Homework 3 Solutions

Maxwell Young

October 15, 2003

Question 16.1-2

I explain one possible way the algorithm might be implemented; you didn't need to do this for the assignment (but I gave marks to those who bothered to do so). The algorithm should start by sorting the activities according to decreasing order of start time and put these into an array S, that is:

$S = \{a_1, a_2, ..., a_n\}$ where a_1 was the activity that started last.

Then we start making our largest compatible set A of activities. We begin by choosing the activity with the last start time, that is a_1, and adding it to A. Then for all the remaining activities, we go to the next start time in our sorted array and then check if the finish time of that activity is less than or equal to the start time of the previous activity we just completed. If it is, we add this activity to A. If it is not, we go to the next start time in our sorted array and perform the same check. We keep doing this until we have exhausted all of the activities in our array). That is:

```
for $m$=2 to $n$
    if $f_m \leq s_i$
        then $A = A \cup \{a_m\}$
        $i = m$
return $A$
```

where m is indexing our sorted array and a_i is the activity we just added. One simple way of viewing this new activity selection process is to draw out your activities in the bar format; then, rotate your page by 180 degrees, thereby treating start times as finish times and finish times as start times.

Anyway, in general, given the description of the algorithm given by the question, this algorithm is greedy because it chooses the best'local' option available to it at the given time; there is no attempt to plan ahead. The way to prove that this algorithm finds the optimal is similar to that given in the book; that way works (ie. proof that the subproblem is empty etc.). Another method, which I like better is the following. Say that our greedy algorithm returns a set A of compatible activities and say that some supreme being provides us with an optimal solution (doesn't matter how it is obtained). Let a_n be the first activity in which A differs from B, so we have:

$$A = a_1, a_2, ..., a_k, a_{k+1}, ...a_p$$
$$B = b_1, b_2, ..., b_k, b_{k+1}, ...b_p$$

where $a_i = b_i$ for $i = 1, 2, ..., k-1$. Since the greedy algorithm constructed A, a_k must have been chosen to start no earlier than b_k. Hence, we can construct $B' = b_1, b_2, ..., a_k, a_{k+1}, ..., a_p$ which is a valid activity selection and must be no larger than B (since B was optimal) and is the same size as A. Therefore, $|B'| = |A| \geq |B|$ and we have show A is an optimal.

Question 16.1-4

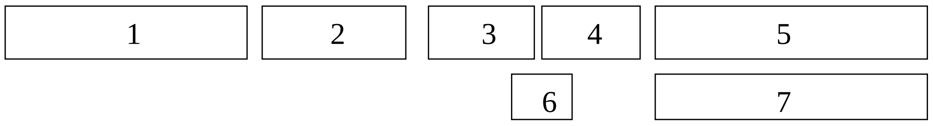
I have presented a counter-example for each case. They are presented in Figure 1 using bar diagrams.

Question 16.2-3

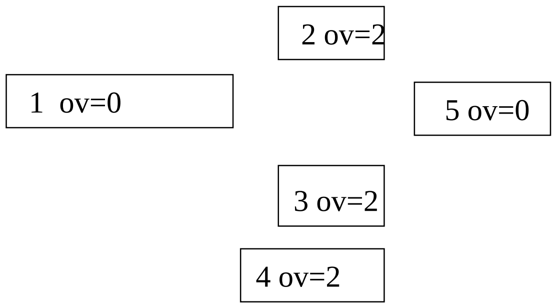
In accordance with the title of this chapter, the best algorithm is a greedy algorithm which takes each item in order until the knapsack can hold no more. That is, we have the following set up:

```
Item  1       2       3       4       ...     n
W     w_{1}  w_{2}  w_{3}   w_{4}           w_{n}
$     v_{1}  v_{2}  v_{3}   v_{4}    ...    v_{n}
```

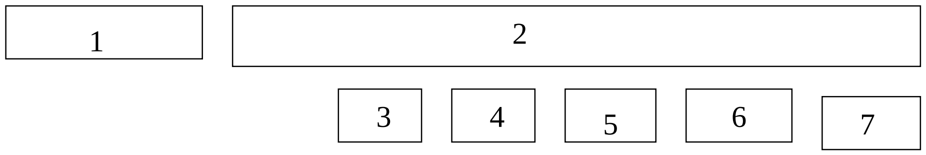
Say we have the partial solution {1,2} for:



The above gives a case where selecting the activity of shortest duration results in a suboptimal solution − that we get {1,2,6,7 or 5} vs the optimal solution of {1,2,3,4,5,6}.



