Name: Keaton Whitehead

ID: 104 668 391

CSCI 3104, Algorithms **Profs. Grochow & Layer**

Problem Set 3 **Spring 2019, CU-Boulder**

Hyperlinks for convenience: 1a 1b 1c 2a 2b 2c 3a 3b 3c

1. *(10 pts total) For parts* (1a) *and* (1b)*, justify your answers in terms of deterministic QuickSort, and for part* (1c)*, refer to Randomized QuickSort. In both cases, refer to the versions of the algorithms given in the lecture notes for Week 3.*

   (a) *(3 points) What is the asymptotic running time of QuickSort when every element of the input A is identical, i.e., for $1 \leq i, j \leq n$, $A[i] = A[j]$? Prove your answer is correct.*

If all of the inputs in A are identical, this will result in a maximally unbalanced tree. This is because the Partition function will divid the array into $O(1)$ and $O(n)$ elements. We know that the recursion cost of QuickSort is always $f(n) = \Theta(n)$ because this is the cost of calling Partition. We can then use the worst-case-recurrence-relation equation we learned in class which is:

$$T(n) = T(n-1) + T(1) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

In this equation $T(n)$ is considered tail recursive which is logically equivalent to a for loop. Thus, we can solve it using the unrolling method.

$$T(n) = T(n-1) + \Theta(n)$$
$$T(n) = T(n-2) + \Theta(n-1) + \Theta(n)$$
$$T(n) = T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n)$$
$$\vdots$$
$$T(n) = \Theta(1) + \dots + \Theta(n-1) + \Theta(n)$$
$$T(n) = \sum_{i=1}^{n} \Theta(i)$$
$$T(n) = \Theta(n^2)$$

This is true because this is the formula for the worst case of QuickSort and we can compare this to QuickSort's worst case because having all 1 elements that are identical is just like having all elements in a sorted ascending order. This can be said because of the if statement within the partition function.

Name: Keaton Whitehead

ID: 104668391

CSCI 3104, Algorithms **Profs. Grochow & Layer**

Problem Set 3 **Spring 2019, CU-Boulder**

(b) *(3 points) Let the input array A be $[2, 1, -1, 4, 5, -4, 6, -3, 3, 0]$. What is the number of times a comparison is made to the element with value -3 (note the minus sign)?*

The best way to test this is by actually stepping through the algorithm. So let's begin:

$A = [2, 1, -1, 4, 5, -4, 6, -3, 3, \textcircled{0}]$ ↑ Pivot

* The Pivot is $A[9] = 0$

$A[x] \leq 0$ $\quad$ $A[x] > 0$

$A_1 = [-1, -4, -3]$ ↑ Pivot $\quad$ $A_2 = [2, 1, 4, 5, 6, 3]$ ↑ Pivot

+1 * Here we have one comparison to -3

$A_1[x] < -3$ $[\,]$

$A_1[x] > -3$ $[-1, -4]$ ↑ Pivot

$A_2[x] \leq -3$ $[2, 1]$

$A[x] > -3$ $[4, 5, 6]$

+2 * Here we have -3 as a pivot and we get two comparisons

- We can stop here because all other comparisons that follow do not contain the value -3 or are compared to it.
- As you can see we have 3 total comparisons to -3.

Name: Keaton Whitehead

ID: 104668391

CSCI 3104, Algorithms **Profs. Grochow & Layer**

Problem Set 3 **Spring 2019, CU-Boulder**

(b) *(3 points) Let the input array A be $[2, 1, -1, 4, 5, -4, 6, -3, 3, 0]$. What is the number of times a comparison is made to the element with value -3 (note the minus sign)?*

The best way to test this is by actually stepping through the algorithm. So let's begin:

$A = [2, 1, -1, 4, 5, -4, 6, -3, 3, \textcircled{0}]$ ↑ Pivot

* The Pivot is $A[9] = 0$

$A[x] \leq 0$ ↙ ↘ $A[x] > 0$

$A_1 = [-1, -4, -3]$ (Pivot) $A_2 = [2, 1, 4, 5, 6, 3]$ (Pivot)

+1 * Here we have one comparison to -3

$A_1[x] < -3$ $[\,]$

$A_1[x] > -3$ $[-1, -4]$ ↑ Pivot

$A_2[x] \leq -3$ $[2, 1]$

$A[x] > 3$ $[4, 5, 6]$

+2 * Here we have -3 as a pivot and we get two comparisons

- We can stop here because all other comparisons that follow do not contain the value -3 or are compared to it.
- As you can see we have 3 total comparisons to -3.

Name: Keaton Whitehead

ID: 104 666 391

CSCI 3104, Algorithms **Profs. Grochow & Layer**
Problem Set 3 **Spring 2019, CU-Boulder**

(c) *(4 points) How many calls are made to* `random-int` *in (i) the worst case and (ii) the best case? Give your answers in asymptotic notation.*

Random QuickSort and the normal Quicksort Method have the exact same algorithm with one exception, the logic behind determining the pivot value in the Partition function. For Quick Sort the worst case scenario would be either all input in a sorted accending order or if all values are the same (as proved earlier). So because this is essentially the same as Random Quick Sort, we can assume with caution that it must be the same for Random Quick Sort's worst-case. This is true for all values of the array being the same but not true for the values being in ascending sorted order. Remember, the difference for Random Quick Sort is the selection of the pivot, so once the pivot is selected and moved to the last element we might end up with an average case. That's not what this question is asking so I want go any further with trying to prove this case to be true, although it can be done. You just have to pick the very specific order for the array in such a way that after swapping the pivot to the back and stepped into smaller arrays, that you end up forming the ordered array, thus inadvertenly creating the worst case possible of being in ascendeding sorted order.

