INT102

Algorithmic Foundations And Problem Solving

Summary

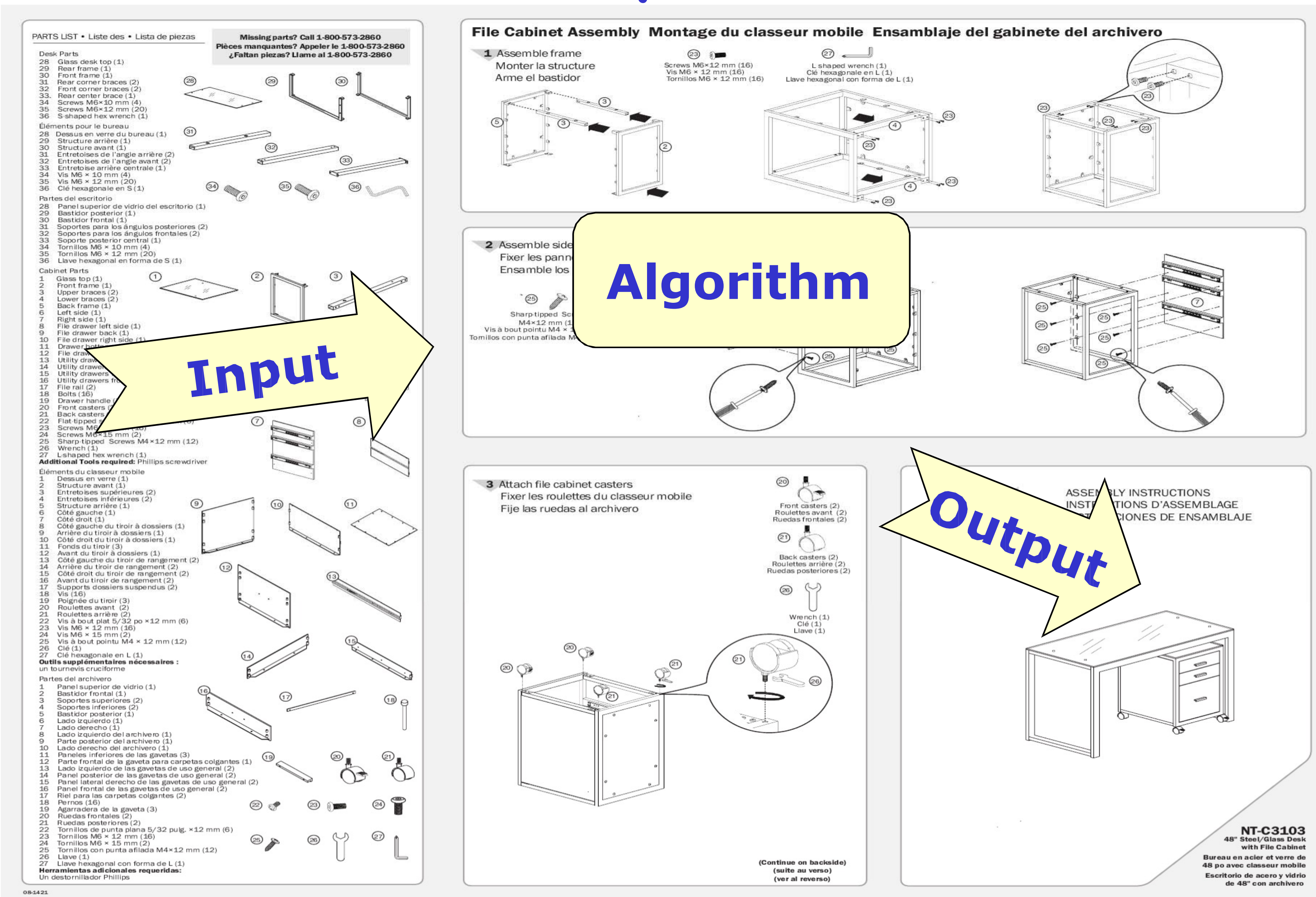
Dr Jia WANG

Department of Computer Science

西交利物浦大学
Xi'an Jiaotong-Liverpool University

Acknowledgment: The slides are adapted from ones by Dr. Prudence Wong

Desk Assembly Manual



What is INT102?

