Dynamic Programming (DPV Chapter 6)

I can not be held responsible for a wrong answer.

These are my notes for CS6515 when I took the class Fall, 2020.

As of Spring, 2021, I am a CS6515 TA. This makes things a little complicated...

First, these notes were written when I was a student and do not have any additional insights from me as a TA. Trust these as you would trust a student.

Second, these notes are not official, nor are they supported by the class. Do not ask another TA to help with a solution. Do not ask another TA to make a correction. Send all edit requests to myself (Joves) either in slack or in Piazza.

Finally, please do not copy or share these notes outside of class. I will be releasing problems throughout the current semester. Some of these problems are used as assigned HWs, and I have to make sure all students have the opportunity to work on them for themselves.

This document includes solutions to:
1. Lecture Problems (always available).
2. HW Practice Problems (added as practice solutions are released).
3. HW Assigned Problems (added as HW deadlines pass).

Addressing some common questions/comments:
- Edit Distance was not in the lectures, but I added it because 6.26 is a variant which was assigned for practice.
- I will not be doing any backtracking. I know the provided solutions have it, but we've been told many, many times by Rocko that they don't care about this.
- You don't always need a base case. I confirmed with Rocko in OH. It is almost always needed, but it's possible you don't need one. Best to state this in your solution with "No base case needed" to make it explicit. **Be VERY careful about this. When in doubt, just use a base case.**
- Your subproblem (part a) does not need the base case in the range. The base case is not a valid call (usually). Seems like TAs are okay with either as shown in their provided solutions.
- You do need to state your base case range in your recurrence (part b) and pseudocode (part c) because this is your internal table for tracking.

Approach

Rules:

- No Recursion.
- No Memoization.

Steps:

1. Find the subproblem.
   - State your ranges
     - For 1 ≤ i ≤ n
   - Define the subproblem in words.
   - Define the problem on ith item.
     - T(i) or T(i, j) is ...
   - Possibly add a constraint such as include last element
     - T(i) is ... that includes x[i].
   - Might be different than what the overall question asks.
   - For example, a True/False final answer can usually be set up with Min/Max first and then checked at the end.
     - T(i) or T(i, j) is the maximum ...

2. Find the recurrence.
   - Must include base case(s) if needed
     - T(0) = ...
   - State your ranges
     - For 1 ≤ i ≤ n
   - Define the current problem as a relationship of smaller subproblems.
     - T(i) in terms of T(i-1)
     - T(i, j) in terms T(i-1, j), T(i, j-1), and/or T(i-1, j-1)

3. Pseudocode for algorithm.
   - No real code allowed.
   - Must include base case(s) if needed.
   - Always return what the problem asks for.
   - May be a different return from the subproblem definition.

4. Analyze the runtime.
   - Can annotate code with each runtime and compute at the end.
   - Can explain in words.
   - Include all non trivial (arithmetic) lines.

Template:

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...
b. State recurrence for entries of your table in terms of smaller subproblems.
c. Write pseudocode for your algorithm to solve this problem.
d. State and analyze the running time of your algorithm.

Tips on how to identify the problem

The key to solving DP problems quickly and efficiently is to first figure out what kind of problem it is. Then start with the generic solution that you learned in lectures, and apply any details specific to the problem.

LIS

- If your problem only has a single array.
- If your problem is comparing to itself.
- If your problem only looks back one element at a time (not a window, or a bag).
- This is normally (there are exceptions) a 1-D table.
- There are two variations:
    - With O(n) lookback.
        - If you need to check all previous elements.
        - If you need to check all previous elements > or < some requirement.
        - This results in O(n^2)
    - With O(1) lookback.
        - If you need to check one element back.
        - If you need to check a constant number of elements back.
        - If your restriction moves forward with i.
        - If you need to carry the max, sum, count, etc. forward.
            - Since you carry your answer forward, you only compare against i-1.
        - This results in O(n)

LCS

- If your problem is comparing between two arrays.
- If your problem is looking for something in common.
- This is normally (there are exceptions) a 2-D table.
- There are two variations:
    - Substring.
        - If you want the matches to be consecutive.
        - Add 1 (or whatever) if the comparison succeeds.
        - 1 + T(i-1, j-1)
        - Reset to 0 when a comparison fails.
        - The solution is the max of T()
        - This results in O(n*m) or O(n^2) depending if the two arrays have different sizes or not.
    - Subsequence.
        - If you want the comparison to be able to skip mismatches.
        - Add 1 (or whatever) if the comparison succeeds.
        - 1 + T(i-1, j-1)
        - Take the max of either side if the comparison fails.
        - max {T(i, j-1), T(i-1, j)}
        - The solution is the final results T(n,m) or T(n,n)

        - This results in O(n*m) or O(n^2) depending if the two arrays have different sizes or not.

Edit Distance

- This is very similar to LCS and can be considered a 3rd variation.
- If your problem is comparing two arrays.
- If your problem is looking to minimize differences.
- If your problem assigns some penalty to differences.
- If your problem assigns some points to matches.
- If you are aligning two arrays.
- This is normally (there are exceptions) a 2-D table.
- For i = 1->n
- For j = 1->m
- T(i,j) = min/max {T(i-1, j-1) + function(i,j),
  T(i-1, j) + function(i,j),
  T(i, j-1) + function(i,j)}
- Basically, you declare a function to determine the value, penalty, etc. of aligning i and j.
- Then you find the optimal position of i and j by checking the 3 possible ways to align them.
- This results in O(n*m) or O(n^2) depending if the two arrays have different sizes or not.

CMM

- If your problem needs a windowing check.
- If your problem is asking for the most efficient way to split an array.
- If your problem is asking if an array can be split in some way to do something.
- This is normally (there are exceptions) a 2-D table.
- There are two variations:
    - Windowing only:
        - Start a window size of 1->n
        - Set i to 1->n-window
        - Set j to i+window
        - Calculate T(i,j) using smaller windows between i and j.
        - You can do this because your window is growing, so all smaller windows T(i,j-1) and T(i+1,j) are already calculated.
        - This results in O(n^2)
    - Windowing and break at midpoint:
        - This (to me) is the hardest class of DP problems.
        - Start a window size of 1->n
        - Set i to 1->n-window
        - Set j to i+window
        - Set k (break) from i+1 -> j-1
        - Track (not in table) the max of T(i,j) by moving k and finding what the optimal position for k is to maximize T(i,j).

