Introduction to Algorithm Design

Lecture Notes 1

Introduction

Algorithms are “methods for solving problems which are suited for computed implementation..” [Sedgewick]

An algorithm is “a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.” [Aho, Hopcroft, & Ulman]

Introduction

"*Algorithmics* [defined as the study of algorithms -- A.L.] is more than a branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant to most of science, business, and technology." [David Harel, "Algorithmics: The Spirit of Computing"]

Syllabus

Instructor	Dr. F. Erdoğan Sevilgen
Email	sevilgen@gyte.edu.tr
Office	BM 206
Office Hour	Wednesdays 14:00-16:00
Phone	0262-605 2210
Teaching Assistant	Salih Cebeci
Email	scebeci@bilmuh.gyte.edu.tr
Office	
Office Hour	
Phone	0262-605

Syllabus

Web Page

- On the course web

Textbook

- Anany Levitin. Introduction to The Design and Analysis of Algorithms, Addison Wesley, second edition.

Other Books

- Cormen, Leiserton, Rivest, Introduction to Algorithms, MIT Press, 2001.
- Horowitz, Sahni, Rajasekaran, Computer Algorithms, Computer Science Press, 1998.
- Sahni, Data Structures, Algorithms, and Applications in C++, McGraw-Hill, 1998.
- Weiss, Data Structures and algorithm and Algorithm Analysis in C, Addison Wesley, 1997.

Prerequisites:

- BİL 222 Data Structures and Algorithms
- BİL 211 Discrete Mathematics

Syllabus-Catalog Description:

- Basic definitions and data structures.
- Introduction to analysis of algorithms.
- Standard algorithm design techniques;
  - divide-and-conquer,
  - greedy,
  - dynamic programming,
  - branch-and-bound,
  - backtracking,
  - etc.
- Basic algorithms;
  - sorting and searching,
  - graph algorithms,
  - etc.
- Introduction to complexity classes.

Syllabus - Educational Objectives:

This course aims to introduce basic algorithms and algorithm design techniques which can be used in designing solutions to real life problems. After this course, you

- will be able to design a new algorithms for a problem using the methods discussed in the class,
- will be able to analyze an algorithm with respect to various performance criteria such as memory use and running time,
- will be able to choose the most suitable algorithm for a problem to be solved,
- will be able to implement an algorithm efficiently.

Syllabus - Attendance Policy:

- Class attendance is advised but will not be a part of your final grade.
- Attendance for tests and the final exam is mandatory. If it is impossible for you to be present for a scheduled midterm, you must let me know BEFORE the test, so a make-up test can be scheduled.
- Please be considerate of your classmates during class.
  - Students are expected to show courtesy and respect toward their classmates.
  - Please do not carry on side discussions with other students during lecture time
  - When you have a question, please raise your hand and ask the question so that everyone may benefit from it.
  - Please try to make sure that your cellular phone and/or pager does not interrupt during lecture time, and especially during exams.

Syllabus - Assessment:

Homework	**10%**
2 Midterm Exams	**50%**
Final Exam	**40%**

Syllabus - Homeworks:

- Homework assignments and their due dates will be announced through the class web page. Late homework assignments have a **one time 30% penalty** and will not be accepted more than **three days** late.
- Homework assignments will be graded by the Teaching Assistant on the basis of correctness, quality of design and presentation. Homework **grades** and their **solutions** will be posted on the web page **in a week**.
- Unless stated otherwise, all homework assignments are **individual** assignments and are expected to be your own work. **TAKE PRIDE IN THE WORK YOU DO!!! DON'T CHEAT**. You may engage in general discussions about the problem, but sharing the solution of a problem will be considered as cheating and will be dealt with accordingly (**all parties will get -100**).
- You should submit **at least 70%** of the homework assignments and collect **at least 40 out of 100 on average**. Otherwise you will not be able to take the final exam, in other words, you will get the grade of VF.

ROAD MAP

- **Mathematical Background**
- **Fundamental Data Structures**
  - **Linear Data Structures**
  - **Graphs**
  - **Trees**
- What is an algorithm ?
- Analysis of algorithms
- Different Problems and their analysis
- Running time functions

Mathematical Background

- Functions
- Logarithm
- Summation
- Probability
- Asymptotic Notations
- Recursion
  - Recurrence equation

Fundamental Data Structures

- A ***data structure*** is a particular scheme of organizing related data items
  - Linear Data Structures
    - Array
    - Linked list
    - Stack
    - Queue
    - Prioritiy Queue
  - Graphs
  - Trees

Linear Data Structures

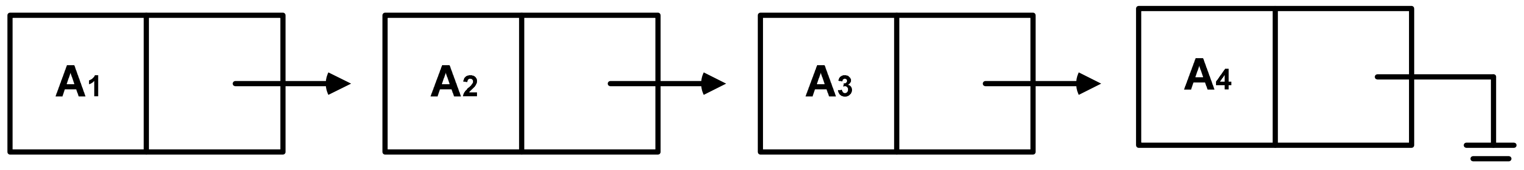
- **Array**
  - An *array* is a sequence of *n* items of the same data type that are stored contiguously in computer memory
  - Array is accessible by specifying a value of the array's *index*

Item[0]	*Item[1]*	…	*Item[n-1]*

Array of *n* elements

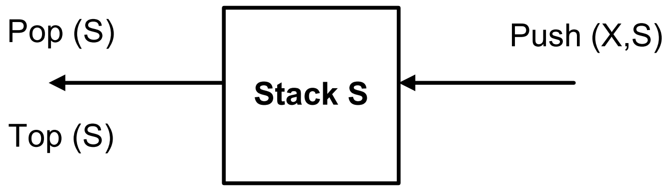
Linear Data Structures

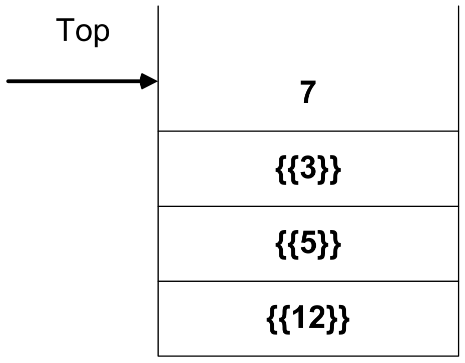
- **Linked List**
  - A *linked list* is a sequence of zero or more elements called *nodes*
  - Each node contains two kinds of information :
    - some data
    - one or more linkes called *pointers* to other nodes of the linked list



Linear Data Structures

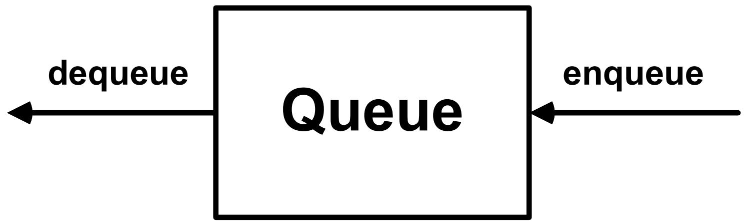
- **Stack**
  - A *stack* is a list in which insertions and deletions can be done only at the end (called *top)*
  - Last In First Out (LIFO)





Linear Data Structures

- **Queue**
  - A *queue* is a list whose elements are deleted from one end of the structure, called *front*
    - *dequeue operation*
  - New elements are added to the other end of the queue, called *rear*
    - *enqueue operation*
  - First In First Out (FIFO)