Above, we could start with {1} and then if we pick the next compatible job that overlaps with the fewest remaining jobs we add 5 to our activity list. However, an optimal activity selection has size three ex. {1,2,5}.



Say we start by adding activity 1 to our selection of activities. Then, if we choose the activity with earliest start time, we pick activity 2 and our set of activities is just {1,2}; however, the optimal is {1,3,4,5,6,7}.

Figure 1: Examples of why selection methods don't work

And, by the ordering that we are given, we know that $w_i \leq w_{i+1}$ and $v_i \geq v_{i+1}$.

Now we prove optimality (we are trying to optimize our value). Suppose that at a certain point we have placed the following items in our bag: $1, 2, ..., j-1$ and current value of all these items $V = k$ and the current weight is $W = t$. Now, according to the greedy algorithm we would simply take the next items $j, j+1, ..., l$ if we could fit them into our bag; in this case, $V = k + v_j + v_{j+1} + ... + v_l$ and $W = w_j + w_{j+1} + ... + w_l$. But perhaps there is some better selection of items - is it possible that we might be able to come up with some other group of the remaining items from the remaining items that gives us a **higher** V value? Well, say we have this optimal group of items $m, m+1, ..., q$ (where $m > j$) which gives us $V' = k + v_m + v_{m+1} + ... + v_q$ and $W' = t + w_m + w_{m+1} + ... + w_q$. Furthermore, by our assumption, we have that $W' \leq W$ and $V' > V$. But by the way the v values are ordered, since $m > j$ it must be the case that $v_m + v_{m+1} + ... + v_q \leq v_j + v_{j+1} + ... + v_l$. Hence, we have a contradiction since our supposedly new optimal solution actually has total value less than our greedy solution. Hence, the greedy algorithm is best.

Question 16.2-4

The algorithm we employ here is again greedy. In order to have a minimum number of gas stops, Prof. Midas must drive for as far as he can go and then stop at the gas stop which is:

1. nearest to the n marker (when he runs out of fuel).
2. between him and the n marker.

Does this greedy approach work or is there some way he can make fewer stops? Well, assume that there is some selection S of gas stops which improves upon the greedy strategy. In this better selection, it must be true that none of the distances between stops exceeds n. Say Midas is at a gas stop and now consider two cases that all stops of this better gas stop selection fall into:

1. Midas has travelled k miles so far and the next gas stop is more than $n-k$ miles away.
2. Midas has travelled k miles so far and the next gas stop is $\leq n-k$ miles away.

Start at the beginning of the trip and look at this supposed optimal selection of stops. Well, for case one, we cannot improve and must stop. In the second case, we can lessen the overall number of gas stops by simply not stopping and driving onwards. Hence, if we make this modification we have a scheme that does at least as good as this hypothetical better gas stop selection. BUT this is the greedy algorithm! Hence, the greedy algorithm will allow for a minimal number of stops.

Question 16.2-5

Again, we use a greedy algorithm. Imagine our points in a line. Our greedy algorithm works by starting at the first point and using it as the beginning of the first unit interval we are drawing. We draw the unit interval, covering all points within a distance of 1 from the first point. Then we look to the right and for the first point to the right, we do exactly the same thing. We repeat this process until all points have been covered by unit intervals.

Now let us denote points as p_i. Let (p_i, p_j) denote that points p_i, p_j are contained within the same unit interval. Now, how do we prove the optimality of our greedy algorithm above? First, let us assume we have any optimal solution using S intervals (that is not reached by applying the greedy algorithm). Furthermore, assume that, as given, this solution contains no overlapping intervals (otherwise, we can always fix this and get an equally or even better solution - why?).

Take this optimal solution and start at the left-most point. If the first interval does not have its (left) start point at the first point p_1, then slide the interval to the right until this is the case. Now the interval contains all the points it originally contained. Additionally, it may contain extra points that were originally contained in the interval to its right. If it doesn't happen, then no harm has been done since the interval in question contains the same points and the rest of the optimal solution, from the right onward, remains unchanged. On the other hand, if by moving the interval it *does* contain extra points, then we take the second interval. We push this second interval to the right to the first point not covered by the first interval. If the this relocated second interval does not overlap the third interval, then no harm has been done since intervals 1 and 2 still hold the same points overall. If interval 2 does now overlap interval 3, repeat the process with interval 3. Keep doing this, repeatedly shoving the interval to the right, until no more points remain. At termination, we have certainly not made anything worse since our number of intervals is $\leq S$. Hence, our greedy algorithm is provides an optimal solution.