        - You can do this because your window is growing, so all smaller windows T(i,k) and T(k,j) are already calculated.
        - This results in O(n^3)

Knapsack

- If your problem needs to check if something can fit or can add up to.
- If your problem needs a maximum value for a specific budget.
- If your problem needs to find the minimum budget needed for a specific value.
- There are two variations:
    - Limited items.
        - This is normally (there are exceptions) a 2-D table.
        - For each increment of the budget:
        - For each increment of the items:
        - If the item can fit
            - T(b,i) = max{T(b, i-1), T(b-w[i], i-1) + v[i]}
            - You want to know if this same budget, before i is considered, is worth more or less than removing enough room to add this new item.
        - If the item cannot fit
            - T(b, i) = T(b, i-1)
            - It's as if this item was never considered.
        - This results in O(Bn)
    - Unlimited items.
        - This is normally (there are exceptions) a 1-D table. Can be 2-d if you don't simplify.
        - Set the initial value of each b to 0
        - For each increment of the budget:
        - For each increment of the items:
        - If the item can fit
            - T(b) = max{T(b), T(b-w[i]) + v[i]}
            - You want to know if this same budget, before the current instance of the item is considered, is worth less or more than removing enough room to add this item.
            - The main difference between Limited and Unlimited is that we don't check i-1 because we don't care if i was considered or not.
        - If the item can't fit.
            - Don't do anything.
            - The main difference between Limited and Unlimited is that we don't check i-1 because we don't care if i was considered or not
        - This results in O(Bn)

Fibonacci

a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

For 1 ≤ i ≤ n
Let T(i) = the ith Fibonacci number

b. **State recurrence for entries of your table in terms of smaller subproblems.**

T(0) = 0
T(1) = 1

For 2 ≤ i ≤ n
T(i) = T(i-1) + T(i-2)

c. **Write pseudocode for your algorithm to solve this problem.**

```
T(0) = 0
T(1) = 1

for i = 2 -> n:
      T(i) = T(i-1) + T(i-2)
return T(n)
```

d. **State and analyze the running time of your algorithm.**

for i = 2 -> n: O(n)

O(n) = O(n)

Longest Increasing Subsequence (LIS)

a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

```
For 1 ≤ i ≤ n
Let T(i) = length of LIS in a[1...i] which includes a[i]
```

b. **State recurrence for entries of your table in terms of smaller subproblems.**

No base case needed.

```
For 1 ≤ i ≤ n
T(i)  = 1 + max{T(j) if a[j] < a[i]: 1 ≤ j ≤ i-1}
      = 1 otherwise
```

c. **Write pseudocode for your algorithm to solve this problem.**

```
for i = 1 -> n:
      T(i) = 1
      for j = 1 -> i-1:
            if a[j] < a[i]:
                  T(i) = max{T(i), T(j) + 1}
return max{T(·)}
```

d. **State and analyze the running time of your algorithm.**

```
for i = 1 -> n: O(n)
      for j = 1 -> i-1: O(n)
return max{T(·)}: O(n)
```

$O(n) * O(n) +$ ~~$O(n)$~~ $= O(n^2)$

Longest Common Subsequence (LCS)

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

For $1 \le i \le n$
For $1 \le j \le n$
Let T(i, j) = length of longest common subsequence in x[1...i] and y[1...j]

b. State recurrence for entries of your table in terms of smaller subproblems.

$T(i, 0) = 0$ for $0 \le i \le n$
$T(0, j) = 0$ for $0 \le j \le n$

For $1 \le i \le n$
For $1 \le j \le n$

```
T(i, j)     = T(i-1, j-1) + 1       if x[i] = y[j]
            = max {T(i-1, j), T(i, j-1)}  if x[i] ≠ y[j]
```

c. Write pseudocode for your algorithm to solve this problem.

```
for i = 0 -> n:
      T(i, 0) = 0

for j = 0 -> n:
      T(0, j) = 0

for i = 1 -> n:
      for j = 1 -> n:
            if x[i] = y[j]:
                  T(i, j) = T(i-1, j-1) + 1
            else:
                  T(i, j) = max{T(i-1, j), T(i, j-1)}
return T(n, n)
```

d. State and analyze the running time of your algorithm.

```
for i = 0 -> n: O(n)
for j = 0 -> n: O(n)
for i = 1 -> n: O(n)
      for j = 1 -> n: O(n)
```

~~$O(n) + O(n)$~~ $+ O(n) * O(n) = O(n^2)$

Edit Distance

a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

```
Let diff(i, j) = 0 if x[i] = y[j], 1 otherwise

For 1 ≤ i ≤ n
For 1 ≤ j ≤ m
Let T(i, j) = edit distance between x[1...i] and y[1...j]
```

b. **State recurrence for entries of your table in terms of smaller subproblems.**

```
T(i, 0) = i for 0 ≤ i ≤ n
T(0, j) = j for 0 ≤ j ≤ m

For 1 ≤ i ≤ n
For 1 ≤ j ≤ m
T(i, j) = min{T(i-1, j-1) + diff(i, j), T(i-1, j) + 1 , T(i, j-1) + 1}
```

c. **Write pseudocode for your algorithm to solve this problem.**

```
for i = 0 -> n:
      T(i, 0) = i

for j = 0 -> m:
      T(0, j) = j

for i = 1 -> n:
      for j = 1 -> m:
            T(i, j) = min{T(i-1, j-1) + diff(i, j),
                                  T(i-1, j) + 1 , T(i, j-1) + 1}
return T(n, m)
```

d. **State and analyze the running time of your algorithm.**

```
for i = 1 -> n: O(n)
for j = 0 -> m: O(m)
for i = 1 -> n: O(n)
      for j = 0 -> m: O(m)
```

~~O(n) + O(m)~~ + O(n) * O(m) = O(nm)

Knapsack (no repeat)

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

```
For 1 ≤ b ≤ B
For 1 ≤ i ≤ n
Let T(b, i) = max value possible using subset of objects 1, ..., i
                  and total weight ≤ b
```

b. State recurrence for entries of your table in terms of smaller subproblems.

```
T(b, 0) = 0 for 0 ≤ b ≤ B
T(0, i) = 0 for 0 ≤ i ≤ n

For 1 ≤ b ≤ B
For 1 ≤ i ≤ n
T(b, i)    = max{T(b, i-1), T(b-w[i], i-1) + v[i]}   if b ≥ w[i]
           = T(b, i-1)                               if b < w[i]
```

c. Write pseudocode for your algorithm to solve this problem.

```
for b = 0 -> B:
      T(b, 0) = 0

for i = 0 -> n:
      T(0, i) = 0

for b = 1 -> B:
      for i = 1 -> n:
            if b ≥ w[i]:
                  T(b, i) = max{T(b, i-1), T(b-w[i], i-1) + v[i]}
            else:
                  T(b, i) = T(b, i-1)
return T(B, n)
```

d. State and analyze the running time of your algorithm.

```
for b = 0 -> B: O(B)
for i = 0 -> n: O(n)
for b = 1 -> B: O(B)
      for i = 1 -> n: O(n)
```

~~O(B) + O(n)~~ + O(B) * O(n) = O(Bn)

Knapsack (repeat)

