CS 58000 - Algorithm Design, Analysis & Implementation (3 cr.)

Chapter 00_01

Introducing Foundation

Body of Knowledge Coverage:
Software Development Fundamentals (SDF)

- Algorithms and Design (SDF)
  - Concept and properties of algorithms,
  - Role of algorithms,
  - Problem-solving strategies,
  - Separation of behavior and implementation

Introduction –

- What is an algorithm?
- What is computer program?
- What is a problem?
- What are the parameters to a problem
- What is an instance of the problem?
- What is a solution of an instance of the problem?
- What is an algorithm for the problem?

Introduction – What is an Algorithm?

An algorithm is

- a well-defined procedure
  - a sequence of *unambiguous* instructions
    - for solving a well-specified computational problem
      - for obtaining a desired, *required* output
        - from *any given legitimate* input
        - in a *finite amount of time*.

{input specifications} Algorithm {output specifications}

Introduction – What is a computer program?

A computer program

- composed of individual modules,
  - understandable by a computer,
  - that solve specific tasks (such as sorting, searching, …).

- Our concern is
  - the design of these individual modules
    - that accomplish the specific tasks, that are called problems.
  - But NOT *the design of entire programs*

Introduction – What is a problem (task)?

- A problem is a question to which we seek an answer.
- An example of a problem 0.1.1:
    - Sort a list S of n numbers in nondecreasing order. (Question)
    - The answer is the numbers in sorted sequence.
- An example of a problem 0.1.2:
    - Determine whether the number x is in the list S of n numbers. (Question)
    - The answer is yes if x is in S, and no if it is not.

Introduction – What are parameters to a problem?

- They are variables without assigned specific values to the statement of the problem.
- Example 0.1.1: An example of a problem
    - **Question:** Sort a list *S* of *n* numbers in nondecreasing order.
    - **Answer:** the numbers in sorted sequence.
    - The 2 parameters to the problem are:
        - *S* (the list) and *n* (the number of items in *S*).
        - The parameter *n* is redundant,
            - since its value is uniquely determined by *S*.
        - n facilitates the problems' descriptions.

Introduction – What are parameters to a problem?

- Example 0.1.2: An example of a problem
  - **Question**: Determine whether the number x is in the list S of n numbers.
  - **Answer:** yes if x is in S and no if it is not.
  - The 3 parameters to the problem are:
    - S, n and the number x.
    - Again, the problem does not need the parameter n
      - since its value is uniquely determined by S.
    - n facilitates the problems' descriptions.

Summary: Problem? Questions? Answers? Parameters? Instances of an problem? An instance's Solution?

Introduction – What is an instance of the problem?

What is a solution of an instance of the problem?

- A *problem* containing parameters represents a class of problems.
    - one for each assignment of values to the parameters.
- An *instance* of the problem:
    - a specific assignment of values to the parameters.
- A *solution* to an instance of a problem:
    - The answer to the question asked by the instance of a problem.

Summary: Problem? Questions? Answers? Parameters? Instances of an problem? An instance's Solution?

Introduction – What is an instance and its solution of the problem?

- Example 0.1.1: An example of a problem

  Question: Sort a list S of n numbers in nondecreasing order.

  Answer: the numbers in sorted sequence.

- An instance of this problem in Example 0.1.1 is

  An instance of the problem: S = [10, 7, 11, 5, 13, 8] and $n = 6$.

  Solution: The solution to the instance is [5, 7, 8, 10, 11, 13].

Summary: Problem? Questions? Answers? Parameters? Instances of an problem? An instance's Solution?

Introduction – What is an instance and its solution of the problem?

- Example 0.1.2: An example of a problem

  Question: Determine whether the number x is in the list S of n numbers.

  Answer: The answer is yes if x is in S and no if it is not.

- An instance of the problem in Example 0.1.2 is

  $S = [10, 7, 11, 5, 13, 8]$, $n = 6$, and $x = 5$.

  The solution to this instance is, "yes, x is in S".

Introduction – What is an algorithm for the problem?

- An algorithm must specify
  - a step-by-step procedure
    - for producing the solution to *each* instance.
- We say that the algorithm solves all instances of the problem.

Introduction – What is an algorithm for the problem?

- Example 0.1.2: An example of a problem

  Question: Determine whether the number x is in the list S of n numbers.

  Answer: yes if x is in S and no if it is not.

- An algorithm for the problem in Example 0.1.2:

  - Starting with the first item in S,
  - compare x with each item in S in sequence until x is found or S is exhausted.
  - If x is found, answer yes; if x is not found, answer no.

Algorithm A 1.1 Sequential Search

A 1.1 Sequential Search

Problem Given:	Is the key K in the array S of n keys?
Inputs (parameters):	A positive integer n, array of keys S indexed from 0 to n-1 and a key K.
Outputs:	The index (location) of the first element of S that matches K, or -1 if there are no matching elements.

```
Algorithm SequentialSearch(S[0 .. n-1], K)
```

// Searches for a given value *K* in a given array *S* by sequential search

Input: An array *S*[*0* .. *n-1*] and a search key *K*

Output: The index (location) of the first element of *S* that matches *K* or -1 if there are no matching elements

```
i := 0;
while (i < n and S[i] ≠ K)
do    {i := i + 1; }
if  (i < n) return i;
else        return -1;
```

Q: which is the basic operation? Why? What is the running time (in terms of execution time)?

