CS 161
Autumn 2017

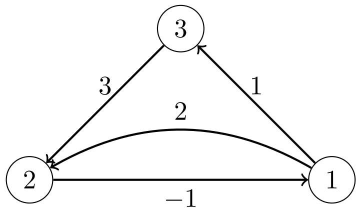
Problem Set 6
Due: Friday November 17, 2017, at 3pm on Gradescope

Exercises

Exercises should be completed **on your own.**

---

1. **(2 pt.)** Consider the graph below:



Step through the Floyd-Warshall algorithm on this graph (using the order suggested by the vertex labels), and write down what your tables $D^{(i)}$ look like for $i = 0, 1, 2, 3$. (You may either code this up or do it by hand).

If it helps, here is the LaTeXcode for one such table:

```
\begin{tabular}{c|c|c|c|}
$D^{(i)}$  & 1 & 2 & 3 \\ \hline
1 & - & - & - \\ \hline
2 & - & - & - \\ \hline
3 & - & - & - \\ \hline
\end{tabular}
```

[**We are expecting: Your filled-in tables. No explanation is required.**]

2. **(3 pt.)** Let A be an array of length n containing real numbers. A *longest increasing subsequence* (LIS) of A is a sequence $0 \le i_1 < i_2 < \dots i_\ell < n$ so that $A[i_1] < A[i_2] < \dots < A[i_\ell]$, so that ℓ is as long as possible. For example, if $A = [6, 3, 2, 5, 6, 4, 8]$, then a LIS is $i_0 = 1, i_1 = 3, i_2 = 4, i_3 = 6$ corresponding to the subsequence $3, 5, 6, 8$. (Notice that a longest increasing subsequence doesn't need to be unique).

   In the following parts, we'll walk through the recipe that we saw in class for coming up with DP algorithms to develop an $O(n^2)$-time algorithm for finding an LIS.

   (a) **(1 pt.) (Identify optimal sub-structure and a recursive relationship).** We'll come up with the sub-problems and recursive relationship for you, although you will have to justify it. Let $D[i]$ be the length of the longest increasing subsequence of $[A[0], \dots, A[i]]$ that ends on $A[i]$. Expain why

   $$D[i] = \max\{D[k] + 1 \,:\, 0 \le k < i, A[k] < A[i]\}.$$

   [We are expecting: A short informal explanation. It is good practice to write a formal proof, but this is not required for credit.]

   (b) **(1 pt.) (Develop a DP algorithm to find the value of the optimal solution)** Use the relationship about to design a dynamic programming algorithm returns the *length* of the longest increasing subsequence. Your algorithm should run in time $O(n^2)$ and should fill in the array D defined above.

   [We are expecting: Pseudocode. No justification is required.]

   (c) **(1 pt.) (Adapt your DP algorithm to return the optimal solution)** Adapt your algorithm above to return an actual LIS, not just its length. Your algorithm should run in time $O(n^2)$.

   [We are expecting: Pseudocode and a short English explanation.]

   Note: Actually, there is an $O(n \log(n))$-time algorithm to find an LIS, which is faster than the DP solution in this exercise!

Problems

You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

1. **(10 pt.)** Consider the following problem:

   Let S be a set of positive integers, and let n be a non-negative integer. Find the minimal number of elements of S needed to write n as a sum of elements of S (possibly with repetitions).

   For example, if $S = \{1, 4, 7\}$ and $n = 10$, then we can write $n = 1 + 1 + 1 + 7$ and that uses four elements of S. The solution to the problem would be "4."

   Your friend has devised a divide-and-conquer algorithm for to find the minimum size. Their pseudocode is below.

```
def minimumElements(n, S):
     if n == 0:
         return 0
     if n < min(S):
         return None
     candidates = []
     for s in S:
         cand = minimumElements( n-s, S )
         if cand is not None:
              candidates.append( cand + 1 )
     return min(candidates)
```

   (a) **(3 pt.)** Prove that your friend's algorithm is correct.

   [We are expecting: A proof by induction. Make sure to state your inductive hypothesis, base case, inductive step, and conclusion.]

   (b) **(1 pt.)** Argue that for $S = \{1, 2\}$, your friend's algorithm has exponential running time. (That is, running time of the form $2^{\Omega(n)}$).

   [We are expecting: An short but convincing justification, which involves the recurrence relation that the running time of your friend's algorithm satisfies when $S = \{1, 2\}$.]

