmenting path





Augmentationalongpath 2,6,1,7





Augmentationalong3,8,4,9,5,10

Matchingontherightismaximum(*perfect* matching).

<u>Theorem:</u>amatchingmismaximumifandonlyifthereexistsnoaugmentingpathwith respect to m.

Augmentingpathmethod(template)

- Startwithsomeinitial matching.e.g., theemptyset
- Findanaugmentingpathandaugmentthecurrentmatchingalongthatpath.e.g.,using breadth-first search like method
- Whennoaugmenting pathcanbefound,terminateandreturnthelastmatching,whichis maximum

Thestablemarriageproblem.

Stablemarriageproblem

- There is a set y = {m_1,…,m_n} of *n* men and a set x = {w_1,…,w_n} of *n* women.each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).
- A*marriagematching*misasetof*n*pairs(m_i,w_j).
- A pair (*m*, *w*) is said to be a *blocking pair* for matching m if man *m* and woman *w* are not matched in m but prefer each other to their mates in m.
- Amarriagematchingmiscalled*stable*ifthereisnoblockingpairforit;otherwise,it's Called*unstable.*
- The*stablemarriageproblem*istofindastablemarriagematchingformen'sandwomen's Givenpreferences.

Instanceofthestablemarriageproblem

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

<u>Men'spreferences</u> **<u>women'spreferences</u>**

1st 2nd 3rd 1st 2nd

3rdbob:lea ann sue ann:jim tom

bob jim: lea sue ann lea:tom

bob jim tom: sue lea ann

sue:jim tom bob

<u>Rankingmatrix</u>

	Ann	**Lea**	**Sue**
Bob	$\boxed{2,3}$	**1,2**	**3,3**
Jim	**3,1**	$\boxed{1,3}$	**2,1**
Tom	**3,2**	**2,1**	$\boxed{1,2}$

Dataforaninstanceofthestablemarriageproblem.(a)men'spreferencelists;(b)women'spreferencelists.(c) ranking Matrix(withtheboxedcellscomposinganunstablematching).

Stablemarriagealgorithm(gale-shapley)

Step0startwithall themenand women being free

Step1while therearefreemen, arbitrarilyselectoneof themand do the following:

- *Proposal*the selected free man *m* proposes to *w*, the next woman on his preference list
- *Response*if *w* is free, she accepts the proposal to be matched with *m*.if she is not free, shecompares *m*with hercurrent mate.if sheprefers *m* to him, sheaccepts *m*'s proposal, making her former mate free; otherwise, she simply rejects *m*'s proposal, leaving *m* free

Step2returnthe setof*n* matchedpairs

Example

Freemen:bob,jim,tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bobproposedtolea,leaacceptedbob free

men: jim, tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jimproposedtolea,learejected free

men: jim, tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	2,1	1,2

Jimproposedtosue,sueaccepted free

men: tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	2,1	1,2

Tomproposedtosue,suerejected free

men: tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	**2,1**	1,2

Tomproposedtolea,leareplacedbobwithtom free

men: bob

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bobproposedtoann,annaccepted

Anacceptedproposalisindicatedbyaboxedcell;arejectedproposalisshownbyan underlined cell.

Analysisofthegale-shapleyalgorithm

- Thealgorithmterminatesafternomorethann^2iterationswith a stable marriage output.
- The stable matching produced by the algorithm is always *man-optimal*: each man gets the highest rank woman on his list under any stable marriage.one can obtain the *woman-optimal* matching by making women propose to men.
- Aman(woman)optimalmatchingisuniqueforagivensetofparticipantpreferences.
- The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training.

Unitv-copingwiththelimitationsofalgorithm power

Limitationsofalgorithm power

There are many algorithms for solving a variety of different problems. They are very powerful instruments, especially when they are executed by modern computers.

Thepowerofalgorithmsislimitedbecauseofthefollowingreasons:

- Therearesomeproblemscannot be solved byany algorithm.
- Therearesomeproblemscan besolvedalgorithmicallybutnot inpolynomial time.
- There are some problems can be solved in polynomial time bysome algorithms, but they are usually lower bounds on their efficiency.

Algorithmslimitsareidentifiedbythefollowing:

- Lower-boundarguments
- Decisiontrees
- P,npandnp-completeproblems

Lower-boundarguments

We can look at the efficiency of an algorithm two ways. We can establish its **asymptotic efficiency class** (say, for the worst case) and see where this class stands with respect to the **hierarchy of efficiency classes**.

For example, selection sort, whose efficiency is quadratic, is a reasonably fast algorithm, whereas the algorithm for the tower of hanoi problem is very slow because its efficiency is exponential.

Lower bounds means estimating the minimum amount of work needed to solve the problem. We present several methods for establishing lower bounds and illustrate them with specific examples.

1. Triviallowerbounds
2. Information-theoreticarguments
3. Adversaryarguments
4. Problemreduction

In analyzing the efficiency of specific algorithms in the preceding, we should distinguish between a lower-bound class and a minimum number of times a particular operation needs to be executed.

Triviallowerbounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's **input** that must be **processed** and the number of **output** items that need to be **produced**.

Since any algorithm must at least "read" all the items it needs to process and "write" all its outputs, such a count yields a **trivial lower bound**.

For example, any algorithm for generating all permutations of n distinct items must be in Ω(n!) Because the size of the output is n!. And this bound is **tight** because good algorithms for generating permutations spend a constant time on each of them except the initial one.

Consider the problem of **evaluating a polynomial of degree n** at a given point x, given its coefficients $a_n, a_{n-1}, . . . , a_0$. $P(x) = a_nx^n + a_{n-1}x^{n-1} + . . . + a_0$. All the coefficients have to be processed by any polynomial-evaluation algorithm. I.e **Ω(n).** This is tight lower bound.

Similarly, a trivial lower bound for computing **the product of two n × n matrices** is **$\Omega(n^2)$** because any such algorithm has to process $2n^2$ elements in the input matrices and generate n^2 elements of the product. It is still unknown, however, whether this bound is tight.

Thetrivialboundforthe **travelingsalesmanproblem**is$\Omega(n^2)$,becauseitsinputis*n(n-1)/2* intercitydistances andits outputisalistofn +1 citiesmakingup anoptimaltour.butthisboundis useless because there is no known algorithm with the running time being a polynomial function.

Determining the lower bound lies in **which part of an input must be processed** by any algorithm solving the problem. For example, searching for an element of a given value in a sorted array does not require processing all its elements.

Information-theoreticarguments

The information-theoretical approach seeks to establish a lower bound based on**theamount of information it has to produce** by algorithm.

Consider an example "**game of guessing number**", the well-known game of deducing a positive integer between 1 and n selected bysomebodybyasking that person questions with yes/no answers. The amount of uncertainty that any algorithm solving this problem has to resolve can be measured by]$\log_2$ n].

The number of bits needed to specify a particular number among the n possibilities. Each answer to the question gives information about each bit.

1. Isthe firstbitzero?→ no→firstbitis1
2. Isthesecondbitzero?→yes→secondbitis0
3. Isthethirdbit zero?→yes→ thirdbitis0
4. Isthe forthbitzero? →yes→forthbitis0

Thenumberinbinaryis 1000,i.e.8indecimalvalue.

Theabove approach is called the ***information-theoreticargument*** because ofits connection to information theory. This is useful for finding ***information-theoretic lower bounds*** for many problems involving comparisons, including sorting and searching.

Itsunderlyingideacanberealizedthemechanismof ***decisiontrees***. Because

Adversaryarguments

Adversaryargument is a method of proving by **playing a role of adversary (opponent)** in which algorithm has to work more for **adjusting input** consistently.

Consider the game of guessingnumber between positive integer 1 and n byaskinga person (adversary) with yes/no type answers for questions. After each question at least one-half of the numbers reduced. If an algorithm stops before the size of the set is reduced to 1, the adversary can exhibit a number.

Any algorithm needs]$\log_2$ n] iterations to shrink an n-element set to a one-element set by halving and rounding up the size of the remaining set. Hence, at least]$\log_2$ n] questions need to be asked by any algorithm in the worst case. This example illustrates the ***adversary method*** for establishing lower bounds.

Considertheproblemof**mergingtwosortedlists**ofsizena$_1$<a$_2$< ...<a$_n$andb$_1$<b$_2$<. . .<b$_n$intoasinglesortedlistofsize2n.forsimplicity,weassumethatallthea'sandb'sare Distinct,whichgivestheproblemaunique solution.

Merging is done by repeatedly comparing the first elements in the remaining lists and outputting the smaller among them. The number of key comparisons (lower bound) in the worst case for this algorithm for merging is **2n − 1**.

Problemreduction

Problem reduction is a method in which a difficult unsolvable problem p is reduced to another solvable problem b which can be solved by a known algorithm.

A similar reduction idea can be used for finding a lower bound. To show that problem p isat least as hard as another problem q with a known lower bound, we need to reduce q to p (not pto q!). In other words, we should show that an arbitrary instance of problem q can be transformed to an instanceofproblem p, so anyalgorithm solvingp would solveqas well. Then alowerbound forqwillbealowerboundforp.table5.1lists several importantproblemsthatareoften usedfor this purpose.

Table5.1problemsoftenusedforestablishinglowerboundsbyproblemreduction

Problem	Lowerbound	Tightness
Sorting	Ω(n logn)	Yes
Searchinginasortedarray	Ω(logn)	Yes
Elementuniquenessproblem	Ω(n logn)	Yes
Multiplicationofn-digitintegers	Ω(n)	Unknown
Multiplicationofn×n matrices	Ω(n2)	Unknown

Consider the euclidean minimum spanning tree problem as an example of establishing a lower bound by reduction:

Given n points in the cartesian plane, construct a tree of minimum total length whose vertices are the given points. As a problem with a known lower bound, we use the element uniqueness problem.

We can transform any set x1, x2, . . . , xn of n real numbers into a set of n points in the cartesian plane by simply adding 0 as the points' y coordinate: (x1, 0), (x2, 0), . . . , (xn, 0). Let t be a minimum spanning tree found for this set of points. Since t must contain a shortest edge, checking whether t contains a zero length edge will answer the question about uniqueness of the given numbers. This reduction implies that **Ω** (n log n) is a lower bound for the euclideanminimum spanning tree problem,

Note: limitationsofalgorithmcanbestudiedbyobtaininglowerboundefficiency.

Decisiontrees

Important algorithms like sorting and searching are based on comparing items of their inputs. The study of the performance of such algorithm is called a **decision tree**. As an example, figure 5.1 presents a decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node.

