

[Click Here for Design and Analysis of Algorithms full study material.](#)

Analysis and Design of Algorithm

By

Mr. J.Marimuthu

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Srividya College of Engineering & Technology

UNIT - I ALGORITHM ANALYSIS

Algorithm analysis – Time space tradeoff – Asymptotic notations – Conditional asymptotic notation – Removing condition from the conditional asymptotic notation – Properties of Big-oh notation – Recurrence equations – Solving recurrence equations – Analysis of linear search.

1.1 Introduction

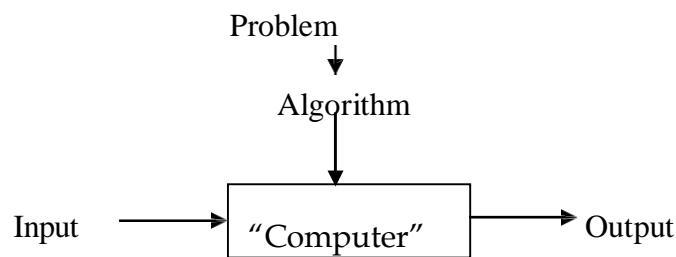
An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.

Definition

“Algorithmic is more than the branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant it most of science, business and technology”

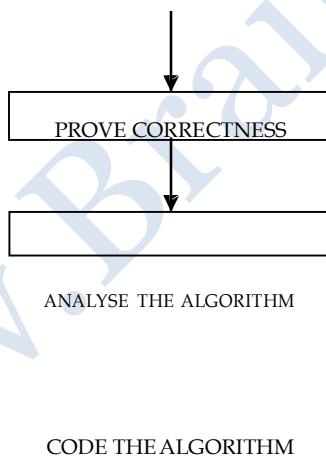
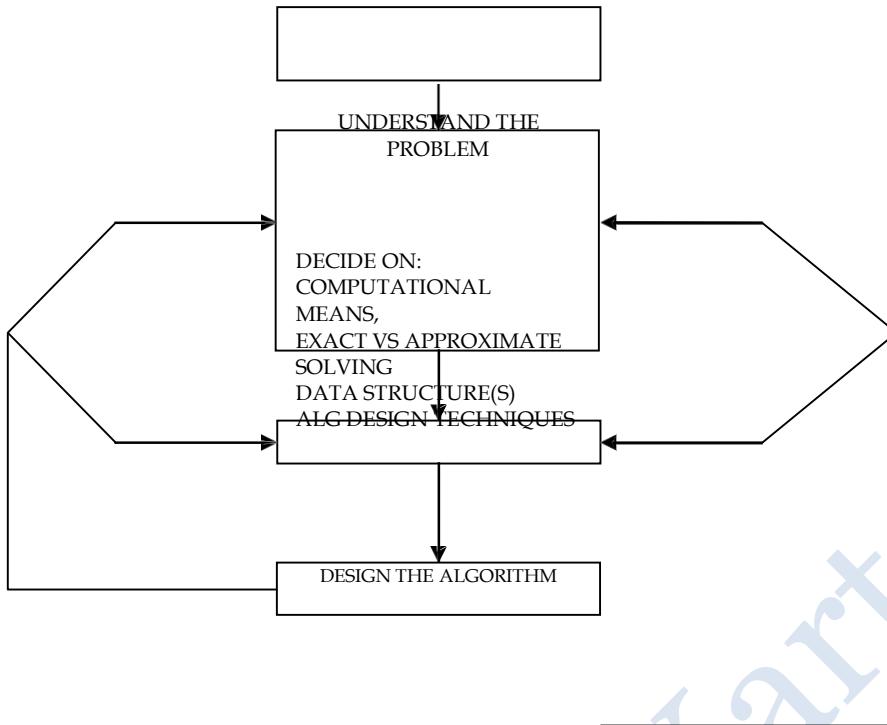
Understanding of Algorithm

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



ALGORITHM DESIGN AND ANALYSIS PROCESS

[Click Here for Design and Analysis of Algorithms full study material.](#)



FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

ANALYSIS FRAME WORK

- ♦ there are two kinds of efficiency
 - ♦ **Time efficiency** - indicates how fast an algorithm in question runs.
 - ♦ **Space efficiency** - deals with the extra space the algorithm requires.

[Click Here for Design and Analysis of Algorithms full study material.](#)

MEASURING AN INPUT SIZE

- ◆ An algorithm's efficiency 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, finding the list's smallest element, and most other problems dealing with lists.
- ◆ For the problem of evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n , it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.
- ◆ There are situations, of course, where the choice of a parameter indicating an input size does matter.
 - ◆ **Example** - computing the product of two n -by- n matrices.
 - ◆ There are two natural measures of size for this problem:
 - ◆ The matrix order n .
 - ◆ The total number of elements N in the matrices being multiplied.
- ◆ Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.
- ◆ The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.
- ◆ We should make a special note about measuring size of inputs for algorithms involving properties of numbers (e.g., checking whether a given integer n is prime).
- ◆ For such algorithms, computer scientists prefer measuring size by the number b of bits in the n 's binary representation:
$$b = \lfloor \log_2 n \rfloor + 1$$
- ◆ This metric usually gives a better idea about efficiency of algorithms in question.

UNITS FOR MEASURING RUN TIME:

- ◆ We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.
- ◆ There are obvious drawbacks to such an approach. They are
 - ◆ Dependence on the speed of a particular computer
 - ◆ Dependence on the quality of a program implementing the algorithm
 - ◆ The compiler used in generating the machine code
 - ◆ The difficulty of clocking the actual running time of the program.
- ◆ Since we are in need to measure algorithm efficiency, we should have a metric that does not depend on these extraneous factors.
- ◆ One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both difficult and unnecessary.
- ◆ The main objective is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

WORST CASE, BEST CASE AND AVERAGE CASE EFFICIENCIES

- ◆ It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.
- ◆ But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.
- ◆ **Example, sequential search.** This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.
- ◆ Here is the algorithm's pseudo code, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition $A[i] = K$ will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

ALGORITHM Sequential Search($A[0..n - 1]$, K)

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n - 1] and a search key K
//Output: Returns the index of the first element of A that matches K
//          or -1 if there are no matching elements
i←0
while i < n and A[i] ≠ K do
    i←i+1
if i < n return i
else return -1
```

- ◆ Clearly, the running time of this algorithm can be quite different for the same list size n.

Worst case efficiency

- ◆ The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.
- ◆ In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$C_{\text{worst}}(n) = n.$$

- ◆ The way to determine is quite straightforward
- ◆ To analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $C(n)$ among all possible inputs of size n and then compute this worst-case value $C_{\text{worst}}(n)$
- ◆ The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{\text{worst}}(n)$ its running time on the worst-case inputs.

Best case Efficiency

- ◆ The best-case efficiency of an algorithm is its efficiency for the best-case input of size n , which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.
- ◆ We can analyze the best case efficiency as follows.
- ◆ First, determine the kind of inputs for which the count $C(n)$ will be the smallest among all possible inputs of size n . (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)
- ◆ Then ascertain the value of $C(n)$ on these most convenient inputs.
- ◆ Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, $C_{\text{best}}(n) = 1$.
- ◆ The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency.
- ◆ But it is not completely useless. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast.
- ◆ Thus, such an algorithm might well be the method of choice for applications dealing with almost sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

Average case efficiency

- ◆ It yields the information about an algorithm about an algorithm's behaviour on a -typical|| and -random|| input.
- ◆ To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n .
- ◆ The investigation of the average case efficiency is considerably more difficult than investigation of the worst case and best case efficiency.
- ◆ It involves dividing all instances of size n into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same.
- ◆ Then a probability distribution of inputs needs to be obtained or assumed so that the expected value of the basic operation's count can then be derived.

The average number of key comparisions $C_{\text{avg}}(n)$ can be computed as follows,

- ◆ let us consider again sequential search. The standard assumptions are,
- ◆ In the case of a successful search, the probability of the first match occurring in the i th position of the list is p_i for every i , and the number of comparisons made by the algorithm in such a situation is obviously i .

- ♦ In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being $(1 - p)$. Therefore,

$$\begin{aligned}
 C_{\text{avg}}(n) &= [\frac{p}{n} + \frac{p}{n} + \dots + \frac{p}{n} + \dots + \frac{p}{n}] + n \cdot (1 - p) \\
 &= \frac{p}{n} [1 + 2 + 3 + \dots + i + \dots + n] + n (1 - p) \\
 &= \frac{n}{2} \frac{p(n+1)}{n} + n (1 - p) \\
 &= \frac{p(n+1)}{2} + n (1 - p)
 \end{aligned}$$

- ♦ Example, if $p = 1$ (i.e., the search must be successful), the average number of key comparisons made by sequential search is $(n + 1)/2$.
- ♦ If $p = 0$ (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

1.2.5 Asymptotic Notations

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Because the values are not exact quantities. We need only comparative statements like $c_1n^2 \leq t_p(n) \leq c_2n^2$.

For example, consider two programs with complexities $c_1n^2 + c_2n$ and c_3n respectively. For small values of n , complexity depend upon values of c_1 , c_2 and c_3 . But there will also be an n beyond which complexity of c_3n is better than that of $c_1n^2 + c_2n$. This value of n is called break-even point. If this point is zero, c_3n is always faster (or at least as fast). Common asymptotic functions are given below.

Function	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	$n \log n$
n^2	Quadratic
n^3	Cubic

2^n	Exponential
$n!$	Factorial

Big'Oh'Notation(O)

$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as

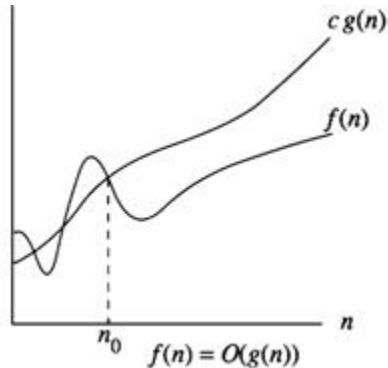


Fig 1.1

Find the Big Oh° for the following functions:

Linear Functions

Example 1.6

$$f(n) = 3n + 2$$

General form is $f(n) \leq cg(n)$

When $n \geq 2$, $3n + 2 \leq 3n + n = 4n$

Hence $f(n) = O(n)$, here $c = 4$ and $n_0 = 2$

When $n \geq 1$, $3n + 2 \leq 3n + 2n = 5n$

Hence $f(n) = O(n)$, here $c = 5$ and $n_0 = 1$

Hence we can have different c, n_0 pairs satisfying for a given function.

Example

$$f(n) = 3n + 3$$

When $n \geq 3$, $3n + 3 \leq 3n + n = 4n$

Hence $f(n) = O(n)$, here $c = 4$ and $n_0 = 3$

Example

$$f(n) = 100n + 6$$

When $n \geq 6$, $100n + 6 \leq 100n + n = 101n$

Hence $f(n) = O(n)$, here $c = 101$ and $n_0 = 6$

Quadratic Functions

Example 1.9

$$f(n) = 10n^2 + 4n + 2$$

When $n \geq 2$, $10n^2 + 4n + 2 \leq 10n^2 + 5n$

When $n \geq 5$, $5n \leq n^2$, $10n^2 + 4n + 2 \leq 10n^2 + n^2 = 11n^2$

Hence $f(n) = O(n^2)$, here $c = 11$ and $n_0 = 5$

Example 1.10

$$f(n) = 1000n^2 + 100n - 6$$

$f(n) \leq 1000n^2 + 100n$ for all values of n .

When $n \geq 100$, $5n \leq n^2$, $f(n) \leq 1000n^2 + n^2 = 1001n^2$

Hence $f(n) = O(n^2)$, here $c = 1001$ and $n_0 = 100$

Exponential Functions

Example 1.11

$$f(n) = 6*2^n + n^2$$

When $n \geq 4$, $n^2 \leq 2^n$

So $f(n) \leq 6*2^n + 2^n = 7*2^n$

Hence $f(n) = O(2^n)$, here $c = 7$ and $n_0 = 4$

Constant Functions

Example 1.12

$$f(n) = 10$$

$f(n) = O(1)$, because $f(n) \leq 10*1$

Omega Notation (Ω)

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as

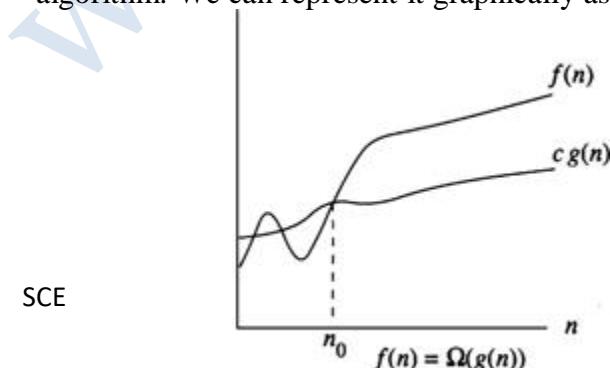


Fig 1.2

Example 1.13

$$f(n) = 3n + 2$$

$3n + 2 > 3n$ for all n .

Hence $f(n) = \Omega(n)$

Similarly we can solve all the examples specified under Big _Oh'.

ThetaNotation(Θ)

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$

If $f(n) = \Theta(g(n))$, all values of n right to n_0 $f(n)$ lies on or above $c_1g(n)$ and on or below $c_2g(n)$. Hence it is asymptotic tight bound for $f(n)$.

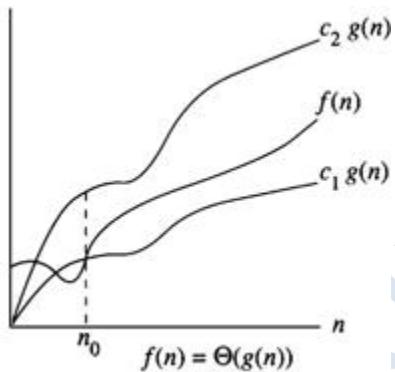


Fig 1.3

Little-O Notation

For non-negative functions, $f(n)$ and $g(n)$, $f(n)$ is little o of $g(n)$ if and only if $f(n) = O(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as " $f(n) = o(g(n))$ ".

This represents a loose bounding version of Big O. $g(n)$ bounds from the top, but it does not bound the bottom.

Little Omega Notation

For non-negative functions, $f(n)$ and $g(n)$, $f(n)$ is little omega of $g(n)$ if and only if $f(n) = \Omega(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as " $f(n) = \omega(g(n))$ ".

Much like Little Oh, this is the equivalent for Big Omega. $g(n)$ is a loose lower boundary of the function $f(n)$; it bounds from the bottom, but not from the top.

Conditional asymptotic notation

Many algorithms are easier to analyse if initially we restrict our attention to instances whose size satisfies a certain condition, such as being a power of 2. Consider, for example, the divide and conquer algorithm for multiplying large integers that we saw in the Introduction. Let n be the size of the integers to be multiplied.

The algorithm proceeds directly if $n = 1$, which requires a microseconds for an appropriate constant a . If $n > 1$, the algorithm proceeds by multiplying four pairs of integers of size $\lceil n/2 \rceil$ (or three if we use the better algorithm). Moreover, it takes a linear amount of time to carry out additional tasks. For simplicity, let us say that the additional work takes at most bn microseconds for an appropriate constant b .

Properties of Big-Oh Notation

Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. For example, if you want to write a function that searches through an array of numbers and returns the smallest one:

```
function find-min(array a[1..n] )
    let j :=
    for i := 1 to n:
        j := min(j, a[i] )
    repeat
    return j
end
```

Regardless of how big or small the array is, every time we run find-min, we have to initialize the i and j integer variables and return j at the end. Therefore, we can just think of those parts of the function as constant and ignore them.

So, how can we use asymptotic notation to discuss the find-min function? If we search through an array with 87 elements, then the for loop iterates 87 times, even if the very first element we hit turns out to be the minimum. Likewise, for n elements, the for loop iterates n times. Therefore we say the function runs in time O(n).

```
function find-min-plus-max(array a[1..n] )
    // First, find the smallest element in the array
    let j := ;
    for i := 1 to n:
        j := min(j, a[i] )
    repeat
    let minim := j

    // Now, find the biggest element, add it to the smallest and
    j := ;
    for i := 1 to n:
        j := max(j, a[i] )
    repeat

    let maxim := j

    // return the sum of the two
    return minim + maxim;
end
```

What's the running time for find-min-plus-max? There are two for loops, that each iterate n times, so the running time is clearly O(2n). Because 2 is a constant, we throw it away and write the running time as O(n). Why can you do this? If you recall the definition of Big-O notation, the function whose bound you're testing can be multiplied by some constant. If $f(x)=2x$, we can see that if $g(x) = x$, then the Big-O condition holds. Thus $O(2n) = O(n)$. This rule is general for the various asymptotic notations.

Recurrence

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence

Recurrence Equation

A recurrence relation is an equation that recursively defines a sequence. Each term of the sequence is defined as a function of the preceding terms. A difference equation is a specific type of recurrence relation.

An example of a recurrence relation is the logistic map:

$$x_{n+1} = rx_n(1 - x_n)$$

Another Example: Fibonacci numbers

The Fibonacci numbers are defined using the linear recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \end{aligned}$$

Explicitly, recurrence yields the equations:

$$\begin{aligned} F_2 &= F_1 + F_0 \\ F_3 &= F_2 + F_1 \\ F_4 &= F_3 + F_2 \end{aligned}$$

We obtain the sequence of Fibonacci numbers which begins:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$$

It can be solved by methods described below yielding the closed form expression which involve powers of the two roots of the characteristic polynomial $t^2 = t + 1$; the generating function of the sequence is the rational function $t / (1 - t - t^2)$.

Solving Recurrence Equation

i. substitution method

The substitution method for solving recurrences entails two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works. The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful, but it obviously

can be applied only in cases when it is easy to guess the form of the answer. The substitution method can be used to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n ,$$

which is similar to recurrences (4.2) and (4.3). We guess that the solution is $T(n) = O(n \lg n)$. Our method is to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$.

We start by assuming that this bound holds for $\lfloor n/2 \rfloor$, that is, that $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$.

Substituting into the recurrence yields

$$\begin{aligned}T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\&\leq cn \lg(n/2) + n \\&= cn \lg n - cn \lg 2 + n \\&= cn \lg n - cn + n \\&\leq cn \lg n,\end{aligned}$$

where the last step holds as long as $c \geq 1$.

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. For the recurrence (4.4), we must show that we can choose the constant c large enough so that the bound $T(n) = cn \lg n$ works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that $T(1) = 1$ is the sole boundary condition of the recurrence. Then for $n = 1$, the bound $T(n) = cn \lg n$ yields $T(1) = c1 \lg 1 = 0$, which is at odds with $T(1) = 1$. Consequently, the base case of our inductive proof fails to hold.

This difficulty in proving an inductive hypothesis for a specific boundary condition can be easily overcome. For example, in the recurrence (4.4), we take advantage of asymptotic notation only requiring us to prove $T(n) = cn \lg n$ for $n \geq n_0$, where n_0 is a constant of our choosing. The idea is to remove the difficult boundary condition $T(1) = 1$ from consideration.

1. In the inductive proof.

Observe that for $n > 3$, the recurrence does not depend directly on $T(1)$. Thus, we can replace $T(1)$ by $T(2)$ and $T(3)$ as the base cases in the inductive proof, letting $n_0 = 2$. Note that we make a distinction between the base case of the recurrence ($n = 1$) and the base cases of the inductive proof ($n = 2$ and $n = 3$). We derive from the recurrence that $T(2) = 4$ and $T(3) = 5$. The inductive proof that $T(n) \leq cn \lg n$ for some constant $c \geq 1$ can now be completed by choosing c large enough so that $T(2) \leq c2 \lg 2$ and $T(3) \leq c3 \lg 3$. As it turns out, any choice of $c \geq 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold. For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n .

2. The iteration method

The method of iterating a recurrence doesn't require us to guess the answer, but it may require more algebra than the substitution method. The idea is to expand (iterate) the

recurrence and express it as a summation of terms dependent only on n and the initial conditions. Techniques for evaluating summations can then be used to provide bounds on the solution.

As an example, consider the recurrence

$$T(n) = 3T(n/4) + n.$$

We iterate it as follows:

$$T(n) = n + 3T(n/4)$$

$$= n + 3(n/4 + 3T(n/16))$$

$$= n + 3(n/4 + 3(n/16 + 3T(n/64)))$$

$$= n + 3n/4 + 9n/16 + 27T(n/64),$$

where $n/4^i = n/16$ and $n/16^i = n/64$ follow from the identity (2.4).

How far must we iterate the recurrence before we reach a boundary condition? The i th term in the series is $3^i n/4^i$. The iteration hits $n = 1$ when $n/4^i = 1$ or, equivalently, when i exceeds $\log_4 n$. By continuing the iteration until this point and using the bound $n/4^i \leq n/4^{\log_4 n}$, we discover that the summation contains a decreasing geometric series:

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \dots + 3^{\log_4 n} \Theta(1) \\ &\leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n). \end{aligned}$$

3. The master method

The master method provides a "cookbook" method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The master method requires memorization of three cases, but then the solution of many recurrences can be determined quite easily, often without pencil and paper.

The recurrence (4.5) describes the running time of an algorithm that divides a problem of size

n into a subproblems, each of size n/b , where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$. (That is, using the notation from [Section 2.3.2](#), $f(n) = D(n)+C(n)$.) For example, the recurrence arising from the MERGE-SORT procedure has $a = 2$, $b = 2$, and $f(n) = \Theta(n)$.

As a matter of technical correctness, the recurrence isn't actually well defined because n/b might not be an integer. Replacing each of the a terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ doesn't affect the asymptotic behavior of the recurrence, however. We normally find it convenient, therefore, to omit the floor and ceiling functions when writing divide-and-conquer recurrences of this form.

Analysis of Linear Search

Linear Search, as the name implies is a searching algorithm which obtains its result by traversing a list of data items in a linear fashion. It will start at the beginning of a list, and mosey on through until the desired element is found, or in some cases is not found. The aspect of Linear Search which makes it inefficient in this respect is that if the element is not in the list it will have to go through the entire list. As you can imagine this can be quite cumbersome for lists of very large magnitude, keep this in mind as you contemplate how and where to implement this algorithm. Of course conversely the best case for this would be that the element one is searching for is the first element of the list, this will be elaborated more so in the -Analysis & Conclusion|| section of this tutorial.

Linear Search Steps:

Step 1 - Does the item match the value I'm looking for?

Step 2 - If it does match return, you've found your item!

Step 3 - If it does not match advance and repeat the process.

Step 4 - Reached the end of the list and still no value found? Well obviously the item is not in the list! Return -1 to signify you have not found your value.

As always, visual representations are a bit more clear and concise so let me present one for you now. Imagine you have a random assortment of integers for this list:

Legend:

-The key is blue

-The current item is green.

-Checked items are red

Ok so here is our number set, my lucky number happens to be 7 so let's put this value as the key, or the value in which we hope Linear Search can find. Notice the indexes of the

array above each of the elements, meaning this has a size or length of 5. I digress let us look at the first term at position 0. The value held here 3, which is not equal to 7. We move on.

--0 1 2 3 4 5

[3 2 5 1 7 0]

So we hit position 0, on to position 1. The value 2 is held here. Hmm still not equal to 7.

We march on.

--0 1 2 3 4 5

[3 2 5 1 7 0]

Position 2 is next on the list, and sadly holds a 5, still not the number we're looking for.

Again we move up one.

--0 1 2 3 4 5

[3 2 5 1 7 0]

Now at index 3 we have value 1. Nice try but no cigar let's move forward yet again.

--0 1 2 3 4 5

[3 2 5 1 7 0]

Ah Ha! Position 4 is the one that has been harboring 7, we return the position in the array which holds 7 and exit.

--0 1 2 3 4 5

[3 2 5 1 7 0]

As you can tell, the algorithm may work find for sets of small data but for incredibly large data sets I don't think I have to convince you any further that this would just be down right inefficient to use for exceeding large sets. Again keep in mind that Linear Search has its place and it is not meant to be perfect but to mold to your situation that requires a search.

Also note that if we were looking for lets say 4 in our list above (4 is not in the set) we would traverse through the entire list and exit empty handed. I intend to do a tutorial on Binary Search which will give a much better solution to what we have here however it requires a special case.

```
//linearSearch Function  
int linearSearch(int data[], int length, int val) {  
  
