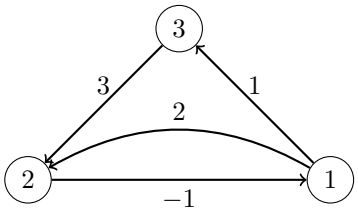
Exercises

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.]

SOLUTION:

$D^{(0)}$	1	2	3
1	0	2	1
2	-1	0	∞
3	∞	3	0

$D^{(1)}$	1	2	3
1	0	2	1
2	-1	0	0
3	∞	3	0

$D^{(2)}$	1	2	3
1	0	2	1
2	-1	0	0
3	2	3	0

$D^{(3)}$	1	2	3
1	0	2	1
2	-1	0	0
3	2	3	0

2. **(3 pt.)** Let A be an array of length n containing real numbers. A *longest increasing subsequence* (LIS) of A is a sequence $0 \le i_1 < i_2 < \ldots i_\ell < n$ so that $A[i_1] < A[i_2] < \cdots < A[i_\ell]$, 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 $[A[0], \ldots, A[i]]$ that ends on $A[i]$. Explain why

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

   [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!

   SOLUTION:

   (a) We will first show that $D[i] \le \max\{D[k] + 1 \,:\, 0 \le k < i, A[k] < A[i]\}$, for all $i \ge 1$.
   Since $D[i]$ is the length of a LIS that ends on $A[i]$, let $i_1 < \ldots < i_{D[i]}$ be that subsequence; notice that $i_{D[i]} = i$, since the sequence ends on $A[i]$. Let $\tilde{k} = i_{D[i]-1}$, and consider the sequence $i_1 < i_2 < \ldots, i_{D[i]-1}$ of length $D[i] - 1$ ending at $\tilde{k}$. This is an increasing subsequence ending at $\tilde{k}$. Thus, $D[i] - 1$ is at most $D[\tilde{k}]$, since $D[\tilde{k}]$ is the length of the *longest* such sequence, and this one had length $D[i] - 1$.
   Since $\tilde{k} = i_{D[i]-1} < i_{D[i]} = i$ and $A[\tilde{k}] < A[i]$ (since the original sequence was increasing) this only gets larger when we maximize over all such k, so Therefore

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

   Now we go the other direction and show that $D[i] \ge \max\{D[k] + 1 \,:\, 0 \le k < i, A[k] < A[i]\}$. Suppose that k is the index which maximizes the right hand side, and suppose that

$$i_1 < \ldots < i_{D[k]}$$

   is the corresponding sequence. Then the sequence

$$i_1 < \ldots < i_{D[k]} < i$$

   is an increasing subsequence ending at i, with length $D[k]+1$. The length of the longest one is at least as long as this, so

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

(b) Our algorithm is as follows:

```
def LIS(array A of length n):
    Initialize an array D of length n full of 1's
    for i = 1,...,n-1:
        for k in range(i):
            if A[k] < A[i]:
                D[i] = max( D[i], D[k] + 1 )
    return max(D)
```

(c) Our algorithm is as follows. We modify our answer from part (b) to additionally store an array P. If $P[i] = k$, the interpretation is that the LIS ending at i had its second-to-last entry at k. After we are done computing the length of the LIS, we can follow the pointers in P backwards to recover the LIS itself.

```
def LIS(array A of length n):
    Initialize an array D of length n full of 1's
    Initialize an array P of length n full of None's
    for i = 1,...,n-1:
        for k in range(i):
            if A[k] < A[i]:
                D[i] = max( D[i], D[k] + 1 )
                if D[i] was updated:
                      P[i] = k
    Find k so that D[k] is maximized
    ret = [k]
    current = k
    while current != None:
        current = P[current]
        ret.append(current)
    reverse ret
    return ret
```

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.]