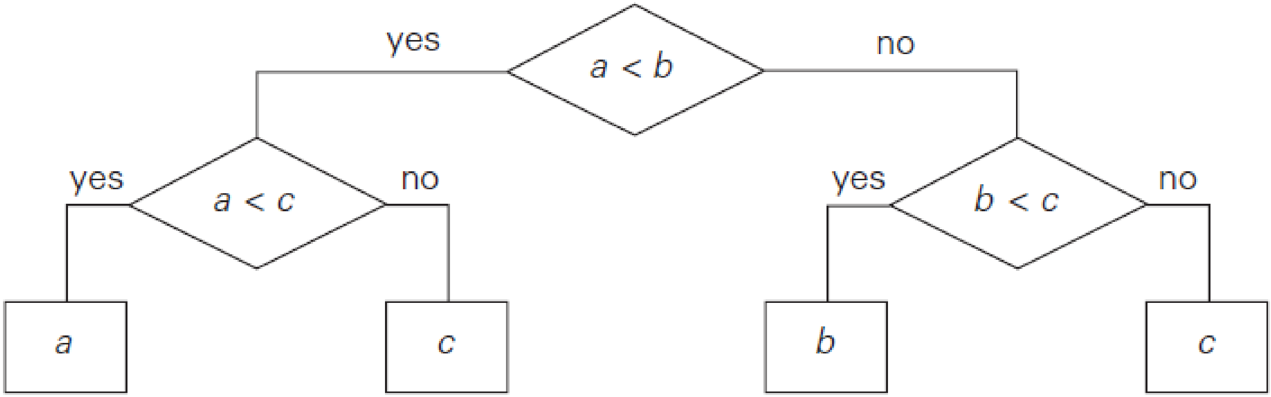
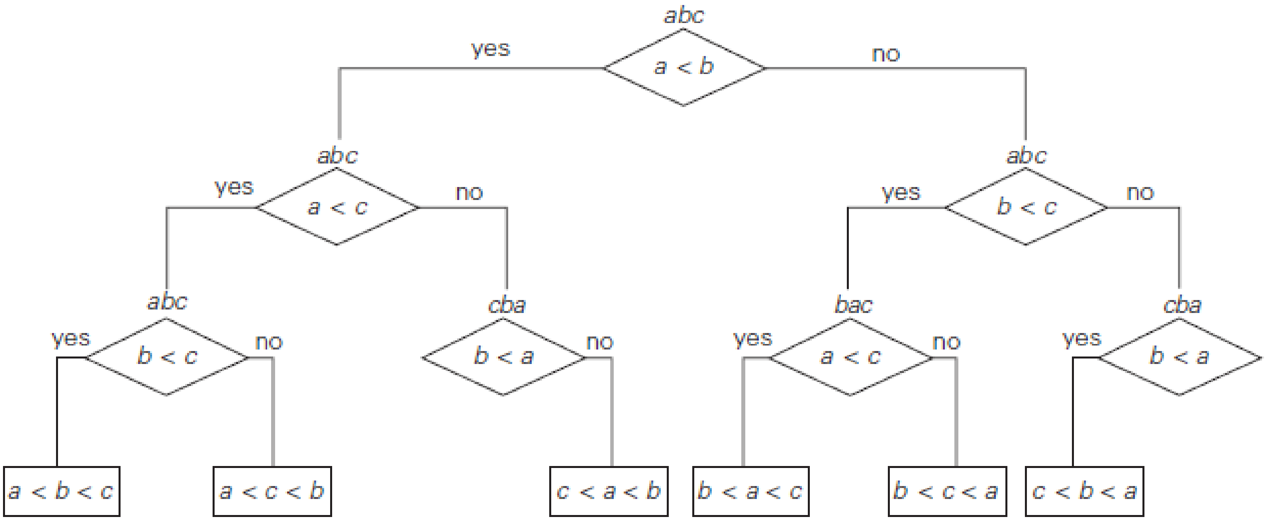


Figure5.1decisiontreeforfindingaminimumofthreenumbers.

Considerabinarydecisiontreewithheighthandleavesn.andheighth, thenh≥]log$_2$n]. A binary tree of height h with the largest number of leaves on the last level is 2^h. In other words, $2^h \geq n$, which puts a lower bound on the heights of binary decision trees. Hence the worst-case number of comparisons made by any comparison-based algorithm for the problem is called the information theoretic lower bound.

Decisiontreesforsorting



Cba 1

2 3

Figure5.2decisiontreeforthetree-elementselectionsort.

A triple above a node indicates the state of the array being sorted. Note two redundant comparisons b <a with a single possible outcome because of the results of some previously made comparisons.

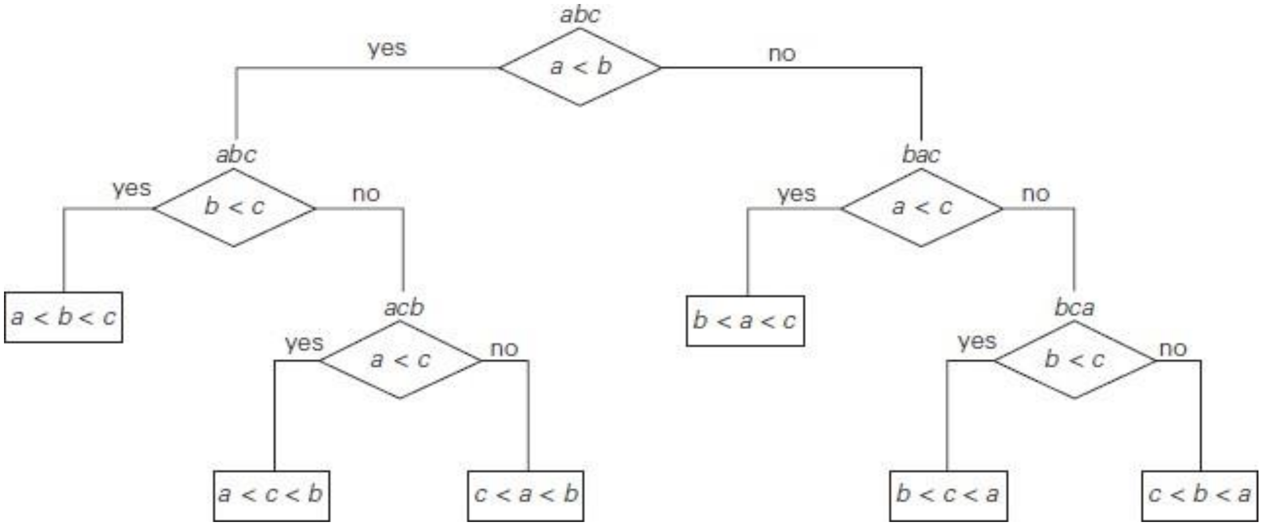


Figure5.3decisiontreeforthethree-element insertionsort.

The three-element insertion sort whose decision tree is given in figure 5.3, this number is(2 + 3 + 3 + 2 + 3 + 3)/6 = 2.66. Under the standard assumption that all n! Outcomes of sorting are equally likely, the following lower bound on the average number of comparisons c_{avg} made byany comparison-based algorithm in sorting an n-element list has been proved:

$$C_{avg}(n) \geq \log 2n!.$$

Decisiontreeisaconvenientmodelofalgorithmsinvolvingcomparisonsinwhich

- Internalnodesrepresent comparisons
- Leavesrepresentoutcomes(orinputcases)

Decisiontreesandsortingalgorithms

- Anycomparison-basedsortingalgorithmcanberepresentedbyadecisiontree(foreach fixed *n*)
- Numberofleaves (outcomes)≥*n*!

- Heightofbinarytreewith$n$!Leaves$\geq\lceil \log_2 n!\rceil$
- Minimum number of comparisons in the worst case$\geq\lceil \log_2 n!\rceil$ for any comparison-basedsortingalgorithm, sincethelongest path representstheworst caseand its length is the height
- $\lceil \log_2 n!\rceil\approx n\log_2 n$(bysterlingapproximation)
- Thislowerboundistight(mergesortorheapsort)

Decisiontreesforsearchingasortedarray

Decision treescanbeusedforestablishinglowerbounds on the number ofkeycomparisons Insearchingasortedarrayofnkeys: a[0]<a[1]<.. .<a[n −1].

Theprincipalalgorithmforthisproblemisbinarysearch. Thenumberofcomparisons made by binary search in the worst case, c_{worst}(n), is given by the formula

$$C_{worst}(n)=L\log_2 n]+1=\lceil \log_2(n+1)\rceil$$

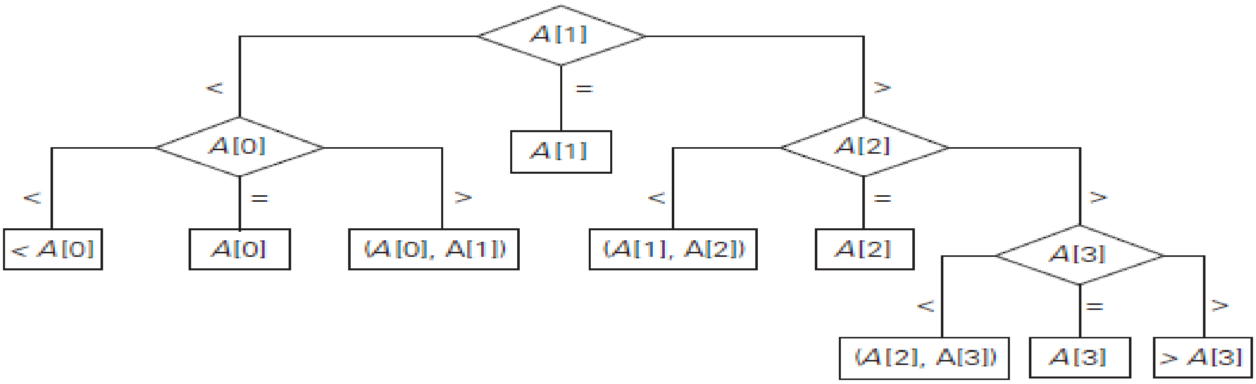


Figure5.4ternarydecisiontreeforbinarysearchinafour-elementarray.

