No.

Date 03-03-21

Homework 1

Ryan Monaghan

1) Linear Search on sorted list (in descending order)

n: length of list A

v: the value we are searching A[0 ... n-1] for and

and A[0] >= A[1] >= ... >= A[n-1]

list index starts from 1

LinearSearch (A, n, v)

```
int LinearSearch (int A[], int n, int v ){

    for(int i=0; i<n; i++){   //for every element i
                              //in the list of n elements
        if (A[i] == v ){      //Check if A[i] == V
            return i;         //if the elem. V is found
                              // return V
        }                     //return;
        else (A[i] >V {       //else if A[i] > V  return -1
          return -1;
      } //end if-else
    } //end for loop
int main () {
      int A[] ={3,4,6,9}
      int n = A.size() -1 / A[0].size() -1;
      //or try
       int n = sizeof(A) / sizeof (A[0]);

       int index = LinearSearch(A, n, v);
         if (index == -1){
      cout << " Value is not present in array A";
       }
       else {
          cout<< "                    "Value found at position
          }                           << index;
} return 0;
```

BubbleSort[A]

```
2)  for J ← 1 to length[A]
    {     for i ← length[A] down to i+1;
          if A[i] < A[i-1]
    }     Swap(A[i], A[i-1])
```

example:

A =

1	2	3	4	5	6
5	1	2	3	6	9

```
for J ← 1 to length[A]
    J ← 1    to 6              (length[A] = 6)

for J = 1,
              for i ← length[A] down to J+1
                  i ← 6 down to (1+1)     w/ J = 1
                  i ← 6 down to 2
for i = 6,    if A[i] < A[i-1]
                 A[6] < A[6-1]            Value of A[6] is 9
                 A[6] < A[5]              & A[5] is 6
                   9 < 6          false   from the array
```

3) a) analyze the running time of the Psuedocode

```
Smallest (a[], left, right)
  if left == right  return a[left] → O(1) takes const.
                                     time only comparison
  mid = (left + right) / 2 → O(1)
  l1 = Smallest (a[left ... mid]) → T(n/2)
  l2 = Smallest (a[mid+1 ... right]) → T(n/2)

  if (l1 > l2)
     return l2  → O(1)
else
     return l1  → O(1)
```

Let $T(n)$ be the total time / steps taken by the function then,

$T(n) = T(\frac{n}{2})$ → (for Smallest (a, left, mid)

$+ T(\frac{n}{2})$ → (Smallest (a[], mid+1, right)

$+ O(1)$ → all other operations take constant time.

$T(1) = 1$ is for the base case where there is only one element

//Aprach :

$$T(n) = \begin{cases} 1, & n=1 \\ 2T(\frac{n}{2})+1, & n>1 \end{cases}$$

$$T(n) = T(\tfrac{n}{2}) + T(\tfrac{n}{2}) + 1$$
$$T(n) = 2T(\tfrac{n}{2}) + 1$$
$$T(\tfrac{n}{2}) = 2T(\tfrac{n}{4}) + 1$$
$$T(\tfrac{n}{2}) = 2T(\tfrac{n}{2^2}) + 1$$
$$T(n) = 2[2T(\tfrac{n}{2^2}) + 1] + 1$$
$$= 2^2 T(\tfrac{n}{2^2}) + 2 + 1$$
$$T(\tfrac{n}{2^2}) = 2T(\tfrac{n}{2^3}) + 1$$
$$T(n) = 2^2 [2T(\tfrac{n}{2^3}) + 1] + 3 \quad (3 = 2^2 - 1)$$
$$T(n) = 2^3 T(\tfrac{n}{2^3}) + 2^2 + 3$$
$$T(n) = 2^3 T(\tfrac{n}{2^3}) + 7 \quad (7 = 2^3 - 1)$$

$\vdots$

$$T(n) = 2^k T(\tfrac{n}{2^k}) + (2^k - 1)$$
$$\frac{n}{2^k} = 1 \rightarrow n = 2^k \rightarrow \log_2 n \rightarrow \log_2 2^k$$
$$\log_2 n = k \log_2 2 = k = \log_2 n$$

