**NAME: Kulsoom Khurshid**

**Reg # SP20-BCS-044**

**Question 1)**

//input: int x, int n, array of integers.

//output: anxn + an-1xn-1 + …… + a1x + a0

BruteForcePolynomialEvalution(P[0 …… n], x)

p <- p[0];

power <- 1;

for i <- to n do

power <- power \* x

p <- p + p [i] \* power

return p;

Complexity:

n(2) = 2n = O(n)

**Question 2)**

//input: Array[0 …… n-1] (having 0’s and 1’s).

//output: Array[0 …… n-1] (sorted as 000… 1111…)

SortBalls(A[0 …… n-1])

for i <- 2n-1 to 1 do{

flag <- false

for j <- 0 to i – 1{

if (A[i] > A[j+1]){

swap A[j + 1] and A[j]

flag <- true

}}

if (! flag)

return

}

Complexity:

The first iteration runs the condition for n times, for second iteration the condition runs for n times so T(n) = n \* n = n2

O(n2)

**Question 3)**

//input: string s.

//output: int num

Algo1(s)

n <- s.length

num <- 0;

for i <- 1 to n do

for j <- i + 1 to n do

if a[i] = ‘A’ and a[j] = ‘B’

num++

return num

Complexity:

O(n2)

Algo2(s)

n <- s.length

if A[n] = ‘B’

B[n] <- 1

else

B[n] <- 0

For i <- n – 1 to 1 do

If A[i] = ‘B’

B[i] <- B[i +1] + 1

else

B[i] <- B[i+1]

num <- 0

for i <- 1 to n – 1 do

if A[i] = ‘A’ then

num <- num + B[i+1]

return num

Complexity:

O(n)

**Question 4)**

//input: Boolean adjacency matrix A[0 … n-1, 0 … n-1], where n > 3

//output: ring = 1, star = 2, mesh = 3

M0 <- 0

for i <- 1 to n – 1 do

M0 <- M0 + A[0,i]

If M0 = 2

return 1

else if M0 = 1

return 2

M1 <- 0

for j <- 0 to n – 1 do

M1 <- M1 + A[1,j]

If M0 = M1

return 3

else

return 2

Complexity:

n2= O(n2)

**Question 5)**

//input: array of coordinates of points, n (criteria of closeness)

//output: array without any points

b <- array of size a.length

for i <- 0 to a.length – 1

//distance of a[i] from origin and insert in empty array b

//sort the new array b using merge sort

for i <- 0 to b.length – 1

if distance(b[i],b[i+1]) < n

remove b[i] from array b

Complexity:

n + n log n + n

O (n log n)

**Question 6)**

//input: array a, int num

//output: (i,j) or none

n <- a.length

//sort the array

for i <- 0 to n – 1

j <- find (num – a[i])

if j > 0 and < n

return (i,j)

return none

Complexity:

n log n + n log n = 2 logn

O (n log n)

**Question 7)**

//input: array a, array b, int num

//output: (i,j) or none

n <- a.length

//sort array b

for i <- 0 to n – 1

j <- find (num – a[i])

if j > b.length and 0 < j

return (i,j)

return none

Complexity:

n log n + n log n = 2 logn

O (n log n)

**Question 8)**

T(s,e,array){

If s == e

Return array[s]

}

merge(TS(S,Le/3), array), TS(le/3) + 1, 2(le/3),array), TS(2 le/3) + e, n,array)

TS(n) {C if s == e, 3TS(n/3) + n

Complexity:

TS(n) = 3TS(n/3) + n

TS(n/3) = 3 TS(n/9) + n/3

TS(n/9) = 3TS(n/27)+ n/9

T(n) = 3(3 TS(n/9)+n/3) + 4

T(n) = 9(3 TS(n/27) + 3n/9) + 24

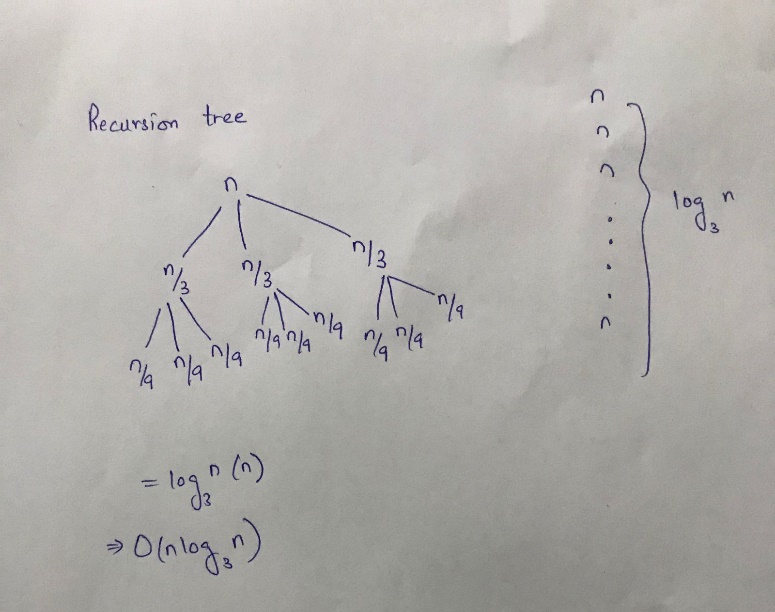
T(n) = 3i (TS( n / 3i) + in

n/3i = 1 => n = 3i => log3n = i

3 log3n + n log3n

O(nlog3n)

Recursion Tree:



Master Theorem:

a = 3, b = 3, d = 1

a = bd

O(n log3 n)

4. Binary merge sort is better because it compute less and has a time complexity log3n which is better. While in terms of space complexity it is same.