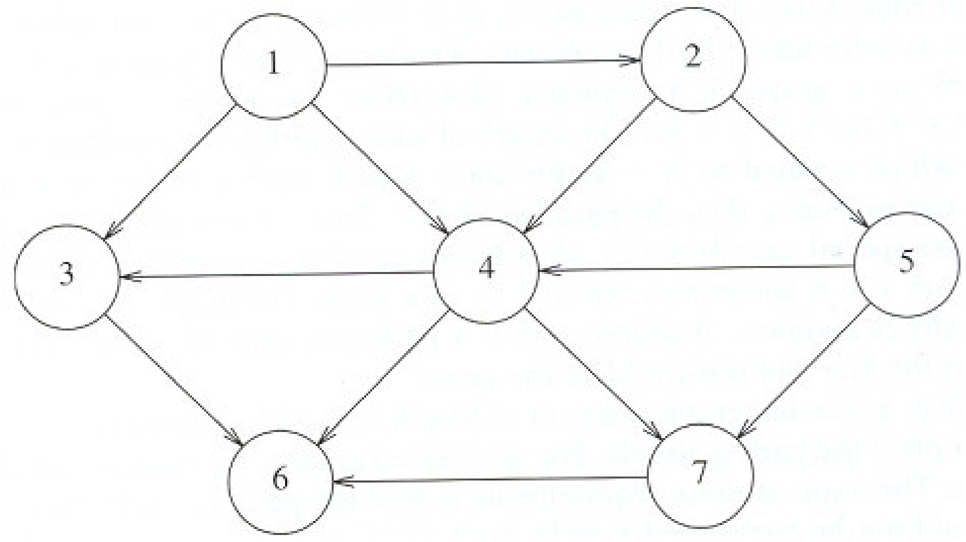
Linear Data Structures

- **Priority Queue**
  - A *priority queue* is a collection of data items from a totally ordered universe
    - e.g. integer or real numbers
  - Requires a selection of an item of the highest priority among a dynamically changing set of candidates
  - Principal operations:
    - Insert → adding a new element
    - Delete → deleting largest/smallest element
    - Search → find largest/smallest element
  - A better implementations of a priority queue is based on an ingenious data structure called ***heap***

Graphs

- A *graph* is a tuple *G=(V,E)*
  - V : set of *vertices* or *nodes*
  - E : set of *edges*
    - each edge is a pair (*v*,*w*), where *v*,*w* Є V
  - If the pairs are ordered, then the graph is *directed*
    - directed graphs are sometimes refered as *digraphs*
  - If the pairs are unordered, then the graph is *undirected*
  - Vertex *w* is adjacent to *v* iff *(v,w)* Є *E*

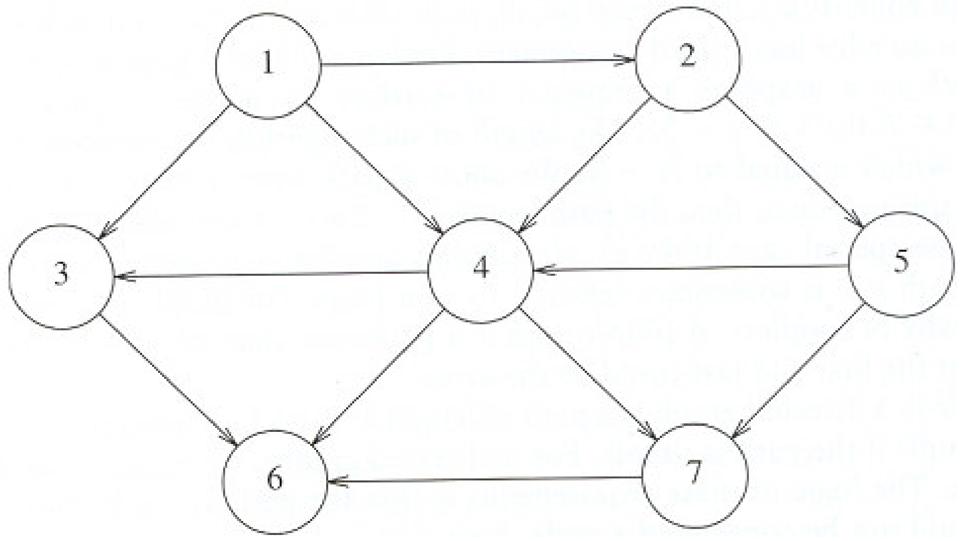
Graphs



A graph with 7 vertices and 12 edges

Graphs

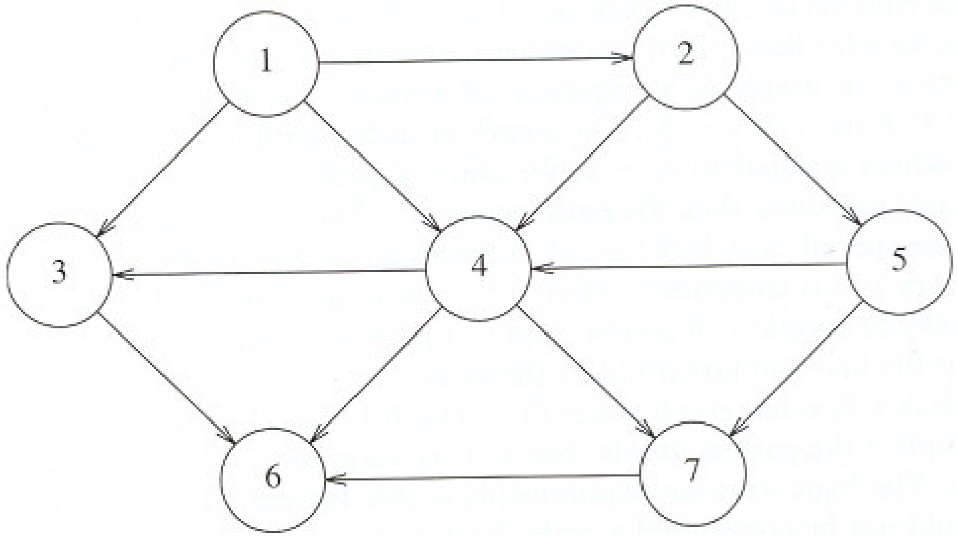
- A *path* is a sequence of vertices from *v* to *w*
- If all edges in a path are distinct the path is said to be *simple*
- *Lenght* is the total number of edges in a path
- A *cycle* is a path with *length* ≥ *1* where *u*=*w*
- A graph is *acyclic* if it has no cycles



A graph with 7 vertices and 12 edges

Graphs

- A graph with every pair of its vertices connected by an edge is called *complete*
- A graph with relatively few possible edges missing is called *dense*
- A graph with few edges relative to the number of its vertices is called *sparce*
- A graph is *connected* if for every pair of vertices *u* and *v* there is a path from *u* to *v*



A graph with 7 vertices and 12 edges

Graph Representations

Adjacency Matrix Representation

- It is *n-by-n* boolean matrix with one row and one column for each of the graph's vertices
  - *ith* row and *jth* column is equal to 1 if there is an edge from the *ith* vertex to the *jth* vertex
  - *ith* row and *jth* column is equal to 0 if there is no such edge

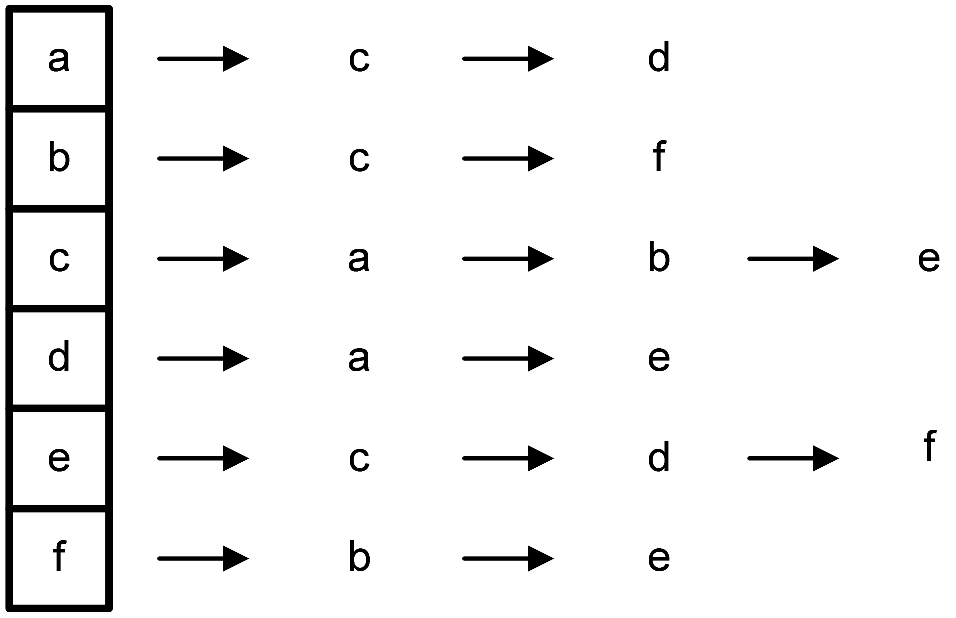
$$\begin{array}{c|cccccc}
 & a & b & c & d & e & f \\
a & 0 & 0 & 1 & 1 & 0 & 0 \\
b & 0 & 0 & 1 & 0 & 0 & 1 \\
c & 1 & 1 & 0 & 0 & 1 & 0 \\
d & 1 & 0 & 0 & 0 & 1 & 0 \\
e & 0 & 0 & 1 & 1 & 0 & 1 \\
f & 0 & 1 & 0 & 0 & 1 & 0
\end{array}$$