Compare their running time:

Algorithm SequentialSearch(*S*[0 .. *n*-1], *K*)

// Searches for a given value *K* in a given array *S*[*0* .. *n-1*] by sequential search

```
i := 0;
while (i < n and S[i] ≠ K)
do     {i := i + 1; }
if  (i < n) return i;
else          return -1;
```

Q: Which is the basic operation? Why?
Is there any difference in terms of execution time, if we design the while-do as the following?
Which one costs most?

```
i := 0;
while (i < n)
    {if (S[i] ≠ K) {i := i +1;}}//end while-do
if (i < n) return i;
else        return –i;
```

Or

```
i := 0;
while (i < n)
do     { if (S[i] = K) {return i;}
           i := i +1;} //end while-do
return -1;
```

```
Algorithm SequentialSearch(S[0 .. n-1], K)
```

// Searches for a given value *K* in a given array *S* by sequential search

Input: An array S[0 .. *n*-1] and a search key *K*

Output: The index (location) of the first element of *S* that matches *K*
or -1 if there are no matching elements

```
S[n] := K;
i := 0;
while (S[i] ≠ K)
do {i := i + 1; }
if  (i < n) return i;
else        return -1;
```

Q:

- Which is the basic operation? Why?
- Is there any difference in terms of execution time between this algorithm and the previous one? Why?

```
i := 0;
while (i < n and S[i] ≠ K)
do    {i := i + 1; }
if  (i < n) return i;
else        return -1;
```

Basic questions about an algorithm

In designing and analyzing an algorithm, the following questions are considered.

1. What is the problem we have to solve? Does a solution exist?
2. Can we find a solution (algorithm), and is there more than one solution?
3. Is the algorithm correct?
   i. Does it halt? (halting problem)
   ii. Is it correct? (partial correctness)
4. How efficient is the algorithm?
   i. Is it fast? (Can it be faster?) (time efficient)
   ii. How much memory does it use? (space efficient)
5. How does data communicate? (data representation/implementable)

Need to know about

i. Design and modeling techniques
ii. Resources – avoid reinventing the wheel

Logical methods of checking correctness

of an algorithm with respect to its input and output.

- Testing
- Correctness proof

- Confidence in algorithms from testing and correctness proof
- Correctness of recursive algorithms: prove directly by induction
- Correctness of iterative algorithms: prove using loop invariants and induction

Testing vs Correctness Proofs

- Testing
  - Try the algorithm on sample inputs
  - Testing may not find obscure bugs

- Correctness Proof
  - Prove mathematically can also contain bugs.

- Use a combination of testing and Correctness proofs

Analysis, Design and Implementation of an Algorithm:

1.0 Analysis of Algorithms?

- Study the complexity of an algorithm according to
  - time efficiency and
  - space efficiency (the amount of resource required to run an algorithm).

Analysis, Design and Implementation of an Algorithm:

1.1 Why Analyze an Algorithm?

- Reasons for analyzing an algorithm is:
  - Discover an algorithm's characteristics
    - Evaluate its suitability for various applications
    - Compare it with other algorithms for the same application.
  - Understand it (algorithm) better, and
  - Improve it.
    - Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

Analysis, Design and Implementation of an Algorithm:

1.2 Computational Complexity.

In theoretical computer science,

- the study of computational complexity theory focuses on studying the complexity of problems:
  - The complexity of a problem is the complexity of the best algorithms that allow solving the problem.
  - Study the complexity of a computational problems according to their inherent difficulty and relating those complexity classes to each other. e.g., TSP is a NP problem.

1.2 Computational Complexity.

In theoretical computer science,

- the study of computational complexity theory focuses on classifying:
  - algorithms according to time and space efficiency, such as $O(n^2)$
  - computational problems, based on their inherent difficulty,

    into classes, such as P class, NP class.
  - They focus on order-of-growth worst-case performance.
  - Such classifications *are not useful* for
    - predicting performance or
    - comparing algorithms in practical applications.
- We focus on analyses that *can* be used to *predict performance and compare algorithms.*

1.3 Analysis of Algorithms.

A complete analysis of the running time of an algorithm involves the following steps:

What are the domain of data and the representation of data?

- [input] Develop a realistic model for the input to the program.
- [input size] Analyze the unknown quantities of the modelled input.
- [algorithm development] Implement the algorithm completely.
- [algorithm analysis]
  - Determine the time required for each basic operation.
  - Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- [efficiency] Calculate the total running time by
  - multiplying the time by the frequency for each operation,
  - then adding all the products.

1.3 Analysis of Algorithms.

Efficiency

- Software is always outstripping hardware
  - need faster CPU, more memory for latest version of popular programs

- Given a problem:
  - what is an efficient algorithm?
  - what is the most efficient algorithm?
  - does there even exist an algorithm?

1.3 Analysis of Algorithms.

How to measure efficiency

- Machine-independent way:
  - analyze "pseudocode" version of algorithm
  - assume idealized machine model
    - one instruction takes one-time unit
- "Big-Oh" notation (order of growth)
  - order of magnitude as problem size increases
- Worst-case analyses
  - provides an upper bound on time taken by the algorithm.
    - The only “safe” analysis.
- Average case analysis
  - requires making some assumptions about the probability distribution of the inputs

1.4 Several Important Problem Types

- Specifying and implementing algorithms
- Basic complexity analysis
- Sorting
  - a set of items
- Searching
  - among a set of items
- String processing
  - text, bit strings, gene sequences
- Graphs
  - model objects and their relationships
- Network flow algorithms
- Tree traversals/State space search
- Combinatorial
  - find desired permutation, combination or subset
- Geometric
  - graphics, imaging, robotics
- Numerical
  - continuous math: solving equations, evaluating functions

1.5 Algorithm Design Strategies/Techniques.

- Simple Recursion
- Brute Force & Exhaustive Search
  - follow definition / try all possibilities
- Divide & Conquer
  - break problem into smaller subproblems
- Transformation
  - convert problem to another one
- Greedy
  - repeatedly do what is best now
- Dynamic Programming
  - break problem into overlapping subproblems
- Backtracking and Branch and Bound
- Iterative Improvement
  - repeatedly improve current solution
- Randomization
  - use random numbers
- Space and Time Tradeoffs

