# TCSS 343 - Week 4

Jake McKenzie

October 27, 2018

**Dynamic Programming**

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

. . .

Richard Bellman's **Principle of Optimality**

"What we choose means more than what was handed to us by chance."

. . .

Ada Palmer

" If 'dynamic programming' didn't have such a cool name, it would be known as 'populating a table'".

. . .

Mark Dominus

1. Today we're going to explore dynamic programming. Below are three implementations of the fibonacci algorithm that I wrote in python. I want you to draw the **"tree"** for each then reflect on how the "bottom up" apprach is different from the other two? (Hint: They are all trees but also different types of trees. This is a key insight in my opinion in idea in understanding dynamic programming)

```python
# recusive fibonacci
def F(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return F(n-1) + F(n-2)
```

```python
# "top down" memoized recursive fibonacci
memo = {}
def Fib(n):
    if n in memo: return memo[n]
    if n == 0: f = 0
    elif n == 1: f = 1
    else: f = Fib(n-1) + Fib(n-2)
    memo[n] = f
    return f
```

```python
# "bottom up" iterative fibonacci
def fib(n):
    fn = [0,1]
    for i in range(n >> 1):
        fn[0] += fn[1]
        fn[1] += fn[0]
    return fn[n % 2]
```

2. I found these really cool recursive Fibonacci formulas: $F_{2n+1} = F_{n+1}^2 + F_n^2$ and $F_{2n} = 2F_{n+1}F_n - F_n^2$ now can you use them to find $F_{11}$ and $F_{10}$(worth noting that $F_2 = 1$, $F_1 = 1$, and $F_0 = 0$)?

3. Can you now write an iterative method that computes $F_{2n}$ and $F_{2n+1}$ using dynamic programming?

4. How is this different from the basic Fibonacci formula? What is the time complexity of this function? Space complexity?

```
# Rules: X,Y > 0
def game_of_Stanley_Gill(X,Y):
    x = X
    u = X
    y = Y
    v = Y
    while x != y:
        if x > y:
            x = x - y
            u = v + u
        elif y > x:
            y = y - x
            v = u + v
    print((x + y) >> 1) #GCD(X,Y)
    print((u + v) >> 1) #LCM(X,Y)
```

5. This is a game shown to Edsgar Dijkstra(pictured right) when he was still an undergraduate, attributed to Stanley Gill(pictured left), an early computer scientist. Now it is true that $2XY = xv + yu$, which can be seen when the variables are initialized. But is it always true given that $X, Y > 0$? Show why this is true. (Hint: You can use the comments to help you along. Remember that the $>> 1$ operation is the same as $/2$.)

6. For the following problem stated as pseudo-code, let $A[\ell \ldots r]$ denote the sublist of the integer list $A$ from the $\ell$-th to the $r$-th element inclusive, let `Cubic(`$A[1 \ldots n]$`)` denote an algorithm that runs in time $\Theta(n^3)$.

```
Three(A[1...n])
      If n ⩽ 1 Then Return // nothing to do
      Cubic(A[1...n])
      Three(A[1...⌊n/2⌋])
      Three(A[⌊n/4⌋ + 1...⌊3n/4⌋])
      Three(A[⌊n/2⌋ + 1...n])
End Three.
```

a. State a recurrence that gives the complexity $T(n)$ for algorithm `Three`.

b. Find the tight complexity of algorithm `Three`.

7. For the following problem stated as pseudo-code, let $A[\ell \ldots r]$ denote the sublist of the integer list $A$ from the $\ell$-th to the $r$-th element inclusive, let `Swift(`$A[1 \ldots n]$`)` denote an algorithm that runs in time $\Theta(n \log(\log n))$.

```
One(A[1...n])
      If n ⩽ 1 Then Return // nothing to do
      Swift(A[1...n])
      One(A[1...⌊n/2⌋])
End One.
```

a. State a recurrence that gives the complexity $T(n)$ for algorithm `One`.

b. Find the tight complexity of algorithm `One`.

8. The rod cutting problem has **overlapping subproblems** and **optimal substructure** which means we can solve it by dynamic programming.

a. State a recurrence that gives the complexity $T(n)$ for the rod cutting problem.