SOLUTION:

(a)
- **Inductive hypothesis:** For all $k \leq n$, `minimizeElements(k,S)` returns a minimal way to make k out of elements in S, or returns `none` if it can't be done.
- **Base case:** When $0 < n < min(S)$, for all $k \leq n$ there are no ways to make k out of elements in S, and the algorithm correctly returns `None`. If $n = 0$, then the empty set is a way to make n from S, and this is what the algorithm returns.
- **Inductive step:** Suppose that the inductive hypothesis is true for $n-1$, with $n \geq min(S)$.
  First, suppose that there is a way to make n out of S, using k numbers. Let A (a multi-set of elements from S) be some way of doing this. Since A is non-empty, $s \in A$, and $A \setminus \{s\}$ is a minimal way to make $n - s$ from S, using $k-1$ numbers. (To see that it is minimal, notice that if there were a way to make $n-s$ with fewer than $|A|-1$ elements of S, then adding s to that would give a way to make n out of fewer than k items of S, contradicting the optimality of A). Since $n - s < n$, the inductive hypothesis implies that running `minimizeElements(n-s,S)` will return a multi-set B of length $k-1$. Then the algorithm adds $B + [s]$, which has length k, to the list of candidates. There can't be anything in the list shorter than k (since k is optimal), so the algorithm returns a solution of length k. Thus, if there is any way to make n out of S, `minimizeElements(n,S)` will return a minimal way.
  On the other hand, suppose there is no way to make n out of S. Then for all $s \in S$, there is no way to make $n - s$ from S (or else adding s would give a way to make n from S after all) and so by induction all the recursive calls return `None`, and the algorithm returns `None`.
- **Conclusion:** By induction the inductive hypothesis holds for all n, which immediately implies that the algorithm is correct.

(b) When $S = \{1, 2\}$, then the algorithm running on n makes two recursive calls, one to $n-1$ and one to $n-2$. The running time of this algorithm satisfies the recurrence

$$T(n) = T(n-1) + T(n-2) + O(1).$$

Thus, the running time of this algorithm grows at least as fast as the Fibonacci numbers. We saw in class that these grow exponentially quickly.

(c) To make a top-down DP, we add an array D to keep track of previous solutions. We set $D[k]$ to be the minimal number of elements of S needed to make k.

```
Initialize a global array D of length n to all -1's.
D[0] = 0
Set D[k] = None for all 0 < k < min(S).
def minimumElements(n, S):
     if n < 0:
         return None
     if D[n] != -1:
         return D[n]
     candidates = []
     for s in S:
         cand = minimumElements( n-s, S )
         if cand is not None:
              candidates.append( cand + 1 )
     D[n] = min(candidates)
     return D[n]
```

The running time of this algorithm is $O(n|S|)$. Each $k \leq n$ gets run at most once, and for each one, we have work $O(|S|)$, to make the recursive calls and aggregate the results.

(d) To make a bottom-up DP, we'll use the same interpretation of the array as before, but fill it in in order.

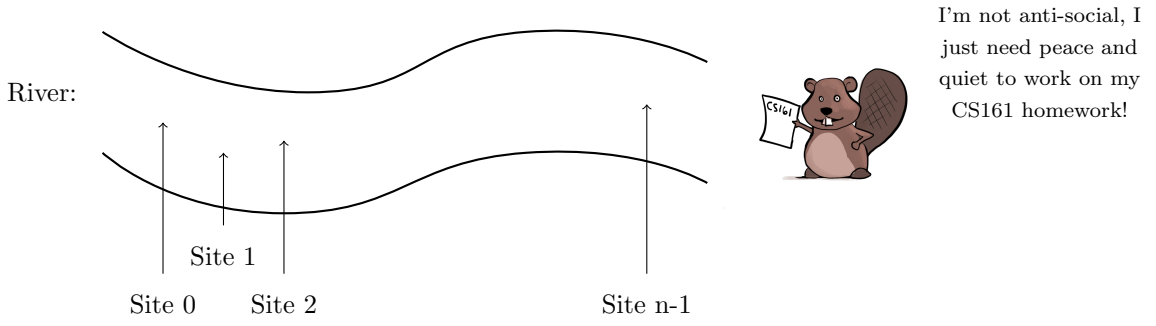
```
Initialize a global array D of length n to all -1's.
D[0] = 0
Set D[k] = None for all 0 < k < min(S).

for i = min(S), ..., n:
     candidates = []
     for s in S:
          if i < s:
              continue
          cand = D[i-s]
          if cand is not None:
                candidates.append( cand + 1 )
     D[i] = min(candidates)

return D[n]
```

Again the running time is $O(n|S|)$, since the outer for loop has n iterations, and the inner for loop has $|S|$ iterations. Inside the inner loop we do $O(1)$ work.

