HW-5

Can Duyar
171044075

Q1)

a)
```
Algorithm findMaxProf(KeepProfits[0... n-1])
    Count ← KeepProfits[0]       } → O(1) constant time operations.
    getMax ← KeepProfits[0]      }

    for t←0 to t←n-1 do
        Count ← Count + KeepProfits[t+1]             → O(1) constant time operations.
        Count ← max(KeepProfits[t+1], Count)
        getMax ← max(getMax, Count)
    endfor
    return getMax
end
```

Time complexity analysis

$$T(n) = \sum_{t\leftarrow 0}^{n} 1 = \underbrace{1+1\ldots+1}_{n+1} = n+1 \in \Theta(n)$$

→ This part comes from constant time operations during dynamic programming implementation

$$\boxed{T(n) \in \Theta(n)}$$

→ This algorithm finds the maximum profit belonging to the most profitable cluster (The cluster must contain a consecutive region) based on dynamic programming approach so it solves the problem from bottom to top. It calculates the maximum summation of the subarray of the given array.

recurrence relation of the algorithm:

$$\boxed{r(n) = \max_{0\le i\le n}\left(r_i + \max(KeepProfits[i+1] + KeepProfits[i])\right)}$$

b) My previous solution was that,

```
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

  if r==l
    return ProfitArray[l]
  endif
  mid ← (l+r)/2

  for i ← mid downto l-1 do
    Summation ← summation + profitArray [i]    ] → operations with constant time → O(1)    $\sum_{t\leftarrow1}^{n/2} 1$
    if summation > left_iter
      left_iter ← summation
    endif
  endfor
  summation ← 0

  for i ← mid+1 to r+1 do
    Summation ← summation + ProfitArray [i]    ] → operations with constant time → O(1)    $\sum_{t\leftarrow1}^{n/2} 1$
    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.
```

Time Complexity Analysis.

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

$$\sum_{i\leftarrow1}^{n/2} + \sum_{i\leftarrow1}^{n/2} = 2\sum_{i\leftarrow1}^{n/2} = 2(n/2) = \underline{\underline{n}}$$

$T(n) = 2T(n/2) + n$ ⟶ We can use master theorem to solve this rearronce relation !

③

$T(n) = \overset{a}{2}T(n/\overset{b}{2}) + \overset{f(n)}{n}$

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

$\log_b^a = 1$

$f(n) = n \to n^k \log^p n$ $\underline{K=1}$ and $\underline{P=0}$.

$\log_b^a = 1 \implies \log_b^a = K$ and $P=0$

$K=1$ $\qquad P>-1$

↳ 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 $P=0$

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

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 $P>-1 \to \Theta(n^k \log^{p+1} n)$

ii) if $P=-1 \to \Theta(n^k \log\log n)$

iii) if $P<-1 \to \Theta(n^k)$

Case-3

if $\log_b^a < K$

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

if $P<0 \to \Theta(n^k)$

Comparison of dynamic programming solution and my previous design

Previous algorithm (from HW3) ⤳ $T(n) = \Theta(n\log n)$

dynamic programming ⤳ $T(n) = \Theta(n)$

⟹ dynamic programming is better than my previous solution in terms of time complexity!

→ dynamic Programming uses bottom to top solution instead of top to bottom.

→ My previous solution was using divide & conquer technique!

Q2)

```
Algorithm ProduceCandles (priceList , n)
    temp ← []
    for t ← 0 to t ← n+1 do
        temp.append (0)                    → constant time operation → O(1)
    endfor

    for i ← 1 to i ← n+1 do
        keepMax ← -sys.maxsize             → constant time operation → O(1)
        for j ← 0 to j ← i do
            keepMax ← max (keepMax, priceList[j] + temp[i-j-1])   → constant time operation → O(1)
        endfor
        temp[i] ← keepMax
    endfor
    return temp[n]
end
```

Time Complexity Analysis:

This part comes from the outer for loop.

This part comes from the inner for loop.

$$T(n) = \sum_{t\leftarrow 0}^{n+1} 1 + \sum_{i\leftarrow 1}^{n+1} \sum_{j\leftarrow 0}^{j\leftarrow i} 1 = (n+2) + \sum_{i\leftarrow 1}^{n+1} (i+1)$$

first for loop with constant time operations

it comes from constant time operations

$$= (n+2) + [(2+3+ \dots +(n+2))] = (n+2) + \frac{(n+1).(n+4)}{2} = \frac{2n+4+n^2+5n+4}{2}$$

$$= \frac{n^2+7n+8}{2} \in \Theta(n^2)$$

$$\boxed{T(n) \in \Theta(n^2)}$$

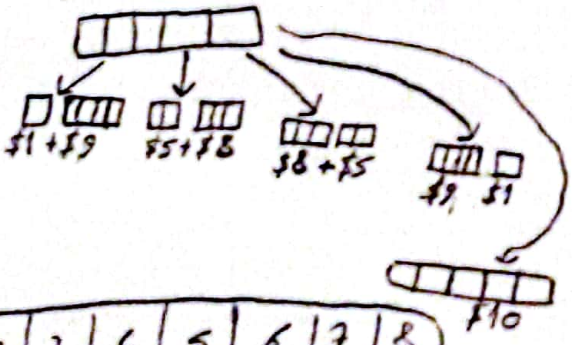
→ This algorithm finds the maximum obtainable value by cutting in different pieces that have different lengths. based on dynamic Programming. So solution approach is based on bottom to top solution instead of top to bottom.