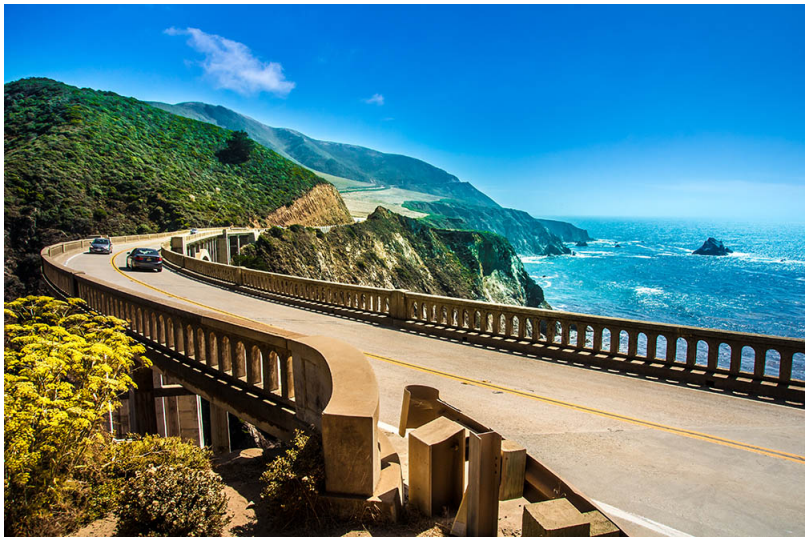
b. State a tight asymptotic bound on the recursive solution to the rod cutting problem from $T(n)$ that you found.

```
Memoized-Cut-Rod(p, n)
  // initialize memo (an array r[] to keep max revenue)
  r[0] = 0
  for i = 1 to n
    r[i] = -∞ // r[i] = max revenue for rod with length=i
  return Memorized-Cut-Rod-Aux(p, n, r)

Memoized-Cut-Rod-Aux(p, n, r)
  if r[n] >= 0
    return r[n] // return the saved solution
  q = -∞
  for i = 1 to n
    q = max(q, p[i] + Memoized-Cut-Rod-Aux(p, n-i, r))
  r[n] = q // update memo
  return q
```

9. Say you have access to a function **dict** that returns true if its input is a valid English word, and false otherwise. We are given as input a sentence from which the punctuation has been stripped (for example: "dynamicprogrammingisfabulous"). Assuming calls to dict takes a unit time, give an $O(n^2)$ time algorithm to figure out whether an input string of length n can be split into a sequence of valid words or not. HINT: Try to remove valid words from the end of the input string.

| r | $sp_r$ | $sc_r$ |
|---|---|---|
| 1 | 5 | 1 |
| 2 | 9 | 6 |
| 3 | 26 | 18 |
| 4 | 31 | 22 |
| 5 | 36 | 28 |



A. So imagine you've graduated from University of Washington in Tacoma and you got a nice job at a startup in San Francisco. It's the weekend and you want to drive down to Monterrey to enjoy some wine with friends. Now depending on what scenery you're driving through you find yourself speeding through that locale for each time unit you're in it. It's the weekend you don't really care how much time it takes you to get to Monterrey, but you defintely don't want a ticket. What's the maximum amount of scenic value you can get out of your weekend drive to Monterrey?

| r\speedlimit | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| {1} | 0 | | | | | | | | | | | | | | | |
| {1,2} | 0 | | | | | | | | | | | | | | | |
| {1,2,3} | 0 | | | | | | | | | | | | | | | |
| {1,2,3,4} | 0 | | | | | | | | | | | | | | | |
| {1,2,3,4,5} | 0 | | | | | | | | | | | | | | | |

B. Let $s[1..m]$ and $t[1..n]$ be the two strings to be matched. Let $M(i,j)$ be the number of mismatches in the best alignment of $s[1..i]$ and $t[1..j]$. Which of the following is a correct recursive formulation of $M(i,j)$ for $i,j > 0$? Note: By convention $s[1..0]$ and $t[1..0]$ are taken to be the empty string, so $M(i,0) = i$ for $i \in \{1,2,...,m\}$ and $M(0,j) = j$ for $j \in \{1,2,...,n\}$.

a) $M(i,j) = 2 + M(i-1,j-1)$, if $s[i] \neq t[j]$
$min(M(i-1,j), M(i,j-1))$, otherwise

b) $M(i,j) = M(i-1,j-1)$, if $s[i] = t[j]$
$min(M(i-1,j), M(i,j-1))$, otherwise

c) $M(i,j) = 1 + M(i-1,j-1)$, if $s[i] = t[j]$
$1 + min(M(i-1,j), M(i,j-1))$, otherwise

d) $M(i,j) = M(i-1,j-1)$, if $s[i] = t[j]$
$1 + min(M(i-1,j), M(i,j-1))$, otherwise

C. You have a bag that you want to fill with toys. There are $N$ toys, and the ith toy has weight $w[i]$. The bag can hold a total weight of at most $W$. Maximize the number of toys you can take with you in the bag.

Formally, choose a set of as many toys as possible, such that the sum of their weights is $\leqslant W$.

Constraints: $N \leqslant 1000$; $w[i] \leqslant 10,000 \ \forall \ i$.

Choose the best option out of the following

a) The simplest solution to this problem is using Dynamic Programming.

b) This problem cannot be solved using Dynamic Programming.

c) This problem can be solved with Dynamic Programming, but has a simpler solution without Dynamic Programming.