

Introduction to Algorithm Design: L1

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Importance of problem solving using algorithms

- What is an Algorithm?
- How should we look at Solving Problems through Algorithms?

2 Characteristic features of an algorithm

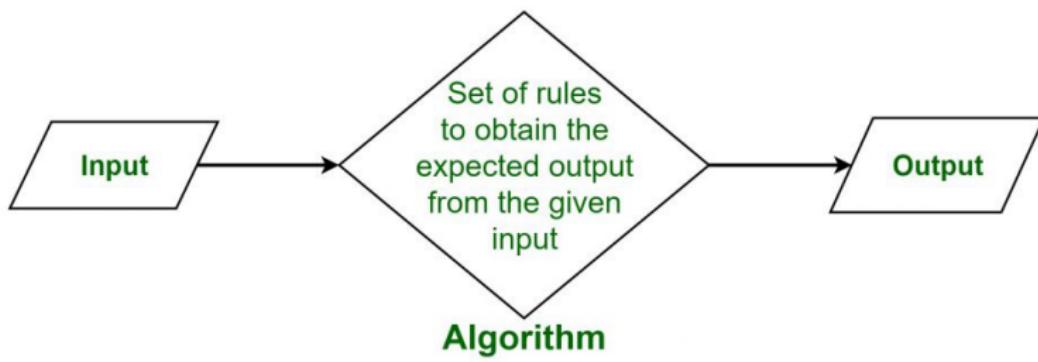
- Input
- Output
- Finiteness
- Definiteness
- Effectiveness
- Correctness
- Platform-Independence

3 Advantages and Disadvantages of Algorithms

4 How are Algorithms Written

What is an Algorithm?

What is Algorithm?



What is an Algorithm?

Definition

Informally, an algorithm is **any well-defined computational procedure** that takes **some value, or set of values, as input** and produces **some value, or set of values, as output**. An algorithm is thus a sequence of computational steps that transform the input into the output.

Formal Example

To sort a sequence of numbers into nondecreasing order. Here is how we formally define the sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle \hat{a}_1, \hat{a}_2, \dots, \hat{a}_n \rangle$ of the input sequence such that $\hat{a}_1 \leq \hat{a}_2 \leq \dots \leq \hat{a}_n$

Algorithm: A Specific Procedure

Here's what baking a cake might look like, written out as a list of instructions, just like an algorithm:

- Preheat the oven
- Gather the ingredients
- Measure out the ingredients
- Mix together the ingredients to make the batter
- Grease a pan
- Pour the batter into the pan
- Put the pan in the oven
- Set a timer
- When the timer goes off, take the pan out of the oven
- Enjoy!

How should we look at Solving Problems through Algorithms?

The deeper issue is that **the subject of algorithms is a powerful lens through which to view the field of computer science in general.** Algorithmic problems form the heart of computer science, but they rarely arrive as cleanly packaged, mathematically precise questions. Rather, they tend to come bundled together with lots of messy, application-specific detail, some of it essential, some of it extraneous. As a result, the algorithmic enterprise consists of two fundamental components: **(i) the task of getting to the mathematically clean core of a problem, and (ii) the task of identifying the appropriate algorithm design techniques, based on the structure of the problem.**

Some Important Problems Solved by Algorithms

- ① The Human Genome Project
- ② The Internet enables people all around the world to quickly access and retrieve large amounts of information with the aid of clever algorithms.
- ③ Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements.

Input

The input is the data to be transformed during the computation to produce the output. **An algorithm should have 0 or more well-defined inputs.** Input precision requires that you know what kind of data, how much and what form the data should be

Output

- The output is the data resulting from the computation (your intended result)
- An algorithm should have 1 or more well-defined outputs, and should match the desired output
- Output precision also requires that you know what kind of data, how much and what form the output should be in.

Finiteness

- The algorithm must stop, eventually.
- Stopping may mean that you get the expected output OR you get a response that no solution is possible. Algorithms must terminate after a finite number of steps.
- There is no point in developing an algorithm which is infinite as it will be useless for us.

Definiteness

- Definiteness means specifying the sequence of operations for turning input into output.
- Each step must be clear, well-defined and precise. There should be no any ambiguity. Details of each step must be also be spelled out (including how to handle errors)

Effectiveness

- All steps required to get to output must be feasible with the available resources. It should not contain any unnecessary and redundant steps.

Correctness

- We need tools to distinguish correct algorithms from incorrect ones, the primary one of which is called a proof.

Steps for an Inductive Proof

- First, you show that the property is true for some simple case: an empty list or a list of length 1, an empty set, a single point. Usually this demonstration is very simple; often it's obviously true that the property is true. This is called the basis case.
- Next, you assume the property is true for size N and show that it's true for some larger size such as N+1. This is called the inductive step, and is usually the more difficult one.

Platform-Independence

- An algorithm should have step-by-step directions, which should be independent of any programming code. It should be such that it could be run on any of the programming languages.

Advantages and Disadvantages of Algorithms

Advantages of Algorithms:

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program.

Disdvantages of Algorithms:

1. Alogorithms is Time consuming.
2. Difficult to show Branching and Looping in Algorithms.
3. Big tasks are difficult to put in Algorithms.

Ways of Representing Algorithms

- **ENGLISH-LIKE:** Written in simple english language statements
- **PSEUDOCODE:** The pseudocode has an advantage of being easily converted into any programming language. This way of writing algorithm is most acceptable and most widely used. In order to write a pseudocode, one must be familiar with the conventions of writing it.
- **FLOWCHART:** Flowcharts pictorially depict a process. They are easy to understand and are commonly used in the case of simple problems.

Algorithm to Find the Largest of 3 Numbers I

Finding the Maximum of Three Numbers.

Input: Three numbers a, b, and c

Output: x, the largest of a, b, and c

```
procedure max(a, b, c)
```

```
    x := a
```

```
    if b > x then
```

```
        x := b
```

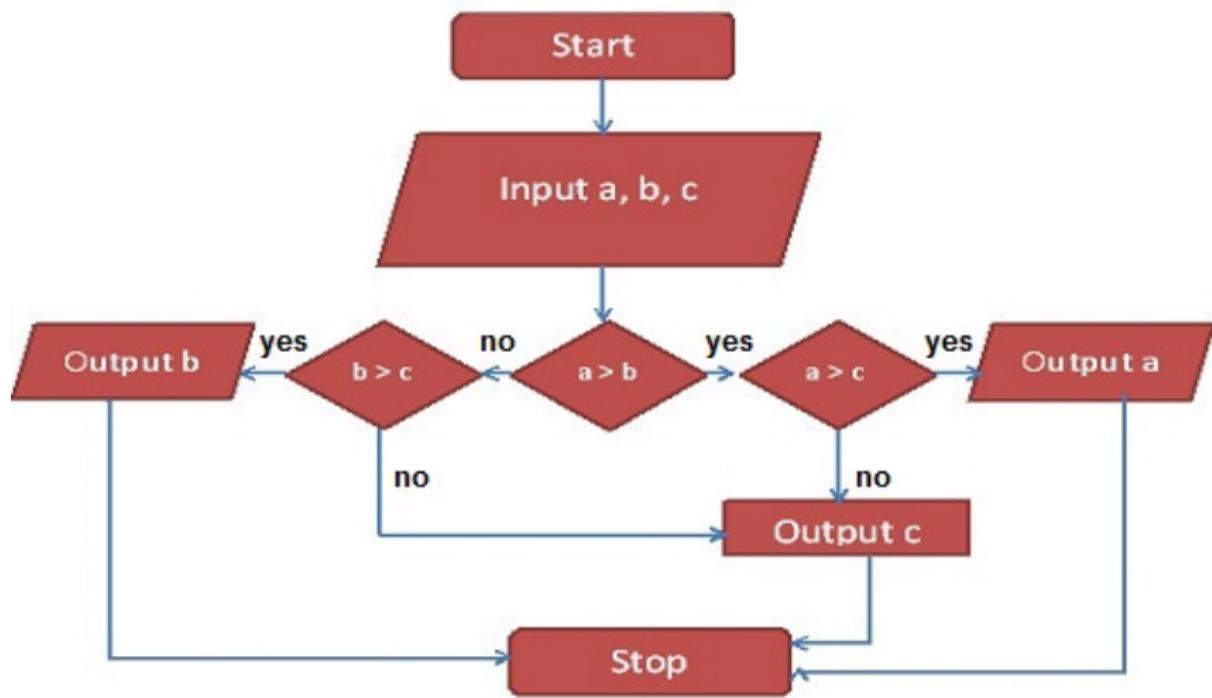
```
    if c > x then
```

```
        x := c
```

```
    return(x)
```

```
end max
```

Algorithm to Find the Largest of 3 Numbers II



Algorithm to Find the Largest of 3 Numbers III

Input

The algorithm definitely has 3 inputs upon which it can process.

Will the Algorithm Terminate

Yes. The list L is of finite length, so after looking at every element of the list the algorithm will stop.

Algorithm to Find the Largest of 3 Numbers IV

Does it produce the correct result?

Let us look at a proof or sketch to understand correctness of the algorithm

- Consider a list of length 1. In this case the largest number is also the only number on the list. `find_max()` returns this number, so it's correct for lists of length 1.
- Assume its correct for length N hence using inductive hypothesis we can prove that the algorithm is correct.

Self Assessment

Take a Self Assesment and write Algorithms for the following. Make sure you verify the characteristics of the algorithm.

- Testing Whether a Positive Integer is Prime.
- Finding the Largest Element in a Finite Sequence.
- Find the factorial of a number

Thank You

Expressing algorithms (pseudocode); Basic aspects of algorithms (design and analysis): L2

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Solution to the Sorting Problem

- Insertion Sort

2 Conventions Used for Pseudocode

3 Analysis of algorithms

Solution to the Sorting Problem

The Problem of Sorting

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- Our first algorithm, **insertion sort**, solves the sorting problem introduced in L_1 .
- The numbers that we wish to sort are also known as the **keys**. Although conceptually we are sorting a sequence, the input comes to us in the form of an **array with n elements**.
- What separates pseudocode from “real” code is that in pseudocode, **we employ whatever expressive method is most clear** and concise to specify a given algorithm.
Sometimes, the clearest method is English, so **do not be surprised if you come across an English phrase or sentence embedded within a section of pseudocode.**

Understanding Insertion Sort I

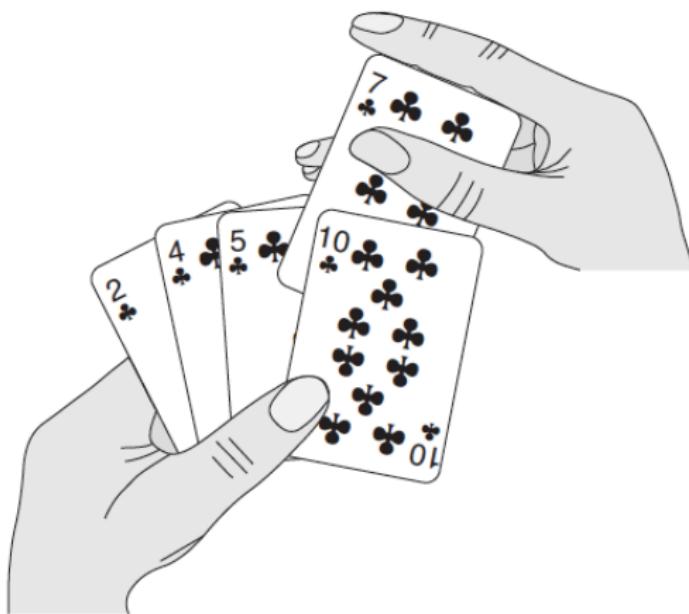


Figure: Sorting a Pack of Cards using Insertion Sort

Pseudocode for Insertion Sort

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

Figure: Pseudocode for Insertion Sort

Dry Run

Example for Insertion Sort

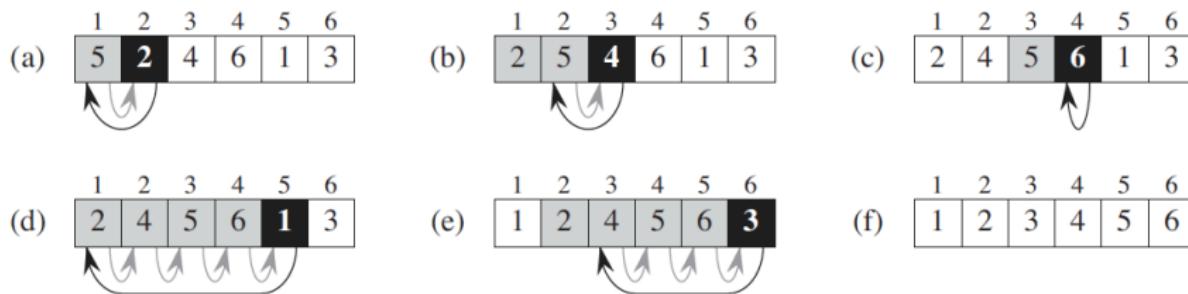


Figure: An Illustrative Example for Insertion Sort

Conventions Used for Pseudocode I

- **Indentation indicates block structure.** For example, the body of the for loop that begins on line 1 consists of lines 2–8, and the body of the while loop that begins on line 5 contains lines 6–7 but not line 8.
- The **symbol “//”** indicates that the remainder of the line **is a comment**.
- A **multiple assignment** of the form $i = j = e$ assigns to both variables i and j the value of expression e ;
- Variables (such as i , j , and key) are local to the given procedure.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the i th element of the array A .

Conventions Used for Pseudocode II

- We typically organize compound data into objects, which are composed of attributes. **the object name, followed by a dot, followed by the attribute name.** To specify the number of elements in an array A, we write A.length.
- We **pass parameters to a procedure by value:** the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is not seen by the calling procedure.'
- A **return statement** immediately transfers control back to the point of call in the calling procedure.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called.

Parameters for Analysis

- Analyzing an algorithm means **predicting the resources that the algorithm requires.**
- Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is **computational time** that we want to measure.
- For most of this book, we shall assume a **generic oneprocessor, random-access machine (RAM) model of computation.** In the RAM model, instructions are executed one after another, with no concurrent operations.
- When working with inputs of size n , we typically assume that integers are represented by $c \lg n$ bits for some constant c . We require c so that each word can hold the value of n , enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily.

Analysis of Insertion Sort

- **The time taken by the INSERTION-SORT procedure depends on the input:** sorting a thousand numbers takes longer than sorting three numbers.
- Moreover, INSERTIONSORT can take different amounts of time to sort two input sequences of the same size depending on **how nearly sorted they already are**. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input.

Running Time and size of inputs

Input size

- The best notion for input size depends on the problem being studied.
- for example, the array size n for sorting problem. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation

Running Time

- The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed. It is convenient to define the notion of step so that it is as machine-independent as possible.

Running Time of Insertion Sort I

- For each $j = 2, 3, \dots, n$, where $n = A.length$, we let t_j denote the number of times the while loop test in line 5 is executed for that value of j .
- When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.
- We assume that comments are not executable statements, and so they take no time.
- The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.

Running Time of Insertion Sort II

INSERTION-SORT(A)	<i>cost</i>	<i>times</i>
1 for $j = 2$ to $A.length$	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\&\quad + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1).\end{aligned}$$

This is the total running time of Insertion sort

Thank You

Expressing algorithms (pseudocode); Basic aspects of algorithms (design and analysis): L3

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

- 1 Running time of Insertion Sort
- 2 Best Case and Worst Case Analysis of Insertion Sort
- 3 Order of Growth
- 4 Designing Algorithms

Running Time and size of inputs

Input size

- The best notion for input size depends on the problem being studied.
- for example, the array size n for sorting problem. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation

Running Time

- The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed. It is convenient to define the notion of step so that it is as machine-independent as possible.

Running Time of Insertion Sort I

- For each $j = 2, 3, \dots, n$, where $n = A.length$, we let t_j denote the number of times the while loop test in line 5 is executed for that value of j .
- When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.
- We assume that comments are not executable statements, and so they take no time.
- The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.

Running Time of Insertion Sort II

INSERTION-SORT(A)

	<i>cost</i>	<i>times</i>
1 for $j = 2$ to $A.length$	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1). \end{aligned}$$

This is the total running time of Insertion sort

Best Case

Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given. **For example, in INSERTION-SORT, the best case occurs if the array is already sorted.**

For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \dots, n$ we then find that $A[i] \leq \text{key}$ in line 5 when i has its initial value of $j-1$. Thus $t_j = 1$ for $j = 2, 3, \dots, n$, and the best-case running time is:

$$\begin{aligned}T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).\end{aligned}$$

Running time of Insertion Sort
○○○

Best Case and Worst Case Analysis of Insertion Sort
○●○○○

Order of Growth
○○

Designing Algorithms
○○○○○

Example of Best Case

Worst Case

If the array is in reverse sorted order that is, in decreasing order the worst case results. We must compare each element $A[j]$ with each element in the entire sorted $A[i\dots j-1]$, and so $t_j = j$ for $j = 2, 3, \dots, n$.

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$

and

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

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Example of Worst Case

Average Case

How long does it take to determine where in subarray $A[i \dots j-1]$ to insert element $A[j]$? On average, half the elements in $A[i \dots j-1]$ are less than $A[j]$, and half the elements are greater. Let's assume that $t_j = (j-1)/2$ to calculate the average case

Therefore, $T(n) = C_1 * n + (C_2 + C_3) * (n - 1) + C_4/2 * (n - 1)(n)/2 + (C_5 + C_6)/2 * ((n - 1)(n)/2 - 1) + C_8 * (n - 1)$ which when further simplified has dominating factor of n^2 and gives $T(n) = C * (n^2)$ or $O(n^2)$

Order of Growth

Why Order of Growth

We shall now make one more simplifying abstraction: **it is the rate of growth, or order of growth**, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), **since the lower-order terms are relatively insignificant for large values of n**. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

Order of Growth for Insertion Sort

For insertion sort, **when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of n^2 from the leading term**. We write that insertion sort has a worst-case **running time of $\theta(n^2)$** (pronounced “theta of n-squared”).

Significance of Worst Case Analysis

- The worst-case running time of an algorithm **gives us an upper bound on the running time for any input**. Knowing it provides a guarantee that **the algorithm will never take any longer**. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.
- The “average case” is often roughly as bad as the worst case.

Greedy Approach

- A Greedy algorithm is an algorithmic paradigm that **builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.** So the problems where choosing locally optimal also leads to a global solution are best fit for Greedy.
- In Greedy Method, sometimes there is **no such guarantee of getting Optimal Solution.**
- Fractional knapsack

Divide and Conquer

- It deals (involves) three steps at each level of recursion:
 - **Divide** the problem into a number of subproblems.
 - **Conquer** the subproblems by solving them recursively.
 - **Combine** the solution to the subproblems into the solution for original subproblems.
- It is **Recursive**. To solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.
- When an algorithm contains a recursive call to itself, **we can often describe its running time by a recurrence equation or recurrence**, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.

Dynamic Programming

- In Dynamic Programming we **make decision at each step considering current problem and solution to previously solved sub problem** to calculate optimal solution .
- It is guaranteed that **Dynamic Programming will generate an optimal solution.**
- Dynamic Programming is **generally slower.**
- 0/1 knapsack problem

Problem Solving

- Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.
- Dry Run and show the steps in sorting an array A=6,2,8,1,19 in non-decreasing order using Insertion Sort
- Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n+1)$ -element array C. State the problem formally and write pseudocode for adding the two integers.

Running time of Insertion Sort
○○○

Best Case and Worst Case Analysis of Insertion Sort
○○○○○

Order of Growth
○○

Designing Algorithms
○○○●

Thank You

Algorithm Correctness: **using counter examples**, loop invariants, induction method: L4

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Algorithm Correctness

- Algorithm Correctness Using Counter Examples
 - Robot Tour Optimization Problem
 - Selecting the Right Job
- Algorithm Correctness Using Loop Invariants
- Algorithm Correctness Using Induction Method

Why Correctness should be Verified

- It is seldom obvious whether a given algorithm correctly solves a given problem.
- Correct algorithms usually come with a proof of correctness, **which is an explanation of why we know that the algorithm must take every instance of the problem to the desired result.**
- The best way to prove that an algorithm is incorrect is to **produce an instance in which it yields an incorrect answer.** Such instances are called counter-examples. No rational person will ever leap to the defense of an algorithm after a counterexample has been identified.

Robot Tour Optimization Problem I

Let's consider a problem that arises often in manufacturing, transportation, and testing applications.

The Problem Statement

Suppose we are **given a robot arm** equipped with a tool, say a soldering iron. In manufacturing circuit boards, **all the chips and other components must be fastened onto the substrate**.

More specifically, each chip has a set of contact points (or wires) that must be soldered to the board.

Robot Tour Optimization Problem II

The Solution Sketch

To program the robot arm for this job, we must **first construct an ordering of the contact points so the robot visits (and solders)** the first contact point, then the second point, third, and so forth until the job is done. thus turning the tool-path into a closed tour, or cycle.

The Intermediate Villains: Constraints

Robots are expensive devices, so we want the tour that minimizes the time it takes to assemble the circuit board. A reasonable assumption is that the robot arm moves with fixed speed, **so the time to travel between two points is proportional to their distance.**

Robot Tour Optimization Problem III

The Formal Problem

Problem: Robot Tour Optimization

Input: A set S of n points in the plane.

Output: What is the shortest cycle tour that visits each point in the set S ?

Robot Tour Optimization Problem IV

Nearest Neighbor Heuristic

Starting from some point p_0 , we walk first to its nearest neighbor p_1 . From p_1 , we walk to its nearest unvisited neighbor, thus excluding only p_0 as a candidate. We now repeat this process until we run out of unvisited points, after which we return to p_0 to close off the tour.

NearestNeighbor(P)

Pick and visit an initial point p_0 from P

$p = p_0$

$i = 0$

While there are still unvisited points

$i = i + 1$

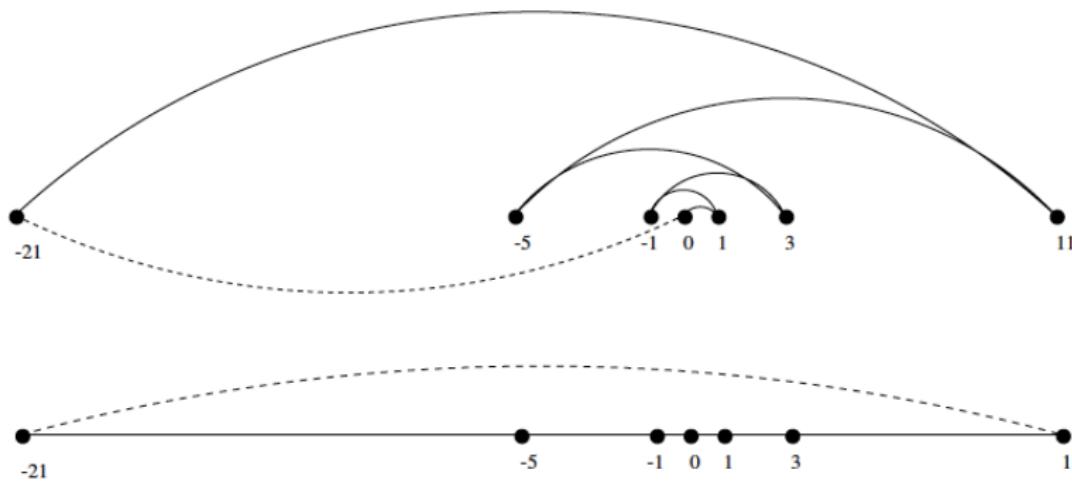
Select p_i to be the closest unvisited point to p_{i-1}

Visit p_i

Return to p_0 from p_{n-1}

simple, easy to implement, efficient, **but it is completely wrong**

Counter Example to Prove that Nearest Neighbor is wrong



The numbers describe the distance that each point lies to the left or right of the point labeled '0'.

Closest Pair Algorithm

A different idea might be to repeatedly connect the closest pair of endpoints whose connection will not create a problem

ClosestPair(P)

Let n be the number of points in set P .

For $i = 1$ to $n - 1$ do

$d = \infty$

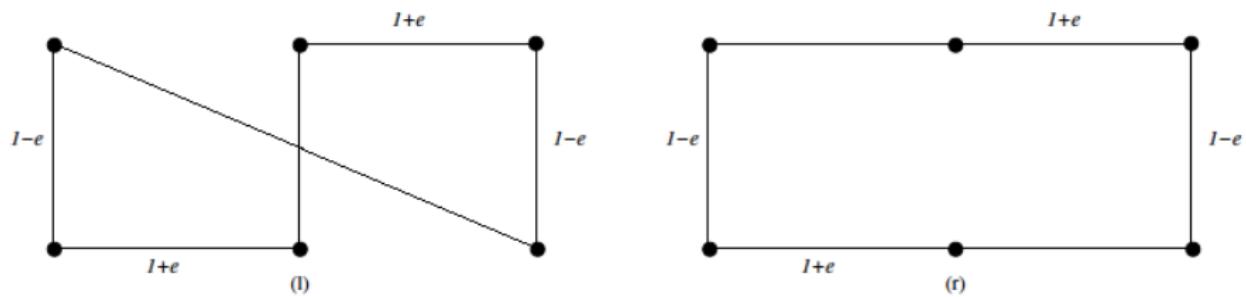
For each pair of endpoints (s, t) from distinct vertex chains

if $dist(s, t) \leq d$ then $s_m = s$, $t_m = t$, and $d = dist(s, t)$

Connect (s_m, t_m) by an edge

Connect the two endpoints by an edge

Counter Example to Prove that Closest Pair is wrong



Rows are little closer than points within a row.

Optimal TSP

OptimalTSP(P)

$$d = \infty$$

For each of the $n!$ permutations P_i of point set P

If ($\text{cost}(P_i) \leq d$) then $d = \text{cost}(P_i)$ and $P_{min} = P_i$

Return P_{min}

Selecting the Right Job I

Imagine you are a highly-indemand actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming. To take the job, you must commit to being available throughout this entire period. Thus you cannot simultaneously accept two jobs whose intervals overlap.

For an artist such as yourself, the criteria for job acceptance is clear: you want to make as much money as possible. Because each of these films pays the same fee per film, this implies you seek the largest possible set of jobs (intervals) such that no two of them conflict with each other.

Selecting the Right Job II

The Formal Problem

Problem: Movie Scheduling Problem

Input: A set I of n intervals on the line.

Output: What is the largest subset of mutually non-overlapping intervals which can be selected from I ?

Tarjan of the Jungle

The President's Algorist

"Discrete" Mathematics

The Four Volume Problem

Process Terminated

Programming Challenges

Calculated Bets

Earliest Job First and Shortest Job First

1. EarliestJobFirst(I) Accept the earliest starting job j from I which does not overlap any previously accepted job, and repeat until no more such jobs remain.
2. Shortest Job First

ShortestJobFirst(I)

 While ($I \neq \emptyset$) do

 Accept the shortest possible job j from I .

 Delete j , and any interval which intersects j from I .

Optimal Scheduling Algorithm

OptimalScheduling(I)

 While ($I \neq \emptyset$) do

 Accept the job j from I with the earliest completion date.

 Delete j , and any interval which intersects j from I .

Reasoning about Correctness

- Hopefully, the previous examples have opened your eyes to the subtleties of algorithm correctness.
- We need tools to distinguish correct algorithms from incorrect ones, the primary one of which is called a proof.
- A proper mathematical proof consists of several parts. First, there is a clear, precise statement of what you are trying to prove. Second, there is a set of assumptions of things which are taken to be true and hence used as part of the proof. Third, there is a chain of reasoning which takes you from these assumptions to the statement you are trying to prove.

Properties of Good Counter Examples

- Verifiability: To demonstrate that a particular instance is a counter-example to a particular algorithm, you must be able to:
 - (1) calculate what answer your algorithm will give in this instance, and
 - (2) display a better answer so as to prove the algorithm didn't find it.
- Simplicity: Good counter-examples have all unnecessary details boiled away. They make clear exactly why the proposed algorithm fails.

Hunting for a good Counter-Example

- Think small
- Think exhaustively
- Hunt for the weakness
- Go for a tie
- Seek Extremes

Thank You

Algorithm Correctness: using counter examples, **loop invariants**, induction method: L5

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

- Algorithm Correctness Using Loop Invariants

Loop invariants and the correctness of insertion sort I

At the start of each iteration of the for loop of lines 1-8, the subarray $A[1, \dots, j - 1]$ consists of the elements originally in $A[1, \dots, j - 1]$ but in sorted order.— **Loop Invariant.**

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, **the invariant gives us a useful property that helps show that the algorithm is correct.**

Loop invariants and the correctness of insertion sort II

Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

Initialization:

We start by showing that the loop invariant holds before the first loop iteration, when $j = 2^1$. The subarray $A[1, \dots, j - 1]$ therefore, consists of just the single element $A[1]$, which is in fact the original element in A . Moreover, this subarray is sorted (trivially, of course).

Maintenance:

Informally, the body of the for loop works by moving $A[j-1], A[j-2], A[j-3] \dots$ and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8).

Termination:

Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that

$j > A.length = n$ Because each loop iteration increases j by 1, we must have $j=n+1$ at that time. Substituting $n+1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$ but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Loop Invariant for Bubble Sort

2-2 Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

`BUBBLESORT(A)`

```
1  for  $i = 1$  to  $A.length - 1$ 
2      for  $j = A.length$  downto  $i + 1$ 
3          if  $A[j] < A[j - 1]$ 
4              exchange  $A[j]$  with  $A[j - 1]$ 
```

- a. Let A' denote the output of `BUBBLESORT(A)`. To prove that `BUBBLESORT` is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n], \quad (2.3)$$

where $n = A.length$. In order to show that `BUBBLESORT` actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

- b. State precisely a loop invariant for the `for` loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.
- c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the `for` loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

Bubble Sort

Solution

Loop invariant: At the start of each iteration of the for loop of lines 1 – 4, the subarray $A[1..i-1]$ consists of the $i-1$ smallest elements in $A[1..n]$ in sorted order. $A[i..n]$ consists of the $n-i+1$ remaining elements in $A[1..n]$.

Initialization: Initially the subarray $A[1..i-1]$ is empty and trivially this is the smallest element of the subarray.

Maintenance: From part (b), after the execution of the inner loop, $A[i]$ will be the smallest element of the subarray $A[i..n]$. And in the beginning of the outer loop, $A[1..i-1]$ consists of elements that are smaller than the elements of $A[i..n]$, in sorted order. So, after the execution of the outer loop, subarray $A[1..i]$ will consist of elements that are smaller than the elements of $A[i+1..n]$, in sorted order.

Termination: The loop terminates when $i=A.length$. At that point the array $A[1..n]$ will consist of all elements in sorted order.

Thank You

Algorithm Correctness: using counter examples, loop invariants, **induction method**: L6

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Algorithm Correctness Using Induction Method

Algorithm Correctness Using Induction Method

- Failure to find a counterexample to a given algorithm does not mean “it is obvious” that the algorithm is correct.
- A proof or demonstration of correctness is needed. Often mathematical induction is the method of choice.

Recursion and Induction

General and Boundary Conditions

- General condition breaks the problem into smaller and smaller pieces.
- The initial or boundary condition terminates the recursion or induction

Structure of Induction

- The initial or base case: prove that the statement holds for 0, or 1.
- The induction step, inductive step, or step case: prove that for every n , if the statement holds for n , then it holds for $n + 1$. In other words, assume that the statement holds for some arbitrary natural number n , and prove that the statement holds for $n + 1$.

A Simple Example

Use mathematical induction to show that, for all integers $n \geq 1$,

$$\sum_{i=1}^n i^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Solution

We proceed by induction on n . When $n = 1$, the left-hand side reduces to $1^2 = 1$, and the right-hand side becomes $\frac{1 \cdot 2 \cdot 3}{6} = 1$; hence, the identity holds for $n = 1$. Assume it holds when $n = k$ for some integer $k \geq 1$; that is, assume that

$$\sum_{i=1}^k i^2 = \frac{k(k+1)(2k+1)}{6}$$

for some integer $k \geq 1$. We want to show that it still holds when $n = k + 1$. In other words, we want to show that

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+2)[2(k+1)+1]}{6} = \frac{(k+1)(k+2)(2k+3)}{6}.$$

From the inductive hypothesis, we find

$$\begin{aligned}\sum_{i=1}^{k+1} i^2 &= \left(\sum_{i=1}^k i^2 \right) + (k+1)^2 \\ &= \frac{k(k+1)(2k+1)}{6} + (k+1)^2 \\ &\stackrel{?}{=} \frac{1}{6} (k+1)[k(2k+1) + 6(k+1)] \\ &= \frac{1}{6} (k+1)(2k^2 + 7k + 6) \\ &= \frac{1}{6} (k+1)(k+2)(2k+3).\end{aligned}$$

Therefore, the identity also holds when $n = k + 1$. This completes the induction.

A More Complex Example

Problem: Prove the correctness of the following recursive algorithm for incrementing natural numbers, i.e. $y \rightarrow y + 1$:

```
Increment(y)
  if  $y = 0$  then return(1) else
    if  $(y \bmod 2) = 1$  then
      return( $2 \cdot Increment(\lfloor y/2 \rfloor)$ )
    else return( $y + 1$ )
```

Solution of The Correctness of Increment

Proof of Correctness for Insertion Sort

- The basis case consists of a single element, and by definition a one-element array is completely sorted.
- In general, we can assume that the first $n - 1$ elements of array A are completely sorted after $n - 1$ iterations of insertion sort.
- To insert one last element x to A, we find where it goes, namely the unique spot between the biggest element less than or equal to x and the smallest element greater than x . This is done by moving all the greater elements back by one position, creating room for x in the desired location.

Practice Questions

Problem: Prove that $\sum_{i=1}^n i \times i! = (n + 1)! - 1$ by induction.

Solution: The inductive paradigm is straightforward. First verify the basis case (here we do $n = 1$, although $n = 0$ would be even more general):

$$\sum_{i=1}^1 i \times i! = 1 = (1 + 1)! - 1 = 2 - 1 = 1$$

Now assume the statement is true up to n . To prove the general case of $n + 1$, observe that rolling out the largest term

$$\sum_{i=1}^{n+1} i \times i! = (n + 1) \times (n + 1)! + \sum_{i=1}^n i \times i!$$

reveals the left side of our inductive assumption. Substituting the right side gives us

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

Thank You

Time and space complexity of an algorithm: L7

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Time and Space Complexity of An Algorithm

2 Asymptotic Notations

Time and Space Complexity of An Algorithm

Analyzing algorithms involves thinking about how their resource requirements—the **amount of time and space** they use—will scale with increasing input size.

As we seek to understand the general notion of computational efficiency, **we will focus primarily on efficiency in running time**: we want algorithms that run quickly. But it is important that algorithms be efficient in their use of other resources as well. In particular, **the amount of space (or memory)** used by an algorithm

Defining Efficiency of an Algorithm

What is efficient for you might not be for me !!

Proposed Definition of Efficiency (1): An algorithm is efficient if, when implemented, it runs quickly on real input instances.

What is missing in this definition:

- The first is the **omission of where, and how well**, we implement an algorithm.- sloppy coding, bad test cases...
- Some input instances can be much harder than others.
- This proposed definition above does not consider **how well, or badly, an algorithm may scale** as problem sizes grow to unexpected levels.

Understanding the Definition of Efficiency : GS Algorithm

Defining Input Size

The input has a natural “size” parameter N ; we could take this to be the total size of the representation of all preference lists, since this is what any algorithm for the problem will receive as input. N is closely related to the other natural parameter in this problem: n , the number of men and the number of women. Since there are $2n$ preference lists, each of length n , we can view $N = 2n^2$,

Worst Case Running Time and Brute-force Method

What is a Search Space

Even when the size of a Stable Matching input instance is relatively small, the search space it defines is enormous (there are $n!$ possible perfect matchings between n men and n women), and we need to find a matching that is stable.

Bruteforce Method

The natural “brute-force” algorithm for this problem would plow through all perfect matchings by enumeration, checking each to see if it is stable.

Proposed Definition of Efficiency (2): An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.

Polynomial Time as a Definition of Efficiency

We'd like a good algorithm for such a problem to have a better scaling property: when the input size increases by a constant factor—say, a factor of 2—the algorithm should only slow down by some constant factor C .

Polynomial Time

Arithmetically, we can formulate this scaling behavior as follows. Suppose an algorithm has the following property: There are absolute constants $c > 0$ and $d > 0$ so that on every input instance of size N , its running time is bounded by cN^d primitive computational steps.

Proposed Definition of Efficiency (3): An algorithm is efficient **if it has a polynomial running time**.

Is the Polynomial Time Definition Correct

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Asymptotic order of Growth

Our discussion of computational tractability has turned out to be intrinsically based on our ability to express the notion that an algorithm's worst-case running time on inputs of size n grows at a rate that is at most proportional to some function $f(n)$. The function $f(n)$ then becomes a bound on the running time of the algorithm.

3 Basic Asymptotic Notations

- Theta(θ)-Asymptotically Tight Bound
- Big-Oh(O)-Asymptotically Upper Bound
- Big-Omega(Ω)- Asymptotically Lower Bound

Asymptotic Notations

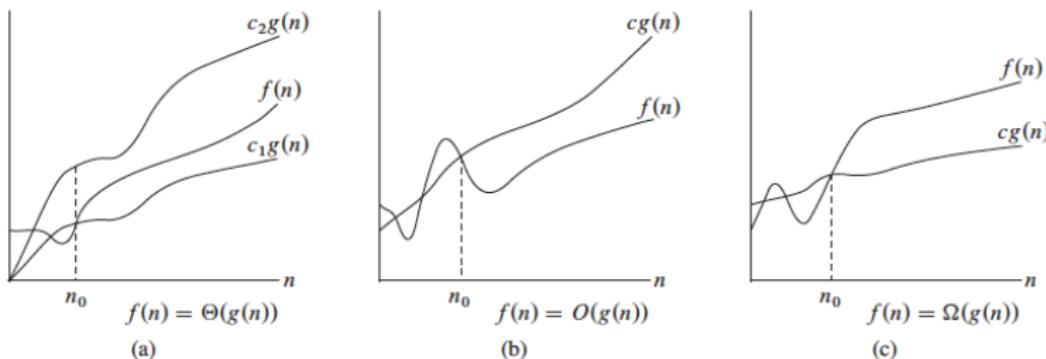


Figure 3.1 Graphic examples of the Θ , O , and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. (a) Θ -notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0 , c_1 , and c_2 such that at and to the right of n_0 , the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. (b) O -notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or below $cg(n)$. (c) Ω -notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or above $cg(n)$.

Thank You

Asymptotic notations; Summations; Logarithms: L8

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Asymptotic Notations

Asymptotic order of Growth

Our discussion of computational tractability has turned out to be intrinsically based on our ability to express the notion that an algorithm's worst-case running time on inputs of size n grows at a rate that is at most proportional to some function $f(n)$. The function $f(n)$ then becomes a bound on the running time of the algorithm.

3 Basic Asymptotic Notations

- Theta(θ)-Asymptotically Tight Bound
- Big-Oh(O)-Asymptotically Upper Bound
- Big-Omega(Ω)- Asymptotically Lower Bound

Need of Asymptotic Order of Growth

On any input of size n , the algorithm runs for at most $1.62n^2 + 3.5n + 8$ steps."

- getting such a precise bound may be an exhausting activity, and more detail than we wanted anyway.
- because our ultimate goal is to identify broad classes of algorithms that have similar behavior, we'd actually like to classify running times at a coarser level of granularity so that similarities among different algorithms, and among different problems, show up more clearly.

For all these reasons, we want to express the growth rate of running times and other functions in a way that is insensitive to constant factors and loworder terms. In other words, we'd like to be able to take a running time like the one we discussed above, $1.62n^2 + 3.5n + 8$, and say that it grows like n^2 , up to constant factors. We now discuss a precise way to do this.

