HW3 ①

Can Duyar
171044075

Q1)

a)

```
Algorithm alg1(L[0... n-1])
  if (n==1) return L[0]
  else
    tmp = alg1(L[0...n-2])
    if (tmp <= L[n-1]) return tmp
    else return L[n-1]
```

→ This algorithm finds the smallest element of the given array.

$T(n) = T(n-1) + 1$ for $n>1$, $T(1)=0$

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

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

substitute $T(n-1)$ $\quad T(n) = [T(n-2)+1] + 1$

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

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

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

⋮ continue for k times

$T(n) = T(n-k) + k$

↳ Assume $n-k=1 \rightarrow k=n-1$

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

$T(n) = \underbrace{T(1)}_{0} + n-1$

$T(n) = n-1$ → Therefore,

Result

Time Complexity → $\boxed{\Theta(n)}$

Scanned with CamScanner

b)

```
Algorithm alg2 (X[l ... r])
    if (l==r) return X[l]
    else
      flr = floor(((l+r)/2)
      tmp1 = alg2(X[l ... flr])
      tmp2 = alg2(X[flr+1 ... r])
      if (tmp1<=tmp2) return tmp1,
      else return tmp2
```

Recurrence Relation

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1$ for $n>1$, $T(1) = 0$

$T(n) = 2T(n/2) + 1$

we can use backward substitution to solve it, with using $n = 2^t$

$T(2^t) = 2T(2^{t-1}) + 1 = 2[2.T(2^{t-2})+1] + 1 = 2^2 T(2^{t-2}) + 2 + 1$

$= 2^2[2T(2^{t-3}) + 1] + 2 + 1 = 2^3 T(2^{t-3}) + 2^2 + 2 + 1 = \dots$

$= 2^p T(2^{t-p}) + 2^{p-1} + 2^{p-2} + \dots + 1 = \dots$ for $p=t$

$= 2^t T(2^{t-t}) + 2^{t-1} + 2^{t-2} + \dots + 1 = 2^t \cdot \overset{0}{\widetilde{T(1)}} + 2^{t-1} + 2^{t-2} + \dots + 1$

$= 2^t - 1 = \underline{\underline{n-1}}$

$T(2^t) = 2^t - 1$ $\quad 2^t = n$

$T(n) = n - 1$

Therefore

Result

$\boxed{\text{Time Complexity} = \Theta(n)}$

$\Longrightarrow$ As a result, time complexity is "$\Theta(n)$" for both of the algorithms, so their performances are same. We can prefer both of them for the same problem! (But second algorithm calles itself as 2 times and first algorithm calles itself as one time, in this case maybe first algorithm can be better in terms of space)

Q2) Algorithm Polynomial_Brute_Force (P[0...n], x)

```
  t ← 0.0
  for p ← n downto 0 do
    degree ← 1
    for g ← 1 to p do
      degree ← degree * x
    endfor
    t ← t + P[n-p] * degree
  endfor
  return t
end
```

Time complexity of the algorithm

$$T(n) = \sum_{p=0}^{n} \sum_{g=1}^{p} 1 = \sum_{p=0}^{n} p = \underbrace{0 + 1 + \ldots + n}_{} = \frac{n.(n+1)}{2}$$

$$= \frac{n^2+n}{2}$$

Therefore, Time complexity $= \boxed{\Theta(n^2)}$

⟹ It's possible to design an algorithm that has better complexity!

Algorithm Polynomial_Brute_Force2 (P[0...n], x)

```
  K ← P[n]
  degree ← 1
  for g ← 1 to n do
    degree ← degree * x       } constant time for two operations
    K ← K + P[n-g] * degree
  endfor
  return K
end
```

Time complexity

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

we can ignore the coefficients

Therefore, Time complexity $= \boxed{\Theta(n)}$

Q3)

Algorithm count_substr_brute-force (str[0...n-1], start, end)

```
Count ← 0
for t ← 0 to n do
    if str[t] == start
        for g ← t+1 to n do
            if str[g] == end
                Count = Count + 1   → operation with constant time
            endif
        endfor
    endif
endfor
return Count
end
```

Time Complexity

$$T(n) = \sum_{t \leftarrow 0}^{n} \sum_{g \leftarrow t+1}^{n} 1 = \sum_{t \leftarrow 0}^{n} n-(t+1)+1 = \sum_{t \leftarrow 0}^{n} n-t = n+(n-1)+(n-2) + \dots + (n-n)$$

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

$$= \frac{n^2+n}{2}$$

Therefore, time complexity is → $\boxed{\Theta(n^2)}$ //

Q4)

```
Algorithm closest-pair_brute-force ( PointArray[(x0,y0),(x1,y1).. (xn-1,yn-1)])
   keep = {}
   keep["point1"] ← PointArray[0]
   keep["point2"] ← PointArray[1]
   keep["dist"] ← sqrt((PointArray[1][0] - PointArray[0][0])**2 +
                      (PointArray[1][1] - PointArray[0][1])**2)

   for t←0 to n-1 do
     for g←t+1 to n do
       dist ← sqrt((PointArray[t][0] - PointArray[g][0])**2 +
              (PointArray[t][1] - PointArray[g][1]**2)

       if dist < keep["dist"]
          keep["dist"] ← dist
          keep["point1"] ← PointArray[t]
          keep["point2"] ← PointArray[g]
       endif
     endfor
  return keep        // it returns pair of points and its distance.
  endfor
end
```

constant time operations

Time complexity Analysis

$$T(n) = \sum_{t \leftarrow 0}^{n-1} \sum_{g \leftarrow t+1}^{n} 1 = \sum_{t \leftarrow 0}^{n-1} n-(t+1)+1 = \sum_{t \leftarrow 0}^{n-1} n-t = n+(n-1)+\ldots 1$$

