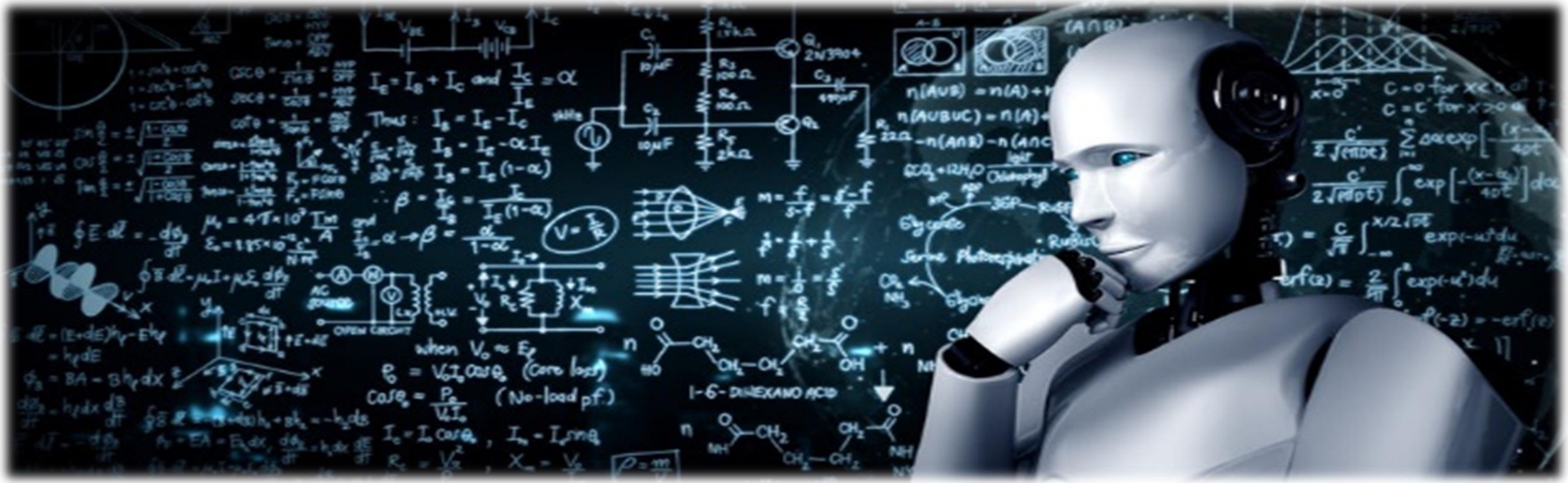


Lecture 01 - Introduction to Algorithms

Design and Analysis of Algorithms – IT1205

Year 02 Semester 02

SCU
SLIIT CITY UNI

Course Identification

- Course Name: Design Analysis of Algorithms (DAA)
- Course Code: IT1205
- Enrolment key: **DAA@2023Oct**
- Credit Points: 4
- Duration: One Semester
- Delivery Plan
  - **Lectures – 2 Hours**
  - **Tutorial – 1 Hour**
  - **Labs – 2 hours**
- Notices
  - All notices will be published on Moodle unit page
  - Check the unit page regularly

Marks Distribution

Assessment Criteria	Marks
Mid Term Test (Written Paper)	***20 %***
Assignments (Code Based)	***20 %***
Final Examination (Written Paper)	***60 %***
Total	***100 %***

Course Contents

1. Big O, recurrences and Data structures
2. Divide and Conquer Method
3. Priority Queues and heaps
4. Graphs
5. Greedy algorithms
6. Data compression
7. String searching
8. Dynamic Programming
9. Matrix multiplication
10. Backtracking

Learning Outcomes

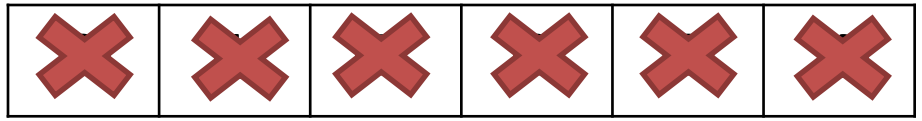
1. LO1: Cast problems in algorithmic terms
2. LO2: Identify efficient algorithms to solve problems
3. LO3: Construct own algorithms to solve problems
4. LO4: Analyze algorithms to determine resource requirements

ALGORITHMS

Algorithm is any well defined computational procedure that takes some value or set of values as input and produce some value or set of values as output.

