CSCI 5454: PS1

Robert Werthman

1.

Let's say these algorithms solve an array sorting problem.

- Let algorithm A be bubblesort with a worst-case runtime of $f(n) = n^2$.
- Let algorithm B be mergesort with a worst-case runtime of $g(n) = n * log(n)$.
- Let C be the newly designed sorting algorithm with a worst-case runtime of $h(n)$.

In this case, $O(min(f(n), g(n)))$ will become $O(n * log(n))$ because it is the smaller of the two runtimes.

If $h(n)$ is $log(n)$ then $h(n)$ achieves the running time $O(min(f(n), g(n)))$ because $log(n)$ does not grow faster than $n * log(n)$ and is therefore bounded above by it.

Yes, you can achieve a running time exactly $min(f(n), g(n))$. Algorithm C would need to be designed in such a way that its running was equal to $min(f(n), g(n))$.

2.

Proposition/Claim: For any real constants a and b, where $b > 0$, the asymptotic relation $(n + a)^b = \Theta(n^b)$ is true.

Theorem: The asymptotic relation $(n + a)^b = \Theta(n^b)$ is true iff:

- There exists positive constants c_1, c_2, n_0 s uch that $0 \leq c_1(n^b) \leq (n + a)^b \leq c_2(n^b)$ for all $n \geq n_0$.

In order to prove the proposition above we must find some constants c_1, c_2, n_0 to satisfy the above bulleted sentence.

Proof:
First we want to find the floor and ceiling of $n + a$ so we can create an inequality similar to the one in the theorem above.

1. If $|a| \leq n$ then we can say that $n + a \leq n + |a| \leq 2n$ (Ceiling of $n + a$).
2. If $|a| \leq \frac{1}{2}n$ then we can say that $n + a \geq n - |a| \geq \frac{1}{2}n$ (Floor of $n + a$).

Now if $2|a| \leq n$ then we can combine the floor and ceilings into an compound inequality that holds true :

$$0 \leq \frac{1}{2}n \leq n + a \leq 2n$$

The only thing missing from this new equation is a power of b. Raising the new equation to a power of b gives:

$$0 \leq (\frac{1}{2}n)^b \leq (n + a)^b \leq (2n)^b \Rightarrow 0 \leq (\frac{1}{2})^b n^b \leq (n + a)^b \leq (2)^b n^b$$

Extracting the constants c_1, c_2, n_0 from this equation yields $c_1 = (\frac{1}{2})^b$, $c_2 = 2^b$, and $n_0 = 2|a|$ since $n \geq 2|a|$. Since we found constants that satisfy the Theorem, the asymptotic relation $(n + a)^b = \Theta(n^b)$ is true.

3.

$f(n) = \Omega g(n)$ means that for all values to the right of some n_0 the value of $f(n)$ is on or above $cg(n)$.

$n!$	e^n	$(\frac{3}{2})^n$	$(lg\, n)!$	n^2	$n\, lg\, n$	$lg(n!)$	n	$(\sqrt{2})^{lg\, n}$	2^{lg*n}	$n^{1/lg\, n}$	1

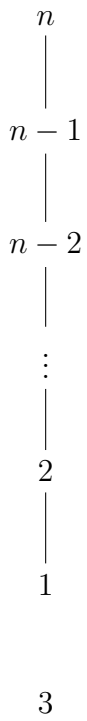
Equivalence Classes

$lg(n!) = \Theta(n\, lg\, n)$
$n^{1/lg\, n} = \Theta(1)$

4.

a.

$T(n) = T(n-1) + n$, $T(1) = 1$
I will use a recurrence tree to solve this recurrence relation.



3

Tree depth = n
Cost per level = i
So $T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$
Therefore, it can be said that $T(n) = O(n^2)$

b.

$T(n) = 2T(n/2) + n^3$, $T(1) = 1$
I will use the master method to solve this recurrence relation.

$a = 2, b = 2, f(n) = n^3$
so $n^{\log_b a} = n^{\log_2 2} = n$
This tells us that the first 2 rules of the master theorem do not apply.