Graph Representations

Adjacency List Representation

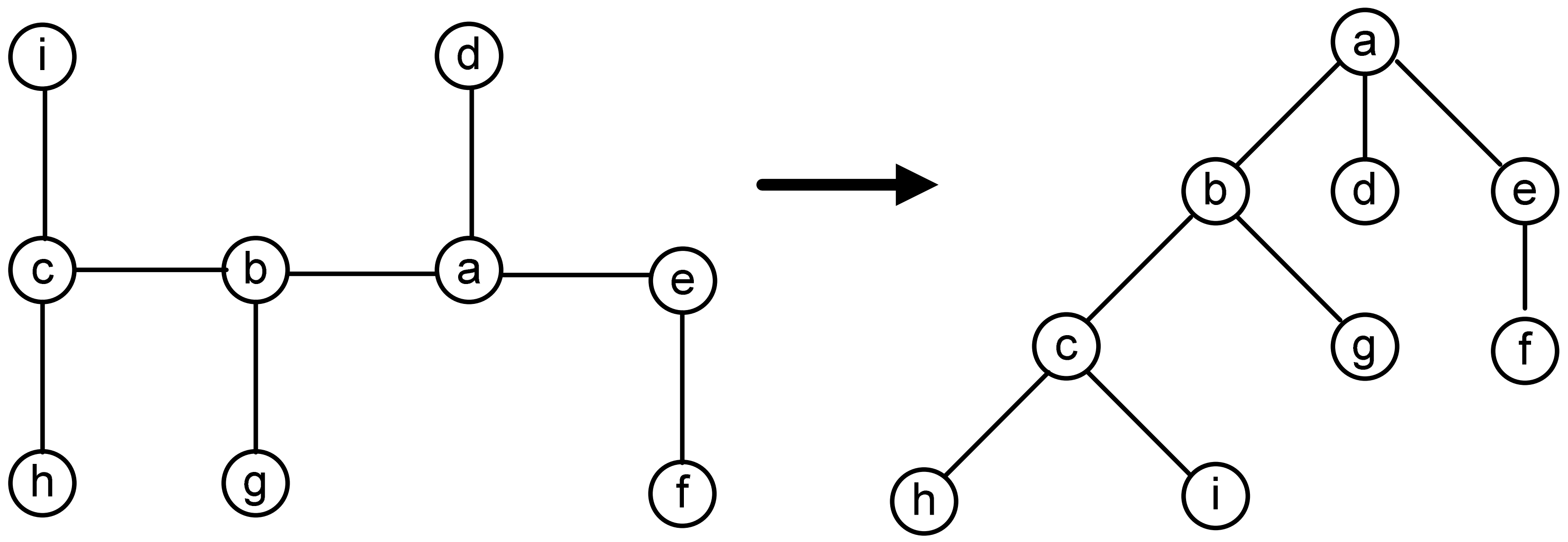
- Is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex

- If the graph is not dense (is *sparse*) adjacency list representation is a better solution



Trees

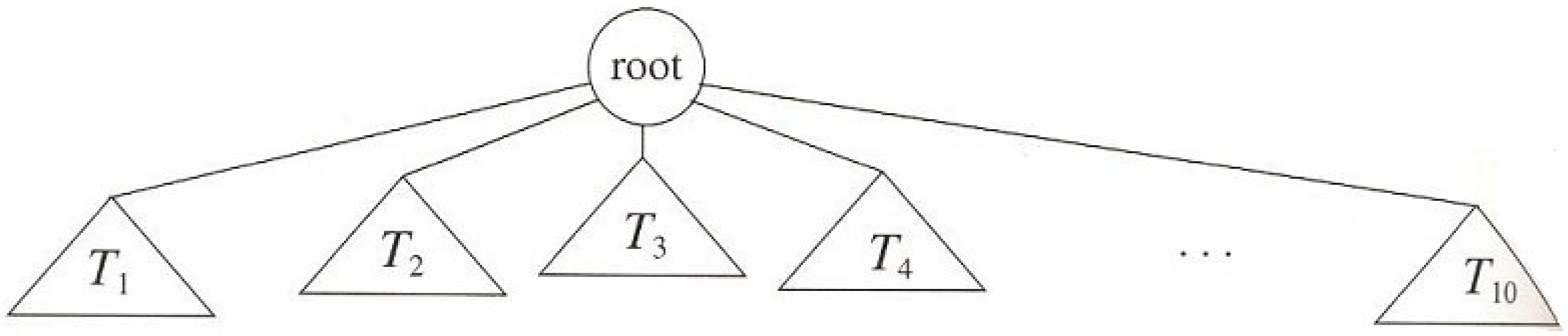
- A tree is a connected acyclic graph
  - *rooted tree*
    - Specialized node caled root



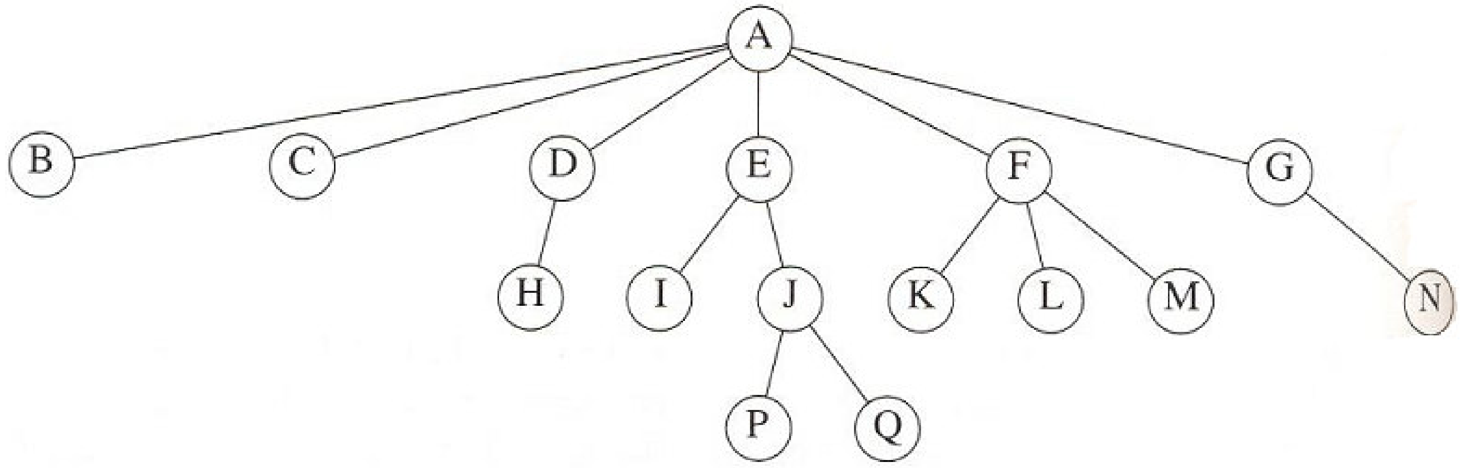
Trees

Recursive Definition of Rooted Trees:

- **Tree** is a collection of **nodes**
  - A tree can be empty
  - A tree contains zero or more **subtrees** T_1, T_2,… T_k connected to a **root node** by **edges**