    for (int i = 0; i <= length; i++) {  
        if (val == data[i] ) {  
            return i;  
        } //end if  
    } //end for  
    return -1; //Value was not in the list  
} //end linearSearch Function
```

Analysis & Conclusion

As we have seen throughout this tutorial that Linear Search is certainly not the absolute best method for searching but do not let this taint your view on the algorithm itself. People are always attempting to better versions of current algorithms in an effort to make existing ones more efficient. Not to mention that Linear Search as shown has its place and at the very least is a great beginner's introduction into the world of searching algorithms. With this in mind we progress to the asymptotic analysis of the Linear Search:

Worst Case:

The worse case for Linear Search is achieved if the element to be found is not in the list at all. This would entail the algorithm to traverse the entire list and return nothing. Thus the worst case running time is:

$$O(N).$$

Average Case:

The average case is in short revealed by insinuating that the average element would be somewhere in the middle of the list or $N/2$. This does not change since we are dividing by a constant factor here, so again the average case would be:

$$O(N).$$

Best Case:

The best case can be reached if the element to be found is the first one in the list. This would not have to do any traversing spare the first one giving this a constant time complexity or:

$$O(1).$$

IMPORTANT QUESTIONS

PART-A

1. Define Algorithm & Notion of algorithm.
2. What is analysis framework?
3. What are the algorithm design techniques?
4. How is an algorithm's time efficiency measured?
5. Mention any four classes of algorithm efficiency.
6. Define Order of Growth.
7. State the following Terms.
 - (i) Time Complexity
 - (ii) Space Complexity
8. What are the various asymptotic Notations?
9. What are the important problem types?
10. Define algorithmic Strategy (or) Algorithmic Technique.
11. What are the various algorithm strategies (or) algorithm Techniques?
12. What are the ways to specify an algorithm?
13. Define Best case Time Complexity .
14. Define Worst case Time Complexity.
15. Define Average case time complexity.
16. What are the Basic Efficiency Classes.
17. Define Asymptotic Notation.
18. How to calculate the GCD value?

PART-B

1. (a) Describe the steps in analyzing & coding an algorithm. (10)
(b) Explain some of the problem types used in the design of algorithm. (6)
2. (a) Discuss the fundamentals of analysis framework . (10)
(b) Explain the various asymptotic notations used in algorithm design. (6)
3. (a) Explain the general framework for analyzing the efficiency of algorithm. (8)
(b) Explain the various asymptotic efficiencies of an algorithm. (8)
4. (a) Explain the basic efficiency classes. (10)
(b) Explain briefly the concept of algorithmic strategies. (6)
5. Describe briefly the notions of complexity of an algorithm. (16)
6. (a) What is Pseudo-code? Explain with an example. (8)
(b) Find the complexity $C(n)$ of the algorithm for the worst case, best case and average case.(Evaluate average case complexity for $n=3$,Where n is the number of inputs) (8)

Divide and Conquer

Divide-and-conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these sub-problems into an overall solution. In many cases, it can be a simple and powerful method.

Analyzing the running time of a divide-and-conquer algorithm generally involves solving a *recurrence relation* that bounds the running time recursively in terms of the running time on smaller instances. We begin the chapter with a general discussion of recurrence relations, illustrating how they arise in the analysis and describing methods for working out upper bounds from them.

We then illustrate the use of divide-and-conquer with applications to a number of different domains: computing a distance function on different rankings of a set of objects; finding the closest pair of points in the plane; multiplying two integers; and smoothing a noisy signal. Divide-and-conquer will also come up in subsequent chapters, since it is a method that often works well when combined with other algorithm design techniques; for example, we will see it combined with dynamic programming to produce a space-efficient solution to sequence alignment, and combined with randomization to yield a simple and efficient algorithm for computing the median of a set of numbers.

One thing to note about many settings in which divide-and-conquer is applied, including these, is that the natural brute-force algorithm may already be polynomial-time, and the divide-and-conquer strategy is serving to reduce the running time to a lower polynomial. This is in contrast to most of the problems in the previous chapter, for example, where brute-force was exponential and the goal in designing a more sophisticated algorithm was to achieve *any* kind of polynomial running time. For example, we discussed in Chapter 2 that the natural brute-force algorithm for finding the closest pair among n points in the plane would simply measure all $\Theta(n^2)$ distances, for a (polynomial) running time of $\Theta(n^2)$. Using divide-and-conquer, we will improve the running time to $O(n \log n)$. At a high level, then, the overall theme of this chapter is the same as what we've been seeing earlier: that

improving on brute-force search is a fundamental conceptual hurdle in solving a problem efficiently, and the design of sophisticated algorithms can achieve this. The difference is simply that the distinction between brute-force search and an improved solution here will not always be the distinction between exponential and polynomial.

A First Recurrence: The Mergesort Algorithm

To motivate the general approach to analyzing divide-and-conquer algorithms, we begin with the *Mergesort* algorithm. We discussed the *Mergesort* algorithm briefly in Chapter 2, when we surveyed common running times for algorithms. *Mergesort* sorts a given list of numbers by first dividing them into two equal halves, sorting each half separately by recursion, and then combining the results of these recursive calls — in the form of the two sorted halves — using the linear-time algorithm for merging sorted lists that we saw in Chapter 2.

To analyze the running time of *Mergesort*, we will abstract its behavior into the following template, which describes many common divide-and-conquer algorithms:

- (†) Divide the input into two pieces of equal size; solve the two sub-problems on these pieces separately by recursion; and then combine the two results into an overall solution, spending only linear time for the initial division and final re-combining.

In *Mergesort*, as in any algorithm that fits this style, we also need a base case for the recursion, typically having it “bottom out” on inputs of some constant size. In the case of *Mergesort*, we will assume that once the input has been reduced to size 2, we stop the recursion and sort the two elements by simply comparing them to each other.

Consider any algorithm that fits the pattern in (†), and let $T(n)$ denote its worst-case running time on input instances of size n . Supposing that n is even, the algorithm spends $O(n)$ time to divide the input into two pieces of size $n/2$ each; it then spends time $T(n/2)$ to solve each one (since $T(n/2)$ is the worst-case running time for an input of size $n/2$); and finally it spends $O(n)$ time, via the linear-time merging technique, to combine the solutions from the two recursive calls. Thus, the running time $T(n)$ satisfies the following *recurrence relation*:

For some constant c ,

$$T(n) \leq 2T(n/2) + cn$$

when $n > 2$, and

$$T(2) \leq c.$$

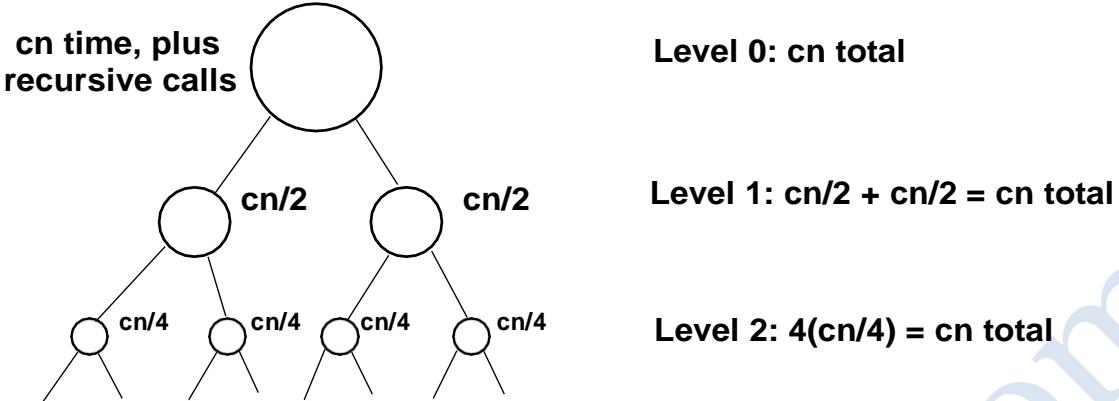


Figure 5.1: Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

The structure of (5.1) is typical of what recurrences will look like: there's an inequality or equation that bounds $T(n)$ in terms of an expression involving $T(k)$ for smaller values k ; and there is a base case that generally says that $T(n)$ is equal to a constant when n is a constant. Note that one can also write (5.1) more informally as $T(n) \leq 2T(n/2) + O(n)$, suppressing the constant c ; however, it is generally useful to make c explicit when analyzing the recurrence.

To keep the exposition simpler, we will generally assume that parameters like n are even when needed. This is somewhat imprecise usage; without this assumption, the two recursive calls would be on problems of size $|n/2|$ and $|n/2\rceil$, and the recurrence relation would say that

$$T(n) \leq T(|n/2|) + T(|n/2\rceil) + cn$$

for $n \geq 2$. Nevertheless, for all the recurrences we consider here (and for most that arise in practice), the asymptotic bounds are not affected by the decision to ignore all the floors and ceilings, and it makes the symbolic manipulation much cleaner.

Now, (5.1) does not explicitly provide an asymptotic bound on the growth rate of the function T ; rather, it specifies $T(n)$ implicitly in terms of its values on smaller inputs. To obtain an explicit bound, we need to solve the recurrence relation so that T appears only on the left-hand-side of the inequality, not the right-hand-side as well.

Recurrence-solving is a task that has been incorporated into a number of standard computer algebra systems, and the solution to many standard recurrences can now be found by automated means. It is still useful, however, to understand the process of solving recurrences and to recognize which recurrences lead to good running times, since the design of an efficient divide-and-conquer algorithm is heavily intertwined with an understanding of how a recurrence relation determines a running time.

Approaches to solving recurrences

There are two basic ways one can go about solving a recurrence, each of which we describe in more detail below.

- (1) The most intuitively natural way to search for a solution to a recurrence is to “unroll” the recursion, accounting for the running time across the first few levels, and identify a pattern that can be continued as the recursion expands. One then sums the running times over all levels of the recursion (i.e. until it “bottoms out” on sub-problems of constant size), and thereby arrives at a total running time.
- (2) A second way is to start with a guess for the solution, substitute it into the recurrence relation, and check that it works. Formally, one justifies this plugging-in using an argument by induction on n . There is a useful variant of this method in which one has a general form for the solution, but does not have exact values for all the parameters; by leaving these parameters unspecified in the substitution, one can often work them out as needed.

We now discuss each of these approaches, using the recurrence in (5.1) as an example. It is often useful to start with the first approach, which is easier to do informally, and thereby get a sense for what the form of the solution of the recurrence will be. Once one has this, it can be used as the starting point for the second approach.

Unrolling the Mergesort recurrence

Let’s start with the first approach to solving the recurrence in (5.1). The basic argument is depicted in Figure 5.1.

- *Analyzing the first few levels:* At the first level of recursion, we have a single problem of size n , which takes time at most cn plus the time spent in all subsequent recursive calls. At the next level, we have two problems size $n/2$ each. Each of these takes time at most $cn/2$, for a total of at most cn , again plus the time in subsequent recursive calls. At the third level, we have four problems each of size $n/4$, each taking time at most $cn/4$, for a total of at most cn .
- *Identifying a pattern:* What’s going on in general? At level j of the recursion, the number of sub-problems has doubled j times, so there are now a total of 2^j . Each has correspondingly shrunk in size by a factor of two j times, and so each has size $n/2^j$ and hence each takes time at most $cn/2^j$. Thus level j contributes a total of at most $2^j(cn/2^j) = cn$ to the total running time.
- *Summing over all levels of recursion:* We’ve found that the recurrence in (5.1) has the property that the same upper bound of cn applies to total amount of performed at

each level. The number of times the input must be halved in order to reduce its size from n to 2 is $\log_2 n$. So summing the cn work over $\log n$ levels of recursion, we get a total running time of $O(n \log n)$.

We summarize this in the following claim.

Any function $T(\cdot)$ satisfying (5.1) is bounded by $O(n \log n)$, when n

Substituting a solution into the Mergesort recurrence

The argument above can be used to determine that the function $T(n)$ is bounded by $O(n \log n)$. If, on the other hand, we have a guess for the running time that we want to verify, we can do so by plugging it into the recurrence as follows.

Suppose we believe that $T(n) \leq cn \log_2 n$ for all $n \geq 2$, and we want to check whether this is indeed true. This clearly holds for $n = 2$, since in this case $cn \log_2 n = c$, and (5.1) explicitly tells us that $T(2) \leq c$. Now suppose, by induction, that $T(m) \leq cm \log_2 m$ for all values of m less than n , and we want to establish this for $T(n)$. We do this by writing the recurrence for $T(n)$ and plugging in the inequality $T(n/2) \leq c(n/2) \log_2(n/2)$. We then simplify the resulting expression by noticing that $\log_2(n/2) = (\log_2 n) - 1$. The full calculation looks as follows:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2c(n/2) \log_2(n/2) + cn \\ &= cn[(\log_2 n) - 1] + cn \\ &= (cn \log_2 n) - cn + cn \\ &= cn \log_2 n. \end{aligned}$$

This establishes the bound we want for $T(n)$ assuming it holds for smaller values $m < n$, and thus it completes the induction argument.

An approach using partial substitution

There is a somewhat weaker kind of substitution one can do, in which one guesses the overall form of the solution without pinning down the exact values of all the constants and other parameters at the outset.

Specifically, suppose we believe that $T(n) = O(n \log n)$, but we're not sure of the constant inside the $O(\cdot)$ notation. We can use the substitution method even without being sure of this constant as follows. We first write $T(n) \leq kn \log_b n$ for some constant k and base b that we'll determine later. (Actually, the base and the constant we'll end up needing are related to each other, since we saw in Chapter 2 that one can change the base of the logarithm by simply changing the multiplicative constant in front.)

Now, we'd like to know whether's any choice of k and b that will work in an inductive argument. So we try out one level of the induction as follows:

$$T(n) \leq 2T(n/2) + cn \leq 2k(n/2) \log_b(n/2) + cn.$$

It's now very tempting to choose the base $b = 2$ for the logarithm, since we see that this will let us apply the simplification $\log_2(n/2) = (\log_2 n) - 1$. Proceeding with this choice, we have

$$\begin{aligned} T(n) &\leq 2k(n/2) \log_2(n/2) + cn \\ &= 2k(n/2)[(\log_2 n) - 1] + cn \\ &= kn[(\log_2 n) - 1] + cn \\ &= (kn \log_2 n) - kn + cn. \end{aligned}$$

Finally, we ask: is there a choice of k that will cause this last expression to be bounded by $kn \log_2 n$? The answer is clearly yes; we just need to choose any k that is at least as large as c , and we get

$$T(n) \leq (kn \log_2 n) - kn + cn \leq kn \log_2 n,$$

which completes the induction.

Thus, the substitution method can actually be useful in working out the exact constants when one has some guess of the general form of the solution.

Further Recurrence Relations

We've just worked out the solution to a recurrence relation, (5.1), that will come up in the design of several divide-and-conquer algorithms later in this chapter. As a way to explore this issue further, we now consider a class of recurrence relations that generalizes (5.1), and show how to solve the recurrences in this class. Other members of this class will arise in the design of algorithms both in this chapter and in later chapters.

This more general class of algorithms is obtained by considering divide-and-conquer algorithms that create recursive calls on q sub-problems of size $n/2$ each, and then combine the results in $O(n)$ time. This corresponds to the *Mergesort* recurrence (5.1) when $q = 2$ recursive calls are used, but other algorithms find it useful to spawn $q > 2$ recursive calls, or just a single ($q = 1$) recursive call. In fact, we will see the case $q > 2$ later in this chapter when we design algorithms for integer multiplication; and we will see a variant on the case $q = 1$ much later in the book, when we design a randomized algorithm for median-finding.

If $T(n)$ denotes the running time of an algorithm designed in this style, then $T(n)$ obeys the following recurrence relation, which directly generalizes (5.1) by replacing 2 with q :

(5.3) *For some constant c ,*

$$T(n) \leq qT(n/2) + cn$$

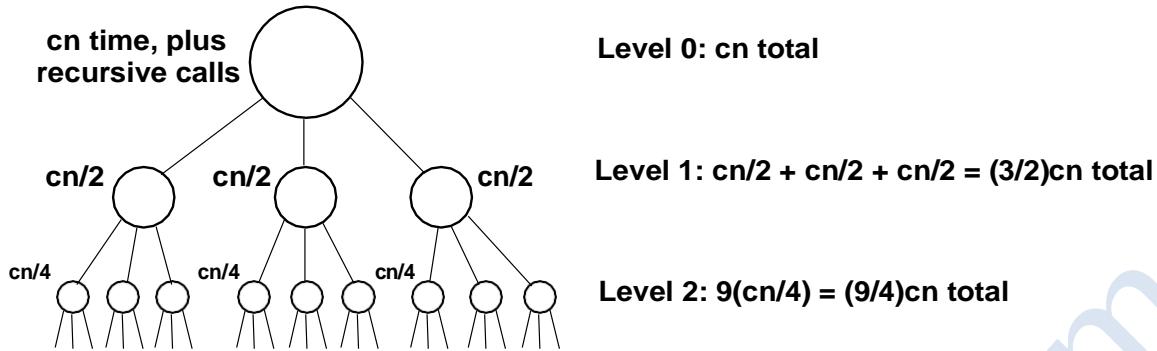


Figure 5.2: Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

when $n > 2$, and

$$T(2) \leq c.$$

We now describe how to solve (5.3) by the methods we've seen above: unrolling, substitution, and partial substitution. We treat the cases $q > 2$ and $q = 1$ separately, since they are qualitatively different from each other — and different from the case $q = 2$ as well.

The case of $q > 2$ sub-problems

We begin by unrolling (5.3) in the case $q > 2$, following the style we used earlier for (5.1) . We will see that the punch-line ends up being quite different.

- *Analyzing the first few levels:* We show an example of this for the case $q = 3$ in Figure 5.2. At the first level of recursion, we have a single problem of size n , which takes time at most cn plus the time spent in all subsequent recursive calls. At the next level, we have q problems each of size $n/2$ each. Each of these takes time at most $cn/2$, for a total of at most $(q/2)cn$, again plus the time in subsequent recursive calls. The next level yields q^2 problems of size $n/4$ each, for a total time of $(q^2/4)n$. Since $q > 2$, we see that the total work per level is *increasing* as we proceed through the recursion.
- *Identifying a pattern:* At an arbitrary level j , we have q^j distinct instances, each of size $n/2^j$. Thus, the total work performed at level j is $q^j(cn/2^j) = (q/2)^j cn$.
- *Summing over all levels of recursion:* As before, there are $\log_2 n$ levels of recursion, and the total amount of work performed is the sum over all these:

$$T(n) \leq \sum_{j=0}^{\log_2 n} \frac{q^j}{2} \sum cn = cn \sum_{j=0}^{\log_2 n} \frac{q^j}{2}.$$

This is a geometric sum, consisting of the first $\log_2 n$ powers of $r = q/2$. We can use the formula for a geometric sum when $r > 1$, which gives us the formula

$$T(n) \leq cn \cdot \frac{r^{1+\log_2 n} - 1}{r - 1} \leq cn \cdot \frac{r^{1+\log_2 n}}{r - 1}.$$

Since we're aiming for an asymptotic upper bound, it is useful to figure out what's simply a constant; we can pull out one factor of r from the numerator, and the factor of $r-1$ from the denominator, both of which are constants, and write the last expression as

$$T(n) \leq \frac{r}{r-1} \sum cn r^{\log_2 n}.$$

Finally, we need to figure out what $r^{\log_2 n}$ is. Here we use a very handy identity, which says that for any $a > 1$ and $b > 1$ we have $a^{\log b} = b^{\log a}$. Thus,

$$r^{\log_2 n} = n^{\log_2 r} = n^{\log_2(q/2)} = n^{(\log_2 q)-1}.$$

Thus, we have

$$T(n) \leq \frac{r}{r-1} \sum cn \cdot n^{(\log_2 q)-1} \leq \frac{r}{r-1} \sum cn^{\log_2 q} = O(n^{\log_2 q}).$$

So we find that the running time is more than linear, since $\log_2 q > 1$, but still polynomial in n . Plugging in specific values of q , the running time is $O(n^{\log_2 3}) = O(n^{1.59})$ when $q = 3$; and the running time is $O(n^{\log_2 4}) = O(n^2)$ when $q = 4$. This increase in running time as q increases makes sense, of course, since the recursive calls generate more work for larger values of q .

Applying partial substitution. The appearance of $\log_2 q$ in the exponent followed naturally from our solution to (5.3), but it's not necessarily an expression one would have guessed at the outset. We now consider how an approach based on partial substitution into the recurrence yields a different way of discovering this exponent.

Suppose we guess that the solution to (5.3), when $q > 2$, has the form $T(n) \leq kn^d$ for some constants $k > 0$ and $d > 1$. This is quite a general guess, since we haven't even tried specifying the exponent d of the polynomial. Now, let's try starting the inductive argument and seeing what constraints we need on k and d . We have

$$T(n) \leq qT(n/2) + cn,$$

and applying the inductive hypothesis to $T(n/2)$, this expands to

$$\begin{aligned} T(n) &\leq qk \frac{n^d}{2^d} + cn \\ &= \frac{q}{2^d} kn^d + cn. \end{aligned}$$

This is remarkably close to something that works: if we choose d so that $q/2^d = 1$, then we have $T(n) \leq kn^d + cn$, which is almost right except for the extra term cn . So let's deal with these two issues: first, how to choose d so we get $q/2^d = 1$; and second, how to get rid of the cn term.

Choosing d is easy: we want $2^d = q$, and so $d = \log_2 q$. Thus, we see that the exponent $\log_2 q$ appears very naturally once we decide to discover which value of d works when substituted into the recurrence.

But we still have to get rid of the cn term. To do this, we change the form of our guess for $T(n)$ so as to explicitly subtract it off: suppose we try the form $T(n) \leq kn^d - An$, where we've now decided that $d = \log_2 q$ but we haven't fixed the constants k or A . Applying the new formula to $T(n/2)$, this expands to

$$\begin{aligned} T(n) &\leq qk \sum_{i=0}^{n-1} \frac{n}{2^i} - qA \sum_{i=0}^{n-1} \frac{n}{2^i} + cn \\ &= \frac{q}{2^d} kn^d - \frac{qA}{2^d} n + cn \\ &= kn^d - \frac{qA}{2^d} n + cn \\ &= kn^d - \left(\frac{qA}{2^d} - c\right)n. \end{aligned}$$

This now works completely, if we simply choose A so that $(\frac{q^A}{2} - c) = A$: in other words, $A = 2c/(q - 2)$. This completes the inductive step for n . We also need to handle the base case $n = 2$, and this we do using the fact that the value of k has not yet been fixed: we choose k large enough so that the formula is a valid upper bound for the case $n = 2$.

The case of one sub-problem

We now consider the case of $q = 1$ in (5.3), since this illustrates an outcome of yet another flavor. While we won't see a direct application of the recurrence for $q = 1$ in this chapter, a variation on it comes up in Chapter 13 as we mentioned earlier.

We begin by unrolling the recurrence to try constructing a solution.

- *Analyzing the first few levels:* We show the first few levels of the recursion in Figure 5.3. At the first level of recursion, we have a single problem of size n , which takes time at most cn plus the time spent in all subsequent recursive calls. The next level has one problem of size $n/2$, which contributes $cn/2$, and the level after that has one problem of size $n/4$, which contributes $cn/4$. So we see that unlike the previous case, the total work per level when $q = 1$ is actually *decreasing* as we proceed through the recursion.
- *Identifying a pattern:* At an arbitrary level j , we still have just one instance; it has size $n/2^j$ and contributes $cn/2^j$ to the running time.

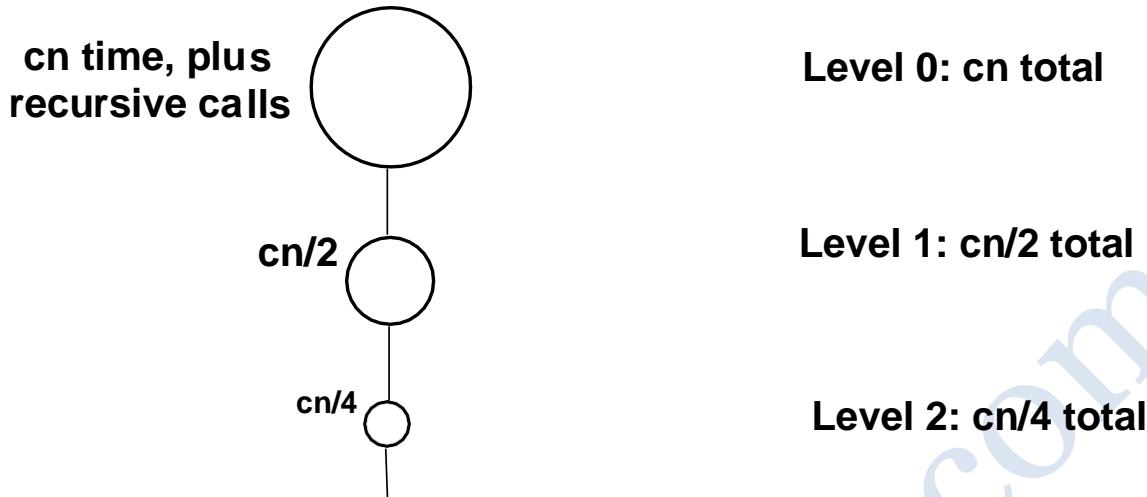


Figure 5.3: Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

- *Summing over all levels of recursion:* There are $\log_2 n$ levels of recursion, and the total amount of work performed is the sum over all these:

$$T(n) \leq \sum_{j=0}^{\log_2 n} \frac{cn}{2^j} = cn \sum_{j=0}^{\log_2 n} \frac{1}{2^j}.$$

This geometric sum is very easy to work out; even if we continued it to infinity, it would converge to 2. Thus we have

$$T(n) \leq 2cn = O(n).$$

This is counter-intuitive when you first see it: the algorithm is performing $\log n$ levels of recursion, but the overall running time is still linear in n . The point is that a geometric series with a decaying exponent is a powerful thing: fully half the work performed by the algorithm is being done at the top level of the recursion.

It is also useful to see how partial substitution into the recurrence works very well in this case. Suppose we guess, as before, that the form of the solution is $T(n) \leq kn^d$. We now try to establish this by induction using (5.3), assuming that the solution holds for the smaller value $n/2$:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right)_d + cn \\ &\quad - \sum_{j=0}^{\log_2 n} + cn \\ &\leq k \frac{n^{2^d}}{2^d} + cn. \end{aligned}$$

If we now simply choose $d = 1$ and $k = 2c$, we have

$$T(n) \leq \frac{k}{2} n + cn = \left(\frac{k}{2} + c\right)n = kn,$$

which completes the induction.

The effect of the parameter q . It is worth reflecting briefly on the role of the parameter q in the class of recurrences $T(n) \leq qT(n/2) + O(n)$ defined by (5.3). When $q = 1$ the resulting running time is linear; when $q = 2$, it's $O(n \log n)$; and when $q > 2$ it's a polynomial bound with an exponent larger than 1 that grows with q . The reason for this range of different running times lies in where most of the work is spent in the recursion: when $q = 1$ the total running time is dominated by the top level, whereas when $q > 2$ it's dominated by the work done on constant-size sub-problems at the bottom of the recursion. Viewed this way, we can appreciate that the recurrence for $q = 2$ really represents a “knife-edge” — the amount of work done at each level is *exactly the same*, which is what yields the $O(n \log n)$ running time.

A Related Recurrence: $T(n) \leq 2T(n/2) + O(n^2)$

We conclude our discussion with one final recurrence relation; it is illustrative both as another application of a decaying geometric sum, and as an interesting contrast with the recurrence (5.1) that characterized *Mergesort*. Moreover, we will see a close variant of it in Chapter 6, when we analyze a divide-and-conquer algorithm for solving the sequence alignment problem using a small amount of working memory.

The recurrence is based on the following divide-and-conquer structure:

Divide the input into two pieces of equal size; solve the two sub-problems on these pieces separately by recursion; and then combine the two results into an overall solution, spending quadratic time for the initial division and final re-combining.

For our purposes here, we note that this style of algorithm has a running time $T(n)$ that satisfies the following recurrence.

(5.4) *For some constant c ,*

$$T(n) \leq 2T(n/2) + cn^2$$

when $n > 2$, and

$$T(2) \leq c.$$

One's first reaction is to guess that the solution will be $T(n) = O(n^2 \log n)$, since it looks almost identical to (5.1) except that the amount of work per level is larger by a factor equal to the input size. In fact, this upper bound is correct (it would need a more careful argument than what's in the previous sentence), but it will turn out that we can also show a stronger upper bound.

We'll do this by unrolling the recurrence, following the standard template for doing this.

- *Analyzing the first few levels:* At the first level of recursion, we have a single problem of size n , which takes time at most cn^2 plus the time spent in all subsequent recursive calls. At the next level, we have two problems each of size $n/2$ each. Each of these takes time at most $c(n/2)^2 = cn^2/4$, for a total of at most $cn^2/2$, again plus the time in subsequent recursive calls. At the third level, we have four problems each of size $n/4$, each taking time at most $c(n/4)^2 = cn^2/16$, for a total of at most $cn^2/4$. Already we see that something is different from our solution to the analogous recurrence (5.1); whereas the total amount of work per level remained the same in that case, here it's decreasing.
- *Identifying a pattern:* At an arbitrary level j of the recursion, there are 2^j sub-problems, each of size $n/2^j$, and hence the total work at this level is bounded by $2^j c(\frac{n}{2^j})^2 = cn^2/2^j$.
- *Summing over all levels of recursion:* Having gotten this far in the calculation, we've arrived at almost exactly the same sum that we had for the case $q = 1$ in the previous recurrence. We have

$$T(n) \leq \sum_{j=0}^{\log_2 n} \frac{cn^2}{2^j} = cn^2 \sum_{j=0}^{\log_2 n} \frac{1}{2^j} \leq 2cn^2 = O(n^2).$$

where the second inequality follows from the fact that we have a convergent geometric sum.

In retrospect, our initial guess of $T(n) = O(n^2 \log n)$, based on the analogy to (5.1), was an overestimate because of how quickly n^2 decreases as we replace it with $(\frac{n}{2})^2$, $(\frac{n}{4})^2$, $(\frac{n}{8})^2$, and so forth in the unrolling of the recurrence. This means that we get a geometric sum, rather than one that grows by a fixed amount over all n levels (as in the solution to (5.1)).

Counting Inversions

We've spent some time discussing approaches to solving a number of common recurrences. The remainder of the chapter will illustrate the application of divide-and-conquer to problems from a number of different domains; we will use what we've seen in the previous sections to bound the running times of these algorithms. We begin by showing how a variant of the *Mergesort* technique can be used to solve a problem that is not directly related to sorting numbers.

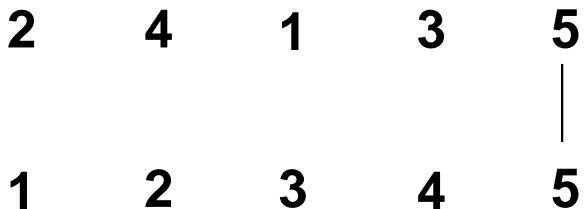


Figure 5.4: Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list — in other words, an inversion.

The Problem

We will consider a problem that arises in the analysis of *rankings*, which are becoming important to a number of current applications. For example, a number of sites on the Web make use of a technique known as *collaborative filtering*, in which they try to match your preferences (for books, movies, restaurants) with those of other people out on the Internet. Once the Web site has identified people with “similar” tastes to yours — based on a comparison of how you and they rate various things — it can recommend new things that these other people have liked. Another application arises in *meta-search tools* on the Web, which execute the same query on many different search engines, and then try to synthesize the results by looking for similarities and differences among the various rankings that the search engines return.

A core issue in applications like this is the problem of comparing two rankings. You rank a set of n movies, and then a collaborative filtering system consults its database to look for other people who had “similar” rankings. But what’s a good way to measure, numerically, how similar two people’s rankings are? Clearly an identical ranking is very similar, and a completely reversed ranking is very different; we want something that interpolates through the middle region.

Let’s consider comparing your ranking and a stranger’s ranking of the same set of n movies. A natural method would be to label the movies from 1 to n according to your ranking, then order these labels according to the stranger’s ranking, and see how many pairs are “out of order.” More concretely, we will consider the following problem. We are given a sequence of n numbers a_1, \dots, a_n ; we will assume that all the numbers are distinct. We want to define a measure that tells us how far this list is from being in ascending order; the value of the measure should be 0 if $a_1 < a_2 < \dots < a_n$, and should increase as the numbers become more scrambled.

A natural way to quantify this notion is by counting the number of *inversions*. We say that two indices $i < j$ form an inversion if $a_i > a_j$, i.e., if the two elements a_i and a_j are “out of order.” We will seek to determine the number of inversions in the sequence a_1, \dots, a_n .

Just to pin down this definition, consider an example in which the sequence is 2, 4, 1, 3, 5. There are three inversions in this sequence: (2, 1), (4, 1), and (4, 3). There is also an appealing geometric way to visualize the inversions, pictured in Figure 5.4: we draw the sequence of input numbers in the order they're provided, and below that in ascending order. We then draw a line segment between each number in the top list and its copy in the lower list. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the two lists — in other words, an inversion.

Note how the number of inversions is a measure that smoothly interpolates between complete agreement (when the sequence is in ascending order, then there are no inversions) and complete disagreement: if the sequence is in descending order then every pair forms an inversion, and so there are $n(n - 1)/2$ of them.

Designing and Analyzing the Algorithm

What is the simplest algorithm to count inversions? Clearly, we could look at every pair of numbers (a_i, a_j) and determine whether they constitute an inversion; this would take $O(n^2)$ time.

We now show how to count the number of inversions much more quickly, in $O(n \log n)$ time. Note that since there can be a quadratic number of inversions, such an algorithm must be able to compute the total number without ever *looking* at each inversion individually. The basic idea is to follow the strategy (\dagger) defined above. We set $m = \lfloor n/2 \rfloor$ and divide the list into the two pieces a_1, \dots, a_m and a_{m+1}, \dots, a_n . We first count the number of inversions in each of these two halves separately. Then, we count the number of inversions (a_i, a_j) , where the two numbers belong to different halves; the trick is that we must do this part in $O(n)$ time, if we want to apply (5.2). Note that these first-half/second-half inversions have a particularly nice form: they are precisely the pairs (a_i, a_j) where a_i is in the first half, a_j is in the second half, and $a_i > a_j$.

To help with counting the number of inversions between the two halves, we will make the algorithm recursively sort the numbers in the two halves as well. Having the recursive step do a bit more work (sorting as well as counting inversions) will make the “combining” portion of the algorithm easier.

So the crucial routine in this process is Merge-and-Count. Suppose we have recursively sorted the first and second halves of the list, and counted the inversions in each. We now have two sorted lists A and B , containing the first and second halves respectively. We want to produce a single sorted list C from their union, while also counting the number of pairs (a, b) with $a \in A$, $b \in B$, and $a > b$. By our discussion above, this is precisely what we will need for the “combining” step that computes the number of first-half/second-half inversions.

This is closely related to the simpler problem we discussed in Chapter 2, which formed the corresponding “combining” step for *Mergesort*: there, we had two sorted lists A and B ,

and we wanted to merge them into a single sorted list in $O(n)$ time. The different here is that we want to do something extra: not only should we produce a single sorted list from A and B , but we should also count the number of “inverted pairs” (a, b) where $a \in A$, $b \in B$, and $a > b$.

It turns out that we will be able to do this in very much the same style that we used for merging. Our Merge-and-Count routine will walk through the sorted lists A and B , removing elements from the front and appending them to the sorted list C . In a given step, we have a *Current* pointer into each list, showing our current position. Suppose that these pointers are currently at elements a_i and b_j . In one step, we compare the elements a_i and b_j being pointed to in each list, remove the smaller one from its list, and append it to the end of list C .

This takes care of merging; how do we also count the number of inversions? Because A and B are sorted, it is actually very easy to keep track of the number of inversions we encounter. Every time the element a_i is appended to C , no new inversions are encountered — since a_i is smaller than everything left in list B , and it comes before all of them. On the other hand, if b_j is appended to list C , then it is smaller than all the remaining items in A , and it comes after all of them — so we increase our count of the number of inversions by the number of elements remaining in A . This is the crucial idea: in constant time, we have accounted for a potentially large number of inversions. See Figure 5.5 for an illustration of this process.

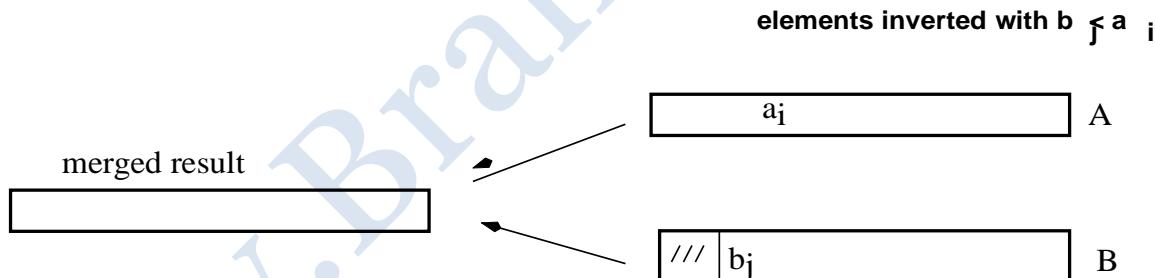


Figure 5.5: Merging the first half of the list A with the second half B

To summarize, we have the following algorithm.

Merge-and-Count(A, B)

Maintain a *Current* pointer into each list, initialized to point to the front elements.

Maintain a variable *Count* for the number of inversions, initialized to 0.

While both lists are non-empty:

Let a_i and b_j be the elements pointed to by the *Current* pointer
Append the smaller of these two to the output list

If b_j is the smaller element then

 Increment **Count** by the number of elements remaining in A

 Advance the **Current** pointer in the list from which the
 smaller element was selected.

EndWhile

Now that one list is empty, append the remainder of the other list
 to the output

The running time of Merge-and-Count can be bounded by the analogue of the argument we used for the original merging algorithm at the heart of *Mergesort*: each iteration of the While loop takes constant time, and in each iteration we add some element to the output that will never be seen again. Thus, the number of iterations can be at most the sum of the initial lengths of A and B , and so the total running time is $O(n)$.

We use this Merge-and-Count routine in a recursive procedure that simultaneously sorts and counts the number of inversions in a list L .

Sort-and-Count(L)

 If the list has one element then
 there are no inversions

 Else

 Divide the list into two halves:

A contains the first $|n/2|$ elements.

B contains the remaining $|n/2\rangle$ elements.

$(r_A, A) = \text{Sort-and-Count}(A)$

$(r_B, B) = \text{Sort-and-Count}(B)$

$(r, L) = \text{Merge-and-Count}(A, B)$

 Endif

 Return $r = r_A + r_B + r$, and the sorted list L

Since our Merge-and-Count procedure takes $O(n)$ time, the running time $T(n)$ of the full Sort-and-Count procedure satisfies the recurrence (5.1). By (5.2), we have

(5.5) *The Sort-and-Count algorithm correctly sorts the input list and counts the number of inversions; it runs in $O(n \log n)$ time for a list with n elements.*

Finding the Closest Pair of Points

The Problem

We now describe another problem that can be solved by an algorithm in the style we've been discussing; but finding the right way to "merge" the solutions to the two sub-problems it generates requires quite a bit of ingenuity. The problem itself is very simple to state: given n points in the plane, find the pair that is closest together.

The problem was considered by M.I. Shamos and D. Hoey in the early 1970's, as part of their project to work out efficient algorithms for basic computational primitives in geometry. These algorithms formed the foundations of the then-fledgling field of *computational geometry*, and they have found their way into areas such as graphics, computer vision, geographic information systems, and molecular modeling. And although the closest-pair problem is one of the most natural algorithmic problems in geometry, it is surprisingly hard to find an efficient algorithm for it. It is immediately clear that there is an $O(n^2)$ solution — compute the distance between each pair of points and take the minimum — and so Shamos and Hoey asked whether an algorithm asymptotically faster than quadratic could be found. It took quite a long time before they resolved this question, and the $O(n \log n)$ algorithm we give below is essentially the one they discovered.

Designing the Algorithm

We begin with a bit of notation. Let us denote the set of points by $P = \{p_1, \dots, p_n\}$, where p_i has coordinates (x_i, y_i) ; and for two points $p_i, p_j \in P$, we use $d(p_i, p_j)$ to denote the standard Euclidean distance between them. Our goal is to find the pair of points p_i, p_j which minimizes $d(p_i, p_j)$.

We will assume that no two points in P have the same x -coordinate or the same y -coordinate. This makes the discussion cleaner; and it's easy to eliminate this assumption either by initially applying a rotation to the points that makes it true, or by slightly extending the algorithm we develop here.

It's instructive to consider the one-dimensional version of this problem for a minute, since it is much simpler and the contrasts are revealing. How would we find the closest pair of points on a line? We'd first sort them, in $O(n \log n)$ time, and then we'd walk through the sorted list, computing the distance from each point to the one that comes after it. It is easy to see that one of these distances must be the minimum one.

In two dimensions, we could try sorting the points by their y -coordinate (or x -coordinate), and hoping that the two closest points were near one another in the order of this sorted list. But it is easy to construct examples in which they are very far apart, preventing us from adapting our one-dimensional approach.

Instead, our plan will be to apply the style of divide-and-conquer used in *Mergesort*: we find the closest pair among the points in the "left half" of P and the closest pair among the points in the "right half" of P ; and then we need to use this information to get the overall solution in linear time. If we develop an algorithm with this structure, then the solution of our basic recurrence from (5.1) will give us an $O(n \log n)$ running time.

It is the last, "combining" phase of the algorithm that's tricky: the distances that have not been considered by either of our recursive calls are precisely those that occur between a point in the left half and a point in the right half; there are $\Omega(n^2)$ such distances, yet we

need to find the smallest one in $O(n)$ time after the recursive calls return. If we can do this, our solution will be complete: it will be the smallest of the values computed in the recursive calls and this minimum “left-to-right” distance.

Setting up the Recursion. Let’s get a few easy things out of the way first. It will be very useful if every recursive call, on a set $P^j \subseteq P$, begins with two lists: a list P_{\downarrow}^j in which all the points in P^j have been sorted by increasing x -coordinate, and a list P_j in which all the points in P^j have been sorted by increasing y -coordinate. We can ensure that this remains true throughout the algorithm as follows.

First, before any of the recursion begins, we sort all the points in P by x -coordinate and again by y -coordinate, producing lists P_x and P_y . Attached to each entry in each list is a record of the position of that point in both lists.

The first level of recursion will work as follows, with all further levels working in a completely analogous way. We define Q to be the set of points in the first $|n/2|$ positions of the list P_x (the “left half”) and R to be the set of points in the final $|n/2|$ positions of the list P_x (the “right half”). See Figure 5.6. By a single pass through each of P_x and P_y , in $O(n)$ time, we can create the following four lists: Q_x , consisting of the points in Q sorted by increasing x -coordinate; Q_y , consisting of the points in Q sorted by increasing y -coordinate; and analogous lists R_x and R_y . For each entry of each of these lists, as before, we record the position of the point in both lists it belongs to.

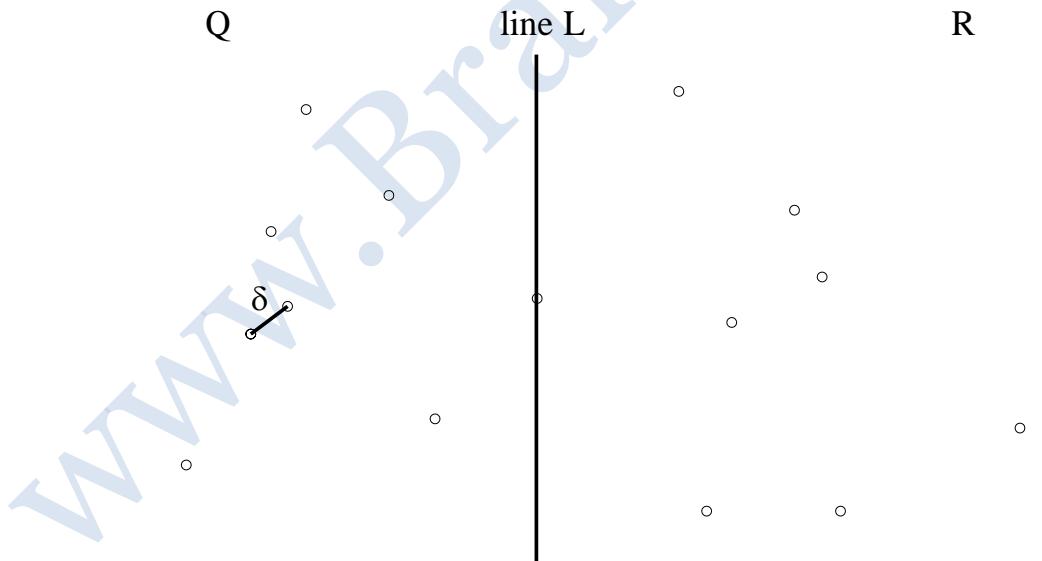


Figure 5.6: The first level of recursion

We now recursively determine the closest pair of points in Q (with access to the lists Q_x and Q_y). Suppose that q_0^* and q_1^* are (correctly) returned as a closest pair of points in Q .

Similarly, we determine the closest pair of points in R , obtaining r^* and r^* .

0 1

Combining the Solutions. The general machinery of divide-and-conquer has gotten us this far, without our really having delved into the structure of the closest-pair problem. But it still leaves us with the problem that we saw looming originally: how do we use the solutions to the two sub-problems as part of a linear-time “combining” operation?

Let δ be the minimum of $d(q^*, q^*)$ and $d(r^*, r^*)$. The real question is: are there points

0 1 0 1

$q \in Q$ and $r \in R$ for which $d(q, r) < \delta$? If not, then we have already found the closest pair in one of our recursive calls. But if there are, then the closest such q and r form the closest pair in P .

Let x^* denote the x -coordinate of the rightmost point in Q , and let L denote the vertical line described by the equation $x = x^*$. This line L “separates” Q from R . Here is a simple fact:

(5.6) *If there exists $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$, then each of q and r lies within a distance δ of L .*

Proof. Suppose such q and r exist; we write $q = (q_x, q_y)$ and $r = (r_x, r_y)$. By the definition of x^* , we know that $q_x \leq x^* \leq r_x$. Then we have

$$x^* - q_x \leq r_x - q_x \leq d(q, r) < \delta$$

and

$$r_x - x^* \leq r_x - q_x \leq d(q, r) < \delta,$$

so each of q and r has an x -coordinate within δ of x^* , and hence lies within distance δ of the line L . ■

So if we want to find a close q and r , we can restrict our search to the narrow band consisting only of points in P within δ of L . Let $S \subseteq P$ denote this set, and let S_y denote the list consisting of the points in S sorted by increasing y -coordinate. By a single pass through the list P_y , we can construct S_y in $O(n)$ time.

We can restate (5.6) as follows, in terms of the set S .

(5.7) *There exist $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$ if and only if there exist $s, s^j \in S$ for which $d(s, s^j) < \delta$.*

It's worth noticing at this point that S might in fact be the whole set P , in which case (5.6) and (5.7) really seem to buy us nothing. But this is actually far from true, as the following amazing fact shows.

(5.8) *If $s, s^j \in S$ have the property that $d(s, s^j) < \delta$, then s and s^j are within 15 positions of each other in the sorted list S_y .*

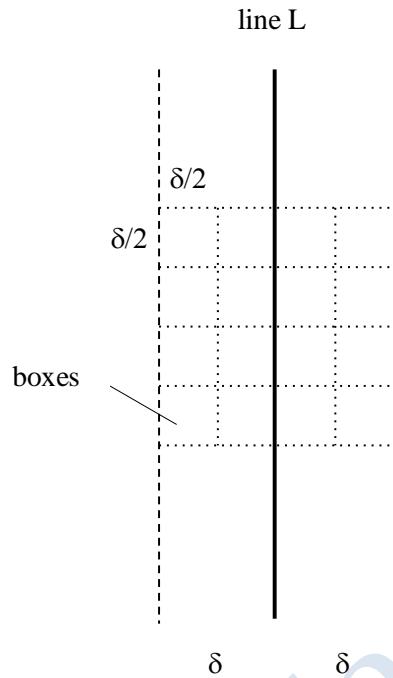


Figure 5.7: There can be at most one point in each box of side length $\delta/2$.

Proof. Consider the subset Z of the plane consisting of all points within distance δ of L . We partition Z into *boxes*: squares with horizontal and vertical sides of length $\delta/2$. One *row* of Z will consist of four boxes whose horizontal sides have the same y -coordinates.

Suppose two points of S lay in the same box. Since all points in this box lie on the same side of L , these two points either both belong to Q or both belong to R . But any two points

in the same box are within distance $\delta \cdot \sqrt{2}/2 < \delta$, which contradicts our definition of δ as the minimum distance between any pair of points in Q or in R . Thus, each box contains at most one point of S .

Now suppose that $s, s^j \in S$ have the property that $d(s, s^j) < \delta$, and that they are at least 16 positions apart in S_y . Assume without loss of generality that s has the smaller y -coordinate. Then since there can be at most one point per box, there are at least three rows of Z lying between s and s^j . But any two points in Z separated by at least three rows must be a distance of at least $3\delta/2$ apart — a contradiction. ■

We note that the value of 15 can be reduced; but for our purposes at the moment, the important thing is that it is an absolute constant.

In view of (5.8), we can conclude the algorithm as follows. We make one pass through S_y , and for each $s \in S_y$, we compute its distance to each of the next 15 points in S_y . (5.8) implies that in doing so, we will have computed the distance of each pair of points in S (if any) that are at distance less than δ from one another. So having done this, we can compare

the smallest such distance to δ , and we can report one of two things: (i) the closest pair of points in S , if their distance is less than δ ; or (ii) the (correct) conclusion that no pairs of points in S are within δ of one another. In case (i), this pair is the closest pair in P ; in case (ii), the closest pair found by our recursive calls is the closest pair in P .

Note the resemblance between this procedure and the algorithm we rejected at the very beginning, which tried to make one pass through P in order of y -coordinate. The reason such an approach works now is due to the extra knowledge (the value of δ) we've gained from the recursive calls, and the special structure of the set S .

This concludes the description of the “combining” part of the algorithm, since by (5.7) we have now determined whether the minimum distance between a point in Q and a point in R is less than δ , and if so, we have found the closest such pair.

A complete description of the algorithm and its proof of correctness are implicitly contained in the discussion so far, but for the sake of concreteness, we now summarize both.

Summary of the Algorithm. A high-level description of the algorithm is the following, using the notation we have developed above.

Closest-Pair(P)

Construct P_x and P_y ($O(n \log n)$ time)
 $(p_0^*, p_1^*) = \text{Closest-Pair-Rec}(P_x, P_y)$

Closest-Pair-Rec(P_x, P_y)

If $|P| \leq 3$ then
 find closest pair by measuring all pairwise distances

Construct Q_x, Q_y, R_x, R_y ($O(n)$ time)
 $(q_0^*, q_1^*) = \text{Closest-Pair-Rec}(Q_x, Q_y)$
 $(r_0^*, r_1^*) = \text{Closest-Pair-Rec}(R_x, R_y)$

$$\delta = \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$$

x^* = maximum x -coordinate of a point in set Q L
 $= \{(x, y) : x = x^*\}$

S = points in P within distance δ of L .

Construct S_y ($O(n)$ time)

For each point $s \in S_y$, compute distance from s
 to each of next 15 points in S_y .

Let s, s^j be pair achieving minimum of these distances
 $(O(n)$ time)

If $d(s, s^j) < \delta$ then

Return (s, s^j)

Else if $d(q^*, q^*) < d(r^*, r^*)$ then
 Return (q_0^*, q_1^*)

Else
 Return (r_0^*, r_1^*)

Analyzing the Algorithm

We first prove that the algorithm produces a correct answer, using the facts we've established in the process of designing it.

(5.9) *The algorithm correctly outputs a closest pair of points in P .*

Proof. As we've noted, all the components of the proof have already been worked out above; so here we just summarize how they fit together.

We prove the correctness by induction on the size of P , the case of $|P| \leq 3$ being clear. For a given P , the closest pair in the recursive calls is computed correctly by induction. By (5.8) and (5.7), the remainder of the algorithm correctly determines whether any pair of points in S is at distance less than δ , and if so returns the closest such pair. Now the closest pair in P either has both elements in one of Q or R , or it has one element in each. In the former case, the closest pair is correctly found by the recursive call; in the latter case, this pair is at distance less than δ , and it is correctly found by the remainder of the algorithm. ■

We now bound the running time as well, using (5.2).

(5.10) *The running time of the algorithm is $O(n \log n)$.*

Proof. The initial sorting of P by x - and y -coordinate takes time $O(n \log n)$. The running time of the remainder of the algorithm satisfies the recurrence (5.1), and hence is $O(n \log n)$ by (5.2). ■

Integer Multiplication

The Problem

We now discuss a different application of divide-and-conquer, in which the “default” quadratic algorithm is improved by means of a different recurrence. The problem we consider is an extremely basic one: the multiplication of two integers.

a) $ \begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} $	b) $ \begin{array}{r} 1100 \\ \times 1101 \\ \hline 1100 \\ 0000 \\ \hline 1100 \\ 1100 \\ \hline 10011100 \end{array} $
--	---

Figure 5.8: The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.

Designing the Algorithm

In a sense, this problem is so basic that one may not initially think of it even as an algorithmic question. But in fact, elementary schoolers are taught a concrete (and quite efficient) algorithm to multiply two n -digit numbers x and y . You first compute a “partial product” by multiplying each digit of y separately by x , and then you add up all the partial products. (Figure 5.8 should help you recall this algorithm. In elementary school we always see this done in base-10, but it works exactly the same way in base-2 as well.) Counting a single operation on a pair of bits as one primitive step in this computation, it takes $O(n)$ time to compute each partial product, and $O(n)$ time to combine it in with the running sum of all partial products so far; since there are n partial products, this is a total running time of $O(n^2)$.

If you haven’t thought about this much since elementary school, there’s something initially striking about the prospect of improving on this algorithm; aren’t all those partial products “necessary” in some way?

But in fact, there are more clever ways to break up the product into partial sums. Let’s assume we’re in base-2 (it doesn’t really matter), and start by writing x as $x_1 \cdot 2^{n/2} + x_0$. In other words, x_1 corresponds to the “high-order” $n/2$ bits, and x_0 corresponds to the “low-order” $n/2$ bits. Similarly, we write $y = y_1 \cdot 2^{n/2} + y_0$. Thus, we have

$$xy = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) = x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0. \quad (5.1)$$

Equation (5.1) reduces the problem of solving a single n -bit instance (multiplying the two n -bit numbers x and y) to the problem of solving four $n/2$ -bit instances (computing the products x_1y_1 , x_1y_0 , x_0y_1 , and x_0y_0). So we have a first candidate for a divide-and-conquer solution: recursively compute the results for these four $n/2$ -bit instances, and then combine them using Equation (5.1). The combining of the solution requires a constant number of

additions of $O(n)$ -bit numbers, so it takes time $O(n)$; thus, the running time $T(n)$ is bounded by the recurrence

$$T(n) \leq 4T(n/2) + cn$$

for a constant c . Is this good enough to give us a sub-quadratic running time?

We can work out the answer by observing that this is just the case $q = 4$ of the class of recurrences in (5.3). As we saw earlier in the chapter, the solution to this is $T(n) \leq O(n^{\log_2 4}) = O(n^2)$.

So in fact, our divide-and-conquer algorithm with four-way branching was just a complicated way to get back to quadratic time! If we want to do better using a strategy that reduces the problem to instances on $n/2$ bits, we should try to get away with only *three* recursive calls; this will lead to the case $q = 3$ of (5.3), which we saw had the solution $T(n) \leq O(n^{\log_2 3}) = O(n^{1.59})$.

Recall that our goal is to compute the expression $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$ in Equation (5.1). It turns out there is a simple trick that lets us determine all three of the terms in this expression using just three recursive calls. The trick is to consider the result of the single multiplication $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$. This has the four products above added together, at the cost of a single recursive multiplication. If we now also determine x_1y_1 and x_0y_0 by recursion, then we get the outermost terms explicitly, and we get the middle term by subtracting x_1y_1 and x_0y_0 away from $(x_1 + x_0)(y_1 + y_0)$.

Thus, in full, our algorithm is