Question 17.1-1

No, a Multipush operation could ruin the $O(1)$ amortized cost. I could do a sequence of n operations, alternating MULTIPUSH(n) and MULTIPOP(n). This sequence of n operations has total time cost $O(n^2)$.

Question 17.1-2

Consider the case where we have the counter at, say, 00000 (so $k = 5$). Then we can make a call to DECREMENT which then flips all 0s to 1s so we have 11111. Then we can make a call to INCREMENT which flips all 1s to 0s and we have 00000 again. We could do this over a sequence of n operations (ie. alternate between DECREMENT and INCREMENT) - each call requires $O(k)$ operations and we would be doing n of them, so the total time would be $O(nk)$.

Question 17.1-3

We have the following scheme:

Operation	1	2	3	4	5	6	7	8	9	10	11	12	...
Cost	1	2	1	4	1	1	1	8	1	1	1	1	...

So, over n operations, what is the cost? Well, how many powers of 2 do we have between 1 and n; we have $\lfloor lg(n) \rfloor$. So then we have $n - lg(n)$ operations that have an associated value of 1. So the cost for n operations is then:

$$\sum_{i=1}^{n} i^{th} operationcost$$
$$= \sum_{i=1}^{lg(n)} 2^i + n - lg(n)$$
$$= 2^{lg(n)+1} - 1 + n - lg(n) = O(n)$$

So the amortized cost per operation is $O(n)/n = 1$.

Question 17.2-3

First, it is helpful to see why the accountant scheme for the simple (incrementing only) binary counter does not work for this set up of the binary counter with RESET. Let $k = 4$ and consider the point where we have incremented the binary counter from 0000 to 0100. At this point we have 1 dollar in the bank (to pay for the switch from 1 to 0). In this scheme, I do not need any fancy pointer scheme to keep track of which bits to switch to 0. I simply INCREMENT by starting from the right and switching all 1s to 0s until I hit my first 0, switch that to a 1, and then I know I have successfully completed an INCREMENT operation. Note, I never have to know where the 1s and 0s in the binary counter lie - either I hit a zero and I'm done, or I hit a 1 and I have money in the bank to deal with that 1 and move on (repeating until I do hit a zero). I could define a cost for a bit traversal of 1 dollar - but in this case, traveral costs never arise.

However, the RESET function is not so simple, we do have traversal costs. So I have 0100 and I call RESET; I have 1 dollar in the bank. Right off the bat, I hit a 0 and what do I do? Well, I don't know that the next bit isn't a one, so I have to check the next one too. And again and again - I have to check all k bits. So, unlike the simpler version of the binary counter equiped only with INCREMENT, I need to take into account the cost of traversing these k bits, and each traversal costs 1 dollar - so I pay k dollars. Under the orginal accountant scheme, I don't have enough money in the bank to pay for this! What do we do?

The hint says to keep a pointer to the high order 1 in our string of k bits making up the binary counter. How does this help us? If we keep a pointer to the highest order 1, then I don't have to check all k bits when I call a RESET; I will only have to check up to the digit pointed to by this pointer.

We also need to change our collection scheme; we charge 3 dollars for each INCREMENT operation. We will immediately spend 1 dollar to set a bit to 1. Then we will keep 1 bit in reserve to change the 1 back to a zero at some future date. Finally, we will keep the 3rd dollar and use it to set the pointer to the highest order 1. We know from the amortized analysis done in the book, that this scheme will satisfy the INCREMENT calls. Now, how many times do we need to reset the pointer to the highest order 1? Well, the highest order 1 changes every power of 2; so for n operations, we will spend $lg(n) - 1$ dollars on setting this pointer. So, will we have enough money to make this work? Yes. We collet a total of $3n$ dollars. We spend $lg(n) - 1$ on resetting the pointer, n on setting a bit initially to 1, n on setting it back. So before any call to RESET we have $3n - (lg(n) - 1) - 2n = n + 1 - lg(n)$ dollars. So for instance, when we have 0100 we have done $n = 4$, and so we have 3 dollars when we call RESET; note that this gives us enough money to traverse the 2 0s and then 1 dollar left to set the 1 to a 0.

In general, the number of zeros we must traverse till we get to the higher 1 is less than or equal to $lg(n)$. Clearly, $n + 1 - lg(n) \geq lg(n)$ and so, at any point, we will always have enough money to pay for the RESET function call. Overall, we collect $3n$ (we are talking gross now) dollars over n operations; hence, a sequence of n operations (INCREMENT,RESET) has amortized cost O(n).