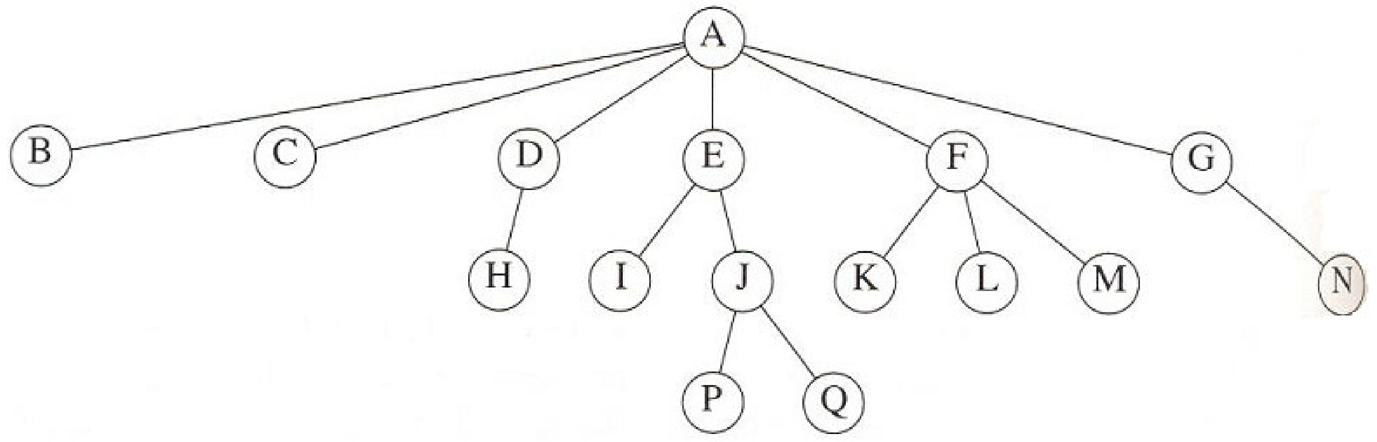


Trees - Terminology



Family Tree Terminology

- child → F is child of A
- parent → A is the parent of F
  - each node is connected to a parent except the root
- sibling → nodes with same parents (K, L, M)
- leaf → nodes with no children (P, Q)
- Ancestor / Descendant



- **Path :** a sequence of nodes n_1, n_2, … n_k, where n_i is the parent of n_i+1 $1 \leq i < k$
- **Lenght :** number of edges on the path (k-1)
- **Depth :** depth of n_i is the lenght of unique path from the root to n_i
  - depth of root is 0
  - depth of a tree = depth of the deepest leaf
- **Height :** height of n_i is the lenght of the longest path from n_i to a leaf
  - height of a leaf is 0
  - height of a tree = height of the root = depth of the tree

Trees

- Ordered tree
  - A rooted tree in which all the children of each vertex are ordered
- Binary tree
  - An ordered tree in which every vertex has no more than two children
  - Each child designated as either a *left child* or a *right child* of its parent

Trees

- Binary Search Tree (BST)
  - A binary tree
    - No repeated element
  - Satisfies Search Tree Property
    - Elements on left subtree is smaller than root
    - Elements on right subtree is greater than root
    - Left and right subtrees are BST
- Heap
  - An implementation of priority queue
  - A binary tree
  - Satisfies heap order property
    - Each element is larger than the parent

ROAD MAP

- **Mathematical Background**
- Fundamental Data Structures
  - Linear Data Structures
  - Graphs
  - Trees
- **What is an algorithm?**
- **Analysis of algorithms**
- Different Problems and their analysis
- Running time functions

What’s an algorithm?

- Dictionary definition: a procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step method for accomplishing some task
- Informal definition: an ordered sequence of instructions (operations) that is guaranteed to solve a specific problem
  - Step 1: Do something
  - Step 2: Do something
  - Step 3: Do something
  - …
  - Step N: Stop, you are finished

Operations

Algorithm is everywhere, not only in CS

Operations in an algorithm

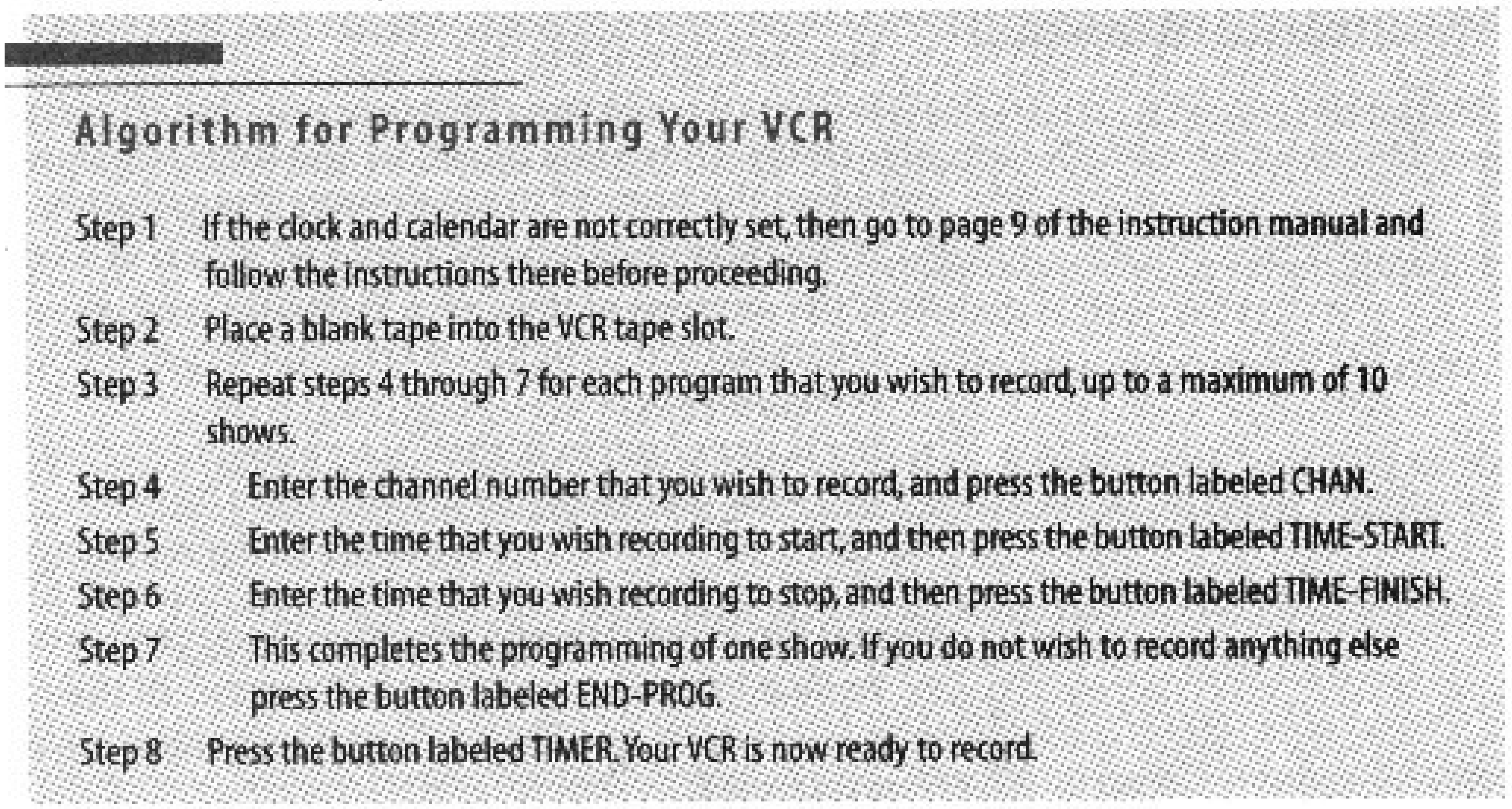
- A sequential operation carries out a single well-defined task. When that task is finished, the algorithm moves on to the next operation.
  - Add 1 cup of butter to the mixture in the bowl
  - Set the value of x to 1
- A conditional operation is the “question-asking” instructions of an algorithm. It asks a question and then select the next operation to be executed according to the question answer
  - If the mixture is too dry, then add 0.5 cup of water to the bowl
  - If x is not equal to 0, then set y equal to 1/x; otherwise, print an error message that says we cannot divide by 0

Operations in an algorithm

- An Iterative operation is a “looping” instruction of an algorithm. It tells us not to go on to the next instruction, but, instead, to go back and repeat the execution of a pervious block of instructions
  - Repeat the previous two operations until the mixture has thickened
  - Repeat steps 1, 2, and 3 until the value of y is equal to +1

Algorithm for programming your VCR

Which instructions are sequential, conditional, and iterative?



Algorithm for Programming Your VCR

Step 1 If the clock and calendar are not correctly set, then go to page 9 of the instruction manual and follow the instructions there before proceeding.

Step 2 Place a blank tape into the VCR tape slot.

Step 3 Repeat steps 4 through 7 for each program that you wish to record, up to a maximum of 10 shows.