```

Recursive-Multiply(x,y)
  Write x = x1 · 2n/2 + x0.
  y = y1 · 2n/2 + y0.
  Compute x1 + x0 and y1 + y0.
  p = Recursive-Multiply(x1 + x0, y1 + y0)
  x1y1 = Recursive-Multiply(x1, y1)
  x0y0 = Recursive-Multiply(x0, y0)
  Return x1y1 · 2n + (p - x1y1 - x0y0) · 2n/2 + x0y0.

```

Analyzing the Algorithm

Now, we can determine the running time of this algorithm as follows. Given two n -bit numbers, it performs a constant number of additions on $O(n)$ -bit numbers, in addition to the three recursive calls. Ignoring for now the issue that $x_1 + x_0$ and $y_1 + y_0$ may have $n/2 + 1$ bits (rather than just $n/2$), which turns out not to affect the asymptotic results, each of these recursive calls is on an instance of size $n/2$. Thus, in place of our four-way branching recursion, we now have a three-way branching one, with a running time that satisfies

$$T(n) \leq 3T(n/2) + cn$$

for a constant c .

This is the case $q = 3$ of (5.3) that we were aiming for, and using the solution to that recurrence from earlier in the chapter, we have

(5.11) *The running time of Recursive-Multiply on two n -bit factors is $O(n^{\log_2 3}) = O(n^{1.59})$.*

Convolutions and The Fast Fourier Transform

As a final topic in this chapter, we show how our basic recurrence from (5.1) is used in the design of the *Fast Fourier Transform*, an algorithm with a wide range of applications.

The Problem

Given two vectors $a = (a_0, a_1, \dots, a_{n-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$, there are a number of common ways of combining them. For example, one can compute the sum, producing the vector $a + b = (a_0 + b_0, a_1 + b_1, \dots, a_{n-1} + b_{n-1})$; or one can compute the inner product producing the real number $a \cdot b = a_0b_0 + a_1b_1 + \dots + a_{n-1}b_{n-1}$. (For reasons that will emerge shortly, it is useful to write vectors in this section with coordinates that are indexed starting from 0 rather than 1.)

A means of combining vectors that is very important in applications, even if it doesn't always show up in introductory linear algebra courses, is the *convolution* $a * b$. The convolution of two vectors of length n (as a and b are) is a vector with $2n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j): i+j=k \\ i,j < n}} a_i b_j.$$

In otherwords,

$$a * b = (a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, \dots, a_{n-2}b_{n-1} + a_{n-1}b_{n-2}, a_{n-1}b_{n-1}).$$

This definition is a bit hard to absorb when you first see it; another way to think about the convolution is to picture an $n \times n$ table whose (i, j) entry is $a_i b_j$, like this:

$$\begin{array}{ccccc} a_0b_0 & a_0b_1 & \dots & a_0b_{n-2} & a_0b_{n-1} \\ a_1b_0 & a_1b_1 & \dots & a_1b_{n-2} & a_1b_{n-1} \\ a_2b_0 & a_2b_1 & \dots & a_2b_{n-2} & a_2b_{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n-1}b_0 & a_{n-1}b_1 & \dots & a_{n-1}b_{n-2} & a_{n-1}b_{n-1} \end{array}$$

and then to compute the coordinates in the convolution vector by summing along the diagonals.

It's worth mentioning that, unlike the vector sum and inner product, the convolution can be easily generalized to vectors of different lengths, $a = (a_0, a_1, \dots, a_{m-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$. In this more general case, we define $a * b$ to be a vector with $m + n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j): i+j=k \\ i < m, j < n}} a_i b_j.$$

We can picture this using the table of products $a_i b_j$ as before; the table is now rectangular, but we still compute coordinates by summing along the diagonals. (From here on, we'll drop explicit mention of the condition " $i < m, j < n$ " in the summations for convolutions, since it will be clear from context that we only compute the sum over terms that are defined.)

It's not just the definition of a convolution that is a bit hard to absorb at first; the motivation for the definition can also initially be a bit elusive. What are the circumstances where you'd want to compute the convolution of two vectors? In fact, the convolution comes up in a surprisingly wide variety of different contexts; to illustrate this, we mention the following examples here.

- (1) A first example (which also proves that the convolution is something that we all saw implicitly in high school) is polynomial multiplication. Any polynomial $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{m-1}x^{m-1}$, can be represented just as naturally using its vector of coefficients, $a = (a_0, a_1, \dots, a_{m-1})$. Now, given two polynomials $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{m-1}x^{m-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1}$, consider the polynomial $C(x) = A(x)B(x)$ that is equal to their product. In this polynomial $C(x)$, the coefficient on the x^k term is equal to

$$c_k = \sum_{(i,j): i+j=k} a_i b_j.$$

In other words, the coefficient vector c of $C(x)$ is the convolution of the coefficient vectors of $A(x)$ and $B(x)$.

- (2) Arguably the most important application of convolutions in practice is for *signal processing*. This is a topic that could fill an entire course, so we'll just give a simple example here to suggest one way in which the convolution arises.

Suppose we have a vector $a = (a_0, a_1, \dots, a_{m-1})$ which represents a sequence of measurements, such as a temperature or a stock price, sampled at m consecutive points in time. Sequences like this are often very noisy due to measurement error or random fluctuations, and so a common operation is to "smooth" the measurements by averaging each value a_i with a weighted sum of its neighbors within k steps to the left and

right in the sequence, the weights decaying quickly as one moves away from a_i . For example, in *Gaussian smoothing*, one replaces a_i with

$$a_i = \frac{1}{Z} \sum_{j=i-k}^{i+k} a_j e^{-\frac{(j-i)^2}{2}},$$

for some “width” parameter k , and with Z chosen simply to normalize the weights in the average to add up to 1. (There are some issues with boundary conditions — what do we do when $i - k < 0$ or $i + k > m$? — but we could deal with these for example by discarding the first and last k entries from the smoothed signal, or by scaling them differently to make up for the missing terms.)

To see the connection with the convolution operation, we picture this smoothing operation as follows. We first define a “mask”

$$w = (w_{-k}, w_{-(k-1)}, \dots, w_{-1}, w_0, w_1, \dots, w_{k-1}, w_k)$$

consisting of the weights we want to use for averaging each point with its neighbors. (For example, $w = \frac{1}{Z} (e^{-k^2}, e^{-(k-1)^2}, \dots, e_{-1}, 1, e_{-1}, \dots, e^{-(k-1)^2}, e^{-k^2})$ in the Gaussian case above.) We then iteratively position this mask so it is centered at each possible point in the sequence a ; and for each positioning, we compute the weighted average. In other words, we replace a_i with $a_i = \sum_{j=-k}^k w_j a_{i+j}$.

This last expression is essentially a convolution; we just have to warp the notation a bit so that this becomes clear. Let’s define $b = (b_0, b_1, \dots, b_{2k})$ by setting $b_A = w_{k-A}$. Then it’s not hard to check that with this definition, we have the smoothed value

$$a_i = \sum_{(j,A): j+A=i+k} a_j b_A.$$

In other words, the smoothed sequence is just the convolution of the original signal and the reverse of the mask (with some meaningless coordinates at the beginning and end).

- (3) We mention one final application, to the problem of combining histograms. Suppose we’re studying a population of people, and we have the following two histograms: one shows the annual income of all the men in the population, and one shows the annual income of all the women. We’d now like to produce a new histogram, showing for each k the number of pairs (M, W) for which man M and woman W have a combined income of k .

This is precisely a convolution. We can write the first histogram as a vector $a = (a_0, \dots, a_{m-1})$, to indicate that there are a_i men with annual income equal to i . We can similarly write the second histogram as a vector $b = (b_0, \dots, b_{n-1})$. Now, let c_k

denote the number of pairs (m, w) with combined income k ; this is the number of ways of choosing a man with income a_i and a women with income b_j , for any pair (i, j) where $i + j = k$. In other words,

$$c_k = \sum_{(i,j):i+j=k} a_i b_j$$

so the combined histogram $c = (c_0, \dots, c_{m+n-2})$ is simply the convolution of a and b . (Using terminology from probability that we will develop in Chapter 13, one can view this example as showing how convolution is the underlying means for computing the distribution of the sum of two independent random variables.)

Computing the Convolution. Having now motivated the notion of convolution, let's discuss the problem of computing it efficiently. For simplicity, we will consider the case of equal length vectors (i.e. $m = n$), though everything we say carries over directly to the case of vectors of unequal lengths.

Computing the convolution is a more subtle question than it may first appear. The definition of convolution, after all, gives us a perfectly valid way to compute it: for each k , we just calculate the sum

$$\sum_{(i,j):i+j=k} a_i b_j$$

and use this as the value of the k^{th} coordinate. The trouble is that this direct way of computing the convolution involves calculating the product $a_i b_j$ for every pair (i, j) (in the process of distributing over the sums in the different terms) and this is $\Theta(n^2)$ arithmetic operations. Spending $O(n^2)$ time on computing the convolution seems natural, as the definition involves $O(n^2)$ multiplications $a_i b_j$. However, it's not inherently clear that we have to spend quadratic time to compute a convolution, since the input and output both only have size $O(n)$. Could one design an algorithm that bypasses the quadratic-size definition of convolution and computes it in some smarter way?

In fact, quite surprisingly, this is possible; we now describe a method that computes the convolution of two vectors using only $O(n \log n)$ arithmetic operations. The crux of this method is a powerful technique known as the *Fast Fourier Transform (FFT)*. The FFT has a wide range of further applications in analyzing sequences of numerical values; computing convolutions quickly, which we focus on here, is just one of these applications.

Designing and Analyzing the Algorithm

To break through the quadratic time barrier for convolutions, we are going to exploit the connection between the convolution and the multiplication of two polynomials, as illustrated

in the first application discussed above. But rather than use convolution as a primitive in polynomial multiplication, we are going to exploit this connection in the opposite direction.

Suppose we are given the vectors $a = (a_0, a_1, \dots, a_{n-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$. We will view them as the polynomials $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1}$, and we'll seek to compute their product $C(x) = A(x)B(x)$ in $O(n \log n)$ time. If $c = (c_0, c_1, \dots, c_{2n-2})$ is the vector of coefficients of C , then we recall from our earlier discussion that c is exactly the convolution $a * b$, and so we can then read off the desired answer directly from the coefficients of $C(x)$.

Now, rather than multiplying A and B symbolically, we can treat them as functions of the variable x and multiply them as follows.

- (i) First, we choose $2n$ values x_1, x_2, \dots, x_{2n} and evaluate $A(x_j)$ and $B(x_j)$ for each of $j = 1, 2, \dots, 2n$.
- (ii) We can now compute $C(x_j)$ for each j very easily: $C(x_j)$ is simply the product of the two numbers $A(x_j)$ and $B(x_j)$.
- (iii) Finally, we have to recover C from its values on x_1, x_2, \dots, x_{2n} . Here we take advantage of a fundamental fact about polynomials: any polynomial of degree d can be reconstructed from its values on any set of $d+1$ or more points. This is known as *polynomial interpolation*, and we'll discuss the mechanics of performing interpolation in more detail later. For the moment, we simply observe that since A and B each have degree at most $n - 1$, their product C has degree at most $2n - 2$, and so it can be reconstructed from the values $C(x_1), C(x_2), \dots, C(x_{2n})$ that we computed in step (ii).

This approach to multiplying has some promising aspects and some problematic ones. First, the good news: step (ii) requires only $O(n)$ arithmetic operations, since it simply involves the multiplication of $O(n)$ numbers. But the situation doesn't look as hopeful with steps (i) and (iii): in particular, evaluating the polynomials A and B on a single value takes $\Omega(n)$ operations, and our plan calls for performing $2n$ such evaluations. This seems to bring us back to quadratic time right away.

The key idea that will make this all work is to find a set of $2n$ values x_1, x_2, \dots, x_{2n} that are intimately related in some way, such that the work in evaluating A and B on all of them can be shared across different evaluations. A set for which this will turn out to work very well is the *complex roots of unity*.

The Complex Roots of Unity. At this point, we're going to need to recall a few facts about the complex numbers, and their role as solutions to polynomial equations.

Recall that complex numbers can be viewed as lying in the "complex plane," with axes representing their real and imaginary parts; we can write a complex number using polar

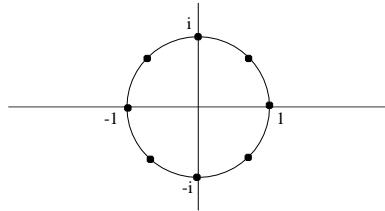


Figure 5.9: The 8th roots of unity on the complex plane.

coordinates with respect to this plane as $re^{\theta i}$, where $e^{\pi i} = -1$ (and $e^{2\pi i} = 1$). Now, for a positive integer k , the polynomial equation $x^k = 1$ has k distinct complex roots, and it is easy to identify them. Each of the complex numbers $\omega_{j,k} = e^{2\pi ji/k}$ (for $j = 1, 2, \dots, k$) satisfies the equation, since

$$(e^{2\pi ji/k})^k = e^{2\pi ji} = (e^{2\pi i})^j = 1^j = 1,$$

and each of these numbers is distinct, so these are all the roots. We refer to these numbers as the k^{th} roots of unity. We can picture these roots as a set of k equally-spaced points lying on the unit circle in the complex plane, as shown in Figure 5.9 for the case $k = 8$.

For our numbers x_1, \dots, x_{2n} on which to evaluate A and B , we will choose the $(2n)^{\text{th}}$ roots of unity. It's worth mentioning (although it's not necessary for understanding the algorithm) that the use of the complex roots of unity is the basis for the name *Fast Fourier Transform*: the representation of a degree- d polynomial P by its values on the $(d+1)^{\text{th}}$ roots of unity is sometimes referred to as the *discrete Fourier transform* of P ; and the heart of our procedure is a method for making this computation fast.

A Recursive Procedure for Polynomial Evaluation. We want to design an algorithm for evaluating A on each of $\omega_{1,2n}, \dots, \omega_{2n,2n}$ recursively, so as to take advantage of the familiar recurrence from (5.1), namely $T(n) \leq 2T(n/2) + O(n)$ where $T(n)$ in this case denotes the number of operations required to evaluate a polynomial of degree $n - 1$ on all the $(2n)^{\text{th}}$ roots of unity. For simplicity in describing this algorithm, we will assume that n is a power of 2.

How does one break the evaluation of a polynomial into two equal-sized sub-problems? A useful trick is to define two polynomials, $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$, that consist of the even and odd coefficients of A respectively. That is (assuming n is even),

$$A_{\text{even}}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n/2-1}x^{(n-1)/2},$$

and

$$A_{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{(n-1)/2}x^{(n-1)/2}.$$

Simple algebra shows us that

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2),$$

and so this gives us a way to compute $A(x)$ in a constant number of operations, given the evaluation of the two constituent polynomials that each have half the degree of A .

Now, suppose that we evaluate each of A_{even} and A_{odd} on the n^{th} roots of unity. This is exactly a version of the problem we face with A and the $(2n)^{\text{th}}$ roots of unity, except that the input is half as large: the degree is $(n - 1)/2$ rather than $n - 1$, and we have n roots of unity rather than $2n$. Thus, we can perform these evaluations in time $T(n/2)$ for each of A_{even} and A_{odd} , for a total time of $2T(n/2)$.

We're now very close to having a recursive algorithm that obeys (5.1) and giving us the running time we want; we just have to produce the evaluations of A on the $(2n)^{\text{th}}$ roots of unity using $O(n)$ additional operations. But this is easy, given the results from the recursive calls on A_{even} and A_{odd} . Consider one of these roots of unity $\omega_{j,2n} = e^{2\pi ji/2n}$. The quantity $\omega_{j,2n}^2$ is equal to $(e^{2\pi ji/2n})^2 = e^{2\pi ji/n}$, and hence $\omega_{j,2n}^2$ is an n^{th} root of unity. So when we go to compute

$$A(\omega_{j,2n}) = A_{even}(\omega_{j,2n}^2) + \omega_{j,2n}A_{odd}(\omega_{j,2n}^2),$$

we discover that both of the evaluations on the left-hand side have been performed in the recursive step, and so we can determine $A(\omega_{j,2n})$ using a constant number of operations. Doing this for all $2n$ roots of unity is therefore $O(n)$ additional operations after the two recursive calls, and so the bound $T(n)$ on the number of operations indeed satisfies $T(n) \leq 2T(n/2) + O(n)$. We run the same procedure to evaluate the polynomial B on the $(2n)^{\text{th}}$ roots of unity as well, and this gives us the desired $O(n \log n)$ bound for step (i) of our algorithm outline.

Polynomial Interpolation. We've now seen how to evaluate A and B on the set of $2n$ values $\omega_{1,2n}, \dots, \omega_{2n,2n}$ using $O(n \log n)$ operations, and as noted above we can clearly compute the products $C(\omega_{j,n}) = A(\omega_{j,2n})B(\omega_{j,2n})$ in $O(n)$ more operations. Thus, to conclude the algorithm for multiplying A and B , we need to execute step (iii) in our outline above using $O(n \log n)$ operations, reconstructing C from its values on $C(\omega_{1,2n}), C(\omega_{2,2n}), \dots, C(\omega_{2n,2n})$.

In describing this part of the algorithm, it's worth keeping track of the following top-level point: it turns out that the reconstruction of C can be achieved simply by defining an appropriate polynomial (the polynomial D below) and evaluating it at the $(2n)^{\text{th}}$ roots of unity. This is exactly what we've just seen how to do using $O(n \log n)$ operations, so we do it again here, spending an additional $O(n \log n)$ operations and concluding the algorithms.

Consider a polynomial $C(x) = \sum_{s=0}^{2n-1} c_s x^s$ that we want to reconstruct from its values $C(\omega_{s,2n})$ at the $(2n)^{\text{th}}$ roots of unity. Define a new polynomial $D(x) = \sum_{s=0}^{2n-1} d_s x^s$, where $d_s = C(\omega_{s,2n})$. We now consider the values of $D(x)$ at the $(2n)^{\text{th}}$ roots of unity.

$$\begin{aligned}
 D(\omega_{j,2n}) &= \sum_{s=0}^{2n-1} C(\omega_{s,2n}) \omega_{j,2n}^s \\
 &= \sum_{s=0}^{2n-1} \sum_{t=0}^{2n-1} c_t \omega_{s,2n}^t \omega_{j,2n}^s \\
 &= \sum_{t=0}^{2n-1} c_t \left(\sum_{s=0}^{2n-1} \omega_{s,2n} \omega_{j,2n}^s \right),
 \end{aligned}$$

by definition. Now recall that $\omega_{s,2n} = (e^{2\pi i/2n})^s$. Using this fact and extending the notation to $\omega_{s,2n} = (e^{2\pi i/2n})^s$ even when $s \geq 2n$ we get that

$$\begin{aligned}
 D(\omega_{j,2n}) &= \sum_{t=0}^{2n-1} c^t \left(\sum_{s=0}^{2n-1} e^{(2\pi i)(st+js)/2n} \right) \\
 &= \sum_{t=0}^{2n-1} c^t \left(\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s \right).
 \end{aligned}$$

To analyze the last line, we use the fact that for any $(2n)^{\text{th}}$ root of unity $\omega \neq 1$, we have $\sum_{s=0}^{2n-1} \omega^s = 0$. This is simply because ω is by definition a root of $x^{2n} - 1 = 0$; since $x^{2n} - 1 = (x - 1)(\prod_{t=0}^{2n-1} x^t)$ and $\omega \neq 1$, it follows that ω is also a root of $(\prod_{t=0}^{2n-1} x^t)$.

Thus, the only term of the last line's outer sum that is not equal to 0 is for c_t such that $\omega_{t+j,2n} = 1$; and this happens if $t + j$ is a multiple of $2n$, i.e., if $t = 2n - j$. For this value, $\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s = \sum_{s=0}^{2n-1} 1 = 2n$. So we get that $D(\omega_{j,2n}) = 2n c_{2n-j}$. So evaluating the polynomial $D(x)$ at the $(2n)^{\text{th}}$ roots of unity gives us the coefficients of the polynomial $C(x)$ in reverse order (multiplied by $2n$ each). We sum this up as follows.

$$\Sigma_{s=0}^{2n-1} (5.12) \text{ For any polynomial } C(x) = \sum_{s=0}^{2n-1} c_s x^s, \text{ and corresponding polynomial } D(x) = \sum_{s=0}^{2n-1} C(\omega_{s,2n}) x^s, \text{ we have that } c_t = \sum_{s=0}^{2n-1} D(\omega_{2n-s,2n}).$$

We can do all the evaluations of the values $D(\omega_{2n-s,2n})$ in $O(n \log n)$ operations using the divide-and-conquer approach developed for step (i).

And this wraps everything up: we reconstruct the polynomial C from its values on the $(2n)^{\text{th}}$ roots of unity, and then the coefficients of C are the coordinates in the convolution vector $c = a * b$ that we were originally seeking.

Solved Exercises

Solved Exercise 1

Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \dots, A[n]$ is *unimodal*: For some index p

between 1 and n , the values in the array entries increase up to position p in A , and then decrease the remainder of the way until position n . (So if you were to draw a plot with the array position j on the x -axis and the value of the entry $A[j]$ on the y -axis, the plotted points would rise until x -value p , where they'd achieve their maximum, and then fall from there on.)

You'd like to find the "peak entry" p without having to read the entire array — in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(\log n)$ entries of A .

Solution. Let's start with a general discussion on how to achieve a running time of $O(\log n)$, and then come back to the specific problem here. If one needs to compute something using only $O(\log n)$ operations, a useful strategy that we discussed in Chapter 2 is to perform a constant amount of work, throw away half the input, and continue recursively on what's left. This was the idea, for example, behind the $O(\log n)$ running time for binary search.

We can view this as a divide-and-conquer approach: for some constant $c > 0$, we perform at most c operations and then continue recursively on an input of size at most $n/2$. As in the chapter, we will assume that the recursion "bottoms out" when $n = 2$, performing at most c operations to finish the computation. If $T(n)$ denotes the running time on an input of size n , then we have the recurrence

(5.13)

$$T(n) \leq T(n/2) + c$$

when $n > 2$, and

$$T(2) \leq c.$$

It is not hard to solve this recurrence by unrolling it, as follows.

- *Analyzing the first few levels:* At the first level of recursion, we have a single problem of size n , which takes time at most c plus the time spent in all subsequent recursive calls. The next level has one problem of size at most $n/2$, which contributes another c , and the level after that has one problem of size at most $n/4$, which contributes yet another c .
- *Identifying a pattern:* No matter how many levels we continue, each level will have just one problem: level j has a single problem of size at most $n/2^j$, which contributes c to the running time, independent of j .

- *Summing over all levels of recursion:* Each level of the recursion is contributing at most c operations, and it takes $\log_2 n$ levels of recursion to reduce n to 2. Thus the total running time is at most c times the number of levels of recursion, which is at most $c \log_2 n = O(\log n)$.

We can also do this by partial substitution. Suppose we guess that $T(n) \leq k \log_b n$, where we don't know k or b . Assuming that this holds for smaller values of n in an inductive argument, we would have

$$\begin{aligned} T(n) &\leq T(n/2) + c \\ &\leq k \log_b(n/2) + c \\ &= k \log_b n - k \log_b 2 + c. \end{aligned}$$

Now, the first term on the right is exactly what we want, so we just need to choose k and b to negate the added c at the end. This we can do by setting $b = 2$ and $k = c$, so that $k \log_b 2 = c \log_2 2 = c$. Hence we end up with the solution $T(n) \leq c \log_2 n$, which is exactly what we got by unrolling the recurrence.

Finally, we should mention that one can get an $O(\log n)$, by essentially the same reasoning, in the more general case when each level of the recursion throws away any constant fraction of the input, transforming an instance of size n to one of size at most an , for some constant $a < 1$. It now takes at most $\log_{1/a} n$ level of recursion to reduce n down to a constant size, and each level of recursion involves at most c operations.

Now let's get back to the problem at hand. If we wanted to set ourselves up to use (5.13), we could probe the midpoint of the array and try to determine whether the "peak entry" p lies before or after this midpoint.

So suppose we look at the value $A[n/2]$. From this value alone we can't tell whether p lies before or after $n/2$, since we need to know whether entry $n/2$ is sitting on an "up-slope" or on a "down-slope." So we also look at the values $A[n/2 - 1]$ and $A[n/2 + 1]$. There are now three possibilities:

- If $A[n/2 - 1] < A[n/2] < A[n/2 + 1]$, then entry $n/2$ must come strictly before p , and so we can continue recursively on entries $n/2 + 1$ through n .
- If $A[n/2 - 1] > A[n/2] > A[n/2 + 1]$, then entry $n/2$ must come after p , and so we can continue recursively on entries 1 through $n/2 - 1$.
- Finally, if $A[n/2]$ is larger than both $A[n/2 - 1]$ and $A[n/2 + 1]$, we are done: the peak entry is in fact equal to $n/2$ in this case.

In all these cases, we perform at most three probes of the array A and reduce the problem to one of at most half the size. Thus we can apply (5.13) to conclude that the running time is $O(\log n)$.

You're consulting for a small computation-intensive investment company, and they have the following type of problem that they want to solve over and over. A typical instance of the problem is: they're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $i = 1, 2, \dots, n$; for each day i , they have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) Suppose during this time period, they wanted to buy 1000 shares on some day, and sell all these shares on some (later) day. They want to know: when should they have bought and when should they have sold in order to have made as much money as possible? (If there was no way to make money during the n days, you should report this instead.)

Example: Suppose $n = 3$, $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Then you should return "*buy on 2, sell on 3*"; i.e. buying on day 2 and selling on day 3 means they would have made \$4 per share, the maximum possible for that period.

Clearly, there's a simple algorithm that takes time $O(n^2)$: try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better.

Show how to find the correct numbers i and j in time $O(n \log n)$.

Solution. We've seen a number of instances in this chapter where a brute-force search over pairs of elements can be reduced to $O(n \log n)$ by divide-and-conquer. Since we're faced with a similar issue here, let's think about how we might apply a divide-and-conquer strategy.

A natural approach would be to consider the first $n/2$ days and the final $n/2$ days separately, solving the problem recursively on each of these two sets, and then figure out how to get an overall solution from this in $O(n)$ time. This would give us the usual recurrence $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$, and hence $T(n \log n)$ by (5.1).

Also, to make things easier, we'll make the usual assumption that n is a power of 2. This is no loss of generality: if n^j is the next power of 2 greater than n , we can set $p(i) = p(n)$ for all i between n and n^j . In this way, we do not change the answer, and we at most double the size of the input (which will not affect the $O()$ notation).

Now, let S be the set of days $1, \dots, n/2$, and S^j be the set of days $n/2 + 1, \dots, n$. Our divide-and-conquer algorithm will be based on the following observation: either there is an optimal solution in which the investors are holding the stock at the end of day $n/2$, or there isn't. Now, if there isn't, then the optimal solution is the better of the optimal solutions on the sets S and S^j . If there is an optimal solution in which they hold the stock at the end of day $n/2$, then the value of this solution is $p(j) - p(i)$ where $i \in S$ and $j \in S^j$. But this value is maximized by simply choosing $i \in S$ which minimizes $p(i)$, and choosing $j \in S^j$ which maximizes $p(j)$.

Thus, our algorithm is to take the best of the following three possible solutions:

- The optimal solution on S .
- The optimal solution on S^j .
- The maximum of $p(j) - p(i)$, over $i \in S$ and $j \in S$.

The first two alternatives are computed in time $T(n/2)$ each by recursion, and the third alternative is computed by finding the minimum in S and the maximum in S^j , which takes time $O(n)$. Thus the running time $T(n)$ satisfies

$$T(n) \leq 2T\frac{n}{2} + O(n),$$

as desired.

We note that this is not the best running time achievable for this problem. In fact one can find the optimal pair of days in $O(n)$ time using dynamic programming, the topic of the next chapter; at the end of that chapter, we will pose this question as Exercise 7.

1. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values — so there are $2n$ values total — and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.

2. Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers a_1, \dots, a_n , which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

3. Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank-cards that they've confiscated, suspecting them of being used in fraud. Each bank-card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank-cards corresponding to it, and we'll say that two bank-cards are *equivalent* if they correspond to the same account.

It's very difficult to read the account number off a bank-card directly, but the bank has a high-tech "equivalence tester" that takes two bank-cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

4. You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their set-up works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus, we can model their structure as consisting of the points $\{1, 2, 3, \dots, n\}$ on the real line; and at each of these points j , they have a particle with charge q_j . (Each charge can be either positive or negative.)

Now, they want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total force on particle j , by Coulomb's Law, is equal to

$$F^j = \sum_{i \neq j} \frac{C q_i q_j}{(j - i)^2}.$$

They've written the following simple program to computes F_j for all j :