ALGORITHMS (Cont.)

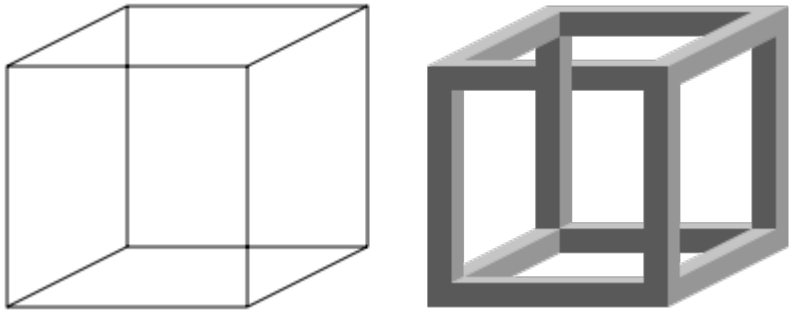
1. Get the smallest value from the input.
2. Remove it and place it at output.
3. Repeat above 1,2 steps for remaining input until there is no item in the input.



1	3	4	7	8	9

Properties of an Algorithm

- Be correct.
- Be unambiguous (Not open to more than one definition).
- Give the correct solution for all cases.
- Be simple.
- It must terminate.



Necker_cube_and_impossible_cube

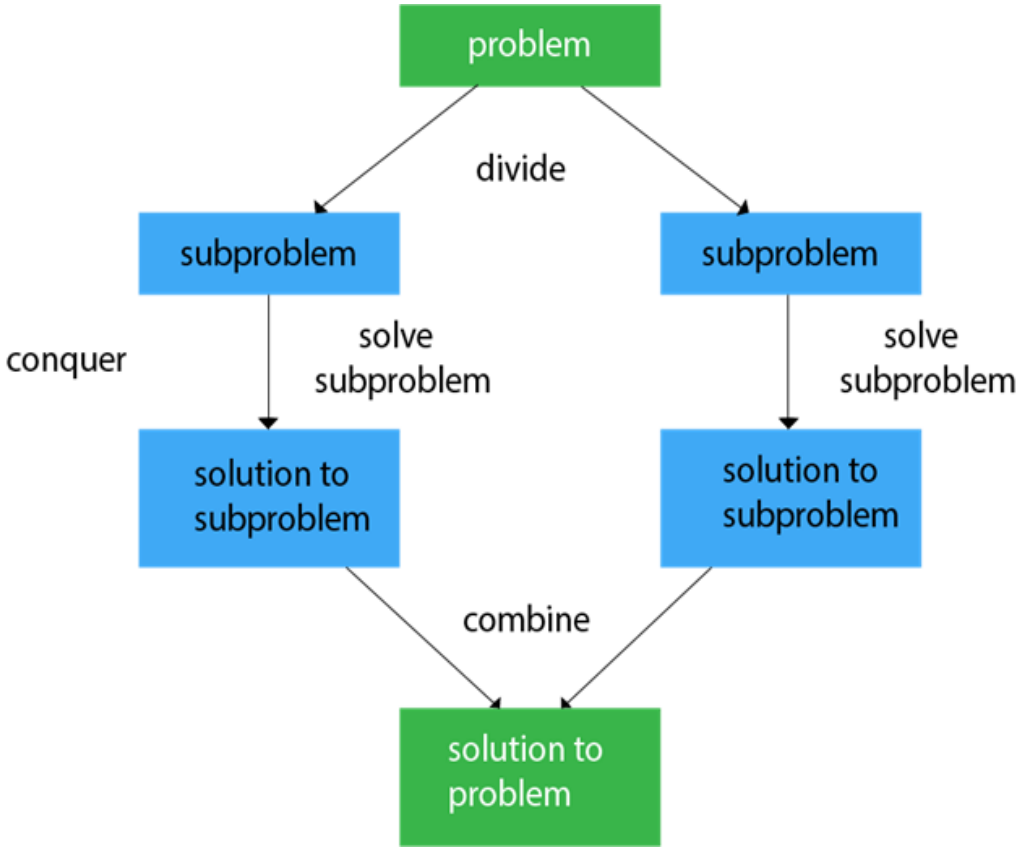
Source:http://en.wikipedia.org/wiki/Ambiguity#Mathematical_interpretation_of_ambiguity