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.



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.

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

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.]

SOLUTION: Our subproblems will be:

$A[i]$ = the maximum quality the beavers could get on the array $[Q[0], Q[1], \ldots, Q[i]]$.

This obeys the recursive relationship

$$A[i] = \max\{A[i-1], Q[i] + A[i-2]\}.$$

This is because either the beavers can choose to build a dam at site i, gaining $Q[i]$ utility plus whatever they were able to get up to $A[i-2]$; or the beavers can choose not to build at i, in which case the optimal solution for i is the same as the optimal solution for $i-1$. Then the best solution will be the maximum of the two.

This leads to the following pseudocode.

```
def buildDams(Q):
     Initialize an array A of length n
     Initialize an array P of length n
     if Q[0] >= 0:
         A[0] = Q[0]
         P[0] = 1
     else:
        A[0] = 0
        P[0] = 0
     if Q[0] > Q[1]:
          A[1] = Q[0]
          P[1] = 0
     else:
          A[1] = Q[1]
          P[1] = 1
     for i in range(2,n):
          if A[i-1] > Q[i] + A[i-2]:
                A[i] = A[i-1]
                P[i] = 0
          else:
                A[i] = Q[i] + A[i-2]
                P[i] = 1
     # Now work backwards through P
     damSet = []
     current = n-1
     while current >= 0:
          else if P[current] == 1:
               damSet.append(current)
               current = current - 2
          else:
               current = current - 1
     return damSet
```

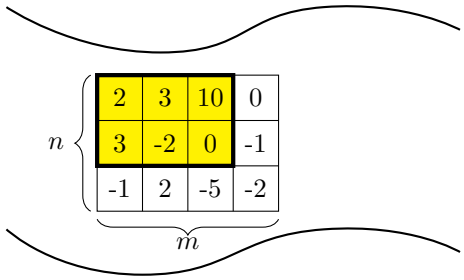
In addition to A, the pseudocode above keeps a table P which records whether or not we would have built a dam on site i given the option. At the end of the algorithm, we work backwards through P to decide where to build the dams.

The algorithm is correct (informally) because the recursive relationship identified above is correct. The running time is $O(n)$ because we take one pass through the array to fill it in, and one more pass backwards to recover the solution.

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).

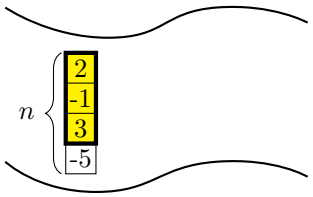
   Billy the Beaver's goal is to find the rectangular dam that maximizes the total quality; he doesn't care about the size, just that it is a rectangle. If the best quality that Billy can achieve is negative, then he will choose not to build a dam at all.

   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.



   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 $O(n)$.

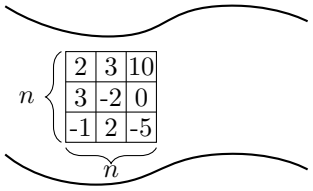
   [We are expecting: Pseudocode, as well as an English explanation of what your algorithm does. Also, an informal justification of the running time.]

   SOLUTION: We'll use a dynamic programming algorithm which has the sub-problems: *Given the array $B[:i]$, so that* either *you must use the final square, or else you can't build a dam at all.*
   More precisely, we will keep an array D, which stores tuples $D[i] = (start, score)$. What this means is that the solution to the subproblem given by $B[:i+1]$ if we must use the i'th square is the range $[B[start], B[start+1], \ldots, B[i]]$. If $start = i$ and $score = 0$, then this means that the best solution is to build the empty dam $B[i:i]$ which gives utility zero.