```

For  $j = 1, 2, \dots, n$ 
    Initialize  $F_j$  to 0.
    For  $i = 1, 2, \dots, n$ 
        If  $i \neq j$  then
            Add  $\frac{C q_i q_j}{(j - i)^2}$  to  $F_j$ 
        Endif
    Endfor
    Output  $F_j$ 
Endfor

```

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i , takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental set-up is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

5. *Hidden surface removal* is a problem in computer graphics that scarcely needs an introduction — when Woody is standing in front of Buzz you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, . . . well, you get the idea.

The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n non-vertical lines in the plane, labeled L_1, \dots, L_n , with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x -coordinate x_0 if its y -coordinate at x_0 is greater than the y -coordinates of all the other lines at x_0 : $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x -coordinate at which it is uppermost — intuitively, some portion of it can be seen if you look down from " $y = \infty$ ".

Give an algorithm that takes n lines as input, and in $O(n \log n)$ time returns all of the ones that are visible. Figure 5.10 gives an example.

6. Consider an n -node complete binary tree T , where $n = 2^d - 1$ for some d . Each node v of T is labeled with a real number x_v . You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T , but the labeling is only specified in the following *implicit* way: for each node v , you can determine the value x_v by *probing* the node v . Show how to find a local minimum of T using only $O(\log n)$ probes to the nodes of T .

7. Suppose now that you're given an $n \times n$ grid graph G . (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a

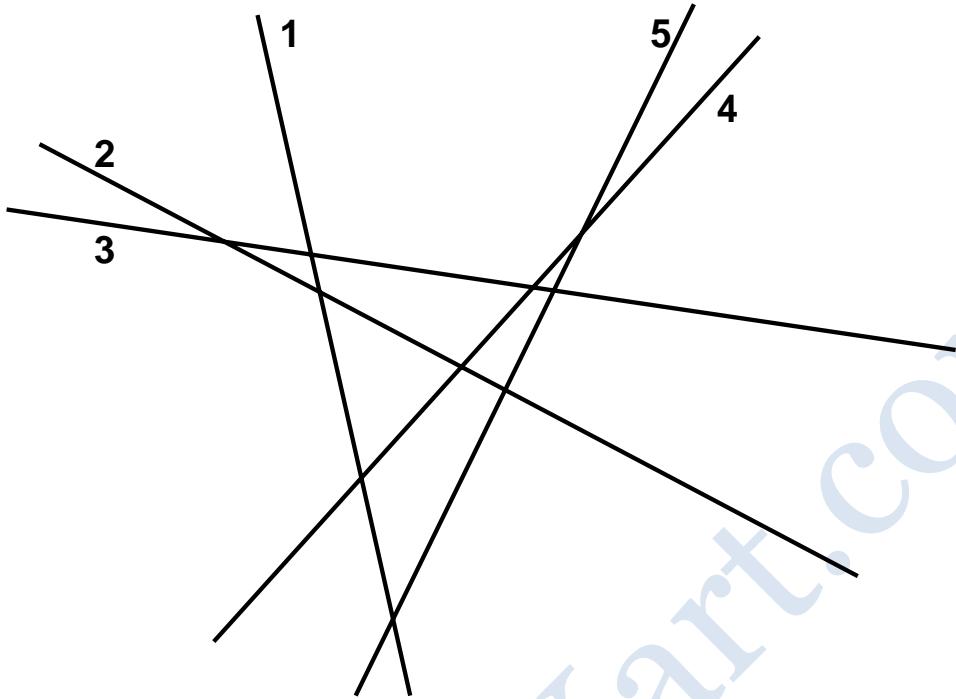


Figure 5.10: An instance with five lines (labeled “1”–“5” in the figure). All the lines except for “2” are visible.

graph whose node set is the set of all ordered pairs of natural numbers (i, j) , where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, A) are joined by an edge if and only if $|i - k| + |j - A| = 1$.)

We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v ; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G . (Note that G has n^2 nodes.)

The militaristic coinage “divide-and-conquer” was introduced somewhat after the technique itself. Knuth (1998) credits von Neumann with one early explicit application of the approach, the development of the Mergesort algorithm in 1945. Knuth (1997) also provides further discussion of techniques for solving recurrences.

The algorithm for computing the closest pair of points in the plane is due to Shamos, and is one of the earliest non-trivial algorithms in the field of computational geometry; the survey paper by Smid (1997) discusses a wide range of results on closest-point problems. A faster randomized algorithm for this problem will be discussed in Chapter 13. (Regarding the

non-obviousness of the divide-and-conquer algorithm presented here, Smid also makes the interesting historical observation that researchers originally suspected quadratic time might be the best one could do for finding the closest pair of points in the plane.) More generally, the divide-and-conquer approach has proved very useful in computational geometry, and the books by Preparata and Shamos (1985) and de Berg et al. (1997) give many further examples of this technique in the design of geometric algorithms.

The algorithm for multiplying two n -bit integers in sub-quadratic time is due to Karatsuba and Ofman (1962). Further background on asymptotically fast multiplication algorithms is given by Knuth (1997). Of course, the number of bits in the input must be sufficiently large for any of these sub-quadratic methods to improve over the standard algorithm.

Press et al. (1988) provide further coverage of the Fast Fourier Transform, including background on its applications in signal processing and related areas.

Exercises. Exercise 7 is based on a result of Llewellyn, Tovey, and Trick.

Greedy vs Dynamic Programming Approach

- Comparing the methods
- Knapsack problem
- Greedy algorithms for 0/1 knapsack
- An approximation algorithm for 0/1 knapsack
- Optimal greedy algorithm for knapsack with fractions
- A dynamic programming algorithm for 0/1 knapsack

Greedy Approach VS Dynamic Programming (DP)

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only show that the principle of optimality applies to the problem.

The 0/1 Knapsack problem

- Given a knapsack with weight $W > 0$.
- A set S of n items with weights $w_i > 0$ and benefits $b_i > 0$ for $i = 1, \dots, n$.
- $S = \{ (\text{item}_1, w_1, b_1), (\text{item}_2, w_2, b_2), \dots, (\text{item}_n, w_n, b_n) \}$
- Find a subset of the items which does not exceed the weight W of the knapsack and maximizes the benefit.

0/1 Knapsack problem

Determine a subset A of $\{ 1, 2, \dots, n \}$ that satisfies the following:

$$\max \sum_{i \in A} b_i \text{ where } \sum_{i \in A} w_i \leq W$$

In 0/1 knapsack a specific item is either selected or not

Variations of the Knapsack problem

- **Fractions are allowed.** This applies to items such as:
 - bread, for which taking half a loaf makes sense
 - gold dust
- **No fractions.**
 - 0/1 (1 brown pants, 1 green shirt...)
 - Allows putting many items of same type in knapsack
 - 5 pairs of socks
 - 10 gold bricks
 - More than one knapsack, etc.
- **First 0/1 knapsack problem will be covered then the**

Brute force!

- Generate all 2^n subsets
- Discard all subsets whose sum of the weights exceed W (*not feasible*)
- Select the maximum total benefit of the remaining (*feasible*) subsets
- What is the run time?
 $O(n 2^n)$, $\Omega(2^n)$
- Lets try the obvious greedy strategy .

Example with “brute force”

$$S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \}, W=25$$

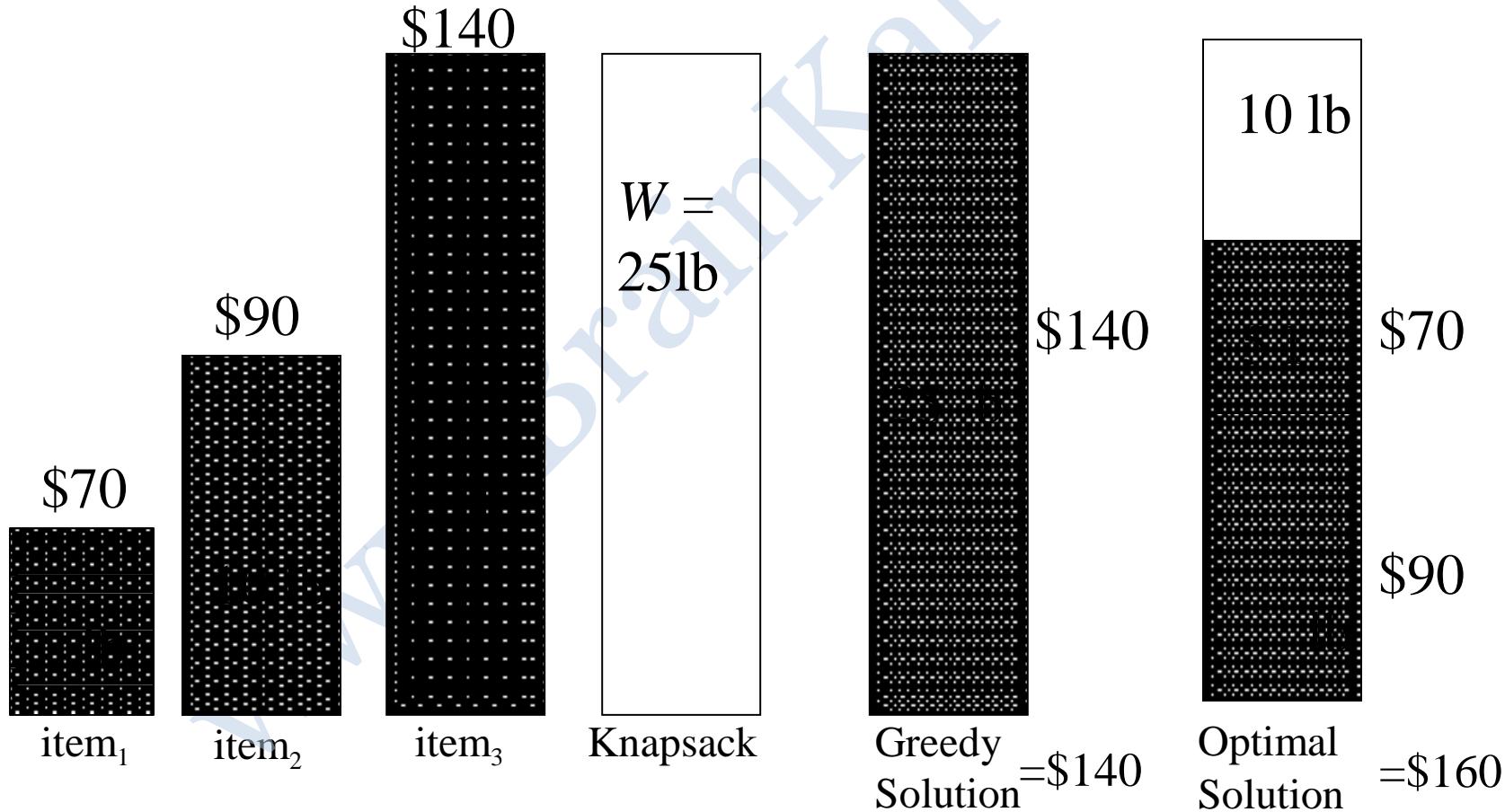
- Subsets:

1. {} Profit=\$0
2. { (item₁, 5, \$70) } Profit=\$70
3. { (item₂, 10, \$90) } Profit=\$90
4. { (item₃, 25, \$140) } Profit=\$140
5. { (item₁, 5, \$70), (item₂, 10, \$90) }. Profit=\$160 ****
6. { (item₂, 10, \$90), (item₃, 25, \$140) } exceeds W
7. { (item₁, 5, \$70), (item₃, 25, \$140) } exceeds W
8. { (item₁, 5, \$70), (item₂, 10, \$90), (item₃, 25, \$140) } exceeds W

Greedy 1: Selection criteria: *Maximum beneficial problems*

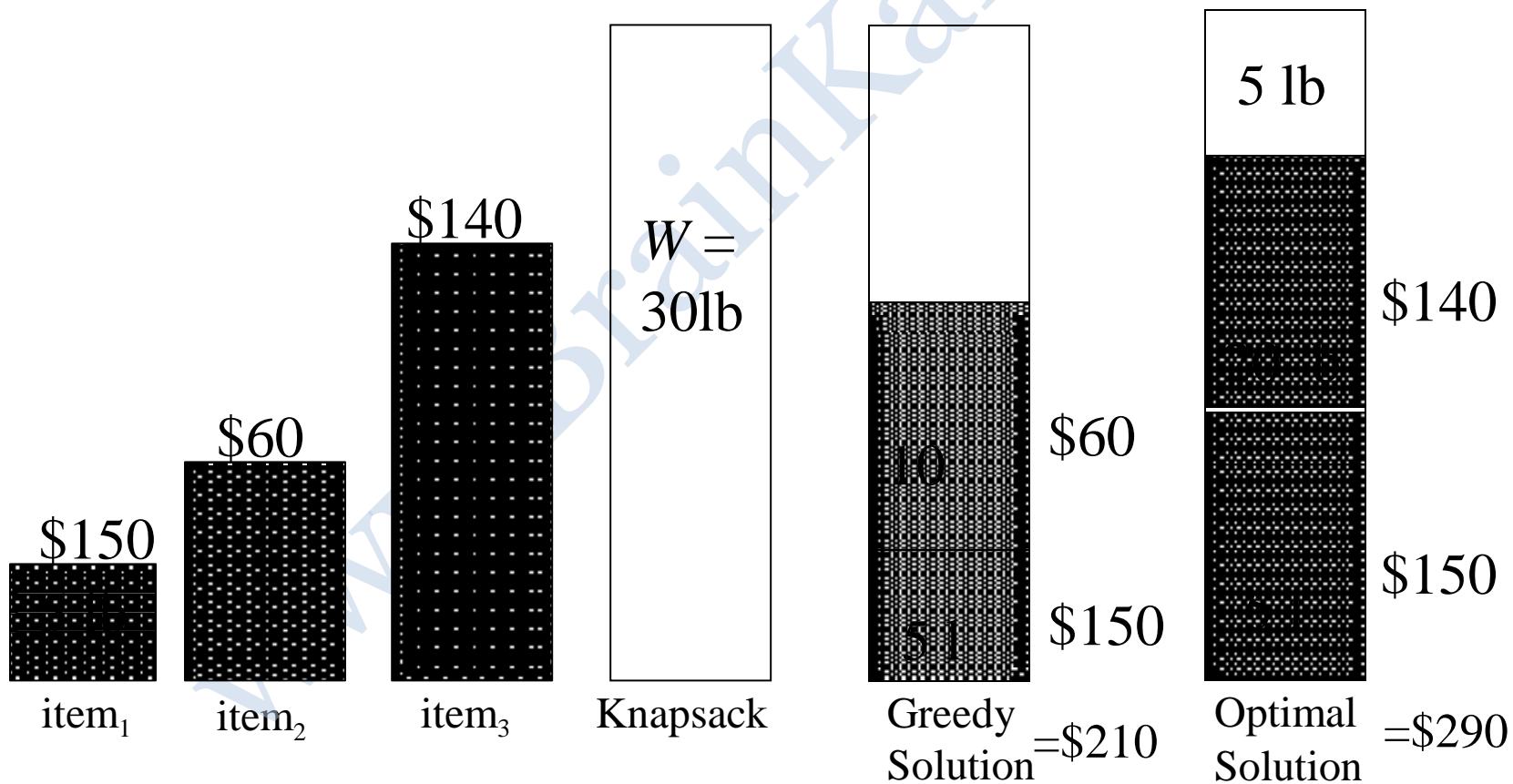
Counter Example:

$$S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \}$$



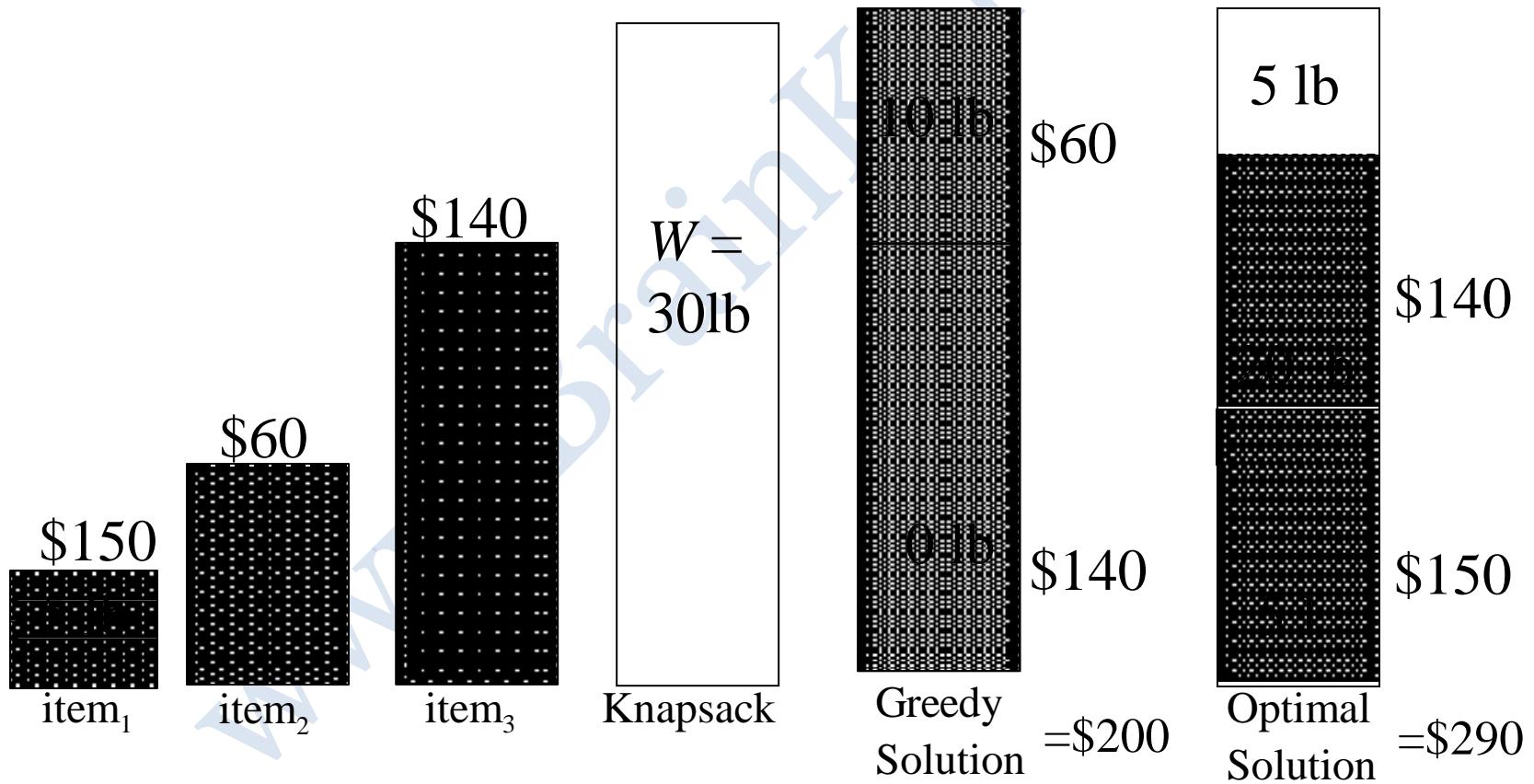
Greedy 2: Selection criteria: *Minimum weight item* Counter Example:

$$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$$



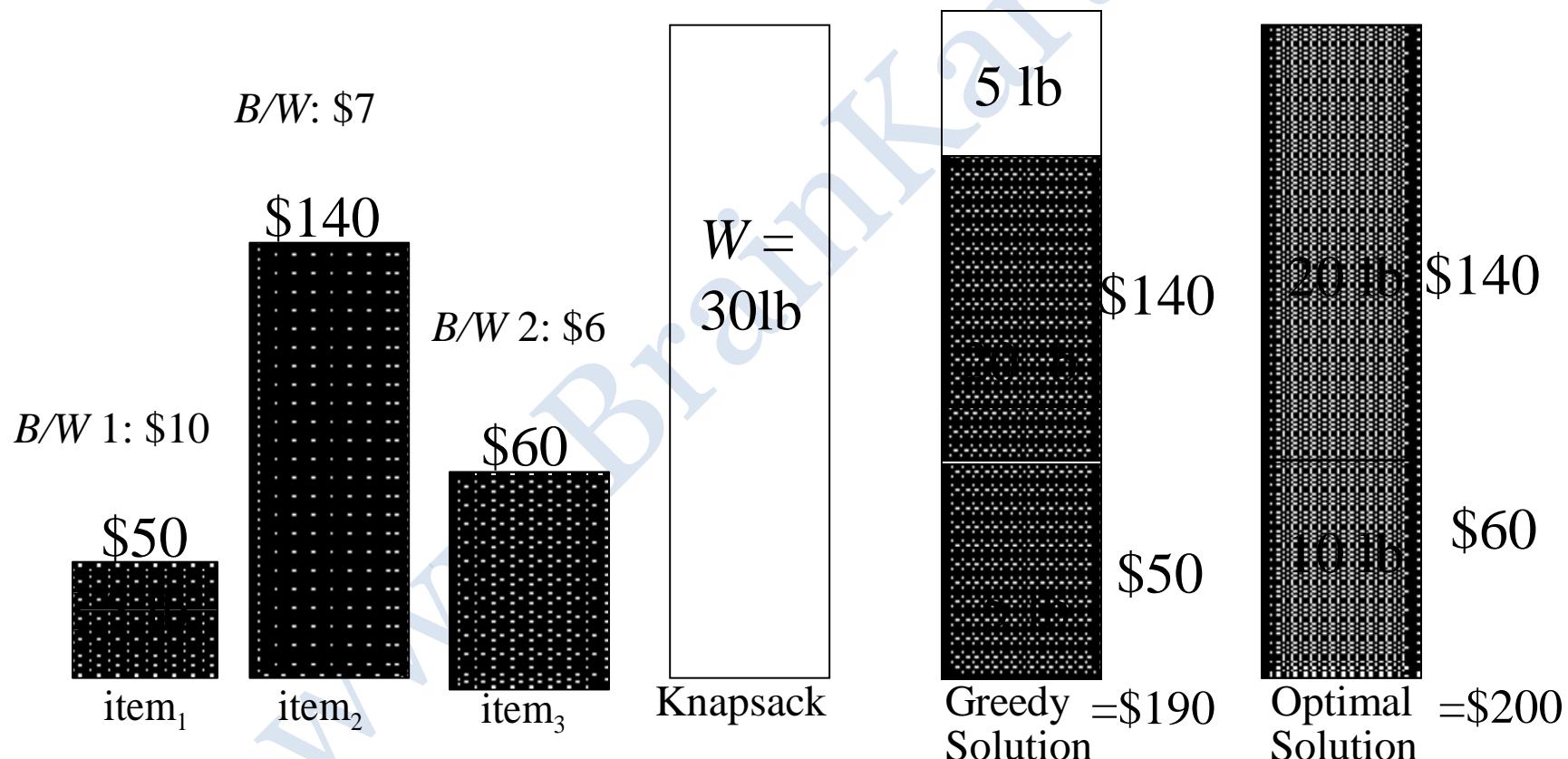
Greedy 3: Selection criteria: *Maximum weight item* Counter Example:

$$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$$



Greedy 4: Selection criteria: *Maximum benefit per unit item* Counter Example

$$S = \{ (item_1, 5, \$50), (item_2, 20, \$140), (item_3, 10, \$60), \}$$



What is the asymptotic runtime of this algorithm?

Approximation algorithms

- *Approximation algorithms* attempt to evaluate how far away from the optimum OPT, are the solutions solAlg provided by an algorithm in the worst case
- Many criteria are used. We use OPT/solAlg for maximization, and attempt to establish $\text{OPT}/\text{solAlg} \leq K$ where K is a constant (solAlg/OPT for minimization)
- The following slides show that the “best” greedy algorithm for 0/1 knapsack, greedy 4 does not satisfy $\text{OPT}/\text{solAlg} \leq K$
- Often greedy4 gives an optimal solutions, but for some problem instances the ratio can become very large
- A small modification of greedy4, however, guarantees, that $\text{OPT}/\text{alg} \leq 2$
- This is a big improvement.
- There are better approximation algorithms for knapsack

Approximation algorithms

- Use greedy 4: select the items with *maximum benefit per unit*.
 - Implement by Sorting S by *benefit per unit*.
- Example where greedy4 provides a very poor solution:
 - Assume a 0/1 knapsack problem with $n=2$
 - very large W .
 - $S=\{ (\text{item1}, 1, \$2), (\text{item 2}, W, \$1.5W) \}$.
- The solution to greedy4 has a benefit of \$2
- An optimal solution has a benefit of $\$1.5W$.
- If we want the best investment and we have $W=10,000$ dollars. We should choose the 2nd one with a profit of \$15,000, and not the first with a profit of \$2.

Approximation Continued

- Let \mathbf{BOpt} denote the optimal benefit for the 0/1 knapsack problem
- Let $\mathbf{BGreedy4}$ be the benefit calculated by greedy4.
 - For last example $\mathbf{BOpt} / \mathbf{BGreedy4} = \$1.5W / 2$
 - Note: W can be arbitrarily large
- We would like to find a better algorithm \mathbf{Alg} such that
- $\mathbf{BOpt} / \mathbf{Alg} \leq K$ where K is a small constant and is independent of the problem instance.

A Better Approximation Algorithm

- Let $\maxB = \max\{ b_i \mid i=1, \dots, n \}$
- The approximation algorithm selects, either the solution to Greedy4, or only the item with benefit \maxB depending on $\max\{ B_{\text{Greedy4}}, \maxB \}$.
- Let $APP = \max\{ B_{\text{Greedy4}}, \maxB \}$
- What is the asymptotic runtime of this algorithm?
- It can be shown that with this modification the ratio $B_{\text{Opt}} / APP \leq 2$ (Optimal benefit at most twice that of APP)

Fractions (KWF)

In this problem a fraction of any item may be chosen
The following algorithm provides the optimal benefit:

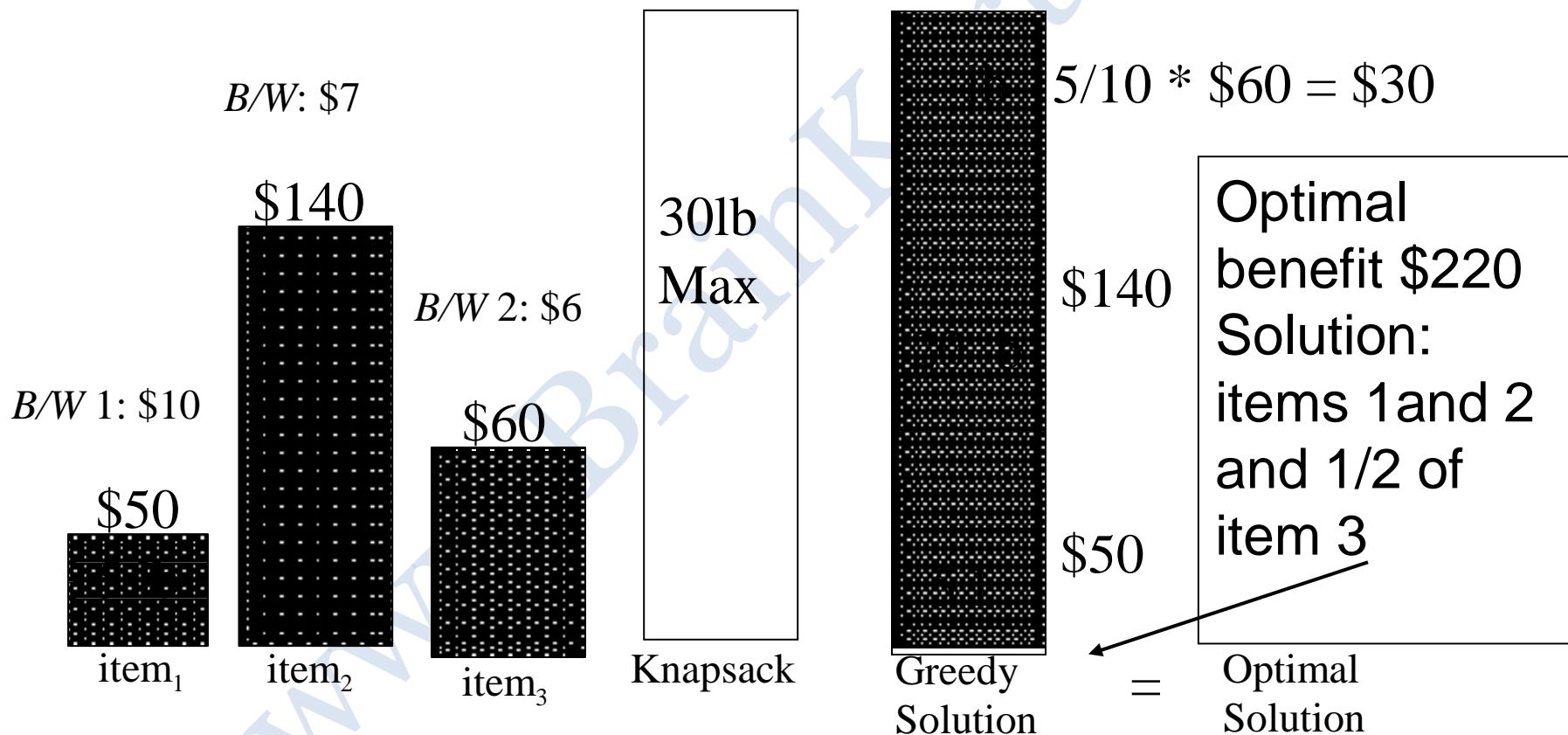
- The greedy algorithm uses the *maximum benefit per unit* selection criteria
 1. Sort items in decreasing b_i/w_i .
 2. Add items to knapsack (starting at the first) until there are no more items, or the next item to be added exceeds W .
 3. If knapsack is not yet full, fill knapsack with a fraction of next unselected item.

KWF

- Let k be the index of the last item included in the knapsack. We may be able to include the whole or only a fraction of item k
- Without item k totweight = $\sum_{i=1}^{k-1} w_i$*
- $$profitKWF = \sum_{i=1}^{k-1} p_i + \min\{(\mathbf{W} - totweight), w_k\} \times (p_k / w_k)$$
- $\min\{(\mathbf{W} - totweight), w_k\}$, means that we either take the whole of item k when the knapsack can include the item without violating the constraint, or we fill the knapsack by a fraction of item k

Example of applying the optimal greedy algorithm for Fractional Knapsack Problem

$S = \{ (\text{item}_1, 5, \$50), (\text{item}_2, 20, \$140), (\text{item}_3, 10, \$60) \}$



Fractional Knapsack Problem

W=30

S = { (item1 , 5, \$50), (item2 ,20, \$140), (item3, 10, \$60) }

Note: items are already sorted by benefit/weight

Applying the algorithm:

Current weight in knapsack=0, Current benefit=0.

Can item 1 fit? $0+5 < 30$ so select it. Current benefit=0+50

Can item 2 fit? $5+20 < 30$, so select. Current benefit = $50+140=190$

Can item 3 fit? $25+10 > 30$. No.

We can add 5 to knapsack (30-25).

So select $5/10=0.5$ of item 3.

Current benefit= $190+30=220$

Greedy Algorithm for Knapsack with fractions

- **To show that the greedy algorithm finds the optimal profit for the fractional Knapsack problem you need to prove there is no solution with a higher profit (see text)**
- **Notice there may be more than one optimal solution**

Principle of Optimality for 0/1 Knapsack problem

- **Theorem:** 0/1 knapsack satisfies the principle of optimality
- **Proof:** Assume that item_i is in the **most beneficial subset** that weighs at most W . If we remove item_i from the subset the remaining subset must be **the most beneficial subset** weighing at most $W - w_i$ of the $n - 1$ remaining items after excluding item_i.
- If the remaining subset after excluding item_i was not the most beneficial one weighing at most $W - w_i$ of the $n - 1$ remaining items, we could find a better solution for this problem and improve the optimal solution. This is impossible.

- Given a knapsack problem with n items and knapsack weight of W .
- We will first compute the maximum benefit, and then determine the subset.
- To use dynamic programming we solve smaller problems and use the optimal solutions of these problems to find the solution to larger ones.

- **What are the smaller problem?**
 - Assume a subproblem in which the set of items is restricted to $\{1, \dots, i\}$ where $i \leq n$, and the weight of the knapsack is w , where $0 \leq w \leq W$.
 - Let $B[i, w]$ denote the maximum benefit achieved for this problem.
 - Our goal is to compute the maximum benefit of the original problem $B[n, W]$
 - We solve the original problem by computing $B[i, w]$ for $i = 0, 1, \dots, n$ and for $w = 0, 1, \dots, W$.
 - We need to specify the solution to a larger problem in terms of a smaller one

3 cases:

1. There are no items in the knapsack, or the weight of the knapsack is 0 - the benefit is 0
2. The weight of item_i exceeds the weight w of the knapsack - item_i cannot be included in the knapsack and the maximum benefit is B[i-1, w]
3. Otherwise, the benefit is the maximum achieved by either not including item_i (i.e., B[i-1, w]), or by including item_i (i.e., B[i-1, w-w_i]+b_i)

$$B[i, w] = \begin{cases} 0 & \text{for } i = 0 \text{ or } w = 0 \\ \max\{B[i-1, w], B[i-1, w - w_i] + b_i\} & \text{if } w_i > w \\ & i \quad i \end{cases}$$

$(n+1) \times (W+1)$ Matrix

Input: $\{w_1, w_2, \dots, w_n\}$, W , $\{b_1, b_2, \dots, b_n\}$

Output: $B[n, W]$,