→ basically, for each length I computed,

p_n
$p_1 + r_{n-1}$
$p_2 + r_{n-2}$
...
$p_{n-2} + r_2$
$p_{n-1} + r_1$

$\Longrightarrow$ The recurrence relation for the rod cutting problem is:

$$\boxed{r_n = \max_{1 \le i \le n} (p_i + r_{n-i})}$$



i	1	2	3	4	5	6	7	8
p_i	1	5	8	9	10	17	17	20

Q3)

```
Algorithm maxGreedy(weight, Val, Cap)
    Prods ← []
Θ(n) ( for v ← 0 to length(Prods) do                        → constant time operations O(1)
           Prods.append(Products(weight[v], Val[v], v))
    endfor
    Prods.Sort(reverse ← True)     → Sorting operation takes time as Θ(n log n)
    sum ← 0
    for t in Prods do
        KeepValue ← int(t.Val)              ) → Constant time operations O(1)
        iterWeight ← int(t.weight)          )
        if Cap - iterWeight < 0
            division ← Cap / iterWeight                 ]
            sum ← sum + (KeepValue * division)          ]
            Cap ← int(Cap - (iterWeight * division))    ]  > Constant time operations O(1)
            break                                       ]
        else
            Cap ← Cap - iterWeight                      ]
            sum ← sum + KeepValue                       ]
    endfor
    return sum
end
```

Time Complexity Analysis

$$T(n) = \sum_{v \leftarrow 0}^{n} 1 + \sum_{t \leftarrow 0}^{n} 1 + \Theta(n \log n)$$

first for loop

constant time operations → second for loop.

→ because of sorting operation.

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

$$T(n) = 2n+2 + \Theta(n \log n)$$

$$T(n) = \cancel{\Theta(n)} + \Theta(n \log n)$$

→ Result.

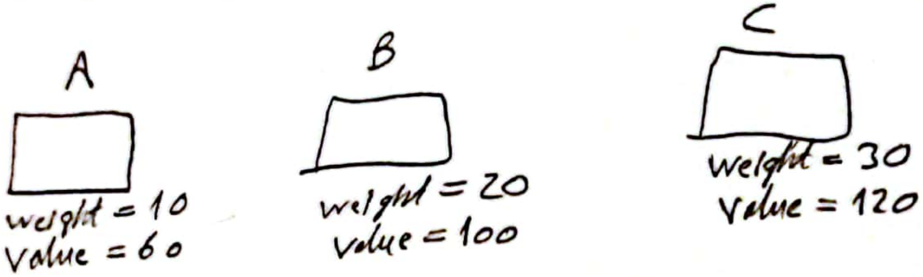
we say that $\boxed{T(n) = \Theta(n \log n)}$

⇒ As main time taking step is "Sorting-operation", the while program can be solved in $\Theta(n \log n)$ only.

→ This algorithm is called as <u>fractional knapsack</u>. In this approach we can break items for maximizing the total value of knapsack

→ using the Greedy method is a good solution. The greedy approach's core principle is to calculate each item's value/weight ratio and arrange the items according to that ratio. Then, stating with the item with the highest ratio, add till we can't add any more of the following item as a whole, and finally, add as much of the next item as possible. which is always the best solution to this problem.

<u>explenation</u>



Let's say that capacity = 50.

According to fractional knapsack approach,

- Take A, B and 2/3rd of C.
- $\sum$ weight $= 10+20+30*(2/3) = 50$
- $\sum$ Value $= 60+100+120*(2/3) = \underline{240}$

Q4)

```
Algorithm maxNumCourses(start[0... n-1], finish[0...n-1])
    courses ← []
    t ← 0                       } → constant time operations O(1)
    courses.append(t)

    for g ← 0 to n do
        if start[g] >= finish[t]
            courses.append(g)
            t ← g
        endif
    endfor
    return len(courses)
end
```

Time complexity Analysis

$$T(n) = \underbrace{\sum_{g \leftarrow 0}^{n}}_{\text{for loop}} \underbrace{1}_{\text{because of constant time operations}} = n+1 \in \Theta(n)$$

$$\boxed{T(n) = \Theta(n)} \rightarrow \text{Result}$$

→ This greedy algorithm finds the minimum number of courses a student can attend among n courses. The greedy option is to always choose the next course whose finish time is the shortest among the remaining courses and whose start time is greater than or equal to the prior course's finish time. We can sort the courses by finishing time so that the next action is always the one with the shortest finishing time.

Briefly, operations are:

1) Sorting the courses according to their finishing time.
2) Select the first course from the sorted array and add it to the new array as first element.
3) Do the following for the remaining courses in the sorted array.
   - if the start time of this course is greater than or equal to the finish time of the previously selected course then select this course and add to the our new array.
4) Function returns the length of our new array (it's the max number of courses a student can attend among n courses)