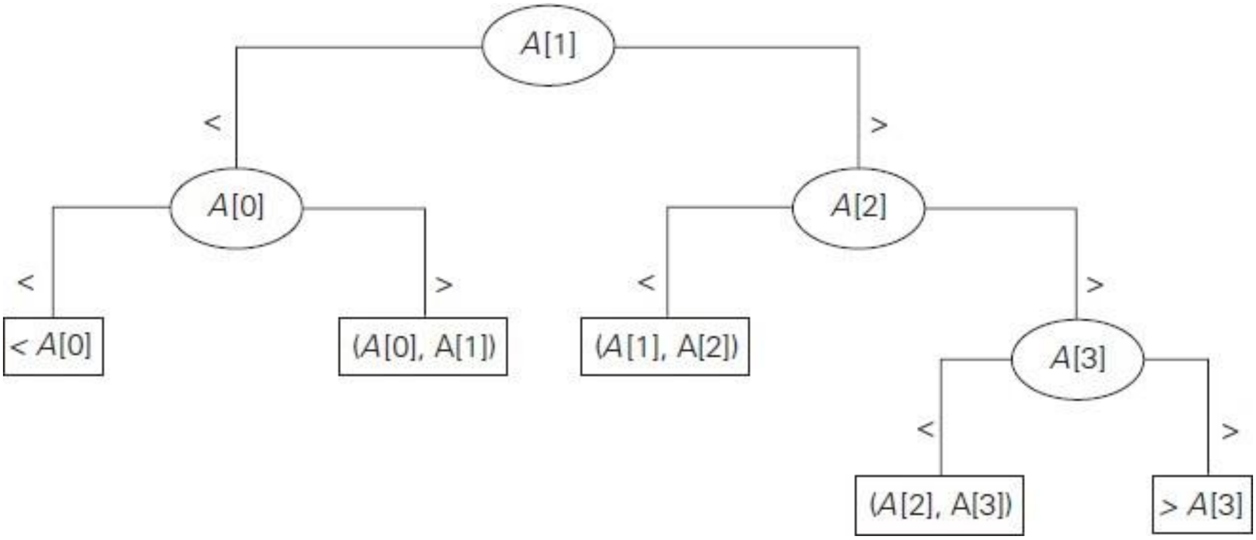


Figure5.5binarydecision treeforbinarysearch inafour-elementarray.

As comparison of the decision trees in the above illustrates, the binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated. Applying inequality tosuch binary decision trees immediately yields $c_{worst}(n)\geq\lceil \log_2(n+1)\rceil$

P,npandnp-completeproblems

Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

Thereareseveral **reasonsforintractability**.

- **First**, we **cannot solve** arbitrary instances of intractable problems in areasonable amount of time unless such **instances are very small**.
- **Second**, although there might be a huge difference between the running times in *o(p(n))*for polynomials of **drastically different degrees**. Where p(n) is a polynomial of the problem's input size n.
- **Third**, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are **always polynomials too**.
- **Fourth**, the choice of this class has led to a development of an extensive theory called ***computational complexity***.

Definition: class *p* is a class of decision problems that can be solved in polynomial time by deterministic algorithms. This class of problems is called ***polynomial class***.

- Problems that can be solved in polynomial time as the set that computer science theoreticians call **p**. A more formal definition includes in p only **decision problems**, which are problems with **yes/no** answers.
- Theclassofdecisionproblemsthataresolvableino(*p*(*n*))**polynomialtime**,where*p*(*n*)is Apolynomial ofproblem'sinputsize*n*

  Examples:
  - Searching
  - Element uniqueness
  - Graphconnectivity
  - Graphacyclicity
  - Primalitytesting(finallyprovedin 2002)
- **Therestrictionofp**todecision problemscanbejustifiedbythefollowing reasons.
  - First, it is sensible to **exclude problems not solvable in polynomial time** becauseoftheirexponentiallylargeoutput.e.g.,generatingsubsetsofagivenset or all the permutations of n distinct items.
  - Second, **many important problems that are not decision problems** in their most natural formulation can be reduced to a series of decision problems that are easier to study. For example, instead of asking about the minimum number of colors needed to color thevertices ofa graph so that no two adjacent vertices are colored the same color. Coloring of the graph's vertices with no more than m colors for m = 1, 2,(the latter is called the **m-coloring problem**.)
  - So,everydecisionproblemcannotbesolvedinpolynomialtime. Some**decision** problems cannot be solved at all by any algorithm. Such problems are called **undecidable**, as opposed to **decidable** problems that can be solved by an algorithm (**halting problem**).
- **Non polynomial-time algorithm:** there are manyimportant problems, however, for which no polynomial-time algorithm has been found.
  - ***Hamiltonian circuit problem*:**determine whether a given graph has a hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
  - ***Traveling salesman problem*:** find the shortest tourthrough n cities with known positive integer distances between them (find the shortest hamiltonian circuit in a complete graph with positive integer weights).

- ***Knapsack problem*****:** find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.
- ***Partition problem*****:** given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
- ***Bin-packing problem*****:** given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.
- ***Graph-coloringproblem*****:**foragivengraph,finditschromaticnumber,whichis the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.
- ***Integer linear programming problem*****:** find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

Definition: a nondeterministic algorithm is a two-stage procedure that takes as its input an instance i of a decision problem and does the following.

1. **Nondeterministic ("guessing") stage:** an arbitrary string s is generated that can bethought of as a candidate solution to the given instance.
2. **Deterministic ("verification") stage:** a deterministic algorithm takes both i and s as its input and outputs yes if s represents a solution to instance i. (if s is not a solution to instance i , the algorithm either returns no or is allowed not to halt at all.)

Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial.

Definition: class *np* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.

Mostdecisionproblemsareinnp.firstofall,thisclassincludesalltheproblemsinp:

P⊆np

This is true because, if a problem is in p, we can use the deterministic polynomial time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simplyignores string s generated in its nondeterministic ("guessing") stage. But np also contains the hamiltonian circuit problem, the partition problem, decision versions of the traveling salesman, the knapsack, graph coloring, and many hundreds of other difficult combinatorial optimization. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in np.

Note that p = np would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm.

Definition:a decision problem *d*1 is said to be ***polynomially reducible*** to a decision problem *d*2, if there exists a function *t* that transforms instances of *d*1 to instances of *d*2 such that:

1. *T*mapsallyesinstancesof*d*1toyesinstancesof*d*2andallnoinstancesof*d*1tono instances of *d*2.
2. *T* is computable byapolynomial time algorithm.

This definition immediately implies that if a problem d1 is polynomially reducible to some problemd2 that can be solved in polynomial time, then problem d1 can also be solved in polynomial time

Definition:adecisionproblem*d*issaidtobe***np-complete***ifitishardasanyprobleminnp.

1. Itbelongstoclass *np*
2. Everyprobleminn*np*ispolynomiallyreducibleto*d*

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the hamiltonian circuit problem is polynomiallyreducible to the decision version of the traveling salesman problem.

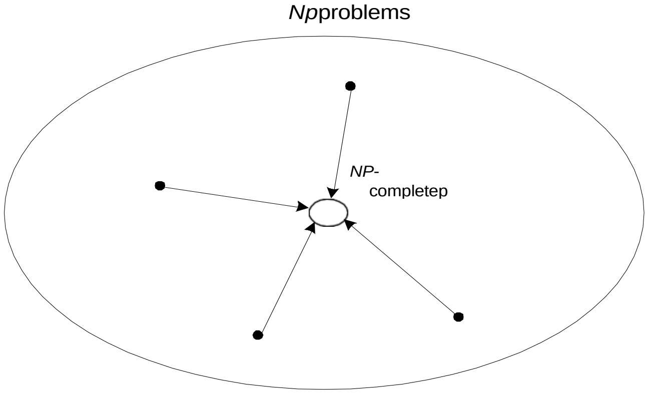


Figure5.6polynomial-timereductionsof*np*problemstoan*np*-complete problem

Theorem:adecisionproblemissaidtobe*np-complete*ifitishardasanyprobleminnp.

Proof: let us prove that the hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

Wecan map a graph gofa given instance ofthe hamiltoniancircuit problem to a complete weighted graph g′representing an instance of the traveling salesman problem byassigning 1 as the weight to each edge in g and adding an edge of weight 2 between any pair of nonadjacent vertices in g. As the upper bound*m* on the hamiltonian circuit length, we take *m* = *n*, where *n* is the number of vertices in g (and g′). Obviously, this transformation can be done in polynomial time.

Let g be a yes instance of the hamiltonian circuit problem. Then g has a hamiltonian circuit, and its image in g′will have length n, making the image a yes instance of the decision traveling salesman problem.

Conversely, if we have a hamiltonian circuit of the length not larger than n in g′, then its length must be exactly n and hence the circuit must be made up of edges present in g, making the inverse image of the yes instance of the decision traveling salesman problem be a yes instance of the hamiltonian circuit problem.

Thiscompletesthe proof.

Theorem: stateandprovecook’stheorem.

Provethatcnf-satis*np*-complete.

Satisfiabilityofboolean formulaforthreeconjuctivenormal formisnp-complete.

Np problems obtained by polynomial-time reductions from a *np*-complete problem

proof: the notion of *np*-completeness requires, however, polynomial reducibility of *all* problems in*np*,bothknownandunknown,totheprobleminquestion.giventhebewilderingvarietyof decision problems, it is nothing short of amazing that specific examples of *np*-complete problems have been actually found.

Nevertheless, this mathematical feat was accomplished independently by stephen cook in the united states and leonid levin in the former soviet union. In his 1971 paper, cook [coo71] showed that the so-called ***cnf-satisfiability problem*** is *np*complete.

x_1	x_2	x_3	$\bar{x}_1$	$\bar{x}_2$	$\bar{x}_3$	$x_1 \vee \bar{x}_2 \vee \bar{x}_3$	$\bar{x}_1 \vee x_2$	$\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$	$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) A (\bar{x}_1 \vee x_2) A (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
T	T	T	F	F	F	T	T	F	F
T	**T**	**F**	F	F	T	T	T	T	**T**
T	F	T	F	T	F	T	F	T	F
T	F	F	F	T	T	T	F	T	F

F	T	T	T	F	F	F	T	T	F
F	T	F	T	F	T	T	T	T	T
F	F	T	T	T	F	T	T	T	T
F	F	F	T	T	T	T	T	T	T

The cnf-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables x_1, x_2, and x_3 and their negations denoted $\bar{x}_1$, $\bar{x}_2$, and $\bar{x}_3$respectively:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\&(\bar{x}_1 \vee x_2)\&(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

The cnf-satisfiability problem asks whether or not one can assign values *true* and *false* to variables of a given boolean expression in its cnf form to make the entire expression *true*. (it is easy to see that this can be done for the above formula: if x_1 = *true*, x_2 = *true*, and x_3 = *false*, the entire expression is *true*.)

