CS360 Assignment 4

14.1-2 Show by means of a counterexample that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the density for a rod of length i to be p_i/i and cut off pieces of length i with the maximal density.

Consider the following pricing table for rods of length 4 with computed densities

length	1	2	3	4
p_i	1	20	33	36
p_i/i	1	10	11	9

The greedy strategy would say to initially select a length 3 piece (since it has the highest density) and then the remaining length 1 piece giving a total revenue for the greedy strategy of

$$r_g = p_3 + p_1 = 33 + 1 = 34$$

However, clearly an optimal solution would be to take two pieces of length 2 giving the optimal total revenue of

$$r' = p_2 + p_2 = 20 + 20 = 40$$

Note that even simply selecting the entire rod with no cuts would give a revenue of 36 which is better than the greedy strategy. This problem is very similar to the *knapsack* problem where simply selecting the most valuable item intially may preclude selecting several slightly less valuable items which would have produced a greater total amount, and demonstrates why simply employing a greedy strategy must be used with caution.

14.1-3 Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

We first note that the only place where a cut enters the calculations is in line 6, thus we simply subtract the per cut cost to the profit in the second term of the max() function. However, we also need to stop the loop early since we do not want to incur the cut cost in the case where the entire rod is taken, i.e. no cut is made. Since the solution is generated bottom-up, future subproblems will automatically incorporate the prior cut cost. Hence the new routine would be

```
NEW_CUT_ROD_WITH_CUT_COST(p,n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4     q = -INF
5     for i = 1 to j-1
6        q = max(q, p[i] + r[j-i] - c)
7     r[j] = max(q,p[j])                  // in case there are no cuts
8  return r[n]
```

14-4 Printing neatly.

Step 1 Characterize optimal solution

Define a variable $wc(i,j) = \sum_{k=i}^{j} l_k$ (where $l_k \leq M$ is the length of each word) which stores the total number of characters (not including spaces) for words i through j.

Then define a variable $ec(i,j) = M - (j-i) - wc(i,j)$ as the number of spaces at the end of a line starting with word i and ending with word j, where M is the specified total number of characters on the line, $(j-i)$ is the number of spaces *between* words i through j, and $wc(i,j)$ is the total number of letters in words i through j. Note that $e(i,j)$ may be negative if the total number of characters exceeds the length of the line M.

Then define a cost for the line as

$$lc(i,j) = \begin{cases} \infty & \text{if } ec(i,j) < 0 \\ 0 & \text{if } j = n, ec(i,j) \geq 0 \\ ec(i,j)^3 & \text{otherwise} \end{cases}$$

The first condition handles the case where the words don't fit by preventing them from being the minimum by setting the cost to infinity, the second condition is for the last line where we don't want the extra spaces at the end to count, and the third condition is the cubic penalty defined for words that fit on lines other than the last as specified by the problem description (note that different cost functions could be defined to produce different results).

Note: We only need to consider cases where $i \leq j$ otherwise the words would be out of order.

Step 2 Define *top-down* recursive solution

We now note that the optimal solution for words $1..n$ where the last line starts at word i must contain an optimal solution for words $1..(i-1)$.

Hence we can define the recursion for the solution by defining a minimum cost for n words as $c(n)$ given by the following formula

$$c(n) = \begin{cases} 0 & n = 0 \\ \min_{1 \leq i \leq n}(c(i-1) + lc(i,n)) & n > 0 \end{cases}$$

i.e. the minimum is 0 if we have no words, otherwise it is the minimum of the first $i-1$ words and the cost of the last line containing words i to n.

To show *optimal substructure*, assume optimality occurs when $i = k$ such that $c(n) = c(k-1) + lc(k,n)$ where $c(n)$ is minimum. Now assume that $c(k-1)$ is *not* optimal, therefore we can find another spacing for the first $k-1$ words such that $c'(k-1) < c(k-1)$.

Then $c'(n) = c'(k-1) + lc(k,n) < c(k-1) + lc(k,n) = c(n) \Rightarrow c'(n) < c(n)$. However, this **contradicts** our assumption that $c(n)$ was the minimum, and thus $c(k-1)$ must also be an optimal minimum.