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

```
For 1 ≤ b ≤ B
For 1 ≤ i ≤ n
Let T(b, i) = max value possible using **multiset** of objects 1, ..., i
                  and total weight ≤ b
```

b. State recurrence for entries of your table in terms of smaller subproblems.

```
T(b, 0) = 0 for 0 ≤ b ≤ B
T(0, i) = 0 for 0 ≤ i ≤ n

For 1 ≤ b ≤ B
For 1 ≤ i ≤ n
T(b, i)     = max{T(b, i-1), T(b-w[i], i) + v[i]}     if b ≥ w[i]
            = T(b, i-1)                               if b < w[i]
```

c. Write pseudocode for your algorithm to solve this problem.

```
for b = 0 -> B:
      T(b, 0) = 0

for i = 0 -> n:
      T(0, i) = 0

for b = 1 -> B:
      for i = 1 -> n:
            if b ≥ w[i]:
                  T(b, i) = max{T(b, i-1), T(b-w[i], i) + v[i]}
            else:
                  T(b, i) = T(b, i-1)
return T(B, n)
```

d. State and analyze the running time of your algorithm.

```
for b = 0 -> B: O(B)
for i = 0 -> n: O(n)
for b = 1 -> B: O(B)
      for i = 1 -> n: O(n)
```

~~O(B) + O(n)~~ + O(B) * O(n) = O(Bn)

Knapsack (repeat) simplified

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

```
For 1 ≤ b ≤ B
Let T(b) = max value possible using **multiset** of objects 1, ..., n
            and total weight ≤ b
```

b. State recurrence for entries of your table in terms of smaller subproblems.

```
T(0) = 0

For 1 ≤ b ≤ B
T(b) = max{T(b-w[i]) + v[i]: for 1 ≤ i ≤ n if b ≥ w[i]}
```

c. Write pseudocode for your algorithm to solve this problem.

```
T(0) = 0

for b = 1 -> B:
      T(b) = 0
      for i = 1 -> n:
            if b ≥ w[i]:
                  T(b) = max{T(b), T(b-w[i]) + v[i]}
return T(B)
```

d. State and analyze the running time of your algorithm.

```
for b = 1 -> B: O(B)
      for i = 1 -> n: O(n)

O(B) * O(n) = O(Bn)
```

Chain Matrix Multiplication (CMM)

a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

Let m[i-1] and m[i] be the dimensions of A[i]

For $1 \le i \le j \le n$
Let T(i, j) = minimum cost for computing A[i] * A[i+1] * ... * A[j]

b. **State recurrence for entries of your table in terms of smaller subproblems.**

T(i, i) = 0 for $1 \le i \le n$

For $1 \le i < j \le n$
T(i, j) = min{T(i, k) + T(k+1, j) + m[i-1] * m[k] * m[j]: $i \le k \le j-1$}

c. **Write pseudocode for your algorithm to solve this problem.**

```
for i = 1 -> n:
      T(i, i) = 0

for w = 1 -> n-1:
      for i = 1 -> n-w:
            j = i + w
            T(i, j) = inf
            for k = i -> j-1:
                  cur = T(i, k) + T(k+1, j) + m[i-1] * m[k] * m[j]
                  T(i, j) = min{T(i, j), cur}
return T(1, n)
```

d. **State and analyze the running time of your algorithm.**

```
for i = 1 -> n: O(n)
for s = 1 -> n-1: O(n)
      for i = 1 -> n-s: O(n)
            for k = i -> j-1: O(n)
```

~~O(n)~~ + O(n) * O(n) * O(n) = O(n^3)

Shortest Path (Single Source) - Bellman-Ford

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

For 0 ≤ i ≤ n-1
For vertex z in V
Let T(i, z) = length of shortest path from s to z using ≤ i edges

b. State recurrence for entries of your table in terms of smaller subproblems.

T(0, s) = 0

For all z in V not equal to s
T(0, z) = inf

For 1 ≤ i ≤ n-1
T(i, z) = min{T(i-1, z), min{T(i-1, y) + w(y, z): for (y, z) in E}}

c. Write pseudocode for your algorithm to solve this problem.

```
for z in V:
      T(0, z) = inf

T(0, s) = 0

for i = 1 -> n-1:
      for z in V:
            T(i, z) = T(i-1, z)
            for (y, z) in E:
                  T(i, z) = min{T(i, z), T(i-1, y) + w(y, z)}
return T(n-1, ·)
```

d. State and analyze the running time of your algorithm.

```
for i = 1 -> n-1: O(n)
      for z in V:
            for (y, z) in E: O(m)
```

Two inner loops combine to go over all edges - O(m)

O(n) * O(m) = O(nm)
O(|V|) * O(|E|) = O(|V||E|)

- To detect a negative weight cycle:
    - ~~for i = 1 -> n-1~~
    - for i = 1 -> n
    - If T(n, z) < T(n-1, z) for any z, then there is a cycle **that s can see**

Shortest Path (All Pairs) - Floyd-Warshall

a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**

```
For 0 ≤ i ≤ n
For 1 ≤ s, t ≤ n
Let T(i, s, t) = length of shortest path from s to t using a subset of
                  {1, ..., i} intermediate vertices
```

b. **State recurrence for entries of your table in terms of smaller subproblems.**

```
For 1 ≤ s, t ≤ n
T(0, s, t)  = w(s, t)         if (s, t) in E
            = inf             if (s, t) not in E

For 1 ≤ i ≤ n
For 1 ≤ s, t ≤ n
T(i, s, t) = min{T(i-1, s, t), T(i-1, s, i) + T(i-1, i, t)}
```

c. **Write pseudocode for your algorithm to solve this problem.**

```
for s = 1 -> n:
      for t = 1 -> n:
            if (s, t) in E:
                  T(0, s, t) = w(s, t)
            else:
                  T(0, s, t) = inf

for i = 1 -> n:
      for s = 1 -> n:
            for t = 1 -> n:
                  T(i, s, t) =
                        min{T(i-1, s, t), T(i-1, s, i) + T(i-1, i, t)}
return T(n, ·, ·)
```

d. **State and analyze the running time of your algorithm.**

```
for s = 1 -> n: O(n)
      for t = 1 -> n: O(n)
for i = 1 -> n: O(n)
      for s = 1 -> n: O(n)
            for t = 1 -> n: O(n)
```

~~O(n) * O(n)~~ + O(n) * O(n) * O(n) = $O(n^3)$
~~O(|V|) * O(|V|)~~ + O(|V|) * O(|V|) * O(|V|) = $O(|V|^3)$

- To detect a negative weight cycle:
  - If T(n, i, i) < 0 for any i, then there is a cycle

6.1 - Contiguous Sum (LIS)

A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S. For instance, if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

Input: A list of numbers, $a_1, a_2, \ldots, a_n$.
Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55.

(*Hint:* For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

Note: We will find the maximum sum instead since we aren't usually asked to backtrack.
Note: This is a different solution than the class provided one because we are treating the empty subsequence (with a sum of 0) as a possible answer.