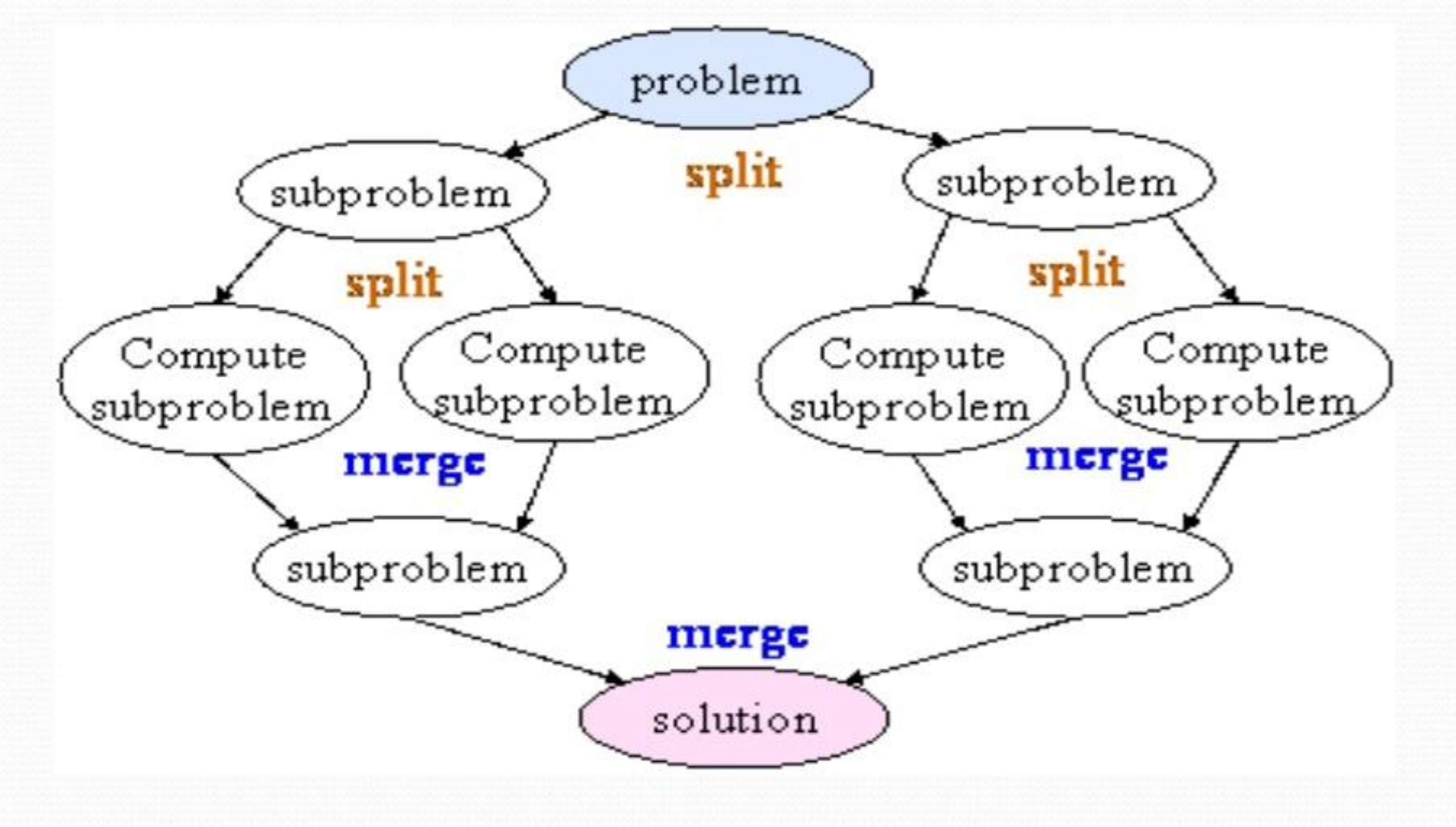
Divide and Conquer

- Break the problems into smaller sub-problems
- Solve each of the sub-problems
- Combine the solutions to obtain the solution to the original problem

Examples

- Binary search in a sorted array (recursion)
- Quick sort algorithm (recursion)
- Merge sort algorithm (recursion)

Divide and Conquer



Input: An array S[0 .. n-1] and a search key K

Output: The index (location) of the first element of S that matches K

or -1 if there are no matching elements

Method: Use sequential search. The order of growth is O(n).

Use binary search. The order of growth is $O(\log_2 n)$.

Analysis: For using binary search, it requires to sort the given array S in order.

Use merge sort or quick sort algorithm, which requires $O(n \log_2 n)$.

Therefore, the total time would be $T(n) = O(n \log_2 n) + O(\log_2 n)$.