Algorithm Design Techniques

1. Divide and Conquer Method
2. Dynamic Programming
3. Greedy Method
4. Backtracking

1. Divide and Conquer

- Based on multi-branched recursion
- Works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem



Divide & conquer method

1. The D&C involves three steps at each level of the recursion:
    - **Divide** the problem into a number of sub problems.
    - **Conquer** the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
    - **Combine** the solutions to the sub problems into the solution for the original problem.
2. D&C does more work on the sub-problems and hence has more time consumption.
3. In D&C the **sub problems are independent** of each other.

Example: Merge Sort, Binary Search

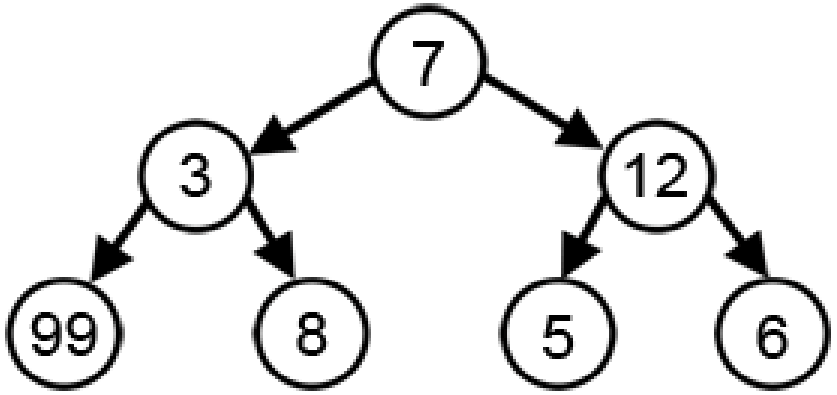
2. Dynamic programming

- Is both a mathematical optimization method and a computer programming method.
- **Saving results of sub-problems so that they do not need to be recomputed**. And can be used in solving other sub-problems.
- Dynamic Programming solves the sub problems only once and then stores it in the table.
- In DP the sub-problems are not independent.

Example : *Matrix chain multiplication*

3. Greedy Method

Follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum solution.



Advantages

- Easier to understand
- Perform better in best scenarios

Disadvantages

- Not accurate in all times
- Doesn't provide optimal solution in all time

4. Backtracking

The Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem

"systematically searches for a solution to a problem among all available option"

Advantages of above methods

- Provide a template.
- Translation to data structures is easy.
- The temporal and spatial requirements can be precisely analyzed.

Pseudocode

An **outline** of a **program**, **written** in a **form** that can easily be **converted** into **real programming** statements

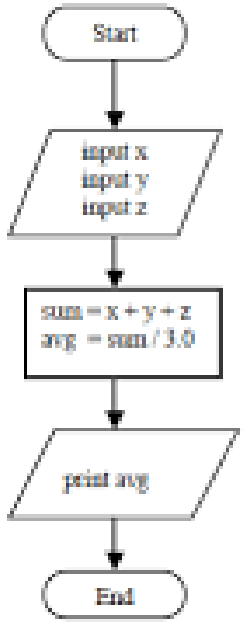
Pseudo code for calculating Average PR by sequences:

```
Procedure Average_by_sequences
Begin
1. For i ← 0 to Len(arr_temp) - 1 do
2. arr_avg[i] ← Average_Rows (arr_temp, Len(arr_temp))
End
```

