ASSIGNMENT 1

Course Name: Algorithm

Course Code: CSE 231

Submitted To

Name: Kashfi Shormita

Department: CSE

Daffodil International University

Submitted by

Name: Riyad Ali Mollik

Id: 221- 15-5096

Section: U

Department: CSE

Daffodil International University

Submission Date: 5/10/2023

1. How bubble sort is different from Selection sort? Describible.

Ans: Bubble sort and selection sort are both simple sorting algorithms, but they are different in their approach to sorting elements in the array.

1. Comparison Based:-

Bubble sort repeatedly compares and swaps adjacent elements, whereas selection sort selects the minimum element and moves it to the sorted part.

2. Iterative:-

Bubble sort can have many unnecessary swaps, while selection sort performs a fixed number of swaps per pass.

3. Stability:-

Bubble sort is stable, while selection sort is not stable by default.

4. <u>Adaptive:-</u>

Bubble sort can be slightly more adaptive if the input data is partially sorted. selection sort is always performs the same number of comparison.

<u>Exmple:</u> consider a unsorted array

[5, 2, 9, 3, 4]

<u>Bubble sort:-</u> In the first pass, the algorith compare adjecent elements and swap if necessary. After the pass, the largest element 9 bubbles up to the end of the array.

[2, 5, 3, 4, 9]

and the second pass, next largest element 5 moving to its correct position

[2, 3, 4, 5, 9]

After several passes, the array becomes sorted.

Selection Sort:- Initially, the algorithm considers the entire array as unsorted. It finds the minimum elements (2) and swaps it with the first element.

[2, 5, 9, 3, 4]

Next it considers the remaining unsorted part and finds the minimum elements (3) and swaps it with the second element.

[2, 3, 9, 5, 4]

The procces is continues and the next minimum elements are selected and place in their correct positions.

[2, 3, 4, 5, 9]

After iterating through the entire array, the array becomes sorted.

2. Describe selection sort with an exmple and its pseudocode.

Ans:- Selection sort algorithm is a simple comparison - based sorting algorithm that works by repeatedly selecting the minimum (or maximum) element from an unsorted portion of the list and moving it to the begining (or end) of the sorted portion. It has a time complexity of $O(n^2)$ and relatively easy to understand and implement.

Selection sort working process step by step:-

1. Find the minimum element in the unsorted portion of the list.

2. Swap this minimum element with the first element in the unsorted portion, and added it to the sorted portion.

3. Repeat the above two steps for the remaining unsorted portion, excluding the elements already sorted.

4. continue this process untill the entire list is sorted.

<u>pseudocode:-</u>

```
n = length (arr)
for i from 0 to n-1;
  minIndex = i
  for j from i+1 to n-1
  if arr[j] < arr [minIndex]
       minIndex = j
  swap (arr[i] , arr [min Index])
end procedure.
```

<u>Exmple:-</u>

sorted | unsorted

0	1	2	3	4
9	13	15	11	2

$\therefore n-1 = 5-1 = 4$ iteration

<u>iter 1:-</u>

sorted | unsorted

0	1	2	3	4
2	13	15	11	9

<u>iter 2:-</u>

sorted | unsorted

0	1	2	3	4
2	9	15	11	13

<u>iter 3:-</u>

sorted | unsorted

0	1	2	3	4
2	9	11	15	13

<u>iter 4:-</u>

0	1	2	3	4
2	9	11	13	15

→ sorted

3. <u>Problem</u>:- You have a sorted array of integers and you want to find if a specific Integer exists in the array and determine its index if it does.

<u>Algorithm</u>:-

1. Initialize two pointers, left and right to the first and last index of the array.

2. While left is less than or equal to right:

a. Calculate the middle index as mid.

b. If mid = = target then return mid as index.

c. If mid < target, update left to mid + 1 to search the right half.

d. If mid > target, update right to mid - 1 to search the left half.

3. If the while loop exits without finding the target element return -1 that the elements is not in the

Code:-

```
#include <stdio.h>
    int binarySearch ( int a[ ], int left,
                                  int right , int target){
  While ( left <= Right){
     int mid = (left + right) / 2 ;
   if ( int arr[mid] == target){
          return mid ;
     }else if ( arr[mid] < target){
           left = mid +1 ;
       } else {
            right = mid -1 ;
            }
       }
  return -1;
}
int main () {
      Int a[ ] = {1, 3, 5, 7, 9, 11}
      int target = 7;
      int n = sizeof (arr) / sizeof (arr[0]);
 int res = binarySearch(a, 0, n-1, target);
if (result != -1){
       printf(" Target %d found at index %d\n
                          target , res ) ;
```

```
else {
    printf(" Target %d not found in
    the array \n", target);
  }
return 0;
}
```

<u>Result:-</u> Target 7 found at index 3.

