BCSE204L - DESIGN AND ANALYSIS OF ALGORITHM

Course Objectives

1. To provide **mathematical foundations** for **analyzing the complexity** of the algorithms
2. To impart the knowledge on **various design strategies** that can help in **solving the real world problems effectively**
3. To synthesize **efficient algorithms** in various engineering **design situation**

Course Outcomes

On completion of this course, student should be able to:

1. Apply the mathematical tools to analyze and derive the running time of the algorithms
2. Demonstrate the major algorithm design paradigms.
3. Explain major graph algorithms, string matching and geometric algorithms along with their analysis.
4. Articulating Randomized Algorithms.
5. Explain the hardness of real-world problems with respect to algorithmic efficiency and learning to cope with it.

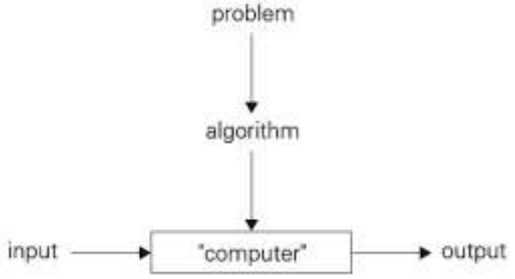
Module 1

1. Overview and Importance of Algorithms - Stages of algorithm development
2. Describing the problem, Identifying a suitable technique, Design algo and time complexity
3. Proof of Correctness of the algorithm, Illustration of Design Stages
4. Greedy techniques- Fractional Knapsack
5. Huffman coding - Divide and Conquer introduction
6. Maximum Subarray, Karatsuba faster integer multiplication algorithm

Why I should?

What is an Algorithm???

- ✓ An algorithm is a **sequence of unambiguous instructions** for solving a pr oblem, i.e., for obtaining a required output for any legitimate input in a fin ite amount of time.
- ✓ The **nonambiguity requirement** for each step of an algorithm cannot be compromised
- ✓ The **range of inputs** for which an algorithm works has to be specified carefully.
- ✓ The same algorithm can be represented in **several different ways**.
- ✓ There may **exist several algorithms** for solving the same problem
- ✓ Algorithms for the same problem can be **based on very different ideas** and can solve the probl em with dramatically **different speeds**.



Algorithm

- ✓ Algorithms are generally developed independently of underlying languages, which means that an algorithm can be implemented in more than one program ming language.
- ✓ **Writing an Algorithm** : **Input, Process and Output**

  Input : {a1,a2,a3….an}

  Output : {a1`,a2`,a3`….an`}

  Logic : previous value lesser than next value (sort)

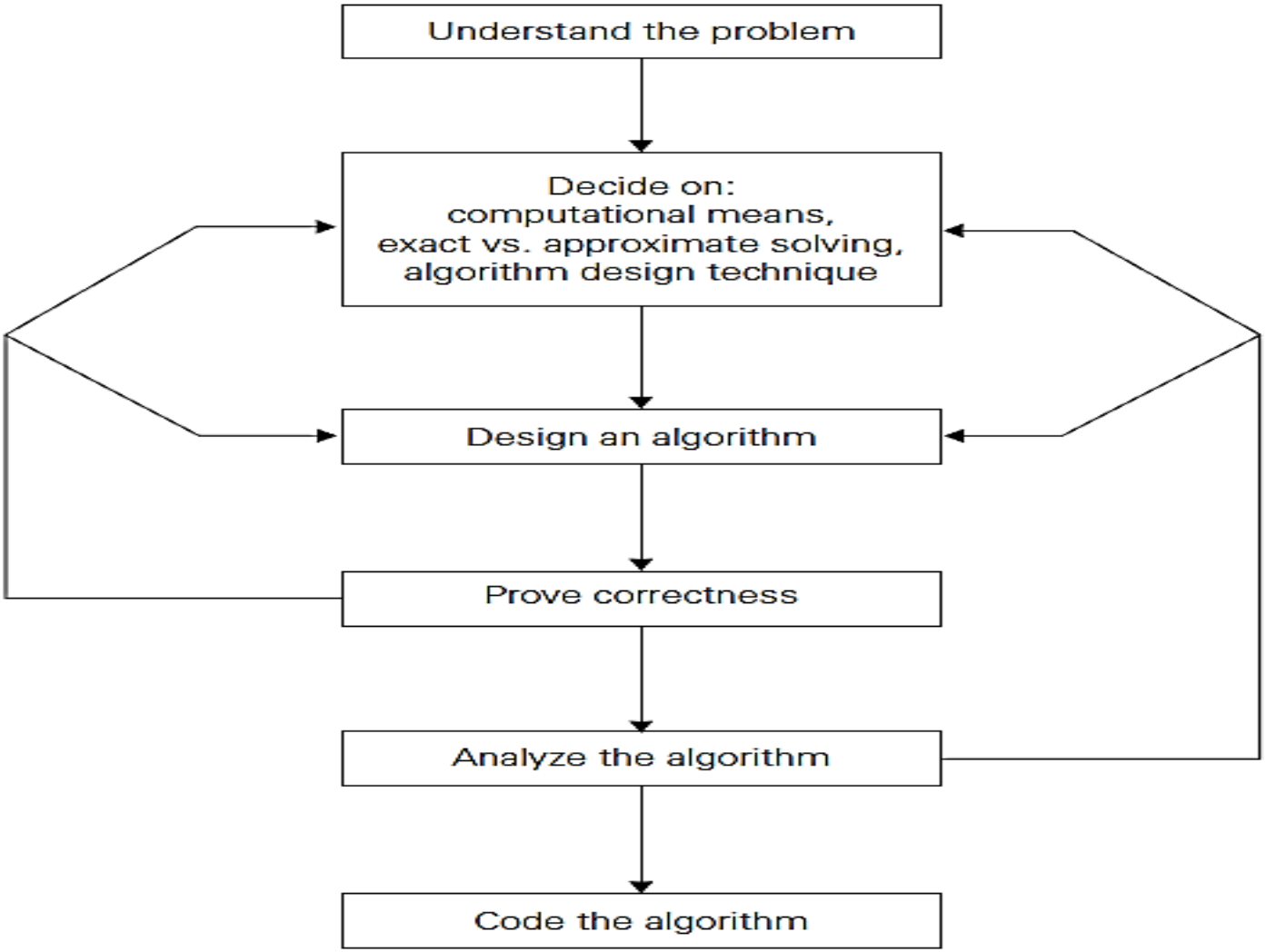
PSEUDOCODE

- ✓ Methods of implementing this step by step procedure
- ✓ **Non syntax, informal language** to develop an algorithm
- ✓ **Natural/ English + Mathematical logic**
- ✓ Simplified form of algorithm steps with logic understanding
- ✓ **High level description** of writing an algorithm

DIFFERENCE

Algorithm	Pseudo code
Step by step instruction to solve the problem	High level algorithm
Special kind of solution to the problem	Methods of writing algorithm with logic
Not required to specify aim, input, output	Specify aim, input, output in the beginning
No specific syntax rules. Doesn't raise error and not required to execute	No Specific syntax rules, does not raise error and not required to execute
Does not executed by computer	Executed by computer
Written in natural language	Written in natural language + Mathematical logic

FUNDAMENTALS OF PROBLEM SOLVING


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

Problem Solving

We can consider algorithms to be procedural solutions to problems.

These solutions are not answers but specific instructions for getting answers.

1. Understand the Problem

- What are the problem objects?
- What are the operations applied to the objects?

2. Deciding on Computational means:

- How the objects would be represented?
- How the operations would be implemented?

3. Design an Algorithm

Build a computational model of the solving process

Problem Solving

4. Prove Correctness

- Correct output for every legitimate input in finite time
- Based on correct math formula
- By induction

5. Analyze the Algorithm

Efficiency: Time and Space (Simplicity)
Generality: Range of inputs, Special cases
Optimality: No other algorithm can do better

6. Code the Algorithm

How the objects and operations in the algorithm are represented in the chosen programming language?

Problem Solving – Example 1

A peasant finds himself on a riverbank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room for only the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Solve this problem for the peasant or prove it has no solution. (Note: The peasant is a vegetarian but does not like cabbage and hence can eat neither the goat nor the cabbage to help him solve the problem. And it goes without saying that the wolf is a protected species.)

Solution..

Initial State:

Left Bank: Peasant, Wolf, Goat, Cabbage

Right Bank: -

Move 1:

Peasant takes Goat to the Right Bank.

Left Bank: Peasant, Wolf, Cabbage

Right Bank: Goat

Move 2:

Peasant returns to the Left Bank alone.

Left Bank: Peasant, Wolf, Cabbage

Right Bank: Goat

Move 3:

Peasant takes Wolf to the Right Bank.

Left Bank: Peasant, Cabbage

Right Bank: Goat, Wolf

Move 4:

Peasant takes Goat back to the Left Bank.

Left Bank: Peasant, Goat, Cabbage

Right Bank: Wolf

Solution..

Move 5:

Peasant takes Cabbage to the Right Bank.
Left Bank: Peasant, Goat
Right Bank: Wolf, Cabbage

Move 6:

Peasant returns to the Left Bank alone.
Left Bank: Peasant, Goat
Right Bank: Wolf, Cabbage

Move 7:

Peasant takes Goat to the Right Bank.
Left Bank: Peasant
Right Bank: Goat, Wolf, Cabbage

Final State:

Left Bank: Peasant
Right Bank: Wolf, Goat, Cabbage

```
crossRiver() {
 // Initial state: A: [P, W, G, C], B: []
        ("Initial State: A: [P, W, G, C], B: []\n");
// Step 1: Peasant takes Goat to side B
            ("Move: P + G -> B\n");
            ("State: A: [P, W, C], B: [G]\n");
// Step 2: Peasant returns alone to side A
            ("Move: P -> A\n");
            ("State: A: [P, W, C], B: [G]\n");
……….
```

Problem Types

1. **Sorting :** Eg. Insertion sort, Merge sort, Bubble sort ..
2. **Searching :** Eg. Binary search, Linear search ..
3. **String processing :** Eg. String matching, pattern matching..
4. **Graph problems :** Eg. DFS,BFS..
5. **Combinatorial problems** Eg. Travelling salesman problem..
6. **Geometric problems** Eg. Closest pair, Convex hull..
7. **Numerical problems** Eg. Solving equations, evaluating functions..

Algorithmic Techniques/ Design Methods

1. **Divide and conquer**
2. **Brute Force**
3. **Greedy Techniques**
4. **Dynamic Programming**
5. **Decrease & Conquer**
6. **Backtracking**
7. **Branch and bound**

Performance Analysis of an Algorithm

- ✓ Similarly, in computer science, there are **multiple algorithms to solve a problem.**
- ✓ When we have more than one algorithm to solve a problem, we need to **select the best one (exact solution)**
- ✓ **Performance analysis** helps us to select the best algorithm from multiple algorithms to solve a problem.
- ✓ To **compare algorithms**, we use a set of parameters or set of elements like **memory required** by that algorithm, the **execution speed** of that algorithm, **easy to understand, easy to implement**, etc.,

Analysis of an Algorithm Efficiency

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

1. Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm. It includes program space and data space (**Space Complexity**).
2. The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution(**Time Complexity**)

Space Complexity

When a program is under execution it uses the computer memory for THREE reasons. They are as follows...

