CSCE 586 - Design and Analysis of Algorithms
Date: Wednesday 7th November, 2018
Name: Micah Hayden
Assignment: Midterm Exam - Take Home Portion, Version A
Documentation: I worked with 2Lt Mireles on Problem 1 to understand why utilizing a directed graph requires a second data structure to maintain the list of vertices currently in the stack (and thus not with their DFS). I worked with 2Lt Mireles on Problem 2 to understand how the convolution can help produce the result $C(x)$ needed to calculate the force. I also consulted Ref. [4], which included in their matching problem statement a hint for the vector $B(x)$. I worked with 2Lts Mireles, Hanson, and Magness on Problem 3 to understand why the solution needs to start from the final column and work backwards from $j = i = n$ because it is the only day which guarantees to process data. I consulted the algorithms at [1] to brainstorm possible solutions to problems 3 and 4.

Problem 1 - Handout

Problem Statement:

Give an $O(|V| + |E|)$-time algorithm to remove all cycles in a directed graph $G = (V, E)$. Removing a cycle means removing an edge of the cycle. If there are k cycles in G, the algorithm should only remove $O(k)$ edges. You should try to make your algorithm as efficient as possible.

Description of Algorithm:

This algorithm is similar to Tarjan's algorithm for finding strongly connected segments [3]. This algorithm utilizes utilizes a set $Visited = V$ and a stack S initialized as follows: $V \leftarrow \emptyset$ and $S \leftarrow \emptyset$. This algorithm also assumes that graph G is connected, otherwise there will be no way to get from all possible starting nodes v to the remaining $G - v$ nodes.

Algorithm 1 Delete-Cycles(S, u)

```
Push u into S
V ← V ∪ u
for Vertices v_i incident to u do
    if v ∉ V then
        Delete − Cycles(S, v)
    else if v ∈ V and v ∈ S
        Delete edge u → v
    end if
end for
Pop u from S
return
```

Asymptotic Analysis:

Worst Case:

$Delete - Cycles(S, u)$ is a depth-first search that utilizes a stack and a set to check for cycles. Each vertex is visited at most once. At each vertex, each incident edge and respective terminal vertex (v) is checked once: if $v \in V$ and $v \in S$, the edge is deleted; otherwise, v is only visited if $v \notin V$. At each function call of $Delete - Cycles(S, u)$ all operations occur in constant time. Thus, the total running time is $O(|V| + |E|)$, indicating that all edges must be traversed before termination, and visiting each vertex. Because of the stack utilization, deleting a cycle does not add any additional time complexity, it simply deletes the edge creating the cycle. If there are k cycles, there are $O(k)$ delete operations.

Best Case:

The best case would occur when there are no cycles, and the directed graph G was a simple path, where each vertex v has 2 incident edges: one incoming and one outgoing, and the first and last have only a single outgoing and incoming edge, respectively. The total number of vertexes that must be visited is still $\Omega(|V|)$, while the number of edges considered for completing a cycle is $\Omega(|E|)$. Thus, the total best case running time is $\Omega(|V| + |E|)$

Average Case:

Because both the best and worst case run time analysis produces the same bound, the function is tightly bounded by $\Theta(|V| + |E|)$. Thus, the average case will be bounded by the same function.

Proof:

Assume that after running the algorithm, there is some edge $(u \rightarrow v)$ that creates a cycle in G. In order to create a cycle, vertex v must have already been visited from some other node k, with edge $(k \rightarrow v)$, this would have invoked $Delete - Cycles(S, v)$, which would append v to V and into stack S. This would follow path P where $P = k \rightarrow v \rightarrow u$, the path from $k \rightarrow v$, from $v \rightarrow u$, and finally from $u \rightarrow v$, and $|P| \geq 3$. $Delete - Cycles(S, u)$ would encounter vertex v. Because $v \in V$ and $v \in S$, $Delete - Cycles(S, u)$ would have deleted edge $u \rightarrow v$.