Asymptotic Notations

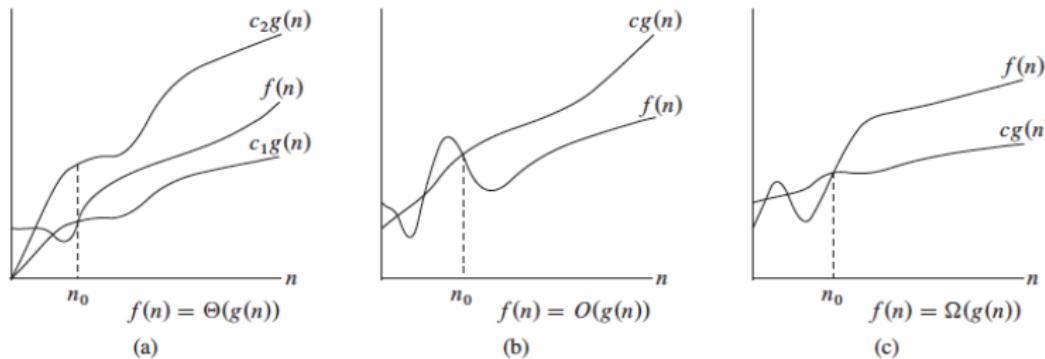


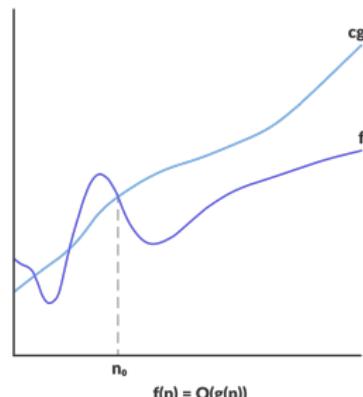
Figure 3.1 Graphic examples of the Θ , O , and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. (a) Θ -notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0 , c_1 , and c_2 such that at and to the right of n_0 , the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. (b) O -notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or below $cg(n)$. (c) Ω -notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or above $cg(n)$.

Big-O Notation

When we have only an asymptotic upper bound, we use O-notation.
For a given function $g(n)$, we denote by $O(g(n))$ (pronounced “big-oh of g of n ” or sometimes just “oh of g of n ”) the set of functions

Definition

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



Properties of Big-O Notation

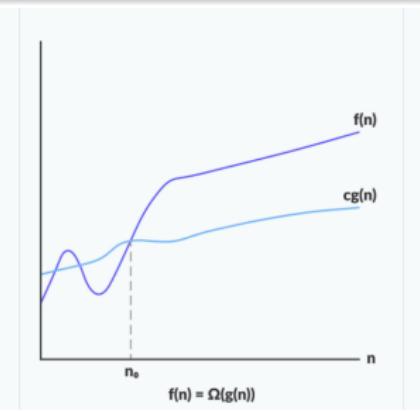
- $f(n) \in O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is.
- Thus, our proof that any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\theta(n^2)$ also shows that any such quadratic function is in $O(n^2)$.
- Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm immediately yields an $O(n^2)$ upper bound on the worst-case running time:
- The $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input. The $\theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\theta(n^2)$ bound on the running time of insertion sort on every input.

Big- Ω Notation

Just as O -notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. For a given function $g(n)$, we denote by $\Omega g(n)$ (pronounced “big-omega of g of n ” or sometimes just “omega of g of n ”) the set of functions

Definition

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



Properties of Big- Ω Notation

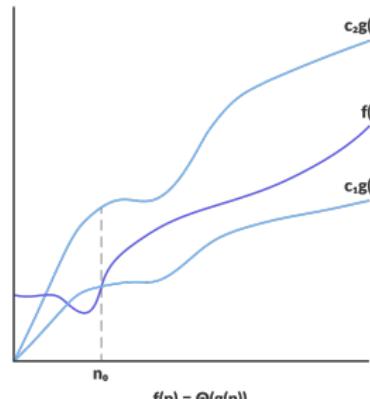
- When we say that the running time (no modifier) of an algorithm is $\Omega g(n)$, we mean that no matter what particular input of size n is chosen for each value of n , the running time on that input is at least a constant times $g(n)$, for sufficiently large n .
- Equivalently, we are giving a lower bound on the best-case running time of an algorithm. For example, the best-case running time of insertion sort is $\Omega(n)$, which implies that the running time of insertion sort is $\Omega(n)$.
- The running time of insertion sort therefore belongs to both $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of n and a quadratic function of n .

θ -notation

We found that the worst-case running time of insertion sort is $T(n) = \theta(n^2)$. Let us define what this notation means.

Definition

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$.¹



Properties of θ Notation

- A function $f(n)$ belongs to the set $\theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be “sandwiched” between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n .
- The definition of $\theta(g(n))$ requires that every member $f(n) \in \theta(g(n))$ be asymptotically non-negative, that is, that $f(n)$ be non-negative whenever n is sufficiently large.
- Since any constant is a degree-0 polynomial, we can express any constant function as $\theta(0)$, or $\theta(1)$.

o -notation and ω -notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.

totically tight. We formally define $o(g(n))$ ("little-oh of g of n ") as the set

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}.$$

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

By analogy, ω -notation is similar to to Ω -notation. We use ω -notation to denote a lower bound that is not asymptotically tight. One way to define it is by

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n ") as the set

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}.$$

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty,$$

General Properties of Asymptotic Notations

Transitivity:

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)) ,$$

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)) ,$$

$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n)) ,$$

$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \text{ imply } f(n) = o(h(n)) ,$$

$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \text{ imply } f(n) = \omega(h(n)) .$$

Reflexivity:

$$f(n) = \Theta(f(n)) ,$$

$$f(n) = O(f(n)) ,$$

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

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) ,$$

$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .$$

Asymptotic Bounds of Some Common Functions I

Polynomials

Recall that a polynomial is a function that can be written in the form $f(n) = a_0 + a_1n + a_2n^2 + \dots + a_dn^d$ for some integer constant $d > 0$, where the final coefficient a_d is nonzero. This value d is called the degree of the polynomial. **A basic fact about polynomials is that their asymptotic rate of growth is determined by their “high-order term”—the one that determines the degree.**

Asymptotic Bounds of Some Common Functions II

Logarithms

Recall that $\log_b n$ is the number x such that $b^x = n$. One way to get an approximate sense of how fast $\log_b n$ grows is to note that, if we round it down to the nearest integer, it is one less than the number of digits in the base- b representation of the number n . In particular, for every base b , the function $\log_b n$ is asymptotically bounded by every function of the form n^x , even for (noninteger) values of x arbitrary close to 0.

Asymptotic Bounds of Some Common Functions III

Exponential

Exponential functions are functions of the form $f(n) = r^n$ for some constant base r . Here we will be concerned with the case in which $r > 1$, which results in a very fast-growing function. In particular, where polynomials raise n to a fixed exponent, exponentials raise a fixed number to n as a power; this leads to much faster rates of growth.

Problems Related To Asymptotic Notations

- Explain why the statement, “The running time of algorithm A is at least $O(n^2)$ ” is meaningless.

Thank You

Recurrences: L9

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

Introduction

- Until now we have seen the running time of algorithms that contain loops.
- Recurrence functions involve a recursive call to itself.
- The running time of a recurrence function can be defined using a recurrence relation.

Write a Program in C to recursively find the first n fibonacci numbers

Input: n = 5

Output:

Fibonacci series of 5 numbers is : 0 1 1 2 3

Write a Program to Find the factorial of n using Recursion

Divide and Conquer Paradigm

Divide. Conquer. Merge

- Divide the problem into a number of subproblems that are smaller instances of the same problem.
- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine the solutions to the subproblems into the solution for the original problem.

Merge Sort

- **Divide:** Divide the n -element sequence to be sorted into two subsequences of $n=2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.



Merge Sort

- The recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.
- The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. We merge by calling an auxiliary procedure $\text{MERGE}(A, p, q, r)$, where A is an array and p , q , and r are indices into the array such that $p \leq q < r$.
- Our MERGE procedure takes time $\theta(n)$, where $n = r-p+1$ is the total number of elements being merged

Algorithm 1

```
MergeSort(arr[], l, r)
If r > l
    1. Find the middle point to divide the array into two halves:
        middle m = l+ (r-l)/2
    2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

Algorithm 2: Merge Procedure

MERGE(A, p, q, r)

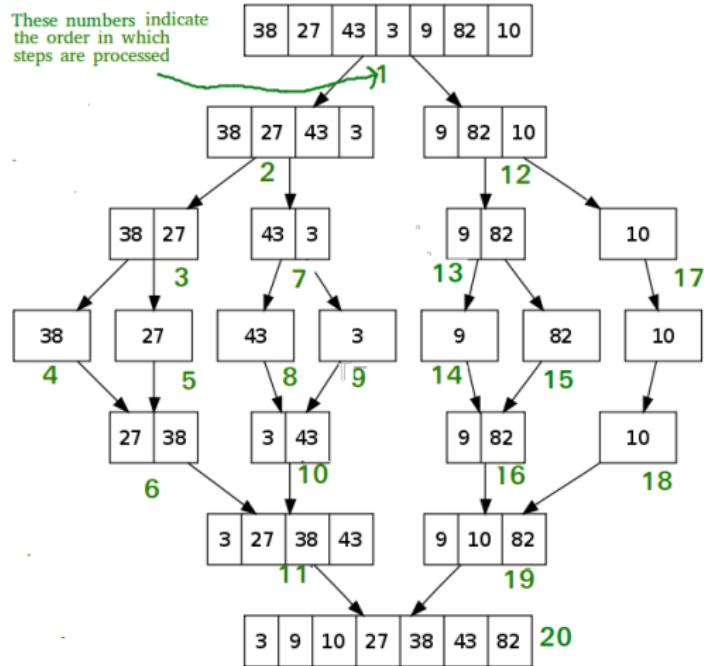
```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

Merge Sort Procedure

MERGE-SORT(A, p, r)

- 1 **if** $p < r$
- 2 $q = \lfloor (p + r)/2 \rfloor$
- 3 MERGE-SORT(A, p, q)
- 4 MERGE-SORT($A, q + 1, r$)
- 5 MERGE(A, p, q, r)

An Example



Analysis of Merge Sort Procedure

- To see that the MERGE procedure runs in $\theta(n)$ time, where $n=r-p+1$, observe that each of lines 1–3 and 8–11 takes constant time, the for loops of lines 4–7 take $\theta(n_1 + n_2) = \theta(n)$ time,⁷ and there are n iterations of the for loop of lines 12–17, each of which takes constant time.
- The procedure MERGE-SORT(A, p, q, r) sorts the elements in the subarray $A[p \dots r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p \dots r]$ into two subarrays: $A[p \dots q]$ containing $\text{Floor}(n/2)$ elements, and $A[q+1 \dots r]$ containing $\text{Floor}(n/2)$ elements

Properties of Merge Sort I

Initialization

Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p \dots k-1]$ is empty. This empty subarray contains the $k-p=0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Properties of Merge Sort II

Maintenance

To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A . Because $A[p \dots k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \dots k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the for loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Properties of Merge Sort III

Termination

At termination, $k = r+1$. By the loop invariant, the subarray $A[p \dots k-1]$, which is $A[p \dots r]$, contains the $k-p=r-p+1$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 1 = r - p + 1$ elements. All but the two largest have been copied back into A , and these two largest elements are the sentinels.

Thank You

Recurrences Contd..: L10 and L11

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Recurrence Relation of Merge Sort

The merge Sort Procedure

MERGE-SORT(A, p, r)

1 if $p < r$

2 $q = \lfloor (p + r)/2 \rfloor$

3 MERGE-SORT(A, p, q)

4 MERGE-SORT($A, q + 1, r$)

5 MERGE(A, p, q, r)

Recurrence Relation of Merge Sort

○●○○○○○○○○○○○○

How the Merge Sort Procedure Works for Input: A={5,2,4,7,1,3,4,6}

The working of the merge procedure

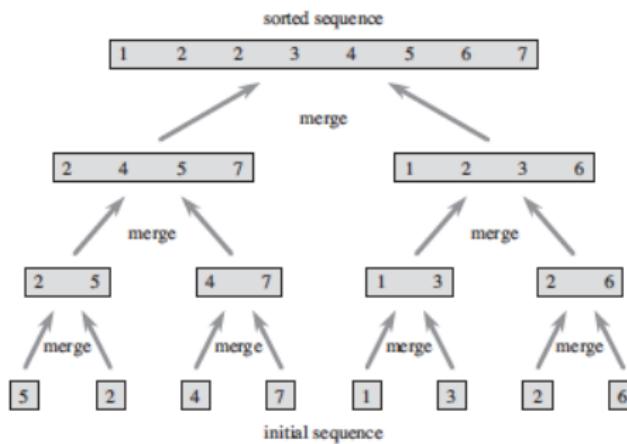


Figure 2.4 The operation of merge sort on the array $A = \{5, 2, 4, 7, 1, 3, 2, 6\}$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Analysis of Merge Sort

Building up the Recurrence

- As before, we let $T(n)$ be the running time on a problem of size n . If the problem size is small enough, say $n \leq c$ for some constant c , the straightforward solution takes constant time, which we write as $\theta(1)$
- Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.)
- If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence
-

$$T(n) = \begin{cases} \theta(1) & \text{for } n = 1, \\ aT(n/b) + D(n) + C(n) & \text{Otherwise} \end{cases}$$

Analysis of Merge Sort

Analysis of the Recurrence

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

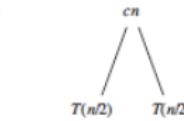
Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n -element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

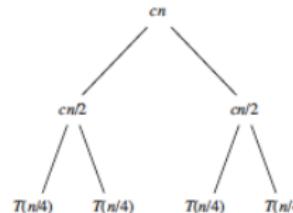
When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\theta(n)$ and a function that is $\theta(1)$. This sum is a linear function of n , that is, $\theta(n)$. Adding it to the $2T(n/2)$ term from the “conquer” step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \theta(1) & \text{for } n = 1, \\ aT(n/b) + \theta(n) & \text{Otherwise} \end{cases}$$

Solving the Recursion

 $T(n)$ 

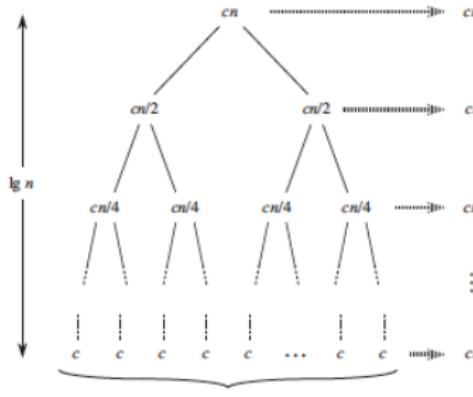
(a)

 cn 

(b)

 (c) 

(a)

 (b) (c) \downarrow 

(d)

Total: $cn \lg n + cn$

Is the runtime Correct

Base

The base case occurs when $n = 1$, in which case the tree has only one level. Since $\lg 1 = 0$, we have that $\lg_n C 1$ gives the correct number of levels.

Hypothesis

Now assume as an inductive hypothesis that the number of levels of a recursion tree with 2^i leaves is $\lg 2^i C 1 \leq i C 1$ (since for any value of i , we have that $\lg 2^i \leq i$). Because we are assuming that the input size is a power of 2, the next input size to consider is $2^{i+1} C 1$.

Induction

A tree with $n = 2^{i+1}$ leaves has one more level than a tree with 2^i leaves, and so the total number of levels is $i + 1 + 1 = \lg 2^{i+1} + 1$.

Cormen Page 29

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A, and exchange it with A[2]. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in θ notation.

Algorithm 4 Selection Sort

```
1: for  $i = 1$  to  $n - 1$  do
2:    $min = i$ 
3:   for  $j = i + 1$  to  $n$  do
4:     // Find the index of the  $i^{th}$  smallest element
5:     if  $A[j] < A[min]$  then
6:        $min = j$ 
7:     end if
8:   end for
9:   Swap  $A[min]$  and  $A[i]$ 
10: end for
```

This yields a running time of

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

Practice Questions I

Exercises

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \{3, 41, 52, 26, 38, 57, 9, 49\}$.

2.4 Inversions

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an *inversion* of A .

- List the five inversions of the array $\{2, 3, 8, 6, 1\}$.

Practice Questions II

2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

2.3-4

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

Practice Questions III

2.3-6

Observe that the **while** loop of lines 5–7 of the **INSERTION-SORT** procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j - 1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Thank You

Solving Recurrences:

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Recurrence Solving Methods

2 MAster's Method

Solving Recurrences

Recall: Integer Multiplication

- Let $X = \boxed{A} \boxed{B}$ and $Y = \boxed{C} \boxed{D}$ where A,B,C and D are $n/2$ bit integers
- Simple Method: $XY = (2^{n/2}A+B)(2^{n/2}C+D)$
- Running Time Recurrence

$$T(n) < 4T(n/2) + 100n$$

How do we solve it?

Solving Recurrences: Substitution

Substitution method

The most general method:

1. **Guess** the form of the solution.
2. **Verify** by induction.
3. **Solve** for constants.

Example: $T(n) = 4T(n/2) + 100n$

1. [Assume that $T(1) = \Theta(1)$.]
2. Guess $O(n^3)$. (Prove O and Ω separately.)
3. Assume that $T(k) \leq ck^3$ for $k < n$.
4. Prove $T(n) \leq cn^3$ by induction.

Solving Recurrences: Substitution

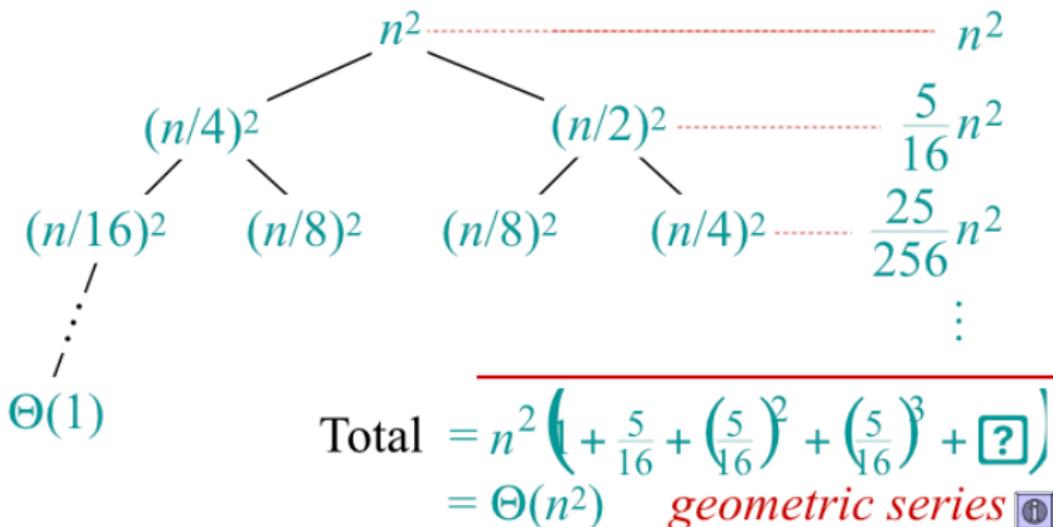
Example of substitution

$$\begin{aligned}T(n) &= 4T(n/2) + 100n \\&\leq 4c(n/2)^3 + 100n \\&= (c/2)n^3 + 100n \\&= cn^3 - ((c/2)n^3 - 100n) \quad \leftarrow \text{desired - residual} \\&\leq cn^3 \quad \leftarrow \text{desired}\end{aligned}$$

whenever $(c/2)n^3 - 100n \geq 0$, for example, if $c \geq 200$ and $n \geq 1$.
residual

Solving Recurrences: Recurrence Tree

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

Solving Recurrences: Recurrence Tree

Appendix: geometric series

$$1 + x + x^2 + \boxed{?} + x^n = \frac{1 - x^{n+1}}{1 - x} \text{ for } x \neq 1$$

$$1 + x + x^2 + \boxed{?} = \frac{1}{1 - x} \text{ for } |x| < 1$$

Return to last
slide viewed.



Solving Recurrences: Master Method

The master method

The master method applies to recurrences of the form

$$T(n) = a T(n/b) + f(n),$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Master Method: Motivations

Motivation: Asymptotic Behavior of Recursive Algorithms

- The time complexity of the algorithm is represented in the form of recurrence relation.
- When analyzing algorithms, recall that **we only care about the asymptotic behavior**
- Rather than solving exactly the recurrence relation associated with the cost of an algorithm, it is sufficient to give an **asymptotic characterization**
- The main tool for doing this is the master theorem

Master Method: A Recurrence

Review

A recursive definition of a sequence specifies

- Initial conditions
- Recurrence relation

Example:

$$a_0=0 \text{ and } a_1=3$$

Initial conditions

$$a_n = 2a_{n-1} - a_{n-2}$$

Recurrence relation

$$a_n = 3n$$

Solution

Master Method: Theorem

Master Theorem

- Let $T(n)$ be a monotonically increasing function that satisfies

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

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Method: Example 1

Master Theorem: Example 1

- Let $T(n) = T(n/2) + \frac{1}{2} n^2 + n$. What are the parameters?

$$a = 1$$

$$b = 2$$

$$d = 2$$

Therefore, which condition applies?

$1 < 2^2$, case 1 applies

- We conclude that

$$T(n) \in \Theta(n^d) = \Theta(n^2)$$

Master Method: Example 2

Master Theorem: Example 2

- Let $T(n) = 2 T(n/4) + \sqrt{n} + 42$. What are the parameters?
 $a = 2$
 $b = 4$
 $d = 1/2$

Therefore, which condition applies?

$$2 = 4^{1/2}, \text{ case 2 applies}$$

- We conclude that

$$T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$$

Master Method: Example 3

Master Theorem: Example 3

- Let $T(n) = 3T(n/2) + 3/4n + 1$. What are the parameters?

$$a = 3$$

$$b = 2$$

$$d = 1$$

Therefore, which condition applies?

$3 > 2^1$, case 3 applies

- We conclude that

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

- Note that $\log_2 3 \approx 1.584\dots$, can we say that $T(n) \in \Theta(n^{1.584})$

No, because $\log_2 3 \approx 1.5849\dots$ and $n^{1.584} \notin \Theta(n^{1.5849})$

Master Method: Example 4

Master Theorem: Example 4

- $T(n) = 2T(\sqrt{n}) + \log n.$
 - Let $n = 2^m \Rightarrow m = \log n$
 - Then $T(2^m) = 2T(2^{m/2}) + m.$
 - Now let $S(m) = T(2^m).$
 - Then $S(m) = 2S(m/2) + m.$
 - This is case-2 of master theorem and has the solution
 - $S(m) = O(m \log m).$
 - So $T(n) = T(2^m)$
 - $\Rightarrow S(m) = O(m \log m) = O(\log n \log \log n).$

Master Method: Example 5

Master Theorem:

Example

What is the value of following recurrence.

$$T(n) = 5T(n/5) + \sqrt{n},$$

$$T(1) = 1,$$

$$T(0) = 0$$

- (A) Theta (n)
- (B) Theta (n^2)
- (C) Theta (sqrt(n))
- (D) Theta (nLogn)

$$a=5, b=5, d=1/2$$

$$a>b^d \Rightarrow \Theta(n^{\log_5 5}) = \Theta(n)$$

Answer: (A)

More Examples of Master Method

More Examples of Master's Theorem

- $T(n) = 3T(n/5) + n$ $\Theta(n)$
- $T(n) = 2T(n/2) + n$ $\Theta(n \log n)$
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$
- $T(n) = T(n/2) + n$ $\Theta(n)$
- $T(n) = T(n/2) + 1$ $\Theta(\log n)$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$

Limitations of Master's Method

Master Theorem: Pitfalls

- You **cannot** use the Master Theorem if
 - $T(n)$ is not monotone, e.g. $T(n) = \sin(x)$
 - $f(n)$ is not a polynomial, e.g., $T(n)=2T(n/2)+2^n$
 - b cannot be expressed as a constant, e.g.

$$T(n) = T(\sqrt{n})$$

- Note that the Master Theorem does not solve the recurrence equation
- Does the base case remain a concern?

Home Assignments I

Practice Problems

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 3T(n/2) + n^2$

2. $T(n) = 4T(n/2) + n^2$

3. $T(n) = T(n/2) + 2^n$

4. $T(n) = 2^n T(n/2) + n^n$

5. $T(n) = 16T(n/4) + n$

6. $T(n) = 2T(n/2) + n \log n$

Home Assignments II

7. $T(n) = 2T(n/2) + n/\log n$

8. $T(n) = 2T(n/4) + n^{0.51}$

9. $T(n) = 0.5T(n/2) + 1/n$

10. $T(n) = 16T(n/4) + n!$

11. $T(n) = \sqrt{2}T(n/2) + \log n$

12. $T(n) = 3T(n/2) + n$

13. $T(n) = 3T(n/3) + \sqrt{n}$

14. $T(n) = 4T(n/2) + cn$

15. $T(n) = 3T(n/4) + n \log n$

16. $T(n) = 3T(n/3) + n/2$

17. $T(n) = 6T(n/3) + n^2 \log n$

18. $T(n) = 4T(n/2) + n/\log n$

19. $T(n) = 64T(n/8) - n^2 \log n$

20. $T(n) = 7T(n/3) + n^2$

21. $T(n) = 4T(n/2) + \log n$

Thank You

Heap and Heap Sort: L13

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

- 1 Sorting and Searching
 - Sorting: Heap Sort

Sorting and Searching

Sorting

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

The input sequence is usually an n -element array, although it may be represented in some other fashion, such as a linked list.

Searching

A search algorithm is the step-by-step procedure used to locate specific data among a collection of data. It is considered a fundamental procedure in computing.

Why Sorting

- Sometimes an application inherently needs to sort information. For example, in order to prepare customer statements, banks need to sort checks by check number.
- Algorithms often use sorting as a key subroutine.
- In fact, many important techniques used throughout algorithm design appear in the body of sorting algorithms that have been developed over the years.
- Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors.

Sorting Algorithms until now

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

Heaps I

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree, as shown in Figure 1.

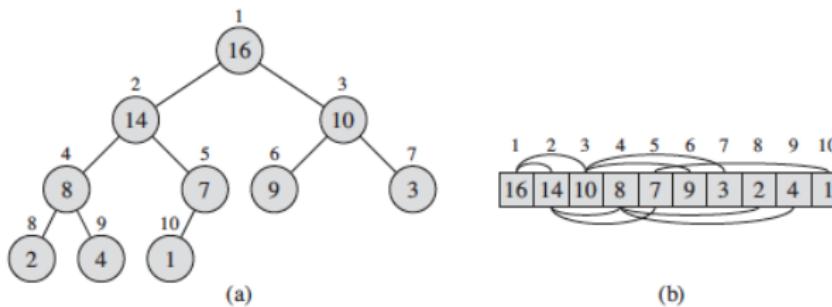


Figure 6.1 A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

Heaps II

The root of the tree is $A[1]$, and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left by one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit. The

Heaps III

There are two kinds of binary heaps: max-heaps and min-heaps.
In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap.

In a max-heap, the max-heap property is that for every node i other than the root.

$$A[\text{PARENT}(i)] \geq A[i] ;$$

Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

The min-heap property is that for every node i other than the root.

$$A[\text{PARENT}(i)] \leq A[i] ;$$

The smallest element in a min-heap is at the root.

Heaps IV

- For the heapsort algorithm, we use max-heaps. Min-heaps commonly implement priority queues.
- Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root.
- Since a heap of n elements is based on a complete binary tree, its height is $\theta(\lg n)$

Major Procedures USed in Heap Sort

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a maxheap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.

Practice Questions

6.1-4

Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

6.1-5

Is an array that is in sorted order a min-heap?

6.1-6

Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

6.1-7

Show that, with the array representation for storing an n -element heap, the leaves are the nodes indexed by $\lceil n/2 \rceil + 1, \lceil n/2 \rceil + 2, \dots, n$.

Maintaining the Heap Property

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $largest = l$ 
5  else  $largest = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7       $largest = r$ 
8  if  $largest \neq i$ 
9      exchange  $A[i]$  with  $A[largest]$ 
10     MAX-HEAPIFY( $A, largest$ )
```

Action of MAX-Heapify

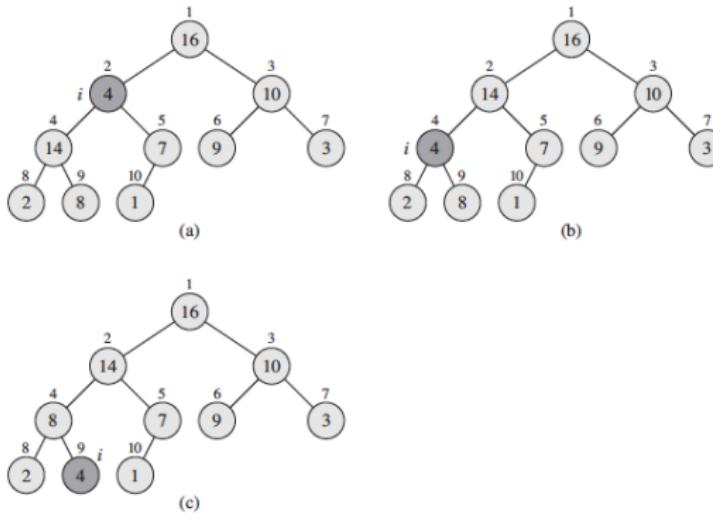


Figure 6.2 The action of $\text{MAX-HEAPIFY}(A, 2)$, where $A.\text{heap-size} = 10$. (a) The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call $\text{MAX-HEAPIFY}(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in (c), node 4 is fixed up, and the recursive call $\text{MAX-HEAPIFY}(A, 9)$ yields no further change to the data structure.

Thank You

Heap and Heap Sort contd...: *L₁₄*

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos,
Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena,
Springer Publication
- (R2) **Introduction to Algorithms by CLRS, PHI
Publication**

Outline

- 1 Sorting and Searching
 - Sorting: Heap Sort

Major Procedures Used in Heap Sort

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.

Maintaining the Heap Property using MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $largest = l$ 
5  else  $largest = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7       $largest = r$ 
8  if  $largest \neq i$ 
9      exchange  $A[i]$  with  $A[largest]$ 
10     MAX-HEAPIFY( $A, largest$ )
```

Action of MAX-HEAPIFY

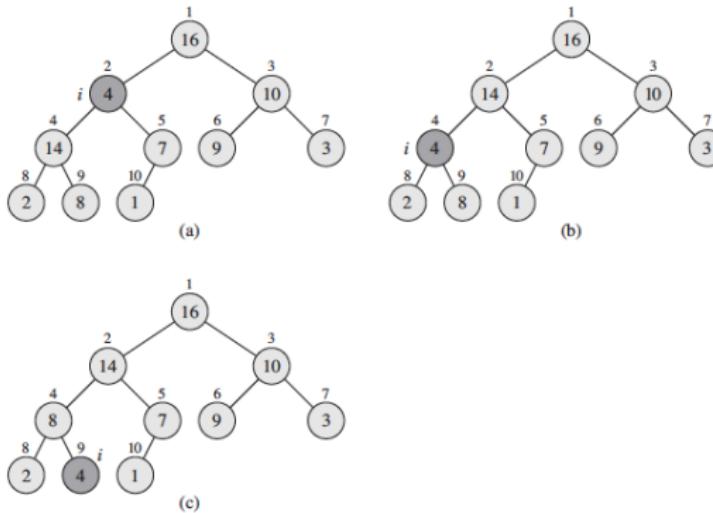


Figure 6.2 The action of $\text{MAX-HEAPIFY}(A, 2)$, where $A.\text{heap-size} = 10$. (a) The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call $\text{MAX-HEAPIFY}(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in (c), node 4 is fixed up, and the recursive call $\text{MAX-HEAPIFY}(A, 9)$ yields no further change to the data structure.

Running Time of MAX-HEAPIFY

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$ —the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1).$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAXHEAPIFY on a node of height h as $O(h)$.

Practice Questions

Exercises

6.2-1

Using Figure 6.2 as a model, illustrate the operation of $\text{MAX-HEAPIFY}(A, 3)$ on the array $A = \{27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\}$.

6.2-2

Starting with the procedure MAX-HEAPIFY , write pseudocode for the procedure $\text{MIN-HEAPIFY}(A, i)$, which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY ?

6.2-3

What is the effect of calling $\text{MAX-HEAPIFY}(A, i)$ when the element $A[i]$ is larger than its children?

6.2-4

What is the effect of calling $\text{MAX-HEAPIFY}(A, i)$ for $i > A.\text{heap-size}/2$?

6.2-5

The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-6

Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

Building a Heap using BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \lfloor A.\text{length}/2 \rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)

The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

Illustration of BUILD-MAX-HEAP

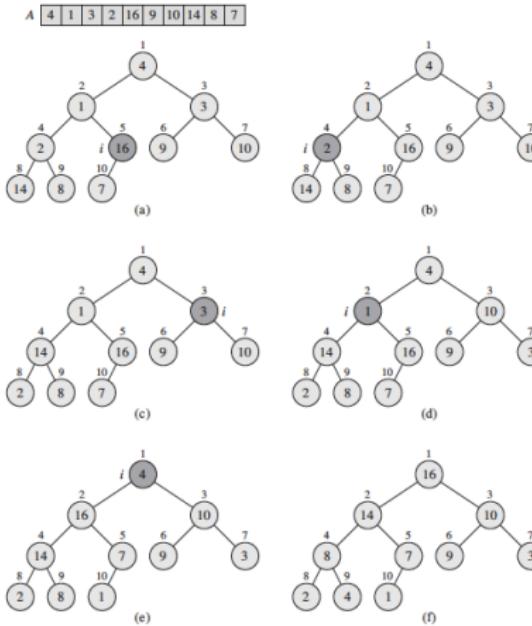


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. (a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). (b) The data structure that results. The loop index i for the next iteration refers to node 4. (c)–(e) Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, both the subtrees of that node are both max-heaps. (f) The max-heap after BUILD-MAX-HEAP finishes.

Proof of correctness using Loop Invariant

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call **MAX-HEAPIFY**(A, i) to make node i a max-heap root. Moreover, the **MAX-HEAPIFY** call preserves the property that nodes $i + 1, i + 2, \dots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap. In particular, node 1 is.

Cost of Running BUILD-MAX-HEAP

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, and so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right).$$

We evaluate the last summation by substituting $x = 1/2$ in the formula (A.8), yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1 - 1/2)^2} \\ &= 2. \end{aligned}$$

Thus, we can bound the running time of BUILD-MAX-HEAP as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n). \end{aligned}$$

Hence, we can build a max-heap from an unordered array in linear time.

Practice Questions

6.3-1

Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2

Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

The Heap Sort Algorithm

HEAPSORT(A)

- 1 BUILD-MAX-HEAP(A)
- 2 for $i = A.length$ downto 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.heap-size = A.heap-size - 1$
- 5 MAX-HEAPIFY($A, 1$)

Example of the Heapsort Algorithm

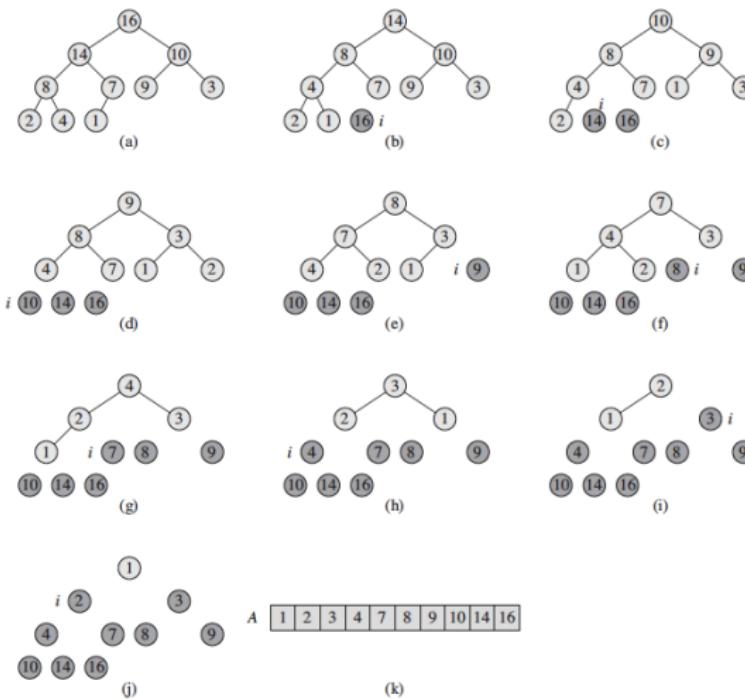


Figure 6.4 The operation of HEAPSORT. (a) The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. (b)-(j) The max-heap just after each call of MAX-HEAPlFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. (k) The resulting sorted array A .

Home Assignment

6.4-5 ★

Show that when all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

Thank You

Sorting and Searching: External and internal sorting; Inplace sorting; Stable sorting; Special cases of sorting and searching etc.: *L₁₅*

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication Reference book
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) **Introduction to Algorithms by CLRS, PHI Publication**

Outline

- 1 Radix Sort
- 2 Comparison Sort V/S Counting Sort
- 3 Internal and External Sorting
- 4 Stable Sorting
- 5 Special Cases of Sorting and Searching

Radix Sort

In computer science, radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix.

- Radix sort incorporates the counting sort algorithm so that it can sort larger, multi-digit numbers.
- If we have $\log_2 n$ bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$\Omega(n+k)$	$\Theta(nk)$	$O(nk)$
Space Complexity			$O(n+k)$

The Process of Radix Sort

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.



Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

82, 901, 100, 12, 150, 77, 55 & 23



Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

The Process of Radix Sort

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

100, 150, 901, 82, 12, 23, 55 & 77



Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundreds placed digit) of every number.

100, 901, 12, 23, 150, 55, 77 & 82



Group all the numbers from queue-0 to queue-9 in the order they have inserted &

Comparison Sort V/S Counting Sort

Comparison Sort

- A comparison sort is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation (often a "less than or equal to" operator or a three-way comparison) that determines which of two elements should occur first in the final sorted list.
- Quicksort Heapsort Shellsort

Merge sort Introsort Insertion sort

Selection sort Bubble sort

Odd–even sort Cocktail shaker sort

Cycle sort Merge-insertion sort

Smoothsort Timsort Block sort

Counting Sort

- counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm.
- Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.
- Radix Sort

Internal and External Sorting

Internal Sort

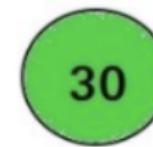
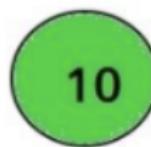
- An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.
- Bubble Sort Insertion Sort Quick Sort Heap Sort Radix Sort Selection sort

External Sort

- External sorting is a class of sorting algorithms that can handle massive amounts of data.
- External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive.
- External sorting algorithms generally fall into two types, distribution sorting, which resembles quicksort, and external merge sort, which resembles merge sort.

Stable Sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.



Sorting is stable because the order of balls is maintained when values are same. The ball with green color and value 10 appears before the orange color ball with value 10. Similarly order is maintained for 20.



Special Cases of Sorting and Searching I

How many keys will you be sorting?

- For small amounts of data (say $n = 100$), it really doesn't matter much which of the quadratic-time algorithms you use. Insertion sort is faster, simpler, and less likely to be buggy than bubblesort.
- When you have more than 100 items to sort, it is important to use an $O(n \lg n)$ -time algorithm like heapsort, quicksort, or mergesort.
- Once you get past (say) 5,000,000 items, it is important to start thinking about external-memory sorting algorithms that minimize disk access.

Special Cases of Sorting and Searching II

Will there be duplicate keys in the data?

- The sorted order is completely defined if all items have distinct keys. However, when two items share the same key, something else must determine which one comes first. In many applications it doesn't matter, so any sorting algorithm suffices.
- Occasionally, ties need to be broken by their initial position in the data set.

Has the data already been partially sorted?

If so, certain algorithms like insertion sort perform better than they otherwise would.

Special Cases of Sorting and Searching III

Are your keys very long or hard to compare?

- If your keys are long text strings, it might pay to use a relatively short prefix (say ten characters) of each key for an initial sort, and then resolve ties using the full key.
- Another idea might be to use radix sort. This always takes time linear in the number of characters in the file, instead of $O(n \lg n)$ times the cost of comparing two keys.

Do I have to worry about disk accesses?

- In massive sorting problems, it may not be possible to keep all data in memory simultaneously. Such a problem is called external sorting, because one must use an external storage device.

Thank You

Basic definitions, applications and representations: L_{16}

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

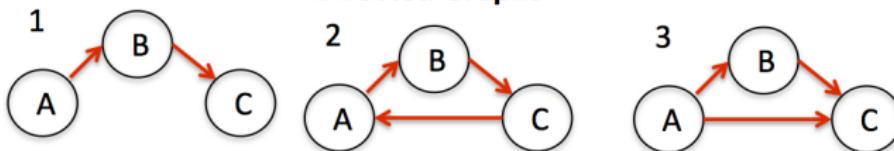
1 Basic definitions, applications and representations

Basic Definition

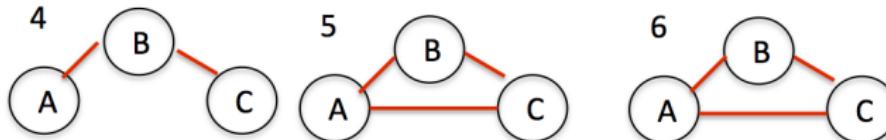
Graph

A directed graph G consists of a set of nodes V and a set of directed edges E . Each $e \in E$ is an ordered pair (u, v) ; in other words, the roles of u and v are not interchangeable, and we call u the tail of the edge and v the head. We will also say that edge e leaves node u and enters node v .