  Step 4 Enter the channel number that you wish to record, and press the button labeled CHAN.

  Step 5 Enter the time that you wish recording to start, and then press the button labeled TIME-START.

  Step 6 Enter the time that you wish recording to stop, and then press the button labeled TIME-FINISH.

  Step 7 This completes the programming of one show. If you do not wish to record anything else press the button labeled END-PROG.

Step 8 Press the button labeled TIMER. Your VCR is now ready to record.

Formal definition of an algorithm

- An Algorithm is a well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time.
- An algorithm is any well-defined computational procedure that takes some inputs and produce some outputs.
  - An algorithm is a sequence of computation steps that transform the input into the output.
- Sorting problem
  - Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$
  - Output: A permutation (reordering) $< a'_1, a'_2, \ldots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
  - An instance of the sorting problem
    - (31, 41, 59, 26, 41, 58) ➔ (26, 31, 41, 41, 58, 59)

Expressing Algorithms

Algorithms can be expressed in

- natural languages
  - verbose and ambiguous
  - rarely used for complex or technical algorithms
- pseudocode, flowcharts
  - structured ways to express algorithms
  - avoid ambiguities in natural language statements
  - independent of a particular implementation language
- programming languages
  - intended for expressing algorithms in a form that can be executed by a computer
  - can be used to document algorithms

Example:

Problem: Find the largest number in an (unsorted) list of numbers.

Idea: Look at every number in the list, one at a time.

Natural Language:

Assume the first item is largest. Look at each of the remaining items in the list and if it is larger than the largest item so far, make a note of it. The last noted item is the largest in the list when the process is complete.

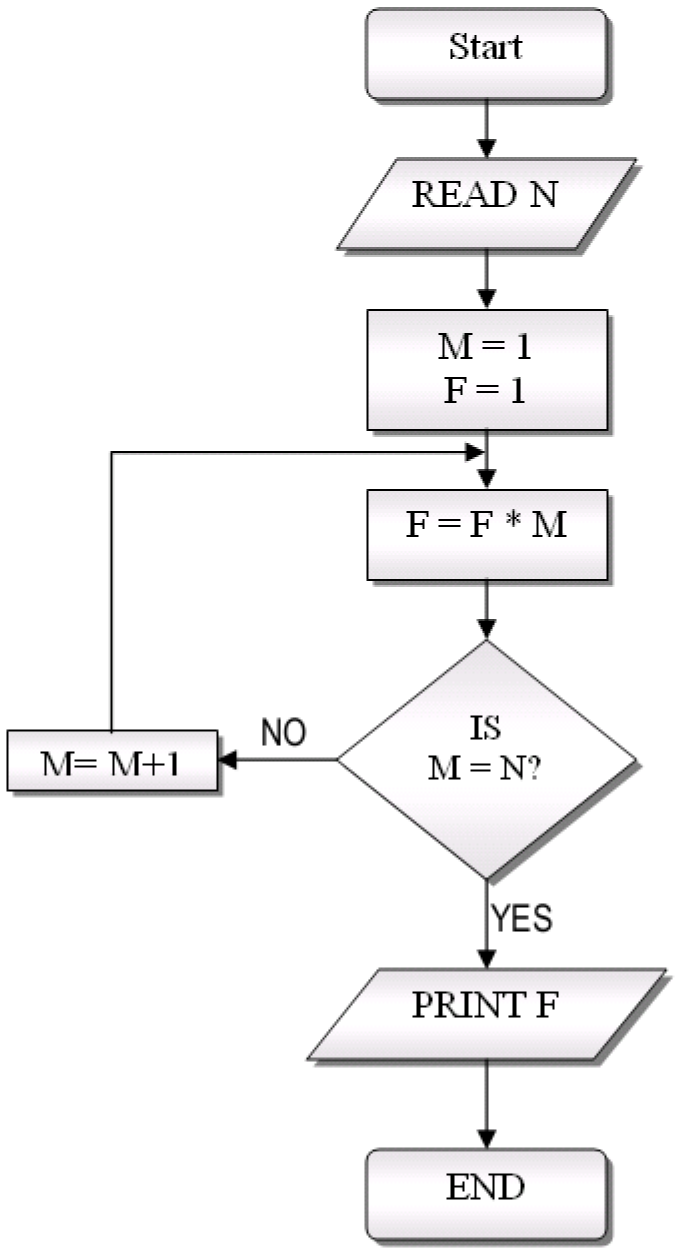
Example:

Pseudocode:

```
Algorithm LargestNumber
   Input: A non-empty list of numbers $L$.
   Output: The $largest$ number in the list $L$.
   $largest \leftarrow L_0$
   for each $item$ in the list $L_{i \geq 1}$, do
       if the $item > largest$, then
           $largest \leftarrow$ the $item$
   return $largest$
```

Example:

Flowchart:



Is the following an algorithm?

- Step 1: Wet hair
- Step 2: Lather
- Step 3: Rinse
- Step 4: Repeat

…a well-ordered collection…

- Clear and un-ambiguous ordering to these operations
  - What’s the next operation when we finish any one operation?
  - At step 4, what operations should be repeated?
- Ambiguous ordering
  - Go back and do it again
    - Go back to step 3 and begin execution from that point
  - Start over
    - Start over from step 1
  - If you understand this material, you may skip ahead
    - If you understand this material, skip ahead to line 21

…of unambiguous and effectively computable operations…

- An unambiguous operation is one that can be understood and carried out directly by the computing agent without needing to be further simplified or explained
  - Primitive (operation)
- Effectively computable or doable operation
  - There exists a computational process that allows the computing agent to complete that operation successfully

…of unambiguous and effectively computable operations… (Cont.)

- Step 1: Make the crust
- Step 2: Make the cherry filling
- Step 3: Pour the filling into the crust
- Step 4: Bake at 350℉ for 45 minutes

- Step 1: Make the crust
  - 1.1 Take one and one-third cups flour
  - 1.2 Sift the flour
  - 1.3 Mix the sifted flour with one-half cup butter and one-fourth cup water
  - Roll into two 9-inch pie crusts
- Step 2: make the cherry filling
  - 2.1 open a 16-ounce can of cherry pie filling and pour into the bowl
  - 2.2 add a dash of cinnamon and nutmeg, and stir

Algorithm for making a cherry pie

…of unambiguous and effectively computable operations… (Cont.)

- Which of the following are primitive operations for a computer?
  - Add x and y to get the sum z
  - See whether x is greater than, equal to, or less than y
  - Sort a list of names into alphabetical order
  - Factor an arbitrary integer into all of its prime factors
  - Make a cherry pie

…of unambiguous and effectively computable operations… (Cont.)

- Find and Print out the 100th prime number
  - Step 1: generate a list L of all the prime numbers
  - Step 2: Sort the list L into ascending order
  - Step 3: Print out the 100th element in the list
  - Step 4: Stop

- Write out the exact decimal value of π
  - π cannot be represented exactly
- Set *Average* to *Sum/Number*
  - What if number = 0 $\sqrt{N}$
- Set the value of result to
  - What if N < 0
- Add 1 to the current value of x
  - What if x currently has no value

… that produces a result…

- In order to know whether a solution is correct, an algorithm must produce a result that is observable to a user
  - What are the results of the VCR algorithm, cherry-pie making algorithm?
  - Sometimes it is not possible for an algorithm to produce the correct answer because for a given set of input, a correct answer does not exist
    - Error messages (result) should be produced instead

… and halts in a finite amount of time…

- The result must be produced after the execution of a finite number of operations, and we must guarantee that algorithm eventually reaches a statement that says “Stop, you are done”
  - The original shampooing algorithm does not stop

Algorithm for shampooing your hair

```
1. Wet your hair
2. Set the value of WashCount to 0
3. Repeat Steps 4 through 6 until the value of WashCount
   equals 2
4.     Lather your hair
5.     Rinse your hair
6.     Add 1 to the value of WashCount
7. Stop, you have finished shampooing your hair
```

Alternative algorithm for shampooing your hair

1. Wet your hair
2. Lather your hair
3. Rinse your hair
4. Lather your hair
5. Rinse your hair
6. Stop, you have finished shampooing your hair

Properties of an Algorithm

- **Effectiveness**
  - Instructions are simple
    - can be carried out by pen and paper
- **Definiteness**
  - Instructions are clear
    - meaning is unique
- **Correctness**
  - Algorithm gives the right answer
    - for all possible cases
- **Finiteness**
  - Algorithm stops in reasonable time
    - `produces an output

What kinds of problems are solved by algorithms?