```
for  $w \leftarrow 0$  to  $W$  do // row 0
     $B[0, w] \leftarrow 0$ 
for  $k \leftarrow 1$  to  $n$  do // rows 1 to n
     $B[k, 0] \leftarrow 0$       // element in column 0
    for  $w \leftarrow 1$  to  $W$  do // elements in columns 1 to  $W$ 
        if ( $w_k \leq w$ ) and ( $B[k-1, w-w_k] + b_k > B[k-1, w]$ )
            then  $B[k, w] \leftarrow B[k-1, w-w_k] + b_k$ 
            else  $B[k, w] \leftarrow B[k-1, w]$ 
```

[Click Here for Design and Analysis of Algorithms full study material.](#)

$$W = 30, S = \{ (i_1, 5, \$50), (i_2, 10, \$60), (i_3, 20, \$140) \}$$

Weight:	0	1	2	3	...	30
MaxProfit { }	0	0	0	0	...	0

Weight:	0	1	2	3	4	5	...	30
MaxProfit { }	0	0	0	0	0	0	...	0
MaxProfit <i>{i}_1</i>	0	0	0	0	0	50	...	50

Weight:	0	...	4	5	...	9	10	...	14	15	...	30
MaxProfit { }	0	...	0	0	...	0	0	...	0	0	...	0
MaxProfit{ i_1 }	0	...	0	50	...	50	50	...	50	50	...	50
MaxProfit{ i_1, i_2 }	0	...	0	50	...	50	60	...	60	110	...	110

- $B[2,10] = \max \{ B[1,10], B[1,10-10] + b_2 \}$
= 60
- $B[2,15] = \max \{ B[1,15], B[1,15-10] + b_2 \}$
= $\max \{50, 50+60\}$
= 110

Wt:	0...4	5 ... 9	10...14	15...	19	20...24	25...29	30
MaxP{ }	0...0	0 ... 0	0 ... 0	0 ... 0	0	0...0	0...0	0
MaxP{i ₁ }	0...0	50...50	50...50	50...	50	50...50	50...50	
	50							
MaxP{i ₁ , i ₂ }	0...0	50...50	60...60	110...110	110	110...	110	110
MaxP{i ₁ , i ₂ , i ₃ }	0...0	50...50	60...60	110...110	140	140...140	190...190	200

- $B[3,20] = \max \{ B[2,20], B[2,20-20] + b_3 \}$
 $= 140$
- $B[3,25] = \max \{ B[2,25], B[2,25-20] + 140 \}$
 $= \max \{ 110, 50+140 \}$
 $= 190$
- $B[3,30] = \max \{ B[2,30], B[2,30-20] + 140 \}$
 $= 200$

Analysis

- It is straightforward to fill in the array using the expression on the previous slide. SO What is the size of the array??
- The array is the (number of items+1) * (W+1).
- So the algorithm will run in $\Theta(nW)$. It appears to be linear BUT the weight is not a function of only the number of items. What if $W = n!$? Then this algorithm is worst than the brute force method. One can show algorithms for 0/1 knapsack with worst case time complexity of $O(\min(2^n, nW))$ (N169)
- No one has ever found a 0/1 knapsack algorithm whose worst case time is better than exponential AND no one has proven that such an algorithm is not possible.

Using $(W+1) * 2$ matrix

Knapsack01($W, w_1, \dots w_n, b_1 \dots b_n$)

Note: Calculating $B[k, w]$ depends only on

$B[k-1, w]$ or $B[k-1, w - w_k]$. $B[n, W] = B[0, W]$ or $B[1, W]$ depending on whether n is even or odd.