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

Therefore,

time complexity $= \boxed{\Theta(n^2)}$

Q5)

a)

```
Algorithm mostProfitableClusters(clusters[0... n-1])
    max ← 0
    Companies = [ ]
    for g ← 0 to n do
        iter ← 0
        for t ← g to n do
            iter ← iter + clusters[t]      ]  operations with
            if iter > max                  ]  Constant time
                max ← iter                 ]

    for i ← 0 to n
        sum ← 0
        for v ← i to n do
            sum ← sum + clusters[v]                   ]  operations with
            if sum == max                             ]  constant time
                companies.append(clusters[i:v+1])     ]

    return Companies     // it returns the profit values of specific companies
                            to identify them.
```

Time complexity

$$T(n) = \underbrace{\sum_{g \leftarrow 0}^{n} \sum_{t \leftarrow g}^{n} 1}_{T_1(n)} + \underbrace{\sum_{i \leftarrow 0}^{n} \sum_{v \leftarrow i}^{n} 1}_{T_2(n)}$$

$$T_1(n) = \sum_{g \leftarrow 0}^{n} (n-g)+1 = (n+1) + n + (n-1) + \dots 1 = \frac{(n+1)\cdot(n+2)}{2} = \frac{n^2+3n+2}{2}$$

$$T_2(n) = \sum_{i \leftarrow 0}^{n} (n-i)+1 = (n+1)+n+(n-1)+\dots 1 = \frac{(n+1)\cdot(n+2)}{2} = \frac{n^2+3n+2}{2}$$

$$T(n) = T_1(n) + T_2(n) = \cancel{2}\cdot\left(\frac{n^2+3n+2}{\cancel{2}}\right) = n^2+3n+2$$

Therefore the time complexity is $\boxed{\Theta(n^2)}$

b)

```
Algorithm maximumProfit(profitArray[0 ... n-1], l ← None, r ← None)
    left_iter ← 0
    right_iter ← 0
    summation ← 0

    if n == 0 return 0

    if r is None and l is None
        l ← 0
        r ← n-1
    endif
    if r == l
        return profitArray[l]
    endif

    mid = (l+r)/2

    for i ← mid downto l-1 do
        summation ← summation + profitArray[i]
        if summation > left_iter
            left_iter ← summation
        endif
    endfor
    summation ← 0

    for i ← mid+1 to r+1 do
        summation ← summation + profitArray[i]
        if summation > right_iter
            right_iter ← summation
        endif
    endfor
    maxKeep ← max(maximumProfit(profitArray, l, mid),
                  maximumProfit(profitArray, mid+1, r))

    return max(maxKeep, left_iter + right_iter)
end
```

operations with constant time.

$\sum_{t \leftarrow 1}^{n/2} 1$

operations with constant time.

$\sum_{t \leftarrow 1}^{n/2} 1$

$T(n) = 2T(n/2) + \dots\dots$

$$\sum_{i=1}^{n/2} + \sum_{i=1}^{n/2} = 2\sum_{i=1}^{n/2} = 2(n/2) = n$$

→ first for loop → second for loop.

$T(n) = 2T(n/2) + n \rightarrow f(n)$

a, b

→ we can use master theorem to solve this recurrence relation!

$\log_b^a = \log_2^2 = 1$

$\log_b^a = 1$

$f(n) = n \rightarrow n^k \log^p n \quad k=1$ and $p=0$

$\log_b^a = 1$, $k=1$ $\quad > \log_b^a = k$ and $p=0$, $p>-1$

so it's the first case of the case 2!

Therefore, the time complexity is $\Theta(n^k \log^{p+1} n)$

→ $k=1$ and $p=0$

$\Theta(n^1 \log^{0+1} n) \rightarrow \boxed{\Theta(n \log n)}$

↓ Result.

Master Theorem

$T(n) = aT(n/b) + f(n)$ $\quad$ $a \geq 1$, $b > 1$, and $f(n)$ is asymptotically +

$f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b^a > k$ then $\Theta(n^{\log_b^a})$

Case 2: if $\log_b^a = k$

i) if $p > -1 \rightarrow \Theta(n^k \log^{p+1} n)$

ii) if $p = -1 \rightarrow \Theta(n^k \log\log n)$

iii) if $p < -1 \rightarrow \Theta(n^k)$

Case 3:

if $\log_b^a < k$

if $p \geq 0 \rightarrow \Theta(n^k \log^p n)$

if $p < 0 \rightarrow \Theta(n^k)$