Practice Set 9: solutions

Question 1

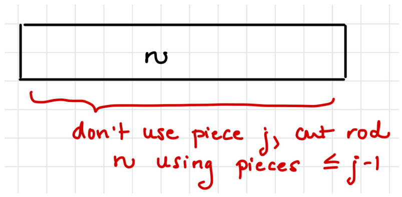
The dynamic programming solution is based on finding a recursive relationship between the original problem and a subproblem. One way to identify this relationship is to examine particular example, say $n = 7$. In the example below we have listed the number of possible ways to cut a rod of length 7. Note that the possibilities are listed in a systematic way, from the order of the cuts with the smallest pieces to the cuts with the largest pieces.

$$(1,1,1,1,1,1,1),(2,1,1,1,1,1),(2,2,1,1,1),(2,2,2,1)$$

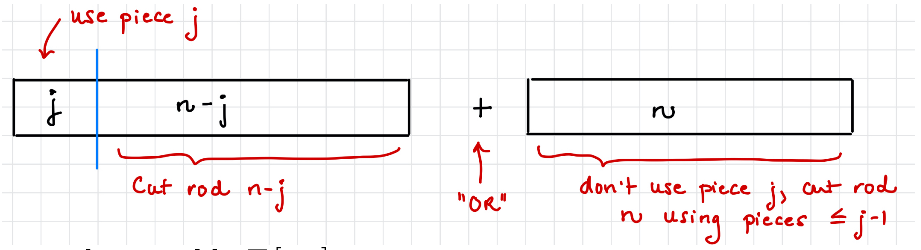
$$(3,1,1,1,1),(3,2,1,1),(3,2,2),(3,3,1),(4,1,1,1),(4,2,1),(4,3),(5,1,1),(5,2),(6,1),(7)$$

This ordering of the possibilities gives us a clue for how to describe the recurrence relationship. Given a rod of length n the number of possible ways to cut the rod can be recursively defined by the maximum size of the first piece. For example, if the first piece has maximum size j, then the number of ways of cutting the rod of size n falls into two cases:

- **Case 1:** If $j > n$, then we can't use piece j and instead we consider cutting the rod of length n using pieces of size at msot $j - 1$.



- **Case 2:** If $j \leq n$ the we can either cut a piece of size j or not:

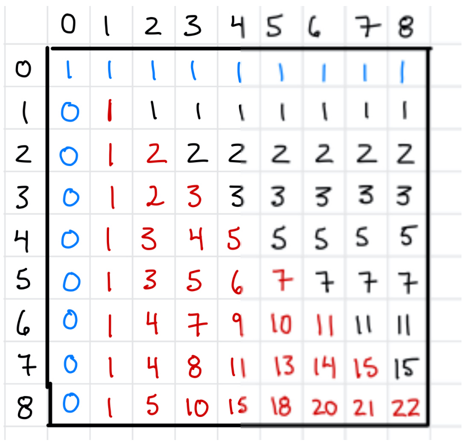


The results of the subproblems will be stored in a table $T[i, j]$:

Define the table $T[0 \ldots n, 0 \ldots n]$:

- $T[i, j]$ represents the number of different ways of cutting a rod of length i using pieces of size at most j.
- If $i = 0$ then the rod has length 0 which actually means that we have cut the rod perfectly and have no more pieces to cut. Therefore $T[0, j] = 1$ for any $j \geq 0$.
- f $j = 0$, then the maximum piece size is 0, which is impossible. Therefore $T[i, 0] = 0$ for any $i > 0$.
- If j is not too big, in other words, $j \leq i$, then we consider the possibilities of using j or not using j as the first piece:
$$T[i, j] = T[i - j, j] + T[i, j - 1]$$
- If $j > i$, then the only way to cut the rod using pieces of size at most j is to actually count the ways using pieces of size at most $j - 1$:
$$T[i, j] = T[i, j - 1]$$
- The final entry $T[n, n]$ is the number of ways to cut the rod of length n using pieces of any size ($\leq n$).