```
 $B \leftarrow 0$  // Initialize  $W$  by 2 matrix
for  $k \leftarrow 1$  to  $n$  do
    for  $w \leftarrow 1$  to  $W$  do //  $w$  is not  $w_i$ 
        if ( $w_k \leq w$ ) and ( $B[\text{mod}(k, 2), w - w_k] + b_k$ 
             $> B[\text{mod}(k, 2), w]$ ) then
             $B[w, \text{mod}(k+1, 2)] \leftarrow$ 
                 $B[\text{mod}(k+1, 2), w - w_k] + b_k$ 
```

Iterative Improvement

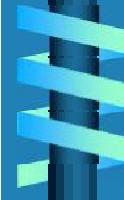


Algorithm design technique for solving optimization problems

- ② Start with a feasible solution
- ② Repeat the following step until no improvement can be found:
 - change the current feasible solution to a feasible solution with a better value of the objective function
- ② Return the last feasible solution as optimal

Note: Typically, a change in a current solution is “small” (local search)

Major difficulty: Local optimum vs. global optimum



Important Examples



- ❑ simplex method
- ❑ Ford-Fulkerson algorithm for maximum flow problem
- ❑ maximum matching of graph vertices
- ❑ Gale-Shapley algorithm for the stable marriage problem

- ❑ local search heuristics



Linear Programming



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

maximize (or minimize) $c_1x_1 + \dots + c_nx_n$

subject to $a_{i1}x_1 + \dots + a_{in}x_n \leq (\text{or } \geq \text{ or } =) b_i, i = 1, \dots, m$
 $x_1 \geq 0, \dots, x_n \geq 0$

The function $z = c_1x_1 + \dots + c_nx_n$ is called the *objective function*;
constraints $x_1 \geq 0, \dots, x_n \geq 0$ are called *nonnegativity constraints*





Example

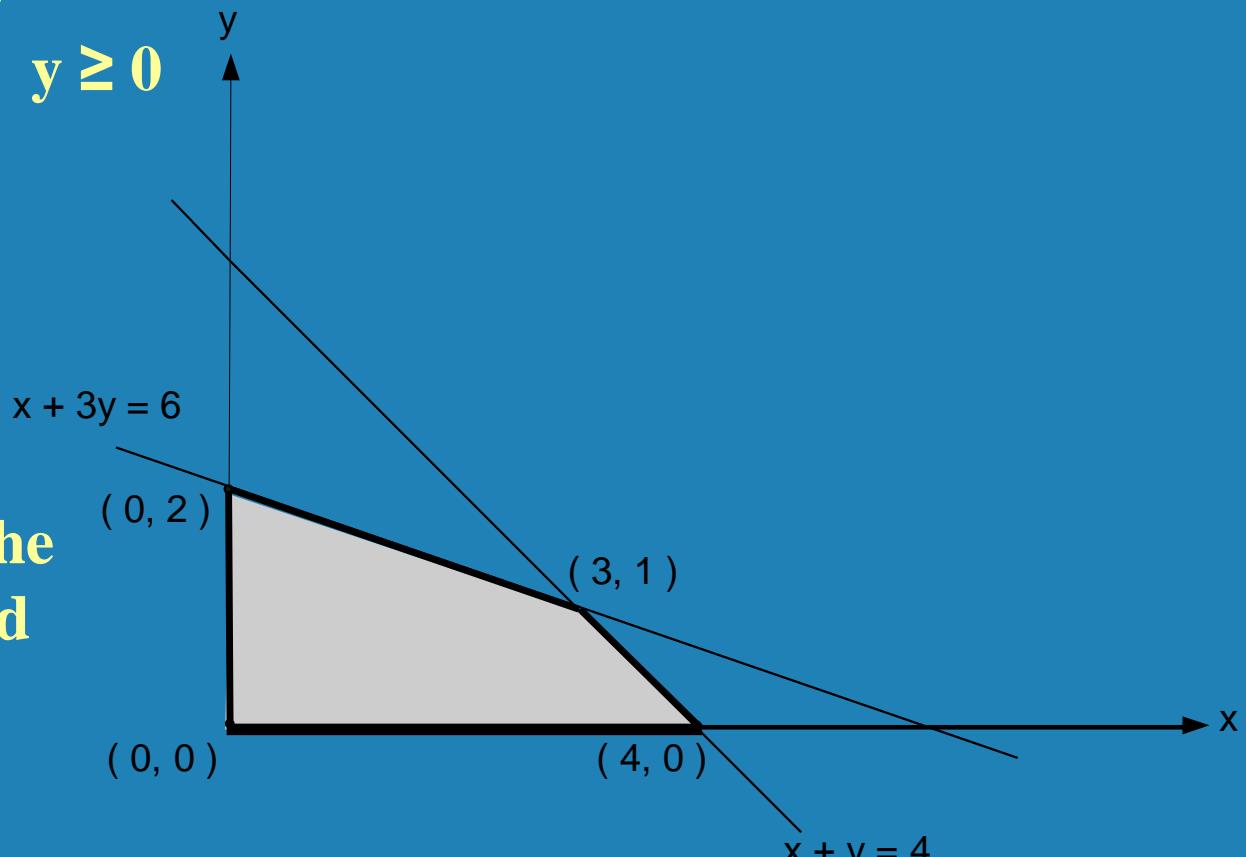
maximize $3x + 5y$

subject to $x + y \leq 4$

$x + 3y \leq 6$

$x \geq 0, y \geq 0$

Feasible region is the set of points defined by the constraints



Geometric solution

maximize

$$3x + 5y$$

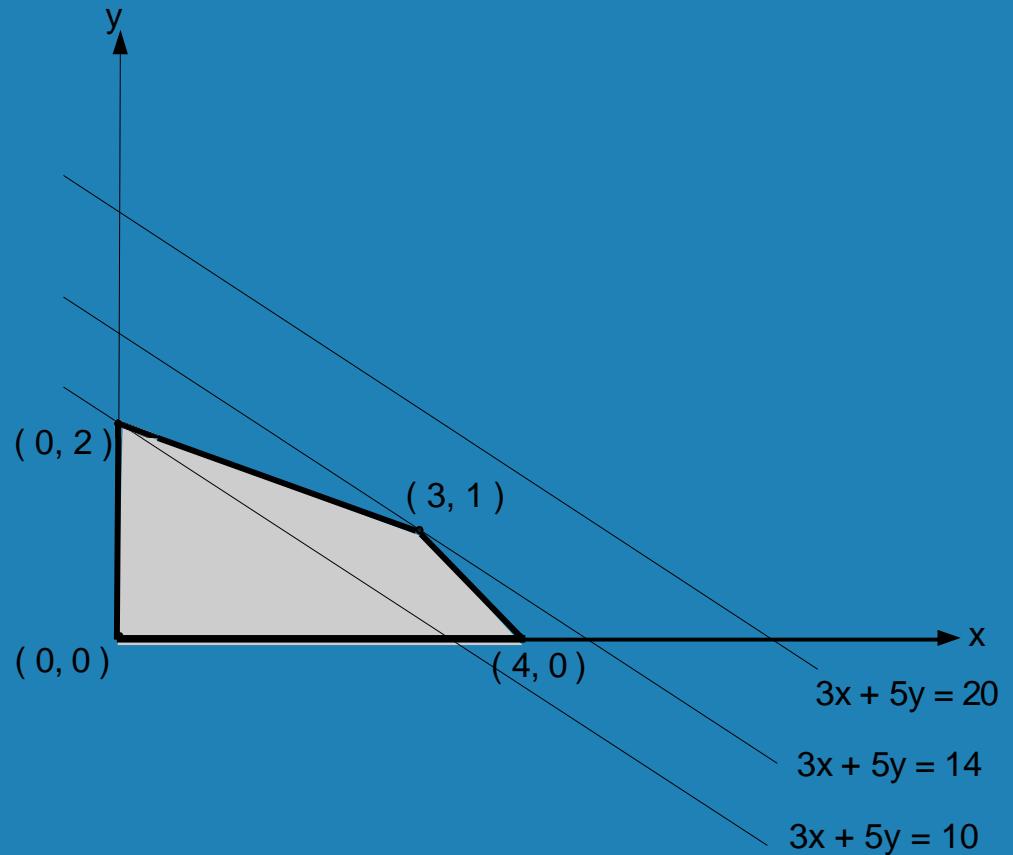
subject to

$$x + y \leq 4$$

$$x + 3y \leq 6$$

$$x \geq 0, y \geq 0$$

Optimal solution: $x = 3, y = 1$



Extreme Point Theorem Any 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.

3 possible outcomes in solving an LP problem



- ∅ has a finite optimal solution, which may no be unique
- ∅ *unbounded*: the objective function of maximization (minimization) LP problem is unbounded from above (below) on its feasible region
- ∅ *infeasible*: there are no points satisfying all the constraints, the constraints are contradictory



The Simplex Method



- ∅ The classic method for solving LP problems;
one of the most important algorithms ever invented
- ∅ Invented by George Dantzig in 1947
- ∅ Based on the iterative improvement idea:
Generates a sequence of adjacent points of the problem's feasible region with improving values of the objective function until no further improvement is possible



Standard form of LP problem



- ∅ must be a maximization problem
- ∅ all constraints (except the nonnegativity constraints) must be in the form of linear equations
- ∅ all the variables must be required to be nonnegative

Thus, the general linear programming problem in standard form with m constraints and n unknowns ($n \geq m$) is

$$\begin{aligned} & \text{maximize } c_1x_1 + \dots + c_nx_n \\ & \text{subject to } a_{i1}x_1 + \dots + a_{in}x_n = b_i, \quad i = 1, \dots, m, \\ & \quad x_1 \geq 0, \dots, x_n \geq 0 \end{aligned}$$

Every LP problem can be represented in such form

Example

maximize $3x + 5y$

subject to $x + y \leq 4$

$x + 3y \leq 6$

$x \geq 0, y \geq 0$



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

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0, y \geq 0, u \geq 0, v \geq 0$

Variables u and v , transforming inequality constraints into equality constraints, are called *slack variables*

Basic feasible 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 by solving the system are called *basic*.

A basic solution is called *feasible* if all its (basic) variables are nonnegative.

Example $x + y + u = 4$
 $x + 3y + v = 6$

(0, 0, 4, 6) is basic feasible solution
(x, y are nonbasic; u, v are basic)

There is a 1-1 correspondence between extreme points of LP's feasible region and its basic feasible solutions.

Simplex Tableau

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

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0, y \geq 0, u \geq 0, v \geq 0$

basic
variables

objective row

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	3	5	0	0	0

basic feasible solution

(0, 0, 4, 6)

value of z at (0, 0, 4, 6)

Outline of the Simplex Method



Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative — 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 variable] For each positive entry in the pivot column, calculate the θ -ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If there are no positive entries in the pivot column — 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.

Example of Simplex Method Application

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

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0, y \geq 0, u \geq 0, v \geq 0$

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	3	5	0	0	0

	x	y	u	v	
u	2	0	1	$\frac{1}{\xi}$	2
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2
	4	0	0	$\frac{5}{3}$	10

	x	y	u	v	
x	1	0	$\frac{3}{2}$	$\frac{1}{3}$	3
y	0	1	$\frac{1}{2}$	$\frac{1}{2}$	1
	0	0	2	1	14

basic feasible sol.

$(0, 0, 4, 6)$

$$z = 0$$

basic feasible sol.

$(0, 2, 2, 0)$

$$z = 10$$

basic feasible sol.

$(3, 1, 0, 0)$

$$z = 14$$

Notes on the Simplex Method



- ❑ Finding an initial basic feasible solution may pose a problem
- ❑ Theoretical possibility of cycling
- ❑ Typical number of iterations is between m and $3m$, where m is the number of equality constraints in the standard form
- ❑ Worse-case efficiency is exponential
- ❑ More recent *interior-point algorithms* such as *Karmarkar's algorithm* (1984) have polynomial worst-case efficiency and have performed competitively with the simplex method in empirical tests

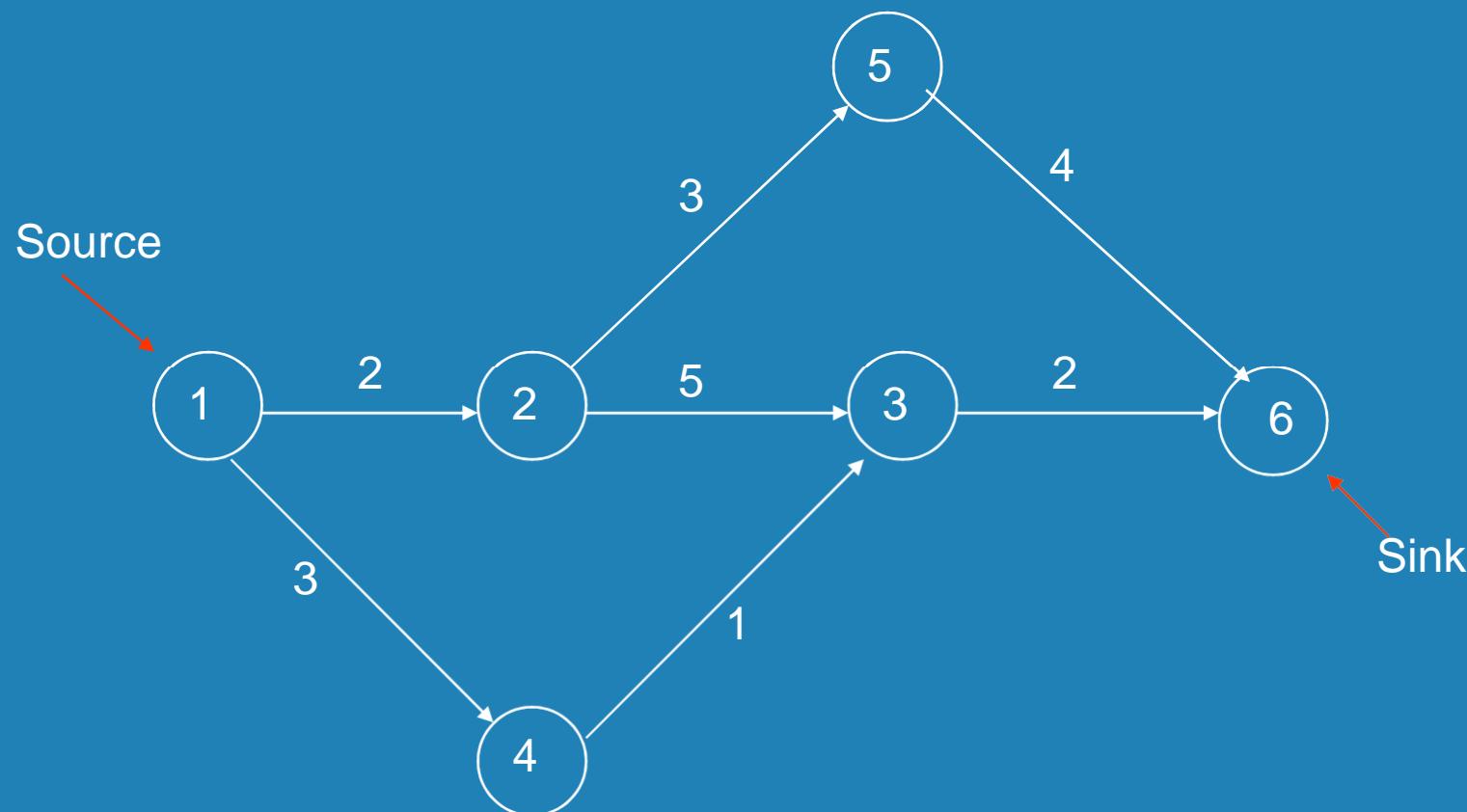
Maximum Flow Problem

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

Formally represented by a connected weighted digraph with n vertices numbered from 1 to n with the following properties:

- contains exactly one vertex with no entering edges, called the *source* (numbered 1)
- contains exactly one vertex with no leaving edges, called the *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

Example of Flow Network



Definition of a Flow

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

② *flow-conservation requirements*

The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex

$$\sum_{j: (j,i) \in E} x_{ji} = \sum_{j: (i,j) \in E} x_{ij} \text{ for } i = 2, 3, \dots, n-1$$

③ *capacity constraints*

$$0 \leq x_{ij} \leq u_{ij} \text{ for every edge } (i,j) \in E$$

Flow value and Maximum Flow Problem



Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

$$\sum_{j: (1,j) \in E} x_{1j} = \sum_{j: (j,n) \in E} x_{jn}$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into the sink).

The *maximum flow problem* is to find a flow of the largest value (maximum flow) for a given network.



Maximum-Flow Problem as LP problem

$$\text{Maximize } v = \sum_{j: (1,j) \in E} x_{1j}$$

subject to

$$\sum_{j: (j,i) \in E} x_{ji} - \sum_{j: (i,j) \in E} x_{ij} = 0 \quad \text{for } i = 2, 3, \dots, n-1$$

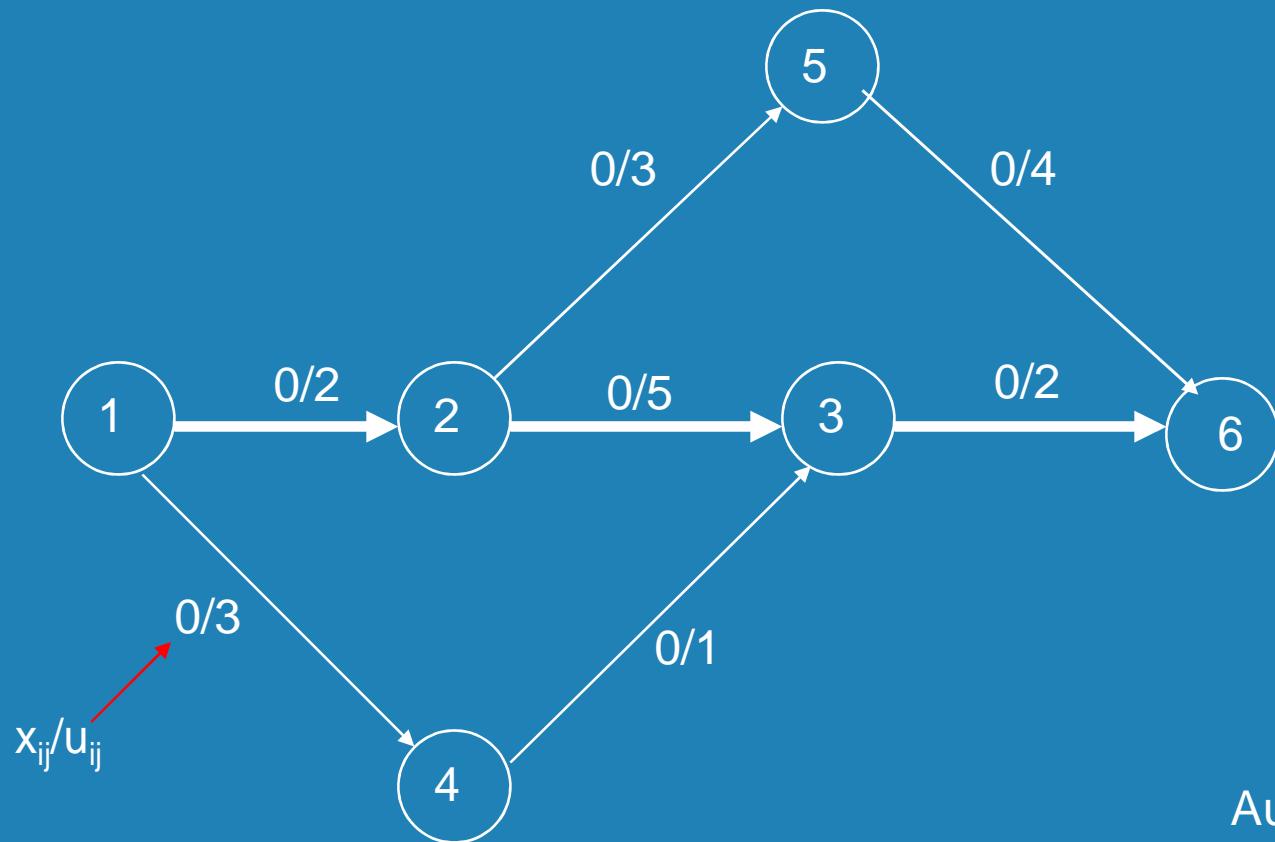
$$0 \leq x_{ij} \leq u_{ij} \quad \text{for every edge } (i,j) \in E$$

Augmenting Path (Ford-Fulkerson) Method

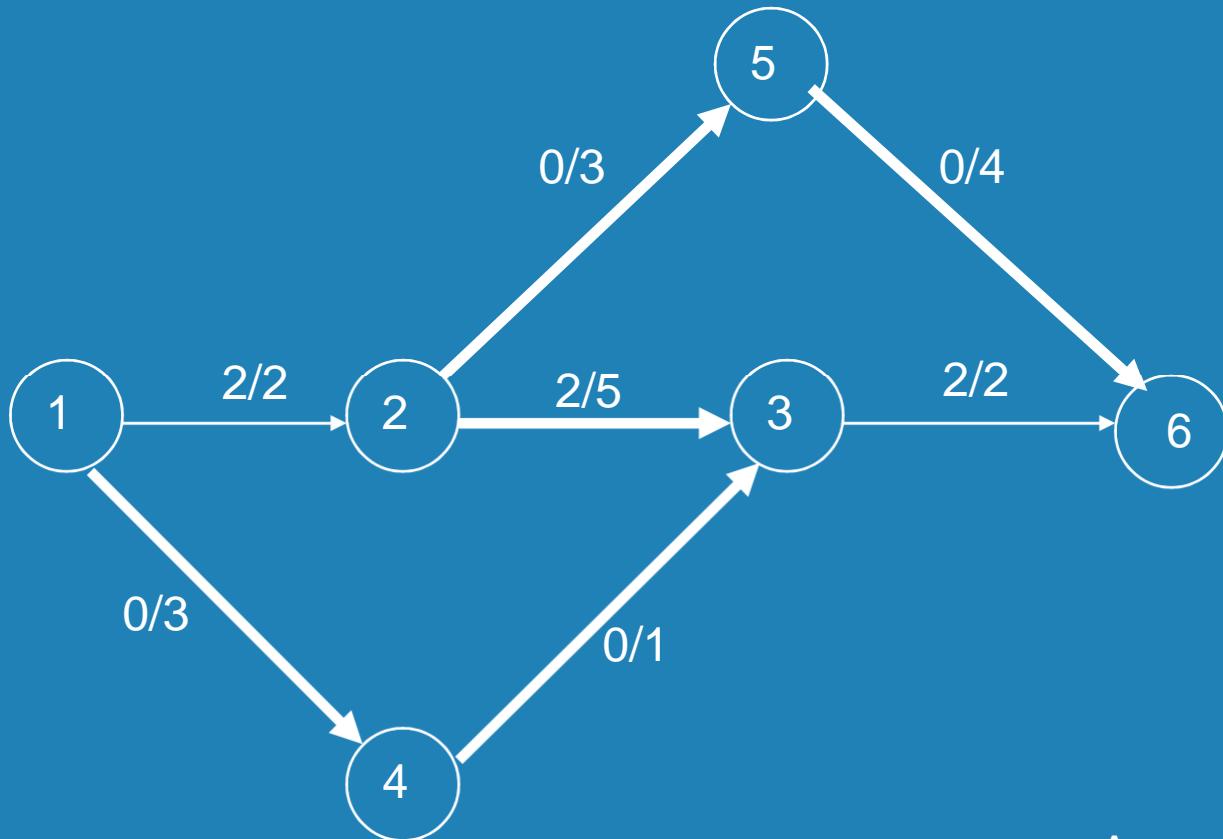


- ② Start with the zero flow ($x_{ij} = 0$ for every edge)
- ② On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent
- ② If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
- ② If no flow-augmenting path is found, the current flow is maximum

Example 1



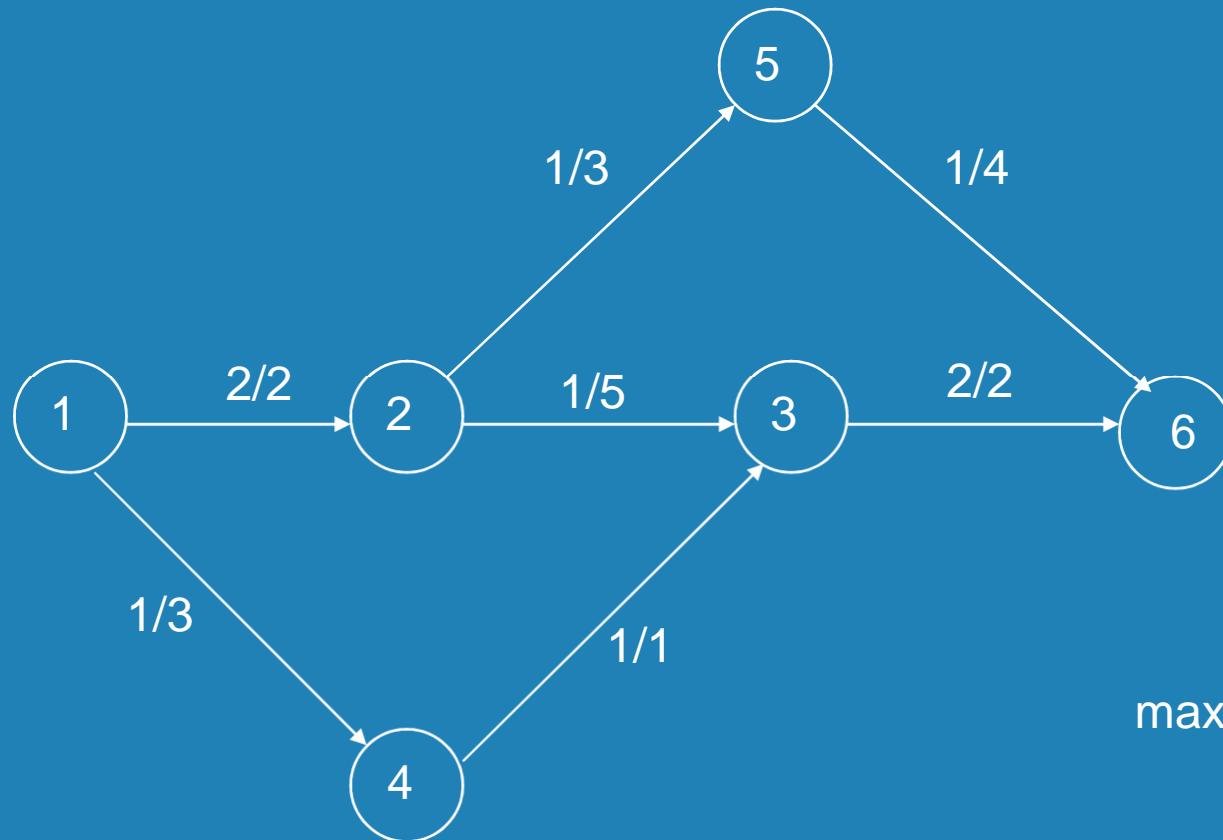
Example 1 (cont.)



Augmenting path:
1 → 4 → 3 ← 2 → 5 → 6



Example 1 (maximum flow)



Finding a flow-augmenting path



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

- connected by a directed edge (i to j) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$
 - known as *forward edge* (\rightarrow)

OR

- connected by a directed edge (j to i) with positive flow x_{ji}
 - known as *backward edge* (\leftarrow)

If a flow-augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge and decreasing x_{ji} by r on each backward edge, where

$$r = \min \{r_{ij} \text{ on all forward edges, } x_{ji} \text{ on all backward edges}\}$$

Finding a flow-augmenting path (cont.)

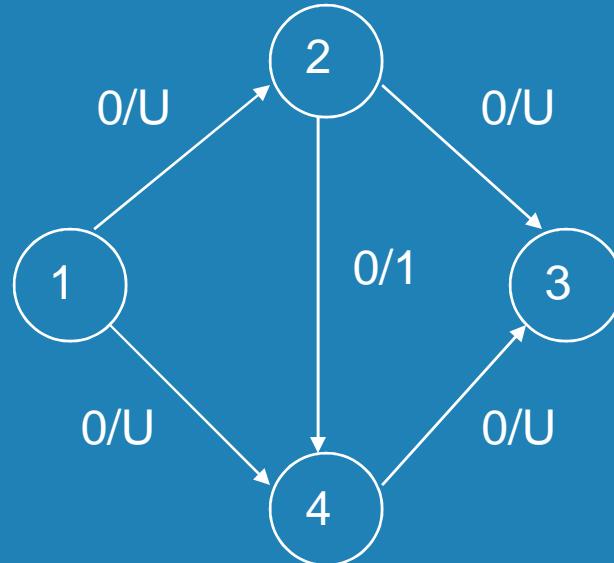
- ❑ Assuming the edge capacities are integers, r is a positive integer
- ❑ On each iteration, the flow value increases by at least 1
- ❑ Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- ❑ The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used



Performance degeneration of the method

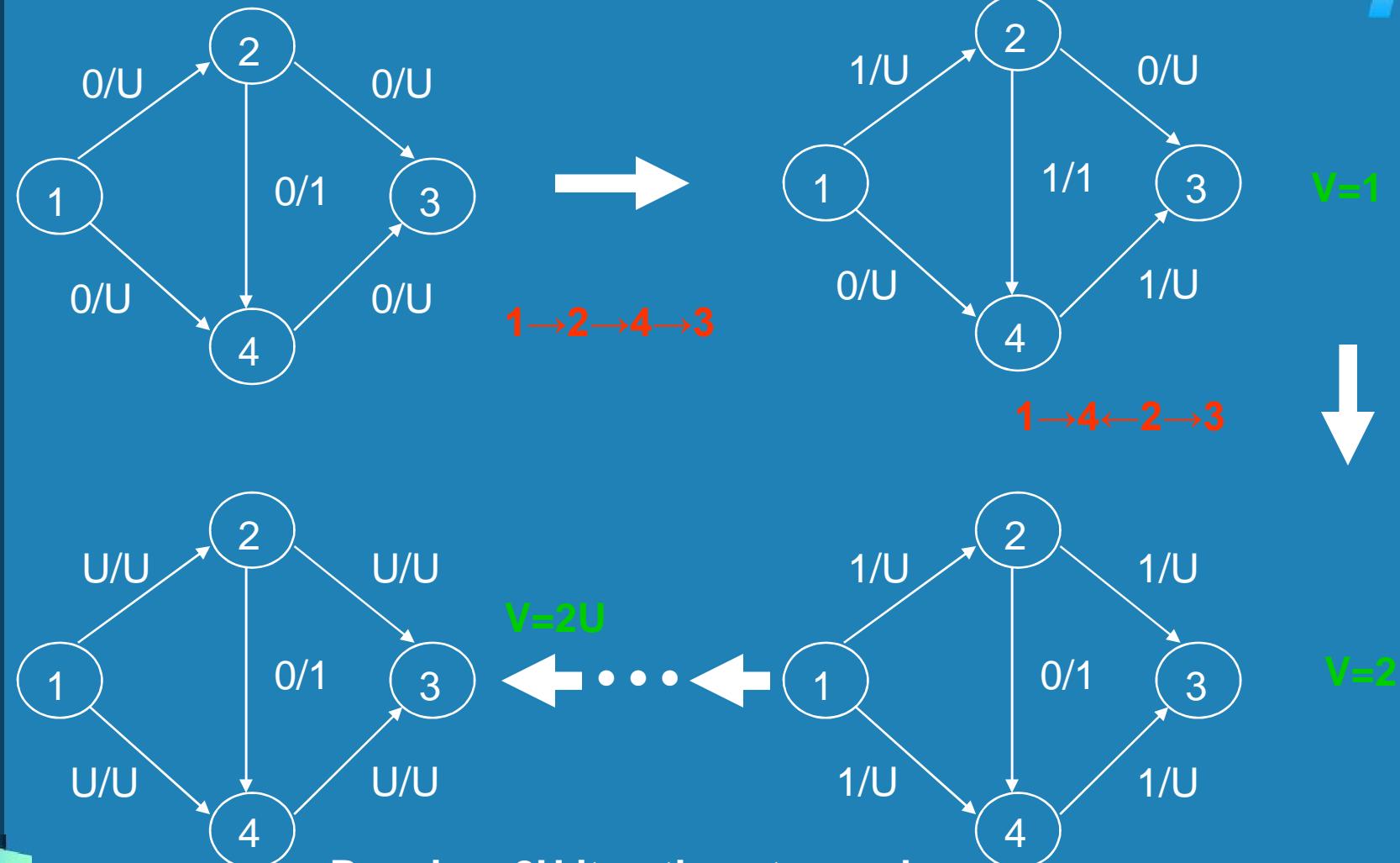
- ❑ The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- ❑ Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example 2



$U = \text{large positive integer}$

Example 2 (cont.)



Requires $2U$ iterations to reach maximum flow of value $2U$

Shortest-Augmenting-Path Algorithm



Generate augmenting path with the least number of edges by BFS as follows.

Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:

- **first label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled**
- **second label – indicates the vertex from which the vertex being labeled was reached, with “+” or “-” added to the second label to indicate whether the vertex was reached via a forward or backward edge**

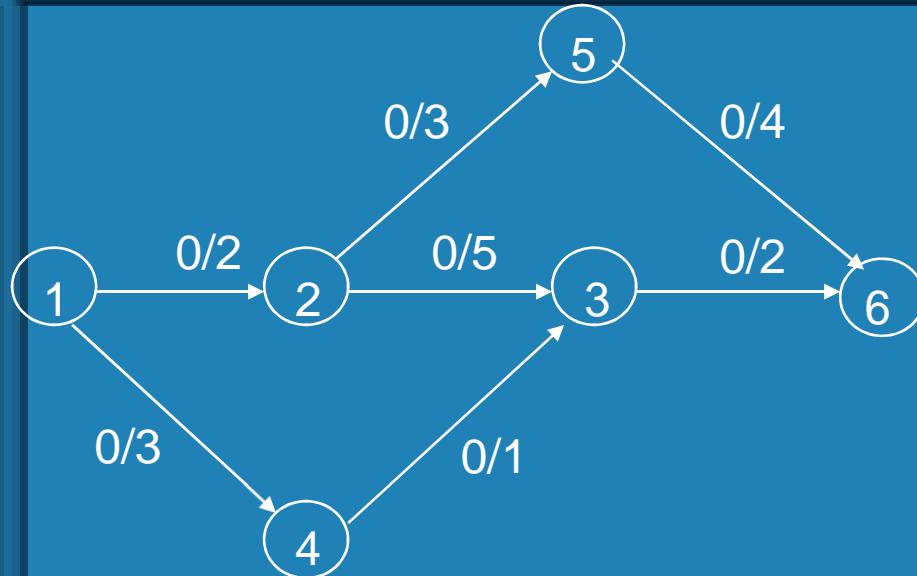
Vertex labeling

- ∅ The source is always labeled with $\infty, -$
- ∅ All other vertices are labeled as 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 $l_j = \min\{l_i, x_{ji}\}$

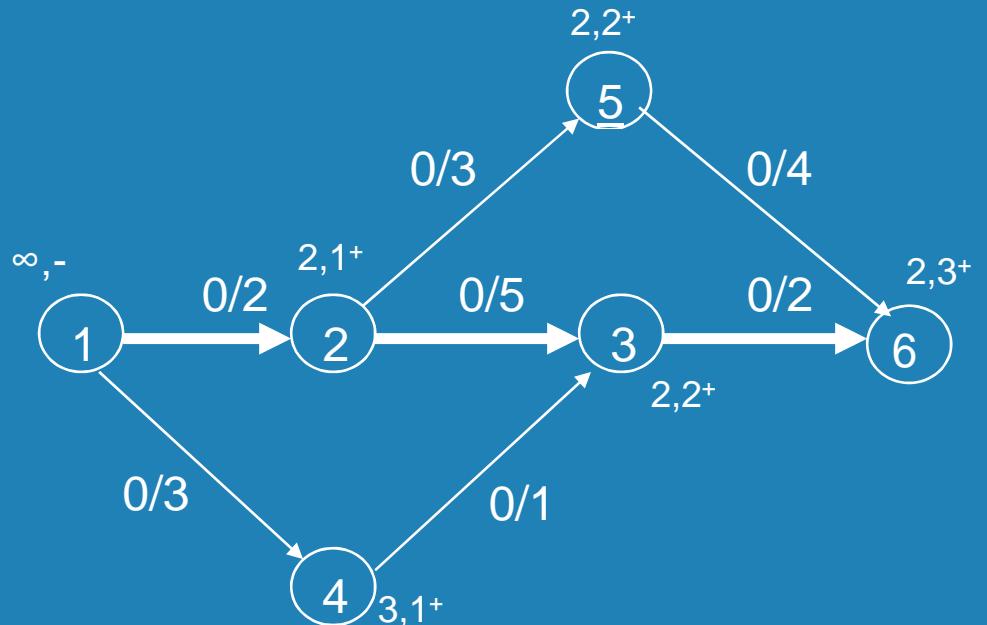
Vertex labeling (cont.)

- ② If the sink ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label
- ② 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
- ② If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops

Example: Shortest-Augmenting-Path Algorithm

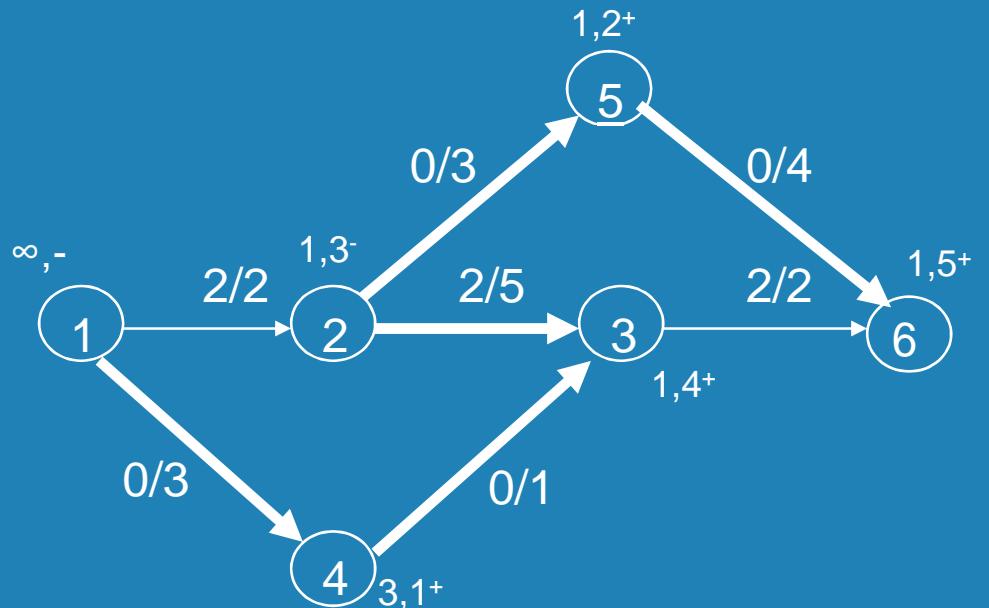
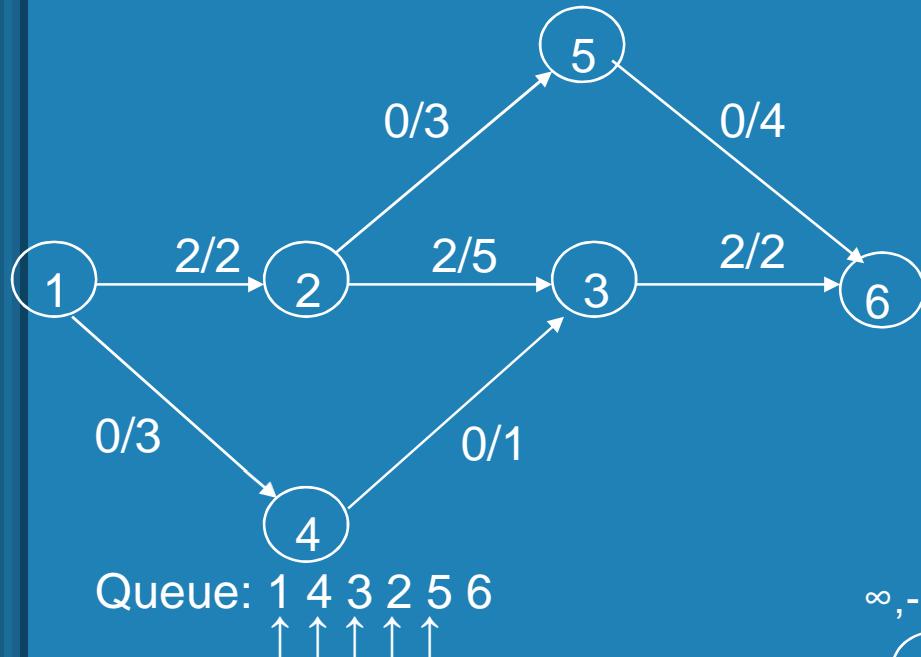


Queue: 1 2 4 3 5 6
 ↑↑↑↑↑



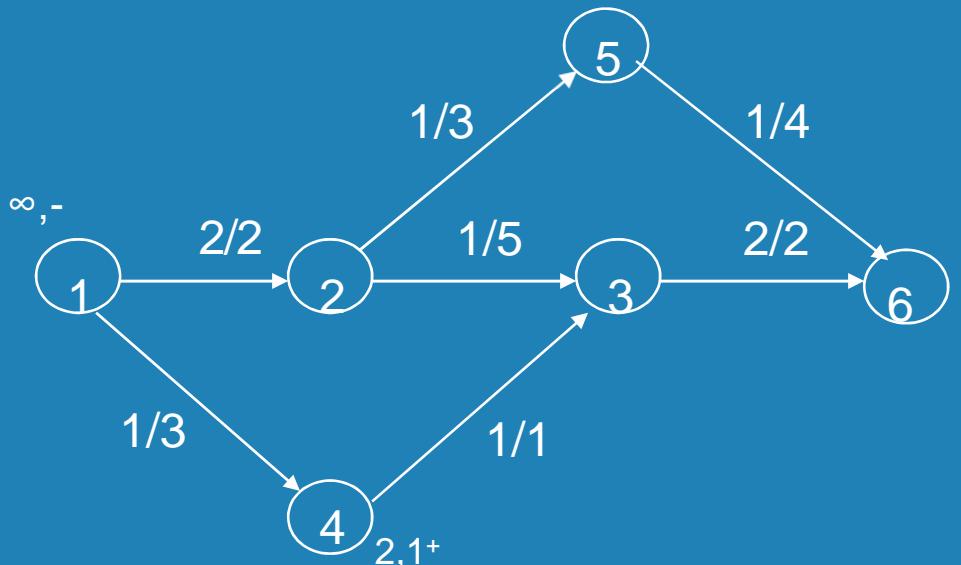
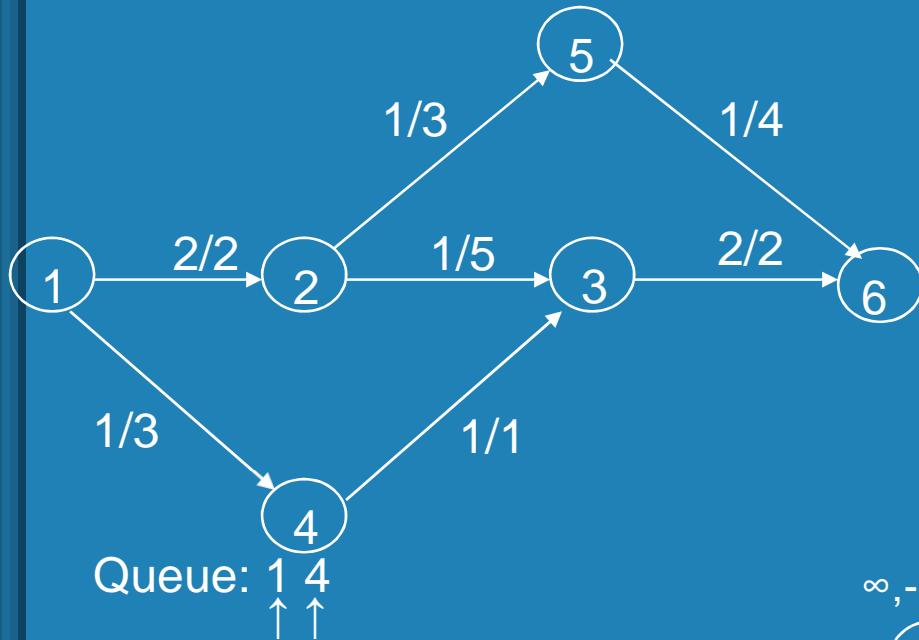
Augment the flow by 2 (the sink's first label) along the path 1 → 2 → 3 → 6

Example (cont.)



Augment the flow by 1 (the sink's first label) along the path $1 \rightarrow 4 \rightarrow 3 \leftarrow 2 \rightarrow 5 \rightarrow 6$

Example (cont.)



No augmenting path (the sink is unlabeled)
the current flow is maximum

Definition of a Cut

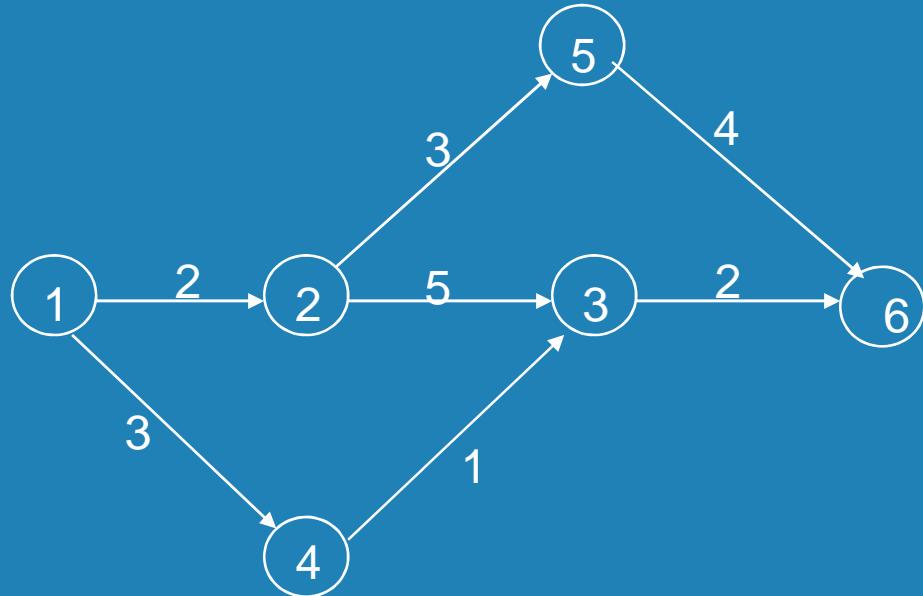


Let X be a set of vertices in a network that includes its source but does not include its sink, and let \bar{X} , the complement of X , be the rest of the vertices including the sink. The *cut* induced by this partition of the vertices is the set of all the edges with a tail in X and a head in \bar{X}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

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

- ❑ We'll denote a cut and its capacity by $C(X, \bar{X})$ and $c(X, \bar{X})$
- ❑ Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- ❑ *Minimum cut* is a cut of the smallest capacity in a given network

Examples of network cuts



If $X = \{1\}$ and $\bar{X} = \{2,3,4,5,6\}$, $C(X, \bar{X}) = \{(1,2), (1,4)\}$, $c = 5$

If $X = \{1,2,3,4,5\}$ and $\bar{X} = \{6\}$, $C(X, \bar{X}) = \{(3,6), (5,6)\}$, $c = 6$