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...
For 1 ≤ i ≤ n
Let T(i) = maximum sum of a contiguous subsequence ending at a[i]

b. State recurrence for entries of your table in terms of smaller subproblems.
T(0) = 0

For 1 ≤ i ≤ n
T(i) = max{0, T(i-1)} + a[i]

c. Write pseudocode for your algorithm to solve this problem.

```
T(0) = 0

for i = 1 -> n:
      T(i) = max{0, T(i-1)} + a[i]
return max{T(·)}
```

d. State and analyze the running time of your algorithm.

```
for i = 1 -> n: O(n):
    T(i) = max ...: O(1)
return max{T(·)}: O(n)
```

O(n) ~~* O(1)~~ + O(n) = 2 * O(n) = O(n)

6.4 - Dictionary lookup (LIS/CMM)

You are given a string of n characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function `dict`$(\cdot)$: for any string w,

$$\texttt{dict}(w) = \begin{cases} \texttt{true} & \text{if } w \text{ is a valid word} \\ \texttt{false} & \text{otherwise} . \end{cases}$$

(a) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to `dict` take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

Note: We skip part b since we aren't usually asked to backtrack.

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

For 1 ≤ i ≤ n
Let T(i) = True if s[1...i] can be reconstituted as a sequence of valid words.

b. State recurrence for entries of your table in terms of smaller subproblems.

T(0) = True

```
For 1 ≤ i ≤ n
T(i)  = True if T(j-1) and dict(s[j...i]) for any 1 ≤ j ≤ i
      = False otherwise
```

c. Write pseudocode for your algorithm to solve this problem.

```
T(0) = True

for i = 1 -> n:
      T(i) = False
      for j = 1 -> i:
            T(i) = T(i) or (T(j-1) and dict(s[j...i]))
return T(n)
```

d. State and analyze the running time of your algorithm.

```
for i = 1 -> n: O(n)
      for j = 1 -> i: O(n)
```

$O(n) * O(n) = O(n^2)$

6.8 - Longest common substring (LCS)

Given two strings $x = x_1x_2\cdots x_n$ and $y = y_1y_2\cdots y_m$, we wish to find the length of their *longest common substring*, that is, the largest k for which there are indices i and j with $x_ix_{i+1}\cdots x_{i+k-1} = y_jy_{j+1}\cdots y_{j+k-1}$. Show how to do this in time $O(mn)$.

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

```
For 1 ≤ i ≤ n
For 1 ≤ j ≤ m
T(i, j) = longest common substring ending with x[i] = y[j]
```

b. State recurrence for entries of your table in terms of smaller subproblems.

```
T(i, 0) = 0 for 0 ≤ i ≤ n
T(0, j) = 0 for 0 ≤ j ≤ m

For 1 ≤ i ≤ n
For 1 ≤ j ≤ m
T(i, j)     = T(i-1, j-1) + 1       if x[i] = y[j]
            = 0                     if x[i] ≠ y[j]
```

c. Write pseudocode for your algorithm to solve this problem.

```
for i = 0 -> n
T(i, 0) = 0

for j = 0 -> m
T(0, j) = 0

for i = 1 -> n:
      for j = 1 -> m:
            if x[i] = y[j]:
                  T(i, j) = T(i-1, j-1) + 1
            else:
                  T(i, j) = 0
return max{T(·, ·)}
```

d. State and analyze the running time of your algorithm.

```
for i = 0 -> n: O(n)
for j = 0 -> m: O(m)
for i = 1 -> n: O(n)
      for j = 1 -> m: O(m)
return max...: O(nm)
```

~~O(n) + O(m)~~ + O(n) * O(m) + O(nm) = 2 * O(nm) = O(nm)

6.17 - Making change 1 (unlimited) (Knapsack)

Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic-programming algorithm for the following problem.

Input: $x_1, \ldots, x_n; v$.
Question: Is it possible to make change for v using coins of denominations $x_1, \ldots, x_n$?

a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**
For 1 ≤ k ≤ v
Let T(k) = max sum ≤ k using **multiset** of coins x[1], ..., x[n]

b. **State recurrence for entries of your table in terms of smaller subproblems.**
T(0) = 0

For 1 ≤ k ≤ v
T(k) = max{T(k-x[i]) + x[i]: for 1 ≤ i ≤ n if k ≥ x[i]}

c. **Write pseudocode for your algorithm to solve this problem.**

```
T(0) = 0

for k = 1 -> v:
      T(k) = 0
      for i = 1 -> n:
            if k ≥ x[i]:
                  T(k) = max{T(k), T(k-x[i]) + x[i]}
return T(v) = v
```

d. **State and analyze the running time of your algorithm.**

```
for k = 1 -> v: O(v)
      for i = 1 -> n: O(n)

O(k) * O(n) = O(vn)
```

6.18 - Making change 2 (Knapsack)

Consider the following variation on the change-making problem (Exercise 6.17): you are given denominations $x_1, x_2, \ldots, x_n$, and you want to make change for a value v, but you are allowed to use each denomination *at most once*. For instance, if the denominations are $1, 5, 10, 20$, then you can make change for $16 = 1 + 15$ and for $31 = 1 + 10 + 20$ but not for 40 (because you can't use 20 twice).

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

```
For 1 ≤ k ≤ v
For 1 ≤ i ≤ n
Let T(k, i) = max sum ≤ k using subset of coins x[1], ..., x[i]
```

b. State recurrence for entries of your table in terms of smaller subproblems.

```
T(k, 0) = 0 for 0 ≤ k ≤ v
T(0, i) = 0 for 0 ≤ i ≤ n

For 1 ≤ k ≤ v
For 1 ≤ i ≤ n
T(k, i)     = max{T(k, i-1), T(k-x[i], i-1) + x[i]}   if k ≥ x[i]
            = T(k, i-1)                               if k < x[i]
```

c. Write pseudocode for your algorithm to solve this problem.

```
for k = 0 -> v:
      T(k, 0) = 0

for i = 0 -> n:
      T(0, i) = 0

for k = 1 -> v:
      for i = 1 -> n:
            if k ≥ x[i]:
                  T(k, i) = max{T(k, i-1), T(k-x[i], i-1) + x[i]}
            else:
                  T(k, i) = T(k, i-1)
return T(v, n) = v
```

d. State and analyze the running time of your algorithm.

```
for k = 1 -> v: O(v)
      for i = 1 -> n: O(n)

O(v) * O(n) = O(vn)
```

6.20 - Optimal binary search tree (CMM)

Optimal binary search trees. Suppose we know the frequency with which keywords occur in programs of a certain language, for instance:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

We want to organize them in a *binary search tree*, so that the keyword in the root is alphabetically bigger than all the keywords in the left subtree and smaller than all the keywords in the right subtree (and this holds for all nodes).