1. $f(n) \neq O(n^{1-\epsilon})$
2. $f(n) \neq \Theta(n)$

This leaves the 3rd rule of the master theorem as the solution.

3. $f(n) = n^3 = \Omega(n^{1+\epsilon})$ if $\epsilon = 1$.
   And $2f(n/2) \leq cf(n) \Rightarrow 2(n/2)^3 \leq cn^3$ if $c = \frac{1}{2}$ and $n \geq 1$.

Therefore, $T(n) = \Theta(n^3)$.

5.

a.

Runtime Analysis: In order to figure out the running time of Insertion Sort we need to add up the cost of each statement in the algorithm.

- If the array is of size n then the statement **for j = 2 to A.length** will execute n times with a cost of c_1.

```
Data: Nearly sorted array of size n integers
Result: Completely sorted array
1 for j = 2 to A.length do
2 |   key = A[j];
3 |   i = j - 1;
4 |   while i > 0 and A[i] > key do
5 |   |   A[i+1] = A[i];
6 |   |   i = i - 1;
7 |   end
8 |   A[i+1] = key;
9 end
```

Algorithm 1: Insertion-Sort(A)

- The statements **key = A[j]** (inserting into an array) and **i=j-1** (setting a variable) will execute $n-1$ times each with a cost of c_2 and c_3 respectively.

- Since k elements are unsorted in this array than any unsorted element is no more than k places away from its sorted position. This means that the statement **while i > 0 and A[i] > key** could be executed in the worst case $\sum_{j=2}^{n} k$ times with a cost of c_4.

- The statements **A[i+1] = A[i]** (inserting into an array) and **i = i + 1** (setting a variable) are executed $\sum_{j=2}^{n} k - 1$ times with a cost of c_5 and c_6 respectively.

- Finally, the statement **A[i+1] = key** (inserting into an array) is executed $n-1$ times with a cost of c_7.

Therefore, the equation for the runtime, $T(n)$, of insertion-sort is:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\sum_{j=2}^{n} k) + c_5(\sum_{j=2}^{n} k - 1) + c_6(\sum_{j=2}^{n} k - 1) + c_7(n-1)$$

$$= c_1 n + c_2(n-1) + c_3(n-1) + c_4(k(n-1)) + c_5(\sum_{j=2}^{n} k - 1) + c_6(\sum_{j=2}^{n} k - 1) + c_7(n-1)$$

Since $k < n$ further reduction of $T(n)$ would yield a linear function of n so we can say the runtime would turn out to be $O(n)$.

b.

The sorting algorithm I suggest to get a $O(n)$ runtime is Counting Sort.

```
   Data: A is the input array of length n
   Data: B is the sorted array of length n
   Data: k is the highest integer in A
 1 let C[0..k] be a new array
 2 for i = 0 to k do
 3 |   C[i] = 0
 4 end
 5 for j = 1 to A.length do
 6 |   C[A[j]] = C[A[j] + 1
 7 end
 8 for i = 1 to k do
 9 |   C[A[j]] = C[i] + C[i − 1]
10 end
11 for j = A.length downto 1 do
12 |   B[C[A[j]]] = A[j]
13 |   C[A[j]] = C[A[j]] − 1
14 end
```

Algorithm 2: Counting-Sort(A, B, k)

Analysis:

- Initializing $C[0..k]$ takes $k + 1$ time to execute and costs c_0.

- The statement **for i = 0 to k** takes $k+1$ times to execute and costs c_1.
- The statements **for j = 1 to A.length** and **j = A.length downto 1** take n times to execute and cost c_3 and c_4 respectively.
- The statement **i = 1 to k** takes k times to execute and costs c_2.

The equation for the runtime, $T(n)$, of Counting Sort is:

$$T(n) = c_0(k+1) + c_1(k+1) + c_3n + c_4n + c_2k \ldots$$

Reducing $T(n)$ further would show that the runtime of Counting Sort is a linear function of n that runs in a linear time of $O(k+n)$. If $k = O(n)$ then the running time is $\Theta(n)$.

c.