Directed Graphs

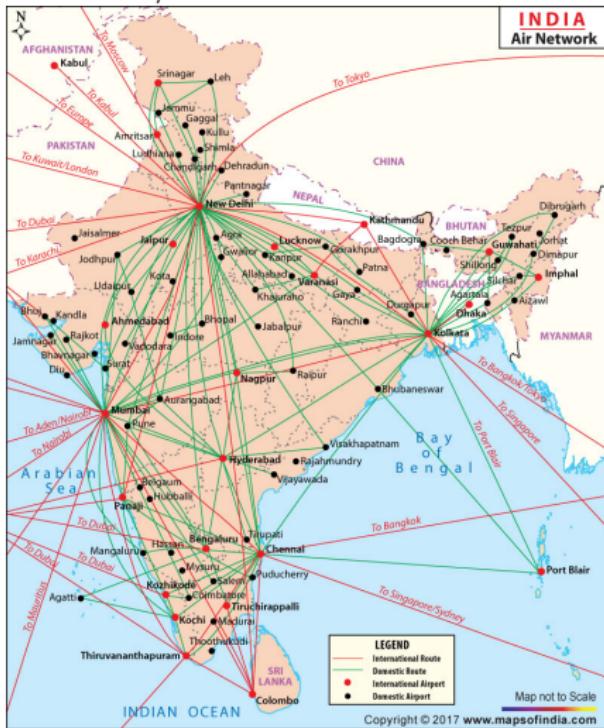


Undirected Graphs



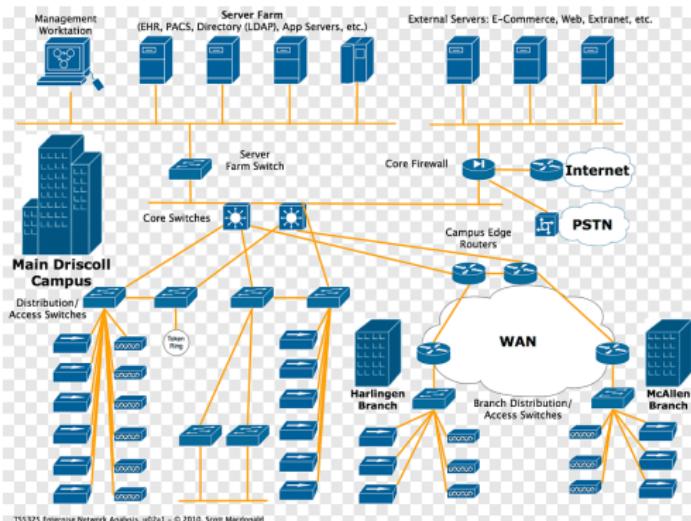
Some Applications I

Transportation Networks/ Airlines Network



Some Applications II

Communication Network/ Network topology



Some Applications III

Social Networks



Some Applications IV

Dependency Networks

It is natural to define directed graphs that capture the interdependencies among a collection of objects. For example, given the list of courses offered by a college or university, we could have a node for each course and an edge from u to v if u is a prerequisite for v . Given a list of functions or modules in a large software system, we could have a node for each function and an edge from u to v if u invokes v by a function call. Or given a set of species in an ecosystem, we could define a graph—a food web—in which the nodes are the different species and there is an edge from u to v if u consumes v .

Some Applications V

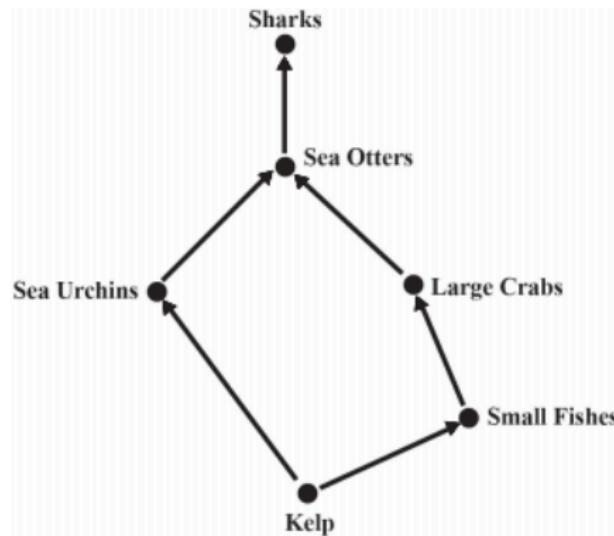


FIGURE 2.1 Simple food web.

Path

One of the fundamental operations in a graph is that of traversing a sequence of nodes connected by edges.

Path

With this notion in mind, we define a path in an undirected graph $G = (V, E)$ to be a sequence P of nodes v_1, v_2, \dots, v_k with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in G . P is often called a path from v_1 to v_k , or a $v_1 - v_k$ path.

Simple Path and Cyclic Paths

A path is called simple if all its vertices are distinct from one another. A cycle is a path v_1, v_2, \dots, v_k in which $k > 2$, the first $k-1$ nodes are all distinct, and $v_1 = v_k$ —in other words, the sequence of nodes “cycles back” to where it began.

Example of a Path

For example, the nodes 4, 2, 1, 7, 8 form a path in Figure 3.1.

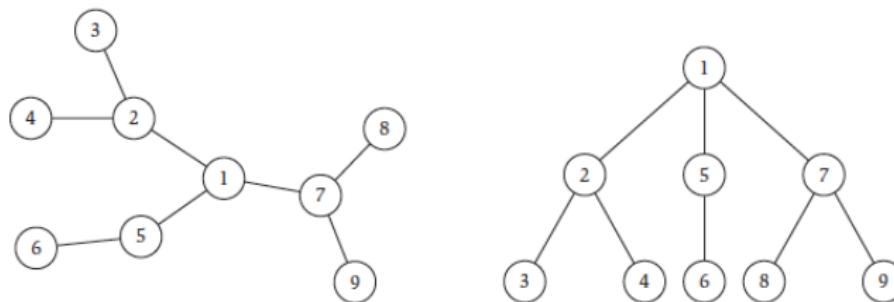


Figure 3.1 Two drawings of the same tree. On the right, the tree is rooted at node 1.

Connectivity

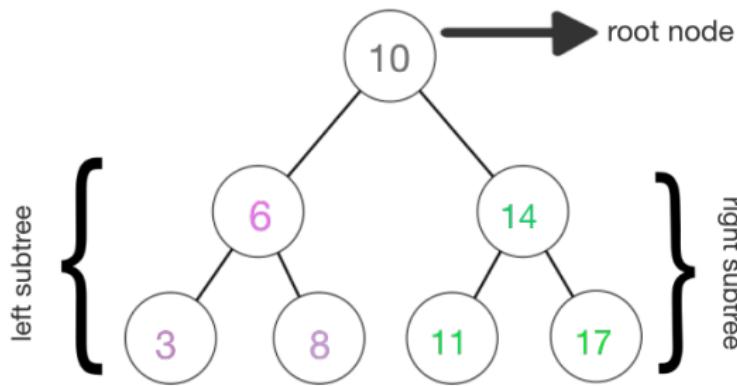
We say that an undirected graph is connected if, for every pair of nodes u and v , there is a path from u to v .

Distance of 2 Nodes

In addition to simply knowing about the existence of a path between some pair of nodes u and v , we may also want to know whether there is a short path. Thus we define the distance between two nodes u and v to be the minimum number of edges in a u - v path. (We can designate some symbol like ∞ to denote the distance between nodes that are not connected by a path.) The term distance here comes from imagining G as representing a communication or transportation network; if we want to get from u to v , we may well want a route with as few “hops” as possible.

Tree

We say that an undirected graph is a tree if it is connected and does not contain a cycle.



A logical analysis

(3.1) *Every n -node tree has exactly $n - 1$ edges.*

In fact, the following stronger statement is true, although we do not prove it here.

(3.2) *Let G be an undirected graph on n nodes. Any two of the following statements implies the third.*

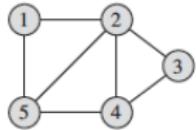
- (i) G is connected.*
- (ii) G does not contain a cycle.*
- (iii) G has $n - 1$ edges.*

We now turn to the role of trees in the fundamental algorithmic idea of *graph traversal*.

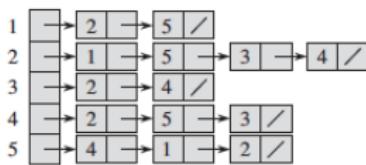
Graph Representation

- We can choose between two standard ways to represent a graph $G \in \mathcal{D}(\mathcal{V}; \mathcal{E})$: as a collection of adjacency lists or as an adjacency matrix.
- Either way applies to both directed and undirected graphs.
- Because the adjacency-list representation provides a compact way to represent sparse graphs—those for which $|E|$ is much less than $|\mathcal{V}|^2 / 2$ —it is usually the method of choice.

Representation of Undirected Graph



(a)



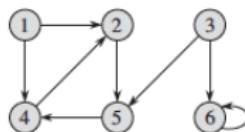
(b)

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

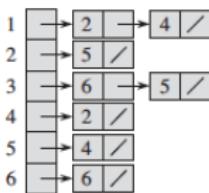
(c)

Figure 22.1 Two representations of an undirected graph. (a) An undirected graph G with 5 vertices and 7 edges. (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G .

Representation of Directed Graph



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figure 22.2 Two representations of a directed graph. (a) A directed graph G with 6 vertices and 8 edges. (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G .

Adjacency List Representation

- The adjacency-list representation of a graph $G(V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V . For each $u \in V$, the adjacency list $\text{Adj}(u)$ contains all the vertices such that there is an edge $(u, v) \in E$.

Adjacency Matrix Representation

For the *adjacency-matrix representation* of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \dots, |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise .} \end{cases}$$

Practice Problems

Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

Thank You

Graph Traversal:BFS and DFS: *L₁₇*

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Breadth First Search

BFS

- Given a graph $G(V,E)$ and a distinguished source vertex s , breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s .
- It computes the distance (smallest number of edges) from s to each reachable vertex.
- It also produces a “breadth-first tree” with root s that contains all reachable vertices.
- For any vertex reachable from s , the simple path in the breadth-first tree from s to corresponds to a “shortest path” from s to in G , that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.
- Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

Coloured Nodes in BFS

- To keep track of progress, breadth-first search colors each vertex white, gray, or black.
- All vertices start out white and may later become gray and then black.
- A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite.
- If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered.
- Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Steps in BFS

- Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s .
- Whenever the search discovers a white vertex in the course of scanning the adjacency list of an already discovered vertex u , the vertex and the edge (u,v) are added to the tree.
- We say that u is the predecessor or parent of v in the breadth-first tree.
- Since a vertex is discovered at most once, it has at most one parent.

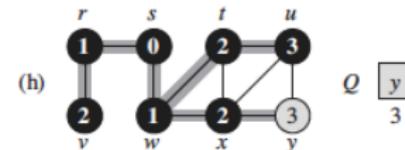
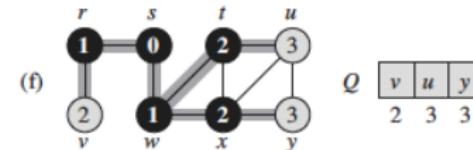
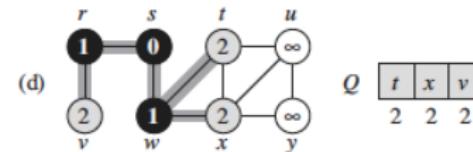
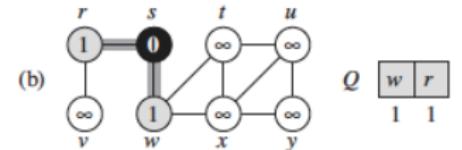
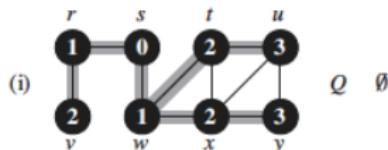
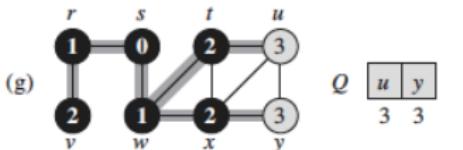
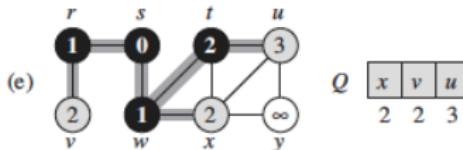
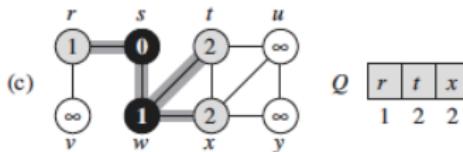
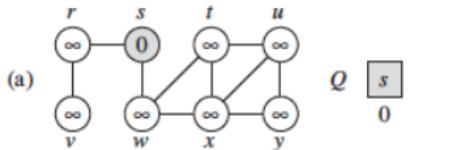
BFS Algorithm

$\text{BFS}(G, s)$

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.\text{color} = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.\text{color} = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.\text{Adj}[u]$ 
13         if  $v.\text{color} == \text{WHITE}$ 
14              $v.\text{color} = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.\text{color} = \text{BLACK}$ 
```

We store the color of each vertex $u \in V$ in the attribute $u.\text{color}$ and the predecessor of u in the attribute $u.\pi$. The attribute $u.d$ holds the distance from the source s to vertex u computed by the algorithm.

Example of BFS



Running time of BFS

After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueueing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $O(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V+E)$.

Shortest Path between (u,v)

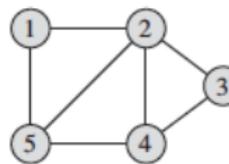
The following procedure prints out the vertices on a shortest path from s to v , assuming that BFS has already computed a breadth-first tree:

```
PRINT-PATH( $G, s, v$ )
1  if  $v == s$ 
2      print  $s$ 
3  elseif  $v.\pi == \text{NIL}$ 
4      print "no path from"  $s$  "to"  $v$  "exists"
5  else PRINT-PATH( $G, s, v.\pi$ )
6      print  $v$ 
```

This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.

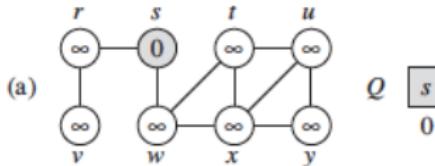
Practice Question

- Show the d and v values that result from running breadth-first search on the directed graph of Figure 22.2(a), using vertex 3 as the source.



(a)

- Show the d and v values that result from running breadth-first search on the undirected graph of Figure 22.3, using vertex u as the source.



(a)

DFS

- The strategy followed by depth-first search is, as its name implies, to search “deeper” in the graph whenever possible.
- Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of 's edges have been explored, the search “backtracks” to explore edges leaving the vertex from which was discovered.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.
- The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in E_{π} are tree edges.

Colouring of nodes in DFS

- As in breadth-first search, depth-first search colors vertices during the search to indicate their state.
- Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Timestamp

Besides creating a depth-first forest, depth-first search also timestamps each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v 's adjacency list (and blackens).

DFS Pseudocode

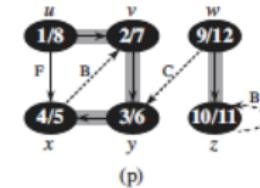
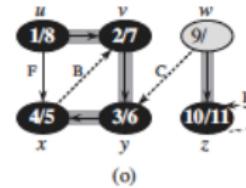
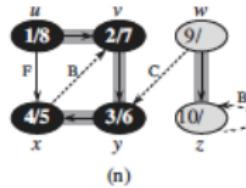
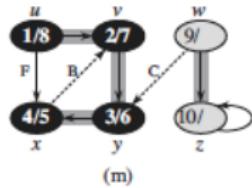
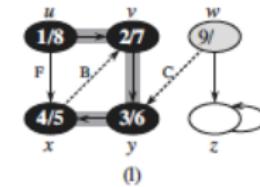
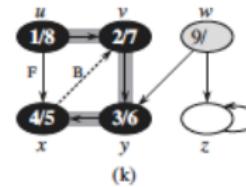
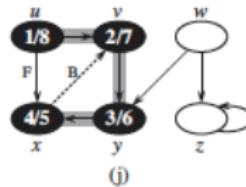
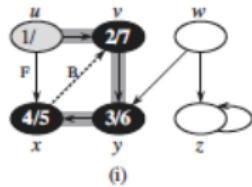
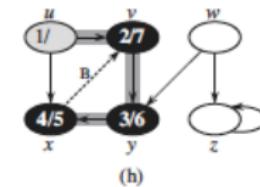
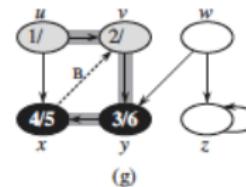
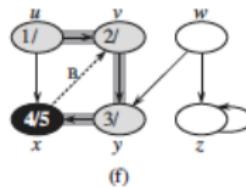
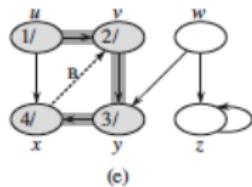
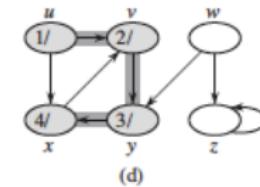
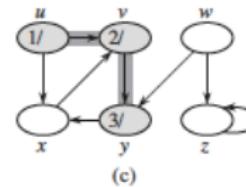
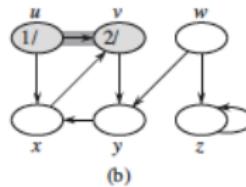
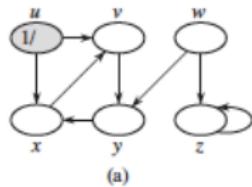
DFS(G)

```
1 for each vertex  $u \in G.V$ 
2      $u.color = \text{WHITE}$ 
3      $u.\pi = \text{NIL}$ 
4      $time = 0$ 
5 for each vertex  $u \in G.V$ 
6     if  $u.color == \text{WHITE}$ 
7         DFS-VISIT( $G, u$ )
```

DFS-VISIT(G, u)

```
1  $time = time + 1$            // white vertex  $u$  has just been discovered
2  $u.d = time$ 
3  $u.color = \text{GRAY}$ 
4 for each  $v \in G.Adj[u]$       // explore edge  $(u, v)$ 
5     if  $v.color == \text{WHITE}$ 
6          $v.\pi = u$ 
7         DFS-VISIT( $G, v$ )
8  $u.color = \text{BLACK}$            // blacken  $u$ ; it is finished
9  $time = time + 1$ 
10  $u.f = time$ 
```

Example of DFS



Breadth First Search
oooooooooooo●○

Properties of DFS

Thank You

Graph Traversal-DFS: L18

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Depth First Search

2 Application of BFS and DFS

3 Variants of BFS and DFS

Depth First Search

- The strategy followed by depth-first search is, as its name implies, to search “**deeper**” in the graph whenever possible.
- Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.** Once all of 's edges have been explored, the search “**backtracks**” to explore edges leaving the vertex from which was discovered.
- If any undiscovered vertices remain**, then depth-first search **selects one of them as a new source, and it repeats the search from that source**. The algorithm repeats this entire process until it has discovered every vertex.
- The predecessor subgraph of a depth-first search **forms a depth-first forest comprising several depth-first trees**. The edges in E_{π} are tree edges.

Colouring of nodes in DFS

- As in breadth-first search, depth-first search **colors vertices during the search** to indicate their state.
- Each vertex is **initially white**, **is grayed when it is discovered** in the search, and **is blackened when it is finished**, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Timestamp

Besides creating a depth-first forest, depth-first search **also timestamps each vertex**. **Each vertex v has two timestamps**: the first timestamp $v.d$ **records when it is first discovered (and grayed)**, and the second timestamp $v.f$ **records when the search finishes examining v 's adjacency list (and blackens)**. **Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter**.

DFS Pseudocode

DFS(G)

```
1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4       $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u$ )
```

DFS-VISIT(G, u)

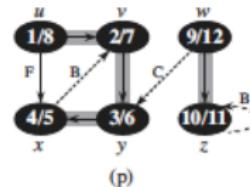
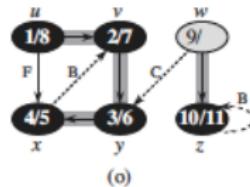
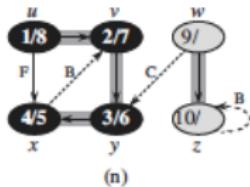
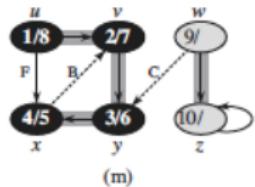
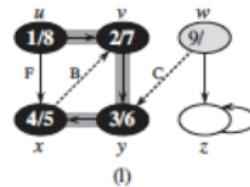
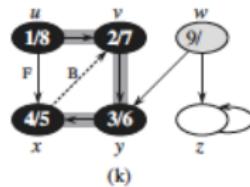
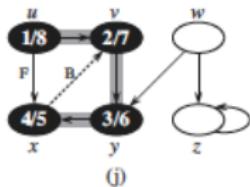
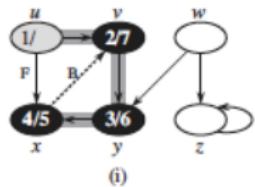
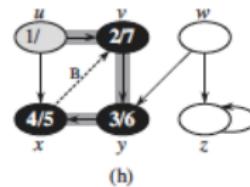
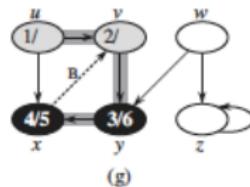
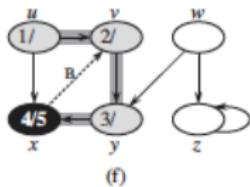
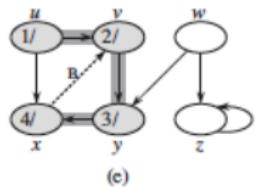
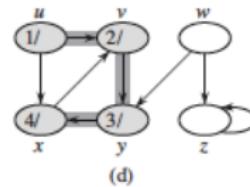
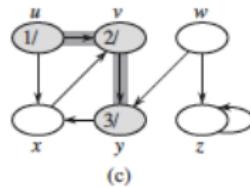
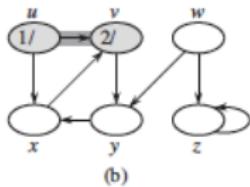
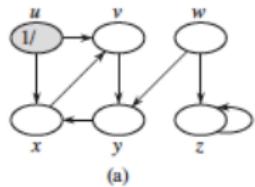
```
1   $time = time + 1$            // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$     // explore edge  $(u, v)$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$          // blacken  $u$ ; it is finished
9   $time = time + 1$ 
10  $u.f = time$ 
```

Depth First Search
○○○●○○○○

Application of BFS and DFS
○○○

Variants of BFS and DFS
○○○○○○

Example of DFS



Running Time of DFS

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop on lines 4–7 executes $|Adj[v]|$ times. Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

the total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V + E)$.

Properties of DFS I

- Depth-first search yields valuable information about the structure of a graph.
- Perhaps the most basic property of depth-first search is that the predecessor subgraph G does indeed form a forest of trees
- Another interesting property of depth-first search is that the search can be used to classify the edges of the input graph $G(V,E)$. The type of each edge can provide important information about a graph.
 - **Tree edges** are edges in the depth-first forest G_π . Edge (u,v) is a tree edge if v was first discovered by exploring edge (u,v) .
 - **Back edges** are those edges (u,v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
 - **Forward edges** (u,v) are those non-tree edges connecting a vertex u to a descendant v in a depth-first tree.
 - **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

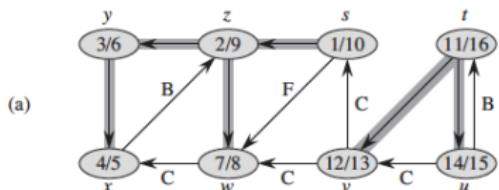
How DFS Identifies Edges

The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when we first explore an edge (u,v) the color of the vertex v tells us something about the edge:

- WHITE indicates a tree edge,
- GRAY indicates a back edge, and
- BLACK indicates a forward or cross edge.

Properties of DFS: Parenthesis Theorem

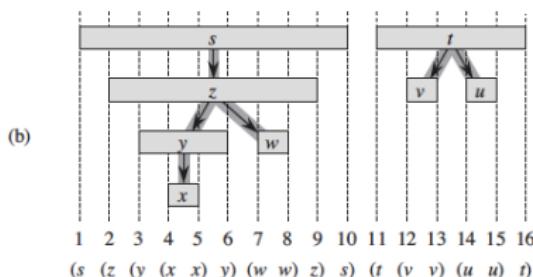
The history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested.



Theorem 22.7 (Parenthesis theorem)

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v , exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.



Practice Problems

- Show how depth-first search works on the graph of Figure 22.6.

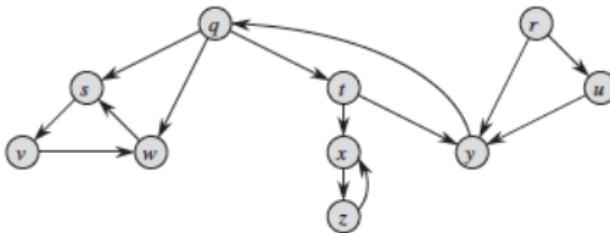


Figure 22.6 A directed graph for use in Exercises 22.3-2 and 22.5-2.

- Give a counterexample to the conjecture that if a directed graph G contains a path from u to, and if $u:d < d$ in a depth-first search of G , then v is a descendant of u in the depth-first forest produced.
- Rewrite the procedure DFS, using a stack to eliminate recursion.

Thank You

Applications of BFS and DFS-Testing Bipartiteness and Connectivity in Directed Graphs: *L₁₉*

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

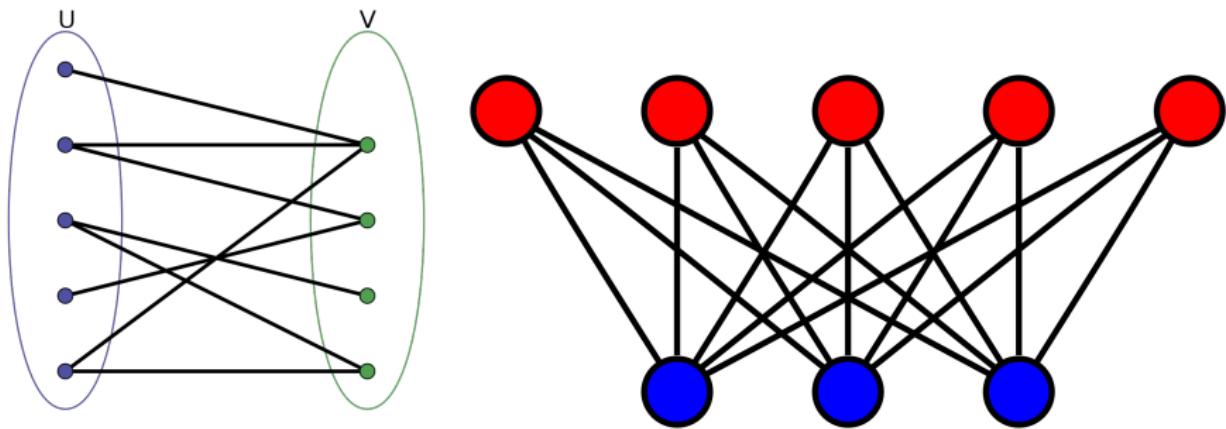
- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

- 1 Bipartite Graphs
- 2 Testing Bipartiteness using BFS
- 3 Testing Bipartiteness using DFS
- 4 Connectivity in Directed Graphs

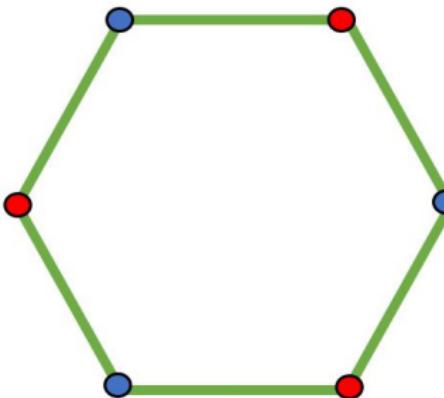
Bipartite Graph

In the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V . Vertex sets U and V are usually called the parts of the graph.



Coloring a Bipartite Graph

- A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color.



Cycle graph of length 6

The Issue with Bipartite Testing

Clearly a triangle is not bipartite, since we can color one node red, another one blue, and then we can't do anything with the third node. More generally, consider a cycle C of odd length, with nodes numbered $1, 2, 3, \dots, 2k, 2k + 1$.

If we color node 1 red, then we must color node 2 blue, and then we must color node 3 red, and so on coloring odd-numbered nodes red and even-numbered nodes blue. But then we must color node $2k + 1$ red, and it has an edge to node 1, which is also red.

This demonstrates that there's no way to partition C into red and blue nodes as required. More generally, if a graph G simply contains an odd cycle, then we can apply the same argument; thus we have established the following.

If a graph G is bipartite, then it cannot contain an odd cycle.

It is easy to recognize that a graph is bipartite when appropriate sets X and Y (i.e., red and blue nodes) have actually been identified for us; and in many settings where bipartite graphs arise, this is natural. But suppose we encounter a graph G with no annotation provided whether there exists a partition into red and blue nodes, as required. How difficult is this? We see from above that an odd cycle is one simple "obstacle" to a graph's being bipartite.



Testing Bipartiteness using BFS

- we pick any node $s \in V$ and color it red; there is no loss in doing this, since s must receive some color.
- It follows that all the neighbors of s must be colored blue, so we do this.
- It then follows that all the neighbors of these nodes must be colored red, their neighbors must be colored blue, and so on, until the whole graph is colored.
- At this point, either we have a valid red/blue coloring of G , in which every edge **has ends of opposite colors, or there is some edge with ends of the same color.**

Similarity to BFS

The first thing to notice is that the coloring procedure we have just described is essentially identical to the description of BFS: we move outward from s , coloring nodes as soon as we first encounter them, coloring odd-numbered layers **blue** and even-numbered layers **red**.



Pseudocode

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Testing Bipartiteness using DFS

- Use a `color[]` array which stores 0 or 1 for every node which denotes opposite colors.
- Call the function `DFS` from any node.
- If the node u has not been visited previously, then assign $!color[v]$ to `color[u]` and call `DFS` again to visit nodes connected to u .
- If at any point, `color[u]` is equal to `color[v]`, then the node is not bipartite. Modify the `DFS` function such that it returns a boolean value at the end.

Pseudocode

```
bool isBipartite(vector<int> adj[], int v,
                  vector<bool>& visited, vector<int>& color)
{
    for (int u : adj[v]) {
        // if vertex u is not explored before
        if (visited[u] == false) {
            // mark present vertic as visited
            visited[u] = true;

            // mark its color opposite to its parent
            color[u] = !color[v];

            // if the subtree rooted at vertex v is not bipartite
            if (!isBipartite(adj, u, visited, color))
                return false;
        }

