CSE 417 _ HW6 _ Yao-Chung Liang

1. (a) (b) (d)

Index j	Compatible P_j	Value V_j	$\max\{(V_j + opt(P_j)), opt(j-1)\} = opt(j)$	S_j	f_j
0	-	0	0	-	-
1	0	25	$\max\{(25+0), 0\} = 25$	1	4
2	0	20	$\max\{(20+0), 25\} = 25$	2	5
3	0	31	$\max\{(31+0), 25\} = 31$	3	6
4	0	40	$\max\{(40+0), 31\} = 40$	3	7
5	1	35	$\max\{(35+25), 40\} = 60$	4	8
6	3	28	$\max\{(28+31), 60\} = 60$	6	9

(c) By traceback algorithm:

Find-solution(6) $\Rightarrow \because (V_6 + OPT(P_6)) < OPT(5) \equiv (28+31) < 60 \Rightarrow$ Find-Solution(6-1)

$\Rightarrow$ Find-solution(5) $\Rightarrow \because V_5 + OPT(P_5) > OPT(4) \equiv (35+25) > 40 \Rightarrow$ print(5)

$\Rightarrow$ Find-solution(P_5) $\equiv$ Find-solution(1) $\Rightarrow \because V_1 + OPT(P_1) > OPT(0) \equiv 25 + 0 > 0 \Rightarrow$ print(1)

Thus, the optimal solution is (1, 5)

(e) if $V_6 = 30$, then $\max\{(V_j|_{j=6} + opt(P_j)|_{j=6}), opt(5)\} = \max\{30+31, 60\} = 61 = opt(6)$

Find-solution(6) $\Rightarrow \because V_6 + OPT(P_6) < OPT(5) \equiv 30+31 > 60 \Rightarrow$ print(6)

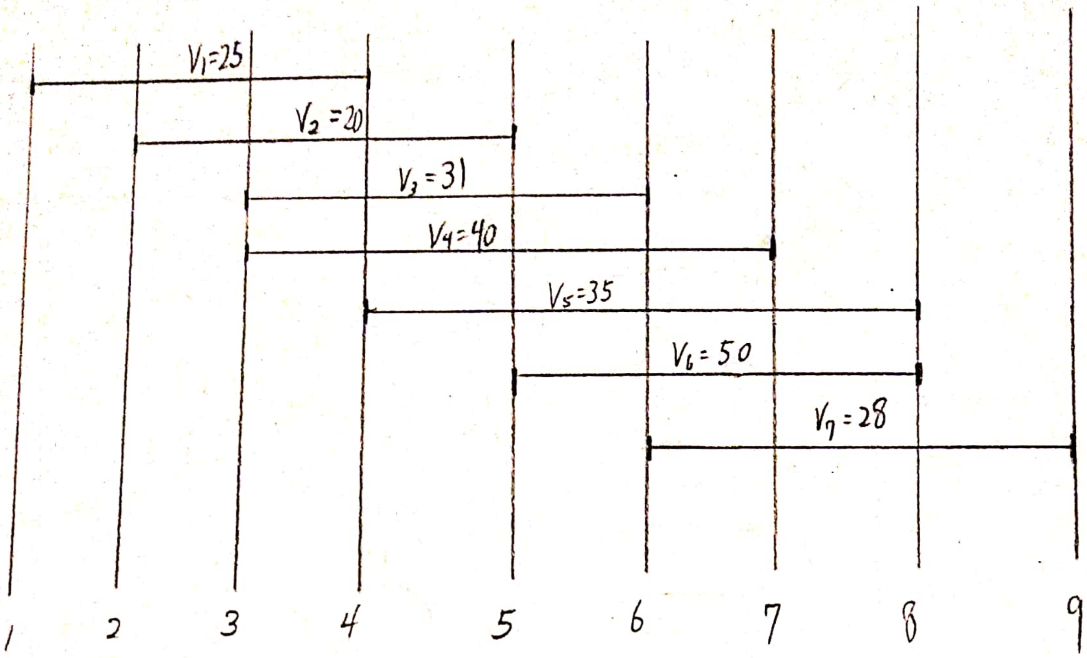
$\Rightarrow$ Find-solution(P_6) $\Rightarrow$ Find-solution(3) $\Rightarrow V_3 + OPT(P_3) > OPT(2) \equiv 31 + 0 > 25 \Rightarrow$ Print(3)

$\Rightarrow$ Find-solution(P_3) $\Rightarrow$ Find-solution(0) $\Rightarrow$ print( )

$\therefore$ optimal solution = (3, 6)

CSE 417

1. (f)



If I choose to use greedy algorithm, in the beginning, I will get V_1 and get a compatible V_5, so the solution would be (V_1, V_5), value is 60, However, optimal solution would be (V_2, V_6) and optimal value is 70, so it gets a suboptimal value and it is not part of any optimal solution.