Figure 6.12 has a nicely-balanced example on the left. In this case, when a keyword is being looked up, the number of comparisons needed is at most three: for instance, in finding "while", only the three nodes "end", "then", and "while" get examined. But since we know the frequency

with which keywords are accessed, we can use an even more fine-tuned cost function, the *average number of comparisons* to look up a word. For the search tree on the left, it is

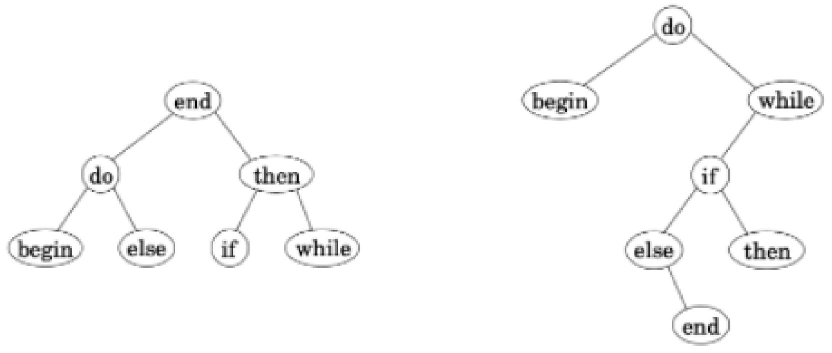
$$\text{cost} \;=\; 1(0.04) + 2(0.40 + 0.10) + 3(0.05 + 0.08 + 0.10 + 0.23) \;=\; 2.42.$$

By this measure, the best search tree is the one on the right, which has a cost of 2.18.

Give an efficient algorithm for the following task.

Input: n words (in sorted order); frequencies of these words: $p_1, p_2, \ldots, p_n$.
Output: The binary search tree of lowest cost (defined above as the expected number of comparisons in looking up a word).

Figure 6.12 Two binary search trees for the keywords of a programming language.



a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**
   For $1 \le i \le j \le n$
   Let T(i, j) = minimum cost to look up words p[i], ..., p[j]

b. **State recurrence for entries of your table in terms of smaller subproblems.**
   T(i, i) = p[i] for $1 \le i \le n$
   T(i, j) = 0 for $0 \le j < i \le n$

   For $1 \le i < j \le n$
   T(i, j) = min{T(i, k-1) + T(k+1, j) + sum {p[i..j]} : $i \le k \le j$}

c. **Write pseudocode for your algorithm to solve this problem.**

```
for i = 1 -> n:
      T(i, i) = p[i]
      for j = 0 -> i-1:
            T(i, j) = 0

for w = 1 -> n-1:
      for i = 1 -> n-w:
            j = i + w
            cost = sum{p[i...j]}
            T(i, j) = inf
            for k = i -> j-1:
                  cur = T(i, k-1) + T(k+1, j) + cost
                  T(i, j) = min{T(i, j), cur}
return T(1, n)
```

d. **State and analyze the running time of your algorithm.**

```
for i = 1 -> n: O(n)
      for j = 0 -> n-1: O(n)
for s = 1 -> n-1: O(n)
      for i = 1 -> n-s: O(n)
            cost...: O(n)
            for k = i -> j-1: O(n)
```

~~O(n) * O(n)~~ + O(n) * O(n) * 2 * O(n) = 2 * O(n^3) = O(n^3)

6.26 - Alignment (Edit Distance)

Sequence alignment. When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are *aligned*. To formalize this, think of a gene as being a long string over an alphabet $\Sigma = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of x and y is a way of matching up these two strings by writing them in columns, for instance:

$$\begin{array}{ccccccc} - & A & T & - & G & C & C \\ T & A & - & C & G & C & A \end{array}$$

Here the "$-$" indicates a "gap." The characters of each string must appear in order, and each column must contain a character from at least one of the strings. The score of an alignment is specified by a scoring matrix δ of size $(|\Sigma|+1) \times (|\Sigma|+1)$, where the extra row and column are to accommodate gaps. For instance the preceding alignment has the following score:

$$\delta(-,T) + \delta(A,A) + \delta(T,-) + \delta(-,C) + \delta(G,G) + \delta(C,C) + \delta(C,A).$$

Give a dynamic programming algorithm that takes as input two strings $x[1 \dots n]$ and $y[1 \dots m]$ and a scoring matrix δ, and returns the highest-scoring alignment. The running time should be $O(mn)$.

a. Define the entries of your table in words. E.g., T(i) or T(i, j) is ...

```
For 1 ≤ i ≤ n
For 1 ≤ j ≤ m
Let T(i, j) = maximum score of x[1...i] and y[1...j]
```

b. State recurrence for entries of your table in terms of smaller subproblems.

```
T(0, 0) = 0
T(i, 0) = T(i-1, 0) + δ(x[i], -) for 1 ≤ i ≤ n
T(0, j) = T(0, j-1) + δ(-, y[j]) for 1 ≤ j ≤ m

For 1 ≤ i ≤ n
For 1 ≤ j ≤ m
T(i, j) = max{T(i-1, j-1) + δ(x[i], y[j]),
              T(i-1, j) + δ(x[i], -),
              T(i, j-1) + δ(-, y[j])}
```

c. Write pseudocode for your algorithm to solve this problem.

```
T(0, 0) = 0

for i = 1 -> n:
      T(i-1, 0) + δ(x[i], -)

for j = 1 -> m:
      T(0, j-1) + δ(-, y[j])

for i = 1 -> n:
      for j = 1 -> m:
            T(i, j) =  max{T(i-1, j-1) + δ(x[i], y[j]),
                                  T(i-1, j) + δ(x[i], -),
                                  T(i, j-1) + δ(-, y[j])}
return T(n, m)
```

d. State and analyze the running time of your algorithm.

Assuming the use of δ is O(1)

```
for i = 1 -> n: O(n)
for j = 1 -> m: O(m)
for i = 1 -> n: O(n)
      for j = 1 -> m: O(m)
```

~~O(n) + O(m)~~ + O(n) * O(m) = O(nm)

Divide and Conquer (DPV Chapter 2)

I can not be held responsible for a wrong answer.

These are my notes for CS6515 when I took the class Fall, 2020.

As of Spring, 2021, I am a CS6515 TA. This makes things a little complicated...

First, these notes were written when I was a student and do not have any additional insights from me as a TA. Trust these as you would trust a student.

Second, these notes are not official, nor are they supported by the class. Do not ask another TA to help with a solution. Do not ask another TA to make a correction. Send all edit requests to myself (Joves) either in slack or in Piazza.

Finally, please do not copy or share these notes outside of class. I will be releasing problems throughout the current semester. Some of these problems are used as assigned HWs, and I have to make sure all students have the opportunity to work on them for themselves.

This document includes solutions to:

1. Lecture Problems (always available).
2. HW Practice Problems (added as practice solutions are released).
3. HW Assigned Problems (added as HW deadlines pass).