        // if two adjacent are colored with same color then
        // the graph is not bipartite
        else if (color[u] == color[v])
            return false;
    }
    return true;
}
```

Connectivity in Directed Graphs

Recall that in a directed graph, the edge (u, v) has a direction: it goes from u to v . In this way, the relationship between u and v is asymmetric, and this has qualitative effects on the structure of the resulting graph.

In order to represent a directed graph for purposes of designing algorithms, we use a version of the adjacency list representation that we employed for undirected graphs. Now, instead of each node having a single list of neighbors, each node has two lists associated with it: one list consists of nodes to which it has edges, and a second list consists of nodes from which it has edges.

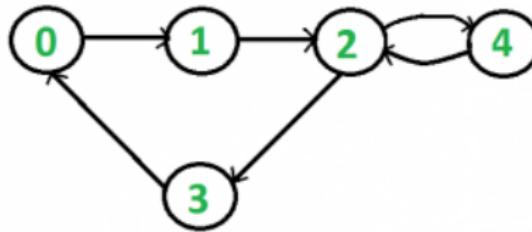
Connectivity in Directed Graphs using BFS

- We start at a node s , define a first layer of nodes to consist of all those to which s has an edge, define a second layer to consist of all additional nodes to which these first-layer nodes have an edge, and so forth.
- In this way, we discover nodes layer by layer as they are reached in this outward search from s , and the nodes in layer j are precisely those for which the shortest path from s has exactly j edges.
- It is important to understand what this directed version of BFS is computing. In directed graphs, it is possible for a node s to have a path to a node t even though t has no path to s ; and what directed BFS is computing is the set of all nodes t with the property that s has a path to t .

Strong Connectivity

a directed graph is strongly connected if, for every two nodes u and v , there is a path from u to v and a path from v to u . It's worth also formulating some terminology for the property at the heart of this definition; let's say that two nodes u and v in a directed graph are mutually reachable if there is a path from u to v and also a path from v to u .

So a graph is strongly connected if every pair of nodes is mutually reachable.

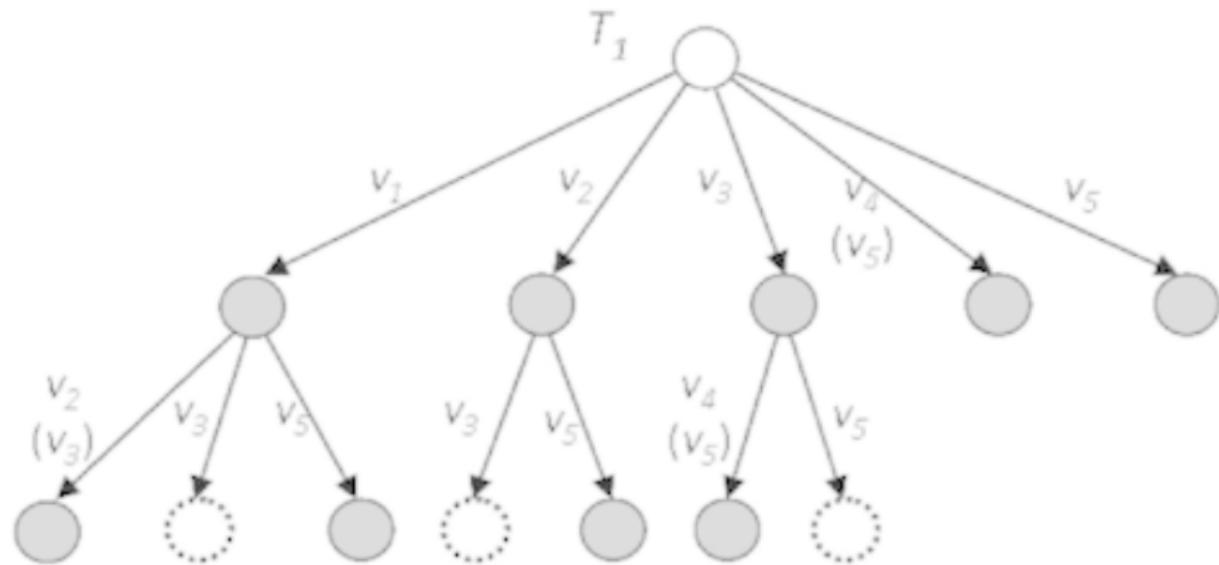


Strongly Connected

Checking Strong Connectivity using BFS

- We pick any node s and run BFS in G starting from s . We then also run BFS starting from s in Grev .
- Now, if one of these two searches fails to reach every node, then clearly G is not strongly connected. But suppose we find that s has a path to every node, and that every node has a path to s .

Practice Problems



Thank You

Implementation of Graphs: *L_{Practice}*

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Implementing a Graph

- Representation using Adjacency Matrix
- Representation using Adjacency List

2 Implementation of Search Algorithms

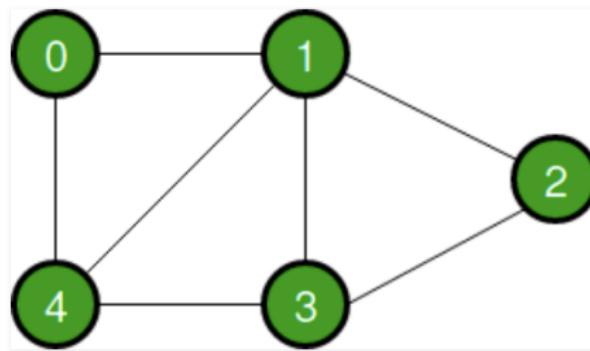
- BFS
- DFS

3 Applications of BFS

- Print Path
- Find the Connected Component by Starting at 's'
- Test The Bipartiteness of a Graph using BFS

4 Home Assignment

Implementing a Graph



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



Adjacency Matrix

- WAP to implement a Graph using Adjacency Matrix

Adjacency List

- WAP to implement a Graph using an Adjacency List

Implementing BFS

- Write a Program to traverse the vertices of a graph using BFS

Implementing BDFS

- Write a Program to traverse the vertices of a graph using DFS.

Print Path

- Write a Program to Print Path

All connected Component

- Write a Program to Find all the connected component of a given graph .

Test The Bipartiteness

- Write a Program to Test The Bipartiteness of a Graph using BFS

Home Assignments

- Write a Program to Test The Bipartiteness of a Graph using DFS.
- Write a Program to Find the connected component of a given graph starting at 's'.
- Write a Program to Delete the least element from a Max heap.

Thank You

DAG & Topological Sorting: L21 and L22

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

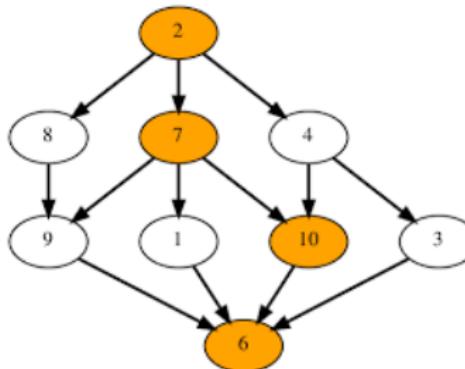
- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 DAG-Directed Acyclic Graphs

DAG-Directed Acyclic Graphs

- If an undirected graph has no cycles, then it has an extremely simple structure: each of its connected components is a tree.
But it is possible for a directed graph to have no (directed) cycles and still have a very rich structure.
- *If a directed graph has no cycles, we call it—naturally enough—a directed acyclic graph, or a DAG for short.*
- DAGs are a very common structure in computer science, because many kinds of dependency networks in CS.



Understanding DAG

Why DAG cannot have cycles

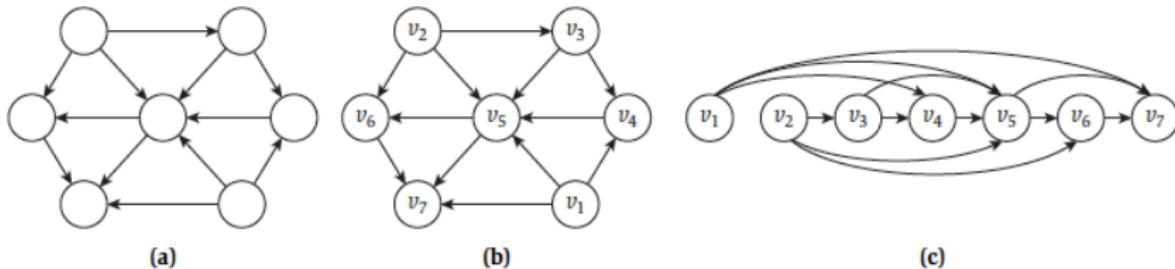
We can represent such an interdependent set of tasks by introducing a node for each task, and a directed edge (i, j) whenever i must be done before j . If the precedence relation is to be at all meaningful, the resulting graph G must be a DAG. Indeed, if it contained a cycle C , there would be no way to do any of the tasks in C : since each task in C cannot begin until some other one completes, no task in C could ever be done, since none could be done first.

Topological Ordering

Specifically, for a directed graph G , we say that a topological ordering of G is an ordering of its nodes as v_1, v_2, \dots, v_n so that for every edge (v_i, v_j) , we have $i < j$. In other words, all edges point “forward” in the ordering. A topological ordering on tasks provides an order in which they can be safely performed; when we come to the task v_j , all the tasks that are required to precede it have already been done.

If G has a topological ordering, then G is a DAG.

Why Topological Ordering



In a topological ordering, all edges point from left to right.

Figure 3.7 (a) A directed acyclic graph. (b) The same DAG with a topological ordering, specified by the labels on each node. (c) A different drawing of the same DAG, arranged so as to emphasize the topological ordering.

Problem: Given a Graph can you compute a Topological ordering?

- Does every DAG have a topological ordering, and if so, how do we find one efficiently?
- A method to do this for every DAG would be very useful: it would show that for any precedence relation on a set of tasks without cycles, there is an efficiently computable order in which to perform the tasks.

Design and Analysis of the Algorithm I

- The key to this lies in finding a way to get started: **which node do we put at the beginning of the topological ordering?**
 - Such a node v_1 would need to have no incoming edges, since any such incoming edge would violate the defining property of the topological ordering, that all edges point forward.
 - **In every DAG G, there is a node v with no incoming edges.[Proof is present in R1]**

Proof by Induction

In fact, the existence of such a node v is all we need to produce a topological ordering of G by induction. Specifically, let us claim by induction that every DAG has a topological ordering. This is clearly true for DAGs on one or two nodes. Now suppose it is true for DAGs with up to some number of nodes n . Then, given a DAG G on $n + 1$ nodes, we find a node v with no incoming edges, as guaranteed by above Theorem.

Design and Analysis of the Algorithm II

- Given a DAG G on $n + 1$ nodes, we place v first in the topological ordering; this is safe, since all edges out of v will point forward
- Now $G - v$ is a DAG, since deleting v cannot create any cycles that weren't there previously.
- Also, $G - v$ has n nodes, so we can apply the induction hypothesis to obtain a topological ordering of $G - v$. We append the nodes of $G - v$ in this order after v ; this is an ordering of G in which all edges point forward, and hence it is a topological ordering.

Pseudocode for Topological Ordering

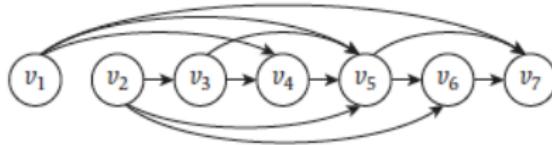
To compute a topological ordering of G :

Find a node v with no incoming edges and order it first

Delete v from G

Recursively compute a topological ordering of $G - \{v\}$
and append this order after v

In a topological ordering, all
edges point from left to right.



Example of Topological Ordering

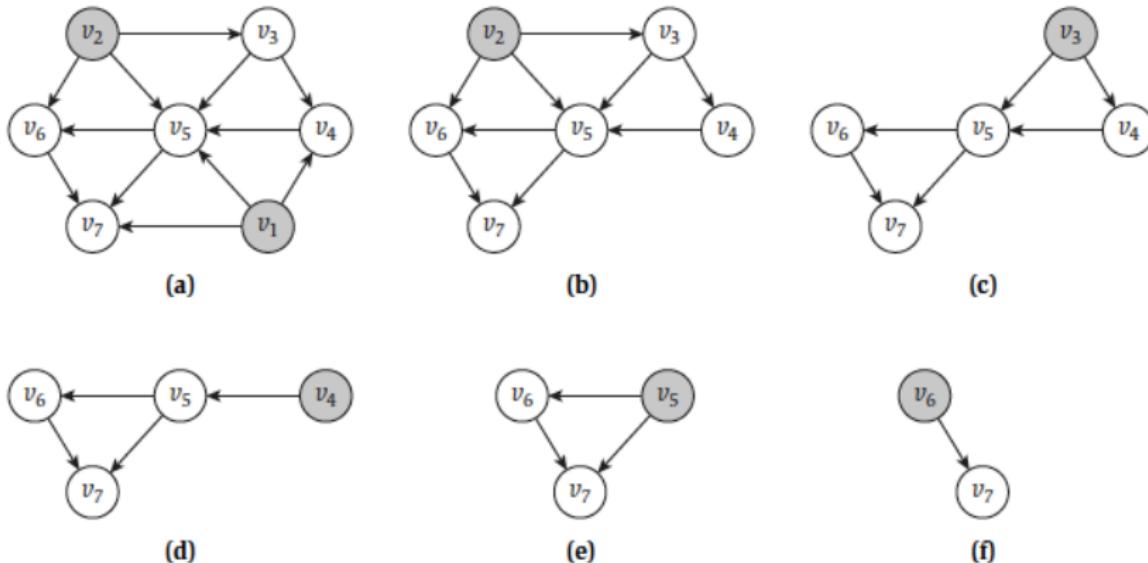


Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm's execution.

Running Time of Topological Ordering

To bound the running time of this algorithm, we note that identifying a node v with no incoming edges, and deleting it from G , can be done in $O(n)$ time. Since the algorithm runs for n iterations, the total running time is $O(n^2)$.

Topological Sorting

A topological sort of a dag $G = \langle V; E \rangle$ is a linear ordering of all its vertices such that if G contains an edge then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible.)

Example of Topological Sort

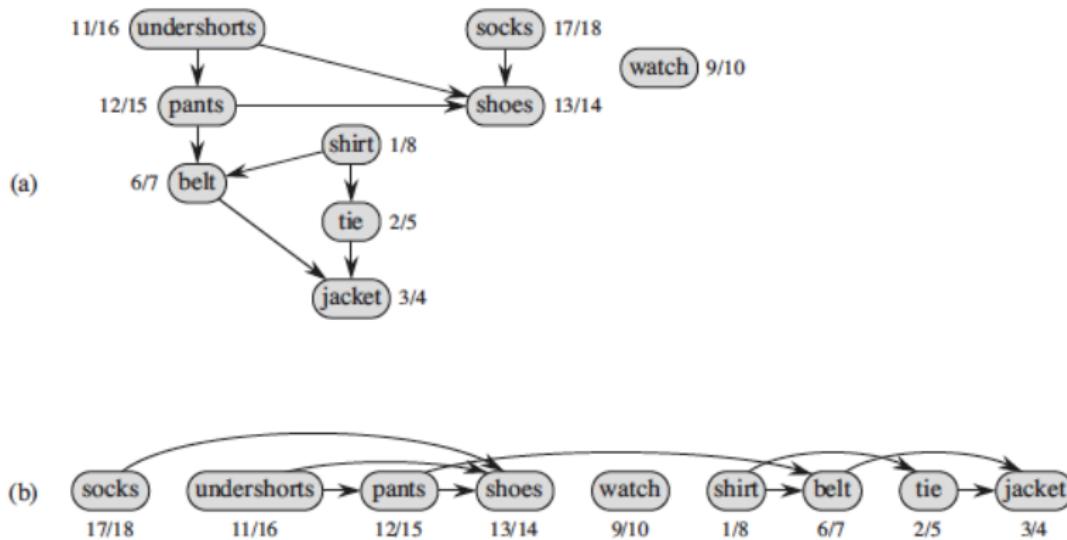


Figure 22.7 (a) Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v . The discovery and finishing times from a depth-first search are shown next to each vertex. (b) The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.

Alternate Algorithm for Topological Ordering or Topological Sorting

TOPOLOGICAL-SORT(G)

- 1 call DFS(G) to compute finishing times $v.f$ for each vertex v
- 2 as each vertex is finished, insert it onto the front of a linked list
- 3 return the linked list of vertices

Home Assignments

Exercises

1. Consider the directed acyclic graph G in Figure 3.10. How many topological orderings does it have?
2. Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

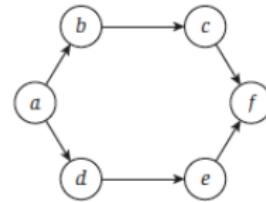


Figure 3.10 How many topological orderings does this graph have?

Thank You

MST using Kruskal's algorithm-the union-find data structure: L23

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

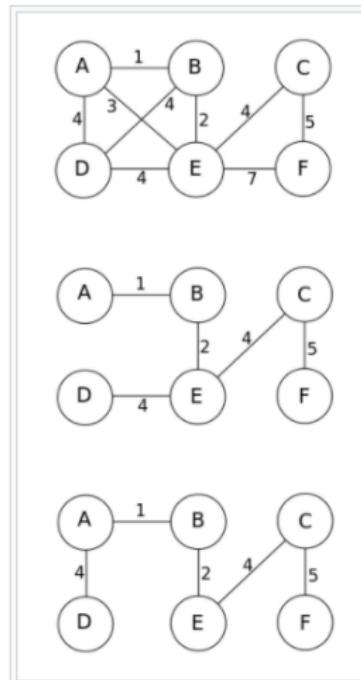
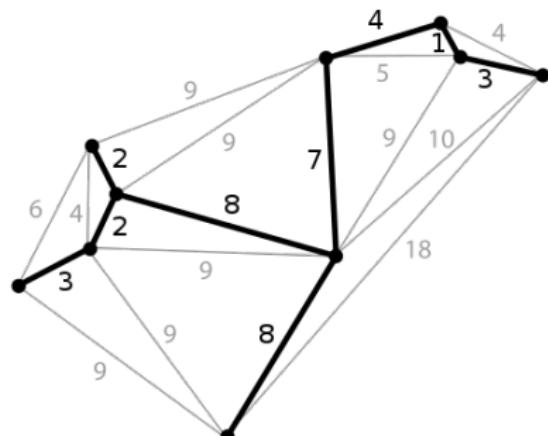
1 Minimum Spanning Tree

2 Union-find Datastructure

Minimum Spanning Tree

- A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a **connected, edge-weighted undirected graph** that connects all the vertices together, without any cycles and with the **minimum possible total edge weight**.
- Application-One example is a telecommunications company trying to lay cable in a new neighborhood. If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths. Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights.

Example of Minimum Spanning Tree



- There may be several minimum spanning trees of the same weight; in particular, if all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
- If there are n vertices in the graph, then each spanning tree has $n - 1$ edges.

Methods To Find Minimum Spanning Tree

- Prims Algorithm
- Krushkal's Algorithm

Union-find Datastructure

- In computer science, a disjoint-set data structure, also called a **union–find data structure** or **merge–find set**, is a **data structure that stores a collection of disjoint (non-overlapping) sets**.
- It provides operations for **adding new sets**, **merging sets (replacing them by their union)**, and **finding a representative member of a set**. The last operation allows to find out efficiently if any two elements are in the same or different sets.
- A disjoint-set data structure maintains a collection $S=\{S_1, S_2, \dots, S_k\}$ of disjoint dynamic sets. **We identify each set by a representative, which is some member of the set.**

Operations in Union-find Datstructure

MAKE-SET

MAKE-SET(x) creates a new set whose only member (and thus representative) is x . Since the sets are disjoint, we require that x not already be in some other set.

UNION

UNION(x,y) unites the dynamic sets that contain x and y , say S_x and S_y , into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of S_x [S_y], although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y , removing them from the collection S . In practice, we often absorb the elements of one of the sets into the other set.

FIND-SET

FIND-SET(x) returns a pointer to the representative of the (unique) set containing x .

Analysing Union-find Datstructure

we shall analyze the running times of disjoint-set datastructures in terms of two parameters:

- n , the number of MAKE-SET operations, and m , the total number of MAKE-SET, UNION,
- and FIND-SET operations.

Since the sets are disjoint, each UNION operation reduces the number of sets by one. After $n-1$ UNION operations, therefore, only one set remains. The number of UNION operations is thus at most $n-1$. Note also that since the MAKE-SET operations are included in the total number of operations m , we have $m > n$. We assume that the n MAKE-SET operations are the first n operations performed.

Make Set Pseudocode

```
function MakeSet(x) is
    if x is not already in the forest then
        x.parent := x
        x.size := 1      // if nodes store size
        x.rank := 0      // if nodes store rank
    end if
end function
```

This operation has constant time complexity. In particular, initializing a disjoint-set forest with n nodes requires $O(n)$ time.

Find Set Pseudocode

```
function Find(x) is
    if x.parent ≠ x then
        x.parent := Find(x.parent)
        return x.parent
    else
        return x
    end if
end function
```

Union Pseudocode

```
function Union(x, y) is
    // Replace nodes by roots
    x := Find(x)
    y := Find(y)

    if x = y then
        return // x and y are already in the same set
    end if

    // If necessary, rename variables to ensure that
    // x has at least as many descendants as y
    if x.size < y.size then
        (x, y) := (y, x)
    end if

    // Make x the new root
    y.parent := x
    // Update the size of x
    x.size := x.size + y.size
end function
```

Linked List Representation of Sets as Forests

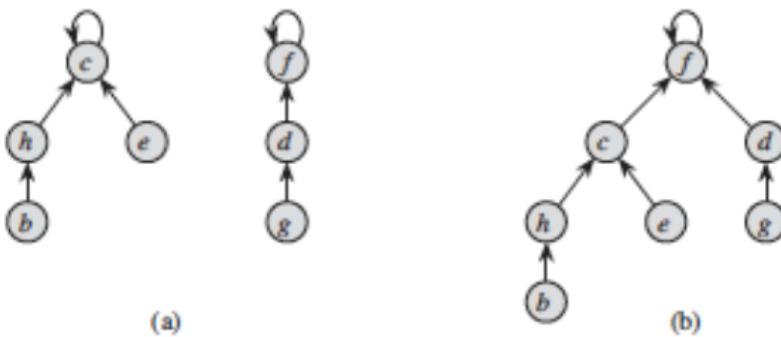


Figure 21.4 A disjoint-set forest. **(a)** Two trees representing the two sets of Figure 21.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of $\text{UNION}(e, g)$.

Example of Union Operation using Linked List

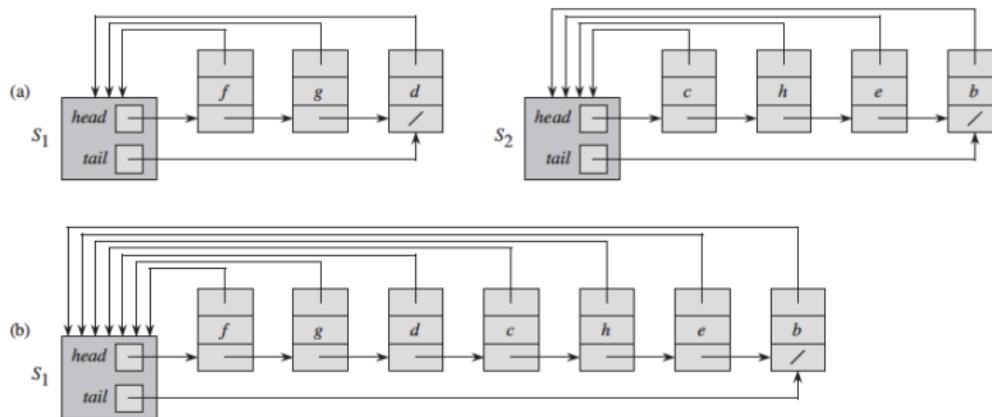


Figure 21.2 (a) Linked-list representations of two sets. Set S_1 contains members d , f , and g , with representative f , and set S_2 contains members b , c , e , and h , with representative c . Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers $head$ and $tail$ to the first and last objects, respectively. (b) The result of $\text{UNION}(g, e)$, which appends the linked list containing e to the linked list containing g . The representative of the resulting set is f . The set object for e 's list, S_2 , is destroyed.

Practice Problems

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic.

Thank You

MST using Kruskal's algorithm-the union-find data structure contd..: L24

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Growth of a Binary Tree

2 Minimum Spanning Tree: Krushkal's Algorithm

Growth of a Binary Tree I

- Given a **connected and undirected graph**, a spanning tree of that graph **is a subgraph that is a tree and connects all the vertices together**. A single graph can have many different spanning trees.
- A **minimum spanning tree (MST) or minimum weight spanning tree** for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. **The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.**
- Assume that we have a connected, undirected graph $G(V,E)$ with a weightfunction $w : e \rightarrow R$, and we wish to find a minimum spanning tree for G . The two algorithms we consider in this chapter use a **greedy approach** to the problem, although they differ in how they apply this approach.

Growth of a Binary Tree II

The Generic Approach

At each step, **we determine an edge (u,v) that we can add to A without violating this invariant**, in the **sense that $A \cup (u,v)$ is also a subset of a minimum spanning**

GENERIC-MST(G, w)

- 1 $A = \emptyset$
- 2 **while** A does not form a spanning tree
- 3 **find an edge (u, v) that is safe for A**
- 4 $A = A \cup \{(u, v)\}$
- 5 **return A**

We use the loop invariant as follows:

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

The tricky part is, of course, finding a safe edge in line 3. One must exist, since when line 3 is executed, the invariant dictates that there is a spanning tree T such that $A \subseteq T$. Within the **while** loop body, A must be a proper subset of T , and therefore there must be an edge $(u, v) \in T$ such that $(u, v) \notin A$ and (u, v) is safe for A .

Some Definitions I

Cut of a Graph

We first need some definitions. A cut $(S, V - S)$ of an undirected graph $G(V, E)$ is a partition of V .

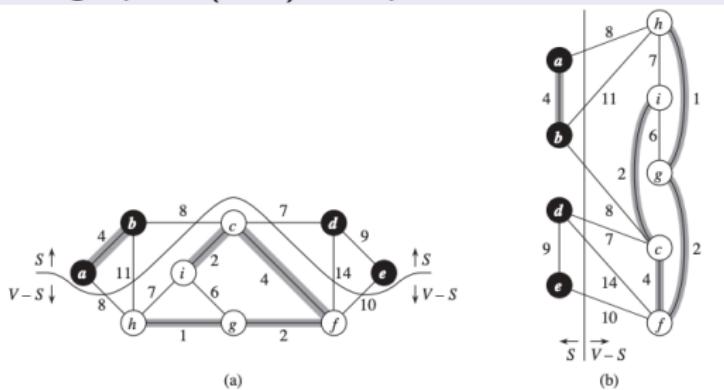


Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. (a) Black vertices are in the set S , and white vertices are in $V - S$. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V - S)$ respects A , since no edge of A crosses the cut. (b) The same graph with the vertices in the set S on the left and the vertices in the set $V - S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

Some Definitions II

- We say that an edge $(u, v) \in E$ crosses the cut $(S, V-S)$ if one of its endpoints is in S and the other is in $V-S$.
- We say that a cut respects a set A of edges if no edge in A crosses the cut.
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut. **Note that there can be more than one light edge crossing a cut in the case of ties.**
- More generally, we say that an edge is a light edge satisfying a given property if its weight is the minimum of any edge satisfying the property.

Minimum Spanning Tree: Krushkal's Algorithm

- The **two minimum-spanning-tree algorithms** described in this section elaborate on the generic method. **They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST.**
- In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. In Prim's algorithm, the set A forms a single tree.

Krushkal's Algorithm

- Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight.

MST-KRUSKAL(G, w)

- 1 $A = \emptyset$
- 2 **for** each vertex $v \in G.V$
3 MAKE-SET(v)
- 4 sort the edges of $G.E$ into nondecreasing order by weight w
- 5 **for** each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
- 6 **if** FIND-SET(u) \neq FIND-SET(v)
7 $A = A \cup \{(u, v)\}$
- 8 UNION(u, v)
- 9 **return** A

Example of Krushkal's Algorithm I

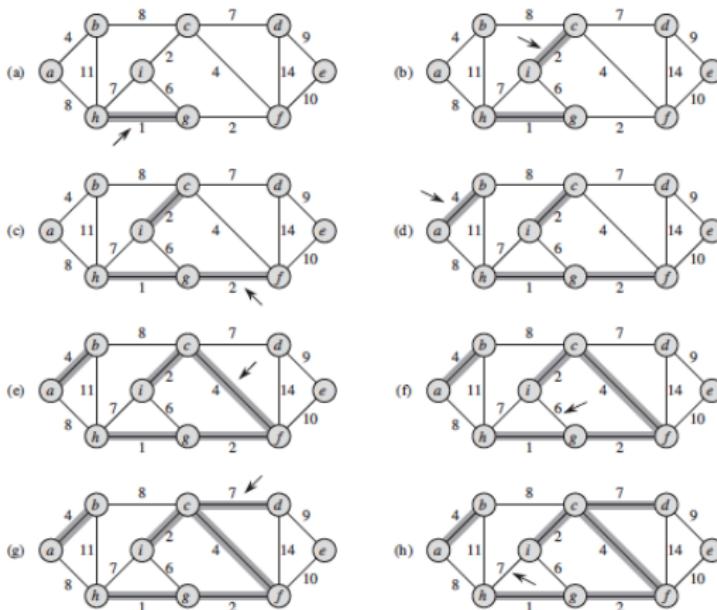


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Example of Krushkal's Algorithm II

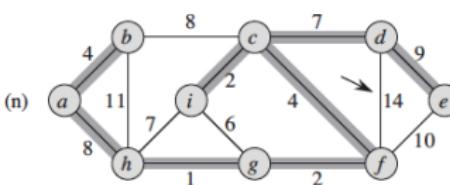
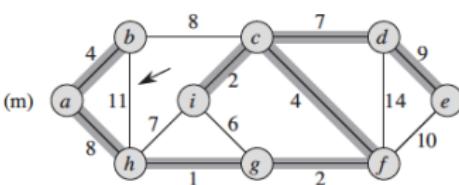
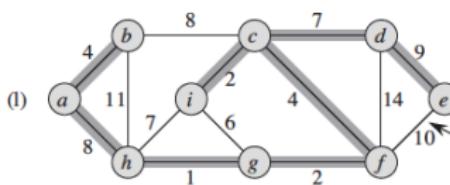
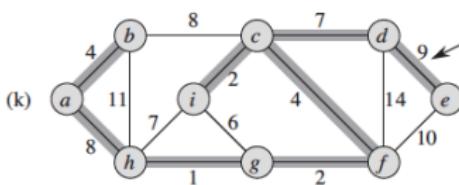
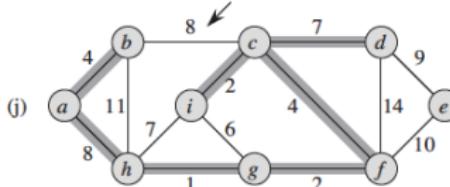
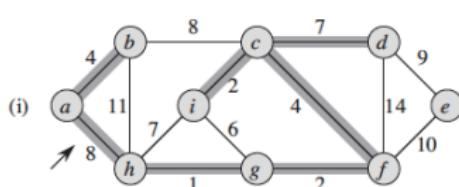


Figure 23.4, continued Further steps in the execution of Kruskal's algorithm.

Practice Problems

Suppose that we represent the graph $G=(V,E)$ as an adjacency matrix. Give a simple implementation of Krushkal's algorithm.

Thank You

MST using Prim's algorithm: L25

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1

Prim's Algorithm

Prim's Algorithm I

- Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we shall see in Section 24.3
- A Prim's algorithm has the property that the edges in the set A always form a single tree.**
- As Figure 23.5 shows, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V . Each step adds to the tree A a light edge that connects A to an isolated vertex—one on which no edge of A is incident.
- In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A . In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm.

Prim's Algorithm II

- During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.
- For each vertex v , the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $v.key = \infty$ if there is no such edge.
- The attribute $v.\pi$ names the parent of v in the tree

Prim's Algorithm

When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus

$$A = \{(v, v.\pi) : v \in V - \{r\}\}.$$

MST-PRIM(G, w, r)

```
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $v.key = w(u, v)$ 
```

Example of Prim's Algorithm I

Example of Prim's Algorithm II

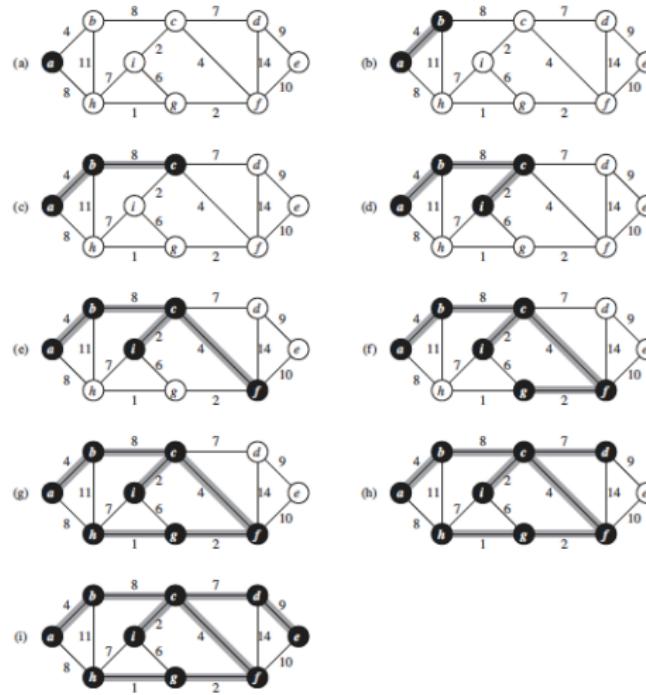
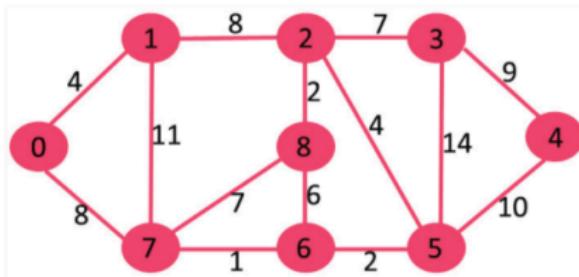


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a . Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Illustration I

Let us understand with the following example:



The set $mstSet$ is initially empty and keys assigned to vertices are $\{0, \text{INF}, \text{INF}, \text{INF}, \text{INF}, \text{INF}, \text{INF}, \text{INF}, \text{INF}\}$ where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in $mstSet$. So $mstSet$ becomes $\{0\}$. After including to $mstSet$, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

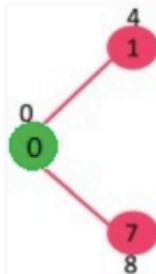
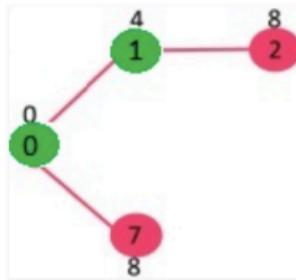


Illustration II

Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).

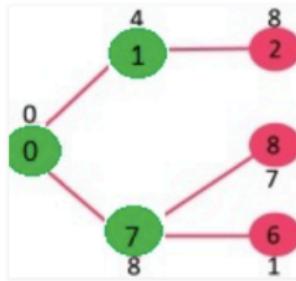
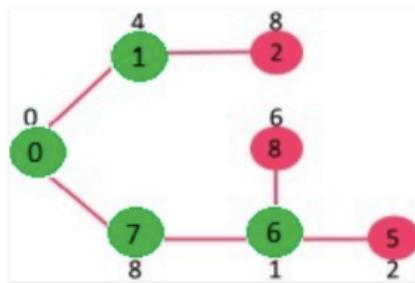
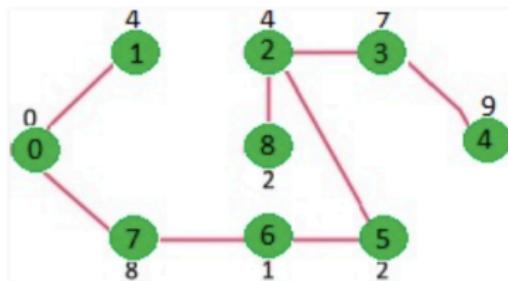


Illustration III

Pick the vertex with minimum key value and not already included in MST (not in *mstSet*). Vertex 6 is picked. So *mstSet* now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



Running Time of Krushkal's Algorithm

Both **Prim's and Kruskal's algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few major differences between them.

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to $O(E + \log V)$ using Fibonacci heaps.	Kruskal's algorithm's time complexity is $O(E \log V)$, V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

Clustering I

Clustering

Clustering arises whenever one has a collection of objects—say, a set of photographs, documents, or microorganisms—that one is trying to classify or organize into coherent groups. Faced with such a situation, it is natural to look first for measures of how similar or dissimilar each pair of objects is.

Clustering II

Spacing

Suppose we are seeking to divide the objects in U into k groups, for a given parameter k . We say that a k -clustering of U is a partition of U into k nonempty sets C_1, C_2, \dots, C_k . We define the spacing of a k -clustering to be the minimum distance between any pair of points lying in different clusters. Given that we want points in different clusters to be far apart from one another, a natural goal is to seek the k -clustering with the maximum possible spacing.

The Problem

The question now becomes the following. There are exponentially many different k -clusterings of a set U ; how can we efficiently find the one that has maximum spacing?

Clustering III

The Solution

We are doing exactly what Kruskal's Algorithm would do if given a graph G on U in which there was an edge of cost $d(p_i, p_j)$ between each pair of nodes (p_i, p_j) . The only difference is that we seek a k -clustering, so we stop the procedure once we obtain k connected components.

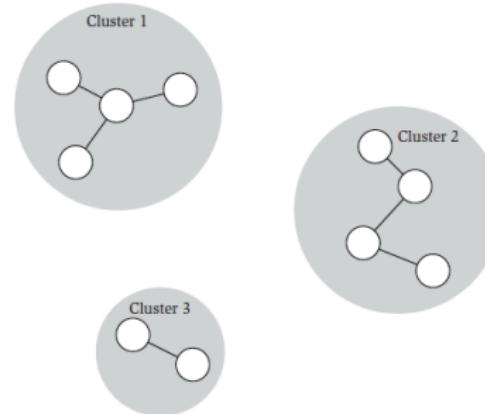


Figure 4.14 An example of single-linkage clustering with $k = 3$ clusters. The clusters are formed by adding edges between points in order of increasing distance.

(4.26) *The components C_1, C_2, \dots, C_k formed by deleting the $k - 1$ most expensive edges of the minimum spanning tree T constitute a k -clustering of maximum spacing.*



Practice Problems

Suppose that we represent the graph $G=(V,E)$ as an adjacency matrix. Give a simple implementation of prim's algorithm.

Thank You

Practice Questions: L26

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Implementation of Prim's, Krushkal's, Strongly Connected Component

2 Practice Questions

MST

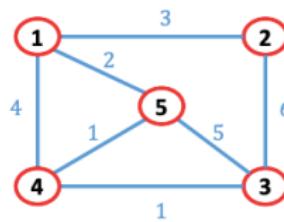
- Implement the Prim's Algorithm for Adjacency Matrix Representation.
- Implement the Krushkal's Algorithm for Adjacency Matrix Representation.
- Find the Strongly Connected components of a Graph

Practice Questions I

- Design an abstract data type Graph to represent a data structure for an undirected graph. Vertices of the graph are uniquely identified using an integer ID field, and a templated value field. The edges of the graph should have a value field that is templated to accept any class, and the edge is uniquely identified by the two vertex IDs it is incident upon. The graph ADT should allow you to add, remove and list vertices and edges. The edges incident on a vertex should be maintained as an adjacency list. The ADT should allow you to traverse to incident edges and adjacent vertices from a given vertex. You should use C++ STL where possible.
- Using the graph ADT you have defined above, implement the following graph algorithms:

Practice Questions II

- Depth First Search (DFS): The method `dfs(Graph g, int vid)` should be implemented to traverse the graph `g` in a depth-first manner from the source vertex identified by `vid`. At each step of the DFS you should pick the outgoing edge to the adjacent non-visited vertex with the smallest ID to traverse on. The DFS traversal should print the IDs of the vertices in the order traversed, on a single line separated by a single space between the IDs, and prefixed with the string `DFS`. E.g. the DFS traversal of the graph in Fig. 1 from a source vertex ID 5 should produce the line: `DFS 5`



1 2 3 4 terminated by a newline

Figure 1: Sample Graph

Practice Questions III

Single Source Shortest Path (SSSP): The method `sssp(Graph g, int vid)` should take the identifier `vid` of a source vertex and implement Dijkstra's shortest path algorithm to find the shortest distances from the source vertex to every vertex in the graph `g`, including the source. Assume that the edge weights are integers for the templated edge class. For each vertex in the connected component that the source vertex is present in, the SSSP method call should print one line of output with the `vid` distance, preceded by a separate line having the string `SSSP`, e.g., running SSSP on the graph in Fig. 1 from a source vertex ID 5 should produce:

SSSP

5 0

4 1

1 2

3 2

2 5

The order of the lines does not matter.

Practice Questions IV

K-means Clustering: The method `kmeans(Graph g, int k, int maxcuts)` should take the number of clusters k to be identified in the graph g , and the maximum of the sum of the undirected edge cuts that are allowed between clusters, maxcuts . The algorithm should pick k vertices as centers at random and color neighbors using a uniform BFS traversal initiated from each center until all vertices in the connected component are colored with one of the centers. Ignore the weights of the edges during traversal. Sum the number of edge cuts between vertices with different colors, and if strictly greater than the maxcuts threshold, repeat the process with a new set of k random center vertices. Else, if less than equal to the threshold, stop and print the clusters as output. Print each cluster in a single line as a list of vertex IDs separated by space followed by the number of cuts in a separate line. These should be preceded by a separate line having the string `KMEANS`. E.g., with $k = 3$ for a given graph having 10 vertices, you will print three lines having the vertex IDs of each cluster as follows, and a separate line with the sum of the number of edge cuts between clusters, 4:

Practice Questions V

- Given a tree represented as undirected graph. Count the number of nodes at given level l . It may be assumed that vertex 0 is root of the tree.
- Count number of trees in a forest
- In the case of disconnected graph or any vertex that is unreachable from all vertex, the previous implementation of BFS that you have done will not give the desired output, so propose a modification to BFS.
- Detect cycle in a direct graph using colors
- Clone a Directed Acyclic Graph.
- All topological sorts of a Directed Acyclic Graph

Thank You

Shortest path problem (Dijkstra algorithm):

L27

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Single Source Shortest Path

Introduction

Shortest Path Problem

In this chapter and in Chapter 25, we show how to solve such problems efficiently. In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow \mathbb{R}$ mapping edges to real-valued weights. The **weight** $w(p)$ of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i).$$

We define the **shortest-path weight** $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

A **shortest path** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

In the Phoenix-to-Indianapolis example, we can model the road map as a graph: vertices represent intersections, edges represent road segments between intersections, and edge weights represent road distances. Our goal is to find a shortest path from a given intersection in Phoenix to a given intersection in Indianapolis.

Variants of shortest Path Algorithm

In this chapter, we shall focus on the single-source shortest-paths problem: given a graph $G = \langle V, E \rangle$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$. The algorithm for the single-source problem can solve many other problems, including the following variants.

- **Single-source shortest-paths problem:** In this chapter, we shall focus on the single-source shortest-paths problem: given a graph $G(V, E)$, we want to find a shortest path from a given source vertex s to each vertex $v \in V$. The algorithm for the single-source problem can solve many other problems, including the following variants.
- **Single-destination shortest-paths problem:** Find a shortest path to a given destination vertex t from each vertex. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.
- **Single-pair shortest-path problem:** Find a shortest path from u to v for given vertices u and v . If we solve the single-source problem with source vertex u , we solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.
- **All-pairs shortest-paths problem:** Find a shortest path from u to v for every pair of vertices u and v . Although we can solve this problem by running a single-source algorithm once from each vertex, we usually can solve it faster. Additionally, its structure is interesting in its own right. Chapter 25 addresses the all-pairs problem in detail.

Optimal Substructure of Shortest Path

- **Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.**
- **Dijkstra's algorithm**, is a **greedy algorithm**, and the Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices, is a dynamic-programming algorithm.

Lemma 24.1 (Subpaths of shortest paths are shortest paths)

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$, let $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j . Then, p_{ij} is a shortest path from v_i to v_j .

Proof If we decompose path p into $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k . ■

Negative weight Edges

- Some instances of the single-source shortest-paths problem may include edges whose weights are negative.
- If the graph $G \in D, V, E$ contains no negative-weight cycles reachable from the source s , then for all $v \in V$, the shortest-path weight remains well defined, even if it has a negative value.
- Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative, as in the road-map example.

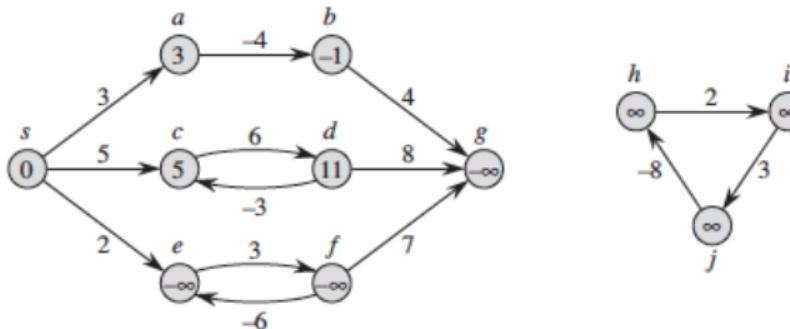


Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s , they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as h , i , and j are not reachable from s , and so their shortest-path weights are ∞ , even though they lie on a negative-weight cycle.

Can a shortest path contain cycles?

As we have just seen, it cannot contain a negative-weight cycle. Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.

That leaves only 0-weight cycles. We can remove a 0-weight cycle from any path to produce another path whose weight is the same. Thus, if there is a shortest path from a source vertex s to a destination vertex that contains a 0-weight cycle, then there is another shortest path from s to without this cycle. **As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free.**

Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles, i.e., they are simple paths.

Representation of Shortest Path

- We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well. We represent shortest paths similarly to how we represented breadth-first trees in Section 22.2.
- Given a graph $G(V, E)$, we maintain for each vertex $v \in V$ a predecessor $v.\pi$ that is either another vertex or NIL. The shortest-paths algorithms in this chapter set the attributes so that the chain of predecessors originating at a vertex runs backwards along a shortest path from s to $.$

defined. A *shortest-paths tree* rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G ,
2. G' forms a rooted tree with root s , and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G .

Representation of Shortest Path

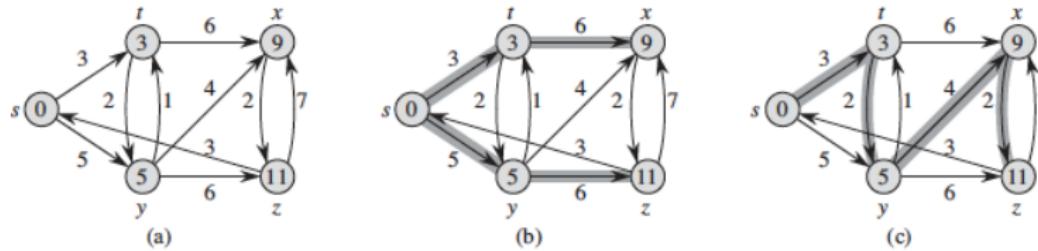


Figure 24.2 (a) A weighted, directed graph with shortest-path weights from source s . (b) The shaded edges form a shortest-paths tree rooted at the source s . (c) Another shortest-paths tree with the same root.

Initialize Single Source

The algorithms in this chapter use the technique of relaxation. For each vertex $v \in V$, we maintain an attribute $v.d$ which is an upper bound on the weight of a shortest path from source s to vertex v . We call $v.d$ a shortest-path estimate. We **initialize** the shortest-path estimates and predecessors by the following $\theta(V)$ time procedure:

INITIALIZE-SINGLE-SOURCE(G, s)

- 1 for each vertex $v \in G.V$
- 2 $v.d = \infty$
- 3 $v.\pi = \text{NIL}$
- 4 $s.d = 0$

After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$, and $v.d = \infty$ for $v \in V - \{s\}$.

The process of *relaxing* an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$. A relaxation step¹ may decrease the value of the shortest-path

Relaxation

RELAX(u, v, w)

- 1 **if** $v.d > u.d + w(u, v)$
- 2 $v.d = u.d + w(u, v)$
- 3 $v.\pi = u$

Figure 24.3 shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

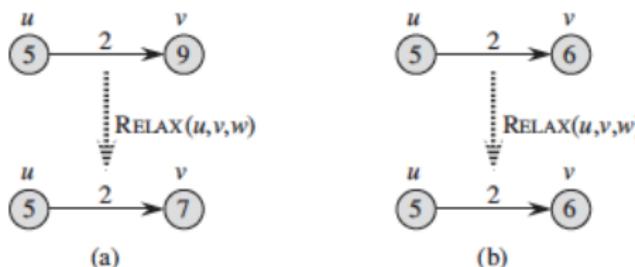


Figure 24.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. (a) Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. (b) Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation step leaves $v.d$ unchanged.

Properties of Shortest Path and Relaxation

Triangle inequality (Lemma 24.10)

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)

We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)

If there is no path from s to v , then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v) , then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)

If $p = \langle v_0, v_1, \dots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k , and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p .

Predecessor-subgraph property (Lemma 24.17)

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s .

Dijkstra's Algorithm I

- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G(V,E)$ for the case in which all edge weights are non-negative.
- In this section, therefore, we assume that $w(u,v) > 0$ for each edge $(u,v) \in E$.
- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
- The algorithm repeatedly selects the vertex $u \in V \setminus S$ with the minimum shortest-path estimate, adds u to S , and relaxes all edges leaving u . In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values

Dijkstra's Algorithm II

DIJKSTRA(G, w, s)

- 1 **INITIALIZE-SINGLE-SOURCE**(G, s)
- 2 $S = \emptyset$
- 3 $Q = G.V$
- 4 **while** $Q \neq \emptyset$
 - 5 $u = \text{EXTRACT-MIN}(Q)$
 - 6 $S = S \cup \{u\}$
 - 7 **for each** vertex $v \in G.Adj[u]$
 - 8 **RELAX**(u, v, w)

Dijkstra's Algorithm III

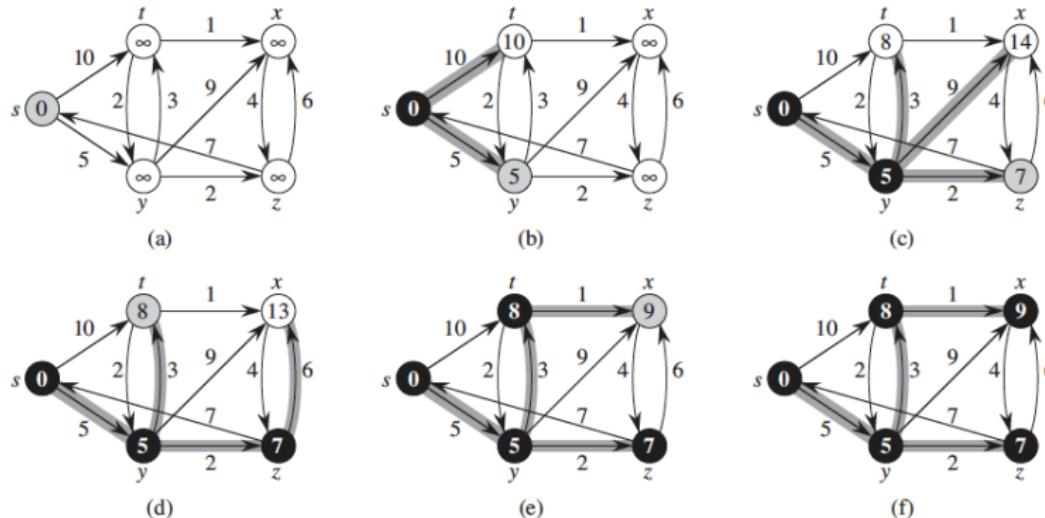


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S , and white vertices are in the min-priority queue $Q = V - S$. (a) The situation just before the first iteration of the while loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. (b)–(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

Correctness of Dijkstra's Algorithm I

Dijkstra's algorithm, run on a weighted, directed graph $G(V,E)$ with non-negative weight function w and source s , terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Correctness of Dijkstra's Algorithm II

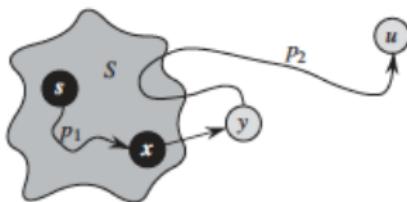


Figure 24.7 The proof of Theorem 24.6. Set S is nonempty just before vertex u is added to it. We decompose a shortest path p from source s to vertex u into $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$, where y is the first vertex on the path that is not in S and $x \in S$ immediately precedes y . Vertices x and y are distinct, but we may have $s = x$ or $y = u$. Path p_2 may or may not reenter set S .

Proof We use the following loop invariant:

At the start of each iteration of the while loop of lines 4–8, $v.d = \delta(s, v)$ for each vertex $v \in S$.

It suffices to show for each vertex $u \in V$, we have $u.d = \delta(s, u)$ at the time when u is added to set S . Once we show that $u.d = \delta(s, u)$, we rely on the upper-bound property to show that the equality holds at all times thereafter.

Initialization: Initially, $S = \emptyset$, and so the invariant is trivially true.

Correctness of Dijkstra's Algorithm III

Maintenance: We wish to show that in each iteration, $u.d = \delta(s, u)$ for the vertex added to set S . For the purpose of contradiction, let u be the first vertex for which $u.d \neq \delta(s, u)$ when it is added to set S . We shall focus our attention on the situation at the beginning of the iteration of the while loop in which u is added to S and derive the contradiction that $u.d = \delta(s, u)$ at that time by examining a shortest path from s to u . We must have $u \neq s$ because s is the first vertex added to set S and $s.d = \delta(s, s) = 0$ at that time. Because $u \neq s$, we also have that $S \neq \emptyset$ just before u is added to S . There must be some path from s to u , for otherwise $u.d = \delta(s, u) = \infty$ by the no-path property, which would violate our assumption that $u.d \neq \delta(s, u)$. Because there is at least one path, there is a shortest path p from s to u . Prior to adding u to S , path p connects a vertex in S , namely s , to a vertex in $V - S$, namely u . Let us consider the first vertex y along p such that $y \in V - S$, and let $x \in S$ be y 's predecessor along p . Thus, as Figure 24.7 illustrates, we can decompose path p into $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$. (Either of paths p_1 or p_2 may have no edges.)

We claim that $y.d = \delta(s, y)$ when u is added to S . To prove this claim, observe that $x \in S$. Then, because we chose u as the first vertex for which $u.d \neq \delta(s, u)$ when it is added to S , we had $x.d = \delta(s, x)$ when x was added

Correctness of Dijkstra's Algorithm IV

to S . Edge (x, y) was relaxed at that time, and the claim follows from the convergence property.

We can now obtain a contradiction to prove that $u.d = \delta(s, u)$. Because y appears before u on a shortest path from s to u and all edge weights are non-negative (notably those on path p_2), we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \quad (\text{by the upper-bound property}) . \end{aligned} \tag{24.2}$$

But because both vertices u and y were in $V - S$ when u was chosen in line 5, we have $u.d \leq y.d$. Thus, the two inequalities in (24.2) are in fact equalities, giving

$$y.d = \delta(s, y) = \delta(s, u) = u.d .$$

Consequently, $u.d = \delta(s, u)$, which contradicts our choice of u . We conclude that $u.d = \delta(s, u)$ when u is added to S , and that this equality is maintained at all times thereafter.

Termination: At termination, $Q = \emptyset$ which, along with our earlier invariant that $Q = V - S$, implies that $S = V$. Thus, $u.d = \delta(s, u)$ for all vertices $u \in V$. ■



Practice Questions

Run Dijkstra's algorithm on the directed graph of Figure 24.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 24.6, show the d and values and the vertices in set S after each iteration of the while loop.

Thank You

Interval Scheduling: The Greedy Algorithm

Stays Ahead: L28

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Interval Scheduling Problem

Interval Scheduling Problem I

- More formally, there will be n requests labeled $1, \dots, n$ with each request i specifying a start time s_i and a finish time f_i . Naturally, we have $s_i < f_i$ for all i .
- Two requests i and j are compatible if the requested intervals do not overlap: that is, either request i is for an earlier time interval than request j ($f_i \leq s_j$), or request i is for a later time than request j ($f_j \leq s_i$).
- We'll say more generally that **a subset A of requests is compatible** if all pairs of requests $i, j \in A, i \neq j$ are compatible. The goal is to select a compatible subset of requests of maximum possible size.

Interval Scheduling Problem II

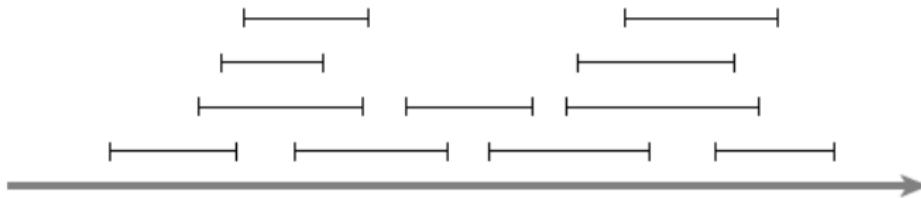


Figure 1.4 An instance of the Interval Scheduling Problem.

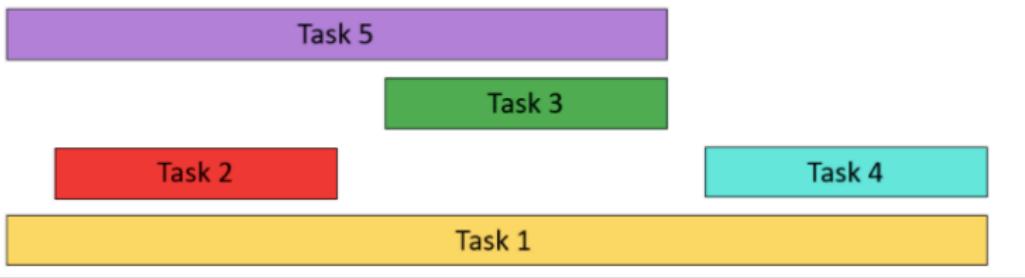
Interval Scheduling Problem III

Application of Interval Scheduling

- Consider the following very simple scheduling problem. You have a resource—it may be a lecture room, a supercomputer, or an electron microscope—and many people request to use the resource for periods of time.
- A request takes the form: **Can I reserve the resource starting at time s, until time f ? We will assume that the resource can be used by at most one person at a time.**
- A scheduler wants to accept a subset of these requests, rejecting all others, so that the accepted requests do not overlap in time. The goal is to maximize the number of requests accepted.

Interval Scheduling Problem IV

Input:



Output: [Task 2, Task 3, Task 4] is an optimal solution because these tasks have no conflicts with each other and any set with 4 tasks will have at least two intervals in conflict.

Designing a Greedy Algorithm

Step-1: The basic idea in a greedy algorithm for interval scheduling is to use a simple rule to select a first request i_1 . Once a request i_1 is accepted, we reject all requests that are not compatible with i_1 .

Step-2: We then select the next request i_2 to be accepted, and again reject all requests that are not compatible with i_2 .

Step-3: We continue in this fashion until we run out of requests.

Natural Selection Parameters that fail!

The challenge in designing a good greedy algorithm is in deciding which simple rule to use for the selection—and there are many natural rules for this problem that do not give good solutions:

- **The most obvious rule might be to always select the available request that starts earliest**

This method does not yield an optimal solution. If the earliest request i is for a very long interval, then by accepting request i we may have to reject a lot of requests for shorter time intervals.

- **This might suggest that we should start out by accepting the request that requires the smallest interval of time—namely, the request for which $f(i) - s(i)$ is as small as possible.**

As it turns out, this is a somewhat better rule than the previous one, but it still can produce a suboptimal schedule.

- for each request, **we count the number of other requests that are not compatible**, and accept the request that has the fewest number of noncompatible requests. This also fails

Counterexample for smallest interval algorithm

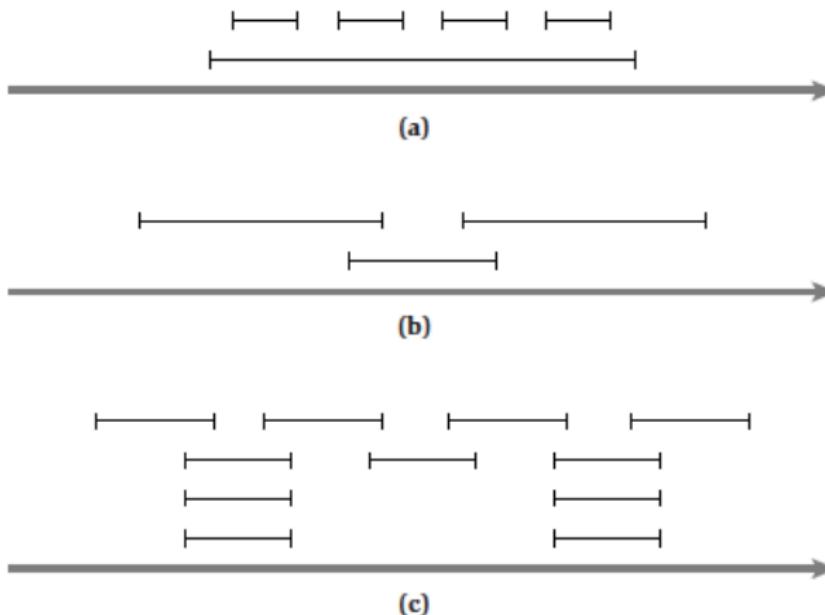


Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

An Optimal greedy rule

A greedy rule that does lead to the optimal solution is based on a fourth idea: we should accept first the request that finishes first, that is, the request i for which $f(i)$ is as small as possible. we ensure that our resource becomes free as soon as possible while still satisfying one request.