Instruction Space: It is the amount of memory used to store compiled version of instructions.

Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.

Data Space: It is the amount of memory used to store all the variables and constants

Space Complexity

Note: When we want to perform an analysis of an algorithm based on its Space complexity, we consider **only Data Space and ignore Instruction Space** as well as Environmental Stack.

That means we calculate only the memory required to store Variables, Constants, Structures, etc.,

To calculate the space complexity, we must know the memory required to store different data type values (according to the compiler).

For example, the C Programming Language compiler requires the following...

- 2 bytes to store Integer value.
- 4 bytes to store Floating Point value.
- 1 byte to store Character value.
- 8 bytes to store double value


```
int square(int a)
{
return a*a;
}
```

- It requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.
- Totally it requires 4 bytes of memory to complete its execution.
- This 4 bytes of memory is fixed for any input value of 'a'.
- This space complexity is said to be Constant Space Complexity.


```
int sum(int A[ ], int n)
{
int sum = 0, i;
for(i = 0; i < n; i++)
sum = sum + A[i];
return sum;
}
```

'n*2' bytes of memory to store array variable a[ ] - 2n
2 bytes of memory for integer parameter ‘n’ - 2
4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each) - 4
2 bytes of memory for return value. – 2

Totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n’.

As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be Linear Space Complexity. – O(n)

Time Complexity

- Every algorithm requires **some amount of computer time** to execute its instruction to perform the task. This computer time required is called time complexity.
- The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- Factors – Speed of computer, compiler used, **Input size and outputs**
- Algorithm running time expressed in **seconds** or **milliseconds**
- How much times the algorithm basic operation is executed on input size ‘n’

$$T(n) = C_{op} \cdot C(n)$$

C_{op} – Basic Operation

C(n) – Input Size

Asymptotic Notations

Refers to defining the mathematical notation of its run-time performance based on the input size. - **Order of growth**

Analysis based on three cases

- ✓ Best Case (Omega Notation (Ω)) – Lower bound : Minimum time required for an algorithm to complete when given the optimal input. It denotes an algorithm operating at its peak efficiency under ideal circumstances.

- ✓ Average Case (Theta Notation (Θ)) : This estimates the typical running time of an algorithm when averaged over all possible inputs. It provides a more realistic evaluation of an algorithm's performance.

- ✓ Worst Case (O Notation(O)) – Upper bound : This denotes the maximum time an algorithm will take to finish for any given input. It represents the scenario where the algorithm encounters the most unfavourable input.

Time Complexity

Steps

- Break the program into smaller segments.
- Find the number of operations performed for each segment(in terms of the input size) assuming the given input is such that the program takes the least amount of time.
- Add up all the operations and simplify it, let's say it is f(n).
- Remove all the constants and **choose the term having the highest order**

❖ Frequently expressed using **Big O Notation.**

❖ It represents the maximum possible running time for an algorithm given the size of the input.

Time Complexity

Time Complexity	Algorithm
O(1)	Looking up a specific element in an array, like this for example: print(my_array[97]) No matter the size of the array, an element can be looked up directly, it just requires o ne operation.
O(n)	Finding the lowest value. The algorithm must do n operations in an array with n value s to find the lowest value, because the algorithm must compare each value one time.
$O(n2)$	Bubble sort, Selection sort and Insertion sort are algorithms with this time complexity. Large data sets slows down these algorithms significantly. With just an increase in n from 100 to 200 values, the number of operations can increase by as much as 30000!
O(nlogn)	The Quicksort algorithm is faster on average than the three sorting algorithms mentioned above, with O(nlogn) being the average and not the worst case time. Worst case time for Quicksort is also $O(n2)$, but it is the average time that makes Quicksort so interesting.

Examples

Find the value 70 in the given array

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Search Key value is present in the first position.

As there is only one operation required to return the first element of the array, the function has an O(1) time complexity.

Examples

Find the value 52 in the given array

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

According to **linear time complexity**, the running time grows linearly with the size of the input.

Search Key value is present in last position. So the loop iterates till the size of input, n.

Time complexity is O(n)

Examples

```
int binarySearch(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
   return -1; // Not found
}
```

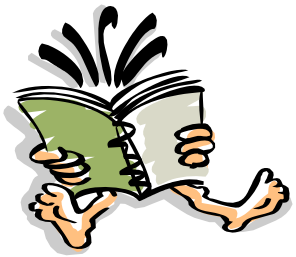
Logarithmic time complexity, the execution time increases logarithmically as the input size increases.

This complexity are often associated with efficient searching or dividing problems in half at each step.

O(log n)

Analysis of Algorithms
Sorting

Sorting – Insertion Sort



The 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 $a_1' \leq a_2' \leq \cdots \leq a_n'$

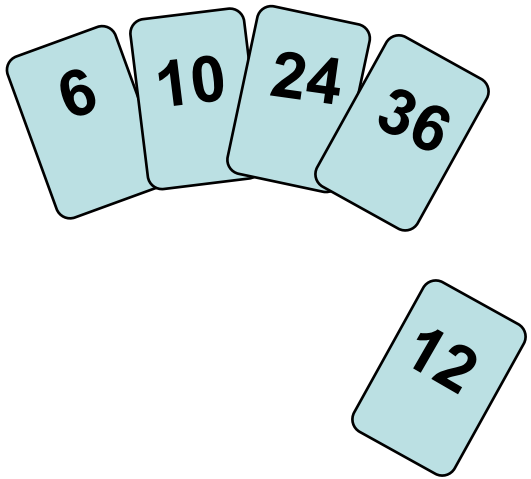
Insertion Sort

- Understanding
- Design / Logic
- Illustration
- Pseudocode
- Proof of Correctness (Loop Invariant)
- Complexity Analysis

Insertion Sort

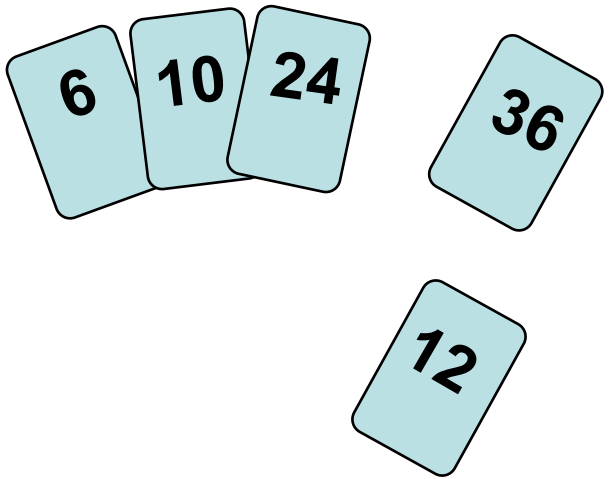
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

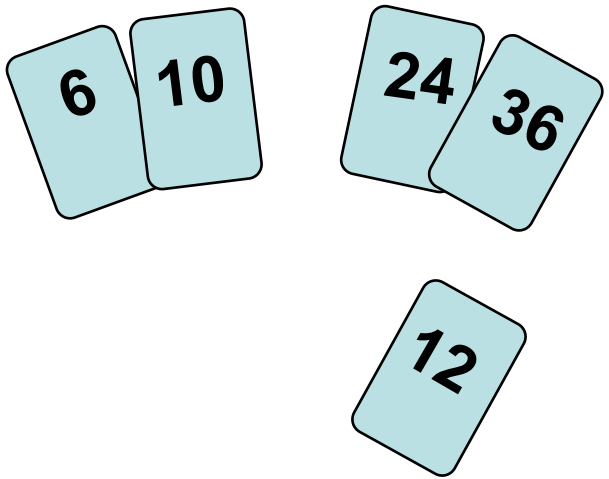


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Demo of Insertion Sort

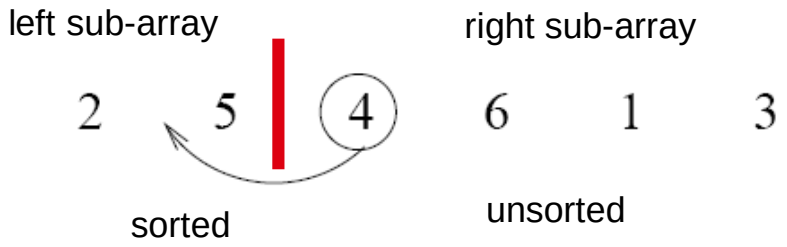
6 5 3 1 8 7 2 4

Insertion Sort

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:



Illustration

1. We will start by assuming the very first element of the array is already sorted. Inside the **key**, we will store the second element.

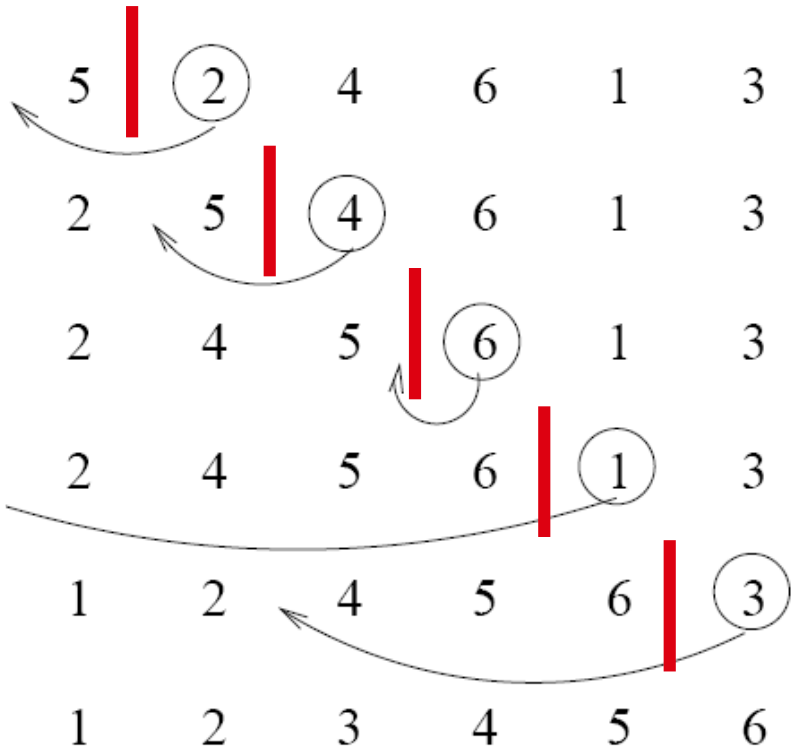
- Next, we will compare our first element with the **key**, such that if **the key** is found to be smaller than the first element, we will interchange their indexes or place the key at the first index. After doing this, we will notice that the first two elements are sorted.