Pseudo code for Average_Rows:

```
Function Double Average_Rows(Double arr[ ][ ],int n, int k)
Begin
1. Double S ← 0.0;
2. Double average ← 0.0;
3. For j ← 0 to Len(arr[k]) - 1 do
4.     Begin
5.         S ← S +  arr[k][j];
6.         average ← S / Len(arr[k]);
7.     End
 Return average;
 End
```

Average of 3 Numbers - sequence




```
Begin
    input x
    input y
    input z
    sum = x + y + z
    avg = sum / 3.0
    print avg
End
```

Pseudocode Conventions

- Give a valid name for the pseudo-code procedure.
- Proper Indentation.
- Separate line for each instruction.
- Looping constructs and conditional constructs.

  ex: Always end an **if** statement with an **end-if.**
  Always end a **for** loop with an **end-for.**
  Always end a **while** loop with an **end-while**

- ← indicate the assignment.

  ex: i← J

Pseudocode (Contd.)

- Array elements are accessed by specifying the array name followed by the index in the square bracket.

  ex: **A[i]** indicates the ***i*** th element of the array **A.**

- The notation "**..**" is used to indicate a range of values within the array.

  ex: **A[1..i]** indicates the sub array of A consisting of elements A[1],A[2],..,A[i].

Algorithm as Technology

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

- These differences can be much more significant than differences due to hardware and software.

Example for sorting problem:

For an example, let us pit a faster computer (**computer A**) running **insertion sort** against a slower computer (**computer B**) running **merge sort**. They **each must sort an array of 10 million numbers**(if numbers are 8byte integers :- 80 megabytes). Suppose that **computer A executes 10 billion instructions per second** and **computer B executes only 10 million instructions per second**.

Computer A is 1000 time faster than Computer B

Algorithm as .. (contd.)

Insertion sort (**IS**) takes time $2n^2$ to sort n numbers.
Merge sort (**MS**) takes time $50n \lg n$

- Assume CPU **A** runs **IS**, and CPU **B** runs **MS**
- **A** takes: $\frac{2 \cdot (10^7)^2 \, instructions}{10^{10} \, instructions/\sec} = 20000 \sec onds (> 5.5 hours)$
- **B** takes: $\frac{50 \cdot 10^7 \cdot \lg 10^7 \, instructions}{10^7 \, instructions/\sec} \approx 1163 \sec onds (< 20 \min utes)$

- *But computer A is 1000 times faster than computer B in raw computing power.*
- In general, as the problem size increases, so does the relative advantage of merge sort.

Analysis of Algorithms

Why do we need a analysis?

- To compare
- Predict the growth of run time

Idea is to predict the resource usage.

- Memory
- Logic Gates
- Computational Time

Analysis of Algorithms

- Study of computer program **performance** and resource usage.
- Efficiency measure:
  - **Speed**: How long an algorithm takes to produce results
  - **Space** requirement
  - **Memory** requirement

- Use the same **computation model** for the analyzed algorithms

Analysis of Algorithms

- In general, the time taken by an algorithm grows with the **size of the input.**
- Input size: depends of problems being studied (e.g. #elements, #nodes, #links).
- ***Running time*** on a particular input can measure with:
  - ***Number of primitive operations***
  - ***Steps executed.***

Analysis of Algorithms(Contd.)

2 ways of measuring Complexity.

- Space Complexity
- **Time Complexity**
  - **Operation Count**
  - **Step Count (RAM Model)**

Operation count

- Methods for time complexity analysis.
- Select one or more operations such as add, multiply and compare.
- Operation count: Omits accounting for the time spent on all but the chosen operations.

Operation count (Contd.)

