CSE 417 HW6

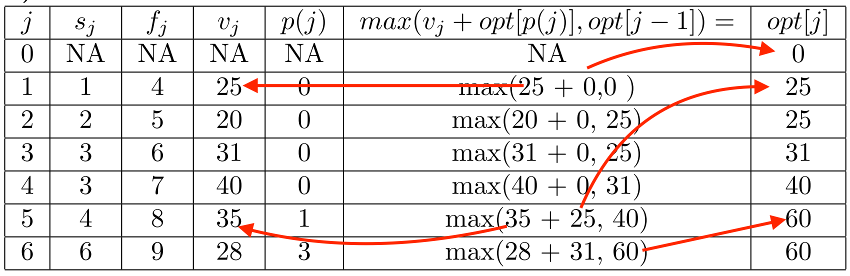
Zachary McNulty (zmcnulty 1636402)

February 2019

1 Problem 1

j	s_j	f_j	v_j	$p(j)$	$max(v_j + opt[p(j)], opt[j-1]) =$	$opt[j]$
0	NA	NA	NA	NA	NA	0
1	1	4	25	0	max(25 + 0,0)	25
2	2	5	20	0	max(20 + 0, 25)	25
3	3	6	31	0	max(31 + 0, 25)	31
4	3	7	40	0	max(40 + 0, 31)	40
5	4	8	35	1	max(35 + 25, 40)	60
6	6	9	28	3	max(28 + 31, 60)	60

c)
optimal solution includes intervals $\{5, 1\}$ and has value 60

d)


j	s_j	f_j	v_j	$p(j)$	$max(v_j + opt[p(j)], opt[j-1]) =$	$opt[j]$
0	NA	NA	NA	NA	NA	0
1	1	4	25	0	max(25 + 0,0)	25
2	2	5	20	0	max(20 + 0, 25)	25
3	3	6	31	0	max(31 + 0, 25)	31
4	3	7	40	0	max(40 + 0, 31)	40
5	4	8	35	1	max(35 + 25, 40)	60
6	6	9	28	3	max(28 + 31, 60)	60

e)
The only row that would change is the final one. Now, $max(v_j + opt[p(j)], opt[j-1])$ is $v_j + opt[p(j)]$, so $opt[6]$ changes from 60 to 61. However, this will certainty change the traceback quite significantly. Now, the optimal interval set is $\{6, 3\}$ with optimal value 61. So even though the optimal value changed only by a single point, the optimal interval set changed completely.

f)
The point of the last question is to show that the solution set to the weighted interval problem is often sensitive to the values of each given interval and thus

considering only the actual intervals themselves can often generate a suboptimal solution. For example, for any weighted interval problem where the greedy algorithm mentioned actually happens to find the optimal solution, I can generate another problem (and in fact many more) where it is completely wrong. Suppose the greedy algorithm finds an optimal solution to the weighted interval problem defined by the intervals $I = \{i_1, ..., i_n\}$ with corresponding weights $\{v_1, ..., v_n\}$. Let $I' \subseteq I$ be the collection of intervals the greedy algorithm chooses. Simply choose an $i_k \in I \setminus I'$ and generate a new problem with all intervals and weights identical except set $v_k = 1 + \sum_{m:i_m \in I'} v_m$ (i.e. set v_k to one above the previous optimal value). This will not change the interval set I' generated by the greedy algorithm as none of the intervals changed, but the solution is now no longer the optimal one. In fact, not a single one of the intervals chosen as part of I' is actually part of the optimal set in this new problem.

2 Problem 2

$$OPT(n,w) = \begin{cases} OPT(n-1,w) & w_n > w \\ max(OPT(n-1,w), w_n + OPT(n-1, w-w_n)) & otherwise \end{cases}$$

n / w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5	0	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	0	0	2	3	4	5	6	7	8	9	10	11	12	12	14	14	14
3	0	0	2	2	4	5	6	7	7	9	9	11	11	11	11	11	11
2	0	0	2	2	2	5	5	7	7	7	7	7	7	7	7	7	7
1	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We can thus see the optimal solution is:

$$\{w_1 = 5, w_2 = 2, w_4 = 3, w_5 = 6\}$$

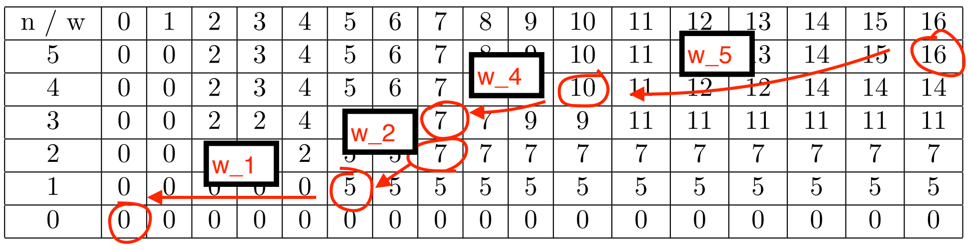
with total value of 16. This is clearly optimal as it produces the maximum allowed weight. To find the traceback, we start at $n = N, andw = W$ and find where $OPT[n,w]$ got its value. If $OPT[n,w] = w_n + OPT[n-1, w-w_n]$ then we add w_n to the optimal solution and continue our search at $OPT[n-1, w-w_n]$. Else, we do not add w_n to our optimal set and we continue our search at $OPT[n-1,w]$. We will stop when we reach $n = 0$ or $w = 0$.