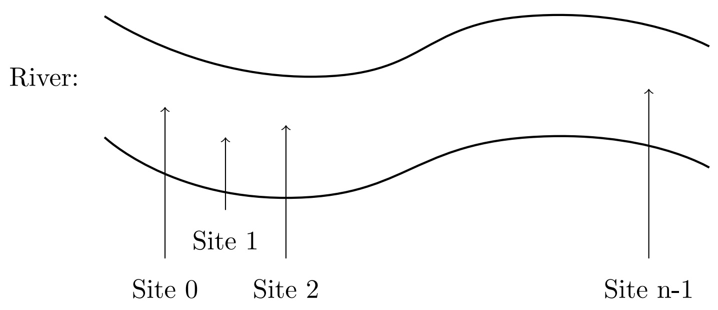
   (c) **(3 pt.)** Turn your friend's algorithm into a top-down dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

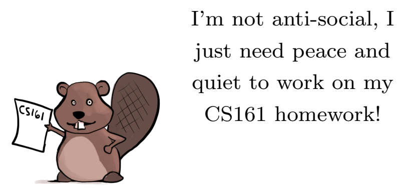
   [We are expecting: Pseudocode, and a short English description of the idea of your algorithm. You should also informally justify the running time.]

   (d) **(3 pt.)** Turn your friend's algorithm into a bottom-up dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

   [We are expecting: Pseudocode, and a short English description of the idea of your algorithm. You should also informally justify the running time.]

2. **(6 pt.)** You arrive at a river where there is a population of extremely antisocial beavers. There are n sites along the river that are appropriate for beaver dams, arranged linearly.




CS161
I'm not anti-social, I just need peace and quiet to work on my CS161 homework!

Each site has a quality, which is a real number. The higher the quality, the better the dam. However, because the beavers are antisocial, no two adjacent sites can be developed into dams. You want to find a way to assign beavers to sites[1] in order to maximize the total quality of dams built: that is, if $Q[i]$ is the quality of site i, you want to maximize

$$X = \sum_{i=0}^{n-1} \mathbf{1}\{\text{there is a dam at site } i\} \cdot Q[i],$$

subject to never placing dams at adjacent sites i and $i+1$.

For example, if the qualities Q were given by $Q = [21, 4, 6, 20, 2, 5]$, then the optimal solution would be for three beavers to build dams, at sites $0, 3$ and 5, with a total of $21 + 20 + 5 = 46$ quality.

Design a dynamic programming algorithm to find the optimal locations to build dams, in the sense that it maximizes the quantity X. Your algorithm should take Q as an input, output a list of locations to build dams, and should run in time $O(n)$ and use $O(n)$ space.

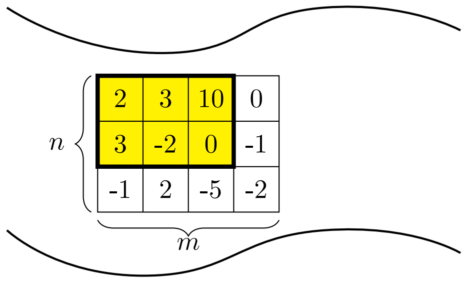
[We are expecting: Pseudocode, and an English description of the idea of your algorithm. (In particular, what are the sub-problems you are using?) You should also give an informal argument that your algorithm is correct and has the desired running time and space.]

[1] Any beavers who don't get sites at this river will go to another river and live happily ever after.

3. **(7 pt.)** Once a beaver has an n-foot by m-foot site to call their own, they have to build the most excellent dam they can on that site. Billy the Beaver wants to build a **rectangular** dam on his site. The $n \times m$ site is divided into $n \cdot m$ one-foot by one-foot squares. As above, each square has a quality, which is a real number. Notice that the qualities might be negative (that is, it costs more to build on that square than the utility gained).

Billy the Beaver's goal is to find the rectangular dam that maximizes the total quality; he doesn't care about the size, just that it is a rectangle. If the best quality that Billy can achieve is negative, then he will choose not to build a dam at all.

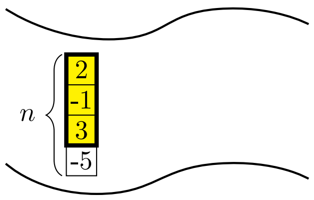
For example, if $n = 3$ and $m = 4$ and the qualities on the site were as depicted below, Billy would choose to build a 2×3 dam with total utility 16, which is highlighted below.



2	3	10	0
3	-2	0	-1
-1	2	-5	-2

In the following parts, you'll help Billy the Beaver achieve his goal in a few different settings.

(a) **(3 pt.)** Suppose that $m = 1$. That is, the site is a $n \times 1$ strip, and the qualities can be represented as an array B of length n.



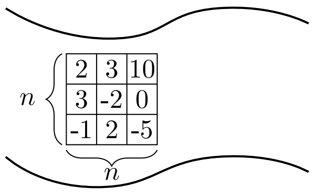
Design an algorithm that takes B as input and returns two indices $a, b \in \{0, \ldots, n-1\}$ so that $[B[a], B[a+1], \ldots, B[b]]$ is the optimal dam site. That is, a and b are numbers so that

$$\sum_{s=a}^{b} B[s]$$

is as large as possible. Your algorithm should run in time $O(n)$.