Step 3 Computing *bottom-up* solution

Replacing n by j and starting with the base case $n = 0 \Rightarrow c(0) = 0$ gives

$$c(j) = \begin{cases} 0 & j = 0 \\ \min_{1 \leq i \leq j}(c(i-1) + lc(i,j)) & j > 0 \end{cases}$$

Which is then computed by looping from $j = 1$ to $j = n$ storing the values of $c(j)$ in an array.

Pseudocode for this implementation is

```
NEAT_PRINTING(l,n,M)
1.  Create wc[n,n], ec[n,n], lc[n,n], c[n], and p[n]
2.  // compute word counts wc[n,n]
3.  for i = 1 to n
4.     wc[i,i] = l[i]
5.     for j = i+1 to n
6.        wc[i,j] = wc[i,j-1] + l[j]
7.  // compute spaces ec[i,j]
8.  for i = 1 to n
9.      for j = i to n
10.         ec[i,j] = M - (j - i) - wc[i,j]
11.  // compute line cost lc[i,j]
12.  for i = 1 to n
13.      for j = i to n
14.          if ec[i,j] < 0
15.              lc[i,j] = INF
16.          else if j = n
17.              lc[i,j] = 0
18.          else
19.              lc[i,j] = ec[i,j]^3
20.  // compute c[j]
21.  c[0] = 0
22.  for j = 1 to n
23.      c[j] = INF
24.      for i = 1 to j
25.          if c[j] > c[i-1] + lc[i,j]
26.              c[j] = c[i-1] + lc[i,j]
27.              p[j] = i
```

Note in the above algorithm we include an extra array $p[]$ which keeps track of which word each line breaks at which can be used to print the formatted paragraph. Clearly the run time of this algorithm is dominated by the nested loops which run in $\Theta(n^2)$, e.g. lines 3-6, 8-10, and 12-17 have two nested $\Theta(n)$ loops as does the final computation of $c[n]$ on lines 22-27. There are many simplifications that can be made to improve on this basic algorithm in terms of space usage.

16.1-4 Given a set of activities to schedule among a large number of lecture halls, we wish to schedule all the activities using as few lecture halls as possible. Give a greedy algorithm to solve the problem.

A straightforward solution is to simply use the activity selection algorithm to assign maximal sets of compatible activities to successive halls. In other words,

starting with the set of all activities $S^{(1)}$, let $A^{(1)}$ be the maximum set of compatible activities in $S^{(1)}$ and schedule those in lecture hall 1. Then let $S^{(2)} = S^{(1)} - A^{(1)}$ (i.e the remaining activities) and find $A^{(2)}$ to schedule in lecture hall 2. Continue by finding $A^{(i)}$ for $S^{(1)} = S^{(i-1)} - A^{(i-1)}$ until $S^{(i)} = 0$ (i.e. all activities are scheduled).
This gives an optimal solution since at any step if there would be an activity that could be scheduled in a previous hall, it would have been in the solution to that hall's subproblem. Alternatively, at each step the only remaining activities are ones that occur concurrently and hence need to be scheduled in separate halls.
Since each subproblem runs in $O(n)$ (assuming presorting of the inputs) and there are at most n subproblems (all activities overlap), the algorithm runs in $O(n^2)$.

Extra Credit: HOWEVER THERE IS A BETTER WAY! (A *greedy*, greedy algorithm)
Initially sort <u>all</u> times (both start and finish) in increasing order (with finish times first in the case of ties).
Create an increasing order priority queue of occupied halls and a stack of available halls. Initially push all halls onto the stack.
Going through the list of sorted times, whenever a start time occurs, pop a hall from the stack and put it in the queue using its finish time as the priority key. The activity can be marked with this hall if necessary to produce the final schedule.
Whenever a finish time occurs, dequeue the first element of the queue and push it back on the stack (i.e. the hall has become available). This ensures that halls that have already been used are reused first.
Keeping track of the queue size gives the maximum number of halls needed (which corresponds to the maximum number of simultaneous activities occuring at any one time).
Initial sorting of $2n$ times takes $O(n \lg n)$. The stack push/pop operations take $O(1)$. The queue enqueue/dequeue operations take $O(\lg n)$. So for the $2n$ times we get a total running time of $O(n \lg n) + 2nO(1) + nO(\lg n) + nO(\lg n) = O(n \lg n)$!