```
Initially let  $R$  be the set of all requests, and let  $A$  be empty
While  $R$  is not yet empty
    Choose a request  $i \in R$  that has the smallest finishing time
    Add request  $i$  to  $A$ 
    Delete all requests from  $R$  that are not compatible with request  $i$ 
EndWhile
Return the set  $A$  as the set of accepted requests
```

Example of the Greedy Approach

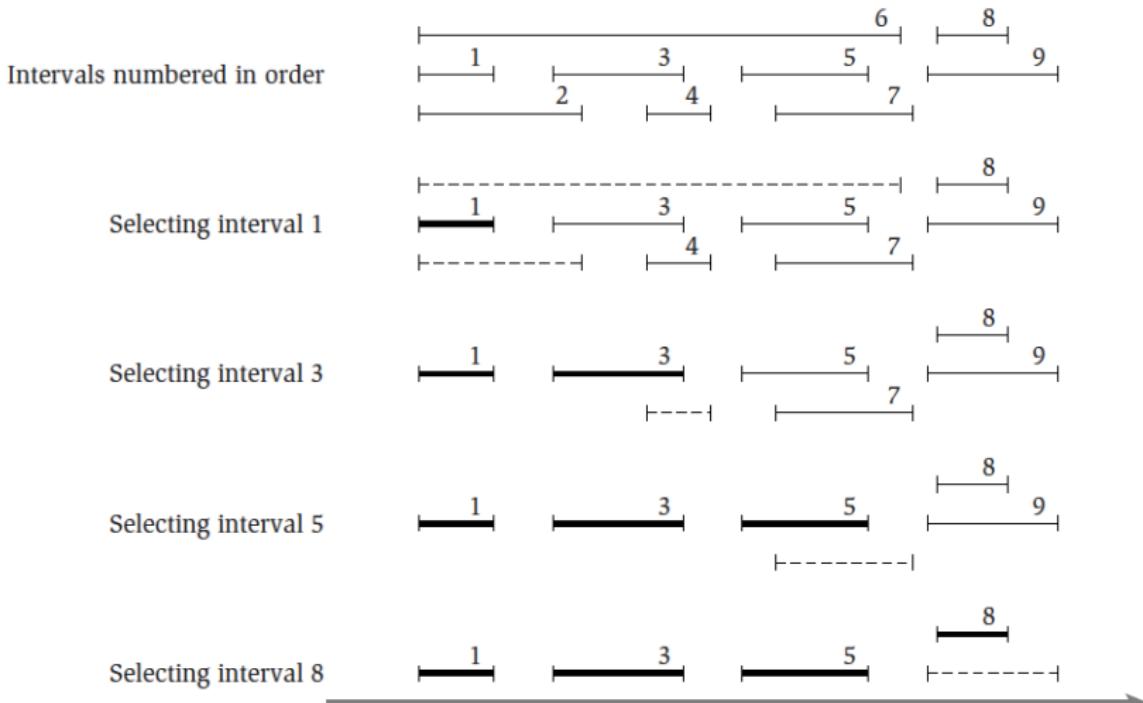


Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

Analysis of the Algorithm

As a start, we can immediately declare that the intervals in the set A returned by the algorithm are all compatible.

So, for purposes of comparison, let O be an optimal set of intervals. Ideally one might want to show that $A = O$, but this is too much to ask: there may be many optimal solutions, and at best A is equal to a single one of them.

We introduce some notation to help with this proof. Let i_1, \dots, i_k be the set of requests in A in the order they were added to A. Note that $|A| = k$. Similarly, let the set of requests in O be denoted by j_1, \dots, j_m . Our goal is to prove that $k = m$.

Assume that the requests in O are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points. Note that the requests in O are compatible, which implies that the start points have the same order as the finish points. Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request. And indeed, our greedy rule guarantees that $f(i_1) < f(j_1)$.

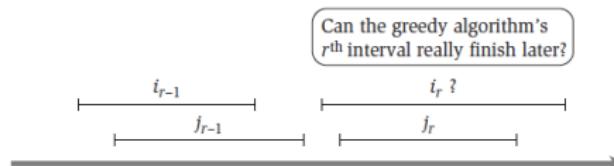


Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

Running Time of the Greedy Algorithm

Implementation and Running Time We can make our algorithm run in time $O(n \log n)$ as follows. We begin by sorting the n requests in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time $O(n \log n)$. In an additional $O(n)$ time, we construct an array $S[1 \dots n]$ with the property that $S[i]$ contains the value $s(i)$.

We now select requests by processing the intervals in order of increasing $f(i)$. We always select the first interval; we then iterate through the intervals in order until reaching the first interval j for which $s(j) \geq f(1)$; we then select this one as well. More generally, if the most recent interval we've selected ends at time f , we continue iterating through subsequent intervals until we reach the first j for which $s(j) \geq f$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus this part of the algorithm takes time $O(n)$.

Practice Questions

Implement the Interval Scheduling Greedy Algorithm

Thank You

Scheduling to Minimize Lateness: An Exchange Argument: L29

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Interval Scheduling Problem: Extensions

Extensions to Interval Scheduling I

- Type-I In defining the problem, **we assumed that all requests were known to the scheduling algorithm** when it was choosing the compatible subset. It would also be natural, of course, to think about the version of the problem in which the scheduler needs to make decisions about accepting or rejecting certain requests before knowing about the full set of requests. **Online Algorithms**
- Type-II Our goal was to maximize the number of satisfied requests. But we could **picture a situation in which each request has a different value to us**. For example, each request i could also have a value v_i (**the amount gained by satisfying request i**), and the goal would be to **maximize our income**: the sum of the values of all satisfied requests.

A Related Problem: Scheduling All Intervals I

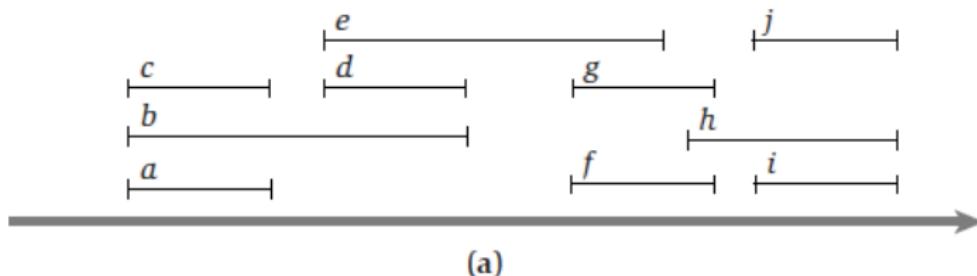
The Problem

In the Interval Scheduling Problem, there is a single resource and many requests in the form of time intervals, so we must choose which requests to accept and which to reject. A related problem arises if we have many identical resources available and we wish to schedule all the requests using as few resources as possible.

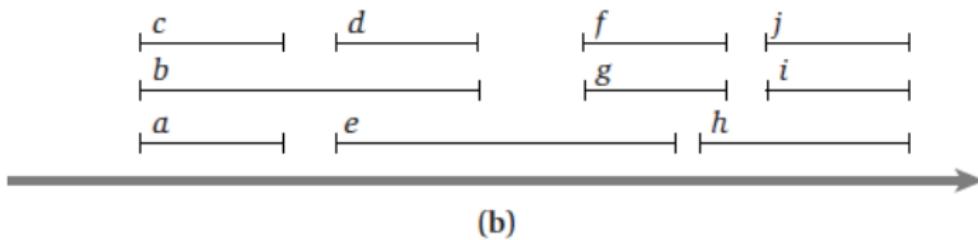
Because the goal here is to partition all intervals across multiple resources, we will refer to this as the Interval Partitioning Problem.

In general, one can imagine a solution using k resources as a rearrangement of the requests into k rows of nonoverlapping intervals: the first row contains all the intervals assigned to the first resource, the second row contains all those assigned to the second resource, and so forth.

A Related Problem: Scheduling All Intervals II



(a)



(b)

Figure 4.4 (a) An instance of the Interval Partitioning Problem with ten intervals (*a* through *j*). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.

Minimum number of resources: Lower Bound

Now, **is there any hope of using just two resources in this sample instance?**

Clearly the answer is no. **We need at least three resources since, for example, intervals a, b, and c all pass over a common point on the time-line, and hence they all need to be scheduled on different resources.**

(4.4) *In any instance of Interval Partitioning, the number of resources needed is at least the depth of the set of intervals.*

Proof. Suppose a set of intervals has depth d , and let I_1, \dots, I_d all pass over a common point on the time-line. Then each of these intervals must be scheduled on a different resource, so the whole instance needs at least d resources. ■

Relation between Interval Scheduling and Scheduling All Intervals

- First, can we design an efficient algorithm that schedules all intervals using the minimum possible number of resources?
- Second, is there always a **schedule using a number of resources that is equal to the depth?**

Algorithm for Scheduling All Intervals I

Designing the Algorithm

Let d be the depth of the set of intervals; we show how to assign a label to each interval, where the labels come from the set of numbers $1, 2, \dots, d$, and the assignment has the property that overlapping intervals are labeled with different numbers. This gives the desired solution.

Understanding the Algorithm

The algorithm we use for this is a simple one-pass greedy strategy that orders intervals by their starting times. We go through the intervals in this order, and try to assign to each interval we encounter a label that hasn't already been assigned to any previous interval that overlaps it.

Algorithm for Scheduling All Intervals II

```
Sort the intervals by their start times, breaking ties arbitrarily
Let  $I_1, I_2, \dots, I_n$  denote the intervals in this order
For  $j = 1, 2, 3, \dots, n$ 
    For each interval  $I_i$  that precedes  $I_j$  in sorted order and overlaps it
        Exclude the label of  $I_i$  from consideration for  $I_j$ 
    Endfor
    If there is any label from  $\{1, 2, \dots, d\}$  that has not been excluded then
        Assign a nonexcluded label to  $I_j$ 
    Else
        Leave  $I_j$  unlabeled
    Endif
Endfor
```

Analysis of the Algorithm

Claim 1

(4.5) If we use the greedy algorithm above, every interval will be assigned a label, and no two overlapping intervals will receive the same label.

Proof. First let's argue that no interval ends up unlabeled. Consider one of the intervals I_j , and suppose there are t intervals earlier in the sorted order that overlap it. These t intervals, together with I_j , form a set of $t + 1$ intervals that all pass over a common point on the time-line (namely, the start time of I_j), and so $t + 1 \leq d$. Thus $t \leq d - 1$. It follows that at least one of the d labels is not excluded by this set of t intervals, and so there is a label that can be assigned to I_j .

Next we claim that no two overlapping intervals are assigned the same label. Indeed, consider any two intervals I and I' that overlap, and suppose I precedes I' in the sorted order. Then when I' is considered by the algorithm, I is in the set of intervals whose labels are excluded from consideration; consequently, the algorithm will not assign to I' the label that it used for I . ■

Claim 2

The algorithm and its analysis are very simple. Essentially, if you have d labels at your disposal, then as you sweep through the intervals from left to right, assigning an available label to each interval you encounter, you can never reach a point where all the labels are currently in use.

Since our algorithm is using d labels, we can use (4.4) to conclude that it is, in fact, always using the minimum possible number of labels. We sum this up as follows.

(4.6) *The greedy algorithm above schedules every interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.*

Practice Questions

Implement the All Interval Scheduling Greedy Algorithm

Thank You

Scheduling to Minimize Lateness: An Exchange Argument Contd.: L29

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

PS-4 Questions

- Implement the Interval Scheduling Problem using Earliest start and Earliest finish time
- Schedule All Intervals using the optimal scheduling algorithm

Scheduling to Minimize Lateness

- Consider again a situation in which we have a **single resource** and a set of **n requests** to use the resource for an interval of time. Assume that the resource is available starting **at time s**.
- Instead of a start time and finish time, the request **i has a deadline** d_i , and it requires a contiguous time **interval of length** t_i , but it is willing to be scheduled at any time before the deadline.
- Each accepted request must be assigned an interval of time of length t_i , and different requests must be assigned non-overlapping intervals.
- We say that **a request i is late** if it misses the deadline, that is, if $f(i) > d_i$. The lateness of such a request i is defined to be $l_i = f(i) - d_i$. **We will say that $l_i = 0$ if request i is not late.**

Scheduling to Minimize Lateness: The Problem

The goal in **our new optimization problem** will be **to schedule all requests, using nonoverlapping intervals, so as to minimize the maximum lateness**, $L = \max_i (l_i)$. This problem arises naturally when scheduling jobs that need to use a single machine, and so we will refer to our requests as jobs.

Example of Lateness Minimization

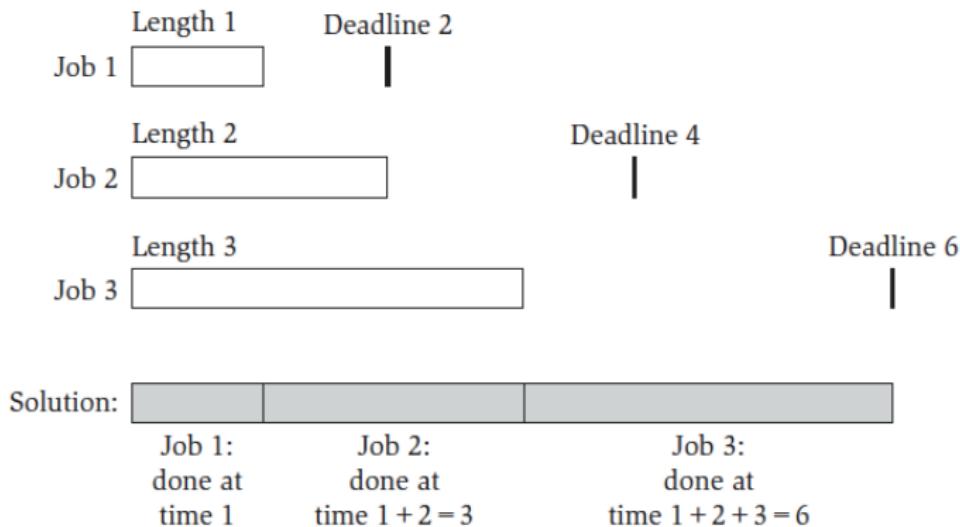


Figure 4.5 A sample instance of scheduling to minimize lateness.

Designing the Algorithm I

- One approach would be to **schedule the jobs in order of increasing length t_i , so as to get the short jobs out of the way quickly.** looks too simplistic, since it completely ignores the deadlines of the jobs. And indeed, consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 100$, while the second job has $t_2 = 10$ and $d_2 = 10$. Then the second job has to be started right away if we want to achieve lateness $L = 0$, and scheduling the second job first is indeed the optimal solution.
- The previous example suggests that we should be concerned about jobs whose available slack time $d_i - t_i$ is very small—they're the ones that need to be started with minimal delay. So a more natural greedy algorithm would be to sort jobs in order of increasing slack $d_i - t_i$. Unfortunately, this greedy rule fails as well. Consider a two-job instance

Designing the Algorithm II

where the first job has $t_1 = 1$ and $d_1 = 2$, while the second job has $t_2 = 10$ and $d_2 = 10$. Sorting by increasing slack would place the second job first in the schedule, and the first job would incur a lateness of 9. (It finishes at time 11, nine units beyond its deadline.) On the other hand, if we schedule the first job first, then it finishes on time and the second job incurs a lateness of only 1.

- Optimal- We simply sort the jobs in increasing order of their deadlines d_i , and schedule them in this order. (This rule is often called **Earliest Deadline First**.)

Pseudocode

We can assume that the jobs are labeled in the order of their deadlines, that is, we have $d_1 \leq \dots \leq d_n$. We will simply schedule all jobs in this order. Again, let s be the start time for all jobs. Job 1 will start at time $s = s(1)$ and end at time $f(1) = s(1) + t_1$; Job 2 will start at time $s(2) = f(1)$ and end at time $f(2) = s(2) + t_2$; and so forth. We will use f to denote the finishing time of the last scheduled job.

Order the jobs in order of their deadlines

Assume for simplicity of notation that $d_1 \leq \dots \leq d_n$

Initially, $f = s$

Consider the jobs $i = 1, \dots, n$ in this order

Assign job i to the time interval from $s(i) = f$ to $f(i) = f + t_i$

Let $f = f + t_i$

End

Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, \dots, n$

Practice Question

Implement the Optimal Algorithm and slack based greedy algorithm for minimizing lateness

Thank You

Optimal Caching: A More Complex Exchange Argument: L30

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

Introduction

- The problem is that of cache maintenance. consider a problem that involves processing a sequence of requests of a different form, and we develop an algorithm whose analysis requires a more subtle use of the exchange argument.

Exchange Argument

Exchange arguments are a powerful and versatile technique for proving optimality of greedy algorithms. They work by showing that you can iteratively transform any optimal solution into the solution produced by the greedy algorithm without changing the cost of the optimal solution, thereby proving that the greedy solution is optimal.

Structure of an Exchange Argument

- **Define Your Solutions.** You will be comparing your greedy solution X to an optimal solution X^* , so it's best to define these variables explicitly.
- **Compare Solutions.** Next, show that if $X \neq X^*$, then they must differ in some way. This could mean that there's a piece of X that's not in X^* , or that two elements of X that are in a different order in X^* , etc. You might want to give those pieces names.
- **Exchange Pieces.** Show how to transform X^* by exchanging some piece of X^* for some piece of X . You'll typically use the piece you described in the previous step. Then, prove that by doing so, you did not increase/decrease the cost of X^* and therefore have a different optimal solution. (You might also be able to immediately conclude that you've strictly worsened X^* , in which case you're done. This is uncommon and usually only works if there's just one optimal solution.)
- **Iterate.** Argue that you have decreased the number of differences between X and X^* by performing the exchange, and that by iterating this process you can turn X^* into X without impacting the quality of the solution.

Memory Hierachy

- Data in the **main memory** of a processor can be **accessed much more quickly** than the data on **its hard disk**; but the disk has much more storage capacity. Thus, it is important **to keep the most regularly used pieces of data in main memory, and go to disk as infrequently as possible.**
- The same phenomenon, qualitatively, occurs with on-chip caches in modern processors. These can be **accessed in a few cycles**, and so data can be retrieved from cache **much more quickly than it can be retrieved from main memory.**
- When one uses a Web browser, the disk often acts as a cache for frequently visited Web pages, since going to disk is still much faster than downloading something over the Internet.

Caching

- Caching is a general term for the **process of storing a small amount of data in a fast memory so as to reduce the amount of time spent interacting with a slow memory.**
- In the previous examples, **the on-chip cache reduces the need to fetch data from main memory, the main memory acts as a cache for the disk, and the disk acts as a cache for the Internet.**
- For caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache. To achieve this, a cache maintenance algorithm determines what to keep in the cache and what to evict from the cache when new data needs to be brought in.

Optimal Caching

- We consider a set U of n pieces of data stored in main memory. We also have a faster memory, the cache, that can hold $k < n$ pieces of data at any one time.
- We will assume that the cache initially holds some set of k items. A sequence of data items $D = d_1, d_2, \dots, d_m$ drawn from U is presented to us—this is the sequence of memory references we must process—and in processing them we must decide at all times which k items to keep in the cache.
- When item d_i is presented, we can access it very quickly if it is already in the cache; otherwise, we are required to bring it from main memory into the cache and, if the cache is full, to evict some other piece of data that is currently in the cache to make room for d_i . This is called a cache miss, and we want to have as few of these as possible.

Cache Maintenance Algorithm

Thus, on a particular sequence of memory references, a cache maintenance algorithm determines an eviction schedule—specifying which items should be evicted from the cache at which points in the sequence—and this determines the contents of the cache and the number of misses over time. Let's consider an example of this process.

- Suppose we have three items $\{a, b, c\}$, the cache size is $k = 2$, and we are presented with the sequence

$a, b, c, b, c, a, b.$

Suppose that the cache initially contains the items a and b . Then on the third item in the sequence, we could evict a so as to bring in c ; and on the sixth item we could evict c so as to bring in a ; we thereby incur two cache misses over the whole sequence. After thinking about it, one concludes that any eviction schedule for this sequence must include at least two cache misses.

Under real operating conditions, cache maintenance algorithms must process memory references d_1, d_2, \dots without knowledge of what's coming in the future; but for purposes of evaluating the quality of these algorithms, systems researchers very early on sought to understand the nature of the optimal solution to the caching problem. Given a full sequence S of memory references, what is the eviction schedule that incurs as few cache misses as possible?

Design of the Algorithm

Farthest future Algorithm

OPTIMAL CACHING – FARTHEST-IN-FUTURE THE ALGORITHM

The idea is that there is an algorithm that minimizes the amount of cache misses that occur for a set of requests. The *Farthest-in-Future Algorithm* evicts the item in the cache that is not requested until the farthest in the future. For example, given the following cache with items "a" through "f".

Cache Items	a b c d e f
Future Queries	g a b c e d a b b f

It is clear that "g" does not exist in the set of cache items and so when "g" is requested, a cache miss will result. An item will be evicted because "g" is not in the cache. According to the *Farthest-in-Future Algorithm*, element "f" will be evicted since it is the one that will be used furthest in the future.

DEFINITIONS

Cache Miss – Occurs when it is necessary to bring in an item from the main memory into the cache

Schedule – A list that contains, for each, request, which item is brought into the cache, and which item is evicted

Reduced Schedule – a schedule that only brings in an item when there is a request for that item – so a non-reduced schedule would bring an item into the cache outside of a request for that item

Optimality of Farthest Future: Exchange Argument

Theorem

Farthest-in-Future is the optimal algorithm that minimizes the number of cache misses.

Proof: To prove this, let's say there is a schedule SFF that follows the Farthest-in-Future Algorithm and that there is a schedule optimal S that has a minimum number of cache misses. It can be shown that it is possible to transform S into SFF by performing a series of evictions without increasing the number of cache misses. Since S^* is already optimal, and the number of cache misses did not increase, then SFF must be optimal. (Exchange Argument)

To perform one transformation, the statement below can be used. Right now, it isn't all that clear why this is necessary, but it should become clearer further on.

"Let S be a reduced schedule that is the same as SFF for the first j requests, then there is a reduced schedule S' that is the same as SFF for the first j+1 requests, and incurs no more misses than S"

Exchange argument

Exchange argument proof I

Proof

PROOF

Let's say the cache at the start looks like this. S and SFF have the exact same contents (this is what was defined in the previous section).

S	a b c d
SFF	a b c d

Given a schedule S, it can be shown that any request can be made, and the resulting schedule would be equal to the ideal algorithm and does not increase the number of cache misses. Essentially, we want $S = S' = \text{SFF}$ after a request is made and that the number of cache misses have not increased. If this can be shown for all different scenarios that items are requested, then *Farthest-in-Future* is the ideal algorithm.

CASE 1: ITEM IS ALREADY IN THE CACHE

Now let's say the next item to be requested is item "d". It is clear that this item exists in both caches and no evictions are necessary. Since no evictions are necessary, SFF does not change. It was specified in a prior section that there is a schedule S' that equals SFF, which incurs no more misses than S.

SFF	a b c d	Read d
S	a b c d	Read d
S'	a b c d	Read d

Here, $S = S'$ because no changes had to be made to the cache and there were no cache misses. After the request for item "d", $S = S'$, each with zero cache misses, which is a valid outcome.

Exchange argument proof II

CASE 2: ITEM IS NOT IN THE CACHE, S AND SFF EVICT THE SAME ITEM

Let's say the next item to be requested is item "e". Neither cache has this item, so an eviction will have to be made. Let's also say SFF and S both choose to evict item "a" to make room for "e". It was specified in a prior section that there is a schedule S' that equals SFF, which incurs no more misses than S.

SFF	b c d e	Evict a; Read e
S	b c d e	Evict a; Read e
S'	b c d e	Evict a; Read e

Here $S = S'$ because S and SFF both evicted the same item. Both S and S' each have one cache miss (no increase relative to one another), which is a valid outcome.

Exchange argument proof III

CASE 3: ITEM IS NOT IN THE CACHE, S AND SFF EACH EVICT A DIFFERENT ITEM

This is where the proof can become more confusing (as if it isn't already). Let's say the next item to be requested is "e". Let's also say that the ideal algorithm, SFF, evicts "a" and S evicts "b". It was specified in a prior section that there is a schedule S' that equals SFF, which incurs no more misses than S. This means that in order to get $S' = \text{SFF}$, S' should evict "a" in order to match with SFF.

SFF	b c d e	Evict a; Insert e
S	a c d e	Evict b; Insert e
S'	b c d e	Evict a; Insert e

Here, S does not agree with $S' = \text{SFF}$. Essentially, these two schedules behave the same, so long as future requests do not request items "a" or "b". This shows that there is a schedule S' that equals SFF, but it is not known whether or not S' has no more cache misses than S, since "a" or "b" can be requested in the future, which can determine whether S or S' has more cache misses. From this point onward, S and S' behave the same until one of the following happens:

CASE 3A: ITEM IS IN SFF, BUT NOT S; S EVICTS "A"

Let's say eventually item "b" gets requested. In this case, S' and SFF do not need to change. All S has to do is evict "a" to make room for "b". In this case, S has two total cache misses (from request "e" and "b"), while SFF and S' each have only one cache miss, thus fulfilling the cache miss requirement.

SFF	b c d e	Insert b
S	b c d e	Evict a; Insert b
S'	b c d e	Insert b

CASE 3B: ITEM IS NEITHER IN SFF NOR S

Let's say eventually item "f" gets requested. S', SFF, and S all need to make an eviction. S' and SFF can evict "b" to make room for "f", and S can evict "a" to make room for "f". In this case, each schedule has two cache misses, thus fulfilling the cache miss requirement.

SFF	c d e f	Evict b; Insert f
S	c d e f	Evict a; Insert f
S'	c d e f	Evict b; Insert f

CASE 3C: ITEM IS IN SFF, BUT NOT S; S EVICTS ITEM NOT EQUAL TO "A"

Let's say eventually item "b" gets requested and S evicts an item that isn't "a", say "c". Now, SFF and S' both already have "b" in the cache. In order to synchronize S and S' to make them comparable to see how many cache misses each have, S' and SFF will evict "c" to make room for "a".

SFF	a b d e	Evict c; Insert a
S	a b d e	Evict c; Insert b
S'	a b d e	Evict c; Insert a

Since "a" is being brought into the cache without a request for "a", this makes S' and SFF no longer a reduced schedule. However, the proof calls for S' and SFF being a reduced schedule. Fortunately, it is possible to transform an unreduced schedule into a reduced one. In this case, each schedule has two cache misses, thus fulfilling the cache miss requirement.

CASE 3D: ITEM IS NOT IN SFF, BUT IN S

The only item that is in S and not SFF is item "a". SFF had evicted it initially at the beginning of Case 3. Since SFF had evicted it, it must have been the item that would be requested furthest into the future. Therefore, it is not possible for item "a" to be requested before other items.

Conclusion

After going through all possible eviction decisions that the schedules can do, it was shown that for each scenario, there was indeed a schedule $S' = SFF$, where the cache misses incurred in S' no more than S .

Farthest-in-Future is the optimal algorithm that minimizes the number of cache misses.

Thank You

Huffman Codes and Data Compression: L31

Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Introduction to Data Compression

2 Huffman Codes

Data Compression

Encoding Symbols Using Bits Since computers ultimately operate on sequences of bits (i.e., sequences consisting only of the symbols 0 and 1), one needs encoding schemes that take text written in richer alphabets (such as the alphabets underpinning human languages) and converts this text into long strings of bits. The simplest way to do this would be to use a fixed number of bits for each symbol in the alphabet, and then just concatenate the bit strings for each symbol to form the text. To take a basic example, suppose we wanted to encode the 26 letters of English, plus the space (to separate words) and five punctuation characters: comma, period, question mark, exclamation point, and apostrophe. This would give us 32 symbols in total to be encoded. Now, you can form 2^b different sequences out of b bits, and so if we use 5 bits per symbol, then we can encode $2^5 = 32$ symbols—just enough for our purposes. So, for example, we could let the bit string 00000 represent a, the bit string 00001 represent b, and so forth up to 11111,

This issue of reducing the average number of bits per letter is a



Fixed Length Codes and Variable Codes

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by Figure 16.3.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

Fixed-length code, we need 3 bits to represent 6 characters: a = 000, b = 001, . . . , f = 101. This method requires 300,000 bits to code the entire file.

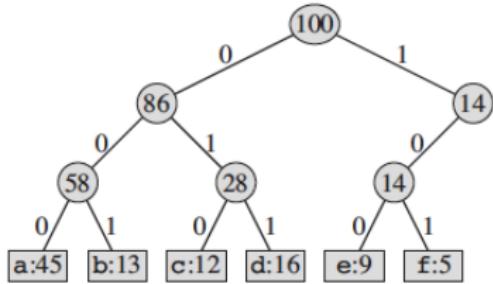
A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.

$$1.45 + 3.13 + 3.12 + 3.16 + 4.9 + 4.5 = 224000$$

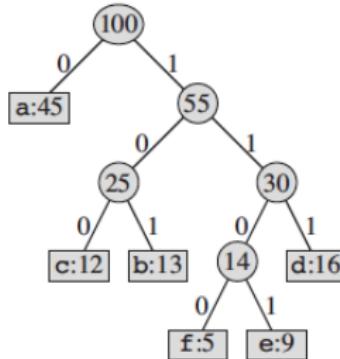
Prefix Codes

- We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called prefix codes.
- a prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.
- For example, with the variablelength prefix code of Figure 16.3, we code the 3-character file abc as $0.101.10 = 0101100$, where “.” denotes concatenation.
- Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.

Decoding a Prefix Code



(a)



(b)

Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. (a) The tree corresponding to the fixed-length code $a = 000, \dots, f = 101$. (b) The tree corresponding to the optimal prefix code $a = 0, b = 101, \dots, f = 1100$.

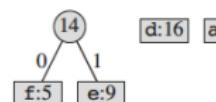
Huffman Codes

- Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- We consider the data to be a sequence of characters.
- Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

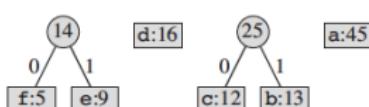
Example

(a) [f:5] [e:9] [c:12] [b:13] [d:16] [a:45]

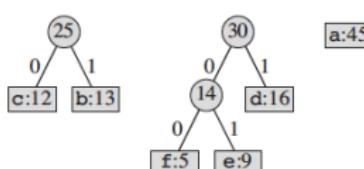
(b) [c:12] [b:13] [d:16] [a:45]



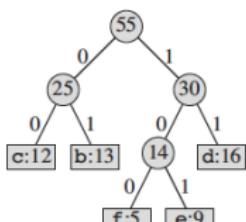
(c) [14] [d:16] [a:45]



(d) [25] [30] [a:45]



(e) [a:45]



(f) [100]

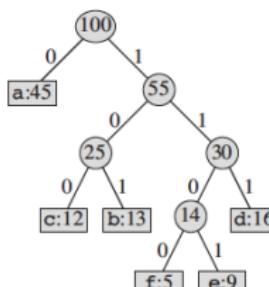


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left



Constructing the Huffman code I

HUFFMAN(C)