Because the algorithm follows a depth-first search, a vertex v is only removed from S if and only if the DFS has returned from each recursive call for node v' incident to v, and each node incident to v', etc. down the depth of the tree. Thus, the algorithm will remove a cycle from G at the first edge that completes the cycle.

If there are k cycles, there are at most k edges that must be deleted from the graph to delete all cycles.

Chapter 5, Problem #4

Problem Statement:

Design an algorithm which computes all forces F_j in $O(n \cdot \log n)$ time.

Description of Algorithm:

This algorithm leverages the gains from utilizing a convolution such as the one in [2] for computing the Fast Fourier Transform (FFT).

Create a vector $A(x)$ where $A_i = q_i \cdot x^{i-1}$ for $1 \leq i \leq n$

Create a vector $B(x)$ where $B(x) = \frac{-1}{n^2} \cdot x^0, \frac{-1}{(n-1)^2} \cdot x^1, \frac{-1}{(n-2)^2} \cdot x^2, \ldots, 0 \cdot x^{n+1}, \ldots, \frac{1}{(n-1)^2} \cdot x^{2n}, \frac{1}{n^2} \cdot x^{2n+1}$ [4].

Determine the product polynomial $C(x)$ using the FFT, computing the convolution of vectors $A(x)$ and $B(x)$ in $O(n \cdot \log n)$ time by **5.15** [2].

Loop through $j = 1 \ldots n$, calculating $F_j = C \cdot q_j \cdot C(x_k)$ where $C(x_k)$ is the term from $C(x)$ with x^k corresponding to j.

Asymptotic Analysis:

Worst Case:
The worst case evaluation of the convolution $C(x)$ is $O(n \cdot \log n)$ by **5.15** in [2]. The evaluation of F_j for $1 \leq j \leq n$ is $O(n)$ because the multiplication occurs in constant time, and there are n values to compute. Thus, the total run time is $O(n \cdot \log n) + O(n) = \boxed{O(n \cdot \log n)}$.
Best Case:
The best case would occur when there was a single particle to consider, $n = 1$. Because a particle cannot exert a force upon itself, $i = j$, the $F_j = F_1 = 0$. This would occur in constant time, $\boxed{\Omega(1)}$. However, for $n > 1$, the result would match the timing of the worst case. The convolution would still need to be calculated, taking $O(n \cdot \log n)$ operations, as well as requiring n operations to calculate F_j, for any $n > 1$. Thus, the algorithm is $\Theta(n \cdot \log n)$.
Average Case:
The average case would produce no change to the worst case time analysis. To compute the convolution would still require $O(n \cdot \log n)$ operations, and calculating F_j for each n would take n operations.

Proof:

To begin, the equation for F_j must be manipulated into a form for which two vectors $\mathbf{a}$ and $\mathbf{b}$ can be convoluted, giving the desired result. When calculating F_j for particle j, j is a constant. Thus, $C \cdot q_j$ can be factored out from the summation.

$$F_j = \sum_{i<j} \frac{C \cdot q_i \cdot q_j}{(j-i)^2} - \sum_{i>j} \frac{C \cdot q_i \cdot q_j}{(j-i)^2}$$
$$F_j = C \cdot q_j \cdot [\sum_{i<j} q_i \cdot \frac{1}{(j-i)^2} - \sum_{i>j} q_i \cdot \frac{1}{(j-i)^2}]$$

The summation can be combined for the convolution, noting that $\sum_{i=j} = 0$. The only remaining step is to determine the two vectors A and B to match the form $\sum_{(i,j):i+j=k} a_i \cdot b_j$. Let $A(x) = q_i \cdot x^{i-1}$, this gives the following for the interior of each summation:

$$Interior = \frac{A(x)}{x^{i-1}} \cdot \frac{1}{(j-i)^2}$$

Let $B(x) = \frac{-1}{n^2} \cdot x^0, \frac{-1}{(n-1)^2} \cdot x^1, \frac{-1}{(n-2)^2} \cdot x^2, \ldots, 0 \cdot x^{n+1}, \ldots, \frac{1}{(n-1)^2} \cdot x^{2n}, \frac{1}{n^2} \cdot x^{2n+1}$
If we denote $(j - i)$ to be the distance between particles i and j, there is at most $n - 1$ distance between the two particles. However, there is a total of $2 \cdot (n - 1)$ possible values: if $j = 1, i = n$, all values are subtracted from F_j; if $j = n, i = 1$, then all values are added to F_j, and if $i = j$, nothing is added or subtracted from F_j. By utilizing $B(x)$ starting at $\frac{1}{n^2}$, it guarantees that there will be a resulting term in $C(x)$ for every combination of i and j.
Vector $A(x)$ is the polynomial representation of the charges q_i, and vector $B(x)$ is a polynomial representation of the distances between two particles.

The polynomial $C(x)$ is the convolution of the two vectors, where a term $c_k = d \cdot x^k$ is multiplied by $C \cdot q_j$, producing a corresponding value for the force F_j.

$$F_j = C \cdot q_j \cdot \frac{c_k}{x^k}$$

Because the two vectors match the form given in the text, the convolution can be utilized for this calculation, proven by **5.15** [2].
Example of Solution: Let $n = 3$, and $q_i = 10, 20, 50$ for $i = 1, 2, 3$.
Polynomial representation of vectors:

$$A(x) = 10 \cdot x^0 + 20 \cdot x^1 + 50 \cdot x^2$$
$$B(x) = \frac{-1}{3^2} \cdot x^0 + \frac{-1}{(3-1)^2} \cdot x^1 + \frac{-1}{(3-2)^2} \cdot x^2 + 0 \cdot x^3 + \frac{1}{(3-2)^2} \cdot x^4 + \frac{1}{(3-1)^2} \cdot x^5 + \frac{1}{3^2} \cdot x^6$$
$$B(x) = \frac{-1}{9} \cdot x^0 - \frac{1}{4} \cdot x^1 - 1 \cdot x^2 + 0 \cdot x^3 + 1 \cdot x^4 + \frac{1}{4} \cdot x^5 + \frac{1}{9} \cdot x^6$$

Hand Calculation of Solution:

$$j = 1$$
$$Sum = \frac{-20}{(1-2)^2} + \frac{-50}{(1-3)^2} = -20 - \frac{50}{4} = \frac{-65}{2}$$
$$j = 2$$
$$Sum = \frac{10}{(2-1)^2} + \frac{-50}{(2-3)^2} = 10 - 50 = -40$$
$$j = 3$$
$$Sum = \frac{10}{(3-1)^2} + \frac{20}{(3-2)^2} = \frac{10}{4} + 20 = \frac{45}{2}$$

Calculated Values using Algorithm:

$$C(x) = A(x) \cdot B(x)$$
$$C(x) = (10 + 20 \cdot x + 50 \cdot x^2) \cdot (\frac{-1}{9} \cdot x^0 - \frac{1}{4} \cdot x^1 - 1 \cdot x^2 + 0 \cdot x^3 + 1 \cdot x^4 + \frac{1}{4} \cdot x^5 + \frac{1}{9} \cdot x^6)$$
$$C(x) = \frac{50 \cdot x^8}{9} + \frac{265 \cdot x^7}{18} + \frac{505 \cdot x^6}{9} + \frac{45 \cdot x^5}{2} - 40 \cdot x^4 - \frac{65 \cdot x^3}{2} - \frac{185 \cdot x^2}{9} - \frac{85 \cdot x}{18} - \frac{10}{9}$$

As shown in the solution, $j = 1, 2, 3$ correspond to $k = 3, 4, 5$. The corresponding values of C_k match the given sums for $j = 1, 2, 3$.

Chapter 6, Problem #9

Problem Statement:

1. Give an example of an instance with the following properties: there is a "surplus" of data in the sense that $x_i > s_1$ for every i; the optimal solution reboots the system at least twice. In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

2. Give an efficient algorithm that takes values for $x_1, x_2, \ldots, x_n$ and $s_1, s_2, \ldots, s_n$ and returns the total $number$ of terabytes processed by an optimal solution.