(b) doesn't contradict the $\Omega(n\,log\,n)$ lower bound given on page 59 of the textbook because the algorithm is not a comparison sorting algorithm.
It has been proven that any comparison sort must make $\Omega n\,log\,n)$ comparisons in the worst case to sort n elements. Since counting sort is not a comparison sorting algorithm its runtime is not bounded by $\Omega(n\,log\,n)$.

6

Lemma 1: A good minion tells the truth.
Lemma 2: A bad minion could be telling the truth or could be lying.

Let g be the number of good minions, b be the number of bad minions, and n be the total number of minions.

a.

Proposition/Claim : If $n/2$ or more minions are bad, Gru cannot necessarily determine which minions are good.

Proof: This claim is proven by analyzing the cases.

The comparison in the chamber can be between two good minions, two bad minions, or one good and one bad minion.

The claim assumes that $b \geq n/2$.

Case 1: Two good minions size each other up.
The result of that comparison would be:

Minion A	Minion B
good	good

Case 2: Two bad minions size each other up.
The result of the comparison could be:

Minion A	Minion B
good	good
bad	bad
good	bad
bad	good

Case 3: One good minion and one bad minion size each other up.
The result of the comparison if A was good and B was bad could be:

Minion A	Minion B
good	bad
bad	bad

The result of the comparison if A was bad and B was good could be:

Minion A	Minion B
bad	good
bad	bad

Analysis: As can be seen from the cases above, two bad minions sizing each other up in the chamber can lead to the same results as two good minions or one good minion and one bad minion sizing each other up in the chamber. Gru has no way to tell if the results he is seeing in the chamber are from two good minions, two bad minions, or one bad and one good minion.

b.

Proposition/Claim : $n/2$ pairwise tests are sufficient to reduce the problem of finding a single good minion to one of nearly half the size.

The claim assumes that $g > n/2$.

Proof:
$n/2$ pairwise tests means that if n is even, like the number 50, then all minions are tested against another minion. For example, if there are 50 minions then they would be split into 25 (50/2) pairs of minions that will be sent to the chamber to be tested. If n is odd, like the number 51, then one minion will be left out of the initial $n/2$ pairwise tests. For example, if there are 51 minions they would be split into 25 (50/2) pairs of minions with one minion left over and not being paired up.

In order to reduce the problem by nearly half the size we will have to get rid of minions and to do that we need to look at the four possible outcomes of a test. Of the four possible outcomes of minions sizing up each other in the chamber, three have the conclusion that at least one minion in the chamber is bad. With this in mind, only keeping the minions from the outcome that resulted in minion A saying "good" and minion B saying "good" would result in at most one good minion and at least one bad minion being removed for each of the tests that resulted in the other outcomes.

Because we are getting rid of the minions of 3/4 of the possible outcomes of a test the size of the problem, the number of minions, is reduced by at least half if all good minions are tested with bad minions. Additionally, since we get rid of one bad minion with every good minion and more than $n/2$ of the minions are good we know that there will be at least one good minion in the remaining group after the pairwise tests.

c.

Proposition/Claim: Good minions can be identified with $\Theta(n)$ pairwise tests, assuming more than $n/2$ of the minions are good.

Proof: The problem can be categorized as a recurrence starting with n

as the initial size of the problem which is the initial number of minions. n is then continuously divided by 2 each pairwise test iteration as section b states above. We can then create a recurrence equation below which describes this minion testing procedure below:

$$T(n) = T(n/2) + n \tag{1}$$

Part 3 of the master method can be used to solve this recurrence relation. $f(n) = n, b = 2, a = 1$

- $f(n) = n = \Omega(n^{0+\epsilon})$ where ϵ is .5
- $n/2 \leq cn$ where $c = .9$ and all sufficiently large n

Therefore, $T(n) = \Theta(n)$ which proves that the minion testing procedure from b which finds a good minion will only need $\Theta(n)$ pairwise tests to do so.