- Internet routing: single-source shortest paths
- Search engine: string matching
- Public-key cryptography and digital signatures: number-theoretic algorithms
- Allocate scarce resources in the most beneficial way: linear programming
- …

Can every problem be solved algorithmically?

- There are problems for which no generalized algorithmic solution can possibly exist (unsolvable)
- There are also problems for which no efficient solution is known: NP-Complete problem
  - It is unknown if efficient algorithms exist for NP-complete problems
  - If an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them
  - An example: Traveling-Salesman Problem (TSP)
- There are problems that we don’t know how to solve algorithmically

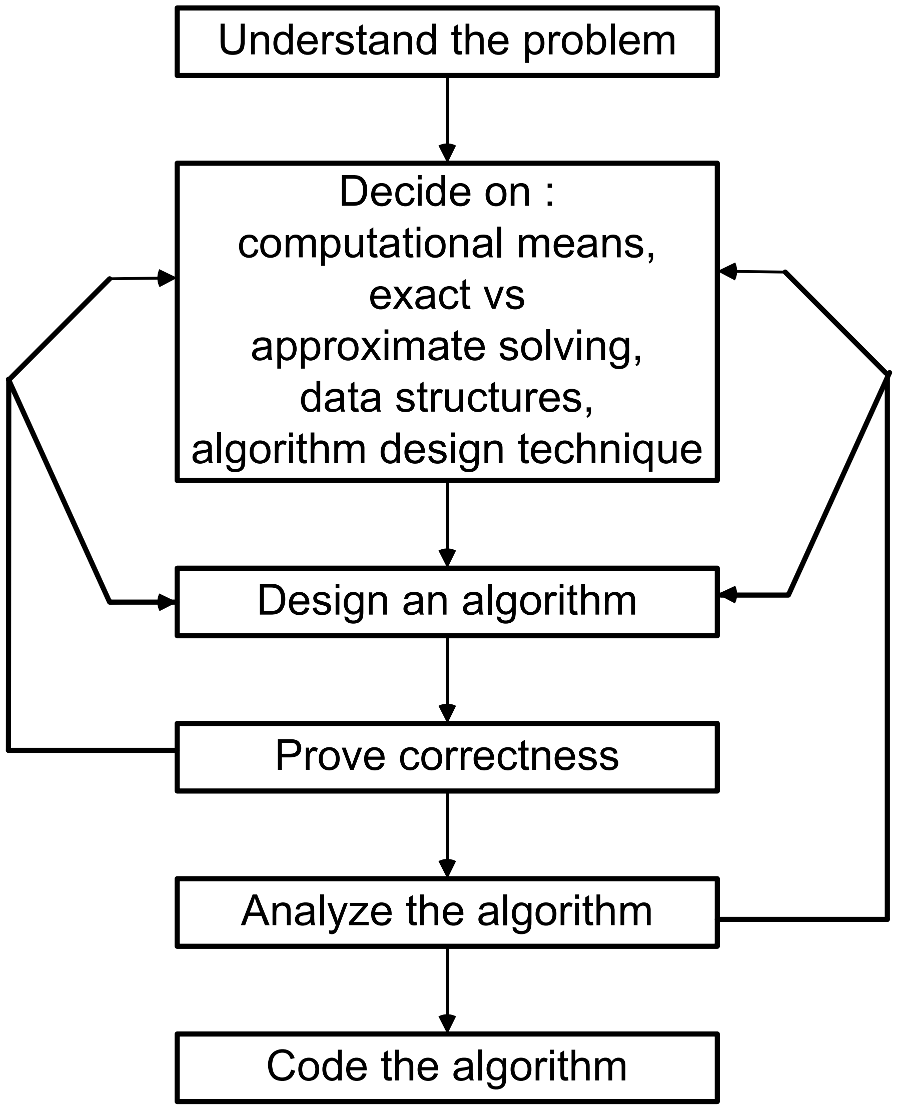
Algorithms and other technologies

- Algorithms are at the core of most technologies used in contemporary computers
  - The hardware design use algorithms
  - The design of any GUI relies on algorithms
  - Routing in networks relies heavily on algorithms
  - Compilers, interpreters, or assemblers make extensive use of algorithms

Notion of an Algorithm

$$\begin{tikzcd}
& \text{problem} \arrow[d, "\text{Algorithm Design}"] & \\
& \text{algorithm} \arrow[d, "\text{Programming the Algorithm}"] & \\
\text{input} \arrow[r] & \boxed{\text{computer}} \arrow[r] & \text{output}
\end{tikzcd}$$

Algorithm Design Process



Understand the problem
Decide on : computational means, exact vs approximate solving, data structures, algorithm design technique
Design an algorithm
Prove correctness
Analyze the algorithm
Code the algorithm

Analysis of Algorithms

- Study complexity of an algorithm
  - How good is the algorithm?
  - How it compare with other algorithms?
  - Is it the best that can be done?

Analysis of Algorithms

- Complexities
  - Space
  - Time

Analysis of Algorithms

- Space Efficiency
  - Be judged by the amount of information the algorithm must store in the computer’s memory in order to do the job, in addition to the initial data on which the algorithm is operating
  - Number of bits
  - Number of elements

Analysis of Algorithms

- Time Efficiency
  - An indication of the amount the work required by the nature of the algorithm itself and the approach it uses
  - Measure of the inherent efficiency of the method, independent of the machine speed or the specific working data
  - Count the fundamental unit or units of work of an algorithm
    - It is the number of steps each algorithm requires, not the time the algorithm takes on a particular machine, that is important for comparing two algorithms that do the same task
  - Number of operations
    - Depends on model
    - RAM
    - Turing Machines

Run-Time Analysis of Algorithms

- How to measure the running time of an algorithm?
  - The running time usually depends on input size
    - The running time may depend on which input of that size is given
  - The running time of an algorithm on a particular input is the number of primitive operations or "step" executed
    - A constant amount of time is required to execute each step (a line in pseudo code)

- Algorithm complexity is investigated as a function of some parameter *n* indicating problem's size

- Time complexity, T(*n*), is can be computed as the number of times the algorithm's most important operation -- called its basic operation -- is executed
- Space complexity, S(*n*), is usually computed as the size of memory space used during an execution of the algorithm

Types of Complexities

- Worst case

$$T(n) = max_{|I|=n}\{T(I)\}$$

- Average case

$$T(n) = \sum_{|I|=n} T(I).Prob(I)$$

- Best case

$$T(n) = min_{|I|=n}\{T(I)\}$$

• The worst-case running time is an upper-bound on the running time for any input
• For some algorithms, the worst case occurs fairly often
• The "average case" is often roughly as

Table Method

- *Table Method* is used to calculate complexity of an algorithm
- Example :

  <u>Adding elements of an array</u>

	steps/exec	freq	total
`sum = 0`	1	1	1
`for i = 1 to n`	1	n+1	n+1
`    sum = sum + a[i]`	1	n	n
`report sum`	1	1	1
			2n+3

Table Method

- *Table Method* is used to calculate complexity of an algorithm
- Example :

  <u>Adding elements of an array</u>

	steps/exec	freq	total
`sum = 0`	1	1	1
`for i = 1 to n`	1	n+1	n+1
`    sum = sum + a[i]`	1	n	n
`report sum`	1	1	1
			2n+3

What is the basic operation? How many times it is executed?

Table Method

- *Table Method* is used to calculate complexity of an algorithm
- Example :

  <u>Adding elements of an array</u>

	steps/exec	freq	total
`sum = 0`	1	1	1
`for i = 1 to n`	1	n+1	n+1
`    sum = sum + a[i]`	1	n	n
`report sum`	1	1	1
			2n+3

Basic operation is executed n times and n is proportional to 2n+3

Table Method

- Example :

  <u>Matrix addition</u>

  Assume *a, b, c* are mxn matrices

	steps/exec	freq	total
for i = 1 to m	1	m+1	m+1
for j = 1 to n	1	m(n+1)	mn+m
c[i,j] = a[i,j] + b[i,j]	1	mn	mn
			2mn+2m+1

ROAD MAP

- **Mathematical Background**
- Fundamental Data Structures
  - Linear Data Structures
  - Graphs
  - Trees
- What is an algorithm?
- Analysis of algorithms
- **Different Problems and their analysis**
- Running time functions

Important Problem Types

- Sorting
- Searching
- String Processing
- Graph Problems
- Combinatorial Problems
- Geometric Problems
- Numerical Problems

Searching

<u>Problem definition:</u>

- Given a list, locate the search key

Linear Search

Approach

```
1. start with the first element
2. if the element is the search key
           return the position
3. otherwise continue with the next element
4. return fail if the end of list reached
```

Linear Search

Pseudo – code

Given A[1], … , A[n] and *search key*

```
1. i=1
2. while i ≤ n
3.      if A[i] = key return i
4.      else i = i+1
5. return fail
```

Linear Search

Pseudo – code

Given A[1], … , A[n] and *search key*

```
1. i=1
2. while i ≤ n
3.      if A[i] = key return i
4.      else i = i+1
5. return fail
```

Basic operation? How many executions?