Example 1

```
int sum = 0;                // 1 time
int i = 0;                  // 1 time
  while (i < n) {           // n+1 times
    sum++;                  // n times
    i++;                    // n times
  }
```

T(n) = 3n+3

Example 2

```
int sum = 0;                // 1 time
int i = 1;                  // 1 time
  while (i < n) {           // n times
    sum++;                  // n-1 times
    i++;                    // n-1 times
  }
```

T(n) = 3n

Operation count (Contd.)

- Calculate the running time as Operation count for the following process parts.

Activity 1

```
int i = 0;
while (i <=10) {
        print i;
        i=i+2;
 }
```

Activity 2

```
int i = 1;
while (i <10) {
        print i;
        i=i+2;
 }
```

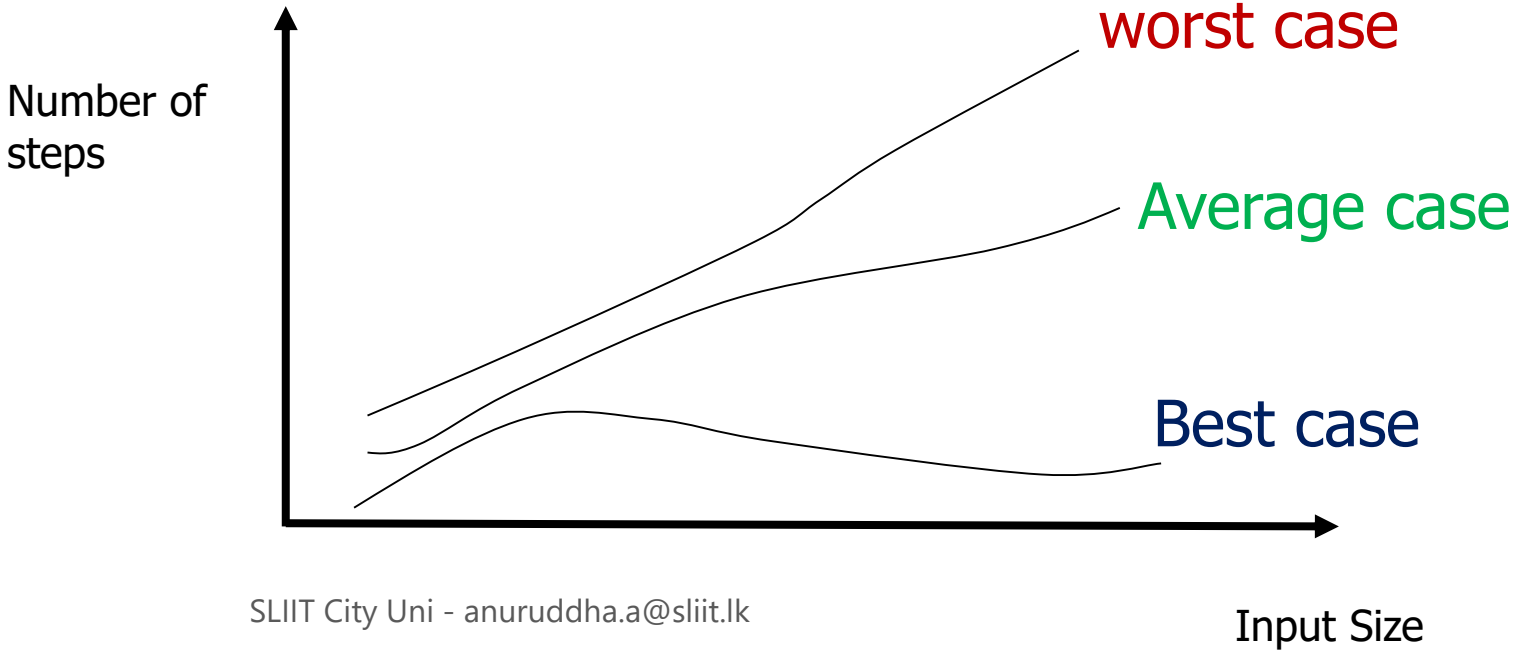
Activity 3

```
int sum = 0;
for (int i = 0; i < n;
i++)
{
        sum++;
}
```

Worst, Best and Average case

Running time will depend on the chosen instance characteristics.