(b) <u>Best case:-</u> In the best case senario, binary search performs easily. Suppose, target elements is found at the middle of the array. This means that binary search immediately identifies the target without the need for further iteration.

<u>Mathematical Explanation:-</u> Binary search performs a single comparison to determine that the target is found.

The time complexity is $O(1)$.

Exmple:- Consider an array with 7 elemen [1, 3, 5, 7, 9, 11, 13] and we want to find the target 7. In this case, binary search directly identifies the target in the first comparison.

Worst-case:- Worst-case senerio is :, the target element is not present in the array and binary search has to check every element in the array before determing that the target is not there

Mathematical Explanation:- Binary search repeatedly divides the search interval by half until the remaining interval has only one element and it has to do this for each level of division.

Time complexity O(log n).

Exmple:- In 7 elements array [1, 3, 5, 7, 9, 11, 1 we find 8. In this case, binary search will have to perform the maximum number of comparison, going through all levels of division until it determines that the target is not in the array.

Average case:- We assume that, the target element is equally likely to be in ~~any~~ any position within the array.

Mathematical explanation:- Binary Search perform $\log_2 n$ comparison on average because, in a well distributed dataset, it has an equal chance of finding the target in any part of the array. Thus, the avg case time complexity is $O(\log n)$.

Exmple:- In 7 elements array [1, 3, 5, 7, 9, 11, 13] and we want to find randomly chosen target from the array. Binary search will require $\log_2(7) \approx 2.81$ comparisions,

9. a. The coin change problem is a classic problem in computer science and mathematics. Given a set of coin denominations and a target amount of money, the goal is to find the minimum number of coins required to make up that amount.

Working process (Algorithm):

1. Sort the coin denominations in descending order.
2. Initialize a variable to keep track of the number of coin used:
3. Start with the largest denomination coin.
4. While the target amount is greater than zero:- • If the current denominations is less than or equal to the remaining target amount, use as many of that denomination as possible.

• Update the target amount by subtracting the value of the used coins.

• Increment the count of coins used.

• Move to the next smaller denomination

5. Repeat this process untill the target

The greedy Algorithm works optimally for the coin change problem when the coin denominations satisfy two property:-

1. <u>Greedy choice property</u>: At each step, choosing the coin with the largest denomination that doesn't exceed the remaining target amount is a locally optimal choice.

2. <u>optimal substructure</u>:- The problem exhibits ~~prob~~ optimal substructure when the optimal solution for the entire amount can be constructed from optimal solutions to smaller subproblems.

b) Denominations {1, 5, 10, 25}

Descending order = {25, 10, 5, 1}

(i) Start largest denomination 25.

(ii) 25 < 63 . so ~~25~~ is use as many 25 - cent coins as possible.

(iii) In this case I use 2 times

$2 * 25 = 50$ cents, and remaining amount is $(63 - 50) = 13$ cents.

(iv) More to next smaller denomination. Now use 1 10-cent coin.

remaining amount $(13-10) = 3$ cents.

count of coins 1

(v) Now I have 3 cent to change. Next coin is 5 cent. which is $3 < 5$. so, we go the next coin 1 cent.

It used and count 3 times.

remaining amount $(3-3) = 0$.

$\therefore$ coin need $25\overset{\text{coin}}{(2)} + 10\overset{\text{coin}}{(1)} + 1\overset{\text{coin}}{(3)}$

$= 63\ (6) \rightarrow$ coin

5. Item: 1 2 3 4
Weight: 2 3 4 5
Value: 4 5 7 10

n = 4
max weight =

w →

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4
2	0	0	4	5	5	9	9
3	0	0	4	5	7	9	11
4	0	0	4	5	7	10	$\boxed{11}$

item : 1 2 3 4
1 0 1 0

The maximum value that can be achieved by including these items [{item 3 (weight 4, value 7)}, and {item 1 (weight 2, value 4)}] in the knapsack is 7+4 = 11 which matches the maximum value calculated earlier.

So, the optimal solution is to include item 1 and Item 3 in your knapsack to maximize the total value while not exceeding the weight capacity of ~~6~~

item :	1	2	3	4	Max weight capacity = 6
weight :	2	3	4	5	
value :	4	5	7	10	
Density $\frac{v}{w}$:	2	1.67	1.75	2	

<u>Sorted Table:</u> (According density)

Item	weight	value	Density
1	2	4	2
4	5	10	2
3	4	7	1.75
2	3	5	1.67

<u>Fractional knapsack:</u>

Item	weight	value	Density	Total value	Total weigh
4	5	10	2	10	5
1	1	2	2	12	6