**Question 9)**

TS(l, r, val, arr)

If r>= 1

mid1 = 1 +(r-1)/3

mid2 = r + (r –l) /3

if val == arr[mid1]

return mid1

else if val == arr[mid2]

return mid2

if val < arr[mid1]

return TS(l, mid1-l, val, arr)

else if val > arr[mid2]

return TS(mid2+l, r, val, arr)

else

TS(mid1 +l, mid2 – l , val, arr)

TS(n) = {c if val == mid1 or val == mid2 TS(n/3) +c

1. Back substitution

TS(n) = TS(n / 3) + c

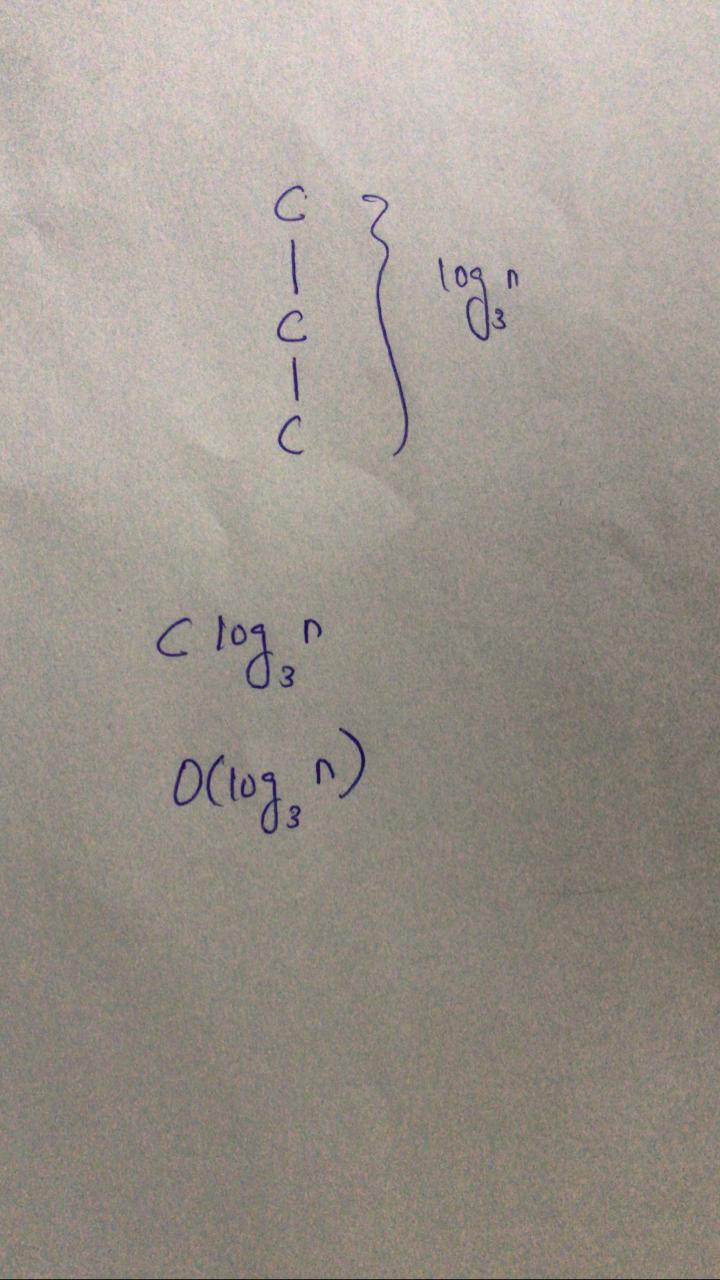
TS(n) = TS (n/9) + 2c

TS(n) = TS(n/3i) + ic

n/3i = 1 => i = log3 nc

O(log3n)

1. Recursion tree:



1. Master theorem:

a = 1, b = 3, d = 0

a = bd

nd log3n

O(log3n)

**Question 10)**

1. Brute force:

ans <- ans

for i <- 2 to n

ans <- ans \* a

return ans

Complexity:

n => O (n)

1. Decrease by 1:

f(a,n){

if n = 1

return a

return a \*f(a,n-1);

}

Complexity:

f(n) = f(n-1) + c

f(n) = f(n-2) + 2c

f(n) = f(n – n + 1) + (n – 1)c

f(n) = 1 + n – c

O(n)