- **Best case: <u>minimum number</u>** of steps taken on any instance of size n.
- **Worst case: <u>maximum number</u>** of steps taken on any instance of size n.
- **Average case: An <u>average number</u>** of steps taken on any instance of size n.



Step Count (RAM Model)

- Assume a generic one processor.
- Instructions are executed one after another, with no concurrent operations.
- ***+,-,=,it takes exactly one step***.
- ***Each memory access takes exactly 1 step***.
- Running Time T(n) = Sum of the steps.

RAM Model Analysis(Step Count)

n ← 100	1step
n ← n + 100	2steps
Print n	1step
	4steps

$sum \leftarrow 0$

$\text{for } i \leftarrow 0 \text{ to } n$

$sum \leftarrow sum + A[i]$

1 assignment

n+2 assignments
n+2 comparisons
n+1 additions

n+1 assignments
n+1 additions

n+1 memory accesses

Running Time T(n) = Steps 6n+9

Problems with RAM Model

- Differ number of steps with different architecture.

  ex: sum ← sum + A[i] is a one step in the CISC processor.

- It is difficult to count the exact number of steps in the algorithm.

  ex: See the insertion sort

Analysis of Insertion sort

- This is an efficient algorithm for *sorting small number of elements*.

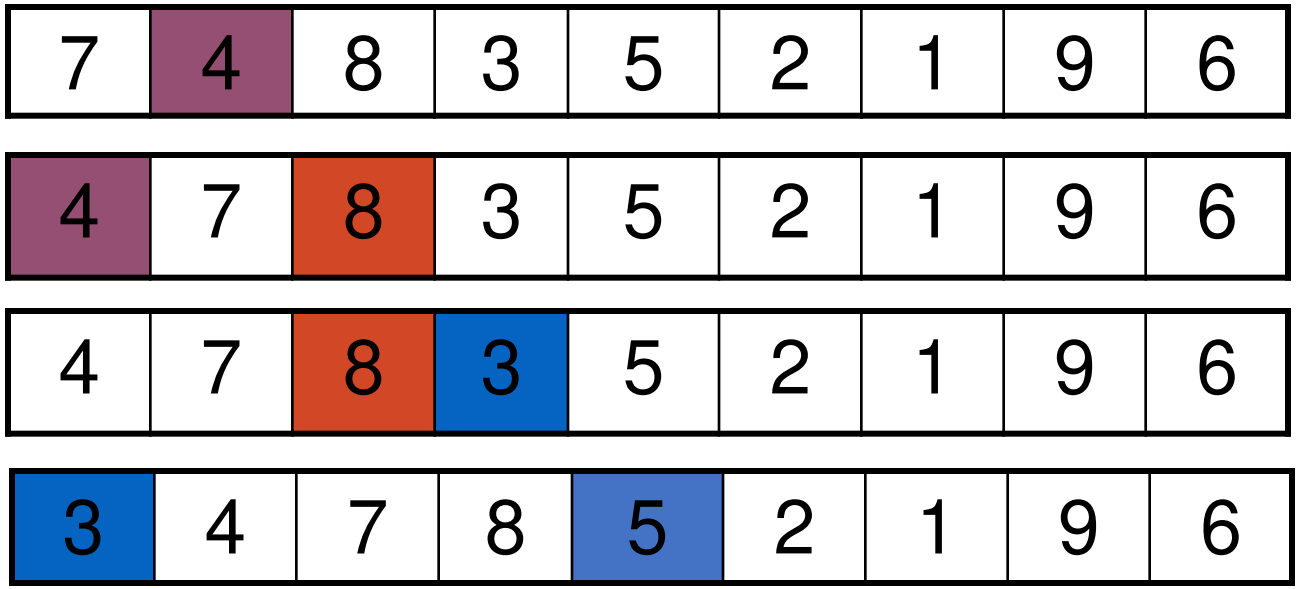
  ex: Sorting a hand of cards using insertion sort.