Question 17.3-2

Again, we have the following scheme:

Operation	1	2	3	4	5	6	7	8	9	10	11	12	...
Cost	1	2	1	4	1	1	1	8	1	1	1	1	...

Here is a potential function that works:

$$\Phi_i = \begin{cases} 0 \text{ , if i is a power of 2} \\ 2(\cdot\text{sum of all 1s from it to the last power of 2}) \text{ , otherwise} \end{cases}$$

So, for example, $\Phi_8 - \Phi_7 = 0 - 6 = -6$ in the first case, whereas $\Phi_7 - \Phi_6 = 6 - 4 = 2$ falls under the second case. Now, we calculate the amortized cost of an operation. In the case where $i = 2^k$ and $i - 1$ is not, we have:

$$\begin{aligned} \hat{c} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 2^k + 0 - (2 \cdot (2^{k-1} - 2)) = 2 \end{aligned}$$

In the case where i is not a power of 2, but $i - 1 = 2^j$, we have:

$$\begin{aligned}\hat{c} &= c_i + \Phi_i - \Phi_{i-1}\\ &= 1 + 2 + 0 = 3\end{aligned}$$

In the case where neither i nor $i-1$ is a power of 2:

$$\begin{aligned}\hat{c} &= c_i + \Delta(\Phi)\\ &= 1 + 2 = 3\end{aligned}$$

In both cases, the amortized cost of an operation is constant.

Question 17.3-7

Let us use an unsorted array to hold our integers. We can keep a variable or pointer to the first unoccupied index in this array; hence, INSERT(S,x) runs in $O(1)$ time. For DELETE-LARGER-HALF(S), we first use a median finding algorithm to find the median (in linear time) and then call a partition algorithm to parition the elements around this median element (smaller elements to the left, larger elements to the right); this can also be done in linear time. Deletion of the partition of larger elements requires constant time, since we simpy reset our index pointer to the median.

Now, how do we prove that m operations run in $O(m)$ time. One way to do this is to use the accountant method. Say we charge 5 dollars per item that we want to put on the array. 1 dollars will go to putting the element in the array. The other 4 dollars will be set in reserve for that element. We pay 1 dollar per element to find the median, 1 dollar per element to paritition; after this, each element has 2 dollars on it. Then we delete half of the elements; but before getting rid of them, we take off the money that these items (to be deleted), 2 dollars per item to be deleted, and spread it out on the remaining items. Now each item has 4 dollars on it again so we are no worse for wear. In this way, we collect $5m$ dollars over m operations so the amortized cost per operation is constant OR, in other words, any sequence of m operations takes $O(m)$ time.

Question 17.4-3

There are three cases, I will only demonstrate of them here since the third one is similar to the case where the table does not contract and the load factor is less than 1/2.

Let the load factor be less than 1/2. First case where the table doesn't do a contraction:

amortized cost per analysis=$c_i + \phi_i - \phi_{i-1}$
=$1 + |2num_i - size_i| - |2num_{i-1} - size_{i-1}|$
and since the table is not contracting we have that $size_i = size_{i-1}$ and $num_i + 1 = num_{i-1}$: =$1 + |2num_i - size_i| - |2num_i - size_i + 2|$

and let subsitute x for $2num_i - size_i$ so we get: =$1 + |x| - |x + 2| \leq 1 + |x| - |x| - 2 = -1$

That we get a negative value is not a problem (since the other case(s) will give us a positive constant and we pick the largest value to be our amortized cost per operation).

Second case where the table does do a contraction. At this point we know that:

- $\frac{num_{i-1}}{size_{i-1}} = 1/3$ since we are doing a contraction.
- $c_i = num_i + 1$ since we do a copy after our 1 deletion.
- $2num_{i-1} = 2(num_i + 1)$

So now we have the following relationship:

$$size_i = (2/3)size_{i-1} = 2num_{i-1} = 2(num_i + 1)$$

And now we can find the amortized cost per operation:
=$c_i + \phi_i + \phi_{i-1}$
=$(num_i + 1) + |2num_i - size_i| - |2num_{i-1} - size_{i-1}|$
=$(num_i + 1) + |2num_i - size_i| - |(2/3)size_{i-1} - size_{i-1}|$
=$(num_i + 1) + |2num_i - size_i| - |(-1/3)size_{i-1}|$
=$num_i + 1 + 2 - (1/3)size_{i-1}$
=$num_i - (1/3)size_{i-1} + 3$
=3

Hence the amortized cost per operation is constant.