Linear Search

Pseudo – code

Given A[1], … , A[n] and *search key*

```
1. i=1
2. while i ≤ n
3.      if A[i] = key return i
4.      else i = i+1
5. return fail
```

What about best, worst and average cases ?

Linear Search

Algorithm Complexity:

- Best case

  A[1] = key
- Worst case

  A[i] ≠ key   for any key
  - time is proportional to the number of elements
  - time complexity of linear search is O(n)
- Average case
  - if any key is equally likely ~ n/2

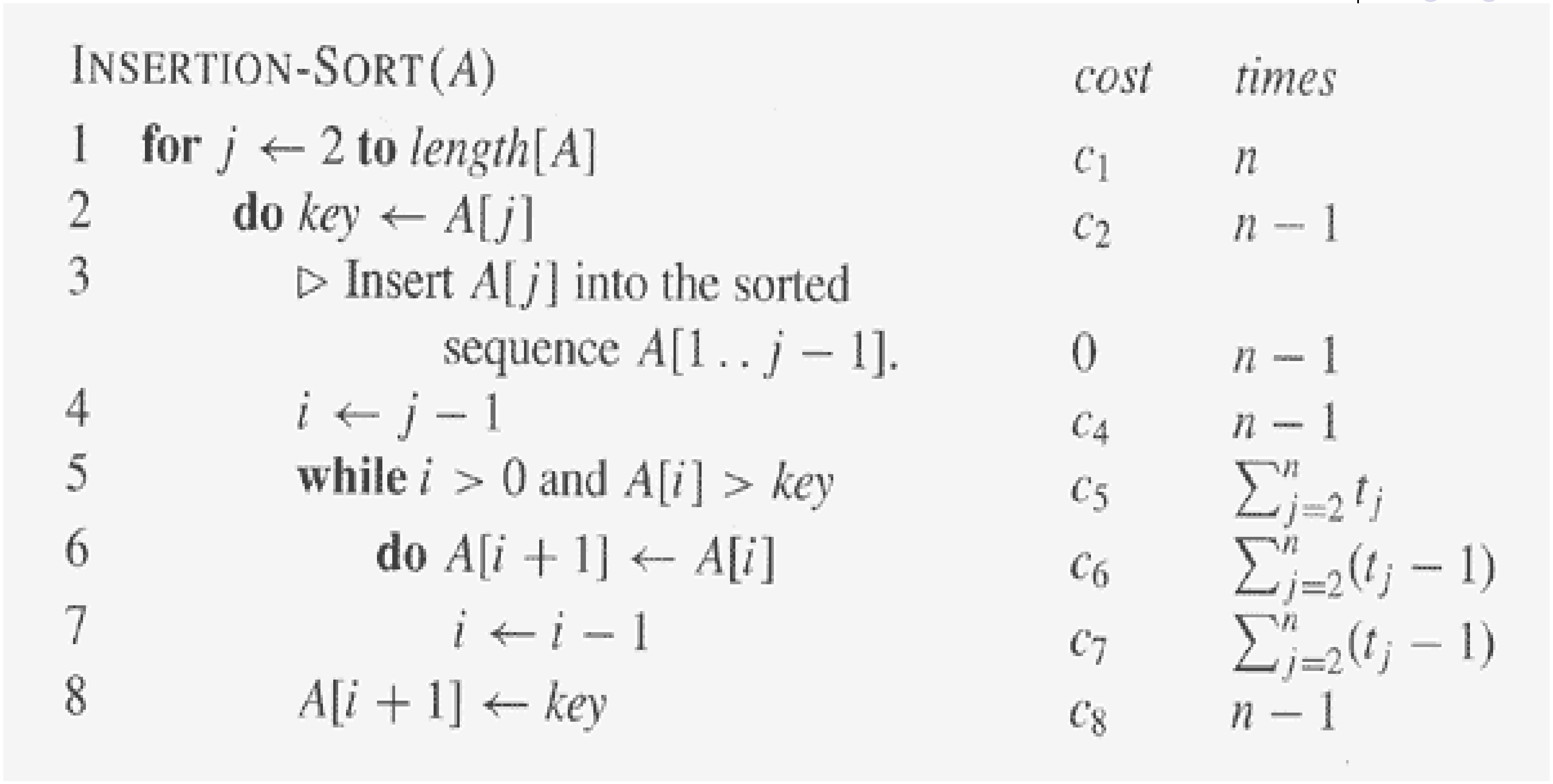
Insertion Sort

- **<u>Goal</u>**
  - Given A[1] … A[n], give a sorted sequence

- **<u>Approach</u>**

```
1. The sequence A[1] is sorted
2. Suppose A[1] ≤ A[2] … ≤ A[j-1] are already
   sorted
3. Pick up the next element and place it with
   its appropriate place
```

Running Time of Insertion Sort



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

tj = number of times the while loop executes for j

Insertion Sort

t_j depends on the problem instance (input)

Best-case

When A is sorted $t_j=0$ for all j

T(n) = linear function in n

= O(n)

Insertion Sort

Worst-case

When A is in reverse sorted order

$t_j = j$ *for* $j=2,\ldots,n$

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

T(n) = quadratic function

= $O(n^2)$

Insertion Sort

Average-case

We choose *n* numbers randomly

Where in A[1…j-1] will A[j] be inserted ?

Roughly half way i.e. $t_j \approx \frac{j}{2}$

$$\sum_{j=1}^{n} t_j = \sum_{j=1}^{n} \frac{j}{2} = \frac{1}{4}(n^2 + n)$$

T(n) → quadratic

Insertion Sort

Not a right way of calculating the average case.

Proof:

- Use notion of inversion
  - The number of inversions is the total number of swaps necessary
- What is the average number of inversions?
  - If the input is uniformly distributed...

Polynomial Evaluation

Given : $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 x^0$

Gool : evaluate $f(y)$

Incremental Approach

```
1. Sum = $a_0$
2. for i = 1 to n
        add $a_i y^i$ to the sum
```

Polynomial Evaluation

- **Analysis**
  - i multiplications and 1 additions at ith step

$$T = \sum_{i=1}^{n} (i+1) + 1 = O(n^2)$$

- **Improvement**
  - We know $\mathbf{y}^{\mathtt{i-1}}$ from previous stage
  - 2 multiplication & 1 addition

$$T(n) = 3n$$

Polynomial Evaluation

- **Further improvement**

$$P_n(x) = x(a_n x^{n-1} + \dots + a_1) + a_0$$

1 multiplication & 1 addition

$$T(n) = 2n$$

GCD Problem

Problem definition

- Calculate the greatest common divisor of two nonnegative integers *m* and *n*
    - $n>m>0$

Ex: gcd(60,24)=12

A Simple Solution to GCD

Approach

- Try all possible values from m to 2

```
1. for i=m to 2
2.     if i divides both m and n
3.         return i
```

Middle School Solution to GCD

Approach

- Find prime factorizations of n and m,
- Identify all the common primes,
- Multiply them to get the g.c.d.