[We are expecting: Pseudocode, as well as an English explanation of what your algorithm does. Also, an informal justification of the running time.]

(b) **(3 pt.)** Now suppose that $m = n$. That is, the site is square. Let A be the $n \times n$ array of qualities.



In order to be thorough, Billy the Beaver wants to compute the score he will get for **every** possible dam he could build. That is, he wants to make an $n \times n \times n \times n$ array D so that for all $x \le i$ and all $y \le j$,

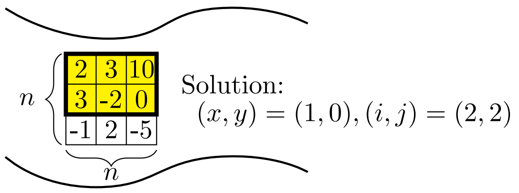
$$D[x][y][i][j] = \sum_{s=x}^{i} \sum_{t=y}^{j} A[s][t].$$

The interpretation of the above is the total quality of the rectangle with lower-left corner (x, y) and upper-right corner (i, j). For other tuples (x, y, i, j) that don't satisfy $x \le i, y \le j$, it doesn't matter what D has in it.

Give an algorithm that takes A as input, and outputs an array D, in time $O(n^4)$.

[We are expecting: Pseudocode, as well as an English explanation of what your algorithm does. Also, an informal justification of the running time.]

(c) **(1 pt.)** Use your algorithm above to design an algorithm for Billy the Beaver to find the optimal rectangular dam in an $n \times n$ grid in time $O(n^4)$. Your algorithm should take A as an input and should output x, y, i, j so that the rectangle $\{x, \ldots, i\} \times \{y, \ldots, j\}$ corresponds to an optimal dam.



[We are expecting: Pseudocode and a brief explanation.]

(d) **(1 bonus pt.)** Give an algorithm that solves part (c) (finding the optimal rectangle in an $n \times n$ grid) in time $O(n^3)$.

[We are expecting: Pseudocode and an English explanation of the main idea. What are the subproblems that you are using?]

Optional problem

This problem is completely optional. It's not worth any points (not even bonus points) and we won't grade it. However, it might be fun!

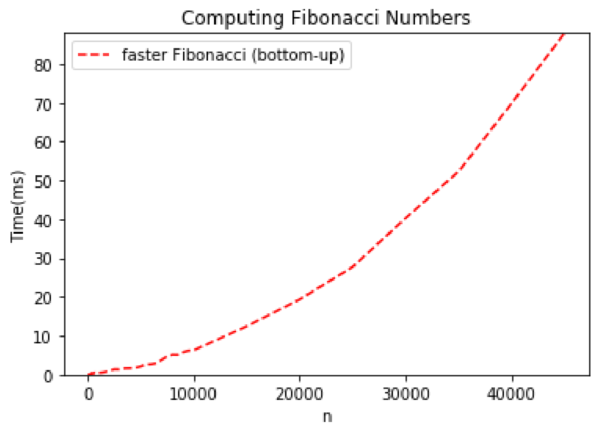
---

4. **(0 pt.)** In this optional problem we'll see how to compute Fibonacci numbers *even faster!*

   (a) In class, we saw the following bottom-up algorithm to compute Fibonacci numbers:

   ```
   def fasterFibonacci(n):
       F = [1 for i in range(n+1)]
       for i in range(2,n+1):
           F[i] = F[i-1] + F[i-2]
       return F[n]
   ```

   This was much faster than the naive divide-and-conquer approach. But what actually is the runtime of this? Your friend says it's $O(n)$: we have n iterations of the loop and we're just adding two numbers inside the loop. But when we look at the running times for really large n, it looks like this:



   That doesn't look linear! Your friend's argument seems pretty reasonable, at least by the standards we've been using in this class. So what's going on? (Note: We haven't really been formal enough in this class to give you the tools to argue this formally. The point of this optional problem is just to get you to think a bit.) **[HINT: How large is $F[n]$? How many bits does it take to represent $F[n]$ in binary? How much time does it take to add $F[n-2] + F[n-1]$?]**

   (b) Let M be the matrix $M = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$. Argue that

   $$M^n \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F[n] \\ F[n-1] \end{pmatrix}.$$

   (c) Come up with an algorithm that uses $O(\log(n))$ multiplications (of unbounded size) to compute $F[n]$, when n is a power of 2.

   (d) If you take into account how big the numbers you are multiplying are in the above, how long would your algorithm from the previous part take? What if you use a fast multiplication algorithm like Karatsuba? What if you use a multiplication algorithm which takes time $O(n \log(n) \log\log(n))$ to multiply n-bit numbers? (This latter thing exists, it's called the *Schonhage-Strassen* algorithm).