If $X = \{1,2,4\}$ and $\bar{X} = \{3,5,6\}$, $C(X, \bar{X}) = \{(2,3), (2,5), (4,3)\}$, $c = 9$

Max-Flow Min-Cut Theorem

- ∅ The value of maximum flow in a network is equal to the capacity of its minimum cut
- ∅ The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:
 - maximum flow is the final flow produced by the algorithm
 - minimum cut is formed by all the edges from the labeled vertices to unlabeled 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

Shortest-augmenting-path algorithm

Input: A network with single source 1, single sink n , and positive integer capacities u_{ij} on its edges (i, j)

Output: A maximum flow x

assign $x_{ij} = 0$ to every edge (i, j) in the network

label the source with ∞ , – and add the source to the empty queue Q

while not $Empty(Q)$ **do**

$i \leftarrow Front(Q); Dequeue(Q)$

for every edge from i to j **do** //forward edges

if j is unlabeled

$r_{ij} \leftarrow u_{ij} - x_{ij}$

if $r_{ij} > 0$

$l_j \leftarrow \min\{l_i, r_{ij}\}$; label j with l_j, i^+

$Enqueue(Q, j)$

for every edge from j to i **do** //backward edges

if j is unlabeled

if $x_{ji} > 0$

$l_j \leftarrow \min\{l_i, x_{ji}\}$; label j with l_j, i^-

$Enqueue(Q, j)$

if the sink has been labeled

 //augment along the augmenting path found

$j \leftarrow n$ //start at the sink and move backwards using second labels

while $j \neq 1$ //the source hasn't been reached

if the second label of vertex j is i^+

$x_{ij} \leftarrow x_{ij} + l_n$

else //the second label of vertex j is i^-

$x_{ji} \leftarrow x_{ji} - l_n$

$j \leftarrow i$

 erase all vertex labels except the ones of the source

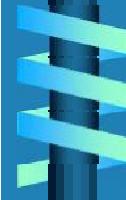
 reinitialize Q with the source

return x //the current flow is maximum

Time Efficiency



- ② 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
- ② 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
- ② More efficient algorithms have been found that can run in close to $O(nm)$ time, but these algorithms don't fall into the iterative-improvement paradigm

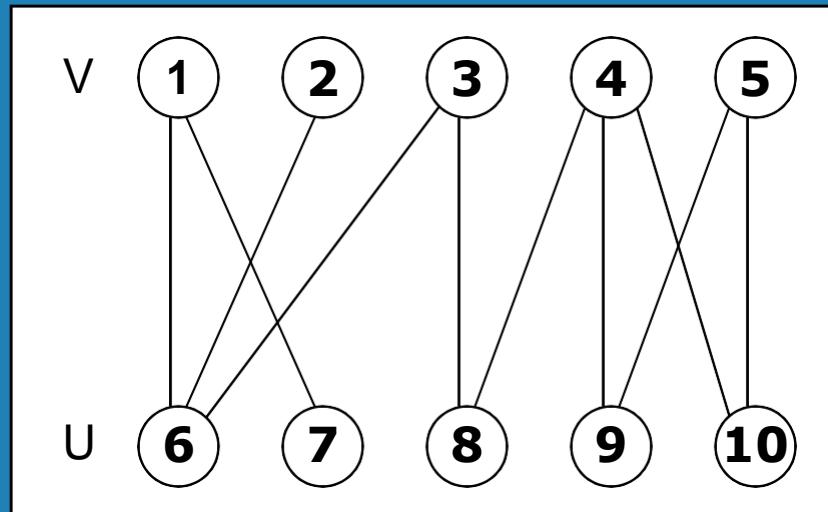


Bipartite Graphs



Bipartite graph: a graph whose vertices can be partitioned into two disjoint sets V and U , not necessarily of the same size, so that every edge connects a vertex in V to a vertex in U

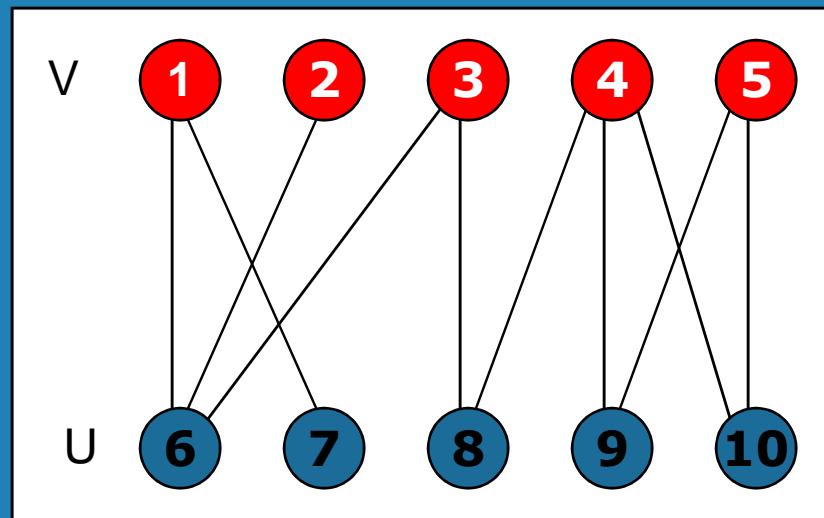
A graph is bipartite if and only if it does not have a cycle of an odd length



Bipartite Graphs (cont.)



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

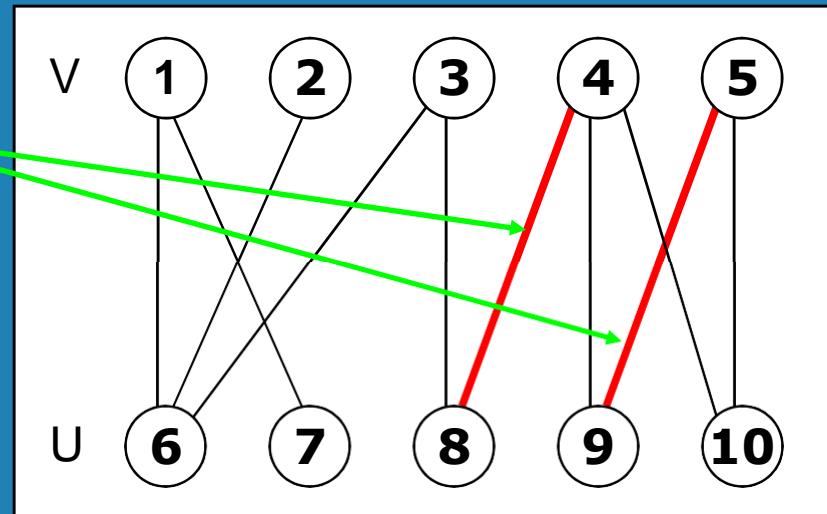


Matching in a Graph



A *matching* in a graph is a subset of its edges with the property that no two edges share a vertex

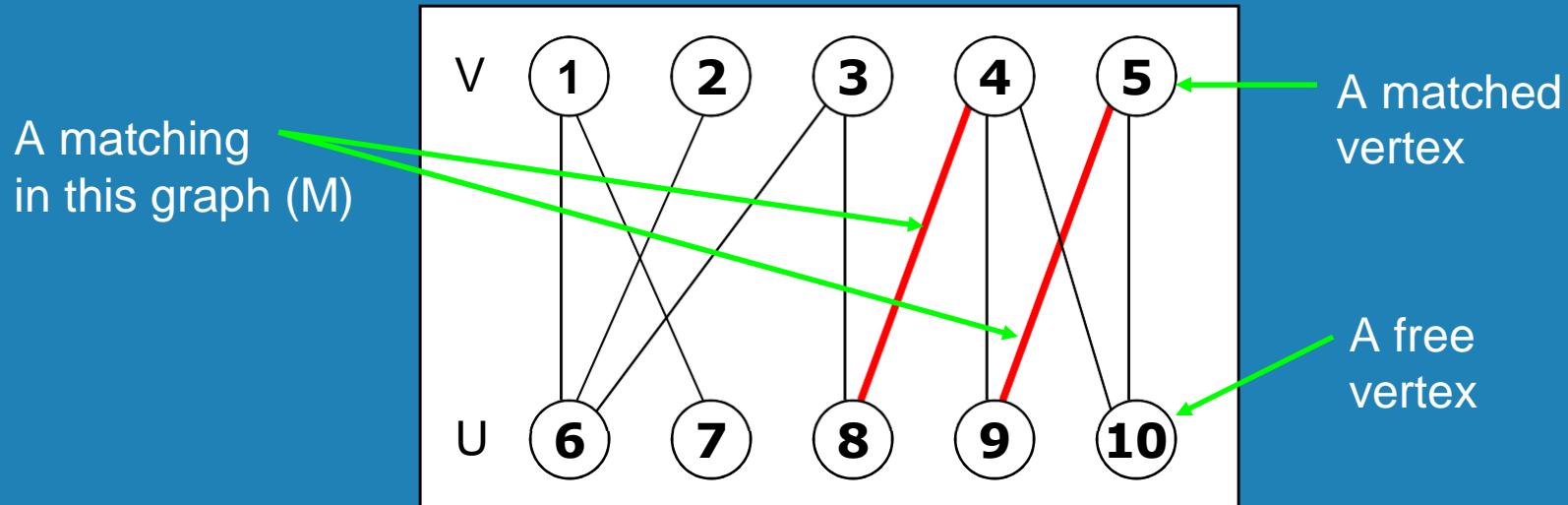
a matching
in this graph
 $M = \{(4,8), (5,9)\}$



A *maximum (or maximum cardinality) matching* is a matching with the largest number of edges

- always exists
- not always unique

Free Vertices and Maximum Matching



For a given matching M , a vertex is called *free* (or *unmatched*) if it is not an endpoint of any edge in M ; otherwise, a vertex is said to be *matched*

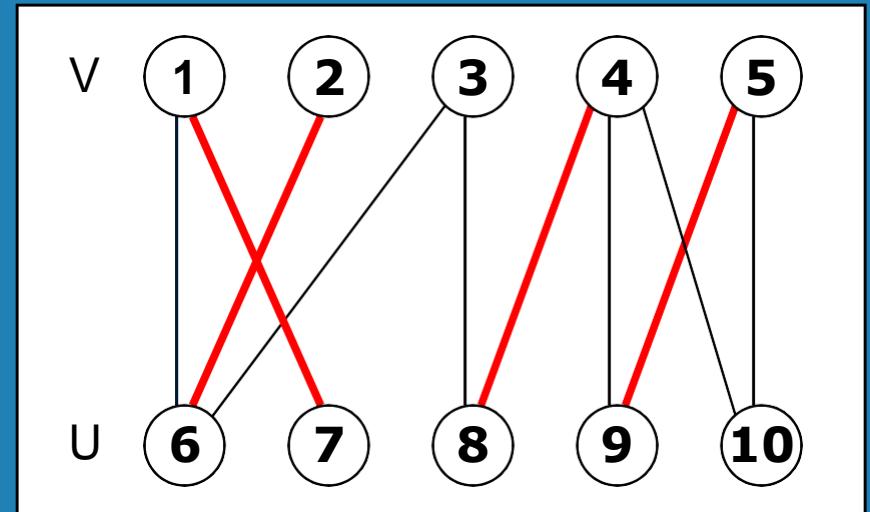
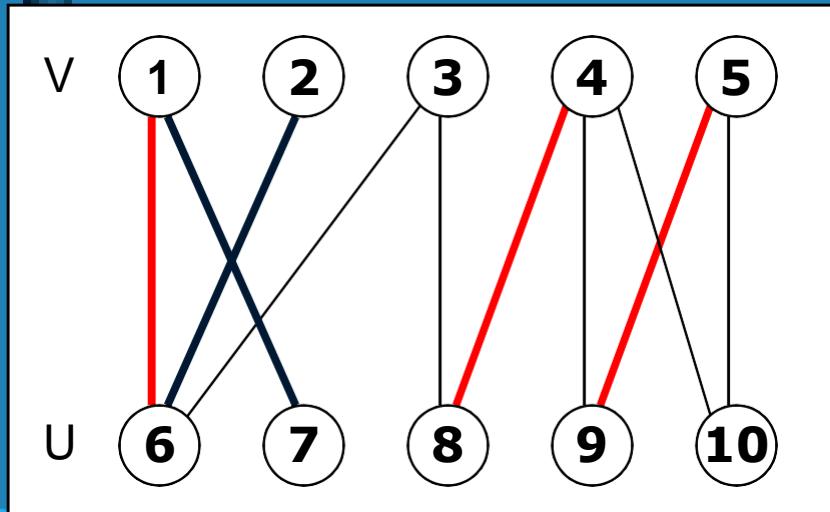
- If every vertex is matched, then M is a maximum matching
- If there are unmatched or free vertices, then M may be able to be improved
- We can immediately increase a matching by adding an edge connecting two free vertices (e.g., (1,6) above)

Augmenting Paths and Augmentation



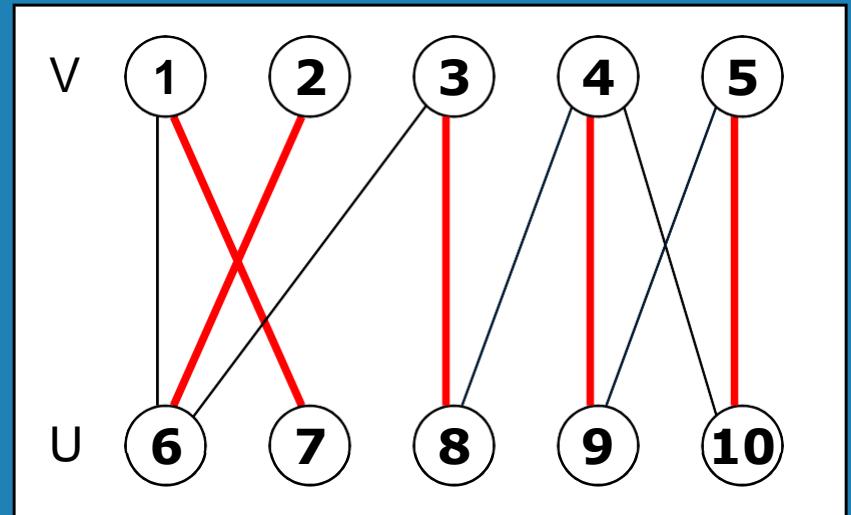
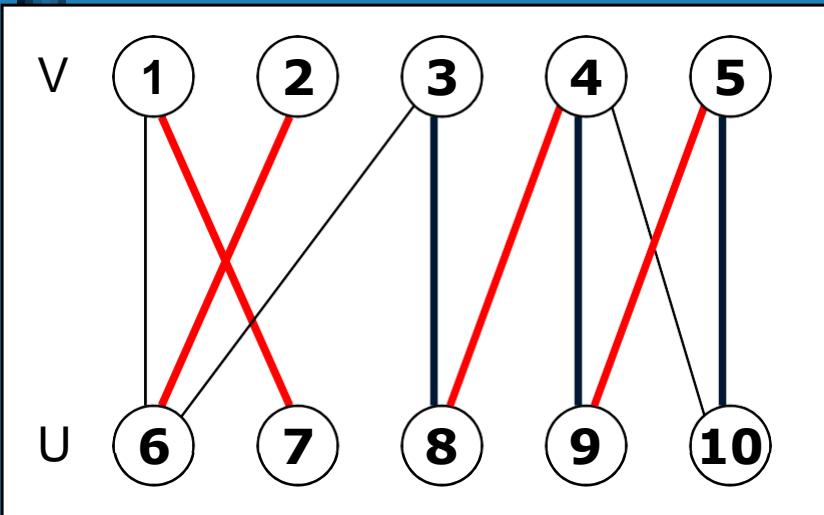
An *augmenting path* for a matching M is a path from a free vertex in V to a free vertex in U whose edges alternate between edges not in M and edges in M

- ∅ The length of an augmenting path is always odd
- ∅ Adding to M the odd numbered path edges and deleting from it the even numbered path edges increases the matching size by 1 (*augmentation*)
- ∅ One-edge path between two free vertices is special case of augmenting path



Augmentation along path 2,6,1,7

Augmenting Paths (another example)



Augmentation along
3, 8, 4, 9, 5, 10

- Matching on the right is maximum (*perfect matching*)
- Theorem A matching M is maximum if and only if there exists no augmenting path with respect to M

Augmenting Path Method (template)

- ② Start with some initial matching
 - e.g., the empty set
- ② Find an augmenting path and augment the current matching along that path
 - e.g., using breadth-first search like method
- ② When no augmenting path can be found, terminate and return the last matching, which is maximum

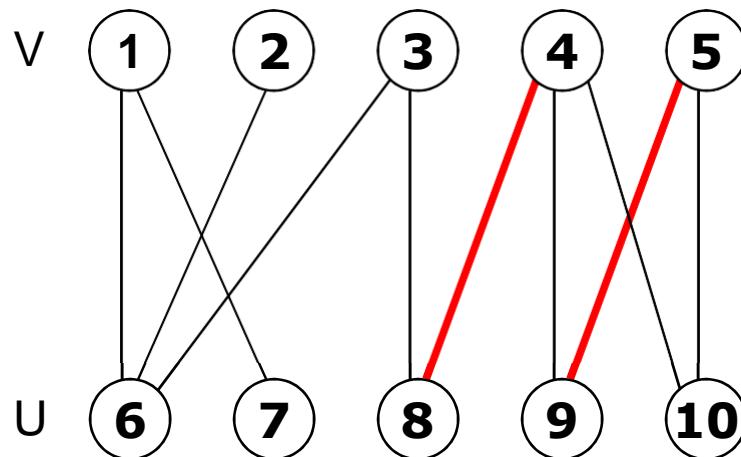
BFS-based Augmenting Path Algorithm



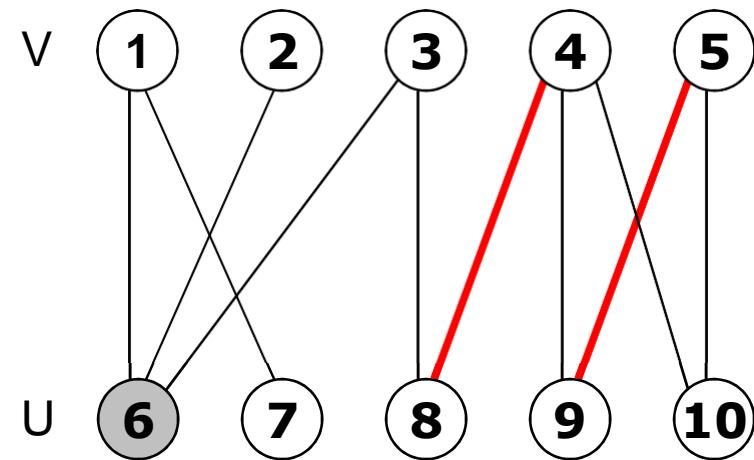
- ❑ Initialize queue Q with all free vertices in one of the sets (say V)
- ❑ While Q is not empty, delete front vertex w and label every unlabeled vertex u adjacent to w as follows:
 - Case 1 (w is in V)
 - If u is free, augment the matching along the path ending at u by moving backwards until a free vertex in V is reached.
 - After that, erase all labels and reinitialize Q with all the vertices in V that are still free
 - If u is matched (not with w), label u with w and enqueue u
 - Case 2 (w is in U) Label its matching mate v with w and enqueue v
- ❑ After Q becomes empty, return the last matching, which is maximum

Example (revisited)

Initial Graph



Resulting Graph



Queue: 1 2 3

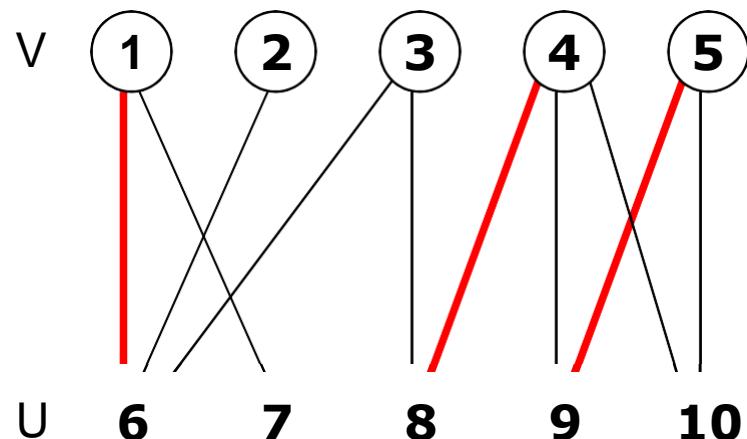
Queue: 1 2 3

Augment
from 6

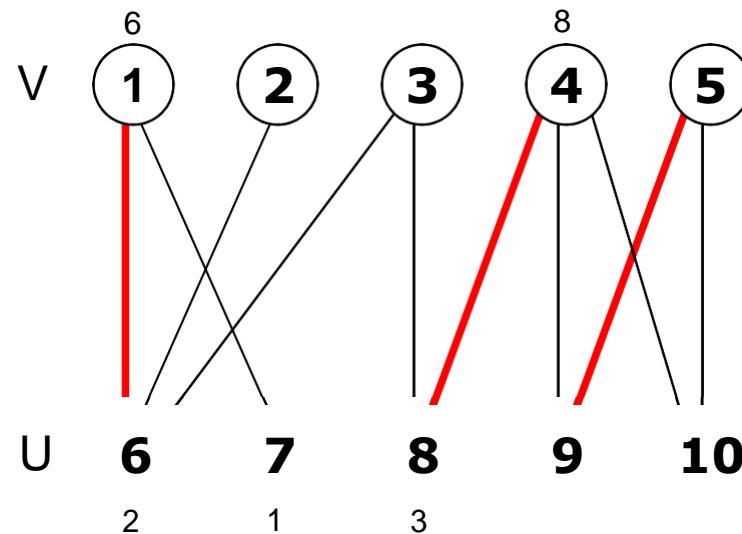
Each vertex is labeled with the vertex it was reached from. Queue deletions are indicated by arrows. The free vertex found in U is shaded and labeled for clarity; the new matching obtained by the augmentation is shown on the next slide.

Example (cont.)

Initial Graph



Resulting Graph



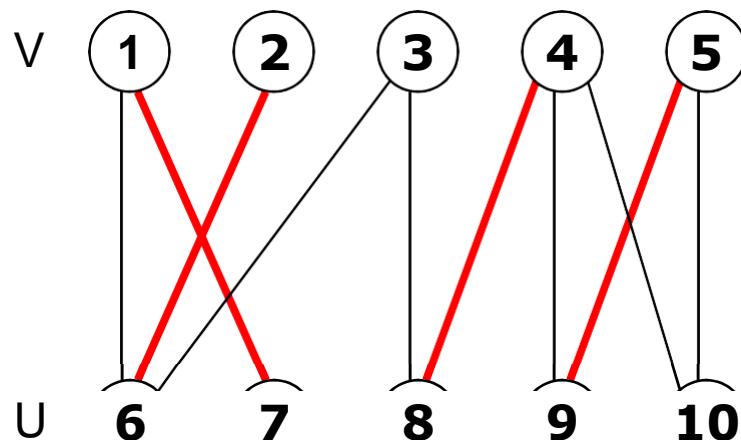
Queue: 2 3

Queue: 2 3 6 8 1 4

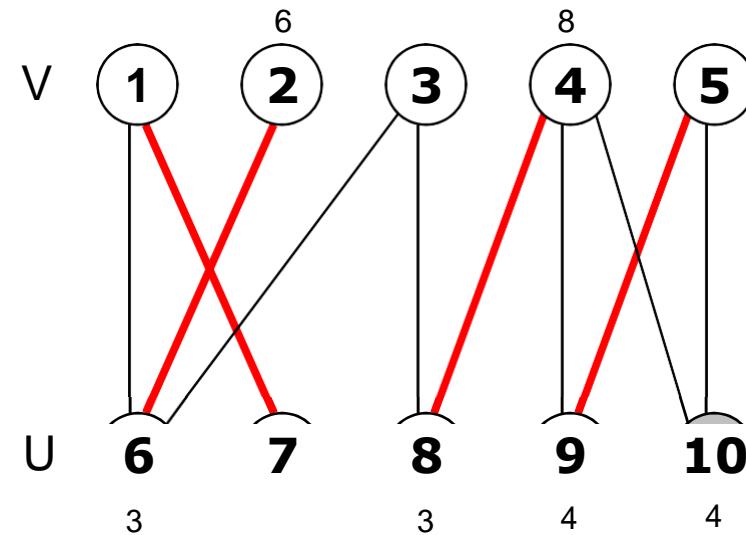
Augment
from 7

Example (cont.)

Initial Graph



Resulting Graph



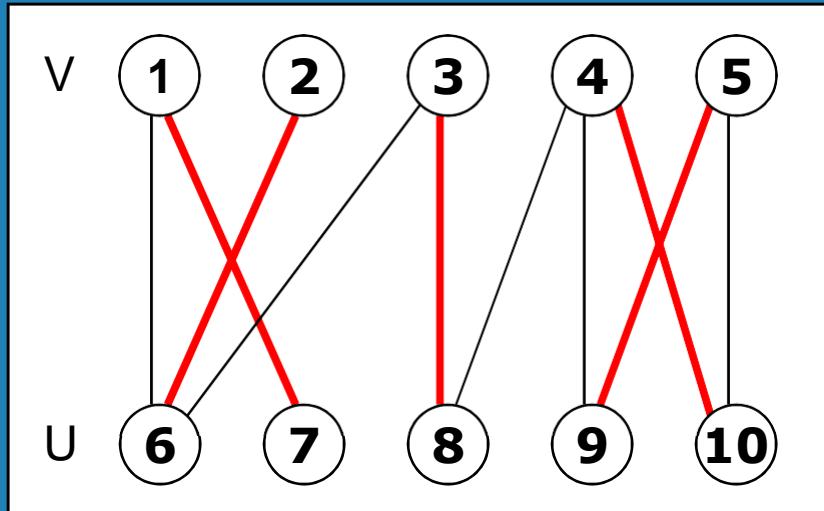
Queue: 3

Queue: 3 6 8 2 4 9

Augment
from 10



Example: maximum matching found



maximum
matching

- ∅ This matching is maximum since there are no remaining free vertices in V (the queue is empty)
- ∅ Note that this matching differs from the maximum matching found earlier

Maximum-matching algorithm for bipartite graphs

Input: A bipartite graph $G = \langle V, U, E \rangle$

Output: A maximum-cardinality matching M in the input graph

initialize set M of edges with some valid matching (e.g., the empty set)

initialize queue Q with all the free vertices in V (in any order)

while not $\text{Empty}(Q)$ **do**

$w \leftarrow \text{Front}(Q); \quad \text{Dequeue}(Q)$

if $w \in V$

for every vertex u adjacent to w **do**

if u is free

 //augment

$M \leftarrow M \cup (w, u)$

$v \leftarrow w$

while v is labeled **do**

$u \leftarrow$ vertex indicated by v 's label; $M \leftarrow M - (v, u)$

$v \leftarrow$ vertex indicated by u 's label; $M \leftarrow M \cup (v, u)$

 remove all vertex labels

 reinitialize Q with all free vertices in V

break //exit the for loop

else // u is matched

if $(w, u) \notin M$ **and** u is unlabeled

 label u with w

$\text{Enqueue}(Q, u)$

else // $w \in U$ (and matched)

 label the mate v of w with “ w ”

$\text{Enqueue}(Q, v)$

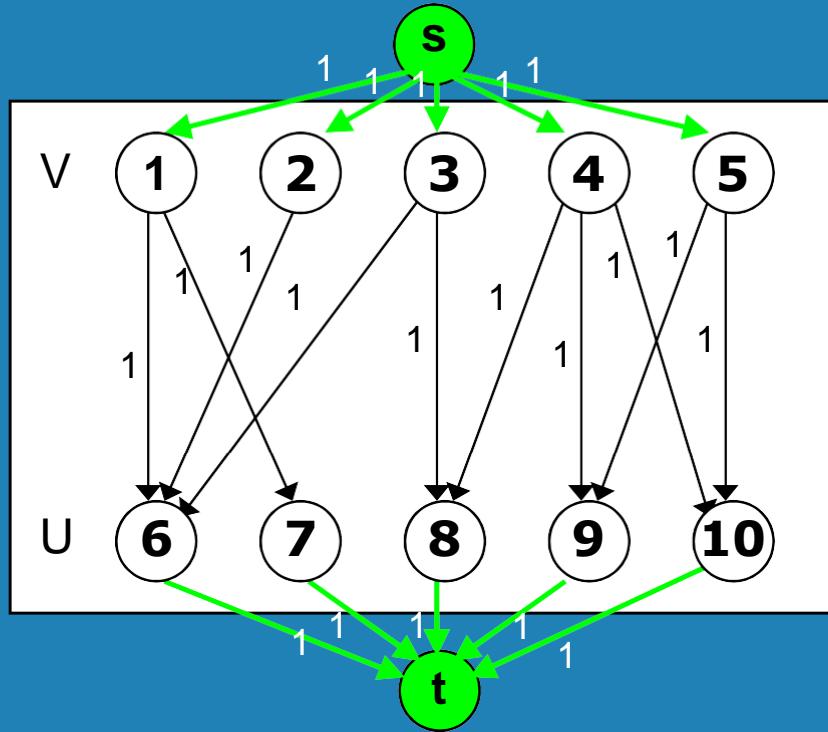
return M //current matching is maximum

Notes on Maximum Matching Algorithm



- ❑ Each iteration (except the last) matches two free vertices (one each from V and U). Therefore, the number of iterations cannot exceed $\lfloor n/2 \rfloor + 1$, where n is the number of vertices in the graph. The time spent on each iteration is in $O(n+m)$, where m is the number of edges in the graph. Hence, the time efficiency is in $O(n(n+m))$
- ❑ This can be improved to $O(\sqrt{n}(n+m))$ by combining multiple iterations to maximize the number of edges added to matching M in each search
- ❑ Finding a maximum matching in an arbitrary graph is much more difficult, but the problem was solved in 1965 by Jack Edmonds

Conversion to Max-Flow Problem



- ② Add a source and a sink, direct edges (with unit capacity) from the source to the vertices of V and from the vertices of U to the sink
- ② Direct all edges from V to U with unit capacity

Stable Marriage Problem



There is a set $Y = \{m_1, \dots, m_n\}$ of n men and a set $X = \{w_1, \dots, w_n\}$ 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 *marriage matching* M is a set of 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 .

A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.

The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

Instance of the Stable Marriage Problem



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.

men's preferences

	1 st	2 nd	3 rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

women's preferences

	1 st	2 nd	3 rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

ranking matrix

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

{(Bob, Ann) (Jim, Lea) (Tom, Sue)} is unstable
{(Bob, Ann) (Jim, Sue) (Tom, Lea)} is stable

Stable Marriage Algorithm (Gale-Shapley)



Step 0 Start with all the men and women being free

Step 1 While there are free men, arbitrarily select one of them and 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, she compares m with her current mate. If she prefers m to him, she accepts m 's proposal, making her former mate free; otherwise, she simply rejects m 's proposal, leaving m free

Step 2 Return the set of n matched pairs

Example

Free men:
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

Bob proposed to Lea
Lea accepted

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	<u>1,3</u>	2,1
Tom	3,2	2,1	1,2

Jim proposed to Lea
Lea rejected

Example (cont.)

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

Jim proposed to Sue
Sue accepted

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	<u>1,2</u>

Tom proposed to Sue
Sue rejected

Example (cont.)

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

Tom proposed to Lea
Lea replaced Bob
with Tom

Free men:
Bob

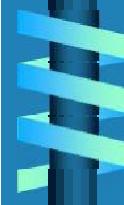
	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Ann
Ann accepted

Analysis of the Gale-Shapley Algorithm



- ∅ The algorithm terminates after no more than n^2 iterations with 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
- ∅ A man (woman) optimal matching is unique for a given set of participant preferences
- ∅ The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training



Tackling Difficult Combinatorial problems



There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):

- β **Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time**

- β **Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time**

Exact Solution Strategies



- β *exhaustive search* (brute force)
 - useful only for small instances
- β *dynamic programming*
 - applicable to some problems (e.g., the knapsack problem)
- β *backtracking*
 - eliminates some unnecessary cases from consideration
 - yields solutions in reasonable time for many instances but worst case is still exponential
- β *branch-and-bound*
 - further refines the backtracking idea for optimization problems



Backtracking



β Construct the state-space tree

- nodes: partial solutions
- edges: choices in extending partial solutions

β Explore the state space tree using depth-first search

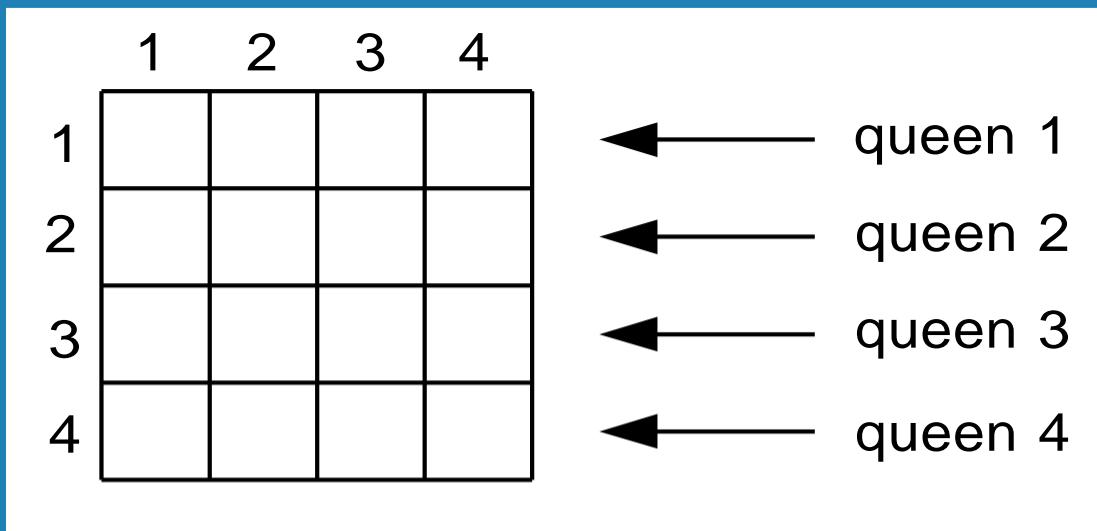
β “Prune” nonpromising nodes

- dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search

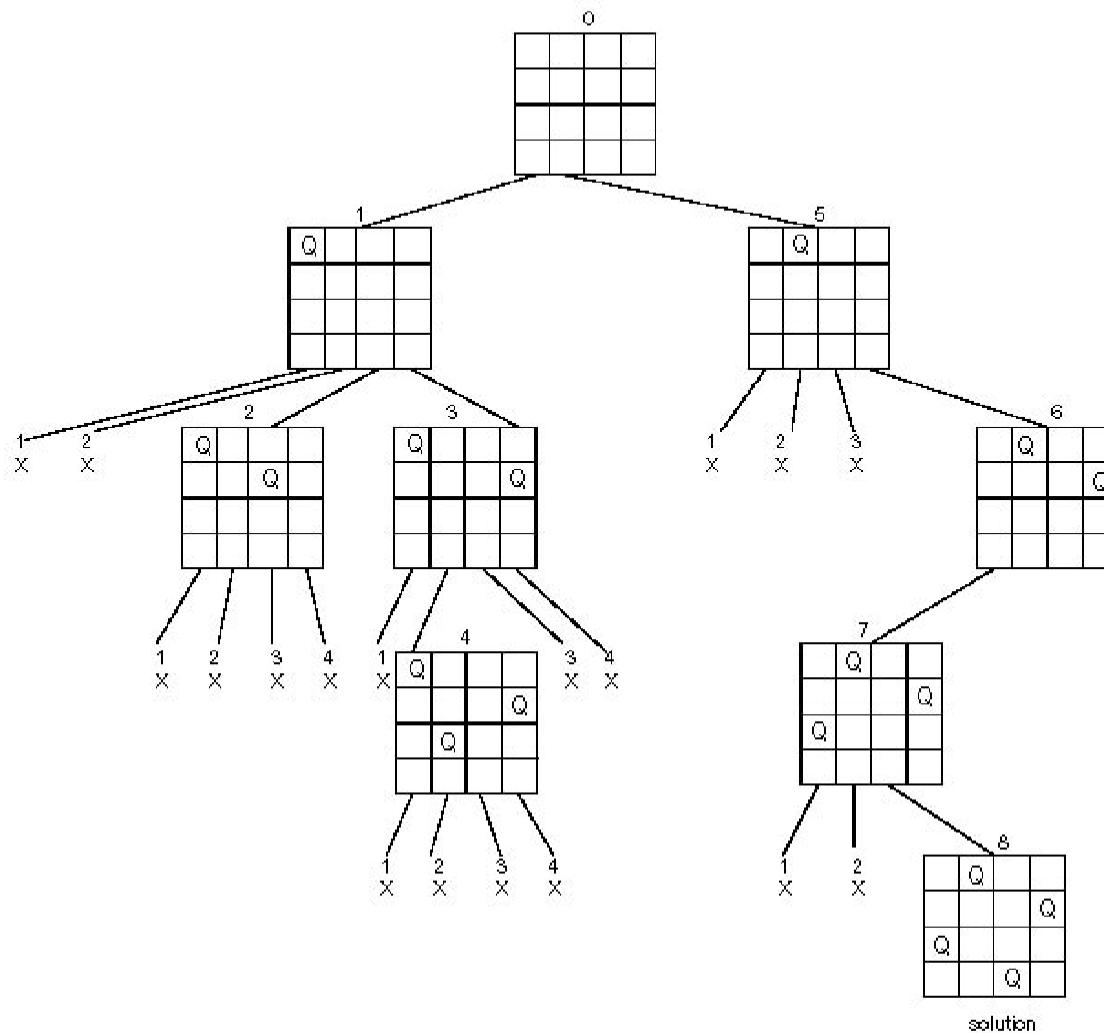


Example: *n*-Queens Problem

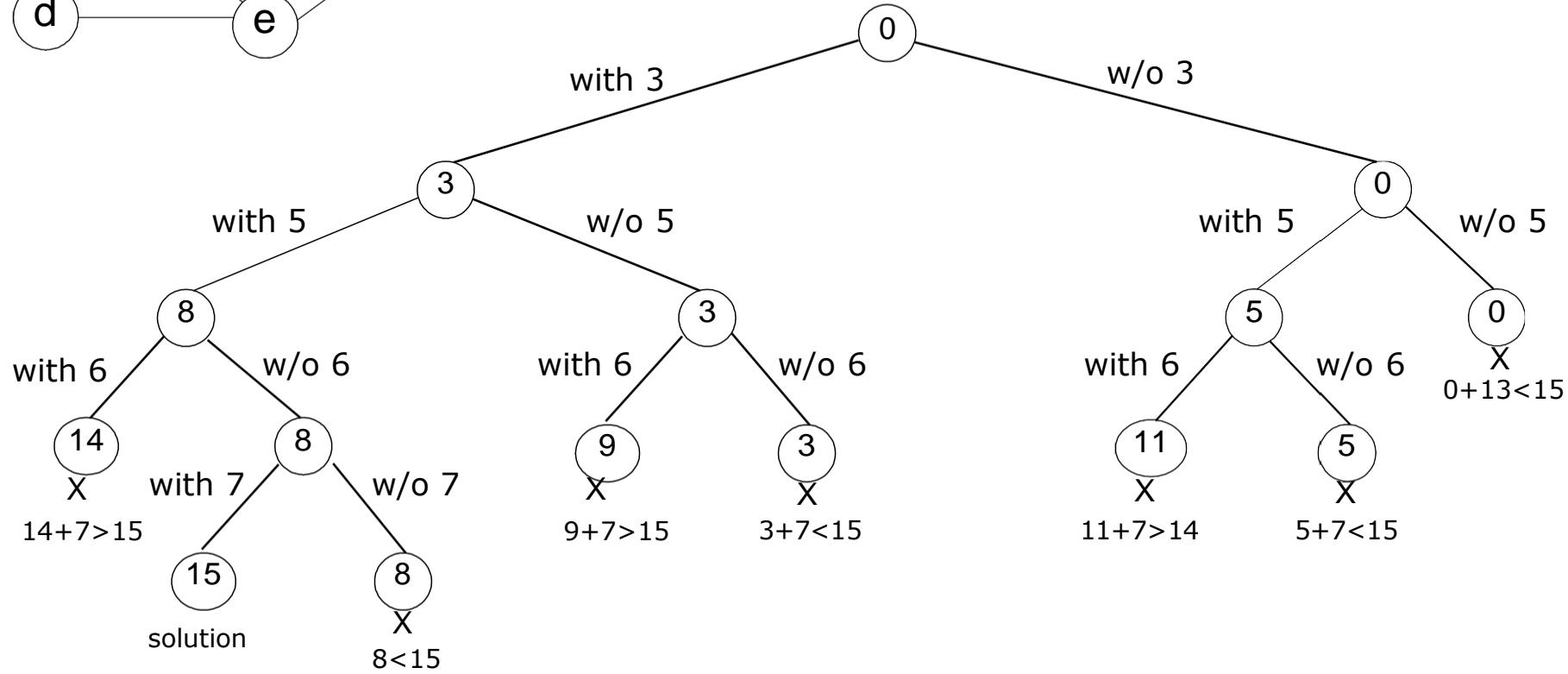