CSE417

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	ϕ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	{1}	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5
2	{1,2}	0	0	2	2	2	5	5	7	7	7	7	7	7	7	7	7	7
3	{1,2,3}	0	0	2	2	4	5	6	7	7	9	9	11	11	11	11	11	11
4	{1,2,3,4}	0	0	2	3	4	5	6	7	8	9	10	11	12	12	14	14	14
5	{1,2,3,4,5}	0	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

$W_1 = 5$

$W_2 = 2$

$W_3 = 4$

$W_4 = 3$

$W_5 = 6$

OPT(i, w)

Optimum solution is (W_1, W_2, W_4, W_5), total value is $2+5+3+6=16$.

if $w_i > w$

$$OPT[i,w] = OPT[i-1,w]$$

else $$OPT[i,w] = \max\{OPT[i-1,w],\ V_i + OPT[i-1, w-w_i]\}$$

In the beginning, I start from (5,16) in the form which is $OPT[5,16] = 16$,

$$OPT[5,16] = \max\{OPT[4,16],\ 6+OPT[4,10]\} = \max\{14, 6+10\} = 16$$

$$\Rightarrow OPT[4,10] = \max\{OPT[3,10],\ 3+OPT[3,7]\} = \max\{9, 3+7\} = 10$$

$$\Rightarrow OPT[3,7] = \max\{OPT[2,7],\ 4+OPT[2,3]\} = \max\{7, 4+2\} = 7$$

$$\Rightarrow OPT[2,7] = \max\{OPT[1,7],\ 2+OPT[1,5]\} = \max\{5, 2+5\} = 7$$

$$\Rightarrow OPT[1,5] = \max\{OPT[0,5],\ 5+OPT[0,0]\} = \max\{0, 5+0\} = 5$$

$\Rightarrow OPT[0,0]$ Hence, I traced back from $(5,16) \to (4,10) \to (3,7) \to (2,7) \to (1,5) \to (0,0)$

CSE417 HW6_1826630_Yao-Chung Liang

3.

Algorithm	Time
Def knapsack_multi_item(W,n,w(1),w(2),…,w(n),v(1),v(2),…,v(n)):	
For w =0 to W:	O(n)
OPT[0,w]=0	O(1)
For i = 1 to n:	O(n)
For w = 1 to W:	O(W)
If w(i) > w:	
OPT[i,w] = OPT[i-1,w]	O(1)
Else:	
m(i)=int(w/w(i)) # maximum number of this item can take	O(1)
If OPT[i-1,w] > m(i)v(i)+OPT[i-1,w-m(i)w(i)]:	
OPT[i,w]=OPT[i-1,w]	O(1)
Else:	
OPT[i,w]=m(i)v(i)+OPT[i-1,w-m(i)w(i)]	O(1)
Return OPT[n,W],	

Runtime analysis:

O(n)*O(1)+O(n)*O(W)*{O(1)+O(1)*(O(1)+O(1))}=O(Wn)

Correctness:

In the double loops, I constructed two matrices, so for the OPT matrix, I stored values and in Amount matrix I stored the amount of items I took. By considering if OPT[i-1,w] > m(i)v(i)+OPT[i-1,w-m(i)w(i)] or not, I can decide which value I want to put into my matrix of OPT.

When i=1, I will get the maximal value in each OPT(1,w).

When i=2, I will get the maximal value in each OPT(2,w).

When i=n, I will get the maximal value in each OPT(n,w).

Thus in the end, I can find the maximal value in OPT[n,W] and by following the "if else loop" in my algorithm, I can trace back all numbers of items I took.

4.

(a)

If I have a path with sequence of weights 5,7,6, I will get only 7 when using the "heaviest-first" greedy algorithm. However, the independent set of maximal total weight should be 5+6=11.

(b)

If I have a path with sequence of weights 10,1,2,3, I will get only 12 when using the algorithm. However, the independent set of maximal total weight should be 10+3=13.

(c)

Input: path v(1),v(2),..,v(n) with weights w(1), w(2),.., w(n) and the number n.