Algorithms + Data Structures + Methods ***Complexity Analysis***

Algorithms

- Particular problem-solving algorithms
- Sorting/ Searching/Shortest path

Method

- Greedy/Recurrences/Dynamic/Brach-and-bond/Approximation/Backtracking

Data Structures

- Graph

Pseudo Code: conditional

Conditional statement

```
if condition then
  statement

if condition then
  statement
else
  statement
```

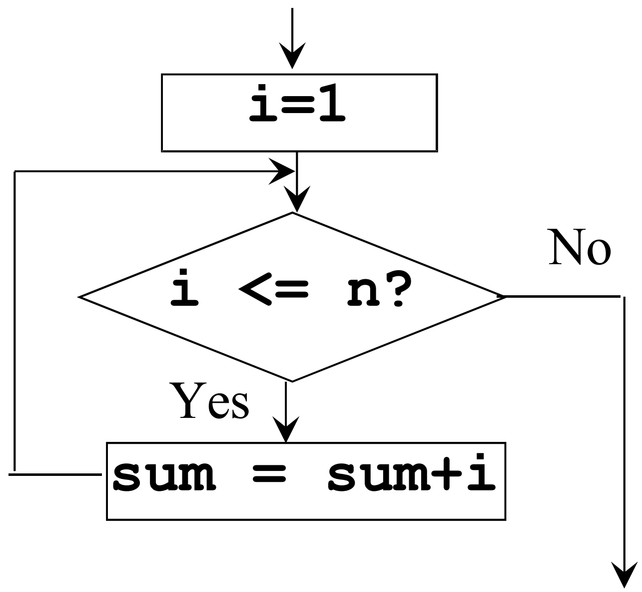
```
if a < 0 then
  a = -a
abs = a
output abs
```

```
if a > 0 then
  abs = a
else
  abs = -a
output abs
```

What do these two algorithms compute?

for loop

```
for var = start_value to end_value do
   statement
```



Computing sum of the first n numbers:

```
input n
sum = 0
for i = 1 to n do
begin
   sum = sum + i
end
output sum
```