Approach

Rules:

- NO PSEUDOCODE - YOU WILL LOSE A LOT OF POINTS
- Each recurrence should be a smaller problem than before.
- Black boxes of known algorithms can be used.
    - Binary search - O(log n)
    - Merge sort - O(n log n)
    - FFT - O(n log n)
    - Median of Medians - O(n)

Steps:

1. Figure out your Black Box if needed.
    - Sorted = Binary Search (usually)
    - Unsorted = Merge sort (usually)
    - Polynomials, convolution, multiplication = FFT (usually)

2. State the Modification if needed. You may use the black box as is.
    - Which part of the black box you are changing.
    - What are your inputs and outputs to the black box.

3. State the steps of your algorithm.
    - NO PSEUDOCODE - use words
    - Must include base case(s) if needed.
    - Always return what the problem asks for.

4. Prove Correctness
    - Proof in words of why this algorithm solves the problem.
    - Prove the base case
    - The decision to select left or right in a Binary Search.
    - Prove that the problem gets smaller.

5. Analyze the runtime.
    - Can explain in words.
    - Can use black box runtime. Must include modifications' run time.
    - Can use Master Theorem.

Template:

a. Step 1:
   Step 2:
   Step 3:
b. Correctness
a. Runtime Analysis

Solving Recurrence

General Form:

$T(n) = a*T(n/b) + O(n^d)$

a = number of recursive calls
b = size of partition in each recursive call
n^d = effort at each recursion

Master Theorem:

$O(n^d)$	if $d > logb(a)$
$O(n^d * logb(n))$	if $d = logb(a)$ - can be simplified to $O(n^d * log(n))$
$O(n^{logb(a)})$	if $d < logb(a)$

Christian's method:

- If you can't or don't want to use the Master Theorem
- Determine work due to subproblem proliferation
- Determine work due to subproblem merging
- Compare them like in the master theorem to determine which dominates or are equal
- Solve the O() based on taking the dominating O(), or combining them.

Christian's post below:

A Conceptual Approach to Recurrences

For students who don't like memorizing seemingly arbitrary formulas, here's a quick crash course on the conceptual approach to solving recurrences that are covered by the Master Theorem. I developed these concepts by solving several recurrences by hand until I noticed patterns, and since then have never had to look back at the Master Theorem. I personally strongly prefer this approach because I find conceptual understanding to be much more enduring and meaningful as a learner. Of course your learning style may lean towards memorizing formulae, and that's fine too!

The runtime of a recurrence conceptually comes from two places: work due to subproblem proliferation vs. work done to merge subproblem results. For example, in the recurrence $T(n) = 4T(n/2) + O(n)$, the $4T(n/2)$ originates the work due to subproblem proliferation, and the $O(n)$ is the work done to merge subproblem results.

Part 1: Determining work due to subproblem proliferation. This is probably best illustrated with examples.

$T(n) = 2T(n/2)$ - two problems, half the size, just as much work as before.
$T(n) = 5T(n/5)$
$T(n) = 8T(n/8)$

All of those are $O(n)$, since we made the subproblems smaller, but still had the same total amount of work.

Here are several problems with quadratic subproblem proliferation; the problems got smaller, but we ended up with more of them:

$T(n) = 4T(n/2)$
$T(n) = 9T(n/3)$
$T(n) = 100T(n/10)$

Those are all $O(n^2)$ since the total amount of work to be done is growing quadratically.

$T(n) = 8T(n/2)$
$T(n) = 64T(n/4)$
$T(n) = 125T(n/5)$

Those are all $O(n^3)$. The problems got smaller, but we ended up with way more of them. Etc.

It could be a non-integer exponent, and that's when we need logs. Consider the middle one below.

$T(n) = 4T(n/2) \rightarrow O(n^2)$
$T(n) = 7T(n/2)$
$T(n) = 8T(n/2) \rightarrow O(n^3)$

We could estimate it to be $O(n^{2.8})$, which is pretty close. We could more accurately say it's $O(n^{\log2(7)})$, since that is, by definition, the exponent that turns 2 into 7. This even works if we get less subproblems than the linear case:

$T(n) = 2T(n/3) \rightarrow O(n^{\log3(2)}) \approx O(n^{0.631})$

Of course, if we only end up with a single problem at each step, we'll shrink down to a single unit of work:

$T(n) = T(n/2) \rightarrow O(1)$

The above is probably the hardest part of this. There are now three possibilities when you compare the work from subproblem proliferation with the work from merging (post processing) the results from the subproblems into the final answer:

1. The work due to subproblem proliferation dominates - the vast majority of the work occurs at the lowest level of recursion. Keep this dominant runtime as the overall runtime.

2. The work due to merging results dominates - the vast majority of the work occurs at the highest level of recursion. Keep this dominant runtime as the overall runtime.

3. The two tie - each level of O(log n) levels of recursion is the same amount of work, so the work at each level is multiplied by log n to get the overall result.

Here are a few overall examples:

$T(n) = 8T(n/2) + O(n^2)$
Well that's $O(n^3)$ from subproblem proliferation vs. $O(n^2)$ for merge, and the former dominates. It's $O(n^3)$ overall.

$T(n) = 3T(n/4) + O(n)$
That's $O(n^{\log4(3)}) \approx O(n^{0.792})$ vs. $O(n)$, so $O(n)$ dominates.

$T(n) = 25T(n/5) + O(n^2)$
This example has $O(n^2)$ work due to subproblem proliferation and $O(n^2)$ work due to merging subproblem results. Since it's a tie, this works out to $O(n^2 \log n)$.

Hope this conceptual crash course helps someone! Feel free, as always, to ask clarifying follow ups.

Roots of Unity

- We work with nth roots of unity which are in the form of
    - n = a power of 2
    - Non power of 2, as well as odd roots of unity exists. We don’t bother with them. They are not special in the context of D&C.
- $\omega_n = \omega_n^1$
    - The principal nth root of unity
    - First nth root of unity counter-clockwise from 1
    - **This is what you pass into FFT**.
- $(\omega_n)^{-1} = \omega_n^{n-1} = \omega_n^{-1}$
    - The inverse of the principal nth root of unity
    - The first nth root of unity clockwise from 1
    - **This is what you pass into IFFT**.
- Other truths - some require n to be even, which we assume
    - $\omega_n^n = \omega_n^0 = 1$
    - $-(\omega_n^k) = \omega_n^{k+n/2}$
    - $(\omega_n^k)^{-1} = \omega_n^{n-k} = \omega_n^{-k}$
    - $(\omega_n^k)^{-1} * \omega_n^k = \omega_n^{n-k} * \omega_n^k = \omega_n^n = \omega_n^0 = 1$
    - $(\omega_n^k)^2 = \omega_{n/2}^k$
    - $\omega_n^k = e^{2\pi ik/n} = \cos(2\pi k/n) + \sin(2\pi k/n)i$