7

The algorithm I propose for this problem is a divide-and-conquer algorithm.

Data: A, half of table initially covered with n tiles which are all face-up
Data: B, half of table initially empty that holds face-down tiles
Result: n tiles on B part of table which are all face-down

```
// If there are no tiles left on A
if size of A == 0 then
    return
end
// If we have more than one tile on A
if size of A > 0 then
    gather all of tiles on A into bag and spill them onto the table
    find and gather tiles that are face-down and move them to B
    return Tile-Flip(A, B)
end
```

Algorithm 3: Tile-Flip(A, B)

a. Algorithm Proof of Correctness:

Loop Invariant

At the start of each call to this method:

- The first half of the table, A, will contain either tiles that are face-up or no tiles at all.
- The second half of the table, B, will contain either tiles that are face-down or no tiles at all.

At the end of each call to this method:

- B will have grown in size by (size of A)/2 due the expected value of the probability of flipping a tile.

Initialization

Prior to the first ever call to Tile-Flip, A initially consists of n tiles that are all face-up and B is initially empty. The part of the algorithm where probability affects what is in B and A has not been reached. All loop invariants are initially maintained.

Maintenance

To see that consecutive calls to Tile-Flip maintain the loop invariants, we need to look at lines 6-10 of the pseudocode above. Lines 1-5 do not change B or A and so do not affect the loop variants.

Suppose that there are more than 0 tiles remaining on A (the check at line 6). In that case, all the tiles on A will be placed into a bag and spilled onto the table (line 7). If any tiles are found to be face-down they are removed from A and moved to B (line 9). If no tiles are found to be face-down then nothing changes with A or B (line 9). For both cases, the first two of the loop invariants are maintained.

How the third loop invariant is maintained needs more explanation. The probability of flipping a tile from face-down to face-up or face-up to face-down when dumping the bag onto the table is 1/2. Let X be the random

variable that represents the number of face-down tiles in the n tiles that were dumped onto the table, so that

$$X = \sum_{i=1}^{n} X_i$$

We wish to find the **expected number** of tiles dumped on the table that ended up face-down because these tiles would be removed from A and put in B reducing the original problem.

In order to do this we take the expectation of both sides of the equation above and solve. Keep in mind that the expected value of a random variable (in this case the number of face-down tiles) associated with an event (in this case flipping) is equal to the probability that the event occurs. The probability that the flipping of a tile occurs is $1/2$. So $E[X] = 1/2$.

$$\begin{aligned} E[X] &= E[\sum_{i=1}^{n} X_i] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} 1/2 \\ &= n/2 \end{aligned}$$

This means that we expect to find at least half of the tiles that are spilled onto the table to be face-down. Those face-down tiles are moved from A to B and B grows in size by (size of A)/2. This satisfies the third part of the loop invariant so the entire loop invariant is maintained.

A recursive call to Tile-Flip repeats the Tile-Flip algorithm.

Termination

Termination of the algorithm occurs at line 3 of the pseudocode. At that point A has no tiles and all the tiles that it did have were moved to B as face-down tiles (line 7-8). Termination depends on the probability of there being

face-down tiles on the table after a bag has been dumped out. That way A will eventually have not tiles and B will have only face-down tiles which would pass the check on line 2 of the pseudocode and start the return of the rest of the recursive function calls. As was shown in the **Maintenance** section above this will eventually happen. Therefore, the algorithm will always (relying on probability) terminate while maintaining the loop invariants.

b. Algorithm Proof of Runtime

Divide

The divide step in Tile-Flip is line 7-8 of the pseudocode where the tiles are gathered, spilled on the table, and then removed. The size of A is divided by 2 at that point since we assume that $n/2$ tiles will be face-down and will be moved to B. Gathering k tiles and then spilling or moving them onto the table takes n/k time where n is the size of A. This time applies to the initial gathering and spilling and then the gathering and moving to B of the the face-down tiles. This means that runtime for