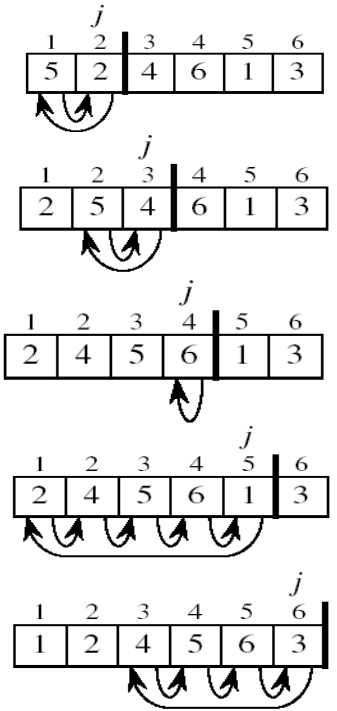
Illustration (Contd..,)

2. Now, we will move on to the third element and compare it with the left-hand side elements. If it is the smallest element, then we will place the third element at the first index.

- Else if it is greater than the first element and smaller than the second element, then we will interchange its position with the third element and place it after the first element. After doing this, we will have our first three elements in a sorted manner.

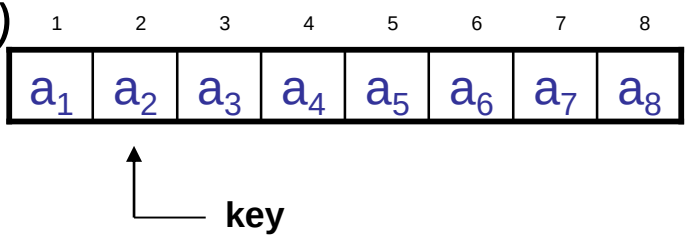
Insertion Sort





INSERTION-SORT - Algorithm

Alg.: INSERTION-SORT(*A*)



for j ← 2 **to** n

do key ← *A*[j]

▷ Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

i ← j - 1

while i > 0 and *A*[i] > key

do *A*[i + 1] ← *A*[i]

i ← i – 1

A[i + 1] ← key

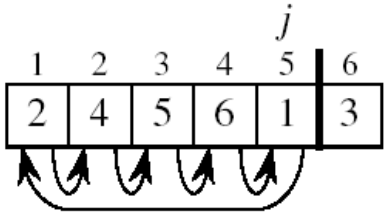
- Insertion sort – sorts the elements in place

Loop Invariant for Insertion Sort (POC)

Alg.: INSERTION-SORT*(A)*

for j ← 2 **to** n

do key ← A[j]



Insert A[j] into the sorted sequence A[1 . . j -1]

i ← j - 1

while i > 0 and A[i] > key

do A[i + 1] ← A[i]

i ← i – 1

A[i + 1] ← key

Invariant: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

Proving Loop Invariants

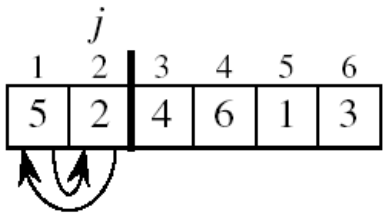
- Proving loop invariants works like induction
- **Initialization (base case):**
  - It is true prior to the first iteration of the loop
- **Maintenance (inductive step):**
  - If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:**
  - When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
  - Stop the induction when the loop terminates

Loop Invariant for Insertion Sort

- **Initialization:**
  - Just before the first iteration, j = 2:

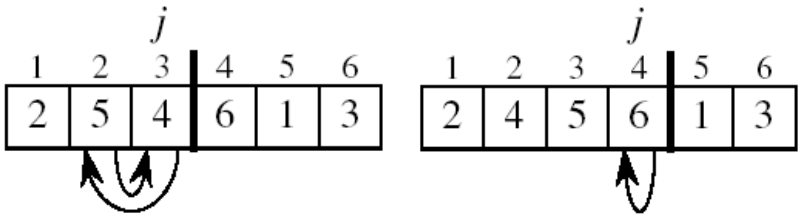
    the subarray $A[1 \ldots j-1] = A[1]$,

    (the element originally in $A[1]$) – is sorted



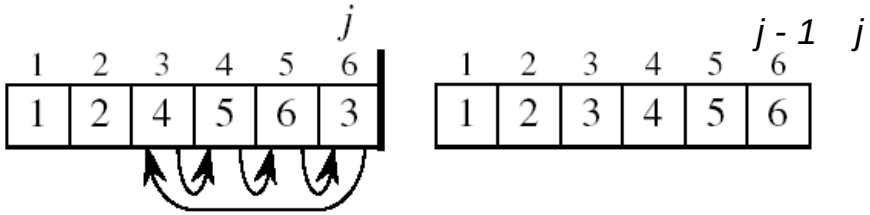
Loop Invariant for Insertion Sort

- **Maintenance:**
  - the **while** inner loop moves $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for **key** (which has the value that started out in $A[j]$) is found
  - At that point, the value of **key** is placed into this position.



Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when $j = n + 1 \Rightarrow j-1 = n$
  - Replace n with $j-1$ in the loop invariant:
    - the subarray $A[1 . . n]$ consists of the elements originally in $A[1 . . n]$, but in sorted order



- The entire array is sorted!

Invariant: at the start of the **for** loop the elements in $A[1 . . j-1]$ are in sorted order

Analysis of Insertion Sort

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

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

Best Case Analysis

- The array is already sorted **"while** i > 0 and A[i] > key"
  - $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

  $= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

  $= an - b = \Theta(n)$

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

Worst Case Analysis

- The array is in reverse sorted order **"while** i > 0 and A[i] > key"
  - Always A[i] > key in **while** loop test
  - Have to compare key with all elements to the left of the j-th position ⇒ compare with j-1 elements ⇒ $t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

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

Complexity Analysis

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **$O(n^2)$**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.

Complexity Analysis

- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the ascending order of an array into the descending order. In this algorithm, every individual element is compared with the rest of the elements, due to which n-1 comparisons are made for every n^{th} element.

- The insertion sort algorithm is highly recommended, especially when a few elements are left for sorting or in case the array encompasses few elements.

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

$\approx n^2/2$ comparisons

$\approx n^2/2$ exchanges

Insertion Sort

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case and Average Case: $\Theta(n^2)$

GREEDY ALGORITHM

Always makes choice that looks best at the moment

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.
- At each decision point, the algorithm makes choice that seems best at the moment.
- **Greedy choice property**: We can assemble a globally optimal solution by making locally optimal (greedy) choices.

If considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

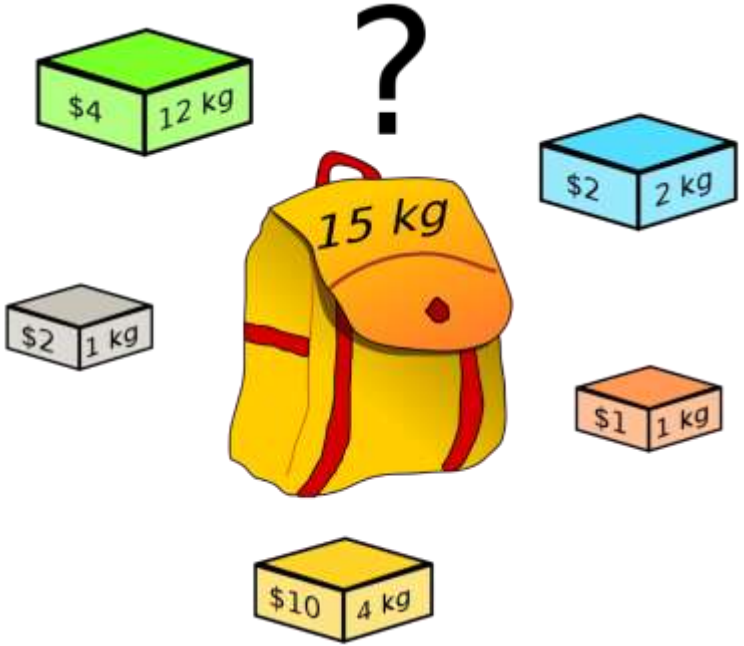
Example: Cash change problem, Eating pizza, Reading emails, Knapsack problem, Huffman coding, activity selection etc…

KNAPSACK PROBLEM

0-1 Knapsack problem

Fractional Knapsack problem

Eg: Thief robbing a store; n items, v-value, w-weight, W- sack holding pounds



KNAPSACK PROBLEM

Given Data

Items: 1, 2, 3, 4, 5, 6, 7

Values: 10, 5, 15, 7, 6, 18, 3

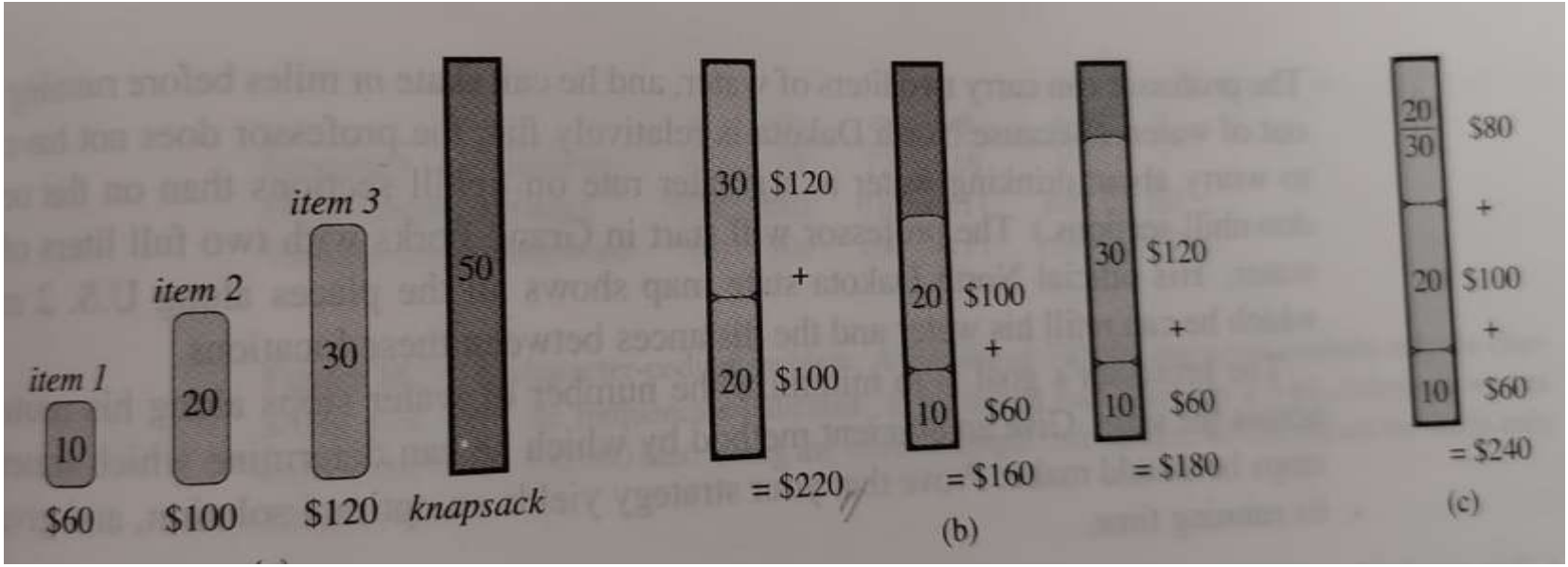
Weights: 2, 3, 5, 7, 1, 4, 1

Knapsack capacity: 15 kg

Steps

1. Calculate the Value-to-Weight Ratio for each item.
2. Sort Items by Value-to-Weight Ratio in descending order.
3. Select Items to maximize value within the given weight constraint.