```
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return EXTRACT-MIN( $Q$ )    // return the root of the tree
```

Constructing the Huffman code II

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue.
The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

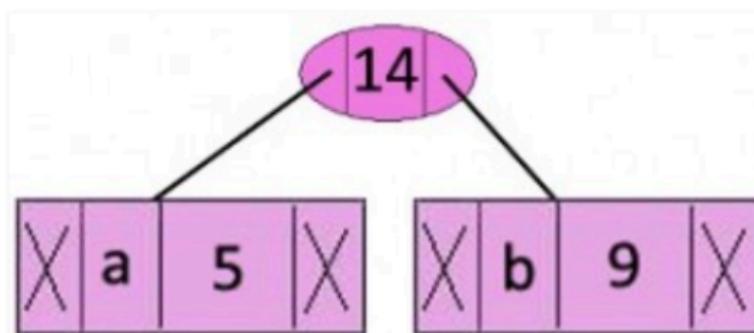
Constructing the Huffman code III

character Frequency

a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency $5 + 9 = 14$.

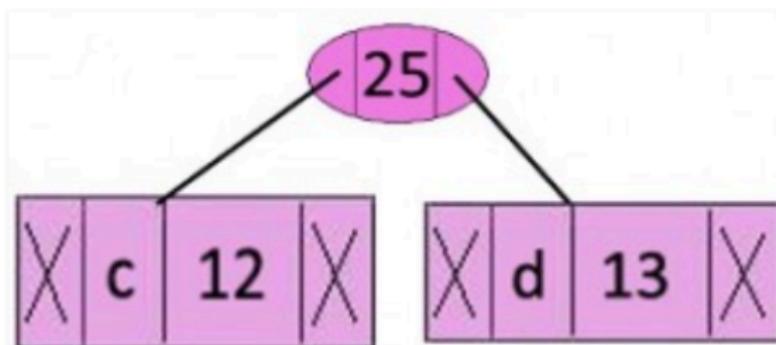


Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

Constructing the Huffman code IV

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

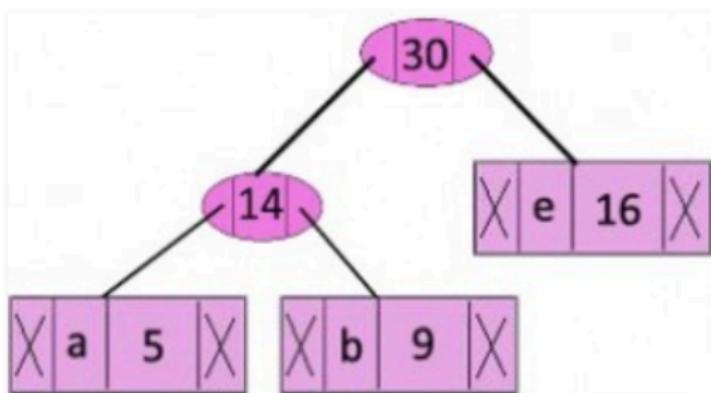
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency $12 + 13 = 25$



Constructing the Huffman code V

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

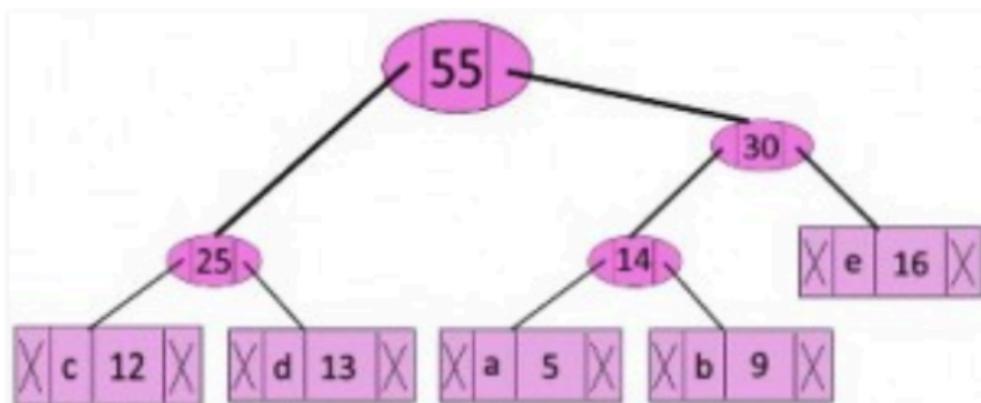
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency $14 + 16 = 30$



Constructing the Huffman code VI

character	Frequency
Internal Node	25
Internal Node	30
f	45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency $25 + 30 = 55$

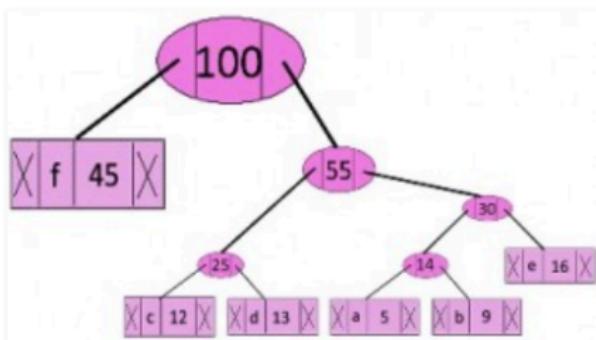


Constructing the Huffman code VII

Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency $45 + 55 = 100$



Now min heap contains only one node.

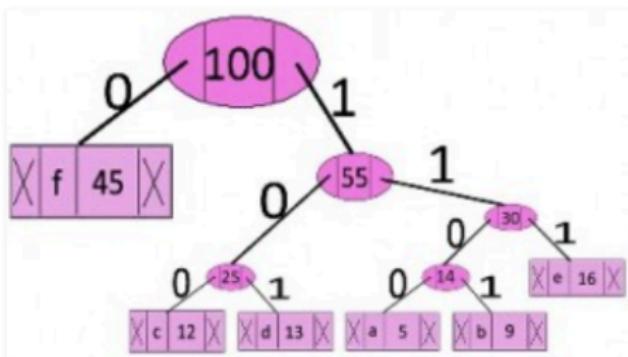
character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Printing the Code

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Thank You

Clustering: L32

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

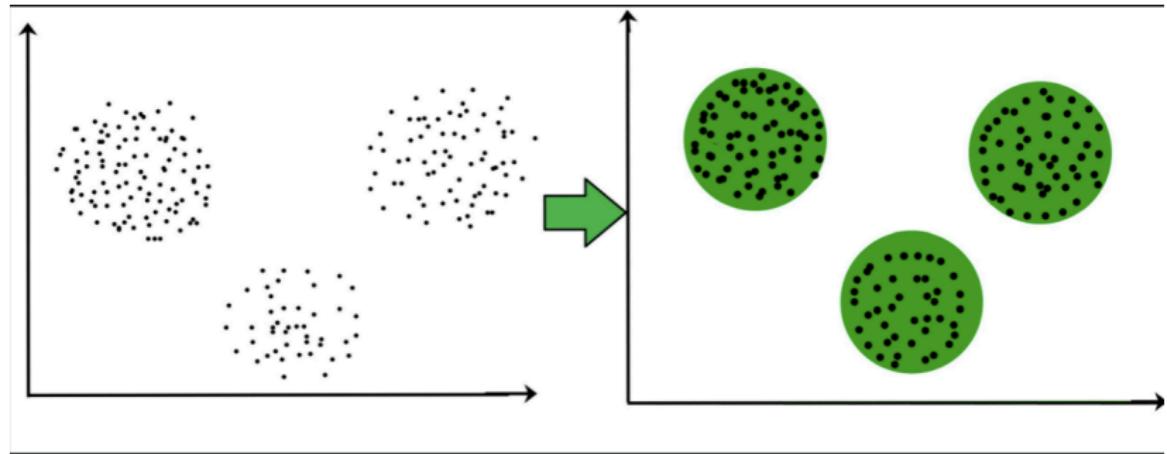
Outline

1 Clustering

Introduction

- Clustering arises whenever one has a collection of objects—say, a set of photographs, documents, or microorganisms—that one is trying to classify or organize into coherent groups.
- Faced with such a situation, it is natural to look first for measures of how similar or dissimilar each pair of objects is.
- One common approach is to define a distance function on the objects, with the interpretation that objects at a larger distance from one another are less similar to each other.
- For example, we could define the distance between two species to be the number of years since they diverged in the course of evolution; we could define the distance between two images in a video stream as the number of corresponding pixels at which their intensity values differ by at least some threshold.

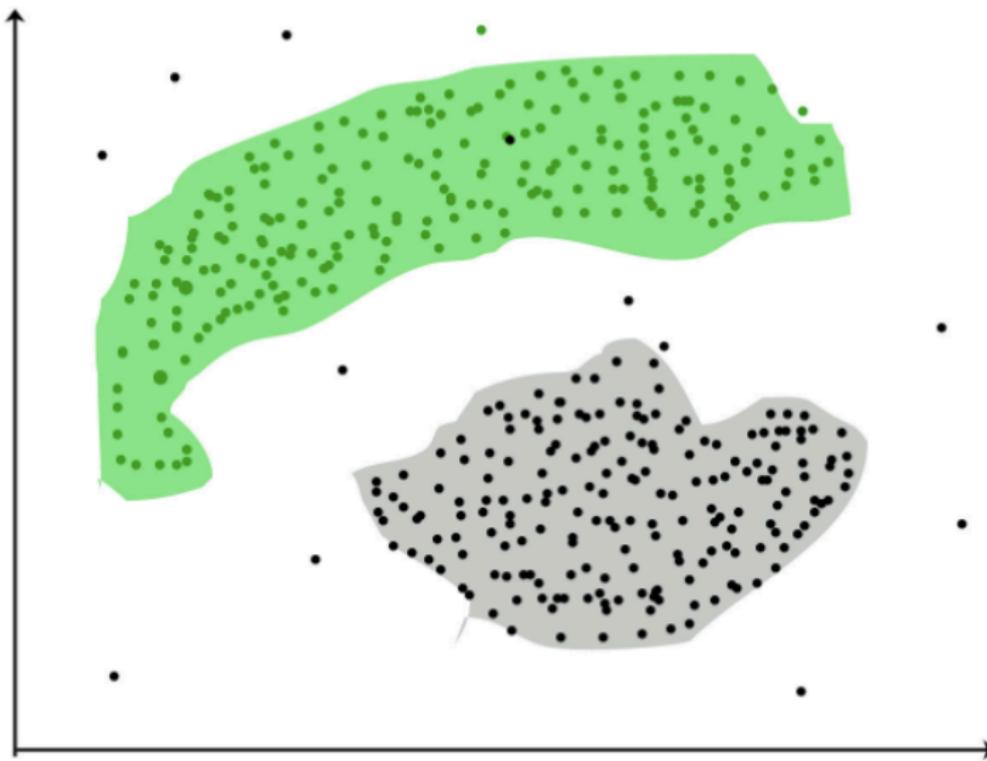
Clustering I



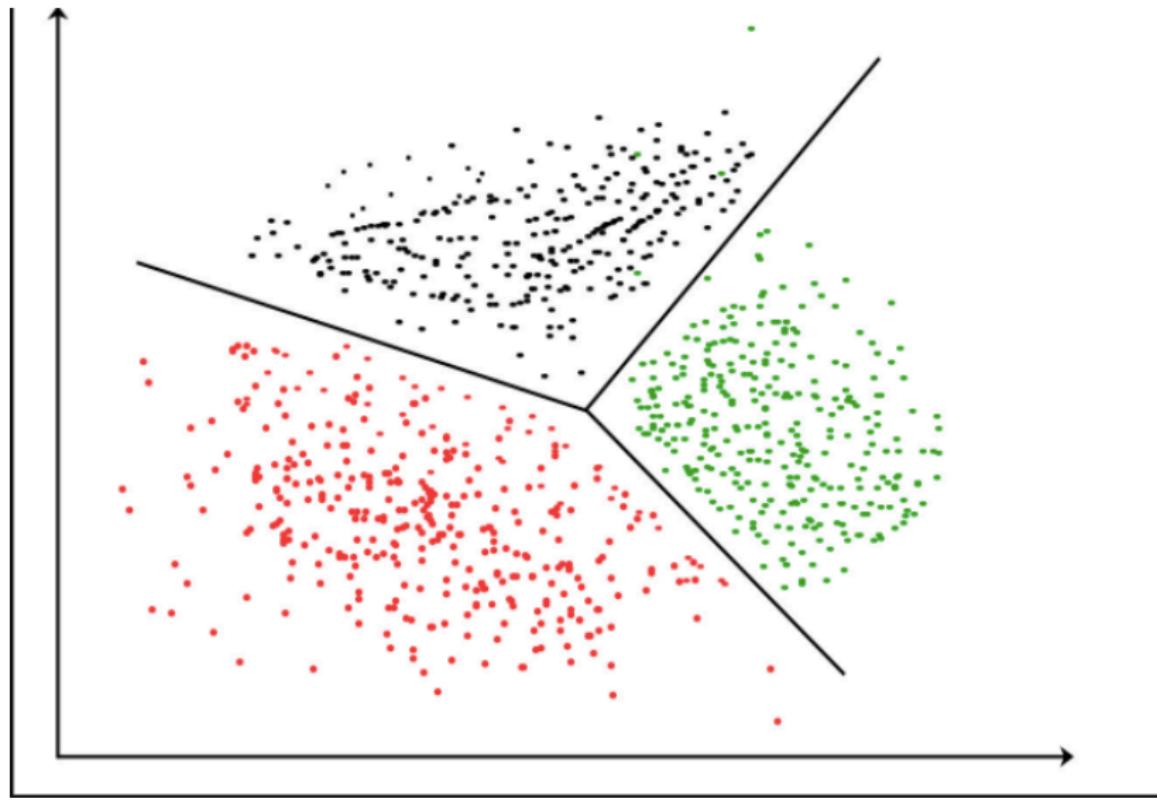
Clustering

○●●●●●○○○○○○○○○○○

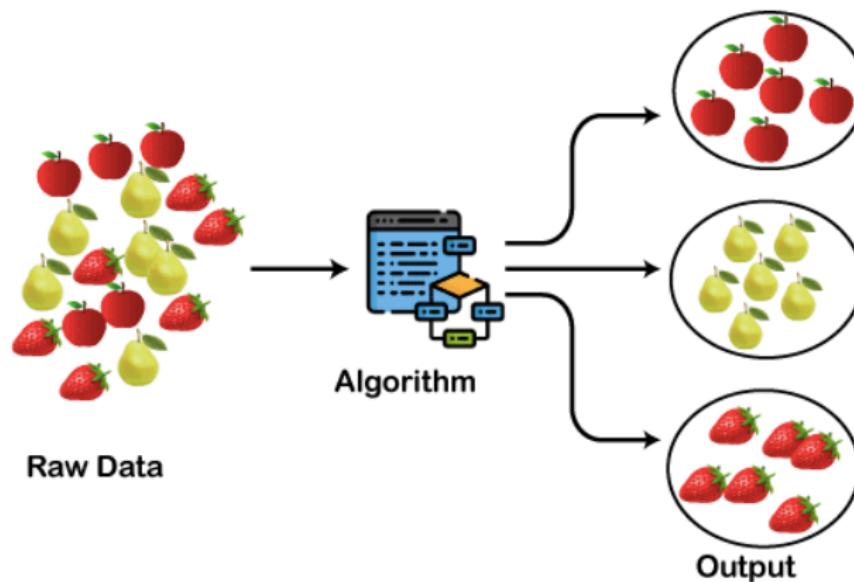
Clustering II



Clustering III



Clustering IV



Clustering V

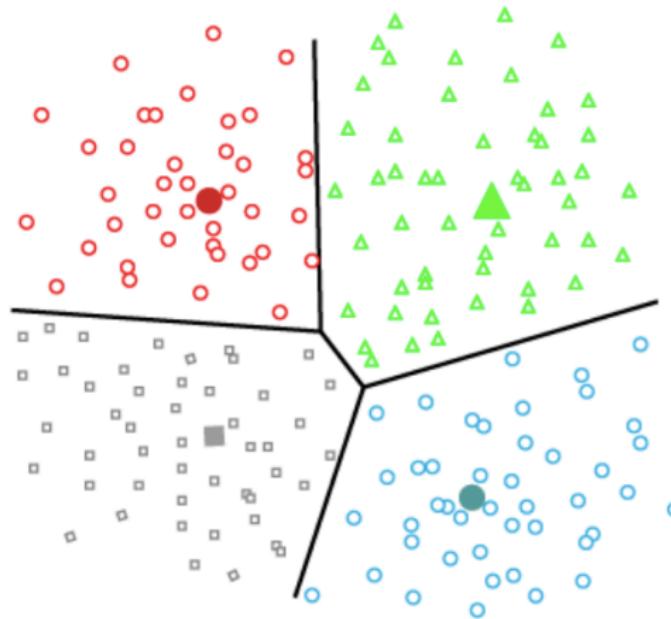
Types of Clustering Methods

The clustering methods are broadly divided into **Hard clustering** (datapoint belongs to only one group) and **Soft Clustering** (data points can belong to another group also). But there are also other various approaches of Clustering exist. Below are the main clustering methods used in Machine learning:

1. **Partitioning Clustering**
2. **Density-Based Clustering**
3. **Distribution Model-Based Clustering**
4. **Hierarchical Clustering**
5. **Fuzzy Clustering**

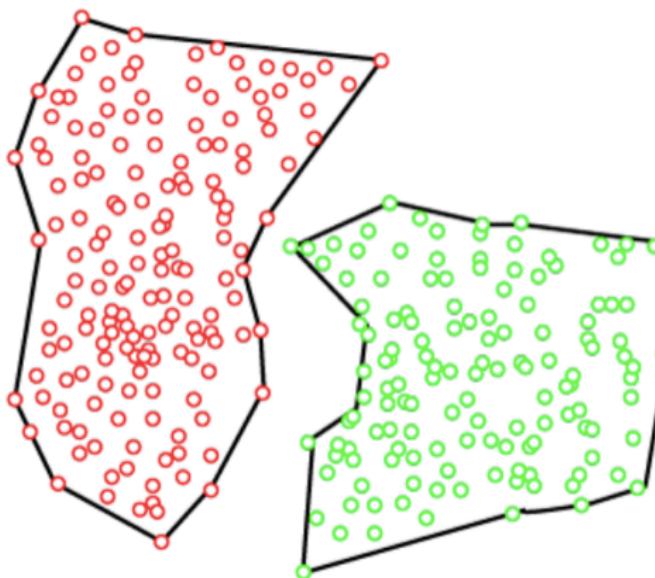
Partitioning

It is a type of clustering that divides the data into non-hierarchical groups. It is also known as the centroid-based method.



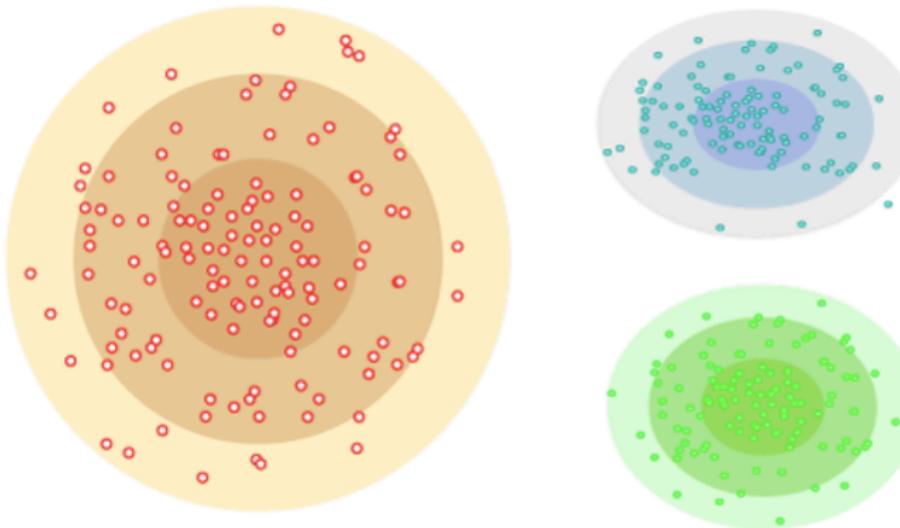
Density-Based Clustering

The density-based clustering method connects the highly-dense areas into clusters, and the arbitrarily shaped distributions are formed as long as the dense region can be connected.



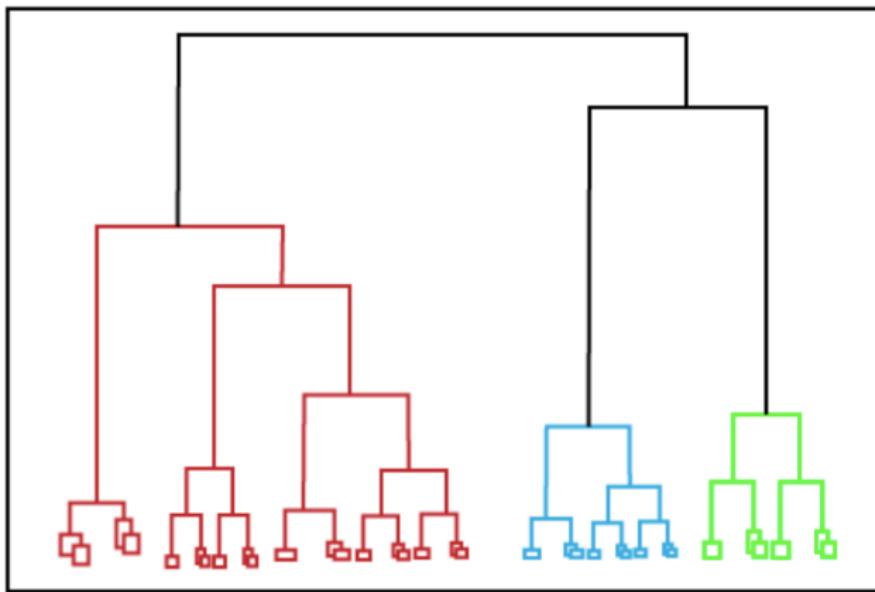
Distribution Model-Based Clustering

The grouping is done by assuming some distributions commonly Gaussian Distribution.



Hierarchical Clustering

In this technique, the dataset is divided into clusters to create a tree-like structure, which is also called a dendrogram.



Clustering

The Clustering Problem

Now, given a distance function on the objects, the clustering problem seeks to divide them into groups so that, intuitively, objects within the same group are “close,” and objects in different groups are “far apart.” Starting from this vague set of goals, the field of clustering branches into a vast number of technically different approaches, each seeking to formalize this general notion of what a good set of groups might look like.

Design of the Algorithm

- To find a clustering of maximum spacing, we consider growing a graph on the vertex set U . The connected components will be the clusters, and we will try to bring nearby points together into the same cluster as rapidly as possible.
- Thus we start by drawing an edge between the closest pair of points. We then draw an edge between the next closest pair of points. We continue adding edges between pairs of points, in order of increasing distance $d(p_i, p_j)$.
- In this way, we are growing a graph H on U edge by edge, with connected components corresponding to clusters.
- Notice that we are only interested in the connected components of the graph H , not the full set of edges; so if we are about to add the edge (p_i, p_j) and find that p_i and p_j already belong to the same cluster, we will refrain from adding the edge—so H will actually be a union of trees.

Connection with Min Spanning Tree

What is the connection to minimum spanning trees? It's very simple:

Although our graph-growing procedure was motivated by this cluster-merging idea, our procedure is precisely Kruskal's Minimum Spanning Tree Algorithm. We are doing exactly what Kruskal's Algorithm would do if given a graph G on U in which there was an edge of cost $d(p_i, p_j)$ between each pair of nodes (p_i, p_j) . The only difference is that we seek a k -clustering, so we stop the procedure once we obtain k connected components. In other words, we are running Kruskal's Algorithm but stopping it just before it adds its last $k - 1$ edges. This is equivalent to taking the full minimum spanning tree T (as Kruskal's Algorithm would have produced it), deleting the $k - 1$ most expensive edges (the ones that we never actually added), and defining the k -clustering to be the resulting connected components C_1, C_2, \dots, C_k . Thus, iteratively merging clusters is equivalent to computing min spanning without the last step

Thank You

Counting Inversions : L34*

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India
***L33 is Merge Sort so we move to L34**

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

Divide and Conquer

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. Divide and conquer algorithm consists of two parts:

- **Divide :** Divide the problem into a number of sub problems.
The sub problems are solved recursively.
- **Conquer :** The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Counting Inversions I

The Problem

- 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.

Counting Inversions II

An Inversion

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 $a_i > a_j$, that is, 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 .

Example

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).

Counting Inversions III

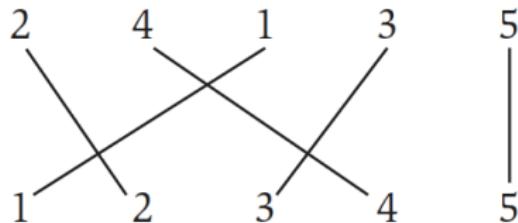


Figure 5.4 Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.

Solution using Divide and Conquer I

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.

$O(n \log n)$ Algorithm

- We set $m = n/2$ 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,

Solution using Divide and Conquer II

$O(n\log n)$ Algorithm

- 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.

Solution using Divide and Conquer III

Merge and Count

- 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 previous discussion, this is precisely what we will need for the “combining” step that computes the number of first-half/second-half inversions.

Solution using Divide and Conquer IV

Relation with Merge Sort

- 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 difference 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.
- 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.

Algorithm

```
merge-and-count(A,B)
; A,B two input lists (sorted)
; C output list
; i,j current pointers to each list, start at beginning
; a_i, b_j elements pointed by i, j
; count number of inversion, initially 0

while A,B != empty
    append min(a_i,b_j) to C
    if b_j < a_i
        count += number of element remaining in A
        j++
    else
        i++
    ; now one list is empty
    append the remainder of the list to C
return count,
```

With merge-and-count, we can design the count inversion algorithm as follows:

```
sort-and-count(L)
if L has one element return 0
else
    divide L into A, B
    (rA, A) = sort-and-count(A)
    (rB, B) = sort-and-count(B)
    (r, L) = merge-and-count(A,B)
return r = rA+rB+r, L
```

$$T(n) = O(n \lg n)$$

End

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

$i = 6$



two sorted halves



auxiliary array

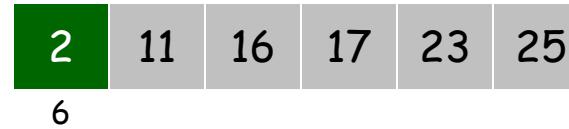
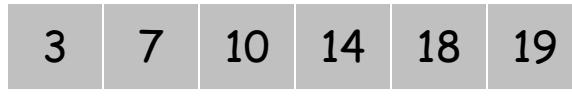
Total:

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

$i = 6$



two sorted halves

2

auxiliary array

Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

$i = 6$



two sorted halves

2

auxiliary array

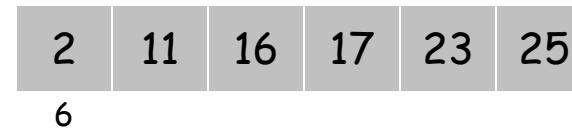
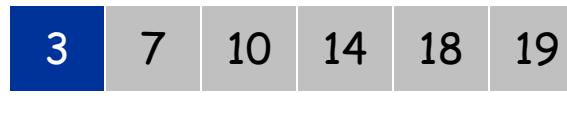
Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

$i = 6$



two sorted halves



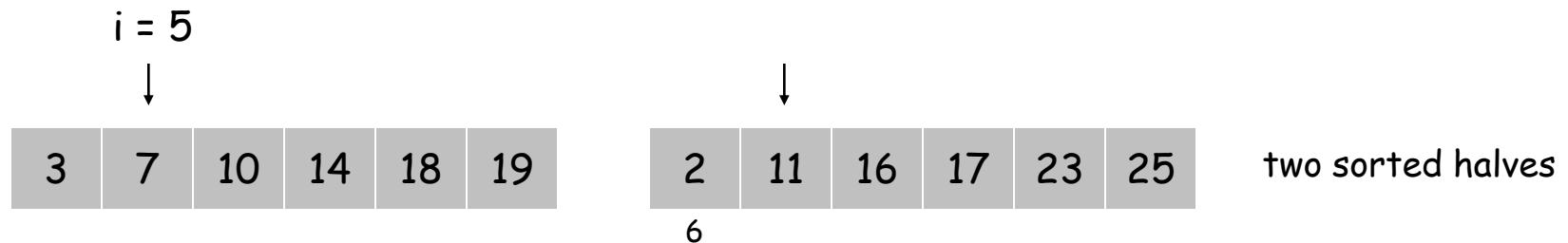
auxiliary array

Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

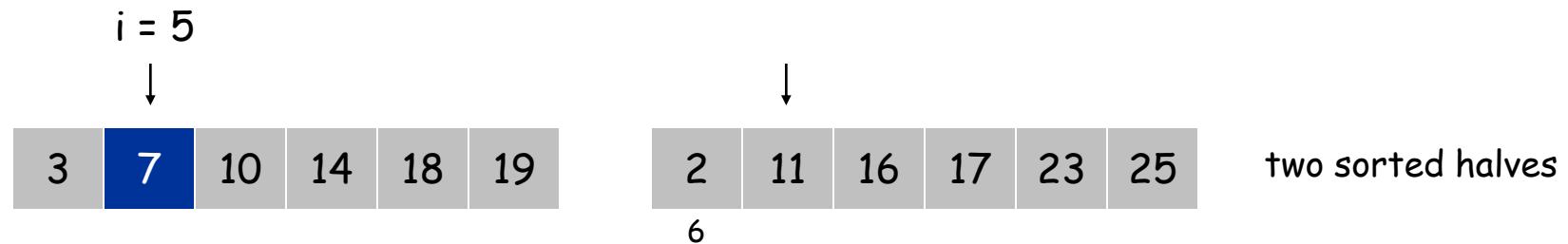


Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

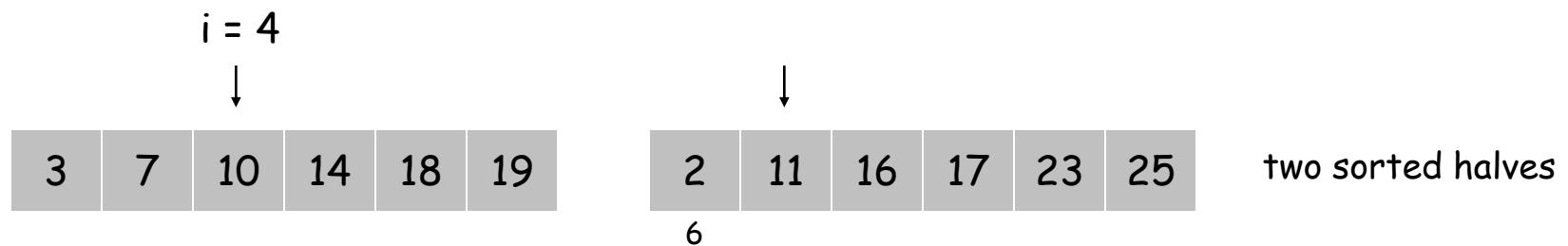


Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

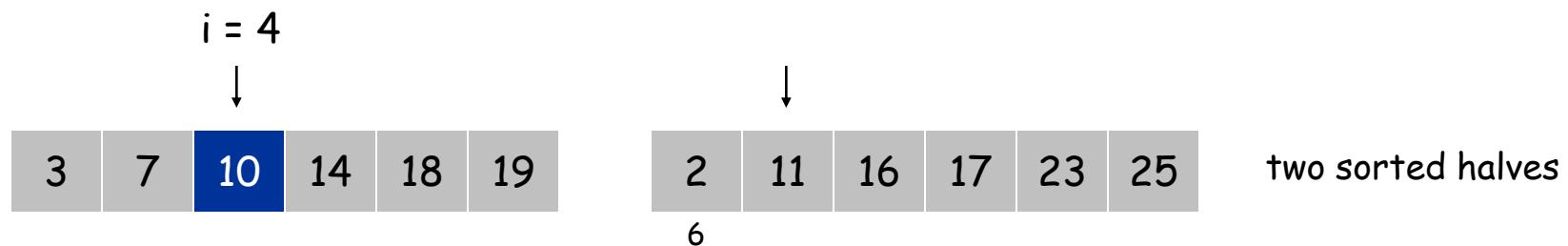


Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

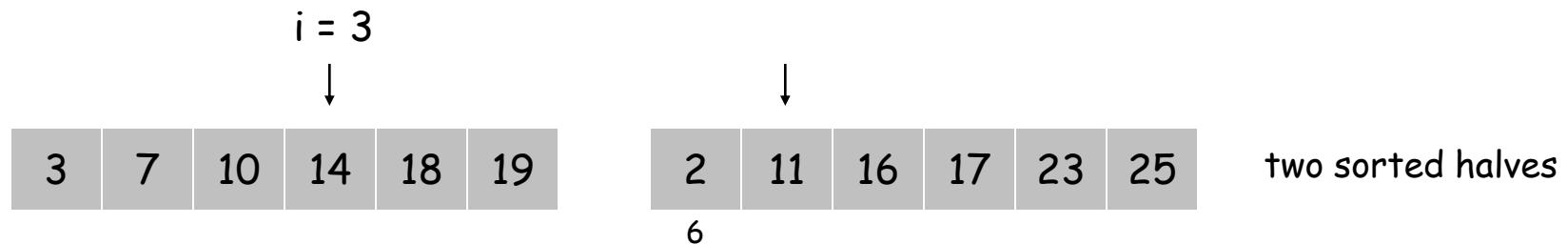


Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

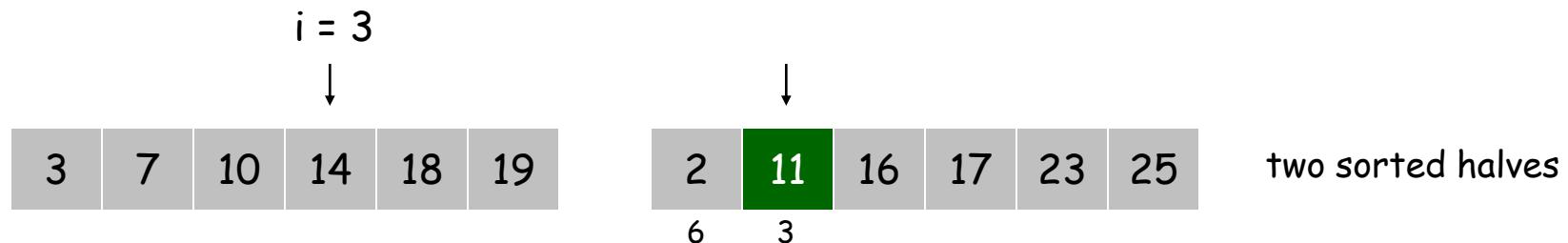


Total: 6

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

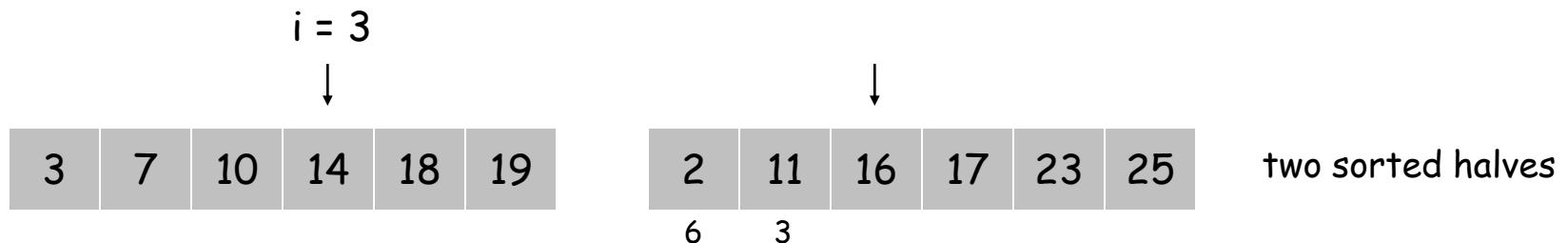


Total: 6 + 3

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

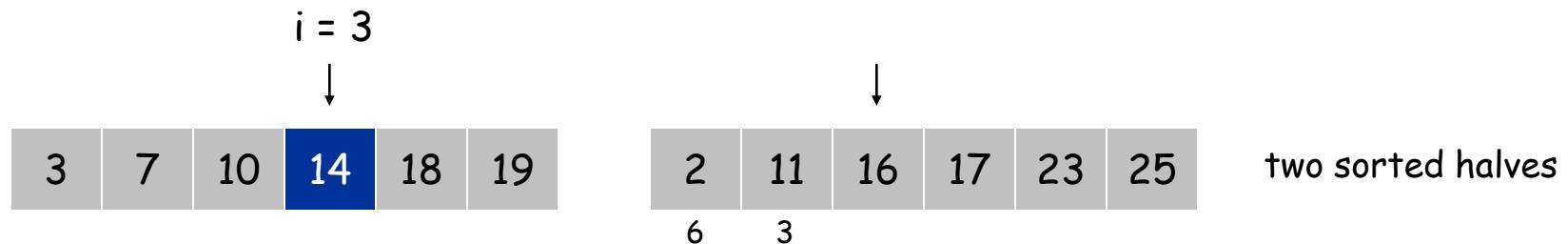


Total: $6 + 3$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

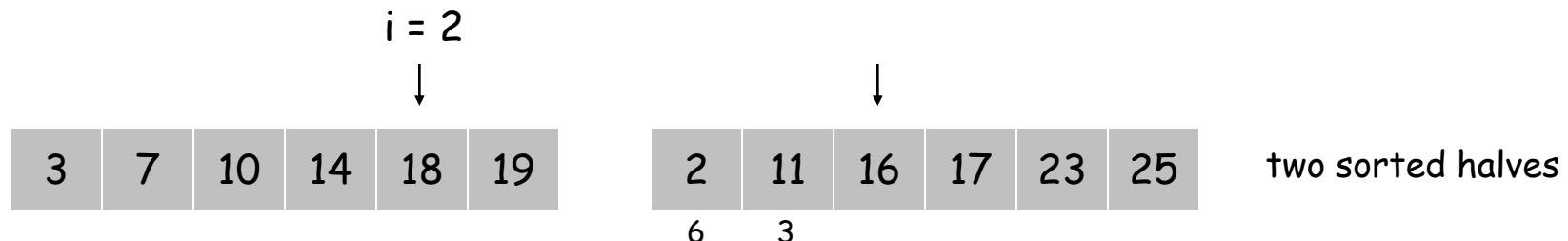


Total: $6 + 3$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



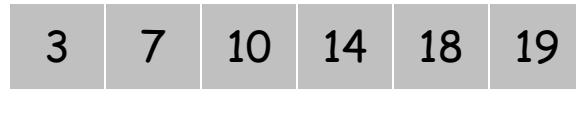
Total: $6 + 3$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

$i = 2$



two sorted halves



auxiliary array

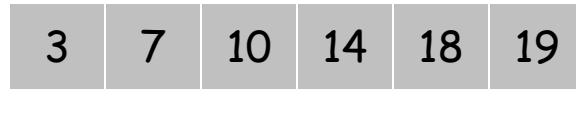
Total: $6 + 3 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

$i = 2$



two sorted halves



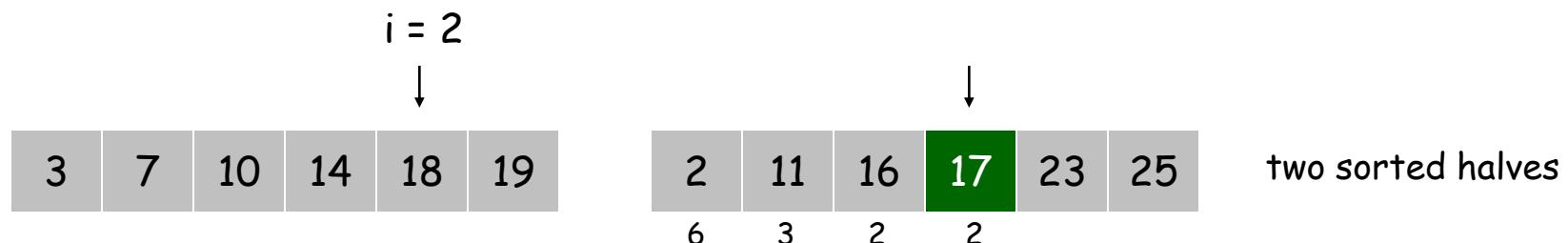
auxiliary array

Total: $6 + 3 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

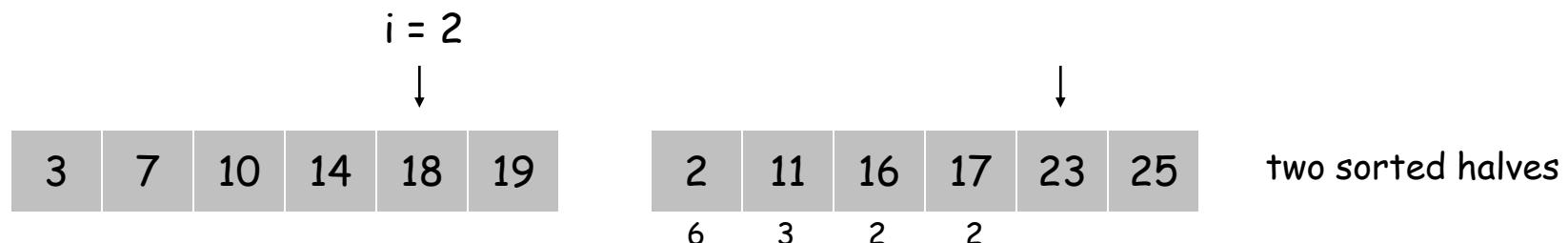


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

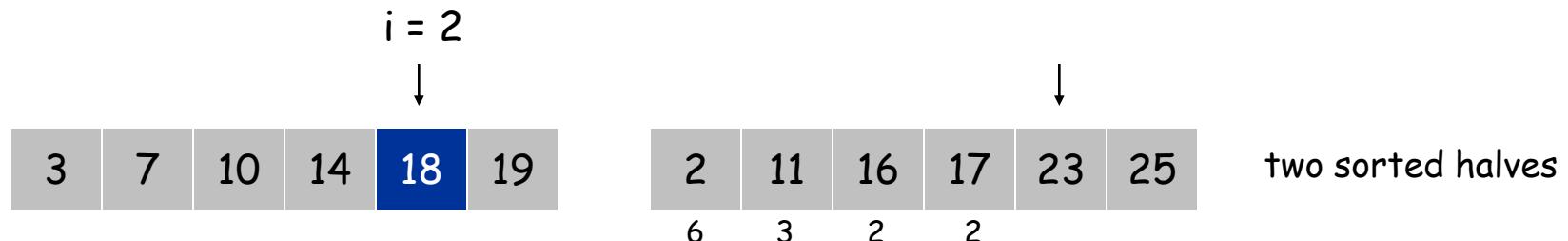


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

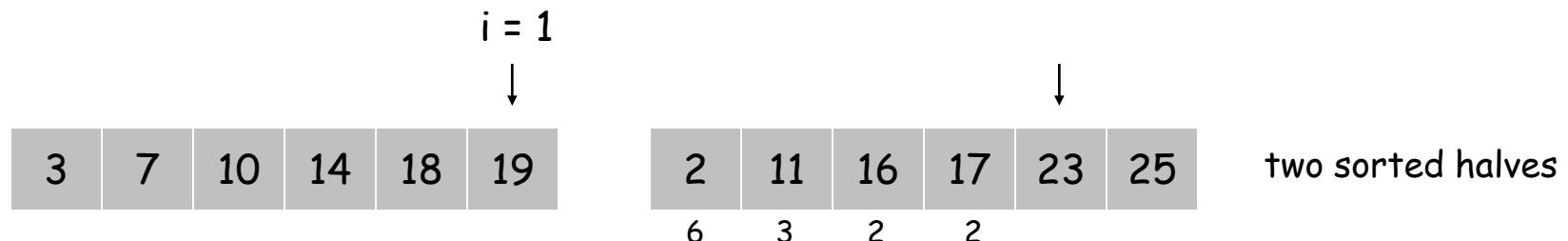


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

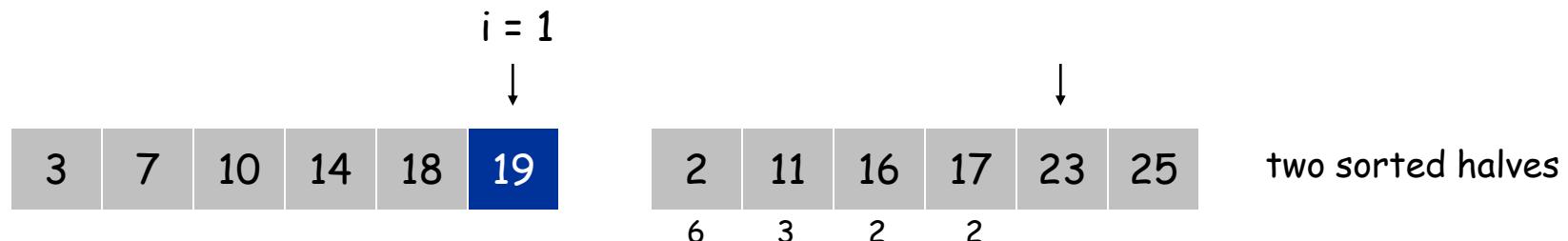


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

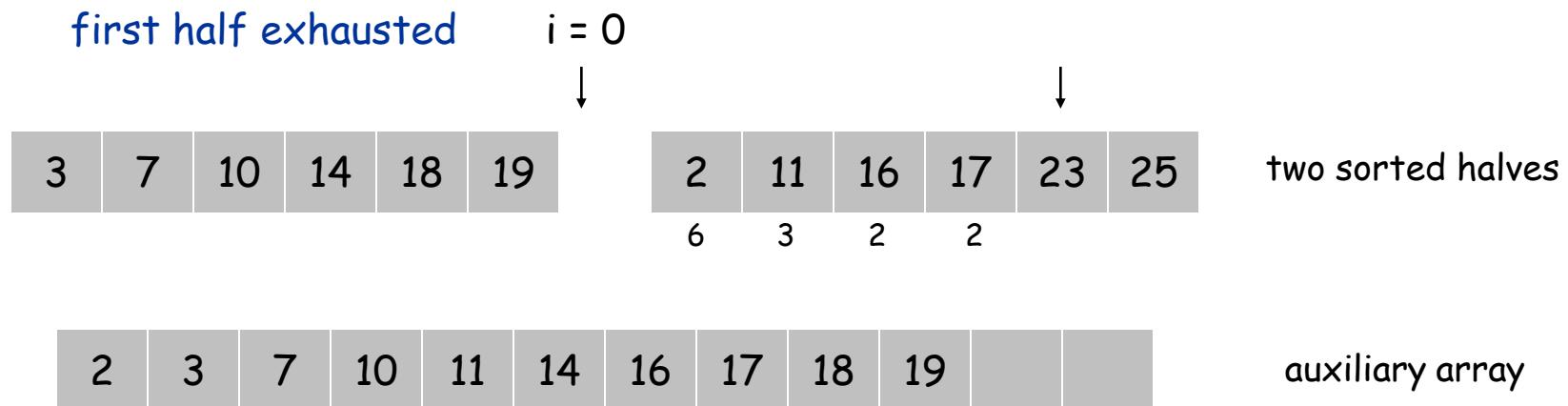


Total: $6 + 3 + 2 + 2$

Merge and Count

Merge and count step.

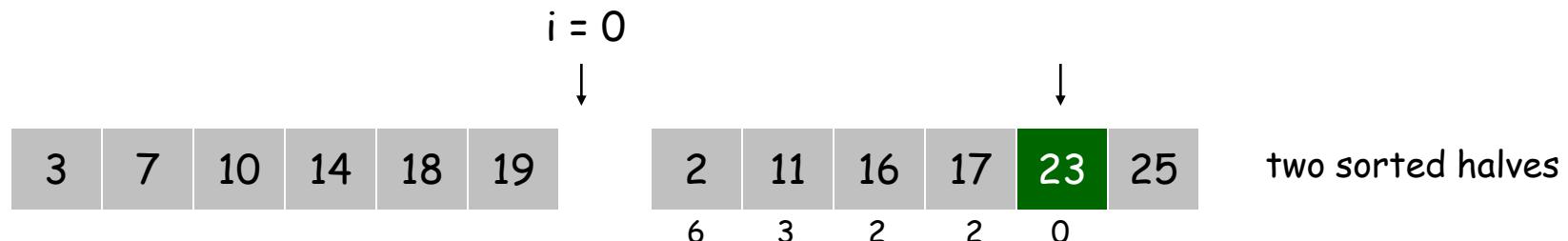
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

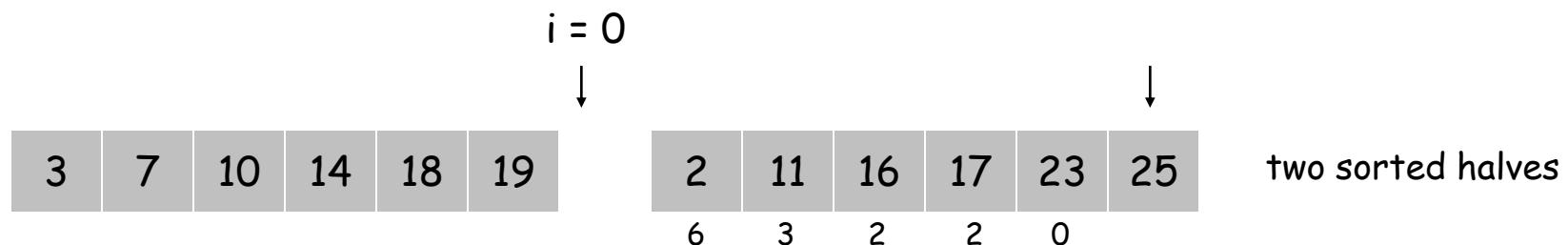


Total: $6 + 3 + 2 + 2 + 0$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

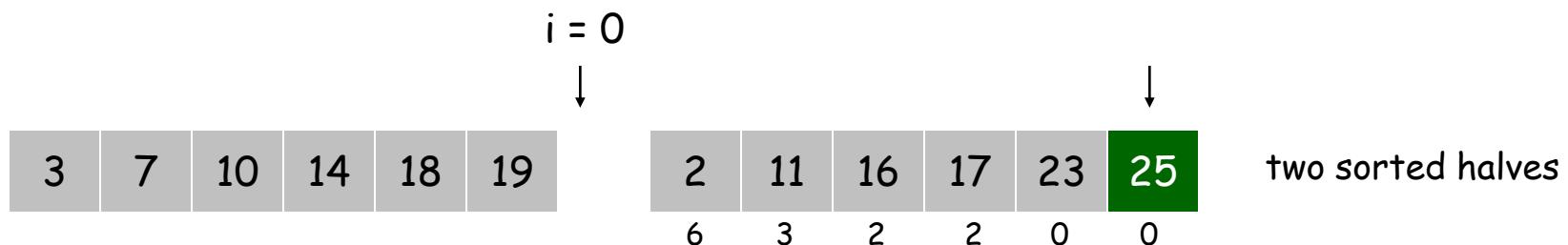


Total: $6 + 3 + 2 + 2 + 0$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.

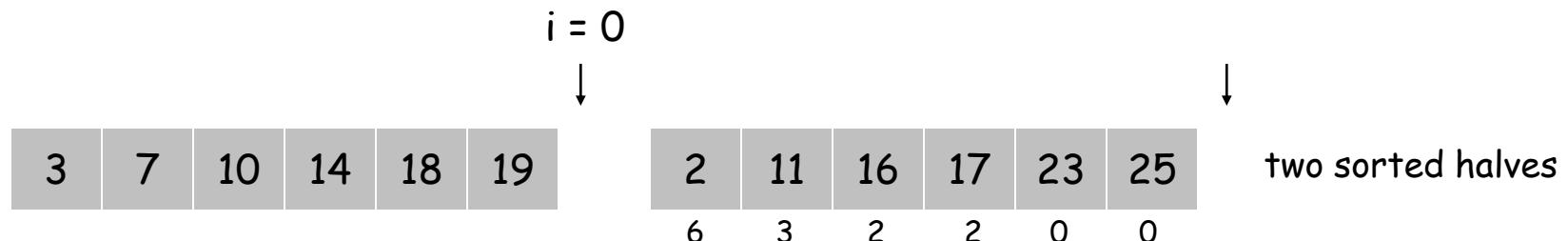


Total: $6 + 3 + 2 + 2 + 0 + 0$

Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



$$\text{Total: } 6 + 3 + 2 + 2 + 0 + 0 = 13$$

Quick Sort : L35*

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Quicksort
ooooooooo

Outline

1 Quicksort

Quick Sort

- The quicksort algorithm has a worst-case running time of θn^2 on an input array of n numbers.
- Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $,n \lg n /$, and the constant factors hidden in the $,n \lg n /$ notation are quite small.
- It also has the advantage of sorting in place
- Quicksort, like merge sort, applies the divide-and-conquer paradigm

Steps of Quicksort

- If the range has less than two elements, return immediately as there is nothing to do. Possibly for other very short lengths a special-purpose sorting method is applied and the remainder of these steps skipped.
- Otherwise pick a value, called a pivot, that occurs in the range (the precise manner of choosing depends on the partition routine, and can involve randomness).
- Partition the range: reorder its elements, while determining a point of division, so that all elements with values less than the pivot come before the division, while all elements with values greater than the pivot come after it; elements that are equal to the pivot can go either way. Since at least one instance of the pivot is present, most partition routines ensure that the value that ends up at the point of division is equal to the pivot, and is now in its final position (but termination of quicksort does not depend on this, as long as sub-ranges strictly smaller than the original are produced).
- Recursively apply the quicksort to the sub-range up to the point of division and to the sub-range after it, possibly excluding from both ranges the element equal to the pivot at the point of division. (If the partition produces a possibly larger sub-range near the boundary where all elements are known to be equal to the pivot, these can be excluded as well.)

Divide and Conquer Strategy

Description of quicksort

Quicksort, like merge sort, applies the divide-and-conquer paradigm introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \dots r]$:

Divide: Partition (rearrange) the array $A[p \dots r]$ into two (possibly empty) subarrays $A[p \dots q - 1]$ and $A[q + 1 \dots r]$ such that each element of $A[p \dots q - 1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q + 1 \dots r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p \dots q - 1]$ and $A[q + 1 \dots r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p \dots r]$ is now sorted.

Description of Quick sort I

The following procedure implements quicksort:

```
QUICKSORT( $A, p, r$ )
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3       $\text{QUICKSORT}(A, p, q - 1)$ 
4       $\text{QUICKSORT}(A, q + 1, r)$ 
```

To sort an entire array A , the initial call is $\text{QUICKSORT}(A, 1, A.length)$.

Pivot Element

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, **given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.** All this should be done in linear time.

Partition Procedure I

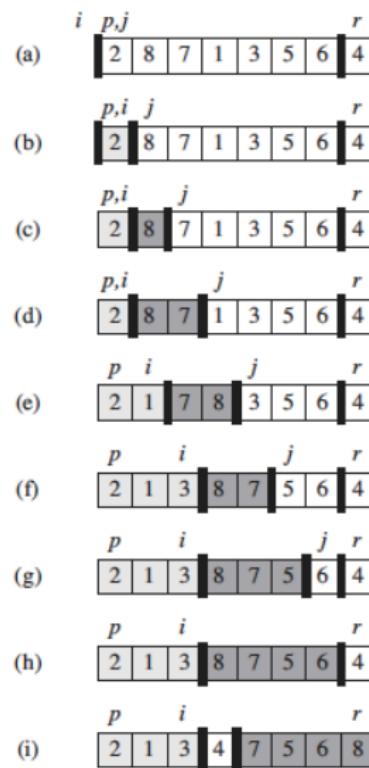
Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \dots r]$ in place.

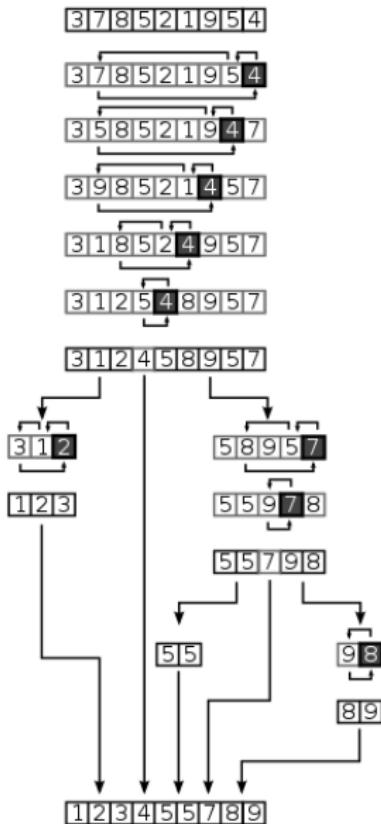
PARTITION(A, p, r)

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

Partition Procedure II

Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot

Example of quicksort



Practice Problems

- Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = 13; 19; 9; 5; 12; 8; 7; 4; 21; 2; 6; 11$.

Performance of Quick Sort and Randomized Quick Sort: L32

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Performance of Quick Sort

2 Randomized Quick Sort

Correctness of Partition Algorithm I

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

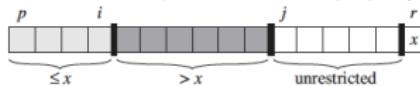


Figure 7.2 The four regions maintained by the procedure `PARTITION` on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x , the values in $A[i + 1..j - 1]$ are all greater than x , and $A[r] = x$. The subarray $A[j..r - 1]$ can take on any values.

Initialization: Prior to the first iteration of the loop, $i = p - 1$ and $j = p$. Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$; the only action in the loop is to increment j . After j is incremented, condition 2 holds for $A[j - 1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \leq x$; the loop increments i , swaps $A[i]$ and $A[j]$, and then increments j . Because of the swap, we now have that $A[i] \leq x$, and condition 1 is satisfied. Similarly, we also have that $A[j - 1] > x$, since the item that was swapped into $A[j - 1]$ is, by the loop invariant, greater than x .

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x , those greater than x , and a singleton set containing x .

Correctness of Partition Algorithm II

The running time of PARTITION on the subarray $A[p \dots r]$ is $\theta(n)$ where $n=r-p+1$

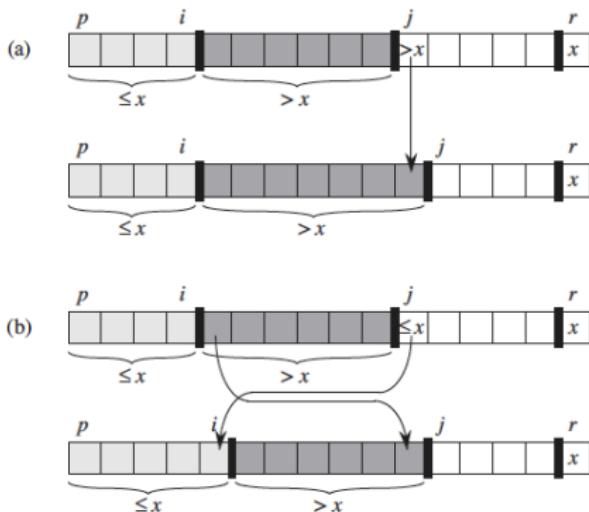


Figure 7.3 The two cases for one iteration of procedure PARTITION. (a) If $A[j] > x$, the only action is to increment j , which maintains the loop invariant. (b) If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Worst Case Partitioning

- The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements.

$$\begin{aligned}T(n) &= T(n-1) + T(0) + \Theta(n) \\&= T(n-1) + \Theta(n).\end{aligned}$$

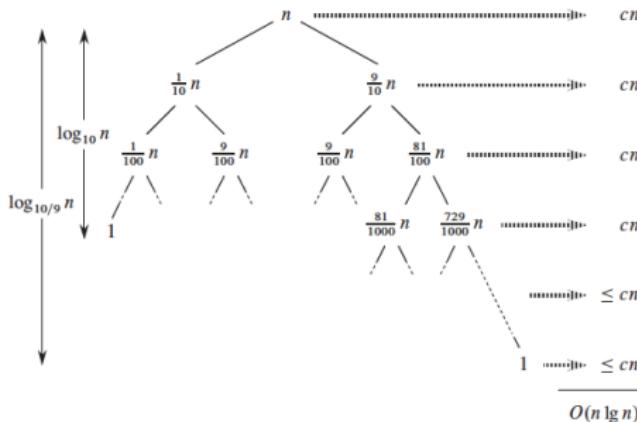
Best Case Partitioning

- In the most even possible split, PARTITION produces two subproblems, each of size no more than $n=2$, since one is of size $bn=2c$ and one of size $dn=2e-1$. In this case, quicksort runs much faster. The recurrence for the running time is then

$$T(n) = 2T(n/2) + \Theta(n),$$

Balanced Partitioning

The average-case running time of quicksort is much closer to the best case than to the worst case



Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) = T(9n/10) + T(n/10) + cn ,$$

Randomized Quick Sort

RANDOMIZED-PARTITION(A, p, r)

- 1 $i = \text{RANDOM}(p, r)$
- 2 exchange $A[r]$ with $A[i]$
- 3 **return** PARTITION(A, p, r)

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

- 1 **if** $p < r$
- 2 $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3 RANDOMIZED-QUICKSORT($A, p, q - 1$)
- 4 RANDOMIZED-QUICKSORT($A, q + 1, r$)

Thank You

Closest Pair of Points: L37

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

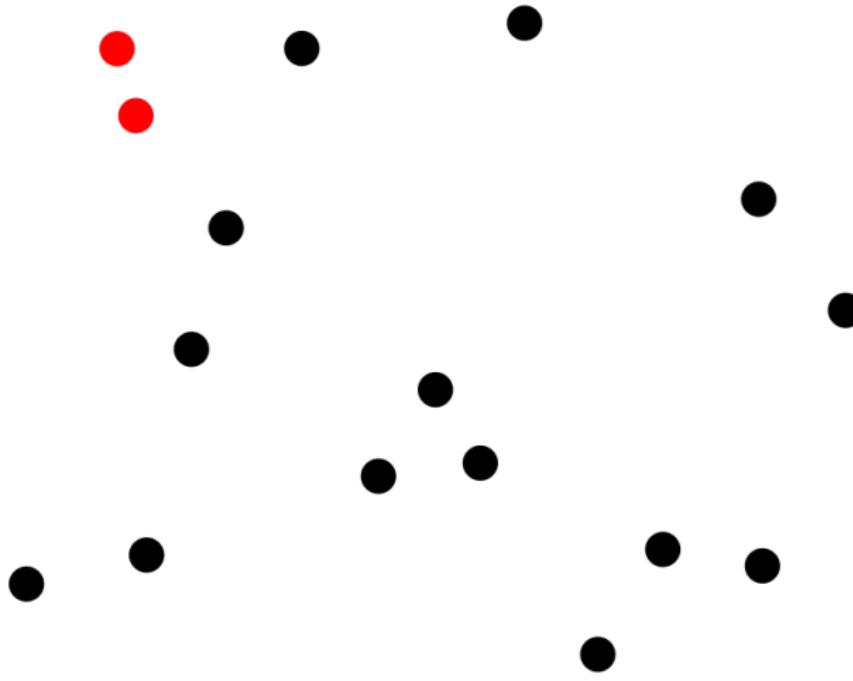
Outline

1 Closest Pair of Points

Introduction

- The problem we consider is very simple to state: Given n points in the plane, find the pair that is closest together.
- 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.
- 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.

Closest Pair of Points



O(nlogn) Algorithm I

- 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 a pair of points p_i, p_j that 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.

Solution

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.

O(nlogn) Algorithm II

1D Approach is inefficient

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.

$O(n \log n)$ Algorithm III

Divide and Conquer

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 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.

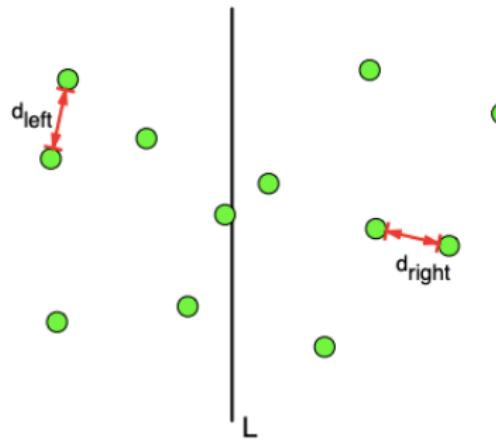
O(nlogn) Algorithm IV

Tricky Combine Step

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 (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.

O(nlogn) Algorithm V

Let $d = \min\{d_{\text{left}}, d_{\text{right}}\}$.



- d would be the answer, except maybe L split a close pair!

Building the Recursion I

Notations

- Let's get a few easy things out of the way first. It will be very useful if every recursive call, on a set $P' \subset P$, begins with two lists: a list P'_x in which all the points in P' have been sorted by increasing x-coordinate, and a list P'_y in which all the points in P' 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.

Building the Recursion II

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 $\lceil n/2 \rceil$ positions of the list P_x (the “left half”) and R to be the set of points in the final $\lfloor n/2 \rfloor$ 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

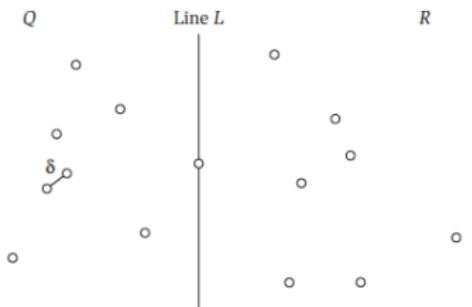


Figure 5.6 The first level of recursion: The point set P is divided evenly into Q and R by the line L , and the closest pair is found on each side recursively.

Building the Recursion III

4 lists

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.

Building the Recursion IV

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 subproblems as part of a linear-time “combining” operation?
- Let σ be the minimum of $d(q^*, 0, q^*, 1)$ and $d(r^*, 0, r^*, 1)$. The real question is: Are there points $q \in Q$ and $r \in R$ for which $d(q, r) < \sigma$? 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 .
- 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 σ of L .
- Let $S \subset P$ 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 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 .
- Statement (5.10) 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 σ from each other.



Building the recurrence V

Combining the Colutions

- So having done this, we can compare the smallest such distance to σ , and we can report one of two things: (i) the closest pair of points in S , if their distance is less than σ ; or (ii) the (correct) conclusion that no pairs of points in S are within σ of each other.
- 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 .
- The reason such an approach works now is due to the extra knowledge (the value of σ) 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 (5.9) we have now determined whether the minimum distance between a point in Q and a point in R is less than σ , and if so, we have found the closest such pair.

Pseudocode

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
    Endif

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

     $\delta = \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$ 
     $x^* = \text{maximum } x\text{-coordinate of a point in set } Q$ 
     $L = \{(x, y) : x = x^*\}$ 
     $S = \text{points in } P \text{ within distance } \delta \text{ of } L.$ 

    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  $s, s'$  be pair achieving minimum of these distances
        ( $O(n)$  time)

    If  $d(s, s') < \delta$  then
        Return  $(s, s')$ 
    Else if  $d(q_0^*, q_1^*) < d(r_0^*, r_1^*)$  then
        Return  $(q_0^*, q_1^*)$ 
```

Illustration of Pseudocode I

As a pre-processing step, the input array is sorted according to x coordinates.

- 1) Find the middle point in the sorted array, we can take $P[n/2]$ as middle point.
- 2) Divide the given array in two halves. The first subarray contains points from $P[0]$ to $P[n/2]$. The second subarray contains points from $P[n/2+1]$ to $P[n-1]$.
- 3) Recursively find the smallest distances in both subarrays. Let the distances be d_L and d_R . Find the minimum of d_L and d_R . Let the minimum be d .

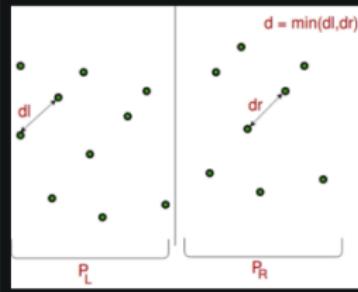
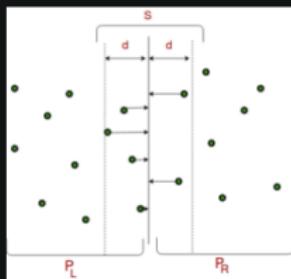


Illustration of Pseudocode II

- 4) From the above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through $P[n/2]$ and find all points whose x coordinate is closer than d to the middle vertical line. Build an array $\text{strip}[]$ of all such points.



- 5) Sort the array $\text{strip}[]$ according to y coordinates. This step is $O(n \log n)$. It can be optimized to $O(n)$ by recursively sorting and merging.
6) Find the smallest distance in $\text{strip}[]$. This is tricky. From the first look, it seems to be a $O(n^2)$ step, but it is actually $O(n)$. It can be shown that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate).
7) Finally return the minimum of d and distance calculated in the above step (step 6)

Thank You

Karatsuba's algorithm for fast integer multiplication: L38

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

- 1 Karatsuba's Algorithm for fast integer multiplication

Introduction

- The problem we consider is an extremely basic one: the multiplication of two integers.
- You first compute a “partial product” by multiplying each digit of y separately by x , and then you add up all the partial products.
- 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 $O(n^2)$.
- But, in fact, it is possible to improve on $O(n^2)$ time using a different, recursive way of performing the multiplication.

Elementary School Algorithm

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

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

Karatsuba algorithm for fast integer multiplication I

Let x and y be represented as n -digit strings in some base B . For any positive integer m less than n , one can write the two given numbers as

$$\begin{aligned}x &= x_1 B^m + x_0, \\y &= y_1 B^m + y_0,\end{aligned}$$

where x_0 and y_0 are less than B^m . The product is then

$$\begin{aligned}xy &= (x_1 B^m + x_0)(y_1 B^m + y_0) \\&= z_2 B^{2m} + z_1 B^m + z_0,\end{aligned}$$

where

$$\begin{aligned}z_2 &= x_1 y_1, \\z_1 &= x_1 y_0 + x_0 y_1, \\z_0 &= x_0 y_0.\end{aligned}$$

Karatsuba algorithm for fast integer multiplication II

The improved algorithm is based on a more clever way 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

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

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

Pseudocode

Recursive-Multiply(x,y):

 Write $x = x_1 \cdot 2^{n/2} + x_0$

$y = y_1 \cdot 2^{n/2} + y_0$

 Compute $x_1 + x_0$ and $y_1 + y_0$

$p = \text{Recursive-Multiply}(x_1 + x_0, y_1 + y_0)$

$x_1y_1 = \text{Recursive-Multiply}(x_1, y_1)$

$x_0y_0 = \text{Recursive-Multiply}(x_0, y_0)$

 Return $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$

Examples I

Example [edit]

To compute the product of 12345 and 6789, where $B = 10$, choose $m = 3$. We use m right shifts for decomposing the input operands using the resulting base ($B^m = 1000$), as:

$$12345 = 12 \cdot 1000 + 345$$

$$6789 = 6 \cdot 1000 + 789$$

Only three multiplications, which operate on smaller integers, are used to compute three partial results:

$$z_2 = 12 \times 6 = 72$$

$$z_0 = 345 \times 789 = 272205$$

$$z_1 = (12 + 345) \times (6 + 789) - z_2 - z_0 = 357 \times 795 - 72 - 272205 = 283815 - 72 - 272205 = 11538$$

We get the result by just adding these three partial results, shifted accordingly (and then taking carries into account by decomposing these three inputs in base 1000 like for the input operands):

$$\text{result} = z_2 \cdot (B^m)^2 + z_1 \cdot (B^m)^1 + z_0 \cdot (B^m)^0, \text{ i.e.}$$

$$\text{result} = 72 \cdot 1000^2 + 11538 \cdot 1000 + 272205 = 83810205.$$

Examples II

Consider the following multiplication: 47×78

```
x = 47  
x = 4 * 10 + 7
```

```
x1 = 4  
x2 = 7
```

```
y = 78  
y = 7 * 10 + 8
```

```
y1 = 7  
y2 = 8
```

```
a = x1 * y1 = 4 * 7 = 28  
c = x2 * y2 = 7 * 8 = 56  
b = (x1 + x2)(y1 + y2) - a - c = 11 * 15 - 28 - 56
```

11 * 15 can in turn be multiplied using Karatsuba Algorithm

Examples III

Example

Step 1: Compute $a \cdot c = 672$

Step 2: Compute $b \cdot d = 2652$

Step 3: Compute $(a+b)(c+d) = 134 \cdot 46 = 6164$

Step 4: Compute $\textcircled{3} - \textcircled{2} - \textcircled{1} = 2840$

Step 5:

$$\begin{array}{r}
 6720000 \\
 2652 \\
 \hline
 284000 \\
 \hline
 7006652 \approx (1234)(5678)
 \end{array}$$

$$\begin{array}{r}
 x = 56 + 8 \\
 y = 1234 \\
 \hline
 c \quad d
 \end{array}$$

Analysis of the Algorithm

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. Using the solution to that recurrence from earlier in the chapter, we have

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

Thank You

Convolution and Fast Fourier Transform:

L39

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

- 1 Fourier Transform
- 2 Convolution
- 3 Fast Fourier Transform

Layman's understanding of Fourier Transform

Here's a plain-English metaphor:

- What does the Fourier Transform do? Given a smoothie, it finds the recipe.
- How? Run the smoothie through filters to extract each ingredient.
- Why? Recipes are easier to analyze, compare, and modify than the smoothie itself.
- How do we get the smoothie back? Blend the ingredients.

Mathematical English

The Fourier Transform takes a **time-based pattern**, measures every possible cycle, and returns the overall "cycle recipe" (**the amplitude, offset, & rotation speed for every cycle** that was found).

Frequency Domain and Time Domain I

A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa.

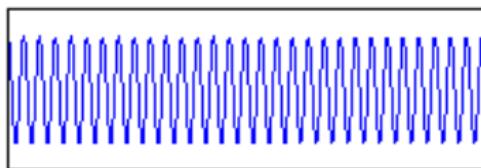
Frequency Domain and Time Domain II

A time-domain graph shows how a signal changes over time, whereas a frequency-domain graph shows how much of the signal lies within each given frequency band over a range of frequencies. A frequency-domain representation can also include information on the phase shift that must be applied to each sinusoid in order to be able to recombine the frequency components to recover the original time signal.

An example is the Fourier transform, which converts a time function into a sum or integral of sine waves of different frequencies, each of which represents a frequency component.

Frequency Domain and Time Domain III

Time Domain

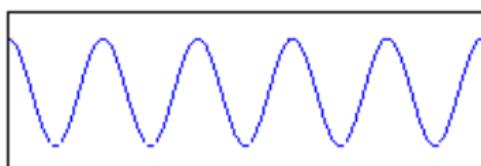


time →

Frequency Domain



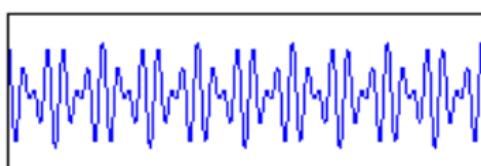
frequency →



time →



frequency →



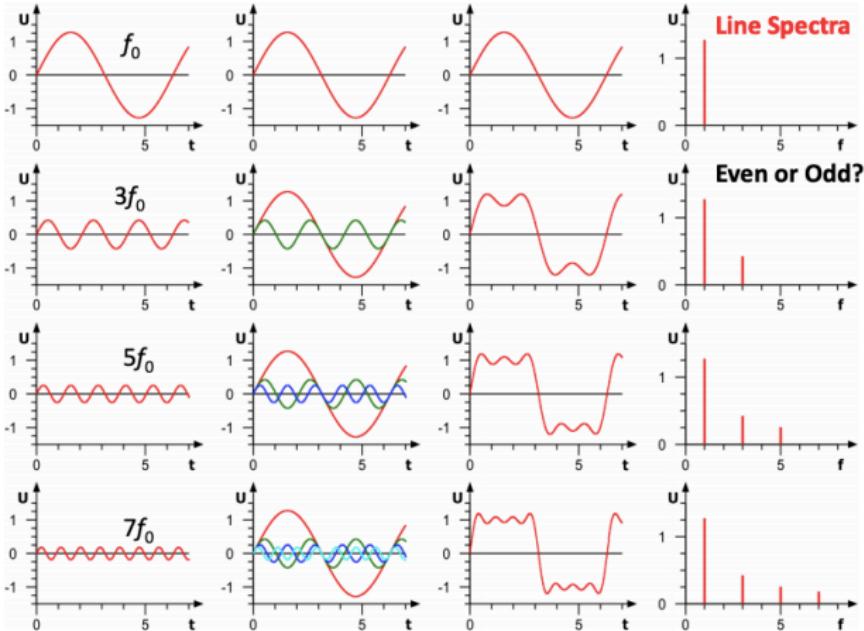
time →



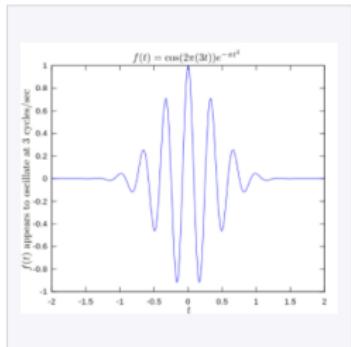
frequency →

Frequency Domain and Time Domain IV

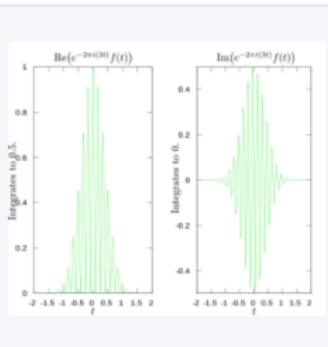
Example: Periodic Square Wave as Sum of Sinusoids



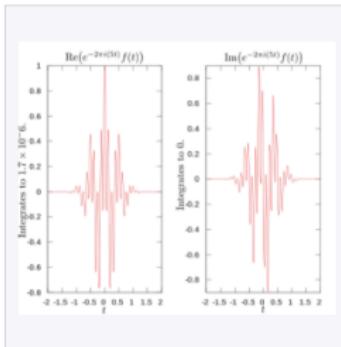
Frequency Domain and Time Domain V



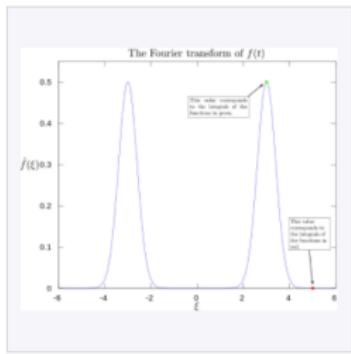
Original function showing oscillation 3 Hz.



Real and imaginary parts of integrand for Fourier transform at 3 Hz



Real and imaginary parts of integrand for Fourier transform at 5 Hz



Magnitude of Fourier transform, with 3 and 5 Hz labeled.

Why Frequency Domain Representation is Required

- One of the main reasons for using a frequency-domain representation of a problem is to simplify the mathematical analysis. For mathematical systems governed by linear differential equations, a very important class of systems with many real-world applications, converting the description of the system from the time domain to a frequency domain converts the differential equations to algebraic equations, which are much easier to solve.
- In addition, looking at a system from the point of view of frequency can often give an intuitive understanding of the qualitative behavior of the system, and a revealing scientific nomenclature has grown up to describe it, characterizing the behavior of physical systems to time varying inputs using terms such as bandwidth, frequency response, gain, phase shift, resonant frequencies, time constant, resonance width, damping factor, Q factor, harmonics, spectrum, power spectral density, eigenvalues, poles, and zeros.
- An example of a field in which frequency-domain analysis gives a better understanding than time domain is music; the theory of operation of musical instruments and the musical notation used to record and discuss pieces of music is implicitly based on the breaking down of complex sounds into their separate component frequencies (musical notes).