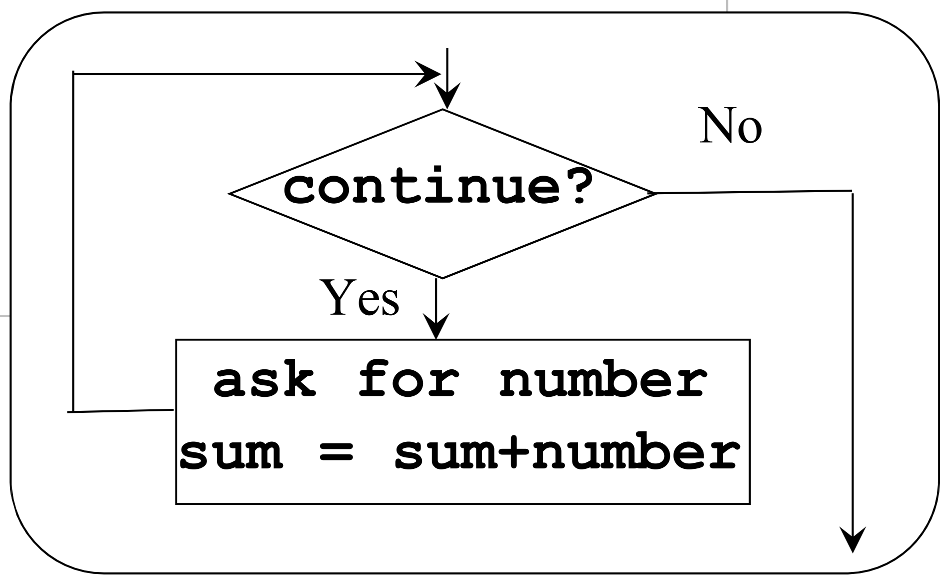
the loop is executed for n times

while loop

this **loop** is executed for **undetermined** number of times

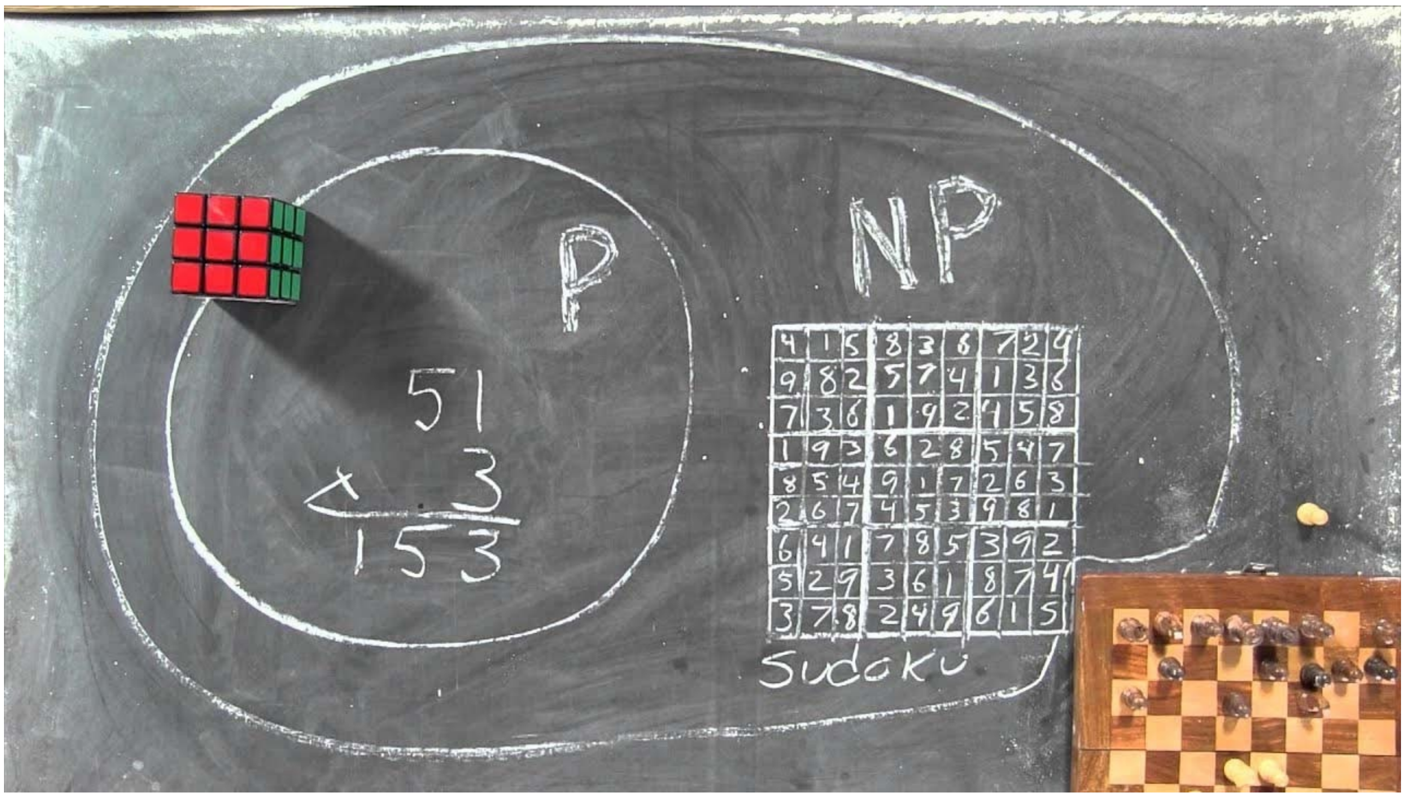
Computing sum of all (keyboard) input numbers:

```
sum = 0
while (user wants to continue) do
begin
   ask for a number
   sum = sum + number
end
output sum
```



Time complexity
- Big O notation …

Able to carry out simple asymptotic analysis of algorithms



Particular problem-solving algorithms

Searching

- Input: a sequence of n numbers $a_0, a_1, \ldots, a_{n-1}$ and a number X
- Output: determine whether X is in the sequence or not
- Algorithm (Linear search):
  1. Binary Search
  2. Linear Search
  3. Pattern Searching

Sorting

- **Input:** a sequence of n numbers $a_0, a_1, \ldots, a_{n-1}$
- **Output:** arrange the n numbers into ascending order, i.e., from smallest to largest
- **Example:** If the input contains 5 numbers 132, 56, 43, 200, 10, then the output should be
10, 43, 56, 132, 200