KNAPSACK PROBLEM



KNAPSACK PROBLEM

Item	Value	Weight	Value/ Weight Ratio
1	10	2	5
2	5	3	1.67
3	15	5	3
4	7	7	1
5	6	1	6
6	18	4	4.5
7	3	1	3

Item	Value	Weight	Value/Weig ht Ratio
5	6	1	6
1	10	2	5
6	18	4	4.5
3	15	5	3
7	3	1	3
2	5	3	1.67
4	7	7	1

KNAPSACK PROBLEM

Add Item 5: Weight = 1 kg, Value = 6 (Total Weight = 1 kg, Total Value = 6)

Add Item 1: Weight = 2 kg, Value = 10 (Total Weight = 3 kg, Total Value = 16)

Add Item 6: Weight = 4 kg, Value = 18 (Total Weight = 7 kg, Total Value = 34)

Add Item 3: Weight = 5 kg, Value = 15 (Total Weight = 12 kg, Total Value = 49)

Add Item 7: Weight = 1 kg, Value = 3 (Total Weight = 13 kg, Total Value = 52)

Add Item 2 (fraction): The remaining weight capacity is 2 kg out of 3 kg of Item 2.

- Fraction of Item 2: 2/3 Item 2's weight and value.
- Weight = 2 kg, Value = $5 \times 23 = 10/3 \approx 3.33$

(Total Weight = 15 kg, Total Value = 55.33)

Based on sorting – Insertion Sort – BC – O(n), WC – O(n^2)
Quick Sort – BC-O(n log n) , WC – O(n^2)
Merge Sort – O(n log n)

HUFFMAN CODING

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.
- Huffman Coding is generally useful to **compress the data** in which there are **frequently occurring characters**.

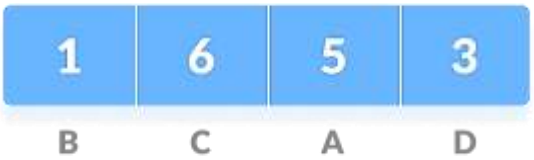
There are mainly two major parts in Huffman Coding

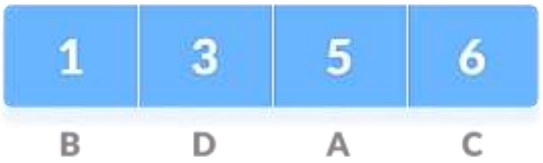
- ✓ Build a Huffman Tree from input characters.
- ✓ Traverse the Huffman Tree and assign codes to characters

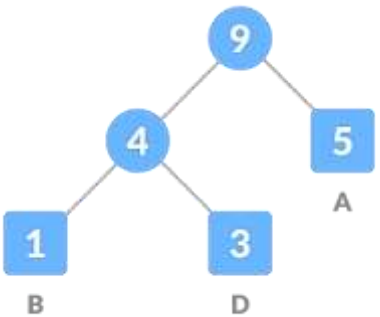
HUFFMAN -ENCODING

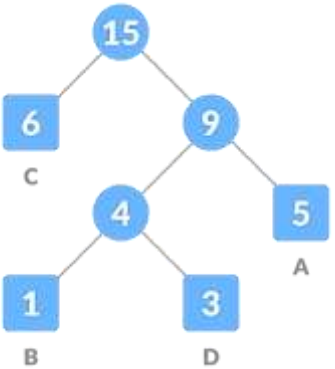
Each character occupies 8 bits.

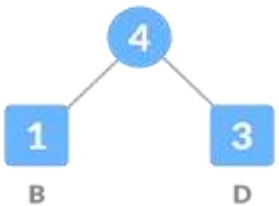
Total 15 characters, 15*8=120 bits





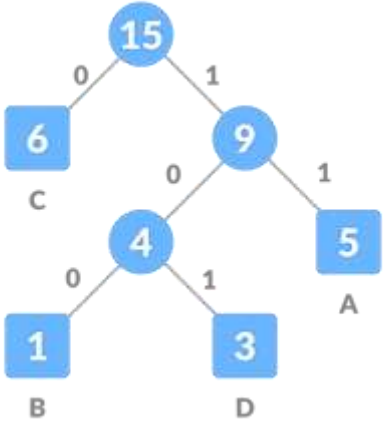






HUFFMAN ENCODING

For each non-leaf node, assign 0 to the left edge and 1 to the right edge



Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

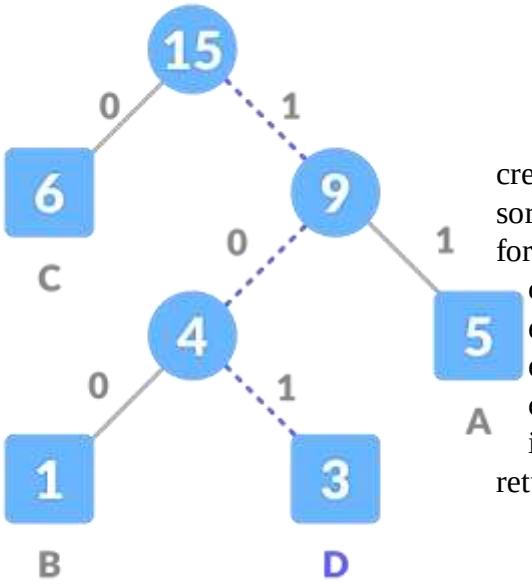
Without encoding, the total size of the string was 120 bits.

After encoding, 32+15+28=75 bits.

HUFFMAN –DECODING

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



create a priority queue Q consisting of each unique character.
sort then in ascending order of their frequencies.
for all the unique characters:
    create a newNode
    extract minimum value from Q and assign it to leftChild of newNode
    extract minimum value from Q and assign it to rightChild of newNode
    calculate the sum of these two minimum values and assign it to the value of newNode
    insert this newNode into the tree
return rootNode

HUFFMAN TIME COMPLEXITY

The time complexity for encoding each unique character based on its frequency **is O(n log n).**

Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is **O(log n).**

Thus the overall complexity is **O(n log n).**

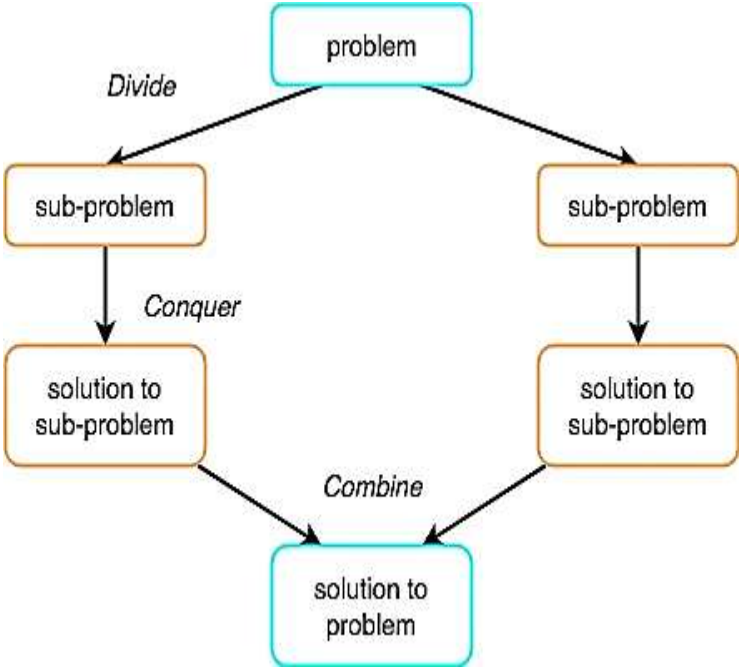
DIVIDE AND CONQUER

Break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

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

- Understanding
- Design / Logic
- Illustration
- Pseudocode
- Proof of Correctness (Loop Invariant)
- Complexity Analysis

Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

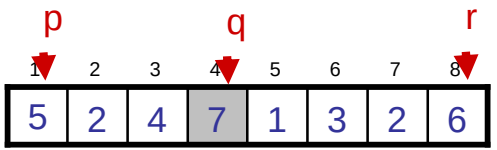
Merge Sort Approach

- To sort an array $A[p \ldots r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Demo of Merge Sort