Fast Fourier transform (FFT)

- **FFT does NOT do multiplication**
- Do not think you can toss two polynomials at it and hope for an answer.
- FFT converts an array of polynomial coefficients in the form of:
    - [a_0x^0, a_1x^1, a_2x^2, a_3x^3, a_4x^4, ..., a_kx^k] where k + 1 is a power of 2
        - Let a_i be the coefficient of the term with i exponent
        - k = 7, then we have size 8
        - k = 15, then we have size 16
        - We can have 0s in places where the degree is missing in the polynomial.
        - Incidentally this means we can pad the end with 0s to get to some k + 1 = a power of 2. Always, always work in a power of 2. FFT requires this.
- It converts this array into an array of values, evaluated at:
    - ω_n^0,..., ω_n^{n-1} or the inverse of this in IFFT (discussed below)
    - n here matches the size of your array, which as we said is where k + 1 is a power of 2
    - **We can rephrase this as, for some n which is a power of 2, an n sized array can hold an n-1 degree polynomial.**

Some info about your input array

- Toss out that 2n-1, or 2n, or 2n + 1 junk. It's confusing and a poor way to explain this. I'll dedicate a section to picking the right n.
- We convert a polynomial into values in order to easily multiply it.
- Remember, a polynomial's degree is the highest degree of any of its terms.
- Zero is a degree. It is the degree of the constant.
    - $7 = 7x^0$
- There are many different implementations of FFT. Some take unpadded coefficients, some take padded coefficients. Some take the omega (principal nth root of unity), some don't need it. Some take in the inverse omega, some simply give IFFT as a separate function.
- We will use what is most "complicated" and safest. In doing so we will fully understand what the FFT needs in order to run.
- So don't ask, do we need to pad? Do we need omega? Do we need both?
- You do. You should. You're going to risk your degree on you being too lazy to add a couple extra words or symbols? They're not even hard.

Determining n

- Let n be the size of the array needed for FFT.
- Use the same size for all of your polynomials if you multiply.
- It will also be the size of your resulting polynomial if you multiply.
- This n can be determined before anything else and used for your entire algorithm involving FFT.
- n = the lowest power of two strictly greater than the sum of your polynomial degrees
- If you want to multiply 8, 9, 10, 12 degree polynomials
  - 8 + 9 + 10 + 12 = 39; n = 64
- If you want to cube a polynomial of degree 10
  - 10 + 10 + 10 = 30; n = 32
- If you want to **convert** a 100 degree polynomial to value representation.
  - 100; n = 128
  - We do not do 2n or anything.
  - **FFT does NOT do multiplication**
  - We are merely converting polynomials to a convenient form for some reason, one of which can be multiplication.
  - If you are not multiplying, why do you need so many values?

Determining omega (ω)

- You use the nth degree of unity. This n is the same n as your array size.
  - ω_n^1 for FFT
  - ω_n^{-1} for IFFT
- You need to pass this into FFT
- FFT will take this omega and use it to calculate inputs for its evaluation.
- For a 7th degree polynomial, with size 8, we use the 8th root of unity.
  - A' = Evaluate A(x) at x = ω_8^0, ω_8^1, ω_8^2, ω_8^3, ω_8^4, ω_8^5, ω_8^6, ω_8^7
- For a n-1 degree polynomial, with size n, we use the nth root of unity.
  - A' = Evaluate A(x) at x = ω_n^0, ω_n^1, ω_n^2, ..., ω_n^{n-1}
- Inverse FFT does the inverse and passess in ω_n^{-1}
  - Inverse A' = Evaluate A(x) at x = ω_n^0, ω_n^{-1}, ω_n^{-2}, ..., $\omega_n^{-(n-1)}$
  - Note that $\omega_n^{-(n-1)} = \omega_n^{n+1} = \omega_n^1$

Multiplying polynomials

P(x) = A(x) * B(x) * C(x)

1. n = lowest power of 2 > sum of highest degrees in A, B, and C.
2. A' = FFT(A, ω_n^1), B' = FFT(B, ω_n^1), C' = FFT(C, ω_n^1)
3. P'(i) = A'(i) * B'(i) * C'(i) for $1 \le i \le n$
4. P = FFT(P', ω_n^{-1}) * 1/n

2.1 - Practice Multiplication

Use the divide-and-conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

Stolen from class solutions:

$X = 10011011 = 1001 * 2^4 + 1011$

$= X_L * 2^4 + X_R$

$10011011 \implies 1001$ and 1011

$Y = 10111010 = 1011 * 2^4 + 1010$

$= Y_L * 2^4 + Y_R$

$10111010 \implies 1011$ and 1010

Then:

$X_L + X_R = 1001 + 1011 = 10100$

$Y_L + Y_R = 1011 + 1010 = 10101$

$(X_L + X_R) * (Y_L + Y_R) = 10100 * 10101 = 110100100$

And:

$X_L * Y_L = 1001 \times 1011 = 1100011$

$X_R * Y_R = 1011 * 1010 = 1101110$

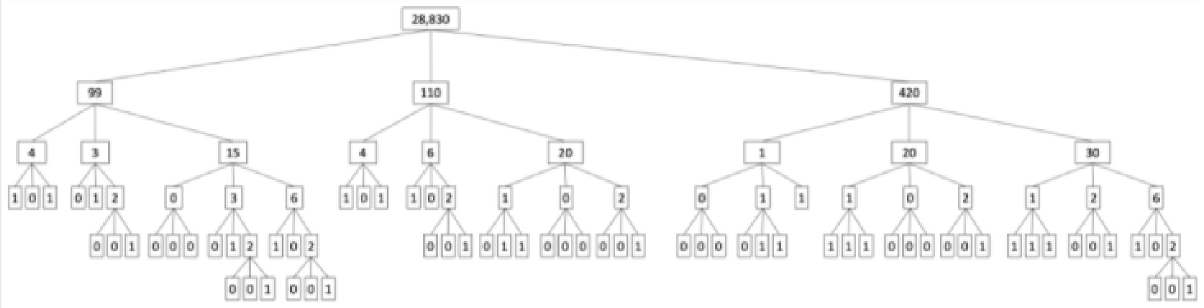
$(X_L * Y_R) + (X_R * Y_L) = (X_L + X_R) * (Y_L + Y_R) - (X_L * Y_L) - (X_R * Y_R)$

$= 110100100 - 1100011 - 1101110 = 11010011$

Therefore:

$X * Y = 2^8 * (X_L * Y_L) + 2^4 * ((X_L * Y_R) + (X_R * Y_L)) + (X_R * Y_R)$

$= 2^8 * 1100011 + 2^4 * 11010011 + 1101110 = 111000010011110$



2.5 - Solve recurrences

Solve the following recurrence relations and give a Θ bound for each of them.