Output: maximum total weight

Algorithm	Time
Def max_total_weights(n, w(1), w(2),..,w(n)):	
sum_weights = a empty list	O(1)
sum_weights(0)=0	O(1)
sum_weights(1)=w(1)	O(1)
for i =2 to n:	O(n)
if sum_weights(i-1)>(w(i)+sum_weights(i-2)):	
sum_weights(i)=sum_weights(i-1)	O(1)
else:	
sum_weights(i)=w(i)+sum_weights(i-2)	O(1)
return sum_weights(n)	

Runtime Analysis:

O(1)+O(1)+O(1)+O(n)*(O(1)+O(1))=O(n)

Correctness:

If I want the node v(i) then I cannot take v(i-1) but I can get the node v(i-2). And if I want the node v(i-1) then I'll not take v(i). With this logic, I can compare sum_weights(i-1) with (w(i)+sum_weights(i-2)) to take the larger value of weight each time I encounter a new node. Thus, the highest set of weights will be inherited and generate the result of maximum total weights.

When i=1, my sum_weights(1) will be w(1), which is the highest value in sum_weights.

When i=2, my sum_weights(2) will consider to take this node or not and compare the result to get a higher value. Thus right now sum_weights(2) is the highest value in sum_weights.

When i=n, my sum_weights(n) will consider to take this node or not and compare the result to get a higher value. Thus right now sum_weights(n) is the highest value in sum_weights.

In the end, I just need to return the value in sum_weights(n) and this is my optimal solution. And If I want to trace back, I just need to follow judgement loop in the algorithm and I will get a set of nodes I take.

5.

(a)

	Week 1	Week 2	Week 3
Low stress job	5	6	20
High stress job	3	50	100

From the algorithm, because high stress job in week 2 is higher than the summation of low stress jobs in week 1 and week2, so I will get 50+20=70. However, optimal solution is 100+5=105.

(b)

Algorithm	Time
Def hacker_job(n , L(1),L(2),…,L(n), H(1),H(2),…,H(n)):	
OPT[0,None]=0	O(1)
OPT[0,high]=0	O(1)
OPT[0,low]=0	O(1)
For i = 1 to n	O(n)
OPT[i,None]=max[OPT[i-1, high] , OPT[i-1, low]]	O(1)
OPT[i,low]=max[OPT[i-1, high] , OPT[i-1, low]]+L(i)	O(1)
OPT[i,high]=OPT[i-1, None] +H(i)	O(1)
Return max value in OPT	

Example:

Input:

	Week 1	Week 2	Week 3	Week 4	Week 5
l(i)	5	6	9	50	30
h(i)	7	12	18	100	90

Output:

OPT table	high	low	None
Week 0	0	0	0
Week 1	12	13	7
Week 2	25	22	13
Week 3	113	75	25
Week 4	115	143 (optimal)	113

Runtime analysis:

O(1)+O(1)+ O(1)+ O(n)*(O(1)+ O(1)+ O(1))=O(n)

Correctness:

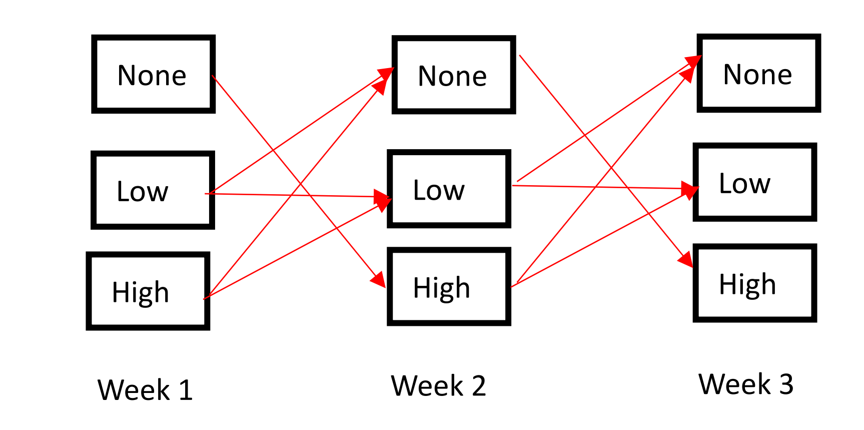


Fig 1. State transition.

Every state, from the beginning in the algorithm, I keep track of every state transition and record what they decide with the value they earned each time. Thus, after accumulating to the end, say week n, there would be three states exist with the maximal reward in it. Thus, I only need to do is to compare the reward inside the three states in week n and I will get the maximal value. And If I want to trace back, I just pick the state with the highest value and follow my state transition rules, you will finally get a list of jobs with the highest total reward.

When i =1, means in week 1, I will record the maximal reward in every states in week 1.

When i=2, I will start to consider like from which way to the current state I will get the maximal reward and record them in the states in week2.

Because I will keep the highest possible reward in my states in every week. When i=n, I will get maximal reward from week n-1.

And from the procedure I present, there must exist a value in week i+1 that is higher or no less than all the value in week I, thus I can infer that there exist a maximal reward in week n that is the optimal solution.