Part 1: Example

The below example satisfies the requirements given

	Day 1	**Day 2**	**Day 3**	**Day 4**	**Day 5**
x	16	16	16	16	16
s	15	4	3	2	1

On each day, there is a surplus of data $x_i > s_1$. The optimal solution will reboot the system on Day 2 and Day 4, giving a total output of 45 Terabytes.

Description of Algorithm:

This algorithm will use a dynamic programming approach to calculate the optimum value of the number of terabytes processed by the solution.

Let $OPT(i, j)$ refer to the total number of terabytes processed from day i to day n, where j is the number of days since the last reboot.

There are two cases for a given day i:

Case 1: Reboot

$$OPT(i, j) = OPT(i + 1, 1)$$

No work is done on day i, so the total amount of work processed will be starting on day $i + 1$, with $j = 1$.

Case 2: Process Day i

$$OPT(i, j) = OPT(i + 1, j + 1) + min(x_i, s_j)$$

This adds the work done on day i to the total from the following day with $j' = j + 1$ because work was done on day i. Thus, the total number of terabytes processed by the optimal solution will be found at $OPT(1, 1)$.

Algorithm 2 Pseudocode of Algorithm:

for $j = 1 \to n$ **do**
  $OPT(n, j) \leftarrow min(x_n, s_j)$
end for
for $i = n - 1 \to 1$ **do**
  for $j = 1 \to i$ **do**
    $OPT(i, j) = max[OPT(i + 1, 1), min(x_i, s_j) + OPT(i + 1, j + 1)]$
  end for
end for
return $OPT(1, 1)$

Asymptotic Analysis:

Worst Case:

This algorithm has a nested for loop with n loops, where each loop consists of constant time operations. This section runs in $O(n^2)$ time. There is another for loop of n operations to output value for the final day's work. Thus, the algorithm will terminate in $O(n^2) + O(n) \rightarrow O(n^2)$.

Best Case:

Because there are no conditions to exit the algorithm more quickly, this is also the best case running time. $f(x) = O(n^2) = \Omega(n^2)$. Thus, the algorithm is $\Theta(n^2)$.

Average Case:

Because the function is tightly bounded by $\Theta(n^2)$, the average case will also terminate in $\Theta(n^2)$ time.

Proof:

Because there are no days after day $i = n$, the final day will process whatever data it may. Thus, $OPT(n, j)$ is initialized to the minimum of x_i and s_j. Each day prior to day i, there exist two options:

1. **Process Data:** $OPT(i, j) = min(x_i, s_j) + OPT(i + 1, j + 1)$.

2. **Reboot:** $OPT(i, j) = OPT(i + 1, 1)$.

When a reboot occurs, no additional information is processed, but $j = 1$ on the following day (with $i' = i + 1$). When you process on Day i, the total amount processed equals the previous total (found from $i + 1$ and $j + 1$) plus the minimum value of x_i and s_j. The algorithm will iteratively build the solution starting from $OPT(n, j)$ to $OPT(1, 1)$.

Because we start from day $i = n$ and work back to day $i = 1$, adding an optimal (maximum) value at each step, the value of $OPT(1, 1)$ will be the optimal solution.

Chapter 6, Problem #12

Problem Statement:

We want to replicate a file over a collection of n servers, labeled $S_1, S_2, \ldots, S_n$. Placing a file at S_i incurs a *placement cost* of $c_i > 0$. If a user requests the file from server S_i, and no file is present: $S_{i+1}, S_{i+2}, \ldots$ are searched until the file is found at server S_j. This results in a *access cost* of $j - i$.

A *configuration* is a choice, for server S_i with $i = 1, 2, \ldots, n-1$, of whether to place a copy at S_i or not.

The *total cost* of the configuration is the sum of all placement costs for servers with a copy of the file, plus the sum of all access costs associated with all n servers.

Give a polynomial-time algorithm to find a configuration of minimum total cost.