Since the cook-levin discovery of the first known *np*-complete problems, computer scientists have found many hundreds, if not thousands, of other examples. In particular, the well-known problems (or their decision versions) mentioned above—hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring—are all *np*-complete. It is known, however, that if *p* != *np* there must exist *np* problems that neither are in *p* nor are *np*-complete.

Showingthatadecisionproblemis *np*-completecanbedoneintwosteps.

1. First, one needs to show that the problem in question is in *np*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents asolution to the problem. Typically, this step is easy.
2. The second step is to show that every problem in np is reducible to the problem in questionin polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known np-complete problem can be transformed to the problem in question in polynomial time.

The definition of *np*-completeness immediately implies that if there exists a deterministic polynomial-time algorithm for just one *np*-complete problem, then every problem in *np* can be solved in polynomial time by a deterministic algorithm, and hence *p* = *np.*

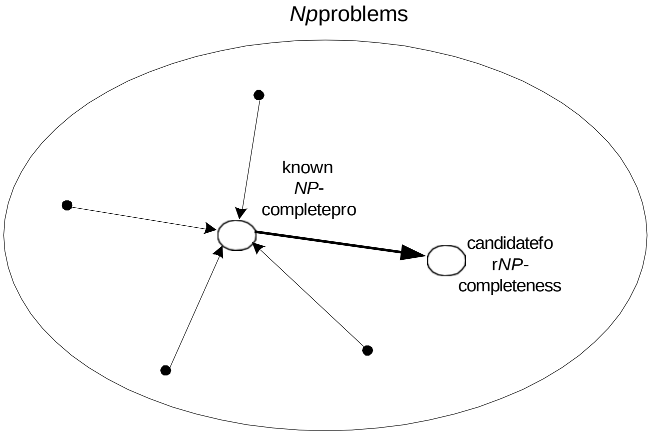


Figure5.7np-completenessbyreduction

Examples: tsp, knapsack, partition, graph-coloring and hundreds of other problems of combinatorial nature p= np would imply that every problem in np, including all np-complete problems, could be solved in polynomial timeif a polynomial-time algorithm for just one np-completeproblemisdiscovered,theneveryprobleminnpcanbesolvedinpolynomialtime,i.e.p
= np most but not all researchers believe that p != np , i.e. P is a proper subset of np. If p != np, then the np-complete problems are not in p, although many of them are very useful in practice.

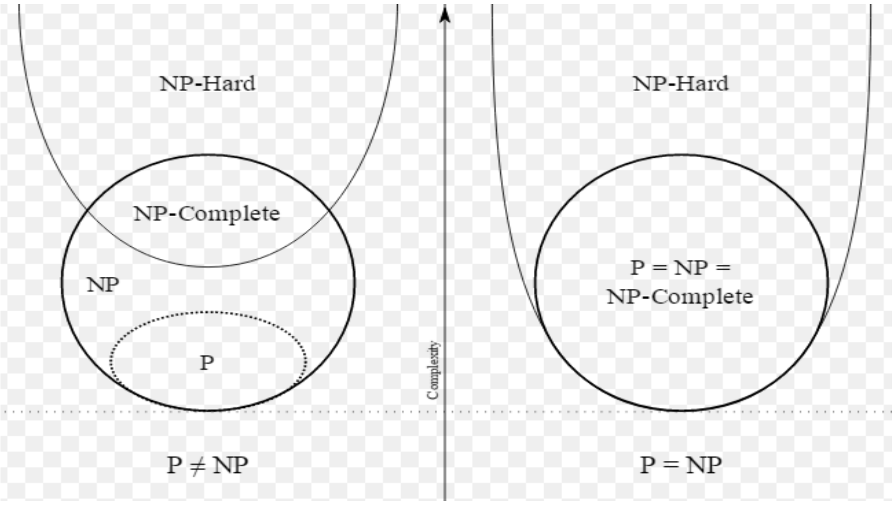


Figure5.8relationamongp,np,np-hardandnpcomplete problems

Copingwiththelimitationsofalgorithm power

Therearesomeproblems thataredifficulttosolvealgorithmically. Atthesametime,fewof them are so important, we must solve by some other technique. Two algorithm design techniques ***backtracking*** and ***branch-and-bound*** that often make it possible to solve at least some large instances of difficult combinatorial problems.

Both backtracking and branch-and-bound are based on the construction of a state-space tree whose nodes reflect specific choices made for a solution's components. Both techniques terminatea node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants

Weconsidera few approximationalgorithmsforsolvingtheassignmentproblem,traveling salesman and knapsack problems. There are three classic methods like the bisection method, the method of false position, and newton's method for approximate root finding.

Exactsolutionstrategiesaregiven below:

***Exhaustivesearch*(bruteforce)-**

- Useful onlyforsmallinstances

Dynamicprogramming

- Applicabletosomeproblems(e.g.,theknapsack problem)

Backtracking

- Eliminatessomeunnecessarycasesfromconsideration
- Yields solutions in reasonable time for many instances but worst case is still exponential

Branch-and-bound

- Furtherrefinesthebacktrackingideaforoptimization problems

Copingwiththelimitationsof algorithmpoweraregivenbelow:

Backtracking

- *N*-queensproblem
- Hamiltoniancircuitproblem
- Subset-sumproblem

Branch-and-bound

- Assignmentproblem
- Knapsackproblem
- Traveling salesman problem

approximationalgorithmsfor*np*-hardproblems

- Approximationalgorithmsforthetravelingsalesman problem
- Approximationalgorithmsfortheknapsackproblem

algorithms for solving nonlinear equations

- Bisectionmethod
- Falsepositionmethod
- Newton'smethod

Backtracking

- Backtrackingisamoreintelligentvariation approach.
- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
- Ifthereisno legitimateoptionforthenextcomponent,noalternatives foranyremaining component need to be considered. In this case, the algorithm backtracks to replace the lastcomponent of the partially constructed solution with its next option.
- It is convenient to implement this kind of processing by constructing a tree of choices being made, called **the state-space tree**.
- Itsrootrepresentsaninitialstatebeforethesearch forasolutionbegins.
- Thenodesofthefirstlevelinthetreerepresentthechoicesmadeforthefirstcomponent of a solution, the nodes of the second level represent the choices for the second component, and so on.
- A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution. Otherwise, it is called ***nonpromising***.
- Leaves represent either nonpromising dead ends or complete solutions found by the algorithm. In the majority of cases, a statespace tree for a backtracking algorithm is constructed in the manner of depthfirst search.
- If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on.
- Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.
- Backtrackingtechniquesareapplied tosolve thefollowingproblems
    - *N*-queensproblem
    - Hamiltoniancircuitproblem
    - Subset-sumproblem

N-queensproblem

The problem is to place *n* queens on an *n* × *n* chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For*n*=1,theproblem has **atrivialsolution**.

Q

Forn= 2,it iseasyto seethat thereis**no solution** to place2queens in 2×2chessboard.

Q	

For*n*= 3,it iseasyto seethat thereis **no solution** to place3queens in 3×3chessboard.

	1	2	3	
1	Q			← Queen1
2			Q	← Queen2
3				

Or

	1	2	3	
1	Q			← Queen1
2				
3		Q		← Queen2

Or

	1	2	3	
1		Q		← Queen1
2				
3	Q			← Queen2

For *n* = 4, there is **solution** to place 4 queens in 4 × 4 chessboard. The four-queens problem solved by the backtracking technique.

Step1:startwiththeemptyboard

	1	2	3	4	
1					←queen1
2					←queen2
3					←queen3
4					←queen4

Step2:placequeen1inthefirstpossiblepositionofitsrow,whichisincolumn1ofrow 1.

	1	2	3	4
1	Q			
2				
3				
4				

Step 3: place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3.

	1	2	3	4
1	Q			
2			Q	
3				
4				

Step4:thisprovestobeadeadendbecausethereisnoacceptablepositionforqueen3.so,the algorithm backtracks and puts queen 2 in the next possible position at (2, 4).

	1	2	3	4
1	Q			
2				Q
3				
4				

Step5:thenqueen3isplacedat(3,2),whichprovestobeanotherdeadend.

	1	2	3	4
1	Q			
2				Q
3		Q		
4				

Step6:thealgorithmthenbacktracksall thewaytoqueen1andmoves itto(1,2).

	1	2	3	4
1		Q		
2				
3				
4				

Step7:thequeen2goesto(2,4).

	1	2	3	4
1		Q		
2				Q
3				
4				

Step8:thequeen3goesto(3,1).

	1	2	3	4
1		Q		
2				Q
3	Q			
4				

Step9:thequeen3goes to(4,3). Thisis asolutionto the problem.

	1	2	3	4
1		Q		
2				Q
3	Q			
4			Q	

Figure5.9solutionfour-queensproblemin4x4 board.