Traceback
$OPT[5,16] = 16 = w_5 + OPT[4,10]$. Thus, w_5 is in our optimal solution.
$OPT[4,10] = w_4 + OPT[3,7]$. Thus, w_4 is in our optimal solution.
$OPT[3,7] = OPT[2,7]$. Thus, w_3 is not in our optimal solution.
$OPT[2,7] = w_2 + OPT[1,5]$. Thus, w_2 is in our optimal solution.
$OPT[1,5] = w_1 + OPT[0,0]$. Thus, w_1 is in our optimal solution.

Thus, the optimal solution is, as we saw earlier:

$$\{w_1 = 5, w_2 = 2, w_4 = 3, w_5 = 6\}$$

Below is the traceback drawn on a copy of the table.



n / w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5	0	0	2	3	4	5	6	7	8	9	10	11	$\boxed{w_5}$	13	14	15	16
4	0	0	2	3	4	5	6	7	$\boxed{w_4}$	[illegible]	10	11	12	12	14	14	14
3	0	0	2	2	4	$\boxed{w_2}$	[illegible]	7	7	9	9	11	11	11	11	11	11
2	0	0	$\boxed{w_1}$	[illegible]	2	5	5	7	7	7	7	7	7	7	7	7	7
1	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3 Problem 3

Algorithm

Define $OPT[n, w]$ to be the optimal total value that can be obtained using only items $\{1, 2, ..., n\}$ with a total weight no more than w.

$$OPT[n, w] = \begin{cases} OPT[n-1, w] & w_n > w \\ max(OPT[n-1, w], v_n + OPT[n, w - w_n]) & otherwise \end{cases}$$

Then, OPT[N, W] will be our optimal value. Calculate the optimal object set by performing the typical traceback technique by recursively finding where $OPT[n, w]$ got its value from. For example, start with $n = N$ and $w = W$, the total object count and weight capacitiy respectively. If $OPT[n, w] = v_n + OPT[n, w - w_n]$, then do add another of object n to the optimal solution and continue our search with $OPT[n, w - w_n]$. Else, $OPT[n, w] = OPT[n-1, w]$ so do not add any more of object n and continue our search with $OPT[n-1, w]$

Correctness

Consider trying to calculate the value of $OPT[n, w]$. If we know the value of $OPT[i, j]$ for all $0 \leq i \leq n$ and $0 \leq j \leq w$, then our problem breaks down into two cases:

1. Case 1: There are none of object n in the optimal solution, in which case $OPT[n, w] = OPT[n-1, w]$.
2. Case 2: There is at least one of object n in the optimal solution. Taking one of these objects and setting it aside, we still have the option of choosing more of object n with the remaining $w - w_n$ weight. Thus, $OPT[n, w] = v_n + OPT[n, w - w_n]$.

Thus, we can clearly break down $OPT[n, w]$ into the two relevant subproblems. We also see that $OPT[n, w]$ strictly relies on the calculation of OPT at lower n and w values. Thus, filling out the OPT table by traversing $n : 0 \to N$ and $w : 0 \to W$ will calculate all necessary subproblems in a consistent manner.

Complexity

There are $N * W$ entries within our OPT table, and each entry is calculated using a simple comparison and basic $O(1)$ arithmetic operations. Thus, the work for each entry is $O(1)$ and this algorithm takes $O(NW)$ time to run.

4 Problem 4

a)

Consider the nodes v_1, v_2, v_3 with corresponding weights $w_1 = 2, w_2 = 3, w_3 = 2$. The given algorithm will generate $\{v_2\}$ as the optimal independent set with value 3, but clearly $\{v_1, v_3\}$ is optimal with value 4.

b)

Consider the nodes v_1, v_2, v_3, v_4 with corresponding weights $w_1 = 10, w_2 = 1, w_3 = 1, w_4 = 9$. The given algorithm will generate $\{v_1, v_3\}$ as the optimal independent set with value 11 but clearly $\{v_1, v_4\}$ is optimal with value 19

c)

Algorithm

Let $\{v_1, v_2, ..., v_n\}$ be our set of nodes and $\{w_1, ..., w_n\}$ their corresponding weights. Define $OPT[i]$ to be the optimal value of the independent set that uses only the first i nodes $\{v_1, v_2, ..., v_i\}$. Then:

$$OPT[i] = max(OPT[i-1], w_i + OPT[i-2])$$

Once the OPT table is filled, out, run the basic traceback algorithm to find where $OPT[i]$ obtained its value from. Starting at $i = n$, if $OPT[i] = w_i + OPT[i-2]$ include v_i in the optimal independent set and move to $OPT[i-2]$. Else, do not include v_i in the optimal set and move to $OPT[i-1]$.