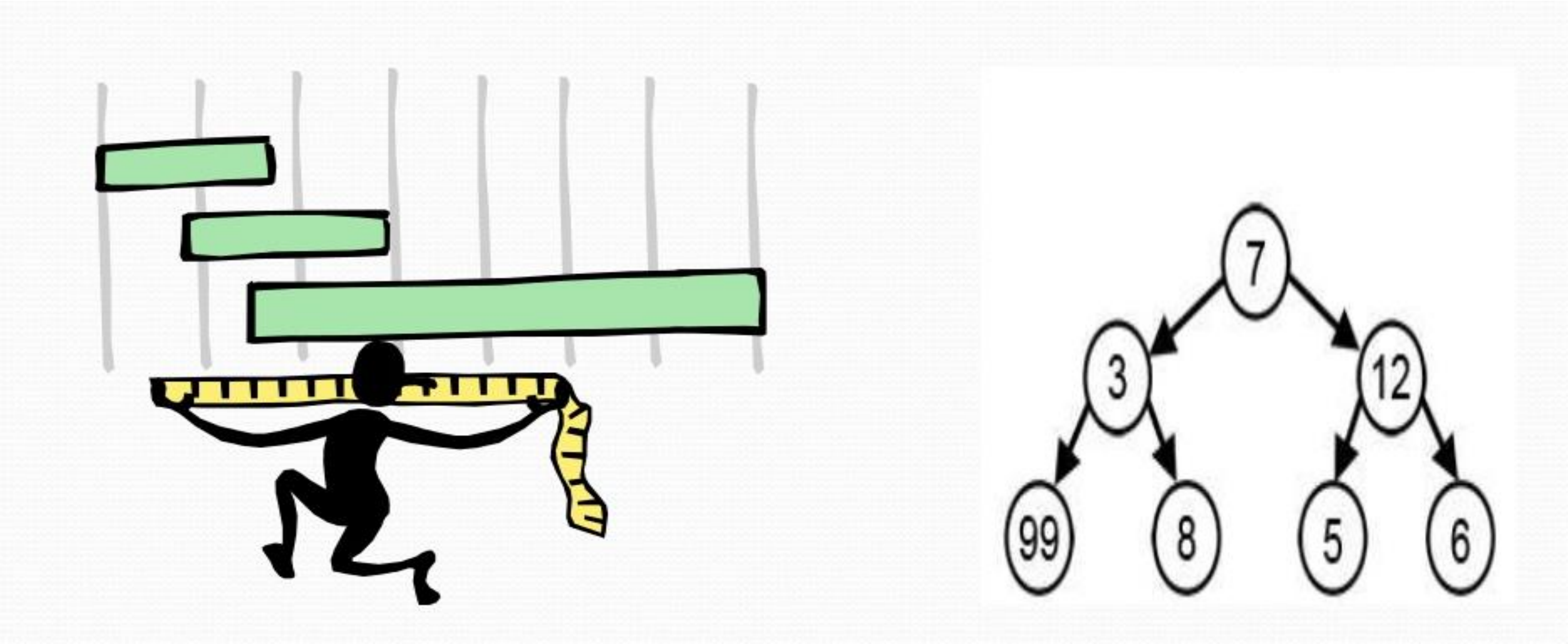
Greedy Algorithms

- An algorithm always takes the best immediate or local solution while finding an answer.
- Greedy algorithms
  - always find the overall or globally optimal solution for *some* optimization problems,
  - but may find less-than-optimal solutions for *some* instances of other problems.