Thestate-spacetreeofthissearchisshowninfigure 12.2

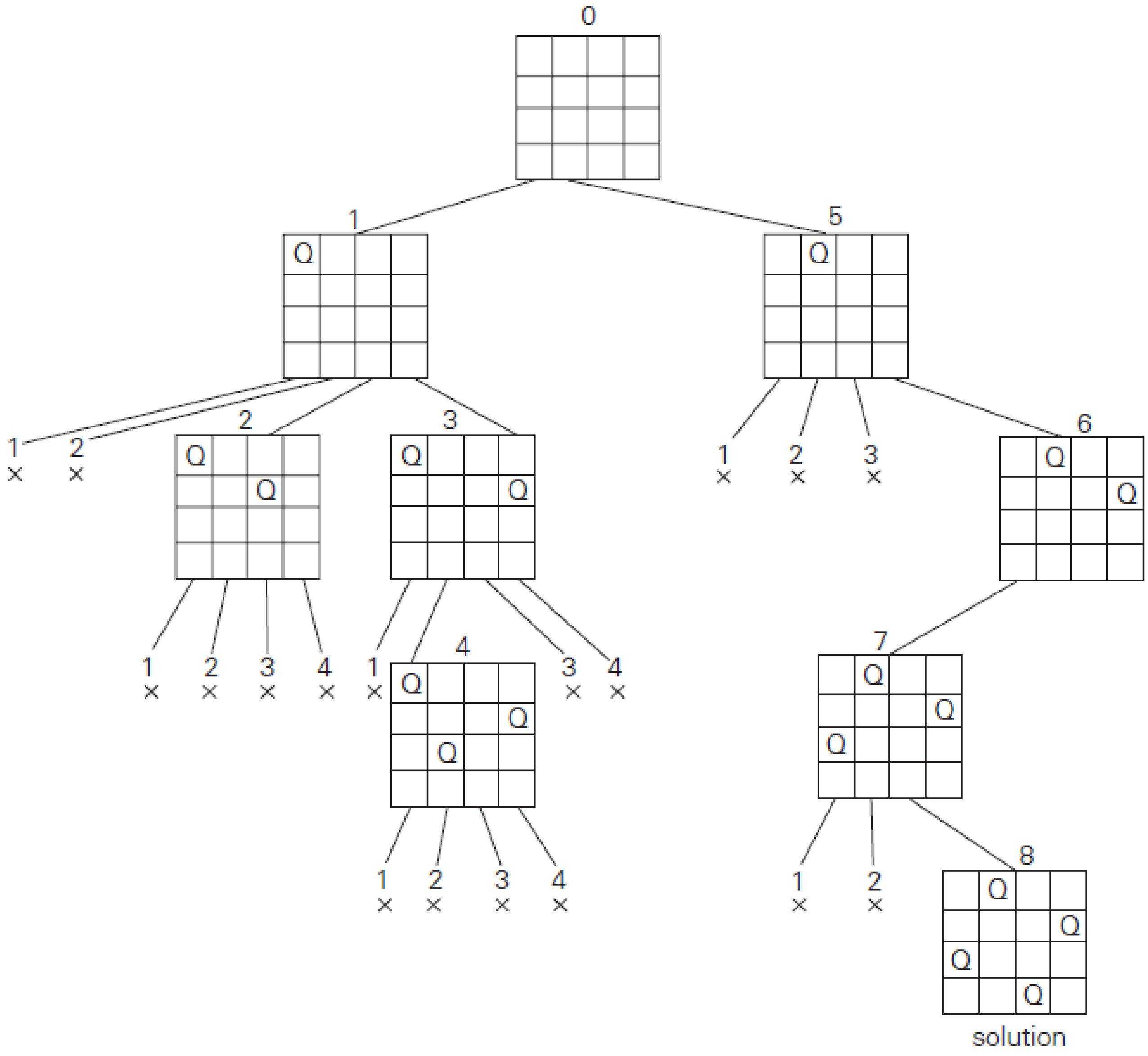


Figure 5.10 state-space tree of solving the four-queens problem by backtracking. × denotes anunsuccessful attempt to place a queen.

For*n*= 8,thereis**solution**toplace8 queensin8×8chessboard.

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4			Q					
5	Q							
6							Q	
7					Q			
8		Q						

Figure5.11solution8-queensproblemin8x8board.

Hamiltoniancircuitproblem

A**hamiltonian circuit** (also called a**hamiltonian cycle**,**hamilton cycle**, or**hamilton circuit)** is a graph **cycle** (i.e., closed loop) through a graph that visits each node exactly once. A graph possessing a **hamiltonian cycle** is said to be a **hamiltonian** graph.

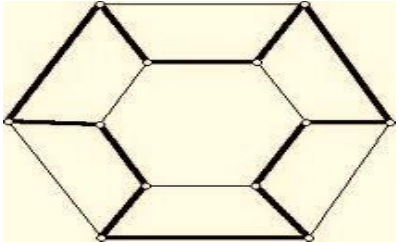


Figure5.12graphcontainshamiltonian circuit

Letusconsidertheproblemoffindingahamiltoniancircuitinthegraphinfigure5.13.

Example:findhamiltoniancircuit startsatvertex *a*.

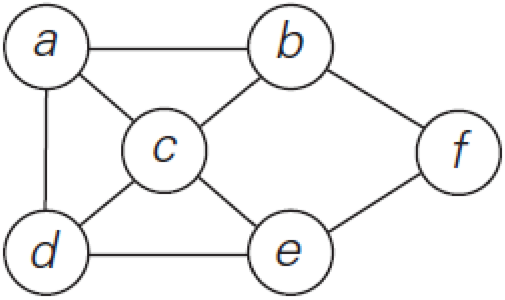


Figure5.13graph.

Solution:

- Assume that if a hamiltonian circuit exists, it starts at vertex *a*. Accordingly, we makevertex *a* the root of the state-space tree as in figure 5.14.
- In a graph g, hamiltonian cycle begins at some vertex $v_1 \in$ g, and the vertices are visited only once in the order $v_1, v_2, . . . , v_{n.}$(v_i are distinct except for v_1 and v_{n+1} which are equal).
- The first component of our future solution, if it exists, is a first intermediate vertex of a hamiltonian circuit to be constructed. Using the alphabet order to break the three-way tie among the vertices adjacent to *a*, we
- Selectvertex*b*.from*b*,thealgorithmproceedsto*c*,thento*d*,thento*e*,andfinallyto*f*, Whichprovesto be adead end.
- So the algorithm backtracks from *f*to *e*, then to *d*, and then to *c*, which provides the first alternative for the algorithm to pursue.
- Going from *c* to *e* eventually proves useless, and the algorithm has to backtrack from *e* to *c* and then to *b*. From there, it goes to the vertices *f* , *e*, *c*, and *d*, from which it canlegitimately return to *a*, yielding the hamiltonian circuit *a*, *b*, *f* , *e*, *c*, *d*, *a*. If we wanted to find another hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.

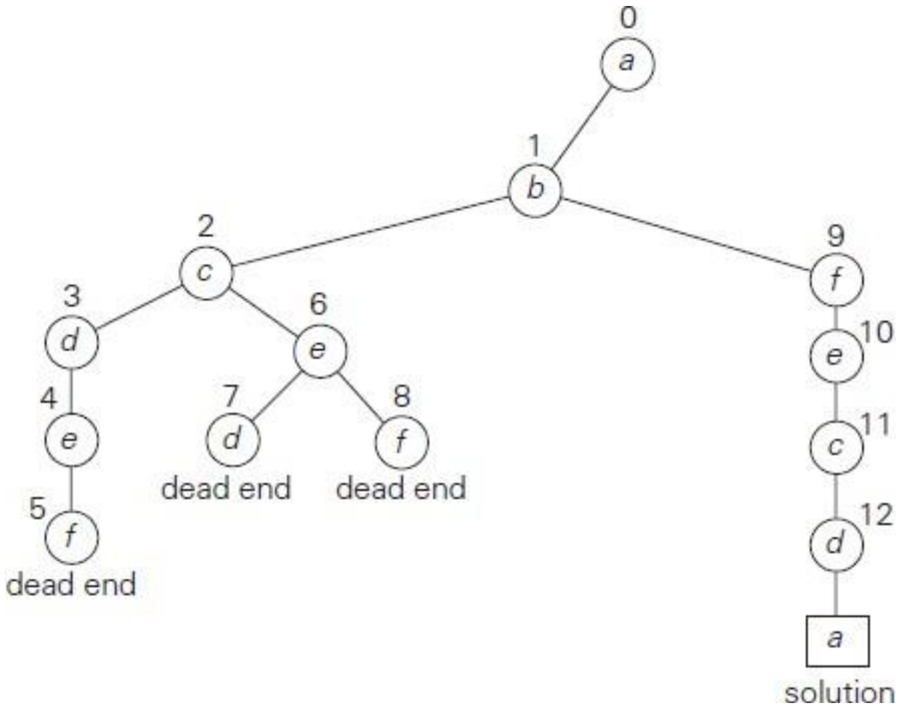


Figure5.14state-spacetreeforfindingahamiltonian circuit.

Subsetsumproblem

The ***subset-sum problem*** finds a subset of a given set *a* = {*a*1, . . . , *an*} of *n* positive integerswhose sumisequaltoagivenpositive integer*d*.forexample,for*a*={1,2,5,6,8}and *d* = 9, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

Itisconvenienttosorttheset'selementsinincreasingorder.so,wewillassumethat *A*1<*a*2 <. . . <*an.*

A={3, 5, 6, 7} and d =15 ofthesubset-sum problem. Thenumberinsideanodeis thesum of the elements alreadyincluded in the subsets represented bythe node. The inequalitybelow a leaf indicates the reason for its termination.

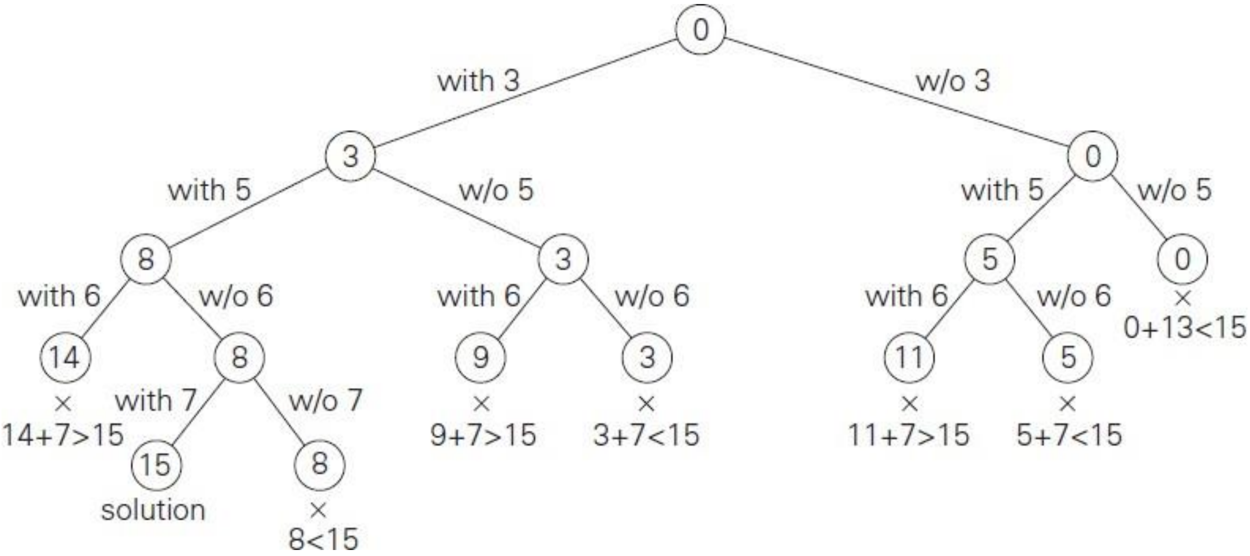


Figure5.15complete state-spacetreeofthebacktrackingalgorithmappliedtothe instance

Example:

- The state-space tree can be constructed as a binary tree like that in figure 5.15 for the instance*a* = {3, 5, 6, 7} and *d* = 15.
- The root of the tree represents the startingpoint,with no decisions aboutthe given elements made as yet.
- Itsleft and right children represent, respectively, inclusion and exclusionof $a_1$in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on.
- Thus,apathfromtheroottoanodeonthe*i*thlevelofthetreeindicateswhichofthefirst*i* Numbershavebeenincluded inthesubsets representedbythat node.
- Werecordthevalueof*s*,thesumofthesenumbers,in the node.
- If *s*isequalto*d*,wehavea solution totheproblem.wecaneither reportthis resultand stop Or,if allthe solutionsneed tobe found,continuebybacktrackingtothenode’sparent.
- If *s* is not equal to *d*, we can terminate the node as nonpromising if either of the followingtwo inequalities holds:

$s+a_{i+1}>d$(thesum*s* is toolarge),

$s+\sum_{1}^{nj=i+} a_j<d$(thesum*s* istoo small).