Correctness

Consider a given node v_i. There are only two options: either v_i is in the optimal independent set including only the first i nodes or it is not. If v_i is not in the optimal independent set, then the optimal independent set must occur entirely inside the first $i-1$ nodes and thus $OPT[i] = OPT[i-1]$. If v_i is in the optimal independent set including only the first i nodes, v_{i-1} cannot be as it is adjacent to v_i. Other than that, there are no restrictions on what other vertices can be in the set. Thus, $OPT[i] = OPT[i-2]$. We can see that each OPT calculation only requires the values of OPT at lower indices, so calculating $OPT[i]$ for $i : 0 \to N$ will allow the OPT table to be filled out consistently (subproblems before problems that require them).

Complexity

We perform a $O(1)$ basic comparison and a pair of $O(1)$ array lookups at each of the N indices of OPT. Thus, overall this algorithm is $O(N)$.

5 Problem 5

a)

day	1	2	3
ℓ_i	5	4	1
h_i	1	10	1000

As $h_2 > \ell_1 + \ell_2$, we will choose to do no job during day 1 and choose the high stress job during day 2. We can now not do job h_3 as we did a job on day 2 so this policy is clearly not optimal.

b)

Algorithm

Let $OPT[i]$ be the optimal value obtainable from taking jobs during only the first i days. Thus :

$$OPT[i] = max(\ell_i + OPT[i-1], h_i + OPT[i-2])$$

Then, the value for the optimal plan will be $OPT[N]$. If we also wanted the optimal schedule, we could run a traceback procedure. Starting at $i = N$, if $OPT[i] = \ell_i + OPT[i-1]$, the optimal schedule includes job ℓ_i. Else, the optimal schedule includes job h_i.

Correctness
Consider trying to construct $OPT[i]$ given that you already know $OPT[k]$ for $0 \leq k < i$. For $OPT[i]$, we do not have to worry about any jobs following job i, so our sole choice is to whether or not we should do a stressful job on this day or an easy job.

1. Case 1: If we take a stressful job, we cannot take a job in the preceeding day. Thus, the optimal way to do this is the optimal way to take the first $i-2$ jobs. This yields a value of $h_i + OPT[i-2]$.

2. Case 2: If we take a relaxed job, there are no restrictions on the preceeding jobs we can take so the optimal way to do this is the same as the optimal way to take the first $i-1$ jobs which we already know. This yields a value of $h_i + OPT[i-1]$

As these are the only two options, the optimal job schedule generates an optimal value of the max of these two. By induction, we have then shown that this generates the optimal schedule. Since each OPT[i] only relies on lowered indexed entries in OPT, we can calculate OPT[i] in the order $i : 0 \to N$ to consistently fill out the table (i.e. already have subproblems filled out by the time we get to the problems they are needed in).

Complexity

The OPT table is of size N and filling in each entry of the table only takes a simple comparison and a pair of table look ups, all $O(1)$ operations, so overall the algorithm runs in $O(N)$ time.

6 Problem 6

Define SLACK[i,j] to be the slack associated with the line containing words w_i through w_j:

$$SLACK[i,j] = L - \sum_{k=i}^{j-1}(c_k + 1) + c_j$$

Firstly, note that SLACK[i,j] = 0 if $j < i$ and that $SLACK[i,j] = c_i$ if $i = j$. Furthermore, note that from this definition for slack it is very easy to calculate adjacent entries in the SLACK matrix:

$$SLACK[i,j] = SLACK[i+1,j] - 1 - c_i = SLACK[i,j-1] - 1 - c_j$$

Using either of these two identities, we can loop over i and j and calculate each entry of SLACK in constant time (rather than the linear time required if we calculated the sum from scratch each time). Thus, the whole matrix can be filled in $O(N^2)$ time. Go through SLACK and set all negative slacks to ∞ to prevent invalid word sets from being chosen and square all the other entries, again only taking $O(N^2)$ time.

Once we have the SLACK matrix, define $OPT[i]$ to be the minimum sum of slack squared using only the first i words. Given that we know OPT for $0 \leq k < i$, the only question is which words do we put on the same line as word w_i? Thus, we can see:

$$OPT[i] = min_t(SLACK[t,i] + OPT[t-1])$$

i.e., we place words $\{w_t, w_{t+1}, ..., w_i\}$ on the same line and then we plop that on top of the optimal way to arrange words w_i through w_{t-1} onto their own lines, OPT[t-1]. Since this table has only N indices and we loop over $O(N)$ of them during the minimization part of finding OPT[i], this is an $O(N)$ operation. Thus, overall this is $O(N^2)$.