6 5 3 1 8 7 2 4

Merge Sort



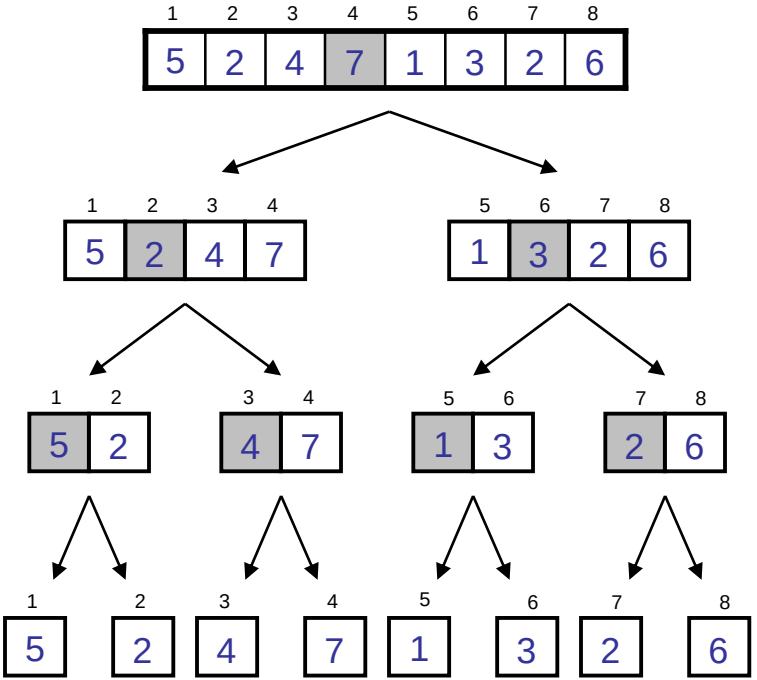
Alg.: MERGE-SORT(A, p, r)

```
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```

- Initial call: MERGE-SORT(A, 1, n)

Example – n Power of 2

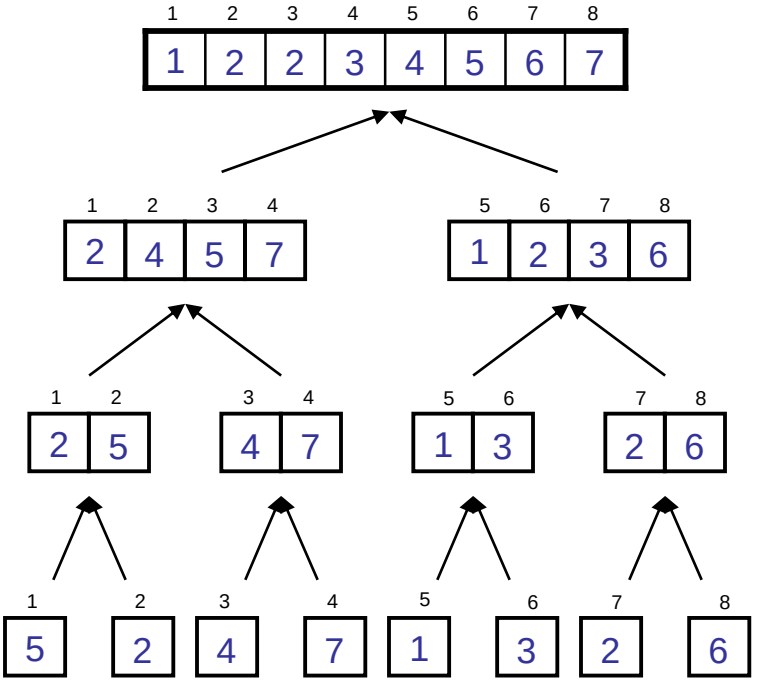
Divide



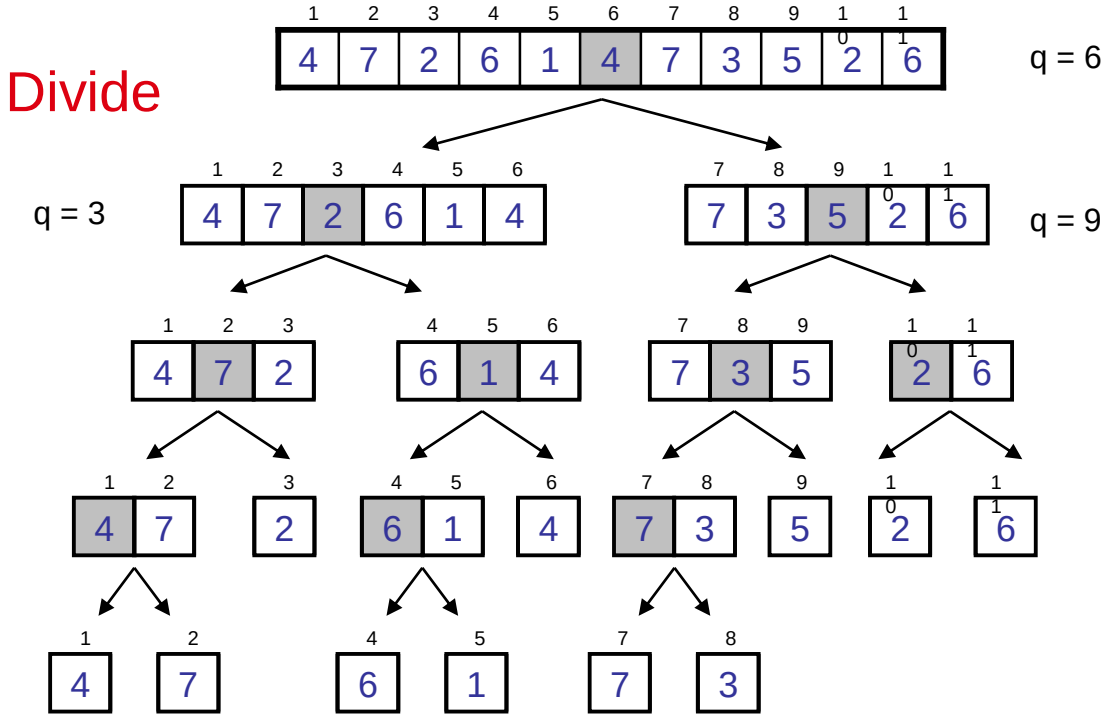
q = 4

Example – n Power of 2

Conquer and Merge

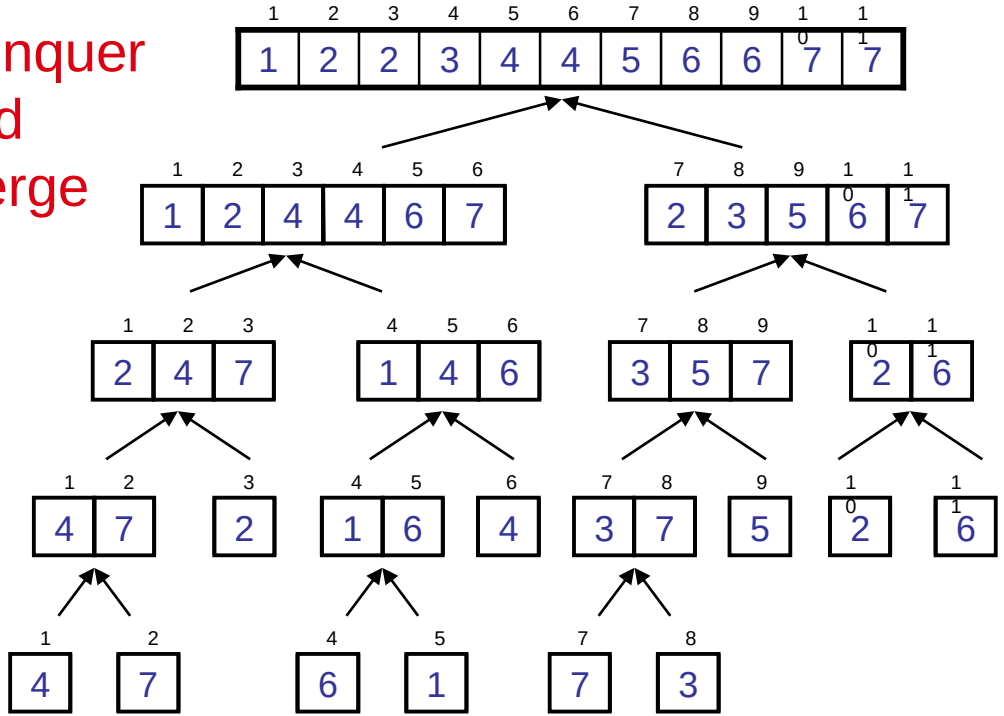


Example – n Not a Power of 2

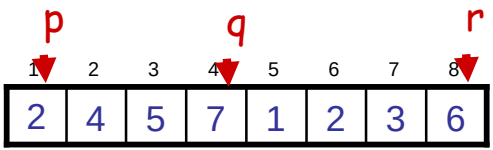


Example – n Not a Power of 2

Conquer and Merge

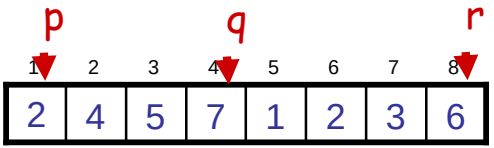


Merge Sort - Understanding

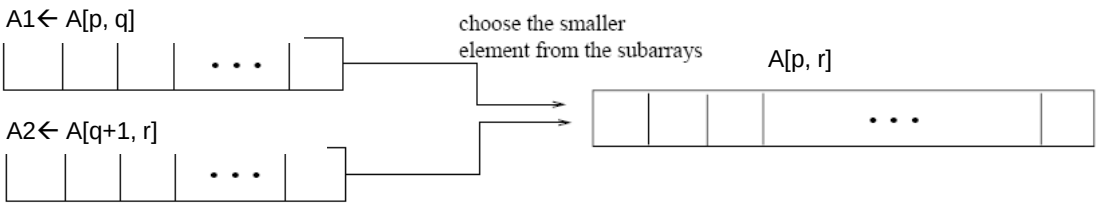


- **Input:** Array A and indices p, q, r such that $p \le q < r$
  - Subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are sorted
- **Output:** One single sorted subarray $A[p\,..\,r]$

Merge Sort - Logic



- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile



Example: MERGE(A, 9, 12, 16)

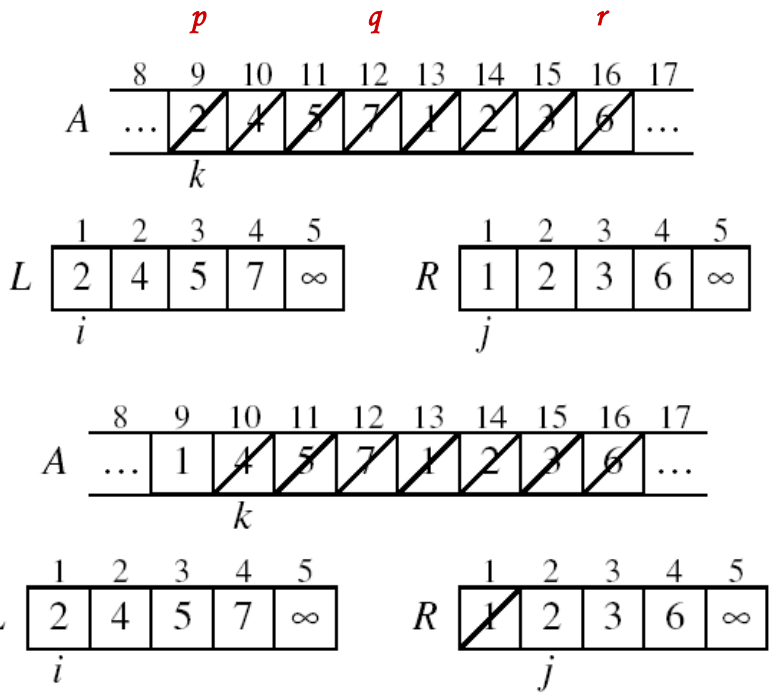
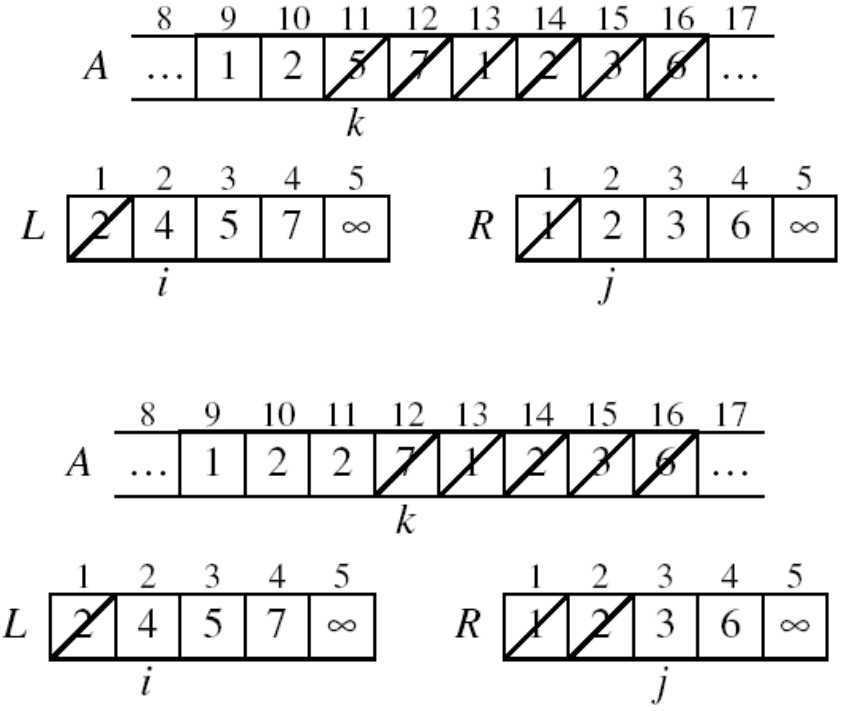
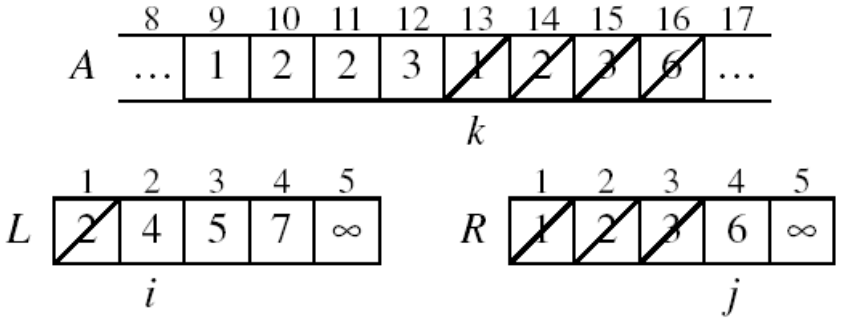
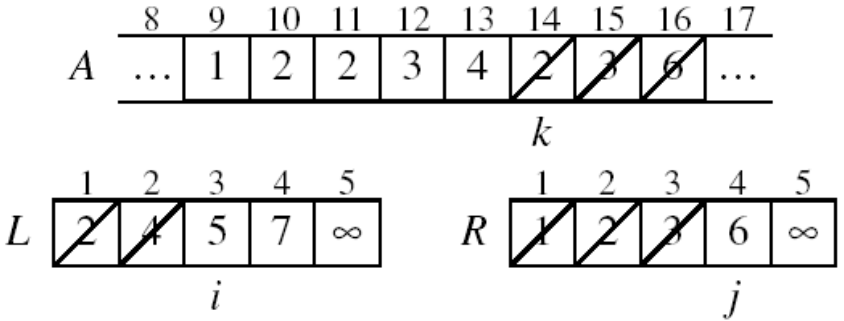


Illustration with Example: MERGE(A, 9, 12, 16)

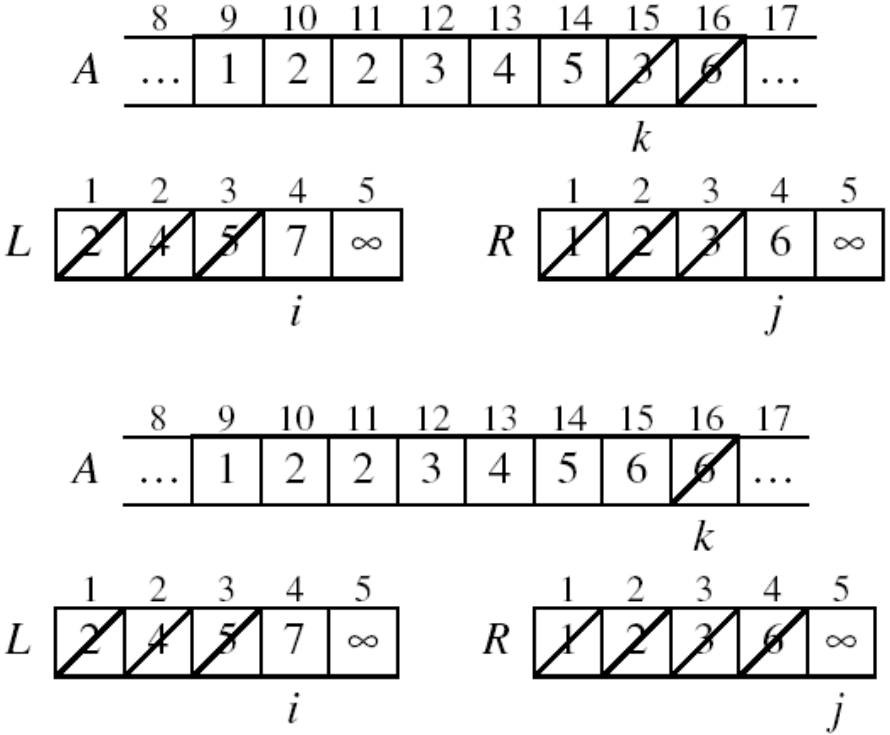


Example (cont.)

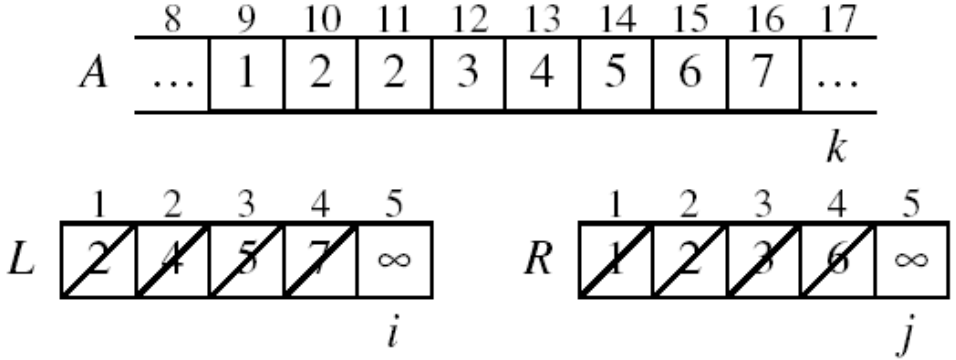




Example (cont.)

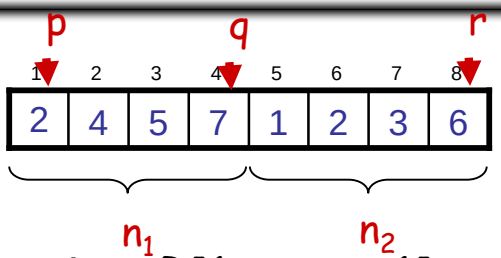


Example (cont.)



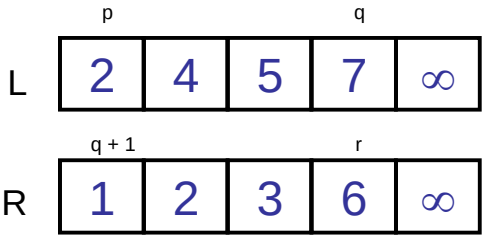
Done!

Merge - Pseudocode



Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \,..\, n_1 + 1]$ and the next n_2 elements into $R[1 \,..\, n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\, i \,] \leq R[\, j \,]$
7. **then** $A[k] \leftarrow L[\, i \,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\, j \,]$
10. $j \leftarrow j + 1$



Alg.: MERGE-SORT(A, p, r)

if p < r

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

MERGE-SORT(A, p, q) ---------- $T(n/2)$

MERGE-SORT(A, q + 1, r) --------- $T(n/2)$

MERGE(A, p, q, r) ---------- $C_1 n + C_2$

- Initial call: MERGE-SORT(A, 1, n)

Proof of Correctness – MERGE

- Loop Invariant for the merge operation

At the start of each iteration of the for loop for k, i and j:

- Subarray A[*p..k-1*] contains the k-p smallest elements of L and R in sorted order.
- L[*i*] and R[*j*] are the smallest elements of L and R that have not been copied back into A.

POC - Initialization

- Loop Invariant for the merge operation

At the start of each iteration of the for loop for k, i and j:

- Subarray A[*p..k-1*] contains the k-p smallest elements of L and R in sorted order.
- L[*i*] and R[*j*] are the smallest elements of L and R that have not been copied back into A.

Before the first iteration:

- A[*p..k-1*] = A[*p..p-1*] is empty/not defined, k-p=0
- i = j = 1. L[*1*] and R[*1*] are the smallest elements of L and R not copied to A.

Maintenance:

Case 1 : L[i] ≤ R[j]

- Let k = k1, i=i1 and j=j1 in the previous loop. Now k=k+1, i=i2, j=j2
- By L1, in the previous loop:
  - A[*p..k1-1*] contained *k1-p* smallest elements of L and R in sorted order.
  - L[*i1*] and R[*j1*] were the smallest elements of L and R not yet copied into A.
- In current loop:

1. A[*p..k-1*] contains the k1+1-p smallest elements of L and R in sorted order.
2. L[*i2*] and R[*j2*] are still the smallest elements of L and R not yet copied into A.

- Line 7 in loop results in A[k1] getting next smallest element. 1 proven.
- In Line 7 L[i1] was copied. In line 8 increments i to i2=i1+1. 2 proven.

Similarly for L[i] >R[j].

Termination:

- Line 5 ensures loop terminates
- On termination, $k = r + 1$
- By L1:
  - A[p..r] contains elements of L and R in sorted order.
- L and R together contain n elements which means the two piles have been copied back into A.

POC – MERGE-SORT

We know Merge works correctly, we will show that the entire algorithm works correctly, using a **proof by induction**.

Initialization:

For the base case, consider an array of 1 element (which is the base case of the algorithm). Such an array is already sorted, so the base case is correct.

Maintenance:

Suppose that MERGE-SORT will correctly sort any array of length less than ***n***, we call MERGE-SORT on an array of size ***n***. It will recursively call MERGE-SORT on two arrays of size *n/2*. By the induction hypothesis, these calls will sort these arrays correctly.

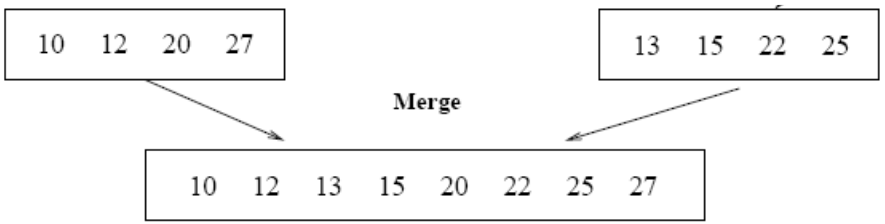
Termination:

Hence, after the recursive calls, array *a* will be sorted between indices

p, . . . m and m+1, . . . q respectively. We have already showed that MERGE works correctly, hence after executing it, *a* will be correctly sorted between *p* and *q*. This concludes our proof

Running Time of Merge (assume last **for** loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



Analyzing Divide-and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size n
  - **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
  - **Conquer** (solve) the subproblems $aT(n/b)$
  - **Combine** the solutions $C(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2$
    $\Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$

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

Sorting

- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time:

  Complexity:

  Best Case: O(n log n), When the array is already sorted or nearly sorted.

  Average Case: O(n log n), When the array is randomly ordered.

  Worst Case: O(n log n), When the array is sorted in reverse order.
- Space Complexity: O(n), Additional space is required for the temporary array used during merging.

Merge Sort - Discussion

- Running time insensitive of the input
- Advantages:
  - Guaranteed to run in $\Theta(n \lg n)$
- Disadvantage
  - Requires extra space $\approx N$

Sorting Challenge

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. Bubble sort
B. Selection sort
C. Mergesort guaranteed to run in time ~NlgN
D. Insertion sort

Sorting Huge, Randomly - Ordered Files

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Mergesort?
  - YES, it is designed for this problem

Sorting Challenge 3

Problem: sort a file that is already almost in order

Applications:

- Re-sort a huge database after a few changes
- Doublecheck that someone else sorted a file

Which sorting method to use?

A. Mergesort, guaranteed to run in time ~NlgN
B. Selection sort
C. Bubble sort
D. A custom algorithm for almost in-order files
E. Insertion sort

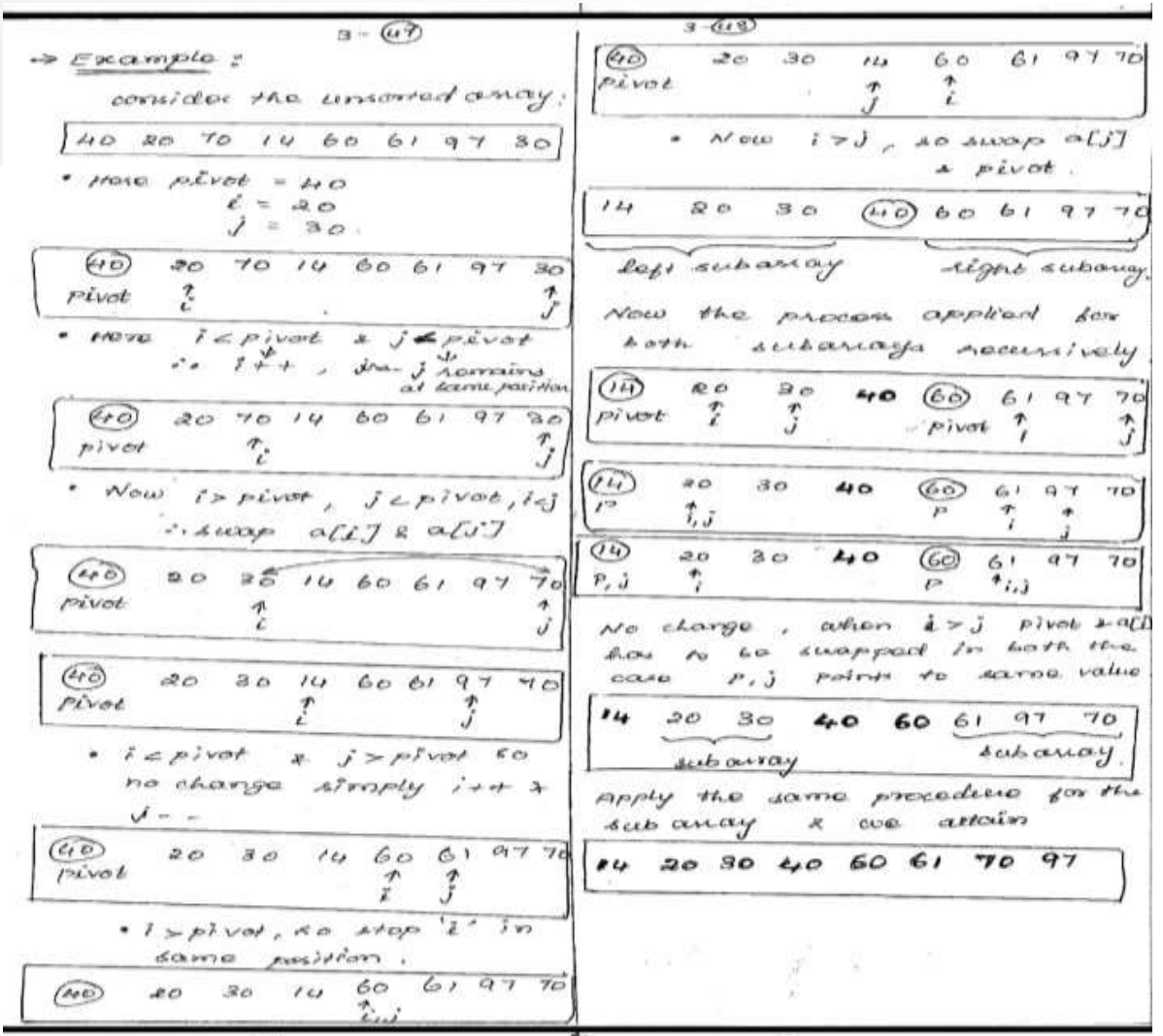
Sorting Files That are Almost in Order

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, bad for some definitions of “almost in order”
  - Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Insertion sort?
  - YES, takes linear time for most definitions of “almost in order”
- Mergesort or custom method?
  - Probably not: insertion sort simpler and faster

QUICK SORT

- It picks an element as a pivot and partitions the given array around the pivot.
- There are many different versions of quickSort that pick the pivot in different ways.
  - ✓ Always pick the first element as a pivot.
  - ✓ Always pick the last element as a pivot.
  - ✓ Pick a random element as a pivot.
  - ✓ Pick the median as the pivot.

ILLUSTRATION



→ Example :

consider the unsorted array :

40	20	70	14	60	61	97	30

- Here pivot = 40
  i = 20
  j = 30

40	20	70	14	60	61	97	30
pivot	i						j

- Here i < pivot & j < pivot
  ∴ i++, but j remains at same position

40	20	70	14	60	61	97	30
pivot		i					j

- Now i > pivot, j < pivot, i < j
  ∴ swap a[i] & a[j]

40	20	30	14	60	61	97	70
pivot		i					j

40	20	30	14	60	61	97	70
pivot			i			j	

- i < pivot & j > pivot so no change simply i++ & j--

40	20	30	14	60	61	97	70
pivot				i	j		

- i > pivot, so stop 'i' in same position.

40	20	30	14	60	61	97	70
				i,j			

40	20	30	14	60	61	97	70
pivot			j	i			

- Now i > j, so swap a[j] & pivot.

14	20	30	40	60	61	97	70

left subarray — right subarray

Now the process applied for both subarrays recursively

14	20	30	40	60	61	97	70
pivot	i	j		pivot			j
					i		

14	20	30	40	60	61	97	70
P	i,j			P	i	j	

14	20	30	40	60	61	97	70
P, j	i			P	i,j		

No change, when i > j pivot & a[j] has to be swapped in both the case P, j point to same value.

14	20	30	40	60	61	97	70
	subarray				subarray		

Apply the same procedure for the sub array & we attain

14	20	30	40	60	61	70	97

ROUTINE ANALYSIS