Examples: Greedy algorithm for

- the Knapsack problem
- Minimal spanning tree

Greedy Algorithm



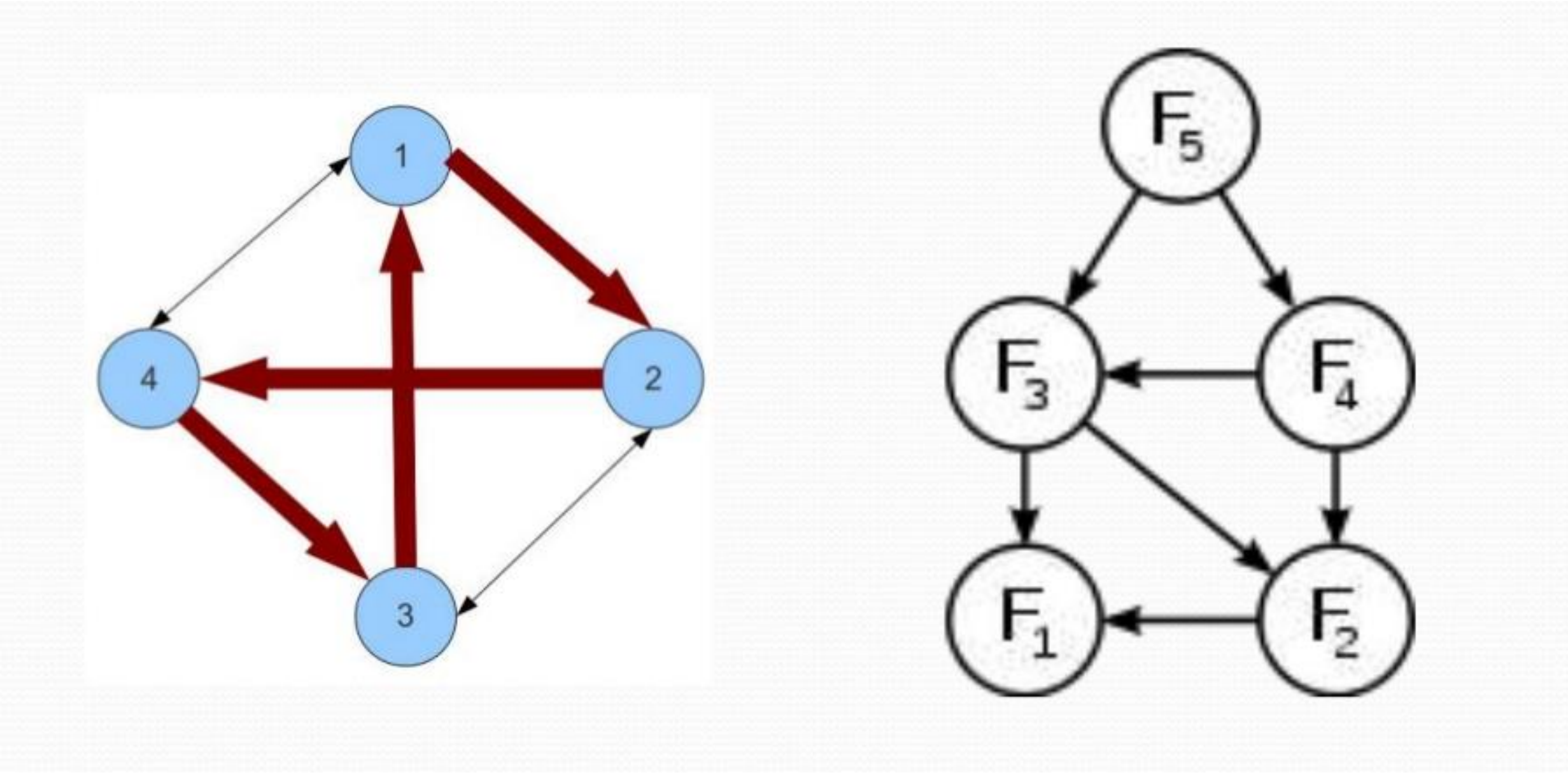
Dynamic Programming

- used to solve an optimization problem which requires the principle of optimality:
  - an optimal solution to *any instance* of an optimization problem is composed of optimal solutions to *its* sub-instances.
- is a bottom-up technique
  - solve the smallest sub-instances first and
  - use the results of these to construct solutions to progressively larger sub-instances.

Examples

- Fibonacci numbers computed by iteration.
- Warshall 's algorithm implemented by iterations.

Dynamic Programming



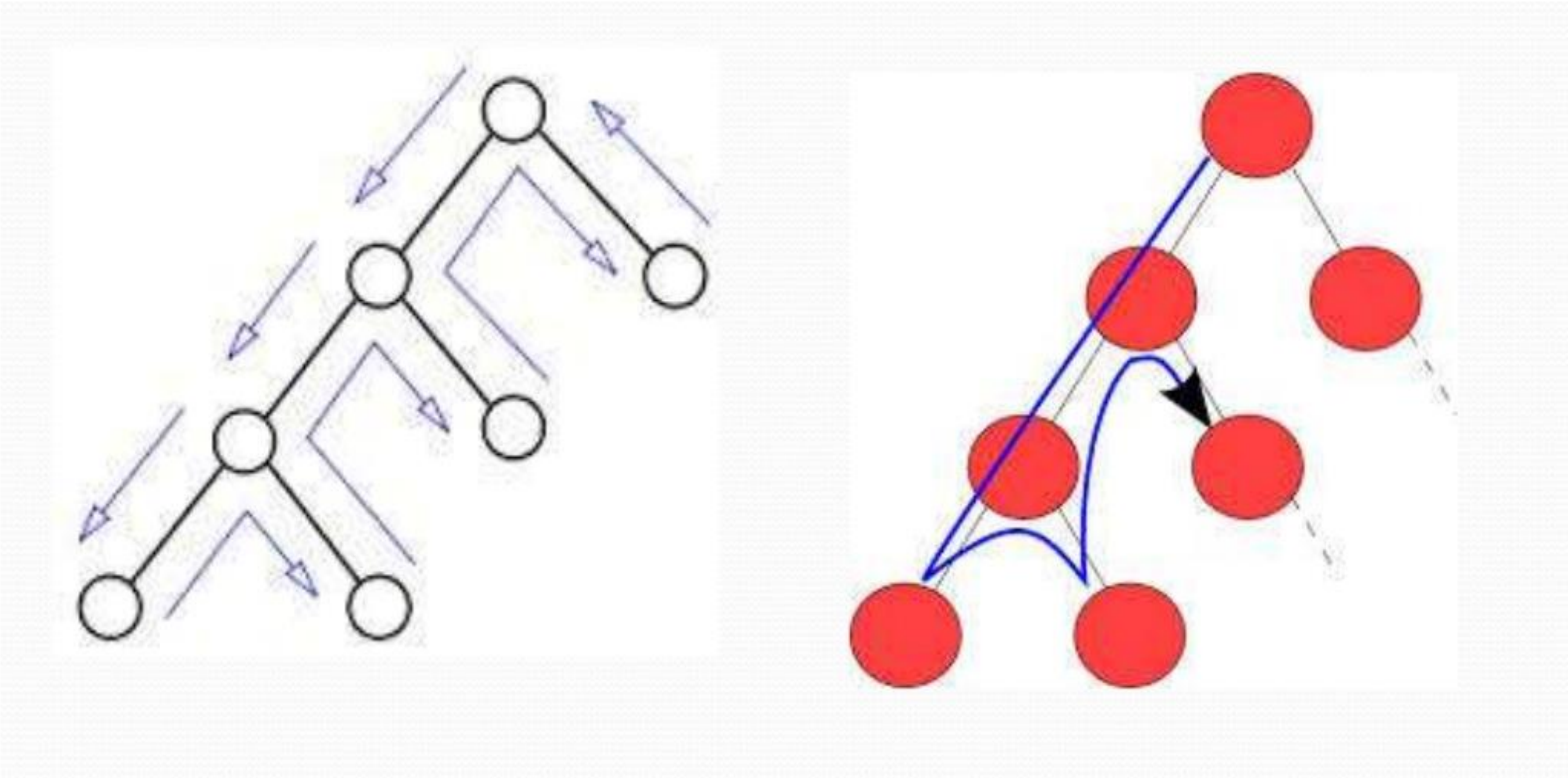
BackTracking

- Backtracking is a general algorithm for finding all solutions to some computational problem
  - incrementally builds candidates to the solutions, and
  - abandons each partial candidate c ("backtracks") when it determines that c cannot possibly be completed to a valid solution.

Examples

- Eight queens puzzle.
- Traveling salesman problem (TSP).