In the figure below, we show an example table for $n = 8$. Note that the tables are drawn with index i as the row and index j as the column (as with matrices).



	0	1	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3	3
4	0	1	3	4	5	5	5	5	5
5	0	1	3	5	6	7	7	7	7
6	0	1	4	7	9	10	11	11	11
7	0	1	4	8	11	13	14	15	15
8	0	1	5	10	15	18	20	21	22

The recursive relationship above shows that the entries of the table $T[i,j]$ reference entries $T[i-j,j]$ and $T[i,j-1]$. Therefore if we fill in the table row by row from left to right then all entries that we reference will already be completed. The algorithm is represented below:

CountCuts(n)

Step 1: Initialize the first row of the table: for $j=0$ to n: $T[0,j]=1$
Initialize the first column of the table: for $i=1$ to n: $T[i,0]=0$

Step 2: Loop through the table row by row from left to right filling in the entires using the above recursive relationship:

```
for $i = 1$ to $n$
    for $j = 1$ to $n$
        if $j \leq i$
            $T[i,j] = T[i-j,j] + T[i,j-1]$
        else $T[i,j] = T[i,j-1]$
Return $T[n,n]$
```

Runtime: The algorithm takes constant time for each cell entry of the table, and therefore runs in time $\Theta(n^2)$.

Question 2

- If there are some piece-sizes that are not possible, we can simply update the input so that the value of those pieces is 0. The input array is again $p[1,\ldots,n]$ where $p[i]$ is set to 0 for any pieces that are not in the original input list. The runtime of the algorithm is still $\Theta(n^2)$
- Consider the altitudes of the cities on the east bank. Let the altitudes of these cities be stored in array $A[1,\ldots,n]$ from north to south. Repeat for the cities on the west bank, using array $B[1,\ldots,m]$. Connecting bridges from one side to the other side is like matching up identical altitudes from $A[]$ and $B[]$. Therefore we can use the longest common subsequence problem using input A and B to determine the maximum number of bridges that can be built.

Question 3

For any position (i,j) in the table, the next move has at most three options: you can drive to either $(i+1,j)$ or $(i,j+1)$ or $(i+1,j+1)$ (there are boundary conditions at the bottom row and last column). For a minimum-toll route, we take the minimum of these three possible next moves, and add the toll at our current location.

Define the table $T[0\ldots n, 0\ldots n]$:

The dynamic programming solutions uses a table $T[i,j]$ for $1 \leq i,j \leq n$ where entry $T[i,j]$ represents the minimum total toll over all routes from (and including) location (i,j) to location (n,n).

Dynamic Programming Solution::

```
1. Initialize the table along the bottom row, since each of those positions only have one route to $(n,n)$:
        Set $T[n,n] = t(n,n)$
        For $j = n-1$ to 1 set $T[n,j] = T[n,j+1] + t(n,j)$
2. Initialize the last column:
        For $i = n-1$ to 1 set $T[i,n] = T[i+1,n] + t(i,n)$
3. Now fill in the remaining entries from bottom row to top, right to left:
        For $i = n-1$ to 1
            For $j = n-1$ to 1
                Set $T[i,j] = t(i,j) + min\{T[i+1,j], T[i,j+1], T[i+1,j+1]\}$
The final answer is stored in $T[1,1]$.
```

Runtime: The algorithm fills in a table of size $\Theta(n^2)$ and does a constant amount of work/entry, and therefore runs in time $\Theta(n^2)$.