So $T(n) = n \cdot T(1) + n - 1$

$$= n \cdot 1 + n - 1 = 2n - 1$$

$$T(n) = O(n)$$

b) LargestOneDayDifference (a[0 ... n-1])

```
largestChange = 0 // initialization
 for i=0 to n-2 // runs n-2 times
     if (abs (a[i+1] - a[i] > largestChange) //runs about
       largestChange = abs (a[i+1] - a[i]) //n-2 times
                                //other statements inside the
 return largestChange           //function take O(1) or
                                //constant amount of time
```

So, $T(n) = n-2 + \overset{//const}{c}$ for all $n > 1$

$= 0$ if $n = 1$

Therefore, $\boxed{T(n) = O(n)}$

c) some_func(data[1...n])

```
    for i=1 to n-4 //n-4 times
    sum[i] = 0
    for J=i to i+3 //i+3 times
    sum[i] = sum[i] + data[J]

  largest = sum[i]
     for i=1 to n-4 // n-4 times
        if (sum[i] > largest) if i=1, J runs from 1 to 5
           largest = sum[i]   if i=2, J runs from 2 to 6
                              if i=3, J runs from 3 to 7
```

Running time is sum of cost of nested looping block, simple looping block, and constants

$T(n) = 5(n-4) + (n-4) + c$

$= 5n - 20 + n - 4 + c$

$= \boxed{6n - 24 + c}$

So, $T(n) = O(n)$

Homework 1 contd.

4) Solve the following recursive relations using method of iteration, given that $T(1) = 1$

a) $T(n) = T(\frac{n}{3}) + 1$

$$T(n) = \begin{cases} 1, & n = 1 \\ T(\frac{n}{3}) + 1, & n > 1 \end{cases}$$

$T(n) = T(\frac{n}{3}) + 1$

$T(n) = T(\frac{n}{2}) + 1$

$T(\frac{n}{2}) = T(\frac{n}{2^2}) + 1$

$T(n) = \left[T(\frac{n}{2^2}) + 1\right] + 1$

$T(n) = \left[T(\frac{n}{2^3})\right] + 2$

$\vdots$

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

if $\frac{n}{2^k} = 1$ and $n = 2^k$ and $k = \log_2 n$

Then $T(n) = T(1) + \log n$

$T(n) = 1 + \log n$

$= O(\log n)$

b) $T(n) = T(n-1) + 3n$

$$\begin{cases} T(n) = 1, & n = 1 \\ T(n-1) + 3n, & n > 1 \end{cases}$$

$T(n) = T(n-1) + 3n$

$T(n-1) = T(n-1-1) + 3(n-1)$

$T(n-1) = T(n-2) + 3n - 3$

$T(n) = (T(n-2) + 3n - 3) + 3n$

$T(n) = T(n-2) + 6n - 3$

$T(n) = (T(n-3) + 3n - 6) + 6n - 3$

$T(n) = T(n-3) + 9n - 9$

$T(n) = T(n-3) + 3^2(n-1)$

$T(n) = T(n-k) + (n-(k-1)) + (n + (k-2) \ldots (n-1) + n$

if $n - k = 1$ and $n - 1 = k$

$T(n) = T(n - n + 1) + (n - n + 2) + (n - n + 3) + \ldots (n - n) + n$

$T(n) = T(1) + 1 + 2 + 3 + 4 + \ldots + n$

$T(n) = 1 + \frac{n(n+1)}{2}$

So, $O(n^2)$

c) $T(n) = 2T(n-1)$

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n-1), & n > 1 \end{cases}$$

$T(n) = 2T(n-1)$

$T(n-1) = 2T(n-1-1)$

$T(n-1) = 2T(n-2)$

$T(n) = 2(2T(n-2))$

$T(n) = 2(2(2T(n-3)))$

$\vdots$

for k^{th}

$T(n) = 2^k T(n-k)$

if $n-k = 1$ and $n-1 = k$

$T(n) = 2^{n-1}\ T(n-n+1)$

$T(n) = 2^{n-1}\ T(1)$

$T(1) = 1$

$T(n) = 2^{n-1}$

So $O(2^n)$

5)

