**A. Time Complexity and Recurrence Relations**

**A.1**

|  |  |  |
| --- | --- | --- |
| Term | Dominant term in O-notation | Order |
| log n + 2n2 + 3n | O(2n2) | 3 |
| log n + | O( | 2 |
| 6 | O(1) | 1 |

**A.2**

1. **T(n) = 16T() + n!**

For this equation: a=16, b=16, f(n)=n!

Intuitively this equation would represent an algorithm that divides the original input into 16 groups, that only uses a sixteenth of the elements at each step and takes … time to combine the results.

Computing: nlogba  = nlog1616  = n

n! > n

Then: f(n) = n! = O(n(1/2)+ε ) which is satisfied for any ε≥ -1/2

**Hence: CASE 3 => T(n) = Θ(n!)**

1. **T(n) = 3nT() + n2**

Does not apply for Master Theorem because a is not a constant.

1. **T(n) = T()+ logn**

For this equation: a =, b=3, f(n)= logn

Intuitively this equation would represent an algorithm that divides the original inputs into groups, that only uses a third of the elements at each step and takes log time to combine the results.

Computing: nlogba  = nlog33^(1/2) = n1/2 =

f(n) <

Then: f(n) = logn = O(n(1/2)-ε ) which is satisfied for any ε≤ ½

**Hence: CASE 1 => T(n) = Θ(nlogba) = Θ()**

**B. Greedy Algorithms**

S = {(1,w1, v1),…(n, wn, vn)

**B.1**

**Strategy 1:**

The items are sorted in the increasing order of their weights(wi≤wi+1). We start with item j=1 and we assign fj=1 as long as , remains less or equal to W, while for all remaining items we assign fj=0.

We will use an example table to demonstrate that this strategy is not optimal.

|  |  |  |
| --- | --- | --- |
| Item | Value | Weight |
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

The maximum capacity of the knapsack is W=11. In order to choose the items for the knapsack we will use: ≤ W.

* Item 1: ≤ W => 1 ≤ 11 : valid (fj=1)
* Item 2: ≤ W => 1+2 ≤ 11 => 3 ≤ 11 :valid (fj=1)
* Item 3: ≤ W => 1+2+5 ≤ 11 => 8 ≤ 11 :valid (fj=1)
* Item 4: ≤ W => 1+2+5+6 ≤ 11 => **14 ≤ 11 :invalid (fj=0)**

Since the items are sorted in the increasing order of their weights we can deduct that item 5 would also exceed the weight limit.

Hence the algorithm takes only the first 3 items whose values are:

V1 =1, V2 = 6, V3 =18

**Total value= V1+ V2 + V3 =25**

We can easily see that this strategy does not lead to an optimal solution since if we chose items 3 and 4 whose combined weight is 11, their value would be V3,4 = 40 > 25.

**Strategy 2:**

The items are sorted in the decreasing order of their values, vi ≥ vi+1.

We start with item j=1 and we assign fj=1 as long as , remains less or equal to W, while for all remaining items we assign fj=0.

Again we will use an example table to demonstrate that this strategy is not optimal.

|  |  |  |
| --- | --- | --- |
| Item | Value | Weight |
| 1 | 28 | 7 |
| 2 | 22 | 6 |
| 3 | 18 | 5 |
| 4 | 6 | 2 |
| 5 | 1 | 1 |

The maximum capacity of the knapsack is W=11. In order to choose the items for the knapsack we will use: ≤ W.

* Item 1: ≤ W => 7 ≤ 11 : valid (fj=1)
* Item 2: ≤ W => 7+6 ≤ 11 => **13 ≤ 11 : invalid (fj=0)**
* Item 3: ≤ W => 7+0+5 ≤ 11 => **12** **≤ 11: invalid (fj=0)**
* Item 4: ≤ W => 7+0+0+2 ≤ 11 => 9 ≤ 11: valid (fj=1)
* Item 5: ≤ W => 7+0+0+2+1 ≤ 11 => 10 ≤ 11:valid(fj=1)

Therefore the algorithm chooses items j=1,4,5 whose values are:

V1=28, V4=6, V5=1

**Total value = V1+ V4 + V5 = 35**

Once again we can see that this strategy is not optimal since if we chose items 2 and 3 whose combined weight is 11, their value would be: V3,4 = 40 > 35.

**B.2**

We assume that we have items j=1,2,…,n

Those items have values v1,…,vn and weights w1,…wn. The knapsack has a maximum capacity W. In order to choose the items for the knapsack the following must be true: ≤ W. We also want: to be maximum.

A greedy selection strategy that would lead to an optimal solution, would be to divide the items into two subsets e.g. A and B of approximately equal size and then compare each item of subset A with all the items in subset B. We then have to find the combination that gives us the greatest value such that the combined weight of the items is equal or less than W.

We will use an example to demonstrate the above strategy.

We assume we have the following items, j=1,2,…,6 and the maximum weight of the knapsack is **W=17**. We assign fj=1 as long as , remains less or equal to W, while for all remaining items we assign fj=0.

|  |  |  |
| --- | --- | --- |
| Item | Value | Weight |
| 1 | 3 | 1 |
| 2 | 7 | 2 |
| 3 | 18 | 5 |
| 4 | 26 | 7 |
| 5 | 45 | 12 |
| 6 | 52 | 13 |

We divide the items into two subsets:

Subset A= {1,2,3}

Subset B= {4,5,6}

We now have to compare each item of subset A with each of the items in subset B and find the combination with the greatest value such that the combined weight of the items is equal or less than W.