Backtracking



Graph Algorithm

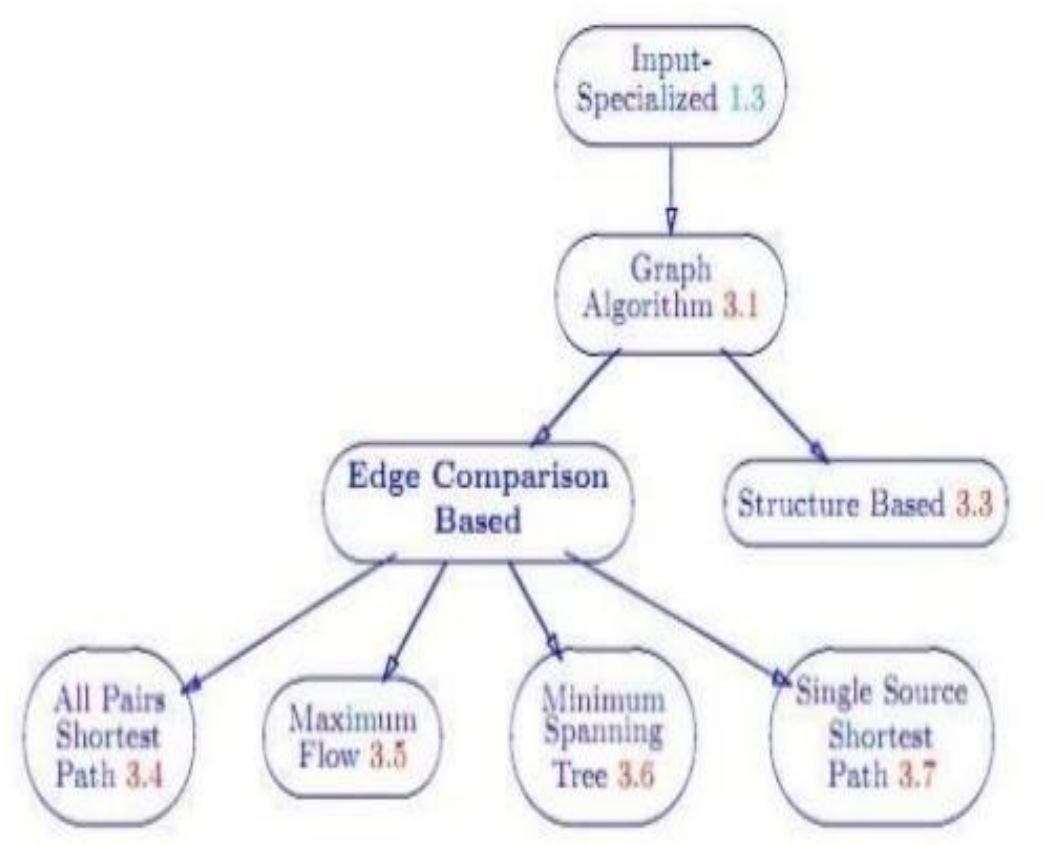
A graph algorithm takes one or more graphs as inputs.

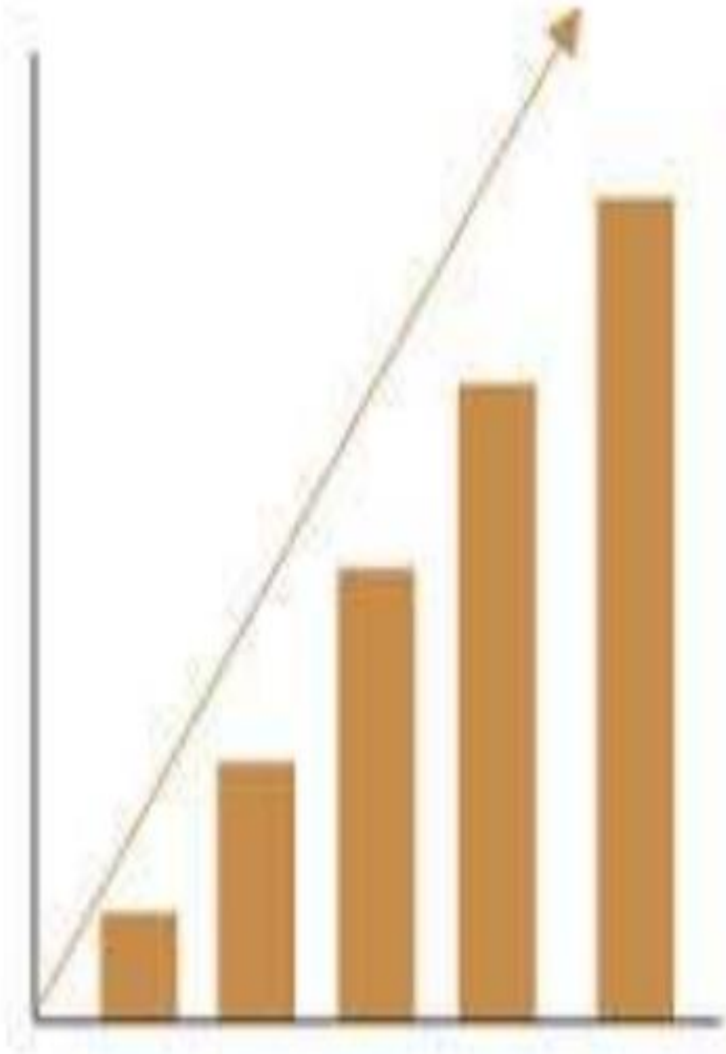
Performance constraints on graph algorithms are generally expressed in terms of

- the number of vertices (|V|) and
- the number of edges (|E|)

in the input graph.