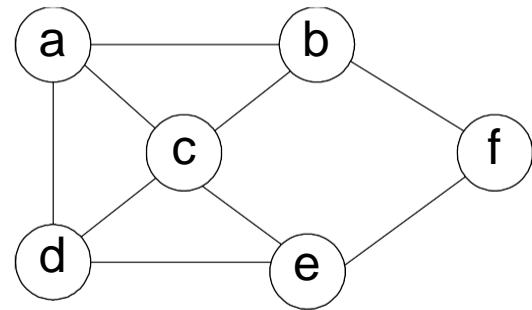
Place *n* queens on an *n*-by-*n* chess board so that no two of them are in the same row, column, or diagonal



State-Space Tree of the 4-Queens problems



Example: Hamiltonian Circuit problems



Branch-and-Bound



- β An enhancement of backtracking
- β Applicable to optimization problems
- β For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)
- β Uses the bound for:
 - ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far
 - guiding the search through state-space

Example: Assignment Problems



Select one element in each row of the cost matrix C so that:

- no two selected elements are in the same column
- the sum is minimized

Example

	Job 1	Job 2	Job 3	Job 4
Person <i>a</i>	9	2	7	8
Person <i>b</i>	6	4	3	7
Person <i>c</i>	5	8	1	8
Person <i>d</i>	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: $2 + 3 + 1 + 4$ (or $5 + 2 + 1 + 4$)

Example: First two levels of the state-space tree

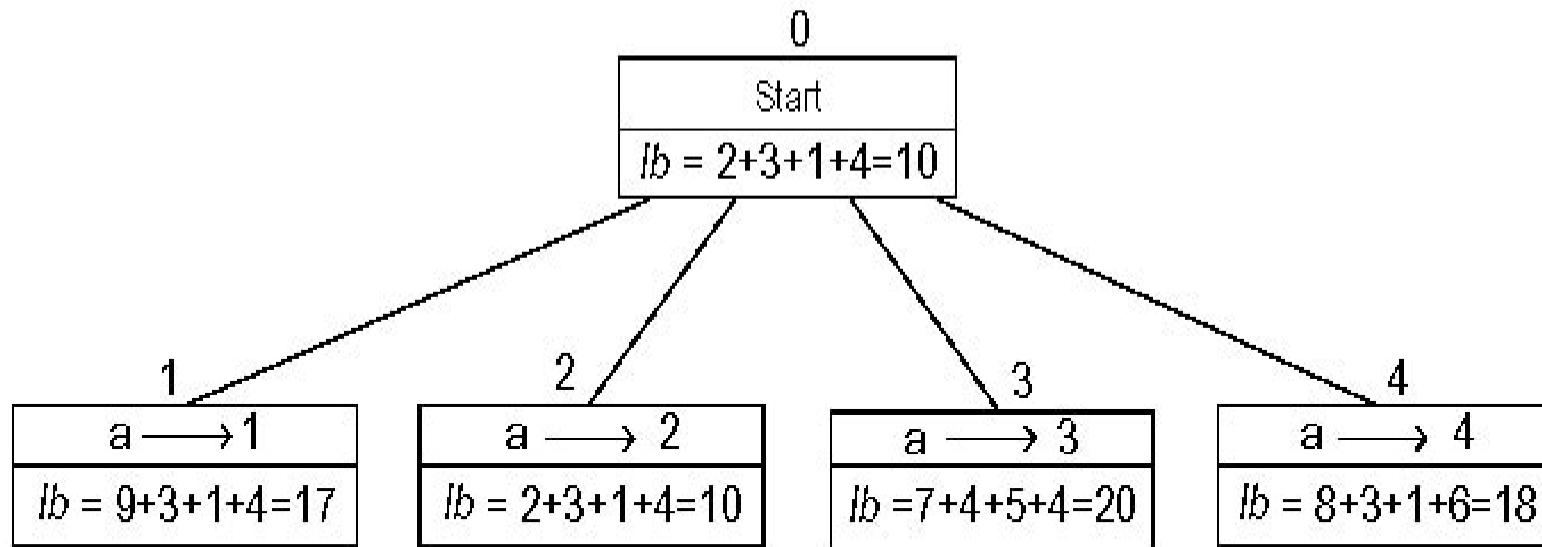


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb , for this node.

Example (cont.)

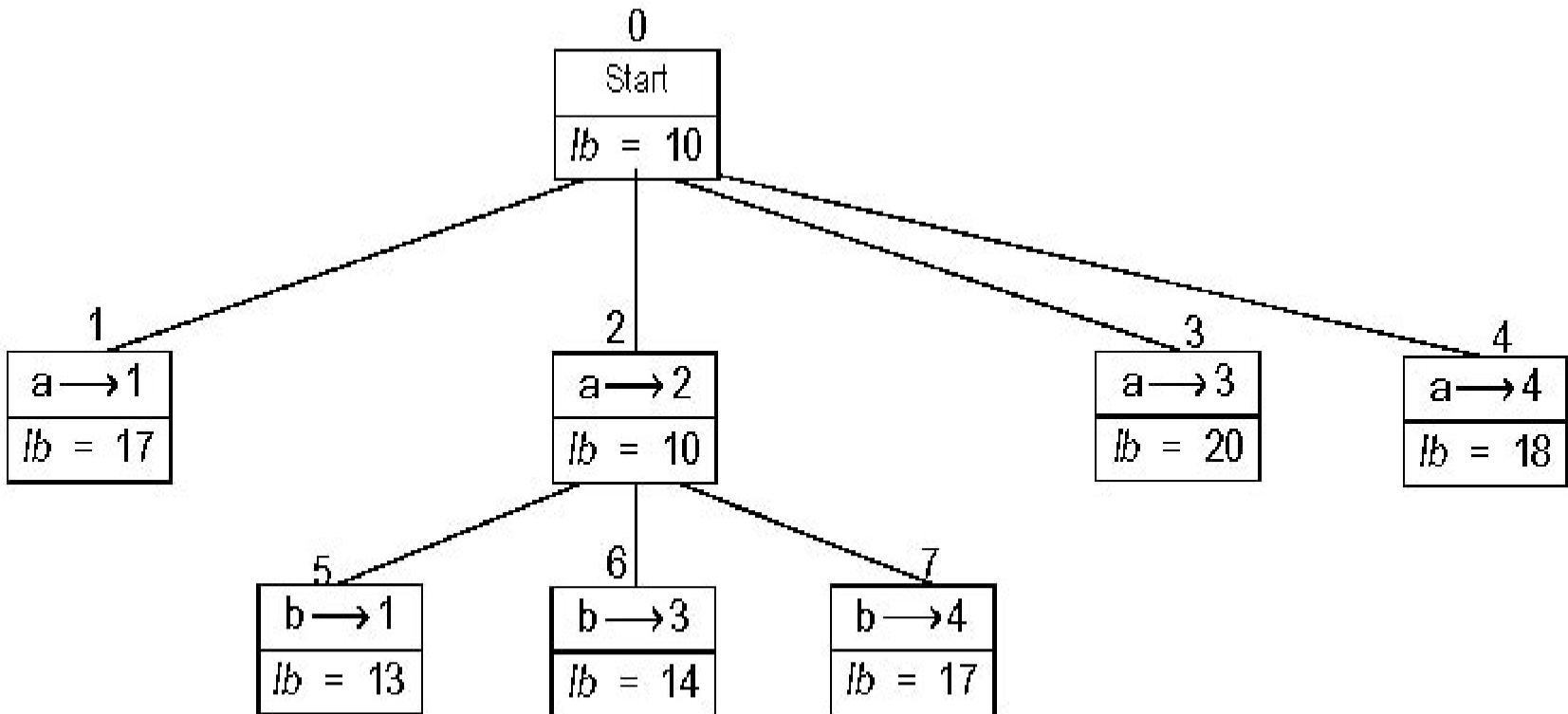


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Example: Complete state-space tree

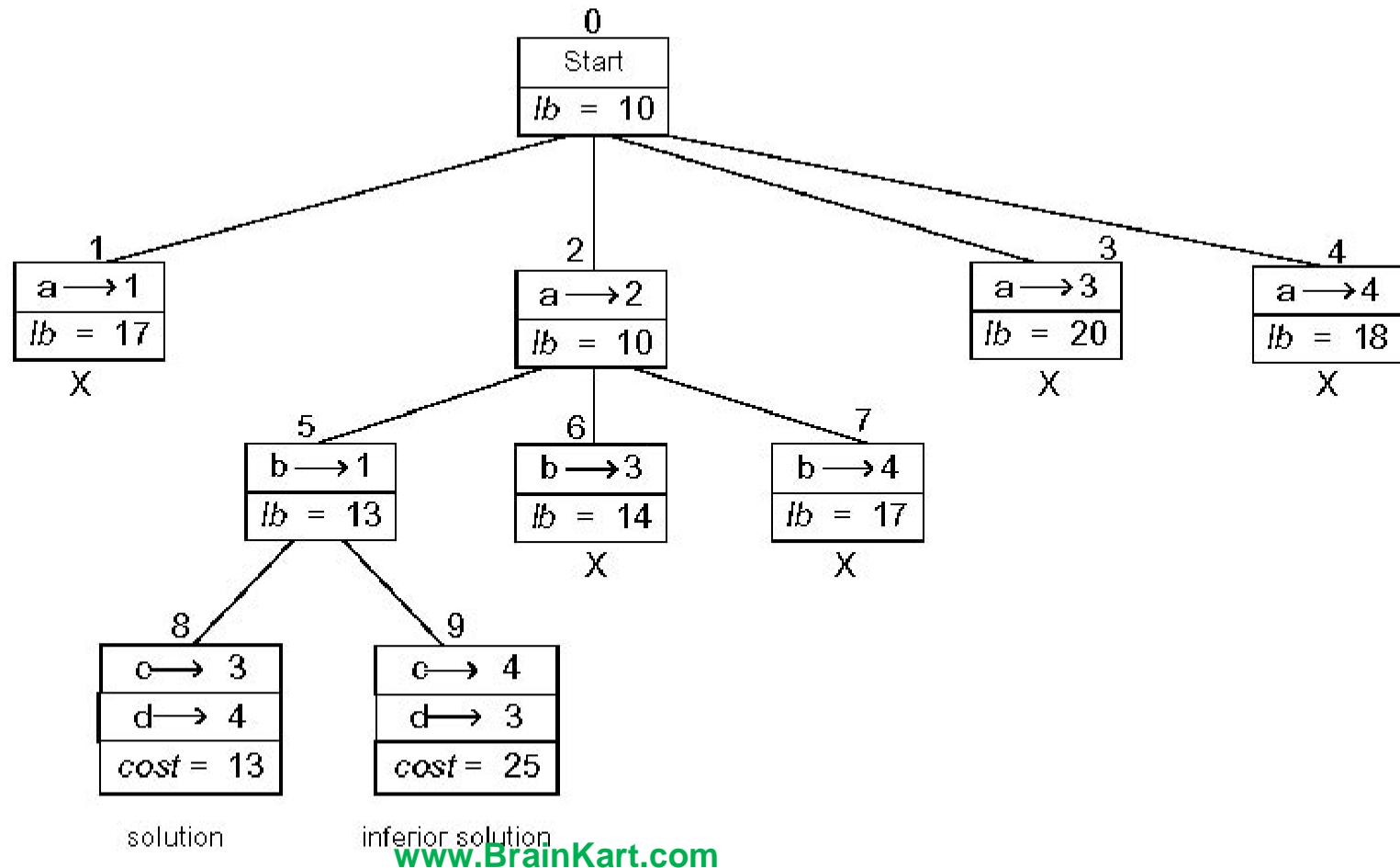
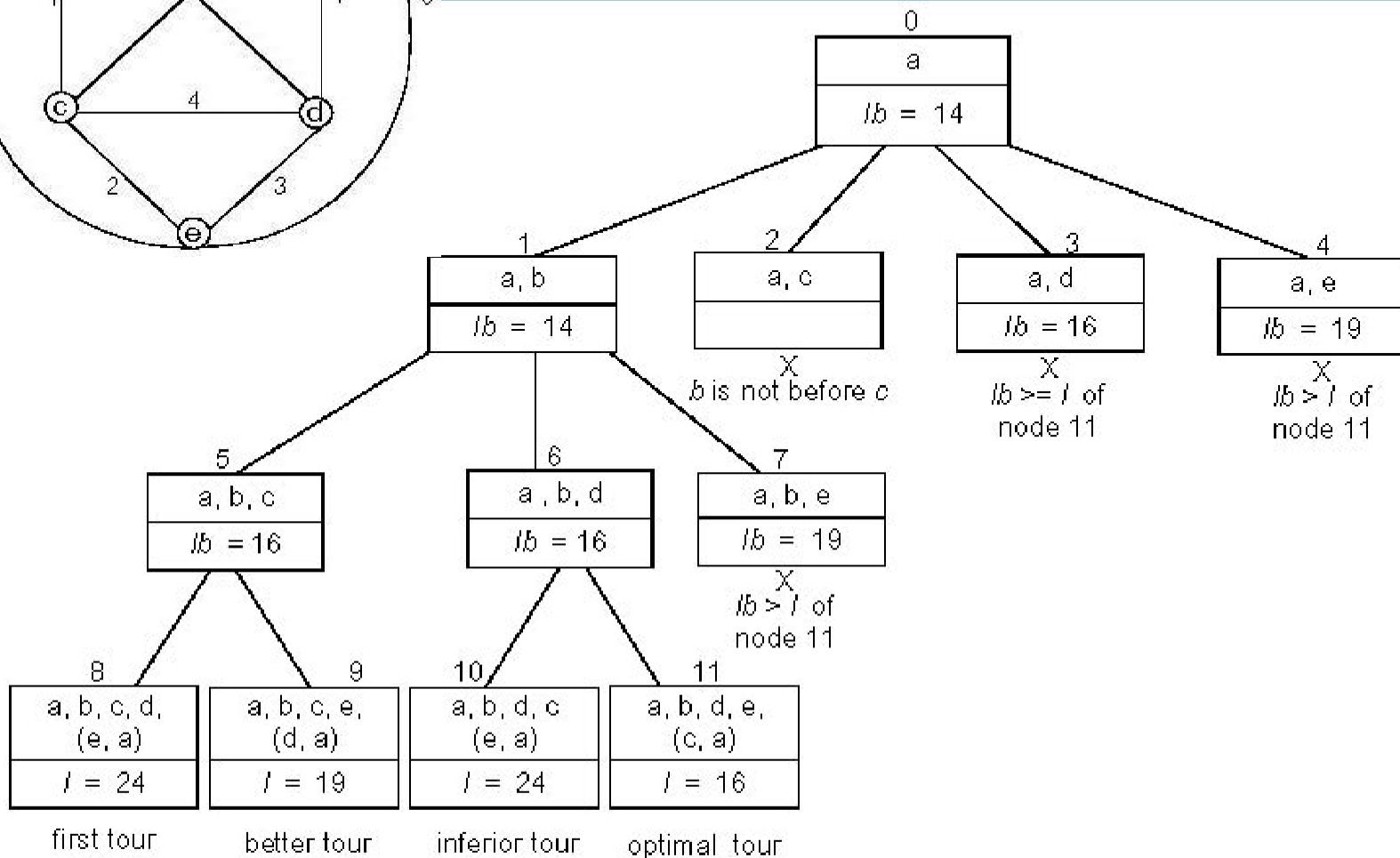
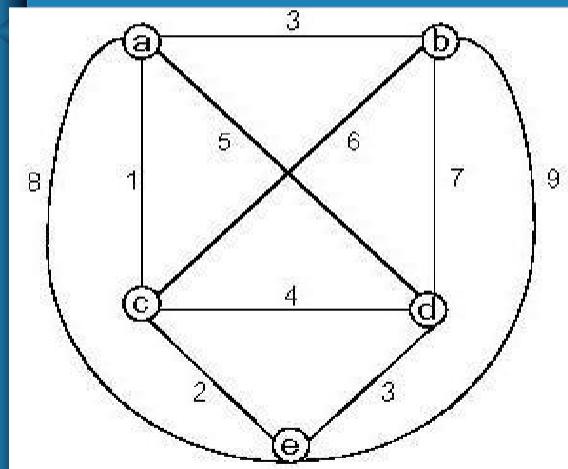


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm



Example: Traveling Salesman Problem



Approximation Approach



Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not necessarily optimal but hopefully close to it

Accuracy measures:

accuracy ratio of an approximate solution s_a

$r(s_a) = f(s_a) / f(s^*)$ for minimization problems

$r(s_a) = f(s^*) / f(s_a)$ for maximization problems

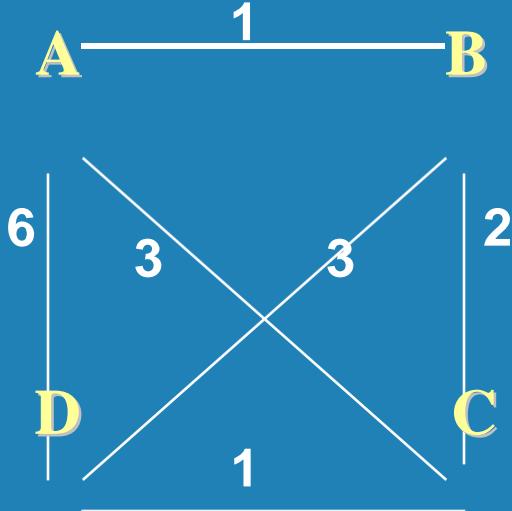
where $f(s_a)$ and $f(s^*)$ are values of the objective function f for the approximate solution s_a and actual optimal solution s^*

performance ratio of the algorithm A

the lowest upper bound of $r(s_a)$ on all instances

Nearest-Neighbor Algorithm for TSP

Starting at some city, always go to the nearest unvisited city,
and, after visiting all the cities, return to the starting one



s_a : A – B – C – D – A of length 10

s^* : A – B – D – C – A of length 8

Note: Nearest-neighbor tour may depend on the starting city

Accuracy: $R_A = \infty$ (unbounded above) – make the length of AD arbitrarily large in the above example

Multifragment-Heuristic Algorithm



Stage 1: Sort the edges in nondecreasing order of weights.

Initialize the set of tour edges to be constructed to empty set

Stage 2: Add next edge on the sorted list to the tour, skipping those whose addition would've created a vertex of degree 3 or a cycle of length less than n . Repeat this step until a tour of length n is obtained

Note: $R_A = \infty$, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Twice-Around-the-Tree Algorithm

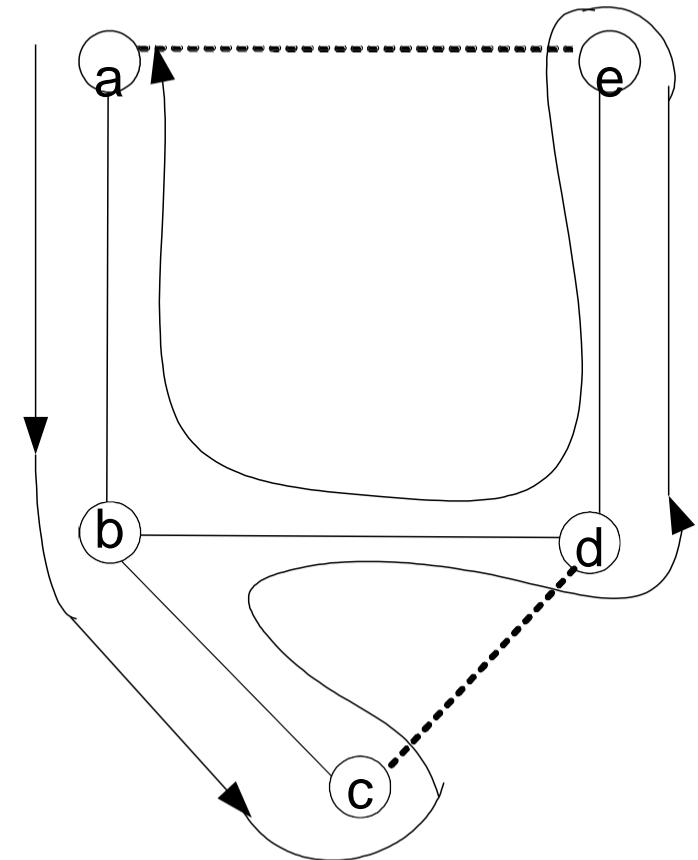
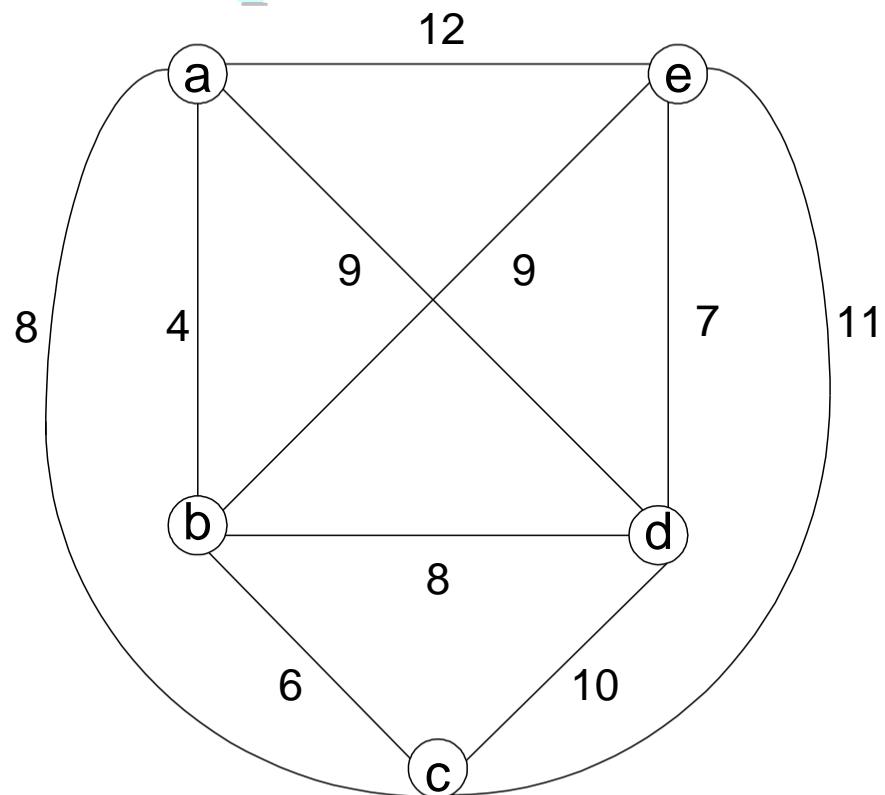
Stage 1: Construct a minimum spanning tree of the graph
(e.g., by Prim's or Kruskal's algorithm)

Stage 2: Starting at an arbitrary vertex, create a path that goes twice around the tree and returns to the same vertex

Stage 3: Create a tour from the circuit constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

Note: $R_A = \infty$ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Example



Walk: a – b – c – b – d – e – d – b – a

Tour: a – b – c – d – e – a

Christofides Algorithm

Stage 1: Construct a minimum spanning tree of the graph



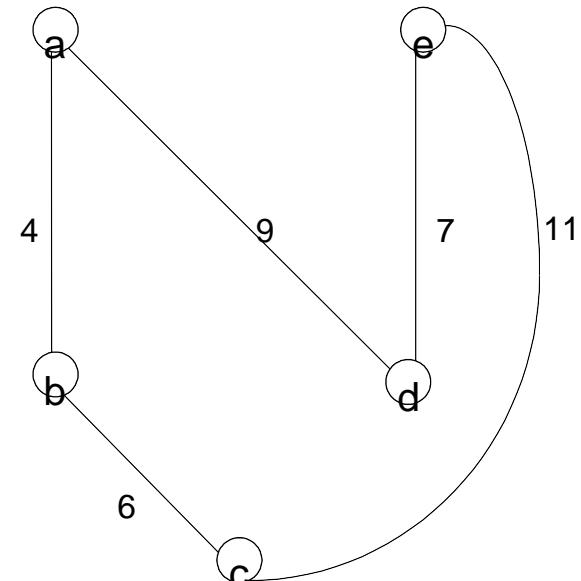
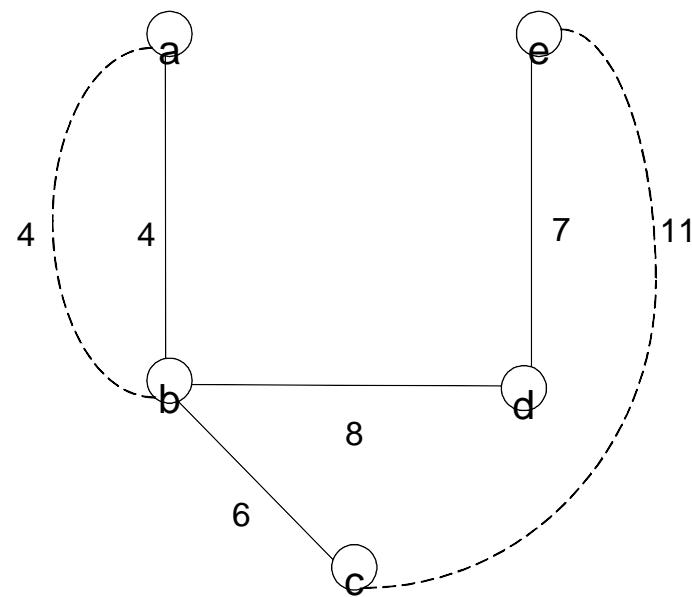
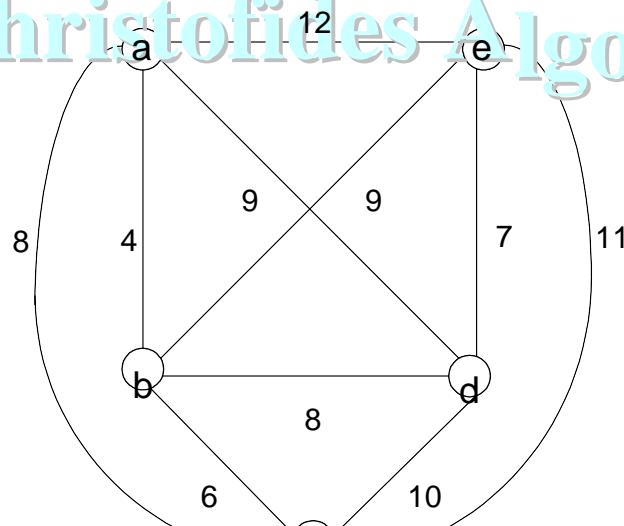
Stage 2: Add edges of a minimum-weight matching of all the odd vertices in the minimum spanning tree

Stage 3: Find an Eulerian circuit of the multigraph obtained in Stage 2

Stage 3: Create a tour from the path constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

$R_A = \infty$ for general instances, but it tends to produce better tours than the twice-around-the-minimum-tree alg.

Example: Christofides Algorithm



Euclidean Instances

Theorem If $P \neq NP$, there exists no approximation algorithm for TSP with a finite performance ratio.

Definition An instance of TSP is called *Euclidean*, if its distances satisfy two conditions:

1. *symmetry* $d[i, j] = d[j, i]$ for any pair of cities i and j
2. *triangle inequality* $d[i, j] \leq d[i, k] + d[k, j]$ for any cities i, j, k

For Euclidean instances:

approx. tour length / optimal tour length $\leq 0.5(\lceil \log_2 n \rceil + 1)$

for nearest neighbor and multifragment heuristic;

approx. tour length / optimal tour length ≤ 2

for twice-around-the-tree;

approx. tour length / optimal tour length ≤ 1.5

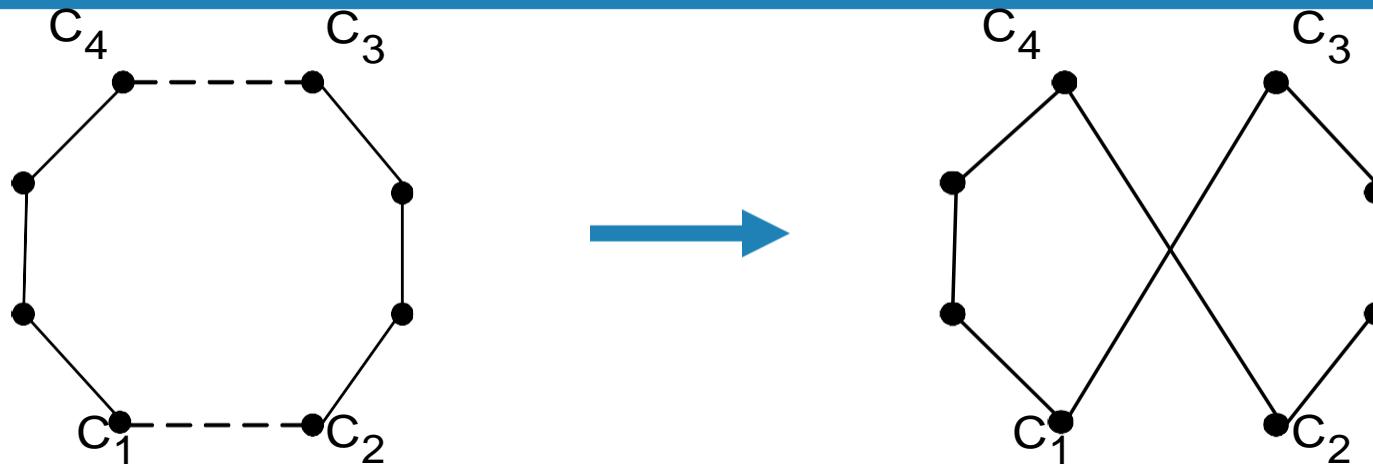
for Christofides



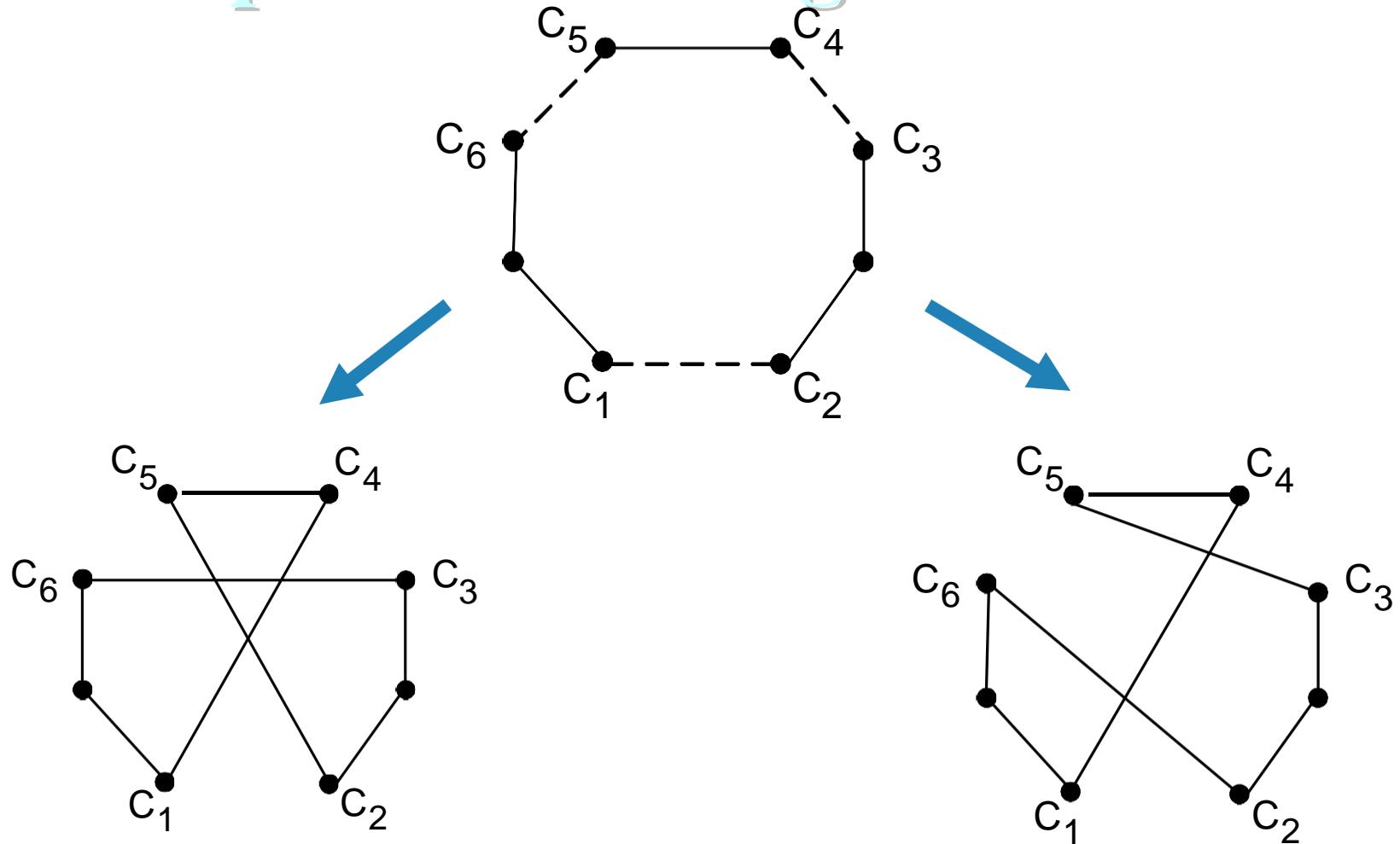
Local Search Heuristics for TSP

Start with some initial tour (e.g., nearest neighbor). On each iteration, explore the current tour's neighborhood by exchanging a few edges in it. If the new tour is shorter, make it the current tour; otherwise consider another edge change. If no change yields a shorter tour, the current tour is returned as the output.

Example of a 2-change



Example of a 3-change



Empirical Data for Euclidean Instances

TABLE 12.1 Average tour quality and running times for various heuristics on the 10,000-city random uniform Euclidean instances [Joh02]

Heuristic	% excess over the Held-Karp bound	Running time (seconds)
nearest neighbor	24.79	0.28
multifragment	16.42	0.20
Christofides	9.81	1.04
2-opt	4.70	1.41
3-opt	2.88	1.50
Lin-Kernighan	2.00	2.06

Greedy Algorithm for Knapsack problems

Step 1: Order the items in decreasing order of relative values:

$$v_1/w_1 \geq \dots \geq v_n/w_n$$

Step 2: Select the items in this order skipping those that don't fit into the knapsack

Example: The knapsack's capacity is 16

item	weight	value	v/w
1	2	\$40	20
2	5	\$30	6
3	10	\$50	5
4	5	\$10	2

Accuracy

- β R_A is unbounded (e.g., n = 2, C = m, w₁=1, v₁=2, w₂=m, v₂=m)
- β yields exact solutions for the continuous version

Approximation Scheme for Knapsack problems

Step 1: Order the items in decreasing order of relative values:

$$v_1/w_1 \geq \dots \geq v_n/w_n$$

Step 2: For a given integer parameter k , $0 \leq k \leq n$, generate all subsets of k items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios

Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the algorithm's output

- Accuracy: $f(s^*) / f(s_a) \leq 1 + 1/k$ for any instance of size n
- Time efficiency: $O(kn^{k+1})$
- There are *fully polynomial schemes*: algorithms with polynomial running time as functions of both n and k

Bin Packing Problem: First-Fit algorithm

First-Fit (FF) Algorithm: Consider the items in the order given and place each item in the first available bin with enough room for it; if there are no such bins, start a new one

Example: $n = 4$, $s_1 = 0.4$, $s_2 = 0.2$, $s_3 = 0.6$, $s_4 = 0.7$

Accuracy

- β Number of extra bins never exceeds optimal by more than 70% (i.e., $R_A \leq 1.7$)
- β Empirical average-case behavior is much better. (In one experiment with 128,000 bins, the relative error was found to be no more than 2%.)

Bin Packing: First-Fit Decreasing algorithm



First-Fit Decreasing (FFD) Algorithm: Sort the items in decreasing order (i.e., from the largest to the smallest). Then proceed as above by placing an item in the first bin in which it fits and starting a new bin if there are no such bins

Example: $n = 4$, $s_1 = 0.4$, $s_2 = 0.2$, $s_3 = 0.6$, $s_4 = 0.7$

Accuracy

- β Number of extra bins never exceeds optimal by more than 50% (i.e., $R_A \leq 1.5$)
- β Empirical average-case behavior is much better, too

Numerical Algorithms



Numerical algorithms concern with solving mathematical problems such as

- β evaluating functions (e.g., \sqrt{x} , e^x , $\ln x$, $\sin x$)
- β solving nonlinear equations
- β finding extrema of functions
- β computing definite integrals

Most such problems are of “continuous” nature and can be solved only approximately

Principal Accuracy Metrics



β Absolute error of approximation (of α^* by α)

$$|\alpha - \alpha^*|$$

β Relative error of approximation (of α^* by α)

$$|\alpha - \alpha^*| / |\alpha^*|$$

- undefined for $\alpha^* = 0$
- often quoted in %



Two Types of Errors

β truncation errors

- Taylor's polynomial approximation

$$e^x \approx 1 + x + x^2/2! + \dots + x^n/n!$$

absolute error $\leq M |x|^{n+1}/(n+1)!$ where $M = \max e^t$ for $0 \leq t \leq x$

- composite trapezoidal rule

$$\int_a^b f(x)dx \approx (h/2) [f(a) + 2\sum_{1 \leq i \leq n-1} f(x_i) + f(b)], \quad h = (b - a)/n$$

absolute error $\leq (b-a)h^2 M_2 / 12$ where $M_2 = \max |f''(x)|$ for $a \leq x \leq b$

β round-off errors

Solving Quadratic Equation



Quadratic equation $ax^2 + bx + c = 0$ ($a \neq 0$)

$$x_{1,2} = (-b \pm \sqrt{D})/2a \quad \text{where } D = b^2 - 4ac$$

Problems:

β computing square root

use Newton's method: $x_{n+1} = 0.5(x_n + D/x_n)$

β subtractive cancellation

use alternative formulas (see p. 411)

use double precision for $D = b^2 - 4ac$

β other problems (overflow, etc.)

Notes on Solving Nonlinear question

- β There exist no formulas with arithmetic ops. and root extractions for roots of polynomials

$$a_n x^n + a_{n-1} x^{n-1} \dots + a_0 = 0 \text{ of degree } n \geq 5$$

- β Although there exist special methods for approximating roots of polynomials, one can also use general methods for

$$f(x) = 0$$

- β Nonlinear equation $f(x) = 0$ can have one, many, infinitely many, and no roots at all

- β Useful:

- sketch graph of $f(x)$
- separate roots



Three Classic Methods

Three classic methods for solving nonlinear equation

$$f(x) = 0$$

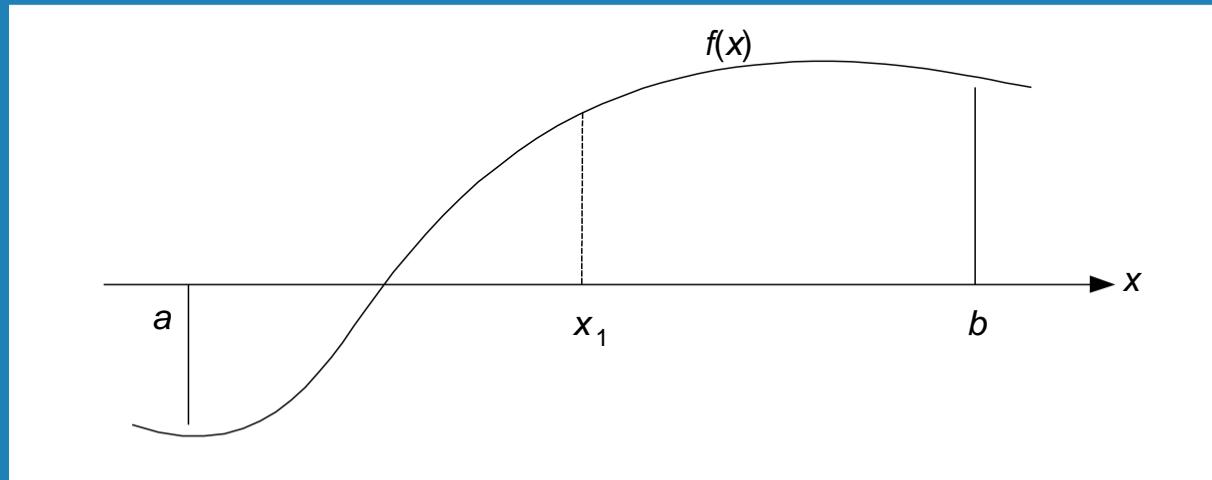
in one unknown:

- β **bisection method**
- β **method of false position (regula falsi)**
- β **Newton's method**

Bisection Method

Based on

- β Theorem: If $f(x)$ is continuous on $a \leq x \leq b$ and $f(a)$ and $f(b)$ have opposite signs, then $f(x) = 0$ has a root on $a < x < b$
- β binary search idea



Approximations x_n are middle points of shrinking segments

- β $|x_n - x^*| \leq (b - a)/2^n$
- β x_n always converges to root x^* but slower compared to others

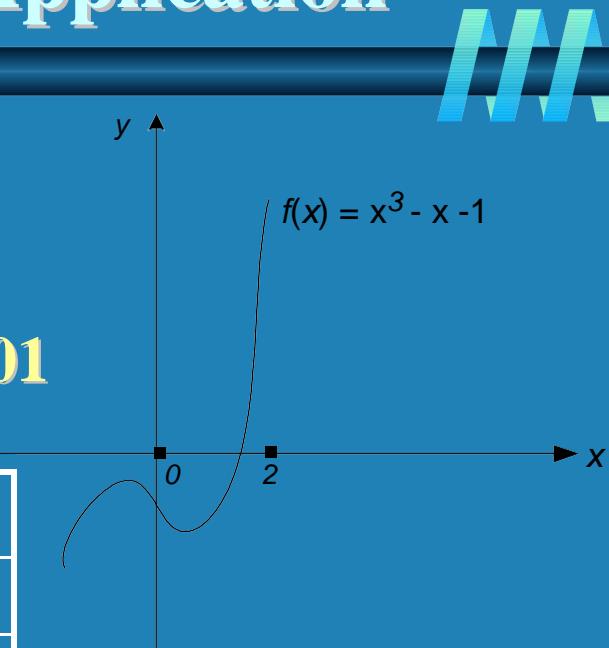
Example of Bisection Method Application

Find the root of

$$x^3 - x - 1 = 0$$

with the absolute error not larger than 0.01

n	a_n	b_n	x_n	$f(x_n)$
1	0.0-	2.0+	1.0	-1.0
2	1.0-	2.0+	1.5	0.875
3	1.0-	1.5+	1.25	-0.296875
4				
5				
6				
7				
8	1.3125-	1.328125+	1.3203125	-0.018711

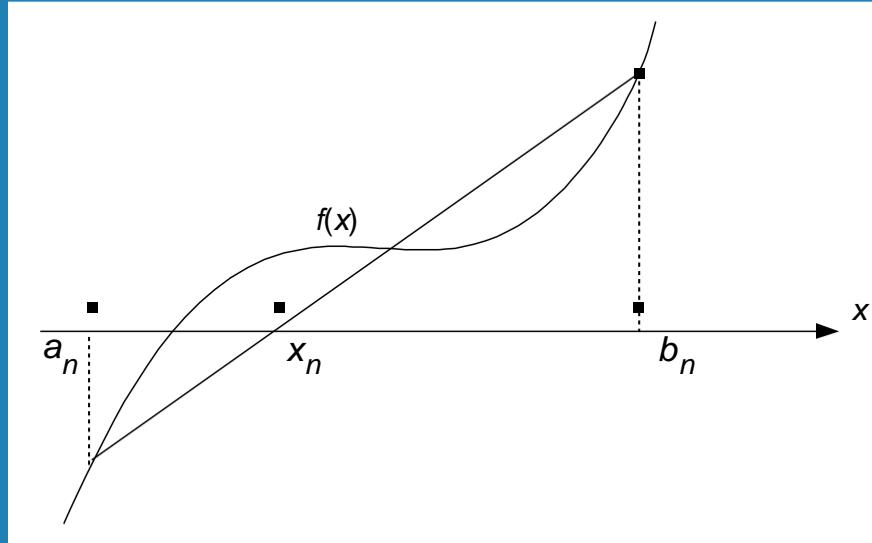


$$x \approx 1.3203125$$



Method of False Position

Similar to bisection method but uses x -intercept of line through $(a, f(a))$ and $(b, f(b))$ instead of middle point of $[a, b]$



Approximations x_n are computed by the formula

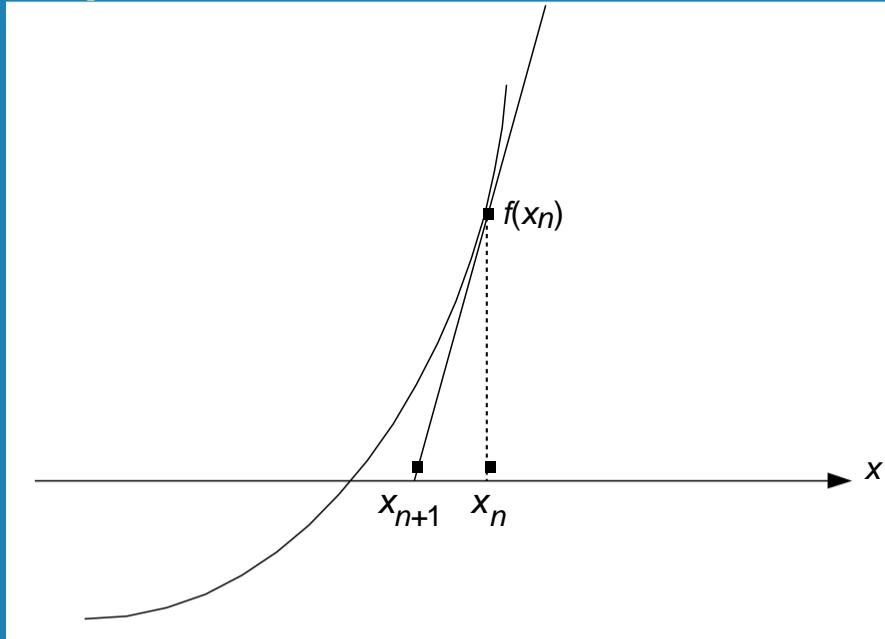
$$x_n = [a_n f(b_n) - b_n f(a_n)] / [f(b_n) - f(a_n)]$$

- β Normally x_n converges faster than bisection method sequence but slower than Newton's method sequence

Newton's Method



Very fast method in which x_n 's are x -intercepts of tangent lines to the graph of $f(x)$



Approximations x_n are computed by the formula

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$



Notes on Newton's Method

β Normally, approximations x_n converge to root very fast but can diverge with a bad choice of initial approximation x_0

β Yields a very fast method for computing square roots

$$x_{n+1} = 0.5(x_n + D/x_n)$$

β Can be generalized to much more general equations