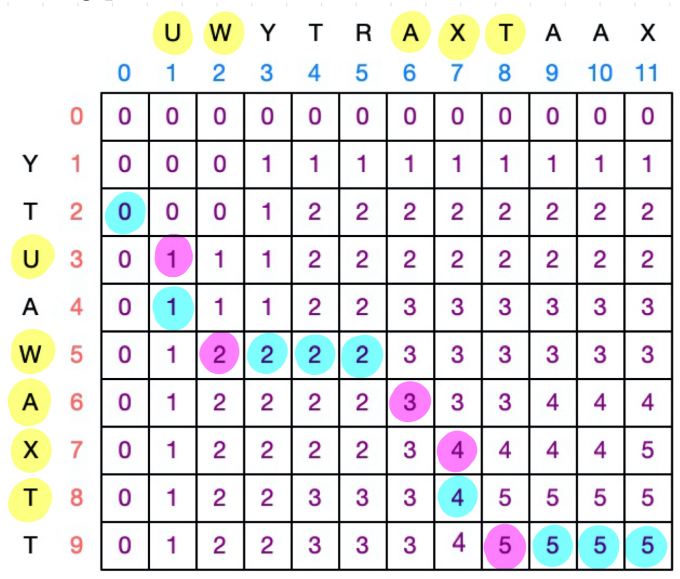
Reconstruction the solution:

In order to output the least-expensive route, we trace through the table from $(1,1)$ to (n,n) by following the option that represents the minimum of the three possible roads we can take next.

```
Initialize $i = 1, j = 1$.
While $(i,j) \neq (n,n)$
    -Output $(i,j)$
    -Find the minimum value of $T$ over each of the possible next steps: $(i+1,j)$, $(i,j+1)$, $(i+1,j+1)$.
     Note that if we are in the last row or column, there may be only one possible next step, in which
     case that is the minimum value.
    -Update $(i,j)$ to the coordinates that represent the minimum of $T$.
```

Question 4

Substring problem:



			U	W	Y	T	R	A	X	T	A	A	X
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
Y	1	0	0	0	1	1	1	1	1	1	1	1	1
T	2	0	0	0	1	2	2	2	2	2	2	2	2
U	3	0	1	1	1	2	2	2	2	2	2	2	2
A	4	0	1	1	1	2	2	3	3	3	3	3	3
W	5	0	1	2	2	2	2	3	3	3	3	3	3
A	6	0	1	2	2	2	2	3	3	3	4	4	4
X	7	0	1	2	2	2	2	3	4	4	4	4	5
T	8	0	1	2	2	3	3	3	4	5	5	5	5
T	9	0	1	2	2	3	3	3	4	5	5	5	5

Rod-cutting problem:

From the table below, we can see that the maximum profit is 18. The size of the first cut should be $s[8] = 4$. The next cut size is $s[4] = 4$. Therefore we simply make two pieces of size 4 to maximize the profit.

	0	1	2	3	4	5	6	7	8
r:	0	1	3	4	9	12	13	15	18

	0	1	2	3	4	5	6	7	8
s:		1	2	2	4	5	5	5	4

$p_1 = 1$	$p_5 = 12$
$p_2 = 3$	$p_6 = 12$
$p_3 = 3$	$p_7 = 14$
$p_4 = 9$	$p_{18} = 17$

$r[2] = \max\{p_1 + 1, p_2 + 0\}$

$r[3] = \max\{p_1 + 3, p_2 + 1, p_3 + 0\}$

$r[4] = \max\{p_1 + 4, p_2 + 3, p_3 + 1, p_4 + 0\}$

$r[5] = \max\{p_1 + 9, p_2 + 4, p_3 + 3, p_4 + 1, p_5 + 0\}$

$r[6] = \max\{p_1 + 12, p_2 + 9, p_3 + 4, p_4 + 3, p_5 + 1, p_6 + 0\}$

$r[7] = \max\{p_1 + 13, p_2 + 12, p_3 + 9, p_4 + 4, p_5 + 3, p_6 + 1, p_7 + 0\}$

$r[8] = \max\{p_1 + 15, p_2 + 13, p_3 + 12, p_4 + 9, p_5 + 4, p_6 + 3, p_7 + 1, p_8 + 0\}$