(a) $T(n) = 2T(n/3) + 1$
(b) $T(n) = 5T(n/4) + n$
(c) $T(n) = 7T(n/7) + n$
(d) $T(n) = 9T(n/3) + n^2$
(e) $T(n) = 8T(n/2) + n^3$
(f) $T(n) = 49T(n/25) + n^{3/2} \log n$
(g) $T(n) = T(n-1) + 2$
(h) $T(n) = T(n-1) + n^c$, **where** $c \geq 1$ **is a constant**
(i) $T(n) = T(n-1) + c^n$, **where** $c > 1$ **is some constant**
(j) $T(n) = 2T(n-1) + 1$
(k) $T(n) = T(\sqrt{n}) + 1$

Using Master Theorem:

a. log3(2) > 0; O($n^{\log3(2)}$)

b. log4(5) > 1; O($n^{\log4(5)}$)

c. log7(7) = 1; O(n*log7(n)) = O(n*log(n))

d. log3(9) = 2; O(n^2*log3(n)) = O(n^2*log(n))

e. log2(8) = 3; O(n^3*log2(n)) = O(n^3*log(n))

Using Christian's Method:

f. log25(49) is log(49)/log(25) ~ 6/5 < 3/2; proliferation is dominated by merge work and gives O($n^{3/2}$ * log(n))

g. O(n) proliferation dominates O(1) merge work; O(n)

h. Each merge step is n^c; n recurrences; n^c * n = O(n^{c+1})

i. O(n) proliferation dominated by O(c^n) merge work; O(c^n)

j. Each recurrence doubles the work since a = 2, n recurrences; $O(2^n)$ proliferation dominates O(1) merge work; $O(2^n)$

k. LOL, I don't know. Pray we don't see this.

(k) $T(n) = T(\sqrt{n}) + 1 = \sum_{i=0}^{k} 1 + T(b)$, where k is an integer such that $n^{\frac{1}{2^k}}$ is a small constant b (the size of the base case). This implies that $k = O(\log\log n)$ and $T(n) = O(\log\log n)$.

2.7 - Roots of unity

What is the sum of the nth roots of unity? What is their product if n is odd? If n is even?

Sum:

Geometric series gives:

$$a + ax + ax^2 + \ldots + ax^{n-1} = a * (1-x^n) / (1-x)$$

$$\begin{aligned} 1 + \omega_n + \omega_n^2 + \ldots + \omega_n^{n-1} &= 1 * (1-\omega_n^n) / (1-\omega_n) \\ &= 1 * (1-1) / (1-\omega_n) \\ &= 1 * 0 / (1-\omega_n) \\ &= 0 / (1-\omega_n) \\ &= 0 \end{aligned}$$

Product:

The provided solution uses fancy math that I'm not smart enough for. So I just reasoned it out instead.

$$(\omega_n^k)^{-1} * \omega_n^k = \omega_n^{n-k} * \omega_n^k = \omega_n^n = \omega_n^0 = 1$$

For odd, there will be n-1 terms that can be paired and multiplied to 1. That leaves us the ω_n^n term without an inverse.

$$1^{(n-1)/2} * \omega_n^n = 1 * 1 = 1$$

For even, there will be n-2 terms that can be paired and multiplied to 1. That leaves us the $\omega_n^{n/2}$ and ω_n^n terms without an inverse.

$$1^{(n-2)/2} * \omega_n^{n/2} * \omega_n^n = 1 * -1 * 1 = -1$$

2.14 - Remove duplicates from array (Merge Sort no mod)

You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

We will use Merge Sort to sort our array and then remove duplicates from the sorted array.

Step 1:
Perform a Merge Sort on array a and get back a sorted array s.

Step 2:
Scan the sorted array, and from s[2] -> s[n]
If s[i] = s[i-1], remove s[i].

Step 3:
Return the sorted array with duplicates removed.

Correctness:
By first sorting the array with merge sort, we place all possible duplicates directly adjacent with each other. We then scan our sorted array and remove any elements that are copies of its predecessor. This will remove all duplicates.

Runtime:
Merge sort has a known O(n log n) runtime. Our scan of the sorted array takes O(n) and each check of the previous element is O(1).

O(n log n) + ~~O(n) * O(1)~~ = O(n log n)

2.14 - Remove duplicates from array (Merge Sort w/mod)

You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

We will modify Merge Sort to remove duplicates during the merge step.

Step 1:
Perform a Merge Sort on the array, but with a modification on the merge step.

Step 2:

```
During the merge step, for current pointers i and j in merge:
If a comparison results in Left[i] = Right[j]
      Add Left[i] to the merged array as usual.
      Discard Right[j].
Otherwise, complete the adding of an element as usual.
```

Step 3:
Return the sorted array with duplicates removed.

Correctness:
The base case of one element can not have a duplicate.

During a merge step of two sorted subarrays, each minimum element of the left and right arrays are compared. If there is a duplicate, they must be compared to each other at this point. The discarding of one element in a duplicate pair will eliminate all duplicates at each merge step.

When the merge is moved to the next level up, each sorted subarray will have no duplicates. Any duplicates would then only be possible across the two subarrays. Which will then be merged and duplicates eliminated as well. This continues until the full sorted array without duplicates is produced.

Runtime:
The modifications to the merge step is simply to not keep Right[j] if it is equal to Left[i]. This is an O(1) comparison and an O(1) elimination. So the overall runtime is given by the Merge Sort runtime of O(n log n).

Bonus - Jumping Frog (FIB)

(Jumping frog)

The official pet of 6515 is a frog named René, who live in a nice pond at GeorgiaTech. The pond has n rocks in a row numbered from 1 to n, and René is trained to jump from rock i to rock $i+1$ or rock number $i+4$, where $1 \leq i \leq n-1$. Find the number of ways René can go from rock 1 to rock n. Two ways are considered different if they jump on different subsets of rocks.

Example: for $n = 6$ there are three ways to get from rock 1 to rock 6. Those are:
$1 \to 2 \to 3 \to 4 \to 5 \to 6$,
$1 \to 2 \to 6$, and
$1 \to 5 \to 6$.

a. **Define the entries of your table in words. E.g., T(i) or T(i, j) is ...**
For 1 ≤ i ≤ n
T(i) = number of ways René can go from rock 1 to rock i.

b. **State recurrence for entries of your table in terms of smaller subproblems.**

```
T(1) = 1

For 2 ≤ i ≤ n
T(i)        = T(i-1)                if 2 ≤ i ≤ 4
            = T(i-1) + T(i-4)       if 5 ≤ i ≤ n
```

c. **Write pseudocode for your algorithm to solve this problem.**

```
T(1) = 1

for i = 2 -> n:
      T(i) = T(i-1)
      if i ≥ 5:
            T(i) = T(i) + T(i-4)
return T(n)
```

d. **State and analyze the running time of your algorithm.**
2 -> n: O(n)

O(n) = O(n)