1. Decrease by factor:

f(a,n){

if n = 1

return a

return f(a, (n/2))\* f(a, (n/2))

}

Complexity:

T(n) = { c if n= 1 2T(n/2) + c

a = 2 , b = 2, n = 0

a = bd

nlog22 => n

O(n)

1. Divide and conquer:

f(a,n){

if n = 1

return a

return f(a, (n/2))\* f(a, (n/2))

}

Complexity:

T(n) = { c if n= 1 2T(n/2) + c

a = 2 , b = 2, n = 0

a = bd

nlog22 => n

O(n)

**Question 11)**

1. Unsorted

max <- a[0]

small <- a[10]

for i <- 1 to n – 1

if max < a[i]

max = a[i]

else if small > a[i]

small = a[i]

return max – small

Complexity:

=> O(n)

1. Sorted:

max <- a[a.len]

small <- a[0]

return max – small

Complexity:

C + C = 2 C => O(1)

1. An array concatenation of 2

max1 <- arr[last item of first sorted list]

max2 <- arr[last item of second sorted list]

min1 <- arr[first element of first list]

min2 <- arr[first item of second sorted list]

max <- max(max1, max2)

small <- min(min1, min2)

return max – small

Complexity:

C + C + C + C + C + C + C= 7 C => O(1)

1. Sorted linked list

max <- last element of list

min <- first element of list

return max – min

Complexity:

n + C = 7 C => O(n)

1. BST

max <- root

min <- smallest item

return max – min

Complexity:

C + log n= O(log n)

1. Comparison

It has same space complexity. While the sorted arrays has least operations to perform hence least time complexity.

**Question 12)**

1. nCr =
2. nCk =
3. dynamic programming is the best designing technique for this task because many subproblems are being repeated.
4. Brute force

For

nfic <- 1

rfic <- 1

diffic <- 1

ans <-1

for i <- 1 to max(n,r)

ans <- ans \* i

if i == n

nfic <- ans

if (n-r) == i

diffic <- ans

if r == n

rfic <- ans

return nfic /(diffic \* rfic)

Complexity:

O(max(n,r))

For

kfic <- 1

nlfic <- 1

klfic <- 1

dif1 <- 1

dif2 <- 1

ans <-1

for i <- 1 to max(n-1,k)

ans <- ans \* i

if i == n-1

nlfic <- ans

if k-1 == i

klfic <- ans

if k == i

kfic <- ans

if (n-1) – (k-1) == i

dif2 <- ans

if (n-1)-k == i

dif1 <- ans

return nlfic /(dif1 \* kfic) + nlfic / difl + klfic

Complexity:

O(max(n-1,k))

1. **Decrease by one:**

For

Let x = (n-r)

F(n,r,x){

If 2n < 1 and r < 1

return 1

if x == 1

return (n/ r) \* f(n-1,r,x)

if r == 1

return n \* f (n-1, 1, 1)

return (n/(x\*r))\* f(n-1, r-1, x-1)

}

Complexity:

F(x) = { C if n < 1

= { f(n-1) + c

f(n-1) = f(n-2) + 2C = f(n-3) + 3C

f(n-n) mC => nC

O(n)

1. Divide and Conquer:

For

f(n,k){

if (k > h)

return 0

if k = 0 || k == 1

return 1

return f(n-1, k-1)+ f(n-1, k)

}

Complexity:

F(x) = { C if n < k or k =1 or k = 0

= { T(n-1,k) + T(n-1,k-1)

O(2n)

1. In this case, brute force approach is considered the best as it hae time complexity n and less space complexity.

**Question 13)**

The algorithm needs to list all possible walks of the robot (where the robot can take steps of 1 or 2 or 3 meter only), thus let us call the algorithm listwalk3.

//input: positive integer n and some string s (that is set to the null string).

* When n = 1, then there is l way for the robot to walk 1 meter: one step of 1 meter.
* When n = 2, then there are 2 ways for the robot to walk 2 meters: two steps of 1 meter each or one step of 2 meter.
* When n = 3, then there is 4 ways for the robot to walk 3 meters: three steps of 1 meter each, one step of 2 meter followed by one step of 1 meter, one step of 1 meter followed by one step of 2 meter, or one step of 3 meter.
* When n > 3, then the robot first takes a step of 1, 2 or 3 meter and there are then sn-1, Sn-2, and sn-3 Ways to walk the remaining meters.

//Input: n, s

//Output: A listing of all possible walks of the robot.

listwalk3(n,s) {

if (n == 1) then

println(s+"take one step of length 1")

return

if (n == 2) then

println(s+"take two steps of length 1")

println(s+"take one step of length 2")

return

if(n==3) then

println( s+"take three steps of length 1")

println(s+"take one step of length 2"-"take one step of length 1")

println(s+"take one step of length 1"+"take one step of length 2")

println(s+"take one step of length 3")

return

t=s+"take one step of length 1"

listwalk3(n-1,t)

u=s+"take one step of length 2"

listwalk3(n -2,u)

v=s+"take one step of length 3"

listwalk3(n-3,v)

}