Middle School Solution to GCD

Example: gcd(60, 24)

60	2	24	2
30	2	12	2
15	3	6	2
5	5	3	3
1		1	

So, gcd(60, 24) = 2*2*3 =12

Another GCD Algorithm

Approach

- Consider the following property :

  gcd(n, m) = gcd(n-m, m)

- Can you prove this property?
- What about the resulting algorithm?
  - Pseudocode
  - Analysis

Euclid’s GCD Algorithm

Approach

- use the following property :

  gcd(n, m) = gcd(m, n mod m)

```
1. divide n by m, let r be the remainder
2. if r=0 return m
3.      set n=m and m=r
4. continue until r=0
```

Euclid’s GCD Algorithm

- **Pseudo-code**
  - Recursive

```
Euclid(n , m)
   if m=0
     return n
   else
     return Euclid(m,n mod m)
```

  - Iterative

```
Euclid(n , m)
   while (m>0)
     t=n mod m
     n=m
     m=t
   return n
```

Euclid's GCD Algorithm

- **Correctness**

  For any two integers $n>m\geq 0$

  $gcd\ (n, m) = gcd\ (m, n \bmod m)$

- **Proof**

  We show that they divide each other

Euclid's GCD Algorithm

$$Let \quad d = \gcd(n,m) \quad then \quad \frac{n}{d} = k_1, \quad \frac{m}{d} = k_2$$

$$\frac{n \bmod m}{d} = \frac{n - \left\lfloor \frac{n}{m} \right\rfloor m}{d} = \underbrace{k_1 - k_2 \left\lfloor \frac{n}{m} \right\rfloor}_{an \quad \text{int}eger}$$

$$Let \quad d = \gcd(m, n \bmod m) \quad then \quad \frac{m}{d} = k_1, \quad \frac{n \bmod m}{d} = k_2$$

$$\frac{n}{d} = \frac{\left(n - \left\lfloor \frac{n}{m} \right\rfloor m\right) + \left(\left\lfloor \frac{n}{m} \right\rfloor m\right)}{d} = \underbrace{k_2 + \left\lfloor \frac{n}{m} \right\rfloor k_1}_{an \quad \text{int}eger}$$

Euclid’s GCD Algorithm Analysis

- **Lemma:** For any two integers m and n such that $n \geq m$, it is always true that

$$n \bmod m < n/2$$

- **Proof**

If $m \leq n/2$ then correct

If $m > n/2$ then $n \bmod m = n - m < n/2$

Euclid's GCD Algorithm Analysis

i	n	m
0	$n_0=n$	$m_0=m$
1	$n_1=m_0$	$m_1=n_0 \bmod m_0$
2	$n_2=m_1$	$m_2=n_1 \bmod m_1=m \bmod m_1$ $m_2<m_0 / 2$
3	$n_3=m_2$	$m_3=n_2 \bmod m_2$
4	$n_4=m_3$	$m_4=n_3 \bmod m_3 =m_2 \bmod m_3$ $m_4< m_2 / 2$
…		
k-1	$n_{k-1}=$ …	1
k	$n_k=$ …	0

In general $n_i= m_{i-1}$

$m_{i-1}= n_{i-2} \bmod m_{i-2}$ $n_i < n_{i-2} / 2$

$m_i < m_{i-2} / 2$

Euclid's GCD Algorithm Analysis

From the lemma

$$m_i < \frac{n_{i-1}}{2} = \frac{m_{i-2}}{2} \qquad for \qquad i \geq 2$$

$$Let \qquad k \qquad be \qquad odd \qquad k = 2d + 1$$

$$1 \leq m_{k-1} < \frac{m_{k-3}}{2} < \frac{m_{k-5}}{2^2} < \ldots < \frac{m_0}{2^d} = \frac{m}{2^d}$$

$$2^d \leq m \qquad then \qquad d \leq \log_2 m \qquad k \leq 1 + 2\log_2 m$$

Euclid's GCD Algorithm Analysis

So,

$$T(n,m) = O(\log m)$$

- Is it true to say "complexity is logarithmic"?
  - Algorithm is very fast?
- What is the problem size?
- What about the resulting complexity?

Complexity of a Problem

Not same as algorithm complexity!..

$$\min_A \{ \text{complexity of } A \}$$

Quite difficult to compute!...

- Upper Bound
  - Algorithm with given complexity guarantee f(n)
- Lower Bound
  - Proof that any algorithm for a problem must have complexity f(n)

Complexity of Searching

- Upper Bound = O(n)
  - Linear search gives O(n) solution
- Lower Bound = O(n)
  - Need to check all the values against the key to find the correct result.

- Complexity of Searching problem is O(n)
- Linear search is an **optimal** algorithm
  - Complexity of the algorithm matches the problem complexity

Complexity of Sorting

Upper Bound = O(n log n)

- Mergesort gives O(n log n) solution

- Lower Bound = O(n log n)
  - We will prove this using decision trees.

- Complexity of sorting is O(n log n)
- Mergesort and heapsort are optimal algorithms.

ROAD MAP

- **Mathematical Background**
- Fundamental Data Structures
  - Linear Data Structures
  - Graphs
  - Trees
- **What is an algorithm ?**
- **Analysis of algorithms**
- Different Problems and their analysis
- **Running time functions**

Running Time Functions

- Definition

  A nondecreasing function is called ***running time function*** if

  $f: Z^+ \rightarrow R$ such that $f(n) > 0$ for all $n \geq m$ where m is some positive integer

  $\mathbf{Z^+ = \{ 1, 2, 3, \ldots \}}$

Asymptotic notations

O notation

- Definition

  Let f and g are running time functions. We denote ***f(n) = O(g(n))*** if there exists a real constant c and integer m such that

$$f(n) \leq c\,(g(n)) \text{ for all } n \geq m$$

O notation

- Ex:
  $7n + 5 = O(n)$
- Ex:
  $10n^2 + 4n + 2 = O(n^2)$
- Ex:
  $7n + 5 = O(n^2)$
- Ex
  $7n + 5 \neq O(1)$

Asymptotic notations

Ω notation

- Definition

  Let f and g are running time functions. We denote ***f(n) = Ω(g(n))*** if there exists a real constant c and integer m such that

$$f(n) \geq c\,(g(n)) \text{ for all } n \geq m$$

Ω notation

- Ex:
  $3n + 2 = \Omega(n)$

- Ex:
  $6.2^n + n^2 = \Omega(2^n)$

- Ex:
  $3n - 7 = \Omega(1)$

Asymptotic notations

θ notation

- <u>Definition</u>

  Let f and g are running time functions. We denote ***f(n) = θ(g(n))*** if there exists real constants c_1 and c_2 and integer m such that

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge m$$

θ notation

- Ex:
  $3n + 2 = \theta(n)$

- Ex:
  $10 \log n + 4 = \theta(\log n)$

- Ex:
  $3n + 2 \neq \theta(1)$

Asymptotic notations

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
- $f(n) = O(g(n))$
  - *g* is an upper bound of *f*
  - *f* grows no faster than *g*
- How tight is this bound ?
  - $n=O(n^2)$
  - $n=O(2^n)$
- $f(n) = O(g(n)) \rightarrow g(n) = O(f(n))$?

Theorem : polynomial

$$f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0$$

$$let \qquad a = \{\max |a_i|\}$$

$$\leq a n^d + a\, n^{d-1} + \ldots + an + a$$

$$= n^d \left(a + a\frac{1}{n} + \ldots + a\frac{1}{n^{d-1}} + a\frac{1}{n^d}\right)$$

$$\leq n^d ad \qquad \forall n \geq 1$$

$m = 1$

$c = ad$ $\qquad f(n) = O(n^d)$

Example

$$\frac{x}{x-1} = 1 + \frac{1}{x} + \frac{1}{x^2} + \ldots$$

$For \quad large \quad n$

$$\frac{x}{x-1} = 1 + \frac{1}{x} + o\left(\frac{1}{n}\right)$$

$$\frac{x}{x-1} = 1 + \frac{1}{x} + O\left(\frac{1}{n^2}\right)$$

Example

$$\sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

$$\sum_{i=1}^{n} i^2 = O(n^3)$$

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

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

Some Rules

- Transitivity

$$f(n) = O(g(n)) \quad \& \quad g(n) = O(h(n)) \quad \Rightarrow \quad f(n) = O(h(n))$$

- Addition

$$f(n) + (g(n)) = O(\max\{f(n), g(n)\})$$

- Polynomials

$$a_0 + a_1 n + \ldots + a_d n^d = O(n^d)$$

Some Rules

- θ is equivalence notation

$$f(n) = \theta(f(n))$$

$$f(n) = \theta(g(n)) \quad \Rightarrow \quad g(n) = \theta(f(n))$$

$$f(n) = \theta(g(n)) \,\&\, g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$$

Some Rules

$$f_1(n) = \theta(g(n)) \,\&\, f_2(n) = \theta(g(n))$$
$$\Rightarrow f_1(n) + f_2(n) = \theta(g(n))$$

$$f_1(n) = \theta(g_1(n)) \,\&\, f_2(n) = \theta(g_2(n))$$
$$\Rightarrow f_1(n) + f_2(n) = \theta(g_1(n) * g_2(n))$$