Fourier Transform

The Fourier transform $F_1[\omega]$ of $f[t]$ is:

$$F_1[\omega] = \int_{-\infty}^{\infty} f[t] e^{-i\omega t} dt$$

Note that it is a function of ω . If we interpret t as the time, then ω is the angular frequency. Thus we have replaced a function of time with a spectrum in frequency.

The inverse Fourier transform takes $F[\omega]$ and, as we have just proved, reproduces $f[t]$:

$$f[t] = \frac{1}{2\pi} \int_{-\infty}^{\infty} F_1[\omega] e^{i\omega t} d\omega$$

You should be aware that there are other common conventions for the Fourier transform (which is why we labelled the above transforms with a subscript). For example, some texts use a different normalisation:

$$F_2[\omega] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f[t] e^{-i\omega t} dt$$

Convolution Theorem

In mathematics, the convolution theorem states that under suitable conditions the Fourier transform of a convolution of two functions (or signals) is the pointwise product of their Fourier transforms. More generally, convolution in one domain (e.g., time domain) equals point-wise multiplication in the other domain (e.g., frequency domain).

Functions of a continuous-variable

Consider two functions $g(x)$ and $h(x)$ with Fourier transforms G and H :

$$\begin{aligned} G(f) &\triangleq \mathcal{F}\{g\}(f) = \int_{-\infty}^{\infty} g(x)e^{-i2\pi fx} dx, \quad f \in \mathbb{R} \\ H(f) &\triangleq \mathcal{F}\{h\}(f) = \int_{-\infty}^{\infty} h(x)e^{-i2\pi fx} dx, \quad f \in \mathbb{R} \end{aligned}$$

where \mathcal{F} denotes the Fourier transform operator.

convolution of g and h is defined by:

$$r(x) = \{g * h\}(x) \triangleq \int_{-\infty}^{\infty} g(\tau)h(x - \tau) d\tau = \int_{-\infty}^{\infty} g(x - \tau)h(\tau) d\tau.$$

In this context the asterisk denotes convolution, instead of standard multiplication.

The convolution theorem states that:^{[1][2]}:eq.8

$$R(f) \triangleq \mathcal{F}\{r\}(f) = G(f)H(f). \quad f \in \mathbb{R} \quad (\text{Eq.1a})$$

Applying the inverse Fourier transform \mathcal{F}^{-1} , produces the corollary:^[2]:eqs.7,10

Convolution theorem

$$r(x) = \{g * h\}(x) = \mathcal{F}^{-1}\{G \cdot H\}, \quad \text{where } \cdot \text{ denotes point-wise multiplication} \quad (\text{Eq.1b})$$

The theorem also generally applies to multi-dimensional functions.

Notion of Convolution I



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

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

Notion of Convolution II

In other words,

$$\begin{aligned}a * b = (a_0 b_0, a_0 b_1 + a_1 b_0, a_0 b_2 + a_1 b_1 + a_2 b_0, \dots, \\ a_{n-2} b_{n-1} + a_{n-1} b_{n-2}, a_{n-1} b_{n-1}).\end{aligned}$$

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,

Notion of Convolution III

5.6 Convolutions and the Fast Fourier Transform

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

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

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

Applications of Convolution

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

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

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.)

Computational Notion of 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., $m = n$), although everything we say carries over directly to the case of vectors of unequal lengths.
- The definition of convolution, after all, gives us a perfectly valid way to compute it: for each k , we just calculate the sum
- 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 (n^2) arithmetic operations. Spending $O(n^2)$ time on computing the convolution seems natural, as the definition involves $O(n^2)$ multiplications $a_i b_j$.

Could one design an algorithm that bypasses the quadratic-size definition of convolution and computes it in some smarter way?

Fourier Transform
oooooooo

Convolution
ooooooo●

Fast Fourier Transform
oooo

$O(n \log n)$

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)**.

Connection between convolution and multiplication of two polynomials

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).

$O(n)$ 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.

Fourier Transform
oooooooo

Convolution
ooooooo

Fast Fourier Transform
oo●o

Fast Fourier Transform FFT

Fourier Transform
oooooooo

Convolution
oooooooo

Fast Fourier Transform
ooo●

Thank You

Dynamic Programming Control Abstraction, Memoization Vs. Recursion, nth Fibonacci Number: L40, L41

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

Dynamic Programming Control Abstraction

- Divide and conquer can sometimes serve as an alternative approach, but the versions of divide and conquer that we saw in the previous chapter are often not strong enough to reduce exponential brute-force search down to polynomial time. We now turn to a more powerful and subtle design technique, dynamic programming.
- To set about developing an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problem that satisfies a few basic properties.
 - There are only a polynomial number of subproblems.
 - The solution to the original problem can be easily computed from the solutions to the subproblems. (For example, the original problem may actually be one of the subproblems.)
 - There is a natural ordering on subproblems from “smallest” to “largest,” together with an easy-to-compute recurrence (as in (6.1) and (6.2)) that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.

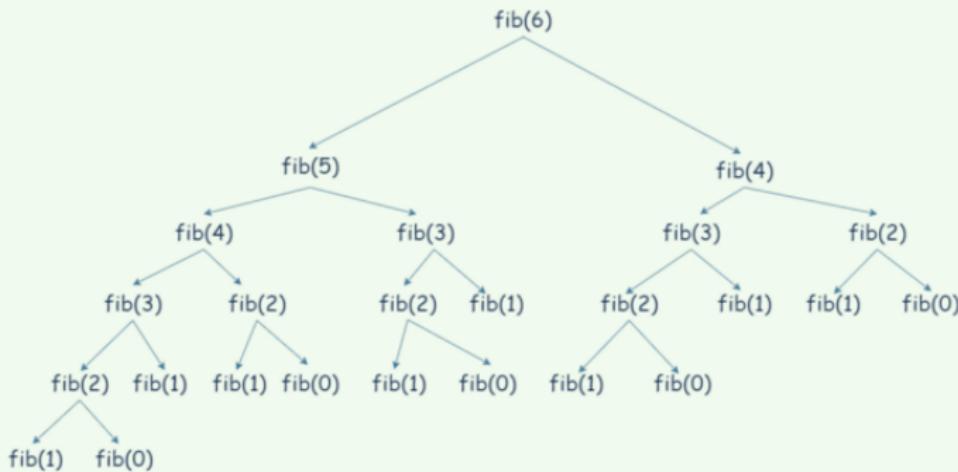
Memoization vs. Recursion

- In recursion, we also break a problem into similar subproblems. The difference is in recursion similar subproblems may be solved multiple times. But in dynamic programming, we keep track of already solved subproblems and don't solve them more than once.
- The problems that appear multiple times are called overlapping subproblems. To solve a problem in dynamic programming style we save the solutions of overlapping subproblems in a data structure. This is known as "Memoization".

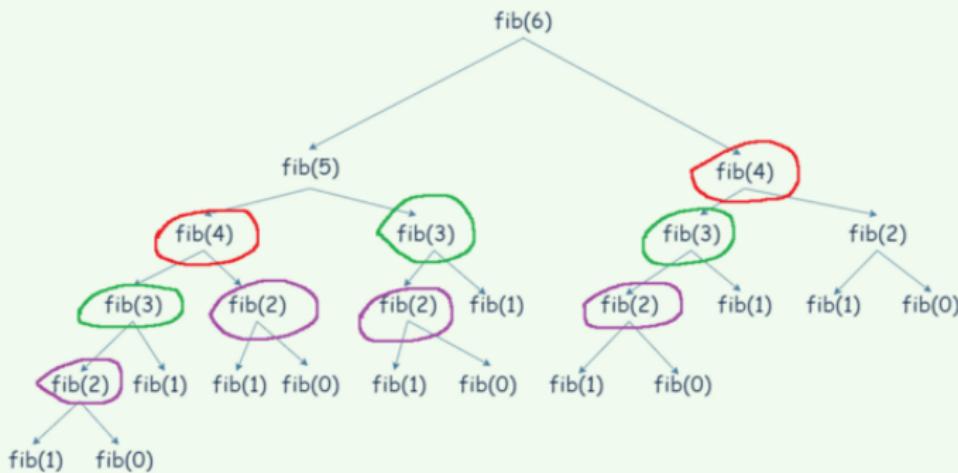
Overlapping Sub-Problems I

Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

Overlapping Sub-Problems II



Overlapping Sub-Problems III



Overlapping Sub-Problems IV

- We can see from above $\text{fib}(4)$ is being called 2 times, $\text{fib}(3)$ is being called 3 times and $\text{fib}(2)$ is being called 4 times. They are overlapping subproblems. We can ignore the base cases. We will make sure that the overlapping subproblems are solved only once using dynamic programming.

Dynamic Programming Approach

- We can store the results of previously solved subproblems in a data structure like a list. And the function fib() will check if a subproblem is already solved or not. If solved before we will not solve it again. Let's take a list memo[]]. Once a subproblem is solved we will store the solution in memo[]]. Saving the solutions like this is called memoization.

```
public class Fibonacci
{
    int fib(int n)
    {
        int f[] = new int[n+1];
        f[0] = 0;
        f[1] = 1;
        for (int i = 2; i <= n; i++)
            f[i] = f[i-1] + f[i-2];
        return f[n];
    }

    public static void main(String[] args)
    {
        Fibonacci f = new Fibonacci();
        int n = 9;
        System.out.println("Fibonacci number is" + " " + f.fib(n));
    }
}
```

Thank You

Matrix Chain Multiplication: L43

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

Matrix Multiplication I

Recalling Matrix Multiplication

Matrix: An $n \times m$ matrix $A = [a[i, j]]$ is a two-dimensional array

$$A = \begin{bmatrix} a[1, 1] & a[1, 2] & \cdots & a[1, m-1] & a[1, m] \\ a[2, 1] & a[2, 2] & \cdots & a[2, m-1] & a[2, m] \\ \vdots & \vdots & & \vdots & \vdots \\ a[n, 1] & a[n, 2] & \cdots & a[n, m-1] & a[n, m] \end{bmatrix},$$

which has n rows and m columns.

Example: The following is a 4×5 matrix:

$$\begin{bmatrix} 12 & 8 & 9 & 7 & 6 \\ 7 & 6 & 89 & 56 & 2 \\ 5 & 5 & 6 & 9 & 10 \\ 8 & 6 & 0 & -8 & -1 \end{bmatrix}.$$

Matrix Multiplication II

Recalling Matrix Multiplication

The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix given by

$$c[i, j] = \sum_{k=1}^q a[i, k]b[k, j]$$

for $1 \leq i \leq p$ and $1 \leq j \leq r$.

Example: If

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix},$$

then

$$C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}.$$

Matrix Multiplication III

Remarks on Matrix Multiplication

- If AB is defined, BA may **not** be defined.
- Quite possible that $AB \neq BA$.
- Multiplication is recursively defined by

$$\begin{aligned} A_1 A_2 A_3 \cdots A_{s-1} A_s \\ = A_1 (A_2 (A_3 \cdots (A_{s-1} A_s))). \end{aligned}$$

- Matrix multiplication is **associative**, e.g.,

$$A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3),$$

so parenthenization does not change result.

Matrix Multiplication IV

Direct Matrix multiplication AB

Given a $p \times q$ matrix A and a $q \times r$ matrix B , the direct way of multiplying $C = AB$ is to compute each

$$c[i, j] = \sum_{k=1}^q a[i, k]b[k, j]$$

for $1 \leq i \leq p$ and $1 \leq j \leq r$.

Complexity of Direct Matrix multiplication:

Note that C has pr entries and each entry takes $\Theta(q)$ time to compute so the total procedure takes $\Theta(pqr)$ time.

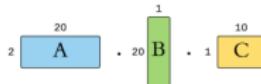
Matrix Chain Multiplication I

Problem

- Given a sequence of matrices, find the most efficient way to multiply these matrices together.
- The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.
- We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same.

Matrix Chain Multiplication II

For example, given the matrices with the following dimensions:



If we define a row of a matrix `A` as `rA` and the column as `cA`, then the total number of multiplications required to multiply `A` and `B` is `rA.cA.cB`.

- $A.(B.C) = 2*20*1 + 2*1*10 = 60$ multiplications
- $(A.B).C = 20*1*10 + 2*20*10 = 600$ multiplications

Solution

We will be discussing two solutions for the problem

1. Recursive Solution

2. Dynamic Programming

Matrix Chain Multiplication III

Example 1

```
Input: M[] = [10, 20, 30, 40, 30]
Output: 30000
Explanation:
Dimensions of M1 = 10 x 20
Dimensions of M2 = 20 x 30
Dimensions of M3 = 30 x 40
Dimensions of M4 = 40 x 30
First, multiply M1 and M2, cost = 10*20*30 = 6000
Second, multiply (Matrix obtained after multiplying M1 and M2) and M3 = 10 * 30 *
Third, multiply (Matrix obtained after multiplying M1, M2 and M3) and M4 = 10 *
Total Cost = 6000 + 12000 + 12000 = 30000
```

Example 2

```
Input: M[] = [10, 20, 30]
Output: 6000
Explanation: There are only two matrices of dimensions 10x20 and 20x30. So there i
```

Matrix Chain Multiplication IV

Given a $p \times q$ matrix A , a $q \times r$ matrix B and a $r \times s$ matrix C , then ABC can be computed in two ways $(AB)C$ and $A(BC)$:

The number of multiplications needed are:

$$\begin{aligned} \text{mult}[(AB)C] &= pqr + prs, \\ \text{mult}[A(BC)] &= qrs + pqs. \end{aligned}$$

When $p = 5, q = 4, r = 6$ and $s = 2$, then

$$\begin{aligned} \text{mult}[(AB)C] &= 180, \\ \text{mult}[A(BC)] &= 88. \end{aligned}$$

A big difference!

Implication: The multiplication “sequence” (parenthesization) is important!!

Matrix Chain Multiplication V

Why Order Matters

Suppose we have 4 matrices:

A: 30×1

B: 1×40

C: 40×10

D: 10×25

$((AB)(CD))$: requires 41,200 scalar multiplications

$(A((BC)D))$: requires 1400 scalar multiplications

Matrix Chain Multiplication VI

The Chain Matrix Multiplication Problem

Given

dimensions p_0, p_1, \dots, p_n

corresponding to matrix sequence A_1, A_2, \dots, A_n

where A_i has dimension $p_{i-1} \times p_i$,

determine the “multiplication sequence” that minimizes
the number of scalar multiplications in computing
 $A_1 A_2 \cdots A_n$. That is, determine how to parenthesize
the multiplications.

$$\begin{aligned}A_1 A_2 A_3 A_4 &= (A_1 A_2)(A_3 A_4) \\&= A_1(A_2(A_3 A_4)) = A_1((A_2 A_3) A_4) \\&= ((A_1 A_2) A_3)(A_4) = (A_1(A_2 A_3))(A_4)\end{aligned}$$

Exhaustive search: $\Omega(4^n/n^{3/2})$.

Question: Any better approach?

Yes – DP

Dynamic Programming Solution to MCM I

Developing a Dynamic Programming Algorithm

Step 1: Determine the structure of an optimal solution
(in this case, a parenthesization).

Decompose the problem into subproblems: For each pair $1 \leq i \leq j \leq n$, determine the multiplication sequence for $A_{i..j} = A_i A_{i+1} \cdots A_j$ that minimizes the number of multiplications.

Clearly, $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

Original Problem: determine sequence of multiplication for $A_{1..n}$.

Dynamic Programming Solution to MCM II

Developing a Dynamic Programming Algorithm

Step 1: Determine the structure of an optimal solution
(in this case, a parenthesization).

High-Level Parenthesization for $A_{i..j}$

For any optimal multiplication sequence, at the last step you are multiplying two matrices $A_{i..k}$ and $A_{k+1..j}$ for some k . That is,

$$A_{i..j} = (A_i \cdots A_k)(A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.$$

Example

$$A_{3..6} = (A_3(A_4A_5))(A_6) = A_{3..5}A_{6..6}.$$

Here $k = 5$.

Dynamic Programming Solution to MCM III

Developing a Dynamic Programming Algorithm

Step 1 – Continued: Thus the problem of determining the optimal sequence of multiplications is broken down into 2 questions:

- How do we decide where to split the chain (what is k)?

(Search all possible values of k)

- How do we parenthesize the subchains

$A_{i..k}$ and $A_{k+1..j}$?

(Problem has optimal substructure property that $A_{i..k}$ and $A_{k+1..j}$ must be optimal so we can apply the same procedure recursively)

Dynamic Programming Solution to MCM IV

Developing a Dynamic Programming Algorithm

Step 1 – Continued:

Optimal Substructure Property: If final “optimal” solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at final step then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in final optimal solution must also be optimal for the subproblems “standing alone”:

If parenthesisation of $A_{i..k}$ was not optimal we could replace it by a better parenthesisation and get a cheaper final solution, leading to a contradiction.

Similarly, if parenthesisation of $A_{k+1..j}$ was not optimal we could replace it by a better parenthesisation and get a cheaper final solution, also leading to a contradiction.

Dynamic Programming Solution to MCM V

Developing a Dynamic Programming Algorithm

Step 2: Recursively define the value of an optimal solution.

As with the 0-1 knapsack problem, we will store the solutions to the subproblems in an array.

For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$.
The optimum cost can be described by the following recursive definition.

Dynamic Programming Solution to MCM VI

Developing a Dynamic Programming Algorithm

Step 2: Recursively define the value of an optimal solution.

$$m[i, j] = \begin{cases} 0 & i = j, \\ \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j) & i < j \end{cases}$$

Proof: Any optimal sequence of multiplication for $A_{i..j}$ is equivalent to some choice of splitting

$$A_{i..j} = A_{i..k} A_{k+1..j}$$

for some k , where the sequences of multiplications for $A_{i..k}$ and $A_{k+1..j}$ also are optimal. Hence

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j.$$

Dynamic Programming Solution to MCM VII

Developing a Dynamic Programming Algorithm

Step 2 – Continued: We know that, for some k

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j.$$

We don't know what k is, though

But, there are only $j - i$ possible values of k so we can check them all and find the one which returns a smallest cost.

Therefore

$$m[i, j] = \begin{cases} 0 & i = j, \\ \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j) & i < j \end{cases}$$

Dynamic Programming Solution to MCM VIII

Developing a Dynamic Programming Algorithm

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Our Table: $m[1..n, 1..n]$.

$m[i, j]$ only defined for $i \leq j$.

The important point is that when we use the equation

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$

to calculate $m[i, j]$ we must have already evaluated $m[i, k]$ and $m[k + 1, j]$. For both cases, the corresponding length of the matrix-chain are both less than $j - i + 1$. Hence, the algorithm should fill the table in increasing order of the length of the matrix-chain.

That is, we calculate in the order

$m[1, 2], m[2, 3], m[3, 4], \dots, m[n - 3, n - 2], m[n - 2, n - 1], m[n - 1, n]$

$m[1, 3], m[2, 4], m[3, 5], \dots, m[n - 3, n - 1], m[n - 2, n]$

$m[1, 4], m[2, 5], m[3, 6], \dots, m[n - 3, n]$

\vdots

$m[1, n - 1], m[2, n]$

$m[1, n]$

Pseudocode

The Dynamic Programming Algorithm

```
Matrix-Chain( $p, n$ )
{   for ( $i = 1$  to  $n$ )  $m[i, i] = 0$ ;
    for ( $l = 2$  to  $n$ )
    {
        for ( $i = 1$  to  $n - l + 1$ )
        {
             $j = i + l - 1$ ;
             $m[i, j] = \infty$ ;
            for ( $k = i$  to  $j - 1$ )
            {
                 $q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]$ ;
                if ( $q < m[i, j]$ )
                {
                     $m[i, j] = q$ ;
                     $s[i, j] = k$ ;
                }
            }
        }
    }
    return  $m$  and  $s$ ; (Optimum in  $m[1, n]$ )
}
```

Complexity: The loops are nested three deep.

Each loop index takes on $\leq n$ values.

Hence the time complexity is $O(n^3)$. Space complexity $\Theta(n^2)$.

Example

Constructing an Optimal Solution: Compute $A_{1..n}$

Example of Constructing an Optimal Solution:

Compute $A_{1..6}$.

Consider the example earlier, where $n = 6$. Assume that the array $s[1..6, 1..6]$ has been computed. The multiplication sequence is recovered as follows.

Mult($A, s, 1, 6$), $s[1, 6] = 3$, $(A_1 A_2 A_3)(A_4 A_5 A_6)$
Mult($A, s, 1, 3$), $s[1, 3] = 1$, $((A_1)(A_2 A_3))(A_4 A_5 A_6)$
Mult($A, s, 4, 6$), $s[4, 6] = 5$, $((A_1)(A_2 A_3))((A_4 A_5)(A_6))$
Mult($A, s, 2, 3$), $s[2, 3] = 2$, $((A_1)((A_2)(A_3)))((A_4 A_5)(A_6))$
Mult($A, s, 4, 5$), $s[4, 5] = 4$, $((A_1)((A_2)(A_3)))(((A_4)(A_5))(A_6))$

Hence the product is computed as follows

$$(A_1(A_2 A_3))((A_4 A_5)A_6).$$

Thank You

String Matching using Edit Distance: L44

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 String Matching with Edit distance

String Matching using Edit Distance I

Definition

- In computational linguistics and computer science, edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other.
- Edit distances find applications in natural language processing, where automatic spelling correction can determine candidate corrections for a misspelled word by selecting words from a dictionary that have a low distance to the word in question. In bioinformatics, it can be used to quantify the similarity of DNA sequences, which can be viewed as strings of the letters A, C, G and T.

String Matching using Edit Distance II

>gb|AC115706.7| Mus musculus chromosome 8, clone RP23-382B3, complete sequence

Query 1650	gtgtgtgtgggtcacatttgcgtgtgtgcgcctgtgtgtgggtgcctgtgtgt	1709
Sbjct 56838	GTGTGTGTTGGAAGTGGAGTCATCTGTGTGCACATGTGTGTGCA--TGCATGCATGTGT	56895
Query 1710	gtg-gggcacatttgcgtgtgtgcgcctgtgtgtgggtgcacatttgcgtgtgc	1768
Sbjct 56896	GTCGGGGCA-----TGCATGTCTGTGTGCATGTGTGTGTGTGCAT--GTGTGAGTAC	56947
Query 1769	cgtgtgtgtgtgcctgtgtgtgggtgcacatttgcgtgtgtgcgcctgtgtgc	1828
Sbjct 56948	CITGTTGTTGATGCTGTATGTGTGTGTGCATGTGTGTAGGTGTGTATATGTGTAAAGT	57007
Query 1829	gggtgcacatttgcgtgtgtgcgcctgtgtgtgggtgcacatttgcgtgtgt	1888
Sbjct 57008	T-----CATCTGTGTATGTGTG--TGTGAGAGTCATGCA---TGTGTGTGTGAGT	57055
Query 1889	gcctgtgtgt--gtgggtgcacatttgcgtgtgtgcgcctgt--tgtgt--gggtgcac	1942
Sbjct 57056	TCATCTGTGTCACTGTATGCTTATGGGTATAACT-TAAGTGTGCATGTGTAACTGTGTTC	57114
Query 1943	atttgtgtgtgtgtgcctgtgtgtgggtgcacatttgcgtgtgtgcgcctgtgtgt	2002
Sbjct 57115	ATCTGTGTATGTGTGTG--TGTGTGAGTTAGTCA---TCTGTGTGTGAGAGTGTGTGA	57168
Query 2003	gtgcacatttgcgtgtgtgcgcctgtgtgtgcgcctgtgtgtgggtgcacatttgc	2062
Sbjct 57169	G---CTCACATGTGTGAGTTCACTGTATGAGTG--TGTGTATGTGTGTACAAATGA	57224
Query 2063	gtgtgtgtgtgcctgtgtgtgggtgcacatttgcgtgtgtgcgcctgtgtgt	2122
Sbjct 57225	GTTCATCTGTGTGCATGTGTGTG-----TTAACGTGTTCATCTG--TGTGGTGT	57274

String Matching using Edit Distance III

prin-ciple
|||| |||xx
prinncipal
(1 gap, 2 mm)

prin-cip-le
|||| ||| |
prinncipal-
(3 gaps, 0 mm)

misspell
||| ||||
mis-pell
(1 gap)

prehistoric
|||||||
---historic
(3 gaps)

aa-bb-ccaabb
|x || | | |
ababbbbc-a-b-
(5 gaps, 1 mm)

al-go-rithm-
|| xx ||x |
alKhwariz-mi
(4 gaps, 3 mm)

String Matching using Edit Distance IV

Given two strings str1 and str2 and below operations that can be performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.

Examples:

Input: str1 = "geek", str2 = "gesek"

Output: 1

We can convert str1 into str2 by inserting a 's'.

Input: str1 = "cat", str2 = "cut"

Output: 1

We can convert str1 into str2 by replacing 'a' with 'u'.

Input: str1 = "sunday", str2 = "saturday"

Output: 3

Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations.

Replace 'n' with 'r', insert t, insert a

Problem I

What are the subproblems in this case?

- The idea is process all characters one by one starting from either from left or right sides of both strings.
- Let us traverse from right corner, there are two possibilities for every pair of character being traversed.
- m: Length of str1 (first string) n: Length of str2 (second string)
- If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
- Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
 - Insert: Recur for m and n-1

Problem II

- Remove: Recur for $m-1$ and n
- Replace: Recur for $m-1$ and $n-1$

Problem III

The Simplest String Comparison Problem

Given: Two strings

$$a = a_1 a_2 a_3 a_4 \dots a_m$$

$$b = b_1 b_2 b_3 b_4 \dots b_n$$

where a_i, b_i are letters from some alphabet like {A,C,G,T}.

Compute how similar the two strings are.

What do we mean by “similar”?

Edit distance between strings a and b = the smallest number of the following operations that are needed to transform a into b :

- mutate (replace) a character
- delete a character
- insert a character

riddle $\xrightarrow{\text{delete}}$ ridle $\xrightarrow{\text{mutate}}$ riple $\xrightarrow{\text{insert}}$ triple



Recursive Exponential Time Solution

```

class EDIST {
    static int min(int x, int y, int z)
    {
        if (x <= y && x <= z)
            return x;
        if (y <= x && y <= z)
            return y;
        else
            return z;
    }

    static int editDist(String str1, String str2, int m,
                        int n)
    {
        // If first string is empty, the only option is to
        // insert all characters of second string into first
        if (m == 0)
            return n;

        // If second string is empty, the only option is to
        // remove all characters of first string
        if (n == 0)
            return m;

        // If last characters of two strings are same,
        // nothing much to do. Ignore last characters and
        // get count for remaining strings.
        if (str1.charAt(m - 1) == str2.charAt(n - 1))
            return editDist(str1, str2, m - 1, n - 1);

        // If last characters are not same, consider all
        // three operations on last character of first
        // string, recursively compute minimum cost for all
        // three operations and take minimum of three
        // values.
        return 1
            + min(editDist(str1, str2, m, n - 1), // Insert
                  editDist(str1, str2, m - 1, n), // Remove
                  editDist(str1, str2, m - 1,
                           n - 1)) // Replace
    }
}

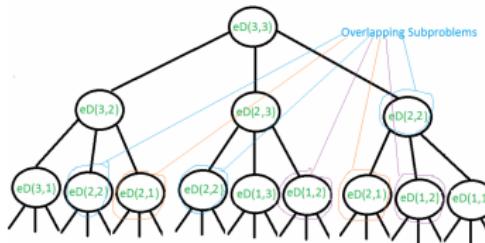
// Driver Code
public static void main(String args[])
{
    String str1 = "sunday";
    String str2 = "saturday";

    System.out.println(editDist(
        str1, str2, str1.length(), str2.length()));
}

```

Exponential Run Time

- The time complexity of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.
- We can see that many subproblems are solved, again and again, for example, $eD(2, 2)$ is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subproblems.



Worst case recursion tree when $m = 3, n = 3$.
Worst case example str1="abc" str2="xyz"

The DP Approach I

Algorithm for Computing Edit Distance

Consider the last characters of each string:

$$\begin{aligned}a &= a_1 a_2 a_3 a_4 \dots a_m \\b &= b_1 b_2 b_3 b_4 \dots b_n\end{aligned}$$

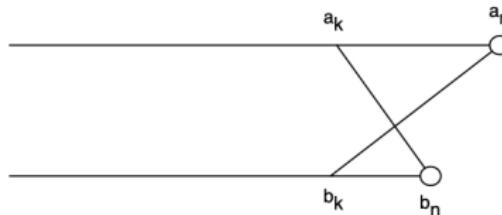
One of these possibilities must hold:

1. (a_m, b_n) are matched to each other
2. a_m is not matched at all
3. b_n is not matched at all
4. a_m is matched to some $b_j (j \neq n)$ and b_n is matched to some $a_k (k \neq m)$.

The DP Approach II

No Crossing Rule Forbids #4

4. a_m is matched to some b_j ($j \neq n$) and b_n is matched to some a_k ($k \neq m$).



So, the only possibilities for what happens to the last characters are:

1. (a_m, b_n) are matched to each other
2. a_m is not matched at all
3. b_n is not matched at all

The DP Approach III

Turn the 3 possibilities into 3 cases of a recurrence:

$$OPT(i, j) = \min \begin{cases} \text{cost}(a_i, b_j) + OPT(i - 1, j - 1) & \text{match } a_i, b_j \\ \text{gap} + OPT(i - 1, j) & a_i \text{ is not matched} \\ \text{gap} + OPT(i, j - 1) & b_j \text{ is not matched} \end{cases}$$

↑
Written in terms of
the costs of smaller
problems

↑
Cost of the optimal
alignment between
 $a_1 \dots a_i$ and $b_1 \dots b_j$

Key: we don't know which of the 3 possibilities is the right one, so we try them all.

Base case: $OPT(i, 0) = i \times \text{gap}$ and $OPT(0, j) = j \times \text{gap}$.

(Aligning i characters to 0 characters must use i gaps.)

The DP Approach IV

Edit Distance Computation

```
EditDistance(X,Y):
    For i = 1,...,m: A[i,0] = i*gap
    For j = 1,...,n: A[0,j] = j*gap

    For i = 1,...,m:
        For j = 1,...,n:
            A[i,j] = min(
                cost(a[i],b[j]) + A[i-1,j-1],
                gap + A[i-1,j],
                gap + A[i,j-1]
            )
    EndFor
EndFor
Return A[m,n]
```

Memoization

```

class EDIST {
    static int min(int x, int y, int z)
    {
        if (x <= y && x <= z)
            return x;
        if (y <= x && y <= z)
            return y;
        else
            return z;
    }

    static int editDistDP(String str1, String str2, int m,
                          int n)
    {
        // Create a table to store results of subproblems
        int dp[][] = new int[m + 1][n + 1];

        // Fill d[][] in bottom up manner
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                // If first string is empty, only option is
                // to insert all characters of second string
                if (i == 0)
                    dp[i][j] = j; // Min. operations = j

                // If second string is empty, only option is
                // to remove all characters of second string
                else if (j == 0)
                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last
                // char and recur for remaining string
                else if (str1.charAt(i - 1)
                         == str2.charAt(j - 1))
                    dp[i][j] = dp[i - 1][j - 1];

                // If the last character is different,
                // consider all possibilities and find the
                // minimum
                else
                    dp[i][j] = 1
                            + min(dp[i][j - 1], // Insert
                                  dp[i - 1][j], // Remove
                                  dp[i - 1][j - 1]); // Replace
            }
        }

        return dp[m][n];
    }

    // Driver Code
    public static void main(String args[])
    {
        String str1 = "sunday";
        String str2 = "saturday";
        System.out.println(editDistDP(
            str1, str2, str1.length(), str2.length()));
    }
}

```

Analysis

- Time Complexity: $O(m \times n)$ Auxiliary Space: $O(m \times n)$
- Space Complex Solution: In the above-given method we require $O(m \times n)$ space. This will not be suitable if the length of strings is greater than 2000 as it can only create 2D array of 2000×2000 . To fill a row in DP array we require only one row the upper row. For example, if we are filling the $i = 10$ rows in DP array we require only values of 9th row. So we simply create a DP array of $2 \times \text{str1 length}$.

Implementation

Implement an algorithm for string matching using edit distance in $O(m*n)$.

Thank You

Longest Common Subsequence: L45

Dr. Suchintan Mishra

Department of Computer Science and Engineering
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Text Books

Text book and References:

- (T1) **Algorithm Design by Jon Kleinberg and Eva Tardos, Pearson Publication**
- (R1) The Algorithm Design Manual by Steven Skiena, Springer Publication
- (R2) Introduction to Algorithms by CLRS, PHI Publication

Outline

1 Longest Common Subsequence

Longest Common Subsequence I

Definition

- The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences).
- It differs from the longest common substring problem: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences.
- The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff utility, and has applications in computational linguistics and bioinformatics.
- It is also widely used by revision control systems such as Git for reconciling multiple changes made to a revision-controlled collection of files.

Longest Common Subsequence II

For example, consider the sequences (ABCD) and (ACBAD). They have 5 length-2 common subsequences: (AB), (AC), (AD), (BD), and (CD); 2 length-3 common subsequences: (ABD) and (ACD); and no longer common subsequences. So (ABD) and (ACD) are their longest common subsequences.

Longest Common Subsequence III

Complexity

- In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n , i.e., find the number of subsequences with lengths ranging from $1, 2, \dots, n - 1$.
- Recall from theory of permutation and combination that number of combinations with 1 element are n_{C_1} . Number of combinations with 2 elements are n_{C_2} and so forth and so on. We know that $n_{C_0} + n_{C_1} + n_{C_2} + \dots + n_{C_n} = 2^n$. So a string of length n has $2n-1$ different possible subsequences since we do not consider the subsequence with length 0.
- This implies that the time complexity of the brute force approach will be $O(n * 2^n)$. Note that it takes $O(n)$ time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

Examples

Longest Common Subsequence

A **subsequence** of a string S , is a set of characters that appear in left-to-right order, but not necessarily consecutively.

Example

ACTTGCG

- *ACT* , *ATTC* , *T* , *ACTTG* are all subsequences.
- *TTA* is not a subsequence

A **common subsequence** of two strings is a subsequence that appears in both strings. A **longest common subsequence** is a common subsequence of maximal length.

Example

$$\begin{aligned}S_1 &= AAACCGTGAGTTATT\color{red}{CGT}\color{black}TAGAA \\S_2 &= CACCCCTAAGGTACCTTTGGTTC\end{aligned}$$

LCS is

ACCTAGTACTTTG

Optimal Substructure I

The LCS problem has an optimal substructure: the problem can be broken down into smaller, simpler subproblems, which can, in turn, be broken down into simpler subproblems, and so on, until, finally, the solution becomes trivial. LCS in particular has overlapping subproblems: the solutions to high-level subproblems often reuse solutions to lower level subproblems.

- Let the input sequences be $X[0..m-1]$ and $Y[0..n-1]$ of lengths m and n respectively. And let $L(X[0..m-1], Y[0..n-1])$ be the length of LCS of the two sequences X and Y . Following is the recursive definition of $L(X[0..m-1], Y[0..n-1])$.
 - If last characters of both sequences match (or $X[m-1] == Y[n-1]$) then $L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])$
 - If last characters of both sequences do not match (or $X[m-1] != Y[n-1]$) then $L(X[0..m-1], Y[0..n-1]) = \text{MAX} (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))$

Optimal Substructure II

Example 1: Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

$$L(\text{“AGGTAB”, “GXTXAYB”}) = 1 + L(\text{“AGGTA”, “GXTXAY”})$$

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

Optimal Substructure III

Example 2: Consider the input strings “ABCDGH” and “AEDFHR”.
Last characters do not match for the strings.
So length of LCS can be written as:

$$L(\text{"ABCDGH"}, \text{"AEDFHR"}) = \text{MAX} (L(\text{"ABCDG"}, \text{"AEDFHR"}), L(\text{"ABCDGH"}, \text{"AEDFH"}))$$

Recursive Implementation of Longest Common Subsequence I

Time complexity of the above naive recursive approach is $O(2^n)$ in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Longest Common Subsequence
oooooooo●oooooooo

Recursive Implementation of Longest Common Subsequence II

```
public class LongestCommonSubsequence
{
    /* Returns length of LCS for X[0..m-1], Y[0..n-1] */
    int lcs( char[] X, char[] Y, int m, int n )
    {
        if (m == 0 || n == 0)
            return 0;
        if (X[m-1] == Y[n-1])
            return 1 + lcs(X, Y, m-1, n-1);
        else
            return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
    }

    /* Utility function to get max of 2 integers */
    int max(int a, int b)
    {
        return (a > b)? a : b;
    }

    public static void main(String[] args)
    {
        LongestCommonSubsequence lcs = new LongestCommonSubsequence();
        String s1 = "AGGTAB";
        String s2 = "GXTXAYB";

        char[] X=s1.toCharArray();
        char[] Y=s2.toCharArray();
        int m = X.length;
        int n = Y.length;

        System.out.println("Length of LCS is" + " " +
                           lcs.lcs( X, Y, m, n ) );
    }
}
```

Dynamic Programming based Approach I

```
public class LongestCommonSubsequence
{
    /* Returns length of LCS for X[0..m-1], Y[0..n-1] */
    int lcs( char[] X, char[] Y, int m, int n )
    {
        int L[][] = new int[m+1][n+1];

        /* Following steps build L[m+1][n+1] in bottom up fashion. Note
           that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
        for (int i=0; i<=m; i++)
        {
            for (int j=0; j<=n; j++)
            {
                if (i == 0 || j == 0)
                    L[i][j] = 0;
                else if (X[i-1] == Y[j-1])
                    L[i][j] = L[i-1][j-1] + 1;
                else
                    L[i][j] = max(L[i-1][j], L[i][j-1]);
            }
        }
        return L[m][n];
    }

    /* Utility function to get max of 2 integers */
    int max(int a, int b)
    {
        return (a > b)? a : b;
    }

    public static void main(String[] args)
    {
        LongestCommonSubsequence lcs = new LongestCommonSubsequence();
        String s1 = "AGGTAB";
        String s2 = "GXTXAYB";

        char[] X=s1.toCharArray();
        char[] Y=s2.toCharArray();
        int m = X.length;
        int n = Y.length;

        System.out.println("Length of LCS is" + " " +
                           lcs.lcs( X, Y, m, n ) );
    }
}
```

Dynamic Programming based Approach II

Time Complexity of the above implementation is $O(mn)$ which is much better than the worst-case time complexity of Naive Recursive implementation.

The above algorithm/code returns only length of LCS.

Example 1

Using Dynamic Programming to find the LCS

Let us take two sequences:

X A | C | A | D | B

The first sequence

Y C | B | D | A

Second Sequence

Example II

1. Create a table of dimension $n+1 \times m+1$ where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

		C	B	D	A	
A	0	0	0	0	0	
	0					
	0					
	0					
	0					
	0					
Initialise a table						

Example III

2. Fill each cell of the table using the following logic.
3. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.

	C	B	D	A
A	0	0	0	0
C	0			
A	0			
D	0			
B	0			

Fill the values

5. Step 2 is repeated until the table is filled.

Example 1

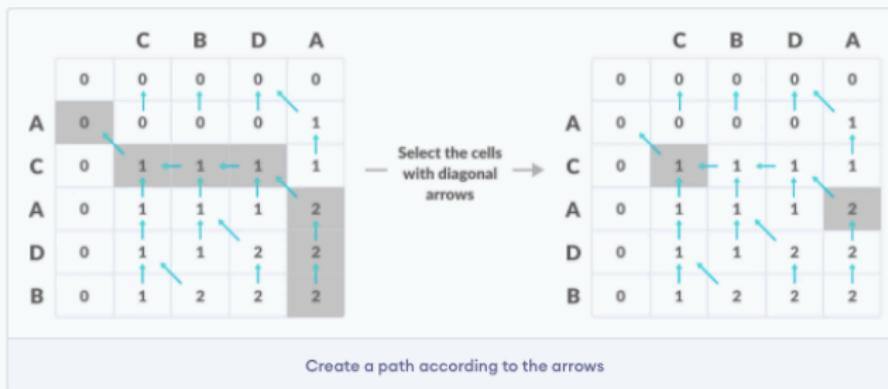
5. Step 2 is repeated until the table is filled.

	C	B	D	A
C	0	0	0	0
A	0	0	0	1
C	0	1	1	1
A	0	1	1	2
D	0	1	1	2
B	0	1	2	2

Fill all the values

Example II

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Thus, the longest common subsequence is [CD].

Thank You