Generalremarks

Fromamoregeneralperspective,mostbacktrackingalgorithmsfitthefollowingescription. An output of a backtracking algorithm can be thought of as an *n*-tuple *(*x_1, x_2, . . . , x_n*)* where each coordinate *xi* is an element of some finite lin early ordered set *si* . For example, for the *n*-queens problem, each *si* is the set of integers (column numbers) 1 through *n.*

A backtracking algorithm generates, explicitly or implicitly, a state-space tree; its nodes represent partiallyconstructedtuples with thefirst i coordinatesdefined bytheearlieractions ofthe algorithm. If such a tuple (x_1, x_2, . . . , x_i) is not a solution, the algorithm finds the next element in s_{i+1}that is consistent with the values of ((x_1, x_2, . . . , x_i) and the problem’s constraints, and adds it to the tuple as its (i + 1)st coordinate. If such an element does not exist, the algorithm backtracks to consider the next value of xi, and so on.

```
Algorithmbacktrack(x[1..i])
    //givesatemplateofa genericbacktrackingalgorithm
    //input:x[1..i]specifiesfirst ipromisingcomponentsofasolution
    //output:allthetuplesrepresentingtheproblem’ssolutions
    If x[1..i]isasolutionwritex[1..i]
    Else    //seeproblemthis section
            Foreachelement x∈si+1consistentwithx[1..i]andtheconstraintsdo
                X[i + 1] ← x
                backtrack(x[1..i+1])
```

Branchandbound

An optimization problem seeks to minimize or maximize some objective function, usually subject to some constraints. Note that in the standard terminology of optimization problems, a ***feasible solution*** is a point in the problem's search space that satisfies all the problem's constraints (e.g., a hamiltonian circuit in the travelling salesman problem or a subset of items whose total weight does not exceed the knapsack's capacity in the knapsack problem), whereas an ***optimal solution*** is a feasible solution with the best value of the objective function (e.g., the shortest hamiltonian circuit or the most valuable subset of items that fit the knapsack).

Comparedtobacktracking,branch-and-boundrequirestwoadditionalitems:

1. A way to provide, for every node of a state-space tree, a bound on the best value of the objective function1 on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
2. Thevalueofthebestsolutionseensofar

If this information is available, we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated (some people say the branch is "pruned"). Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

1. Thevalue ofthenode'sbound is not betterthanthevalue ofthebest solution seen sofar.
2. The node represents no feasible solutions because the constraints of the problem arealready violated.
3. The subset of feasible solutions represented by the node consists of a single point (andhence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Someproblemscanbesolved bybranch-and-boundare:

1. Assignmentproblem
2. Knapsackproblem
3. Travelingsalesman problem

Assignmentproblem

Letusillustratethebranch-and-boundapproachbyapplyingittotheproblemofassigning *n* people to *n* jobs so that the total cost of the assignment is as small as possible. An instance of the assignment problem is specified by an *n* × *n* cost matrix *c.*

$$C = \begin{array}{cccccl} & \text{job } 1 & \text{job } 2 & \text{job } 3 & \text{job } 4 & \\ \left[\begin{matrix} \\ \\ \\ \\ \end{matrix}\right. & \begin{matrix} 9 \\ 6 \\ 5 \\ 7 \end{matrix} & \begin{matrix} 2 \\ 4 \\ 8 \\ 6 \end{matrix} & \begin{matrix} 7 \\ 3 \\ 1 \\ 9 \end{matrix} & \begin{matrix} 8 \\ 7 \\ 8 \\ 4 \end{matrix} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right] & \begin{matrix} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix} \end{array}$$

We have to find a lower bound on the cost of an optimal selection without actually solving the problem. We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For the instance here, this sum is 2 + 3+ 1+ 4 = 10. It is important to stressthat this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17.

It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the ***best-first branch-and-bound***.

Thelower-bound valuefortheroot,denoted *lb*, is10. Thenodes on thefirst level ofthetree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a asshown in* figure 5.15.

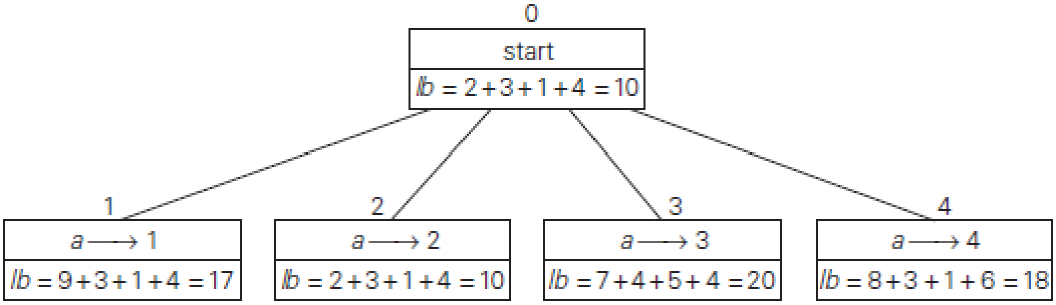


Figure 5.15 levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person*a* and the lower bound value, *lb*, for this node.

.

Sowehavefourliveleaves(promisingleavesarealsocalled***live***)—nodes1through4—
That may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value. Following our best-first search strategy, we branch out from that node first byconsidering the three different ways of selecting an element from the second row and not in the second column—the three different jobs that can be assigned to person *b* (figure 5.16).

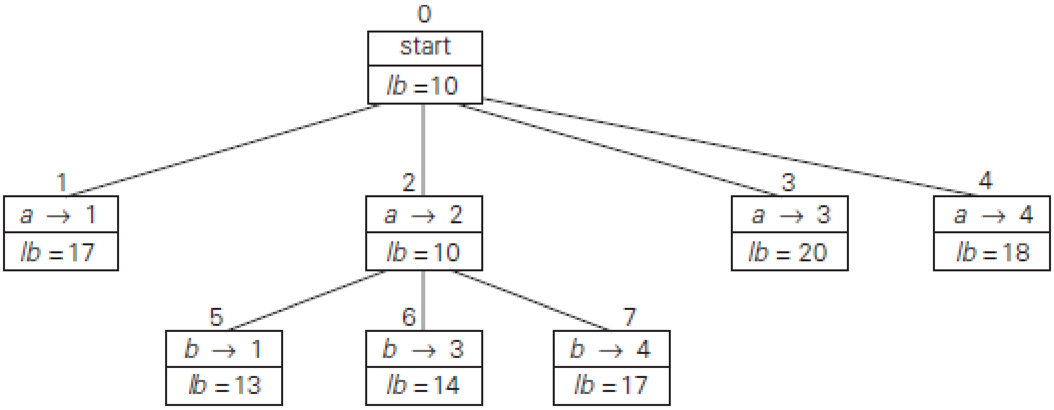


Figure 5.16 levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from *c*'s row (i.e., assigning person *c* to job 3); this leaves us with no choice but to select the element from the fourth column of *d*'s row (assigning person *d* to job 4). This yields leaf 8 (figure 5.17), which corresponds to the feasible solution {*a*→2, *b*→1, *c*→3, *d* →4} with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution {*a*→2, *b*→1, *c*→4, *d* →3} with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)

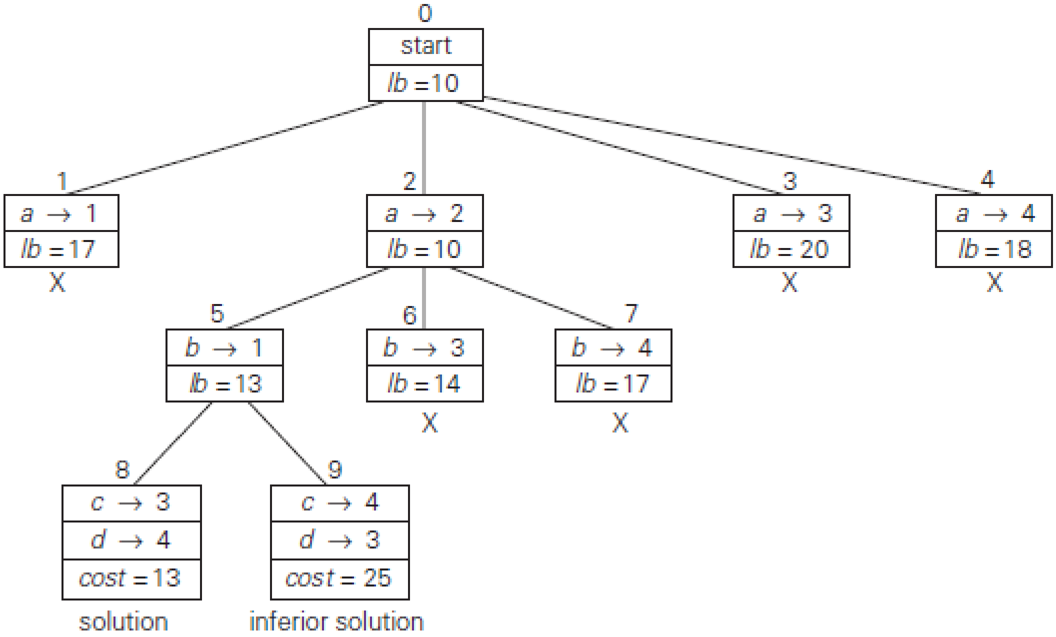


Figure 5.17 complete state-space tree for the instance of the assignment problem solved withthe best-first branch-and-bound algorithm.

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in figure 5.17—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

Knapsackproblem

Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. Given *n* items of known weights $w_i$and values v_i, $i = 1, 2, . . . , n$, and a knapsack of capacity *w*, find the most valuable subset of the items that fit in the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$V1/w1 \geq v2/w2 \geq .. . \geq vn/wn.$$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows. Each node on the *i*th level of this tree, $0 \leq i \leq n$, represents all the subsets of *n* items that include a particular selection made from the first*i* ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion. We record the total weight *w* and the total value *v* of this selection in the node, along with some upper bound *ub*on the value of any subset that can be obtained by adding zero or more items to this selection.