Question 5

Suppose $B(n)$ represents the balance in her bank account at year n. Note that $B(1) = 100$ and $B(2) = 500$. For $n \geq 2$, the amount in her bank account depends on the previous two years:

$$B(n) = 1.5B(n-1) - B(n-2)/2, \text{for} n \geq 2$$

Recursive solution:

A recursive solutionsolves for $B(n)$ by simply making recursive calls to $B(n-1)$ and $B(n-2)$:

```
FindBalance(n):
    If $n = 1$
        return 100
    else if $n = 2$
        return 600
    else
        return FindBalance(n-1)*1.5 - FindBalance(n-2)/2
```

The runtime recurrence for this equation is $T(n) = T(n-1) + T(n-2)$. This is a very famous recurrence, it is the recurrence for the *Fibonacci numbers*. This means that (regardless of what the base-cases are), the runtime is *exponential*. Therefore the runtime is **not** $O(n^2)$.

Dynamic Programming solution:

We can use a table to store the results of the sub-problems, so that they don't need to be re-computed.

```
FindBalanceDP(n):
    Initialize table $B[1, \ldots, n]$
    $B[1] = 100, B[2] = 600$
    for $i = 3$ to $n$:
        B[i] = B[i-1]*1.5 - B[i-2]/2
    return $B[n]$
```

The runtime of the above solution is $\Theta(n)$ since it only loops through a table of size n, performing a constant number of operations at each iteration.

Question 6

In this question, we must find a relationship that defines the subproblems. Consider an example of the input heights, $H[]$:

$$4, 6, 2, 7, 9, 1, 4, 5, 3, 7, 8, 9, 6$$

Suppose that we consider the longest sequence of people we can find, up to and *including* a certain index i. Specifically, let $L[i]$ be the longest sequence of people from the first set of i people, *including* person i. For example, $L[1] = 1$ because we could just select the person of height 4, at position $i = 1$.

$L[2] = 2$ since we could select the sequence $(4, 6)$.
$L[3] = 1$ since if we want to include height 2 in the sequence, we can't include anyone else.
$L[4] = 3$, using sequence $(4, 6, 7)$
$L[5] = 4$, using sequence $(4, 6, 7, 9)$
$L[6] = 1$, since if we include the person with height 1, there is no longer subsequence
$L[7] = 2$ using sequence $(1, 4)$
$L[8] = 3$ using sequence $(1, 4, 5)$
$L[9] = 2$ using sequence $(2, 3)$
$L[10] = 4$ using sequence $(1, 4, 5, 7)$
$L[11] = 5$ using sequence $(1, 4, 5, 7, 8)$
$L[12] = 6$ using sequence $(1, 4, 5, 7, 8, 9)$
$L[13] = 4$ using sequence $(1, 4, 5, 6)$.

The maximum of all these values is 6, and therefore the longest line we can make of increasing height is 6: $(1, 4, 5, 7, 8, 9)$.

Defining the table: $L[1 \dots n]$

- $L[i]$ is the length of the longest line of people (satisfying the requirements) selected from people $1, \dots, i$ such that person i is included.
- $L[1] = 1$, since with just one person, the line length is 1.
- If $i \geq 2$ then the value of $L[i]$ can be found by including person i to the longest sequence in $L[1], \dots, L[i-1]$, as long as person i is not too short.
- The final answer the maximum of $L[1], L[2], \dots L[n]$.

Since the recurrence relationship above references the *previous* entries in the table, then we can fill the table in from left to right. The algorithm is described below:

LongestIncreasingSubsequence($H[]$)

```
Step 1: Initialize the table: $L[1] = 1$
Step 2: Fill in the table from left to right:
        for $k = 2$ to $n$
            max = 1
            for $i = 1$ to $k - 1$
                if $H[i] < H[k]$
                    if $L[i] + 1 >$ max
                        max $= L[i] + 1$
            $L[k] =$ max
        Return maximum of $L[1], L[2], \dots, L[n]$
```

Runtime: The algorithm performs a double loop through a table of size n performing a constant number of operations per entry, and therefore runs in time $\Theta(n^2)$.