Our recursive relationship is as follows. Suppose that $D[i] = (start, score)$. If $score + B[i] > 0$, then the best dam we could build if we *must* include $B[i]$, is $B[start : i + 1]$. To see this, suppose that there were a better dam $B[start' : i + 1]$; but then $B[start' : i]$ would have been a better solution to the previous sub-problem.
On the other hand, if $score + B[i] < 0$, then we would never want to build a dam on $B[: i+1]$ if we have to include $B[i]$. This is because by the previous argument, $score+B[i]$ is the best score that we could get, and it is negative. Thus, we are better off just not including $B[i]$, and we set $D[i] = (i, 0)$.
At the end of the day, we look through the array for the maximum score we could have attained, and return the corresponding dam. The pseudocode is given below.

```
def damn1DDam(A):
    n = len(A)
    D = [None for i in range(n)]
    if A[0] > 0:
        D[0] = (-1,A[0]) # this means that in the range (-1,0], we can get score A[0]
    else:
        D[0] = (0, 0) # this means that in the range (0,0] (aka emptyset) we can get score
    for i in range(1,n):
        start, score = D[i-1]
        if score + A[i] > 0:
            D[i] = (start, score + A[i])
        else:
            D[i] = (i,0) # this means that we're best off just not including i.
    bestScore = 0
    bestStart = None
    bestStop = None
    for i in range(n):
        start, score = D[i]
        if score > bestScore:
            bestScore = score
            bestStart = start + 1
            bestStop = i
    if bestScore > 0:
        return bestStart, bestStop, bestScore
    else:
        return "I don't want to build a dam."
```

(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 $x \leq i$ and all $y \leq j$,

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

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 $x \leq i, y \leq j$, 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.]

SOLUTION: The algorithm steps through all possible rectangles, with the smallest ones first, and fills in values. When $x < i$ (aka, the number of rows in the rectangle is at least two), in order to compute $D[x][y][i][j]$, the algorithm breaks this up into two parts, $D[x][y][i-1][j]$ (that is, all but the top row) and $D[i][y][i][j]$ (the top row). If there is only one row, but more than one column, the algorithm breaks it up into the right-most square and everything else; and if there is only one row and only one column, then the algorithm just consults A.

```
defDef damnDam(A):
    n = len(A)
    D = [[[[ None for i in range(n) ] for j in range(n) ]
                          for x in range(n) ] for y in range(n) ]
    for s in range(n):
        for t in range(n):
            for x in range(n-s):
                i = x + s
                for y in range(n-t):
                    j = y + t
                    if i == x and j == y:
                        D[x][y][i][j] = A[i][j]
                    elif i == x and y < j:
                        D[x][y][i][j] = D[x][y][i][j-1] + A[i][j]
                    else: # then x < i (we should never have i < x)
                        D[x][y][i][j] = D[x][y][i-1][j] + D[i][y][i][j]
    return D
```

To see that this takes time $O(n^4)$, note that there are four for-loops each over n things. Inside the for-loops, we only do $O(1)$ work. (**Note:** The trick above is that the for-loops are ordered so that we do the smallest rectangles first. It would not work in an arbitrary order).

Another way to do this problem which might be easier to think about is to explicitly start with the long skinny rectangles:

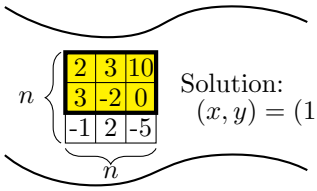
```
def problem3b(A):
    n = len(A)
    D = [[[[ None for i in xrange(n) ] for j in xrange(n) ]
        for x in xrange(n) ] for y in xrange(n)]

    # First, get all possible 1D rectangles
    for x in xrange(n):
        for y in xrange(n):
            D[x][y][x][y] = A[x][y]
            for j in xrange(y + 1, n):
                D[x][y][x][j] = A[x][j] + D[x][y][x][j - 1]
            for i in xrange(x + 1, n):
                D[x][y][i][y] = A[i][y] + D[x][y][i - 1][y]

    # Now, use the 1D rectangles to find the areas of the 2D rectangles
    for x in xrange(n):
        for y in xrange(n):
            for i in xrange(x + 1, n):
                for j in xrange(y + 1, n):
                    D[x][y][i][j] = D[x][y][i - 1][j - 1]
                                  + D[i][y][i][j] + D[x][j][i][j] - D[i][j][i][j]

    return D
```