Item	Weight	Value	Value/weight	Capacity
1	4	$40	10	*W*=10
2	7	$42	6	
3	5	$25	5	
4	3	$12	4	
	W=19	V=119	$V_{i+1}/w_{i+1}=25$	

A simple way to compute the upper bound *ub* is to add to *v*, the total value of the items already selected, the product of the remaining capacity of the knapsack *w* − *w* and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

$$Ub = v+(w - w)(v_{i+1}/w_{i+1}).$$
$$=0+(10-0)(10)$$
$$=100$$

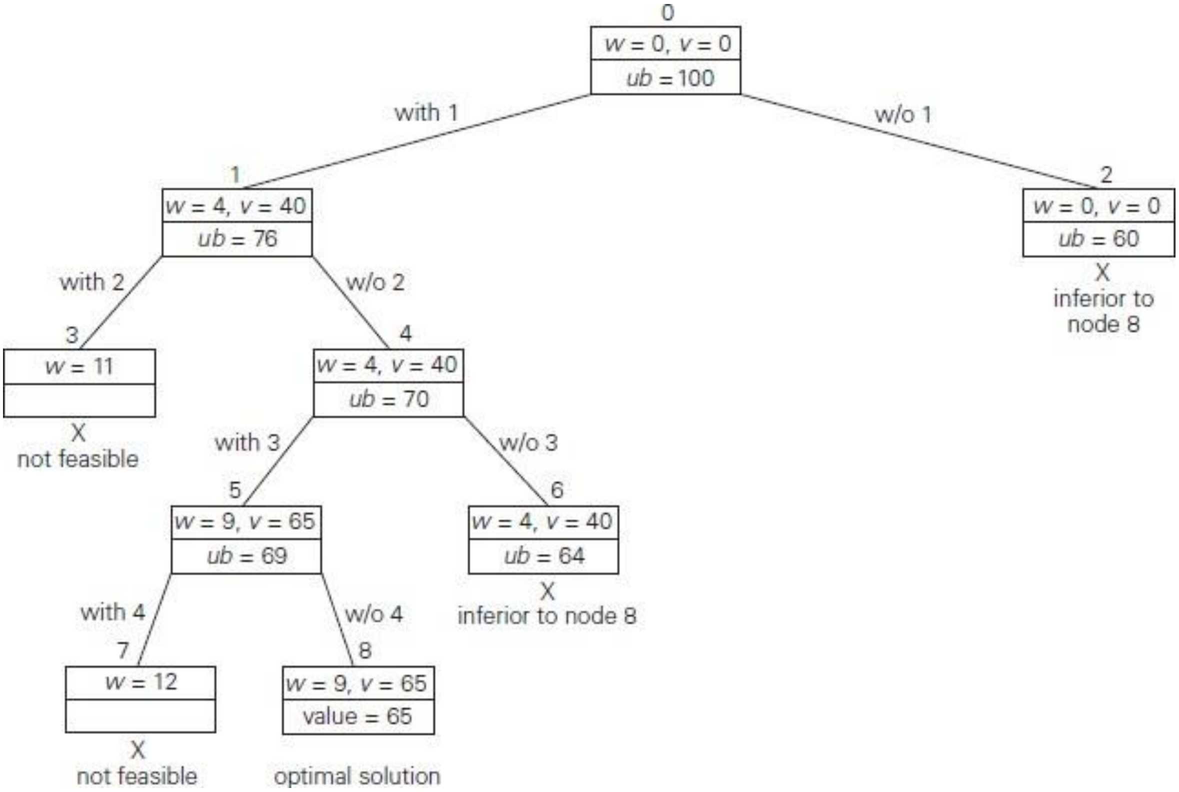


Figure 5.18 state-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

At the root of the state-space tree (see figure 5.18), no items have been selected as yet. Hence, both the total weight of the items already selected *w* and their total value *v* are equal to 0. The value of the upper bound computed by formula (12.1) is $100. Node 1, the left child of theroot, represents the subsets that include item 1. The total weight and value of the items already included are 4 and $40, respectively; the value of the upper bound is 40 + *(*10 − 4*)** 6 = $76. Node 2 represents the subsets that do not include item 1. Accordingly, *w* = 0, *v* = $0, and *ub* = 0 + *(*10 −0*)* * 6 = $60. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children—nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively.

Since the total weight *w* of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of *w* and *v* as its parent; the upper bound *ub* is equal to 40 + *(*10 − 4*)* * 5 = $70. Selecting node 4 over node 2 for the next branching (why?), we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upperbounds for these nodes are computed in the same way as for the preceding nodes. Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset {1, 3} of value $65. The remaining live nodes2and6havesmallerupper-boundvaluesthanthevalueofthesolutionrepresentedbynode 8. Hence, both can be terminated making the subset {1, 3} of node 8 the optimal solution to the problem.

Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic. Typically, internal nodes of a state-space tree do not define a point of the problem's search space, because some of the solution's components remain undefined. If we had done this for the instance investigated above, we could have terminated nodes 2 and 6 before node 8 was generated because they both are inferior to the subset of value $65 of node 5.

Travelingsalesmanproblem

We will be able to apply the branch-and-bound technique to instances of the travelling salesman problem if we come up with a reasonable lower bound on tour lengths. One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix *d* and multiplying it by the number of cities *n*. But there is a less obvious and more informative lower boundforinstances with symmetricmatrix *d*,whichdoes notrequirealot ofwork tocompute. Itis not difficult to show (problem 8 in this section's exercises) that we can compute a lower bound on the length *l* of any tour as follows. For each city *i*, 1≤ *i* ≤ *n*, find the sum *si* of the distances fromcity*i* to the two nearest cities; compute the sum *s* of these *n* numbers, divide the result by 2, and, if all the distances are integers, round up the result to the nearest integer:

$$lb=]s/2]$$

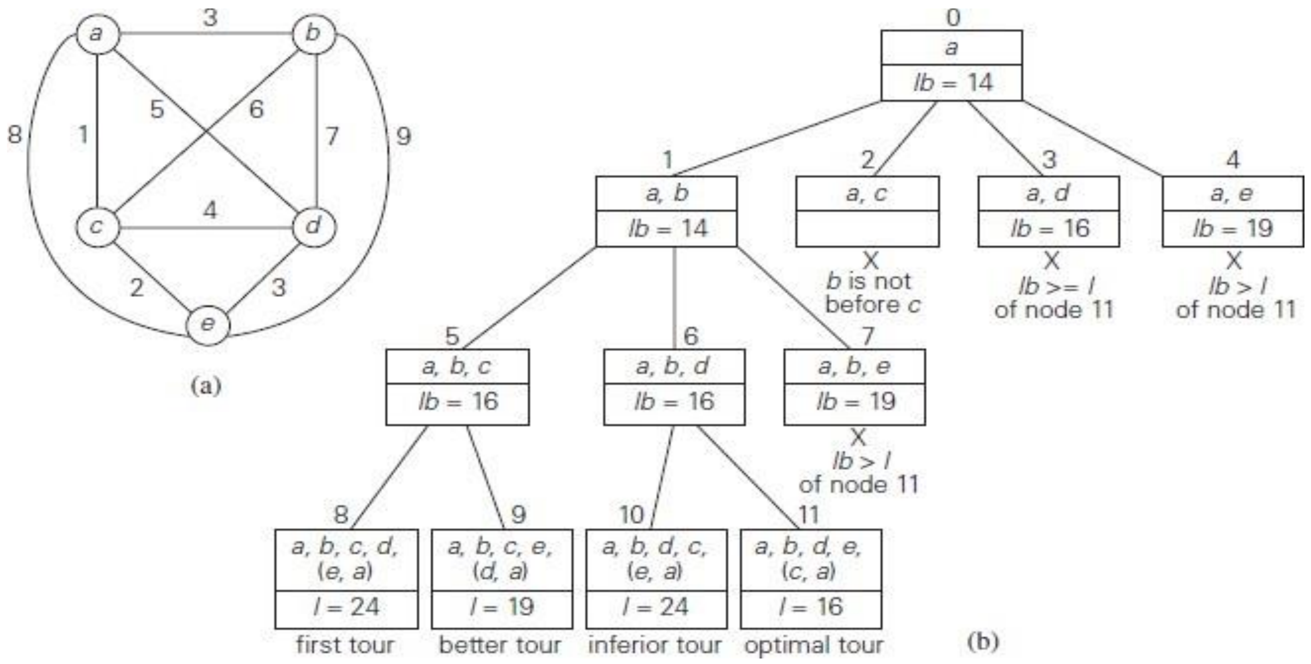


Figure 5.19 (a)weighted graph. (b) state-space tree of the branch-and-bound algorithm to find a shortest hamiltoniancircuit in this graph. Thelist ofvertices in anodespecifiesabeginningpartof the hamiltonian circuits represented by the node.

Forexample,fortheinstancein figureand above formulayields
Lb =][(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2]=14.

Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound accordingly. For example, for all the hamiltonian circuits of the graph in figurethat must include edge *(a, d),* we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges *(a, d)* and *(d, a)*:

][(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2]=16.

We now apply the branch-and-bound algorithm, with the bounding function given by formula, to find the shortest hamiltonian circuit for the graph in figure 5.19a. To reduce theamountofpotentialwork.first,withoutlossofgenerality,wecanconsideronlytoursthatstartat *A*.second,becauseourgraphisundirected,wecangenerateonlytoursinwhich*b*isvisitedbefore *C*. In addition, after visiting *n* − 1= 4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one. The state-space tree tracing the algorithm’s application is givenin figure 5.19b.

Approximationalgorithmsfornphardproblems

Now we are going to discuss a different approach to handling difficult problems of combinatorialoptimization,suchasthe **travellingsalesmanproblemandtheknapsackproblem.** The decision versions of these problems are *np*-complete. Their optimization versions fall in the class of ***np-hard problems***—problems that are at least as hard as *np*-complete problems. Hence, there are no known polynomial-time algorithms for these problems, and there are serioustheoretical reasons to believe that such algorithms do not exist.

Approximation algorithms runa gamut in level of sophistication; most ofthem arebased on some problem-specific heuristic. A ***heuristic*** is a common-sense rule drawn from experience rather thanfrom amathematicallyproved assertion. For example, goingto thenearest unvisited cityin the travelling salesman problem is a good illustration of this notion.