```
Quicksort(a,l,r)
{
if(l<r)
{
pivot=a[l];
i=l+1;
j=r;
for(;;)
{
while (a[i]<pivot)          i++;
while (a[j]>pivot)          j--;
if(i>j)
swap(a[i],a[j])
else break;
}
swap(a[j],pivot);
Quicksort(a,l,j-1)
Quicksort(a,j+1,r);
} }
```

Time complexity

T(n)=2T(n/2)+n

$= 2[2T(n/2)+n/2]+n$

$= 2^k T\left(\frac{n}{2^k}\right)$

$= 2^k \cdot \frac{n}{2^k}$

take log on both sides

T(n)=O(n log n)

MAXIMUM SUBARRAY

- Offered the opportunity to invest in the Volatile Chemical Corporation.
- Stock price of the Corporation is rather volatile.
- You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day
- To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future
- You can always maximize profit by either buying at the lowest price or selling at the highest price - doesn't always work
- So we shall look at the change in prices of each da

Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

MAXIMUM SUBARRAY

- Can be addressed by
  - Naïve approach
  - Brute force
  - Divide and Conquer

NAÏVE APPROACH

Step 1: Read number of elements, n

Step 2: Read 'n' elements in the array

Step 3: Let max = minimum negative value

Step 4: for sub_array_size 1 to n repeat step 5 to 7

Step 5: Generate sub_array of size sub_array_size repeat step 6 and 7

Step 6: Compute sum_of_sub_array generated

Step 7: If sum_of_sub_array > max then make max as sum_of_sub_array

Complexity - O(n^3)

FINE TUNED NAÏVE APPROACH

Idea : We are not interested in the order of the elements but only sum

Traverse array once when you are in ith element find all possible sum of

Sub-arrays that start at index 'i' and formed by including elements at index 'j' >= ‘i’

Step 1: Read number of elements, n

Step 2: Read 'n' elements in the array

Step 3: Let max = minimum negative value

Step 4: for i in range 1 to n repeat step 5 to 8

Step 5: Let subarray_sum = 0

Step 6: for j in range i to n repeat step 7 and 8

Step 7: subarray_sum += element[j]

Step 8: If subarray_sum > max then make max as sum_of_sub_array

Complexity - O(n^2)

BRUTE FORCE

Try every possible pair of buy and sell dates in which the buy date precedes the sell date

Complexity - O(n^2)

DIVIDE AND CONQUER

To find a maximum subarray of the subarray A[low..high]

Fact : Any contiguous subarray A[i..j] of A[low..high] must lie in exactly one of the following places:

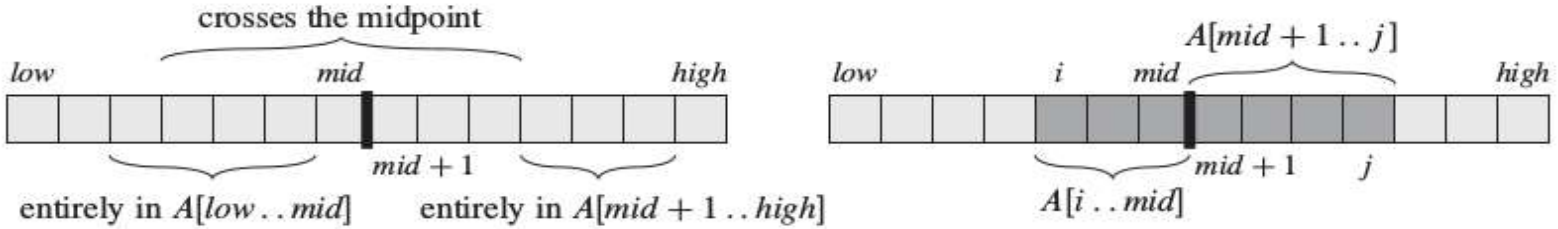
- ✓ entirely in subarray A[low..mid], so that low $\leq i \leq j \leq$ mid
- ✓ entirely in subarray A[mid+1..high] so that mid$<i \leq j \leq$high
- ✓ crossing the midpoint, so that low $\leq i \leq$ mid $\leq j \leq$ high

DIVIDE AND CONQUER

- ✓ We can find maximum subarrays of A[low..mid] and A[mid+1 .. high] recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray.
- ✓ Not a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint

DIVIDE AND CONQUER

- ✓ Figure below shows, any subarray crossing the midpoint is itself made of two subarrays A[i .. mid] and A[mid + 1 .. j], where low ≤ i ≤ mid and mid < j ≤ high
- ✓ Therefore, we just need to find maximum subarrays of the form A[i..mid] and A[mid+1..j] and then combine them.