Graph Algorithms





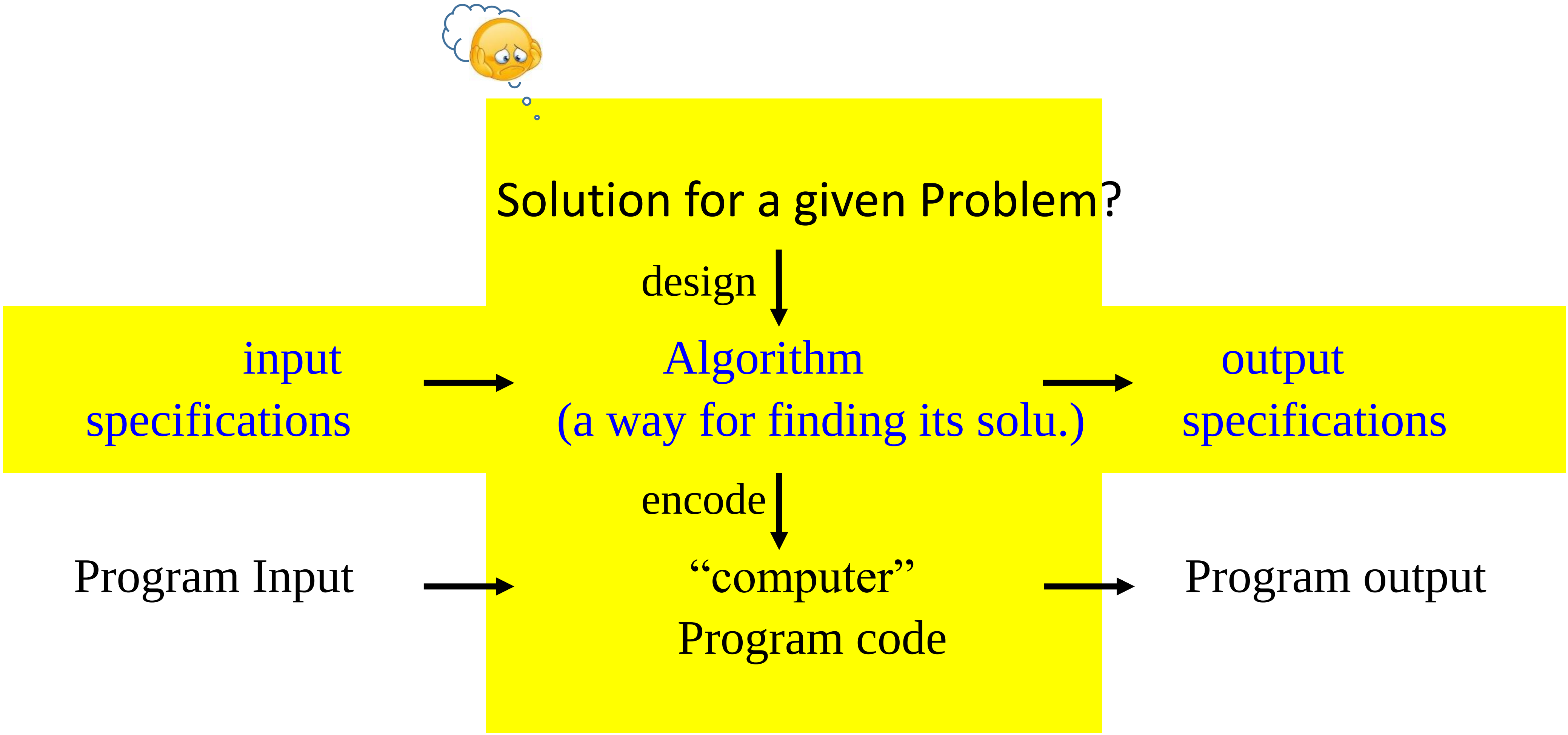


Figure 1.0 Notion of Algorithm

Several characteristics of Algorithms:

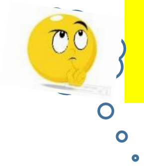
- *[Non-ambiguity]* The non-ambiguity requirement for each step of an algorithm cannot be compromised.
  - Prime Factorization in Middle School Procedure for computing gcd(m, n) is defined ambiguously
- *[Well-specified inputs' range]* The range of inputs for which an algorithm works *has to be specified precisely*.
  - Consecutive integer checking algorithm for computing gcd(m, n) does not work correctly when one of the input numbers is zero.
- …

Several characteristics of Algorithms:

- *[Different ways for specifying an algorithm]* The same algorithm can be written in different ways.
  - Euclid's algorithm can be defined recursively or non-recursively.
- *[Several algorithms for a problem]* Several algorithms for solving the same problem may exist.
  - Euclid, Consecutive Integer Checking, and Middle School Procedure for computing gcd(m, n)
- …

Several characteristics of Algorithms:

- …
- *[Various Speeds of different Algorithms for solving the same problem]* Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.
  - An expontential *algorithm Fibonacci_Number_F(n)* computes recursively the list of the n Fibonacci members based on its definition, and
  - a polynomial *Algorithm_Fib(n)* computes non-recursively the list of its of its n members.



Input Size

- For many algorithms, a reasonable measure of the *input size* - the *size* of the input.
  - For example, *the input size is the number n of items in the array* for sequential search, sorting, and binary search algorithms.
- Some algorithms use two numbers to *measure the size of the input*.
  - For example, when a graph G = (V, E) is the input to an algorithm, *the input size consists of both parameters: number of vertices |V| and edges |E|*.

Input Size

Must be cautious about calling a parameter the input size. For example,

- *Algorithm Euclid (m, n)* computes the greatest common divisor of two numbers m and n,
- *Algorithm Sieve(n*) finds all prime numbers less than or equal to n using the sieve of Eratosthenes method,
- *Algorithm Fibonacci_Number_F(n)* computes recursively the list of the n Fibonacci members based on its definition, and
- *Polynomial_Algorithm_Fib(n)* computes non-recursively the list of its n members.