Pseudocode for insertion sort

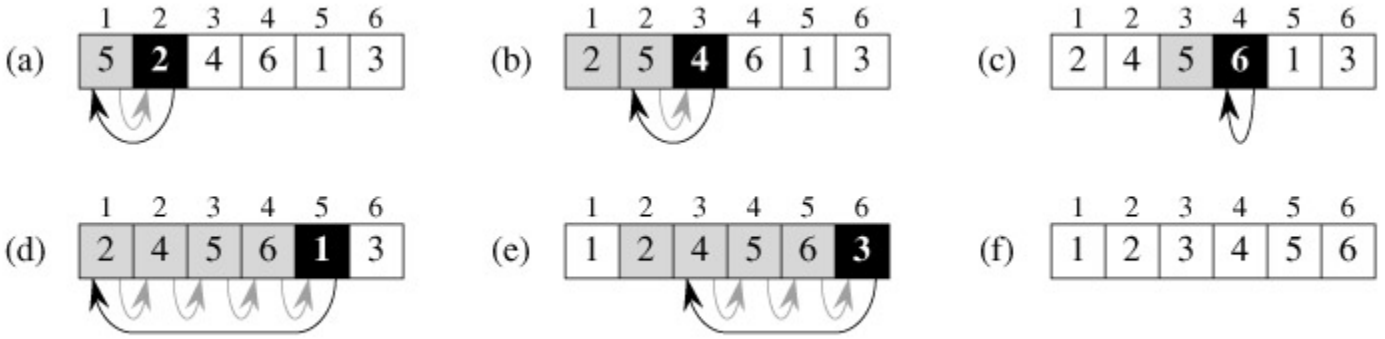
```
INSERTION-SORT(A)
1 for j ←2 to length[A]
2       do key ← A[j]
           ⊲ Insert A[j] into the sorted sequence A[1..j-1]
3        i ← j – 1
4        while i > 0 and A[i] > key
5                    do  A[i+1] ← A[i]
6                        i ←i-1
7         A[i+1] ← key
```

Example

Illustrate the operation of the Insertion sort on the array A=< 7 , 4 , 8 , 3 , 5 , 2 , 1 , 9 , 6 >



Example



The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$.

Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. *(a)-(e).*

The iterations of the *for* loop of lines 1-8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. *(f)* The final sorted array.

Exact analysis of Insertion sort

- Time taken for the algorithm will depend on the input size(number of elements of the array).

- Running Time (Time complexity): This is the number of primitive operations or steps executed of an algorithm on a particular input

Running Time : T(n)

INSERTION-SORT(A)	*cost*	*times*
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 $A[1..j-1]$.	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	$n-1$

Running Time : T(n)

- 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.
- To compute T(n), the running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times columns, obtaining

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

Running Time(contd.)

- 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 each j = 2,3,…,n, we then find that A[i] ≤ key in line 5 when i has its initial value of j - 1. Thus t_j = 1 for j=2,3,…,n, and the best-case running time is

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

- **Best Case T(n)→ an+b**

Worst Case T(n) $\rightarrow an^2 + bn + c$

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare key with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.

Worst Case T(n) $\rightarrow an^2 + bn + c$

- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.
  [The parentheses around the summation are not strictly necessary. They are there for clarity, but it might be a good idea to remind the students that the meaning of the expression would be the same even without the parentheses.]
- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

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

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Worst Case T(n) $\rightarrow an^2 + bn + c$

Insertion sort worst case time based on

- ***Relative speed(on same machine)***
- ***Absolute speed (on different machine)***

Summary

- What is an algorithm?
- Properties.
- Design methods.
- Pseudocode.
- Analysis(Operation count & Step count).
- RAM model.
- Insertion Sort.


SEE YOU
next
week!