```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

MAXIMUM SUB-ARRAY

Time Complexity **O(nlogn)**

Applications :

- ✓ Stock Prices: Finding the best time to buy and sell stock to maximize profit.
- ✓ Temperature Variations: Identifying the longest period of consistently rising temperatures.
- ✓ Sales Data: Determining the most profitable period within a financial quarter.

Example : Array: [2, -1, 3, -4, 5, -2, 2, 1, -3]

KARATSUBA'S FAST INTEGER MULTIPLICATION ALGORITHM

Conventional Method of Multiplication:

- *The Naive approach for multiplication is the process that we study in school.*
- *Conventional method of n digit multiplication total n*n=n^2 multiplication is required.*

  let's take an example of 2-digit multiplication...

  6 5 * 9 7 == 6 3 0 5

- Here first 7 is multiplied by 5 and then 6. It makes two multiplication.
- Again 9 is multiplied by 5 and then 6. It also make two multiplication.

  So, total multiplication required is 4.

  As we can say $n*n=n^2$ multiplication is required.

KARATSUBA'S FAST INTEGER MULTIPLICATION ALGORITHM

Divide and Conquer Method:

- This algorithm reduces the number of multiplication comparing to conventional method of multiplication.
- Using **Divide and Conquer**, we can multiply two integers in less time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y.

Let, X = 1 2 3 4

Y = 9 8 7 6 Here n = 4

Here we divide the number into two halves

a = 1 2 b = 3 4

c = 9 8 d = 7 6

KARATSUBA'S FAST INTEGER MULTIPLICATION ALGORITHM

Here we can write X and Y as..

$X = a*10^{n/2} + b \quad Y = c*10^{n/2} + d$

Let's multiply X and Y

$X * Y = (a*10^{n/2} + b)\,(c*10^{n/2} + d)$

$= (10^{n})ac + (10^{n/2})(ad + bc) + bd$

Here in each step of recursive call we are performing 4 extra multiplication and two extra atomic addition.

In each step two chopped number is passed to recursive function whose length reduces to ½ times each steps.

Overall function becomes

$M(n) = 4M(n/2) + O(n)$ (two atomic addition performs in each steps).

KARATSUBA'S FAST INTEGER MULTIPLICATION ALGORITHM

The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient.

Here we can write

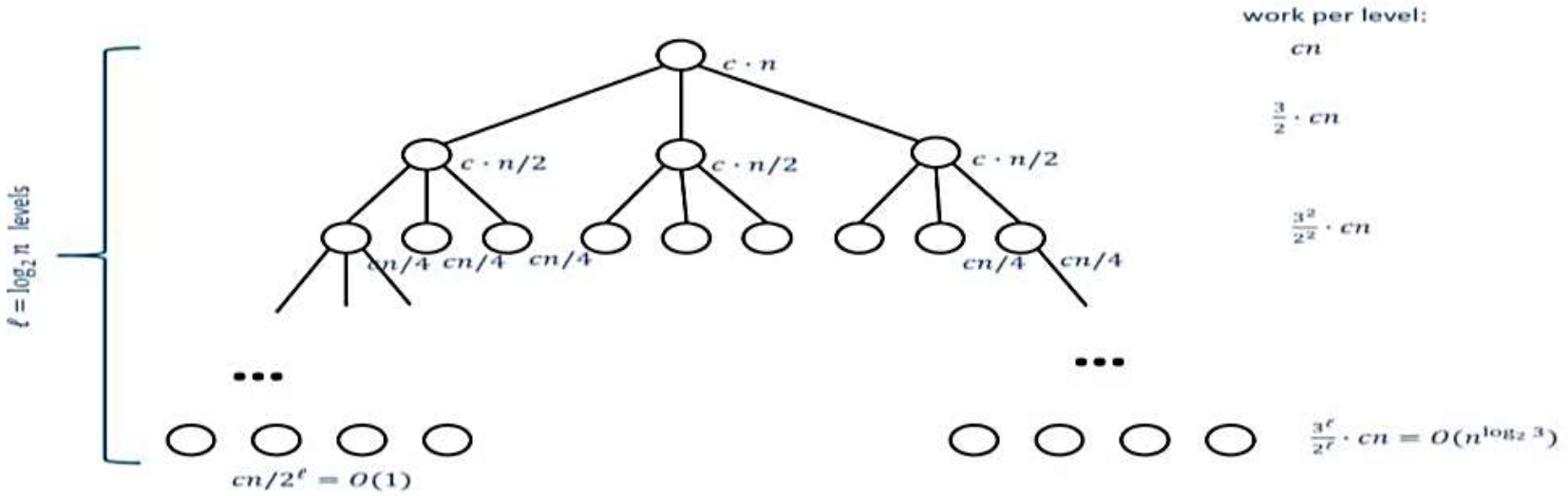
$ad + bc = (a + b)(c + d) - ac - bd$

We are reducing one extra multiplication and now our function becomes....

$M(n) = 3M(n/2) + O(n)$

By doing this we are performing 3 extra multiplications of size n/2 in each step.

KARATSUBA'S FAST INTEGER MULTIPLICATION ALGORITHM



KARATSUBA'S FAST INTEGER MULTIPLICATION ALGORITHM

Complexity of Karatsuba:

Notice that each time digit is reduced by ½ times the previous ones , thus we can write...

$$n = 2^k$$

$$\log_2 n = k \log_2 2$$

$$k = \log_2 n$$

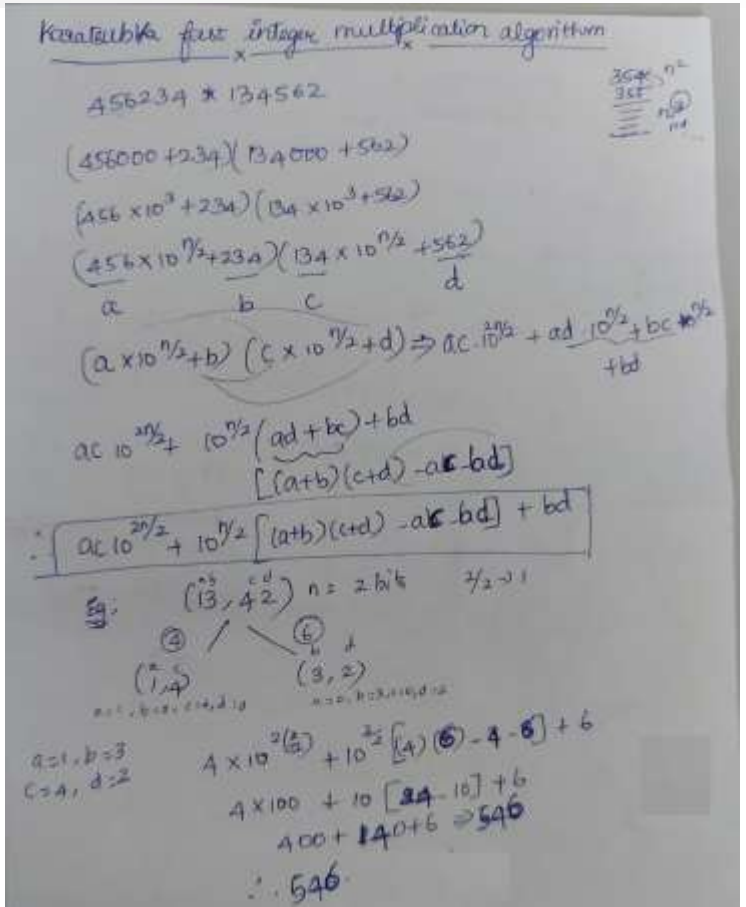
For conventional multiplication $4^k = 4^{\log_2 n} = 2^{(\log 4)(\log n)} = n^{(\log 4)} = n^2$

For karatsuba multiplication $3^k = 3^{\log_2 n} = 2^{(\log 3)(\log n)} = n^{(\log 3)} = n^{1.584...}$

As we can see that complexity of multiplication reduced to $n^{1.584...}$

KARATSUBA'S FAST INTEGER MULTIPLICATION ALGORITHM

Derivation with example:



CASE STUDY

SAMPLE EXERCISE

Consider arranging n numbers stored in an 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. Based on the approach described above, write an algorithm for arranging the given n numbers in an increasing order of the numbers. Compute the best-case running time, worst-case running time and the average case running time of the algorithm. Compare this algorithm with that of the insertion sort algorithm, based on the respective $T(n)$. Based on your analysis, conclude which algorithm performs better for which type of inputs etc.

SOLUTION

Selection Sort Algorithm

1. **Initialize**: Start with the first element.
2. **Find Minimum**: For each position i (from 0 to $n-1$):
   - Find the smallest element in the subarray starting from i to $n-1$.
   - Swap the found minimum element with the element at position i.

```
def selection_sort(A):
    n = len(A)
    for i in range(n-1):
        # Find the minimum element in the remaining unsorted array
        min_index = i
        for j in range(i+1, n):
            if A[j] < A[min_index]:
                min_index = j
        # Swap the found minimum element with the element at position i
        A[i], A[min_index] = A[min_index], A[i]
    return A
```

TIME COMPLEXITY

$O(n^2)$

SOLUTION

Insertion Sort Algorithm:

1. **Initialize**: Start with the second element.
2. **Sort Subarray**: For each element i (from 1 to $n-1$):
   - Insert $A[i]$ into its correct position in the sorted subarray $A[0..i-1]$.

TIME COMPLEXITY

Best Case - O(n) –sorted array

Worst Case - O(n^2)

```
def insertion_sort(A):
    n = len(A)
    for i in range(1, n):
        key = A[i]
        j = i - 1
        # Move elements of A[0..i-1] that are greater than key one position ahead
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key
    return A
```

SOLUTION

Comparison and Conclusion

Selection Sort:

Consistent **O(n^2)** time complexity in all cases.

Makes a total of n(n−1)/2 comparisons and n−1 swaps.

Insertion Sort:

Best-case time complexity is O(n)-when the array is already sorted.

Worst-case and average-case time complexity is **O(n^2)**

CASE STUDY

The first, known as *insertion sort*, takes time roughly equal to c1. n^2 to sort n items, where c1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2.

The second, *merge sort*, takes time roughly equal to c2.n logn, where log n stands for log2 n and c2 is another constant that also does not depend on n.

Insertion sort typically has a smaller constant factor than merge sort, so that c1 < c2.
We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n.

For example, when n=1000, log n is approximately 10, and when n equals one million, log n is approximately only 20.

CASE STUDY

Computer A (Fast) – Insertion Sort

Computer B (Slow)– Merge Sort

Both computer must sort an array of 10 million numbers - if the numbers are eight-byte integers , then the input occupies about 80 megabytes.

A executes 10 billion instructions per second.

B executes only 10 million instructions per second.

Efficient programmer in A, for insertion sort it requires $2n^2$ **to sort n numbers**

Average programmer in B, runs merge sort, it takes 50n log n instructions

To sort 10 million numbers, computer A takes $\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)}$

But in Computer B,

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)}$$

100 million data:
insertion sort – 23 days,
merge sort – 4 hrs

Algorithm running time grows slower

Thank you