Of course, if we use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is. We can quantify the accuracy of an approximate solution $\boldsymbol{s_a}$to a problem of ***minimizing*** some function ***f***by the size of the relative error (***re***) of this approximation,

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)}$$

Where s^* is an exact solution to the problem. Alternatively, *re(sa) = f (sa)/f (s*) −* 1, we can simply use the ***accuracy ratio***

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

Asameasureofaccuracyofs_a.notethatforthesakeofscaleuniformity,theaccuracyratioof approximate solutions to ***maximization*** problems is usually computed as

$$r(s_a) = \frac{f(s^*)}{f(s_a)}$$

Tomakethisratiogreaterthanorequalto1,asitisforminimizationproblems.obviously,the Closer $r(s_a)$ is to 1, the better the approximate solution is. For most instances, however, we cannot compute the accuracy ratio, because we typically do not know *f (s*)*, the true optimal value of the objective function. Therefore, our hope should liein obtaininga good upper bound on the values of $r(s_a)$. This leads to the following definitions.

A polynomial-time approximation algorithm is said to be a ***c approximation algorithm***, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed *c* for any instance of the problem in question: $r(s_a) \leq c$.

The best (i.e., the smallest) value of *c* for which inequality holds for all instances of the problem is called the ***performance ratio*** of the algorithm and denoted r_a.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with ra as close to 1as possible. Unfortunately, as we shall see, some approximation algorithms have infinitely large performance ratios (ra = ∞). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

Approximationalgorithmsfornphardproblemsare:

- Travelingsalesmanproblem(tsp)
- Knapsackproblem

Travelingsalesmanproblem(approximationalgorithm)

Greedy algorithms for the tsp the simplest approximation algorithms for the traveling salesman problem are based on the greedy technique. We will discuss here two such algorithms.

1. Nearest-neighboralgorithm
2. Minimum-spanning-tree–basedalgorithms

Nearest-neighboralgorithm

Thefollowingwell-knowngreedyalgorithmisbasedonthe***nearest-neighbor***heuristic: always go next to the nearest unvisited city.

Step1chooseanarbitrarycityasthestart.

Step 2 repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step3returntothestarting city.

Example 1 for the instance represented by the graph in figure 5.20, with *a* as the starting vertex, the nearest-neighbor algorithm yields the tour (hamiltonian circuit) s_a: $a - b - c - d - a$ of length 10.

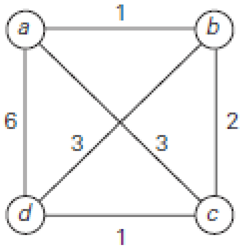


Figure5.20 instanceofthetravelingsalesman problem.

Theoptimalsolution,as canbeeasilycheckedbyexhaustive search,isthetours^*:a–b– d–c–a

Oflength 8.thus, the accuracyratio ofthis approximationis

$$r(s_a)=\frac{f(s_a)}{f(s^*)} = \frac{10}{8}=1.25$$

(i.e.,tour$s_a$is25%longerthantheoptimaltours^*).

Multifragment-heuristicalgorithm

Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have **degree 2.**

Step 1 sort the edges in increasing order of their weights. (ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

Step 2 repeat this step *n* times,where*n* is thenumberofcitiesin theinstancebeingsolved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than *n*; otherwise, skip the edge.

Step3returnthesetoftouredges.

As an example, applying the algorithm to the graph in figure 5.20 yields {*(a, b), (c, d), (b, c), (a, d)*}.this set of edges forms the same tour as the one produced by the nearest-neighbor algorithm.ingeneral,themultifragment-heuristicalgorithmtendstoproducesignificantlybetter

Tours thanthenearest-neighbor algorithm, as weare goingto seefrom the experimental dataquoted at the end of this section.but the performance ratio of the multifragment-heuristic algorithm is also unbounded, of course.

There is, however, averyimportant subset of instances, called ***euclidean***, forwhich we can make a nontrivial assertion about the accuracy of both the nearestneighbor and multifragment-heuristic algorithms. These are the instances in which intercity distances satisfy the following natural conditions:

- ***Triangleinequality***$d[i,j] \le d[i,k] + d[k,j]$foranytripleofcities$i$,$j$,and$k$(thedistance between cities$i$and$j$cannot exceed the length of a two-leg path from$i$to some intermediate city k to j)
- ***Symmetry***$d[i,j] = d[j,i]$foranypairofcities$i$and$j$(thedistancefrom$i$to$j$isthesame as the distance from j to i)

Minimum-spanning-tree–basedalgorithms

There are approximation algorithms for the travelling salesman problem that exploit a connection between hamiltonian circuits and spanning trees of the same graph. Since removing an edge from a hamiltonian circuit yields a spanning tree, we can expect that the structure of a minimum spanning tree provides a good basis for constructing a shortest tour approximation. Here is an algorithm that implements this idea in a rather straightforward fashion.

Twice-around-the-treealgorithm

Step 1 construct a minimum spanning tree of the graph corresponding toa given instanceof the traveling salesman problem.

Step 2 starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (this can be done by a dfs traversal.)

Step 3 scan the vertex list obtained in step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (this step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a hamiltonian circuit, which is the output of the algorithm.

Example2 let usapplythis algorithmtothegraphin figure5.21a. Theminimumspanningtree ofthisgraphismadeupofedges *(a,b)*, *(b,c)*, *(b, d)*,and *(d,e)*(figure5.21b).atwice-around-the- tree walk that starts and ends at *a* is *a, b, c, b, d, e, d, b, a.* Eliminating the second *b* (a shortcutfrom *c* to *d*), the second *d*, and the third *b* (a shortcut from *e* to *a*) yields the hamiltonian circuit *a, b, c, d, e, a* of length 39.

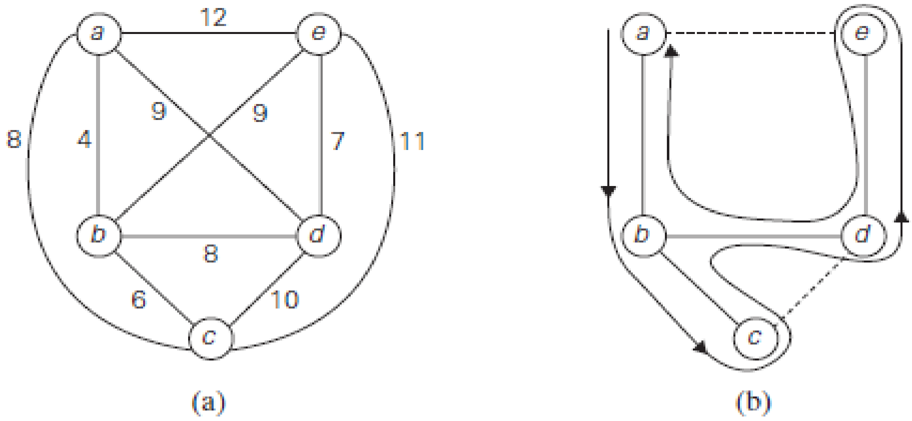


Figure 5.21 illustration of the twice-around-the-tree algorithm. (a) graph. (b) walk around the minimum spanning tree with the shortcuts.

Knapsackproblem(approximationalgorithm)

The knapsack problem is one well-known*np*-hard problem. Given *n* items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$and a knapsack of weight capacity *w*, find the most valuable subset of the items that fits into the knapsack.

Greedyalgorithmsfortheknapsackproblem

We can think of several greedy approaches to this problem. One is to select the items in decreasing order of their weights; however, heavier items may not be the most valuable in the set. Alternatively, if we pick up the items in decreasing order of their value, there is no guarantee that the knapsack's capacity will be used efficiently. We find a greedy strategy that takes into account both the weights and values by computing the value-to-weight ratios v_i/w_i, $i = 1, 2, \ldots, n$, and selecting the items in decreasing order of these ratios. Here is the algorithm based on this greedy heuristic.

Greedyalgorithmforthediscreteknapsackproblem

Step1computethevalue-to-weightratiosr_i=v_i/w_i,i=1,...,n,fortheitemsgiven.

Step 2 sort the items in nonincreasing order of the ratios computed in step 1.(ties can be broken arbitrarily.)

Step 3 repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to thenext item; otherwise, just proceed to the next item.

Example 1 let us consider the instance of the knapsack problem with the knapsack capacity 10 and the item information as follows:

Item	Weight	Value
1	4	$40
2	7	$42
3	5	$25
4	3	$12

Computing the value-to-weight ratios and sorting the items in non increasing order of these efficiency ratios yields

Item	Weight	Value	Value/weight	Capacity
1	4	$40	10	*W*=10
2	7	$42	6	
3	5	$25	5	
4	3	$12	4	

The greedy algorithm will select the first item of weight **4**, skip the next item of weight 7, select the next item of weight **5**, and skip the last item of weight 3. The solution obtained happensto be optimal for this instance. So the total items value in knapsack is **$65**.

Greedyalgorithmforthecontinuousknapsackproblem

Step1computethevalue-to-weightratios v_i/w_i,i=1,...,n,fortheitemsgiven.

Step 2 sort the items in nonincreasing order of the ratios computed in step 1. (ties can be broken arbitrarily.)

Step 3 repeat the following operation until the knapsack is filled to its full capacity or noitemisleftinthesortedlist:ifthe currentitemonthelistfitsintothe knapsackinits

Entirety, take it and proceed to the next item; otherwise, take its largest fraction tofill the knapsack to its full capacity and stop.

Example 2 a small example of an approximation scheme with *k* = 2 is provided. The algorithm yields {1, 3, 4}, which is the optimal solution for this instance.

Item	**Weight**	**Value**	**Value/weight**	**Capacity**
1	4	$40	10	*W*=10
2	7	$42	6	
3	5	$25	5	
4	1	$4	4	

Subset	**Addeditems**	**Value**
{}	1, 3, 4	$69
{1}	3, 4	$69
{2}	4	$46
{3}	1, 4	$69
{4}	1, 3	$69
{1, 2}	Not feasible	
{1, 4}	4	$69
{1, 4}	3	$69
{2, 3}	Not feasible	
{2, 4}	-	$46
{3, 4}	1	$69

For each of those subsets, it needs *o(n)* time to determine the subset's possible extension. Thus, the algorithm's efficiency is in $o(kn^{k+1})$. Note that although it is polynomial in *n*, the time efficiencyofsahni's schemeis exponential in *k*. Moresophisticatedapproximation schemes,called ***fully polynomial schemes***, do not have this shortcoming.