Description of Algorithm:

This algorithm will utilize a dynamic approach to build the optimal solution $OPT(n)$. Note, the final server S_n will always have the file to ensure termination. Let $OPT(i)$ be the minimum cost over servers $1 \to i$.

Algorithm 3 Dynamic Programming for Minimum Server Cost

```
Let OPT(0) = 0 and (1 choose 2) = 0
for 1 ≤ j ≤ (n − 1) do
    OPT(j) ← c_j + min_{0≤i<j}(OPT(i) + (j−i choose 2))
end for
OPT(n) = OPT(n − 1) + c_n.
return OPT(n)
```

Let $OPT(0) = 0$ and $\binom{1}{2} = 0$
for $1 \leq j \leq (n-1)$ **do**
  $OPT(j) \leftarrow c_j + min_{0 \leq i < j}(OPT(i) + \binom{j-i}{2})$
end for
$OPT(n) = OPT(n-1) + c_n$.
return $OPT(n)$

Asymptotic Analysis:

Worst Case:

There are two nested for loops: one looping through j (computing $OPT(j)$), and the other looping through values of $0 \leq 1 < j$. There is a constant amount of work done in each loop, resulting in $O(n^2)$ for the loops. To walk through the values of $OPT(n)$ to output the configuration requires at most n operations, resulting in a total asymptotic run-time of $O(n^2) + O(n) = O(n^2)$.

Best Case:

At best, each server is still considered because it is an iterative solution. There are n outer loops, and $i = j$ inner loops. The total number of inner loops is $\frac{n+0}{2}$, which is equivalent to the average value of j. Thus, at best, the algorithm runs in $\Omega(n \cdot \frac{n}{2} = \Omega(n^2)$.

Average Case:

Because the algorithm is tightly bounded by $f(x) = n^2$, the algorithm is $\Theta(n^2)$, which will be the run time for best, worst, and average.

Proof:

Consider a server i and its respective cases:

1. File placed at S_i: incurs cost of c_i
2. File not at S_i: this incurs the access cost $j - i$, for some $j > i$.

Case 1 is simple to consider, it is simply the placement cost at S_i.
Case 2 requires some additional consideration to get into a workable form for a dynamic programming algorithm. For every server $S_{i'}$ where $i < i' < j$, there will be an access cost of $j - i'$. Assume there is a copy of the file at S_i. Thus,

the below calculation follows:

$$S_i \to S_j = c_i + \sum_{k=i+1}^{j} (j-k) + c_j$$

This sum can be written as the average value of an arithmetic sum with $n = j - i$ terms, $a_1 = (j - i - 1)$, and $a_n = j - j = 0$:

$$\begin{aligned}
Sum &= \frac{(a_n + a_1)}{2} \cdot n \\
&= \frac{(j-i-1)\cdot(j-i)}{2} \\
&= \frac{(j-i)\cdot(j-i-1)}{\cdot(2)} \cdot \frac{(j-i-2)!}{(j-i-2)!} \\
&= \frac{(j-i)!}{(j-i-2)!\cdot 2!} \\
Sum &= \binom{j-i}{2}
\end{aligned}$$

Assume $OPT(i)$ contains the optimal solution up to S_i.

Base Case:

$OPT(1) = c_1$ At the next step, $OPT(2) = c_2 + min(OPT(1) + (2-1))$. This would result in the cost of placing it at S_2, plus the minimum between the cost at S_1 and the access cost to get from S_1 to S_2.

At each step of the algorithm, $OPT(j)$ returns the minimum value of the current placement cost c_j, previous optimal placement $OPT(i)$, and the total access cost from servers $S_{i+1} \to S_j$ The configuration can be found simply by walking through the array of OPT values from $j = n \to j = 1$ in $O(n)$ time.

References

[1] *CS-180 Algorithm Design*. URL: https://github.com/weimin/CS-180.

[2] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson Education. ISBN: 978-93-325-1864-3.

[3] *Tarjan's Algorithm to find Strongly Connected Components*. URL: https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/.

[4] Lenore Zuck. *CS401*. URL: https://www.cs.uic.edu/~i401/ass2-f12-part1.pdf.