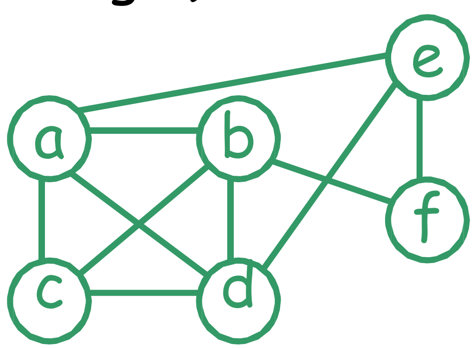
There are many sorting algorithms:
insertion sort, selection sort, bubble sort, merge sort, quick sort

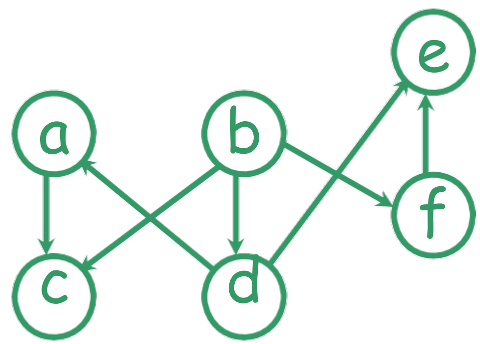
Graph … DATA STRUCTURE

Graphs

Graph theory – an old subject with many modern applications.

An *undirected* graph G=(V,E) consists of a set of vertices V and a set of edges E. Each edge is an *unordered* pair of vertices. (E.g., {b,c} & {c,b} refer to the same edge.)





A *directed* graph G=(V,E) consists of … Each edge is an *ordered* pair of vertices. (E.g., (b,c) refer to an edge from b to c.)

Learning outcomes

- Able to tell what is an undirected graph and what is a directed graph
  - Know how to represent a graph using matrix and list
- Understand what Euler path / circuit and able to determine whether such path / circuit exists in an undirected graph
- Able to <u>apply BFS and DFS</u> to traverse a graph
- Able to tell what a tree is

Methods

1. Greedy methods

- Understand what greedy method is
- Able to apply Prim's algorithm to find minimum spanning tree
- Able to apply Kruskal's algorithm to find minimum spanning tree
- Able to apply Dijkstra's algorithm to find single-source shortest-paths

2. Dynamic Programing

- Understand the basic idea of dynamic programming
- Able to apply dynamic programming to solve problems
  - Assembly line/ Fibonacci numbers/Global alignment/Local alignment/LCS/Knapsack problem

3.Backtracking

- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions

- Explore the state space tree using depth-first search

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

4. Branch-and-Bound

- An enhancement of backtracking

- Applicable to optimization problems

- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)

- Uses the bound for:
  - ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far
  - guiding the search through state-space

5. Approximation

- ➓ Find a "good" solution fast
  - sufficient for many applications
  - we often have inaccurate data to start with, so approximation may be as good as optimal solution

INT102
END OF TEACHING

What we have learnt

methodology / problems	Asymptotic idea	Brute force	Divide & Conquer	Dynamic Programming	Greedy	Space/Time	Branch & Bound	Backtracking	Complexity Theory
Efficiency	Big-O								
Sorting		Selection/ Bubble/insertion	Merge-sort			Count sorting			
Searching			Binary-searching						
String		searching		Alignment/LCS		Horspool algorithm			
Graph/Combinatory		DFS/BFS		Floyd's Algorithm/ Assembly-line Knapsack	MST(Prim's/ Kruskal's) Dikstra's For Shortest path		Traveling salesman, Job assignment	n-Queens Sum of subset Hamiltonian Problem	Approximation: TSP problem: Nearst-Neighbor/twice round/fragement algorithm
Complexity									P/NP Circuit-SAT/3-SAT

How will it be assessed

- **Two assignments (20% of the final mark)**

  1. Assignment 1 (week 6 - week 7)   (10%)
  2. Assignment 2 (week 11 – week 12) (10%)

- **Final Examination (80% of the final mark)**

  written examination: 70% MCQ's + 30% Problem Solving

This is an CLOSED BOOK examination.

Questions 1 to 4 refer to the following algorithm.	