- The input m and n should *NOT* be called the input size.
- Are the values of the parameters, m and n, the input size?

Input Size

For these algorithms: *Algorithm Euclid (m, n), Algorithm Sieve(n), Algorithm Fibonacci_Number_F(n), Polynomial_Algorithm_Fib(n),* and many others,

- a reasonable measure of the input size is
    - the number of symbols used to encode n.
- *When the binary representation is used,*
    - *the input size will be the number of bits it take to encode n,*
        - $\lfloor \log_2 n \rfloor + 1$.

$n = 2^b$.
$\log_2 n = \log_2 2^b$.
$\log_2 n = b \log_2 2$
$\log_2 n = b$

For example:

- Let $2^{b-1} \leq n < 2^b$. For example, let n = 15. Then $2^3 \leq n < 2^4$
- $b \leq \lfloor \log_2 n \rfloor + 1$, an integer value.
- Representing any n in terms of number of bits is $\lfloor \log_2 n \rfloor + 1$.

Input Size

For a given algorithm, the input size is defined as

- the *number of characters* it takes to write the input.

Input Size

If an input is encoded in binary inside computers, then

- the characters used for encoding the input are binary digits (bits), and
- the number of characters it takes to encode a positive integer x is $\lfloor \log_2 x \rfloor + 1$.
- the input size is $\lfloor \log_2 x \rfloor + 1 = \lceil \log_2 x \rceil$ bits.

For example:

- $31 = 11111_2$ and the number of characters used for encoding 31 is $\lfloor \log_2 31 \rfloor + 1 = 5$.

Modeling the Real World- Develop a realistic model for the input

- Cast your application in terms of well-studied abstract data structures

Concrete	*Abstract*
arrangement, tour, ordering, sequence	permutation
cluster, collection, committee, group, packaging, selection	subsets
hierarchy, ancestor/descendants, taxonomy	trees
network, circuit, web, relationship	graph
sites, positions, locations	points
shapes, regions, boundaries	polygons
text, characters, patterns	strings

Real-World Applications

- Hardware design: VLSI chips
- Compilers
- Computer graphics: movies, video games
- Routing messages in the Internet
- Searching the Web
- Distributed file sharing
- Computer aided design and manufacturing
- Security: e-commerce, voting machines
- Multimedia: CD player, DVD, MP3, JPG, HDTV
- DNA sequencing, protein folding
- and many more!

The Objectives of the Course

1. Be able to identify and abstract computational problems.
2. Know important algorithmic techniques and a range of useful algorithms.
3. Be able to implement algorithms as a solution to any solvable problem.
4. Be able to analyze the complexity and correctness of algorithms.
5. Be able to design correct and efficient algorithms.

- Separation of behavior and implementation

Example 1

- Fibonacci numbers:
  $F_0 = 0;$
  $F_1 = 1;$
  $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$
- Fibonacci numbers grow almost as fast as the power of 2:
  $F_n \approx 2^{0.694n}$
- Problem statement:
  computing the n-th Fibonacci number F_n
- Algorithms for computing the n-th Fibonacci number F_n :
  1. Recursion (top-down")
  2. Iteration (bottom-up", memorization)
  3. Divide-and-conquer
  4. Approximation

Example 2

- Problem statement:

  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 a-sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

  (In brief, sort the n numbers in ascending order.)

- Algorithms:

1. Insertion sort
2. Merge sort

Example 2: Insert sort algorithm

- Idea: incremental approach
- Pseudocode

```
InsertionSort(A)
n = length(A);
for (j = 2 to n) {
     key = A[j];
     // insert ``key" into sorted array A[1...j-1]
     i = j-1;
     while (i > 0 and A[i] > key) do {
          A[i+1] = A[i];
          i = i - 1;
       } //end while
      A[i+1] = key;
} //end for
return A;
```

Example 2: **Insert sort algorithm**

Remarks:

- Correctness: argued by “loop-invariant” (a kind of induction)
- Complexity analysis:
  - best-case
  - worst-case
  - average-case
- Insertion sort is a “sort-in-place”, no extra memory necessary
- Importance of writing a good pseudocode = “expressing algorithm to human”
- There is a recursive version of insertion sort (can you do it)

Example 2: Merge sort algorithm

- Idea: divide-and-conquer approach
- Pseudocode

```
MergeSort(A, p, r)                  // Merge-sort of array A[p..r]
 if (p < r) then                    // check for base case
   q = flooring( (p+r)/2 )          // divide
   MergeSort(A, p, q)               // conquer
   MergeSort(A, q+1, r)             // conquer
   Merge(A, p, q, r)                // combine
 end if
```

Example 2: Merge sort algorithm

- Pseudocode, cont'd

```
Merge(A, p, q, r)
n1 = q – p + 1;  n2 = r – q;
for (i = 1 to n1) {              // create arrays L[1...n1+1] and R[1...n2+1]
      L[i] = A[p+i-1];}          // end for
for (j = 1 to n2) {
      R[j] = A[q+j]; }           //end for
L[n1+1] = ∞; R[n2+1] = ∞;    // mark the end of arrays L and R
i = 1;  j = 1;
for (k = p to r) {               // Merge arrays L and R to A
      if (L[i] ≤ R[j]) then
            {A[k] = L[i];
             i = i + 1;}
      else
            {A[k] = R[j];
             j = j + 1;}        //end if
}  //end for
```

Example 2: Merge sort algorithm

- Merge sort is a divide-and-conquer algorithm consisting of three steps: divide, conquer and combine
- To sort the entire sequence A[1...n], we make the initial call

```
MergeSort(A, 1, n)
```

where n = length(A).

- Complexity analysis:

$$T(n) = 2 * T\left(\frac{n}{2}\right) + n - 1 = O(n \log_2(n))$$

- Extra-space is needed.