(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.**]

SOLUTION: Since we can find all of the possible scores in time $O(n^4)$, we can loop through them in time $O(n^4)$ and return the best one:

```
def optimalDamnDam(A):
    D = damnDam(A)  \\ time O(n^4)
    find (x,y,i,j) that maximizes D[x][y][i][j] in time O(n^4) by searching all of D
    if that value isn't negative, return (x,y,i,j)
    otherwise, return ``I don't want to build a dam."
```

(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?**]

SOLUTION:
First, we compute an $n \times n \times n$ table R so that $R[x,i,y] = \sum_{s=x}^{i} A[y,s]$. That is, this is the row sum of the y'th row between x and i. We can do this in time $O(n^3)$ exactly as we did in part (b).
Now to solve this problem, we use the same idea as in part (a), except we use the table R to help. The sub-problems are indexed by (x,i,j). We keep another table D so that $D[x,i,j] = (starty, score)$, where starty is the y so that a rectangle with upper-right corner (i,j) and lower-left corner (x,y) has the maximal score $score$. We allow $y = j$, which means that we would prefer to not build at all then to build with corner (i,j).
To compute $D[x,i,j]$ using earlier entries in the table D:

- If $j \geq 1$, then suppose $D[x,i,j-1] = (starty, score)$. Then if $score + R[x,i,j] > 0$, we set $D[x,i,j] = (starty, score + R[x,i,j])$. On the other hand, if $score + R[x,i,j] < 0$, then we would never want to use this range (x,i) if we have to use the j'th row, so we set $D[x,i,j] = (j,0)$.
- If $j = 0$, then we just set $D[x,i,0] = (0,0)$ if $R[x,i,0] < 0$ and $D[x,i,0] = (-1, R[x,i,0])$ otherwise. (This is the same intuition as in part (a).

We write this in pseudocode as follows:

```
def damnDam2(A):
    n = len(A)
    In time O(n^3), initialize R to be an n-by-n-by-n array so that
       R[x][i][j] = sum( [ A[y][s] for s in range(x,i+1) ] )
       (we can do this just like we did part (b) in time O(n^4))
    D = [ [ [ None for i in range(n) ] for j in range(n) ] for k in range(n) ]
    for x in range(n):
        for i in range(x,n):
            if R[x][i][0] < 0:
                D[x][i][0] = (0,0)
            else:
                D[x][i][0] = (-1, R[x][i][0])
    for j in range(1,n):
        for x in range(n):
            for i in range(x,n):
                starty, score = D[x][i][j-1]
                newscore = score + R[x][i][j]
                if newscore > 0:
                    D[x][i][j] = (starty, newscore)
                else:
                    D[x][i][j] = (j,0)

    best = (None, None, None, None)
    bestScore = 0
    for x in range(n):
        for i in range(x,n):
            for j in range(n):
                starty, score = D[x][i][j]
                if score > bestScore:
                    bestScore = score
                    best = (x,starty+1,i,j)
    return best, bestScore
```

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!

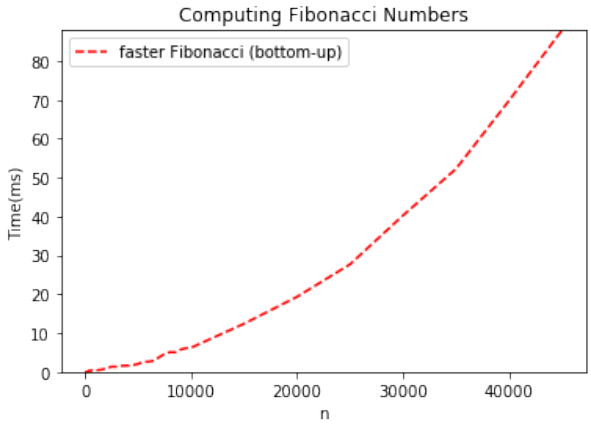
---

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).