- Line 7 is

$$\begin{aligned} n/n + (n/2)/(n/2) + (n/4)/(n/4) + ... + 1 &= 1 + 1 + 1 + ... + 1 \\ &= \sum_{i=1}^{lg(n)} 1 \\ &= lg(n) \end{aligned}$$

- Line 8 is

$$\begin{aligned} n/(n/2) + (n/2)/(n/4) + (n/4)/(n/8) + .. + 2 &= 2 + 2 + 2 + ... + 2 \\ &= \sum_{i=1}^{lg(n)} 2 \\ &= 2lg(n) \end{aligned}$$

Adding the runtime of each of these lines together produces $3lg(n)$ for the total runtime of the divide part of this algorithm.

Conquer

We recursively solve the single subproblem created by the divide part of this algorithm when calling Tile-Flip again at line 9 of the pseudocode. The size of each subproblem is $n/2$ where n is the size of A when it enters the function.

Combine

The moving of the face-down tiles together onto the B half of the table is the combine part of this algorithm. This happens at line 8 of the pseudocode has the runtime of $2lg(n)$.

Runtime Equation

Putting together the times from the divide, conquer, and combine part of Tile-Flip into a recurrence, we get the following equation:

$$T(n) = T(n/2) + 3lg(n) = T(n/2) + \Theta(lg(n))$$

The master theorem cannot be used to solve this recurrence because none of the rules of the master theorem apply to this recurrence.

$a = 1, b = 2, f(n) = lg(n)$
$n^{\log_b a} = n^{\log_2 1} = n^0$

There is however a solution that I took out the book Introduction to Algorithms, 3rd Edition:

If $f(n) = \Theta(n^{\log_b a} lg^k n)$, where $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} lg^{k+1} n)$

If $k = 1$ then we have $f(n) = \Theta(n^0 lg(n)) = \Theta(lg(n))$. Since $f(n) = lg(n)$ this statement is true.

This means that $T(n) = \Theta(n^{\log_b a} lg^{k+1} n) = \Theta(lg^2 n)$

This also means that $T(n)$ which is $\Theta(lg^2 n)$ of Tile-Flip is $o(n)$ because any positive polynomial function (n) grows faster than any polylogrithmic function $(lg^2 n)$.

8

Algorithm

Input: A strongly connected directed graph $G = (V, E)$ with $w(u, v) \geq 0$ for each edge $(u, v) \in E$ and node $n_0 \in V$

Output: A matrix M that contains the shortest paths between all pairs of nodes while passing through v_0.

Find-Shortest-Path-To-All-Pairs-Of-Nodes-Through-Single-Vertex(G, v_0)

1. Run the subroutine Dijkstra(G, w, v_0) to find the $\delta(v_0, v)$ (shortest paths) from v_0 to all vertices $v \in V$

2. Transpose $G = (V, E)$ to $G^T = (V, E^T)$, where $E^T = \{(v, u) \in VxV : (u, v) \in E\}$. In other words, reverse the edges of all the vertices in G.

3. Run the subroutine Dijkstra(G^T, w, v_0) to find the $\delta(v, v_0)$ (shortest paths) from all vertices $v \in V$ to v_0

4. For all vertices $v \in V$ and $u \in V$ combine the shortest path from v to v_0 and v_0 to u into a matrix M such that $M_{v,u}$ is the shortest path from vertex v to vertex u that goes through v_0.

Proof Of Correctness

Invariant

The algorithm above returns a matrix that contains the shortest paths between all vertices in the graph G that go through some vertex v_0.

Initialization

Initially we have a strongly connected graph G which means that for every pair of vertices u and v there is an edge from $u \to v$ and an edge from $v \to u$. One vertex v_0 is chosen from G that every shortest path between each pair of vertices in G must go through. The invariant is satisfied because we have not created the matrix M yet to hold the shortest paths.