(a) Write a recursive algorithm to calculate the exponent function

$$\begin{aligned} a^n &= 1, \text{ if } n=0 \\ &= a, \text{ if } n=1 \\ &= a^k, \text{ if } n \text{ is even}, n=2k \\ &= a^k \times a, \text{ if } n \text{ is odd}, n=2k+1 \end{aligned}$$

```
//Return a^n
//n is a natural number, n=0,1,2,...

Exp(a,n)
```

↓

```
// Return a^n
// n is a natural number, n = 0,1,2,...

int Exp(int a, int n){
   //Base Case
   if(n==0){
      return 1;

// general case if n>=1
// a^n == a(a^(n-1))
      int k;
      if(n%2 ==0){
      //a^n = a^k if n is even, where n=2k
      k = n/2;
      return a x Exp(a,(2xK)-1);
   } else{   //a^n = a^k x a if n is odd, where n=2k+1
          K = (n-1)/2;
          return a x Exp(a,(2xk));
      }
}
```

Simplified Version

```
int Exp (int a, int n){
    //base case
  if (n==0)
      return 1;

   // general case if n ≥ 1
   // a^n == a × a^(n-1)
    return a × Exp(a, n-1);
```

b) Trace the execution of Exp (2,7)
draw the recursion tree

Exp (2, 7)

2 × Exp (2, 7-1) = 2 × Exp (2,6) = 2 × 64 = 128

2 × Exp (2, 6-1) = 2 × Exp (2,5) = 2 × 32 = ~~64~~ 32

2 × Exp (2, 5-1) = 2 × Exp (2,4) = 2 × 16 = 16

2 × Exp (2, 4-1) = 2 × Exp (2,3) = 2 × 8 = 8

2 × Exp (2, 3-1) = 2 × Exp (2,2) = 2 × 4 = 4

2 × Exp (2, 2-1) = 2 × Exp (2,1) = 2 × 2 = 2

2 × Exp (2, 1-1) = 2 × Exp (2,0) = 2 × 1 = 2

Exp (2,7) = 128

c) Fill in the blank in the following binary search algorithm

No.

Date

Let red color be answer to the To-Do

```
if (left+1 == right) {
    if (A[left] == V)
       //add code ToDo
       return left

    else if (A[right] == V)
       return right

   else if (v > A[left])
      // v is larger than A[left], so V should appear before it
       return -1 * left

   else if (v < A[right])
      //v is smaller than A[right], so v should appear
      //before it
       return -1 * (right + 1)
  else
          // v > A[right], but v < A[left] so it would take
          //current A[right] index
          return -1 * right
}
  //general case, length of list >= 3
    mid = (left + right) / 2
    if (A[mid] == Value)
               return mid
    else if (A[mid] > V)
          //V < A[mid], So search right half since descending order
          return BinarySearch(A, V, mid+1, right)
    else
          //V > A[mid], So search left half since in descending order
           return BinarySearch(A, V, left, mid-1)
    }
```

(d) Use the three question rule to verify

- Base case: We have base cases for when the input size is 1 or 2. For n = 1, if the value exists in the list, it will have to be A[left]. If the value is not in the list, it must be greater or smaller than the existing element, to which the proper would-be index will be returned. For n = 2, if the value exists in the list, it will have to be A[left] or A[right]. If the value is not in the list, it must be greater than A[left], smaller than A[right], or it is less than A[left] and greater than A[right]. If this is the case, the proper would-be index will be returned. The base cases will ensure that we are able to exit the algorithm non-recursively

- Smaller Caller: Each recursive call to the algorithm involves a smaller case of the original problem, leading inescapably to the base cases. The call to BinarySearch decreases the range by half which will lead to the base cases as it gets smaller. The algorithm gets the value of the middle index and compares it to the value being searched for. If the value is greater than the middle value, BinarySearch is called with a smaller range to search only the left half of the original list as the list is sorted in descending order. If the value is less than the middle value, BinarySearch is called with a smaller range to search only the right half of the original list as the list is sorted in descending order.

→

- General Case:

There is a general case for when the input size is greater than or equal to 3. The algorithm first finds a middle index by adding the left and right indicies and dividing them by 2. If the middle value is equivalent to the value we are searching for, the algorithm will return the middle index. The algorithm will be called recursively if the value being searched for is less than or greater than the middle value. If so, the algorithm calls itself recursively with a range that is decreased by half, searching a specific half based on whether the value is less than or greater than the middle value.