Algorithm: P($a[l \ldots r]$)

Input: an array $a[l \ldots r]$ of real numbers

begin

if $l = r$ **return** l

else

$ll = \mathrm{P}(a[l \ldots \lfloor (r+l)/2 \rfloor])$

$rr = \mathrm{P}(a[\lfloor (r+l)/2 \rfloor + 1 \ldots r])$

if $a[ll] > a[rr]$ **return** ll

else **return** rr

end

1. Which algorithm design technique is employed in the above algorithm? [A] Brute Force technique [B] Greedy technique [C] Divide- and-Conquer [D] Dynamic Programming [E] Time and Space trade-off	**2.5**
2. For $n = 2^k$ and $k \geq 1$, the time complexity of the algorithm can be best expressed by [A] $T(n) = T(n/2) + 1$ [B] $T(n) = T(n/2)$ [C] $T(n) = 2T(n/2) + 1$ [D] $T(n) = 2T(n/2)$ [E] None of the above	

Analyzing the Time Complexity

To determine the time complexity, let's denote $T(n)$ as the time complexity of the algorithm for an array of size n.

1. **Base Case:** When $l = r$ (i.e., the subarray has only one element), the algorithm takes constant time:
   $T(1) = O(1)$

2. **Recursive Case:** When $l \neq r$, the algorithm divides the array into two subarrays of approximately equal size (each of size $n/2$), solves the problem for each subarray, and then combines the results by comparing two elements:
   $T(n) = 2T\left(\frac{n}{2}\right) + O(1)$

The $O(1)$ term represents the time taken to compare the two elements and return the index of the larger one.

Solving the Recurrence Relation

The recurrence relation for $T(n)$ is:
$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$

3. The time complexity of the algorithm is [A] O(n) [B] O(log n) [C] O(n^2) [D] O(n log n) [E] None of the above	**2.5**
4. What is the output of the algorithm for the input a[0..7] = [12, 12, 12, 12, 12, 12, 12, 12]? [A] 1 [B] 3 [C] 5 [D] 7 [E] 12	**2.5**

Algorithm Explanation

The algorithm recursively finds the index of the maximum element in the array by dividing the array into smaller subarrays until each subarray has only one element. It then compares the maximum elements of these subarrays and returns the index of the larger element. If the elements are equal, it returns the right index.

item	weight	value
1	2	$12
2	1	$10
3	3	$20

The Knapsack Capacity W=3

Let V[i, j] be the value of the most valuable subset of the first i items that fit into the Knapsack of capacity j. Then V[i, j] can be recursively defined as follows:

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{V[i-1, j],\quad v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0 \end{cases}$$

For the above instance, the following is an incomplete table for V[i, j] (i=0, 1, 2, 3; j=0, 1, 2, 3)

		capacity j			
Item	i	0	1	2	3
	0	0	0	0	
w_1=2, v_1=12	1	0	0	12	12
w_2=1, v_2=10	2	0	10		22
w_3=3, v_3=20	3	0	10	12	

22. What is the value of $V[1, 2]$?	**2.5**
[A] 12 [B] 10 [C] 22 [D] 0 [E] 24	
23. What is the value of the most valuable subset?	**2.5**
[A] 12 [B] 10 [C] 22 [D] 0 [E] 24	

```
Algorithm: LIS(A[0…n-1])
// Input: An array A[0…n-1] of n numbers
Begin
    length[0…n-1] = array of size n filled with 1
    // length[i] will hold the length of the LIS ending at position i

    for i from 1 to n-1 do
        for j from 0 to i-1 do
            if A[i] > A[j] and length[i] < length[j] + 1 then
                length[i] = length[j] + 1
    maxLength = 0
    for i from 0 to n-1 do
        if maxLength < length[i] then
            maxLength = length[i]

    return maxLength
End
```

Which algorithm design technique is employed in the above algorithm?

[A] Brute Force technique

[B] Greedy technique

[C] Divide- and-Conquer

[D] Dynamic Programming

[E] Ad hoc technique

The output of the algorithm is

[A] The sum of the longest increasing subsequence.

[B] The length of the shortest increasing subsequence.

[C] The length of the longest increasing subsequence.

[D] The number of increasing subsequences.

[E] None of above.