Maintenance

We assume the correctness of the Dijkstra algorithm and that it returns the shortest paths from a source vertex s to all the vertices a graph. The first call to Dijkstra returns the shortest paths from v_0 to all vertices in G. Since G is strongly connected we know that if we reverse/transpose G (line 2) and then run Dijkstra on v_0 again we can find the shortest paths from all vertices v to the vertex v_0. The invariant is satisfied because the paths found with Dijkstra will be later be used to complete shortest path matrix.

Termination

Our goal is to create a matrix that contains the shortest path from one vertex to another but the path has to run through a single vertex, v_0. Each shortest path for each vertex is found by combining the two outputs of Dijkstra giving shortest paths that look like the following:

$$v \to v_0 \to u \ (u, v) \in V$$

The shortest paths for all pairs of vertices are added to a matrix M where element $M_{v,u}$ is the shortest path from vertex v to vertex u. This matrix is returned and satisfies the invariant stated above.

Proof of Runtime

The runtime for each line of the pseudocode is as follows:

- Line 1: Assuming a binary heap is used to implement the the priority queue of dijkstra it will have a runtime of $O((|V| + |E|)log|V|)$
- Line 2: It takes $O(|V| + |E|)$ to reverse $G = (V, E)$ since we have to visit each vertex and reverse each edge of that vertex.

- Line 3: Assuming a binary heap is used to implement the the priority queue of dijkstra it will have a runtime of $O((|V| + |E|)log|V|)$
- Line 4: Creating and inserting values into a matrix M of $|V|x|V|$ size takes $O(|V|^2)$ time.

So, $T(n)$ (the runtime) of this algorithm is:

$$T(n) = O((|V| + |E|)log|V| + (|V| + |E|) + |V|^2)$$

9

Algorithm

Input: A directed graph $G = (V, E)$; positive edge lengths l_e and positive vertex costs c_v; a starting vertex $s \in V$

Output: An array $cost[]$ such that for every vertex u, $cost[u]$ is the least cost of any path from s to u (i.e., the cost of the cheapest path).

Cheapest-Route(G, s)

1. Create a weight function, w, such that the weight of going from vertex u to vertex v ($w(u, v)$) equals the cost of the edge length between u and v + the cost of vertex v (c_v)
2. Run the Dijkstra subroutine as Dijkstra(G, w, s). According to Figure 4.8 in the Algorithms book this will return an array called $dist[]$ such that $dist[u]$ is the cheapest path from s to u. Set $cost[] = dist[]$.
3. Go through $cost[]$ and add the cost of vertex s (c_s) to each entry.

Proof Of Correctness

Invariant

The algorithm will return an array such that each entry is the least cost of any path from a source vertex s to that entry. The least cost path must include the vertex cost and edge length of every vertex and edge in that path.

Initialization

Initially, we have a directed graph $G = (V, E)$; positive edge lengths l_e and positive vertex costs c_v; a starting vertex $s \in V$. We have all the necessary parts (vertex cost and edge length) to satisfy the invariant.

Maintenance and Termination

We take into account the edge lengths plus the costs of all vertices on the path (not including the source vertex) on line 1. Line 1 creates a new weight function for an edge between two vertices by adding the cost of the end vertex to the cost of the edge length.

We rely on the correctness of the Dijkstra algorithm to find the cheapest path from a source vertex to all the other vertices in the graph using this new weight function.

Line 3 adds the cost of the source vertex s to the cheapest path values found by Dijkstra so both the endpoint vertices costs are now accounted for in the $cost[]$ array and the invariant is satisfied.

Proof of Runtime

The runtime for each line of the pseudocode is as follows:

- Line 1: We have to go through each edge $(u, v) \in E$ and add the cost of vertex v to the edge lengths. This takes $O(|E| + 1)$ time and can be reduced to $O(|E|)$.

- Line 2: Assuming we run Dijkstra with binary heap as the priority queue this takes $O((|V| + |E|)log|V|)$ time.
- Line 3: Going through the array $cost[]$ will take $O(|V|)$ time because that array will contain an entry for all the vertices from the graph G.

So, $T(n)$ (the runtime) of this algorithm is:

$$T(n) = O((|V| + |E|)log|V| + |E| + |V|)$$