* Item 1:

j1,j4 : w=w1+w4=8, v=v1+v4=29

j1,j5 : w=w1+w5=13, v=v1+v5=48

j1,j6 : w=w1+w6=14, v=v1+v6=55

* Item 2:

j2,j4 : w=w2+w4=9, v=v2+v4=3

j2,j5 : w=w2+w5=14, v=v2+v5=52

j2,j6 : w=w2+w6=15, v=v2+v6=59

* Item 3:

j3,j4 : w=w3+w4=12, v=v3+v4=44

**j3,j5 : w=w3+w5=17, v=v3+v5=63**

j3,j6 : w=w3+w6=18>W :invalid

The algorithm chooses items 3 and 5 whose combined weight is:

W3+W5=17 ≤ W and their combined value is: V3+V5=63.

Thus we have maximized the total value while keeping the weights below or equal to W.

**C. Dynamic Programming**

Jobs j=1,…,6

Sj = Start time of each job

Fj = Finish time of each job

S = Maximal weight schedule of compatible jobs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| j | Sj | fj | uj | p(j) |
| 1 | 3 | 6 | 3 | 0 |
| 2 | 4 | 7 | 4 | 0 |
| 3 | 6 | 10 | 3 | 1 |
| 4 | 7 | 11 | 4 | 2 |
| 5 | 10 | 12 | 3 | 3 |
| 6 | 9 | 13 | 5 | 2 |

0 if j=0

OPT (j)=

max(uj + OPT(P(j)), OPT(j-1) ) otherwise

* OPT(0) = 0
* OPT(1) = max(3+OPT(0), OPT(0)) = max (3,0) = 3
* OPT(2) = max(4+OPT(0), OPT(1)) = max (4,3) = 4
* OPT(3) = max(3+OPT(1), OPT(2)) = max (6,4) = 6
* OPT(4) = max(4+OPT(2), OPT(3)) = max (8,6) = 8
* OPT(5) = max(3+OPT(3), OPT(4)) = max (9,8) = 9
* OPT(6) = max(5+OPT(2), OPT(5)) = max (9,9) = 9

**Maximum weight schedule: 9**

In order for a job to be compatible the following must be true:

**Uj + OPT(p(j)) > OPT(j-1),** in which case the post-processing continues by looking into job p(j). If job j is not taken then the post-processing continues with job j-1. Therefore we will create S by starting with job j=6.

* U6 + OPT(2) > OPT(5) => 9 > 9 :invalid, so job 6 ∉ S
* U5 + OPT(3) > OPT(4) => 9 > 8 :valid, so job 5 ∈ S
* U4 + OPT(2) > OPT(3) =>skipped due to post-processing: job 4∉S
* U3 + OPT(1) > OPT(2) => 6 > 4 :valid, so job 3 ∈ S
* U2 + OPT(0) > OPT(1 =>skipped due to post-processing: job 2∉S
* U1 + OPT(0) > OPT(0) => 3 > 0 :valid, so job 1 ∈ S

**S = {1,3,5}**

**D. Sequence Alignment**

We have two strings:

String X: ***T G G A A C***

String Y: ***G T A C***

We have to construct a graph that represents the optimal alignment between String X and String Y.

Mismatch penalty αx.y = 2 for any x ≠ y, x ∈ X , y ∈ Y

Gap penalty δ = 1

To produce the graph we will use the following type:

jδ if i=0

OPT(i,j) = min(αxi, yj)+OPT(i-1, j-1), δ+OPT(i-1,j), δ+OPT(i, j-1) otherwise

iδ  if j =0

The mismatch penalty is the same for all letters and the gap penalty is the same for both strings. Hence the cost can be calculated as follows:

Cost(M) = +

0 T G G A A C

0 0 1 2 3 4 5 6

G 1 2 1 2 3 4 5

T 2 1 2 3 4 5 6

A 3 2 3 4 3 4 5

C 4 3 4 5 4 5 4

The solution graph indicates that the cost of an optimal alignment of strings X and Y is 4. To find the actual optimal alignment we go along the path in the reverse direction from node (|Y|, |X|) to (0, 0). In this case there are multiple optimal alignments (paths) but we will take only the red path.

Red alignment: ***T G G A A C***

**Cost(M) = 1+0+1+2+0+0 = 4**

***- G - T A C***

**D.2**

**a)** An optimal alignment between X and Y with three matches:

0 T G G A A C

0 0 1 2 3 4 5 6

G 1 2 1 2 3 4 5

T 2 1 2 3 4 5 6

A 3 2 3 4 3 4 5

C 4 3 4 5 4 5 4

We have again multiple solutions as before but we will only take the green path:

Green alignment: ***T G G A A C***

**Cost(M) = 1+0+2+0+1+0 = 4**

***- G T A - C***

**b)** An optimal alignment between X and Y with four gaps:

0 T G G A A C

0 0 1 2 3 4 5 6

G 1 2 1 2 3 4 5

T 2 1 2 3 4 5 6

A 3 2 3 4 3 4 5

C 4 3 4 5 4 5 4

Purple alignment:  ***T G - G A A C***

**Cost(M)=1+0+1+1+1+0+0= 4**

***- G T - - A C***