But back to the much easier case of proving it with all values that are identical. The same ideology can be applied as what was proved in 1a. Regardless of what value that is selected as the pivot, will always equate to the value it is being compared to so we will again end up with the same maximum unbalanced tree. Now knowing this we can use this tree to determine how many times random-int() is called. Random-int() resides within the Partition function and is called once at the very begining. So, there is a direct 1:1 ratio of Partion called to randomInt() being called.

I'll illustrate the most unbalanced tree to help depict this (Figure 1):

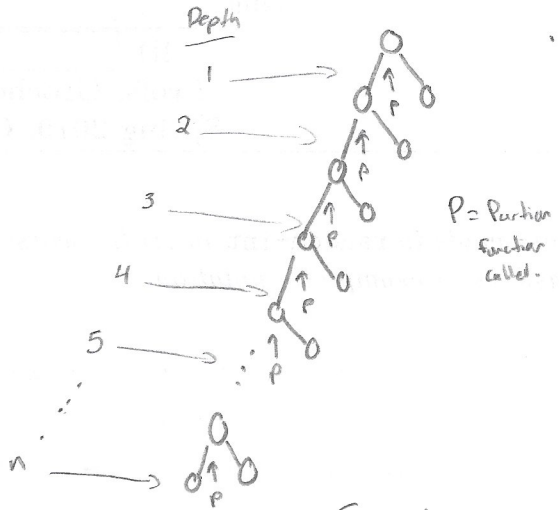

Figure 1

- We know from class that the worst case for Quick Sort has a depth of $\Theta(n)$, which tells us that there are $\Theta(n)$ number of times partitions are called. You can look at the labeling as well to confirm this. Therefore we can conclude that at depth $\Theta(n)$ we can see there are $\Theta(n)$ of partition functions called. Which, because of it's 1:1 ratio with Random-Int() we can safely assume there are $\Theta(n)$ of Random-Int()s called.

Worst Case : $\Theta(n)$

To prove the best case for Random-QuickSort we will also compare it to the best case for Quick Sort. So, QuickSorts best case happens when a perfect, evenly filled binary tree is a result which occures when the array Partitions in half perfectly each iteration. We learned this in class. So using this fully balanced binary tree as a comparison we can assume that this will be the same out come for Randomized-Quick Sort as well, because it is the same algorithm only the pivot selection differs (which means that the array would have to defer from the Quick Sort algorith to account for all of the pivot switching involved to result in the perfectly balanced tree, but I know you aren't asking for that so I won't go into too much detail. It can be done, I think it would be a little too much to try and go into that... anyways...). The ratio for the Partition calls and calls of random-Int() still holds true for the best case, but we will come back to that. Going off the structure of the Know Quick Sort fully balanced tree we know the depth is $\Theta(\log n)$ (we learned this in class). But the amount of Partition calls is not $\Theta(\log n)$, rather we have to account for the amount of times Partition is called within each layer. With the diagram Figure 2. we can see that there are 2^K calls of Partitions where K = the layer. Now we want to add up all the Partitions in the entire tree so we do a summation of $\sum_{i=0}^{k} 2^i$ and this equal to $2^{k+1}-1$ (because Google lol). So $2^{k+1}-1$ tells us how many Partitions there are and $\Theta(\log n)$ tells us how many layers there are given a n. So let's substitute the layer equation in with the # of Partitions and we get $2^{(\log_2 n)+1} - 1 = 2^{\log_2 n} \cdot 2 - 1 = 2n - 1 = \Theta(n)$

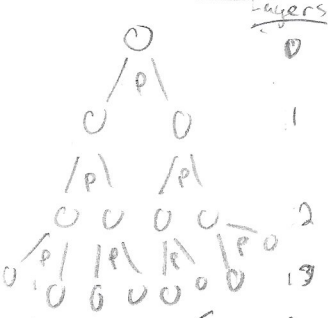

Figure 2

Best case: $\Theta(n)$

Because Partition has a 1:1 ratio to the # of times random-Int() is called.

Name: Keaton Whitehead

ID: 104 668 391

CSCI 3104, Algorithms **Profs. Grochow & Layer**
Problem Set 3 **Spring 2019, CU-Boulder**

2. *(20 pts total) Professor Flitwick needs your help. He gives you an array A consisting of n integers $A[1], A[2], \ldots, A[n]$ and asks you to output a two-dimensional $n \times n$ array B in which $B[i,j]$ (for $i < j$) contains the sum of array elements $A[i]$ through $A[j]$, i.e., $B[i,j] = A[i] + A[i+1] + \cdots + A[j]$. (The value of array element $B[i,j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what the output is for these values.)*

Flitwick suggests the following simple algorithm to solve this problem:

```
flitwickSolve(A) {
   for i = 1 to n {
      for j = i+1 to n {
         s = sum(A[i..j])        // look very closely here
         B[i,j] = s
}}}
```

(a) *(5 points) Choose a function f such that $\Omega(f(n))$ serves as a bound on the running time of this algorithm, and prove that this is the case—that is, prove that the worst-case running time is at least $\Omega(f(n))$ on inputs of size n.*

A function that would bound on the running time of the algorithm is $f(n) = n^3$ so $\Omega(n^3)$, which also happens to be $\Theta(n^3)$ because it is a tight bound.

We can see this is true if we use this summation.

$$\sum_{i=0}^{n} \sum_{j=i+1}^{n} (j-i)+1 = \Theta(n^3)$$

(b) *(5 points) For this same function f you chose in part 2a, show that the running time of the algorithm on any input of size n is also $O(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)*

Look at 2(a), it is the same, $\Theta(n^3)$

Name: Keaton Whitehead

ID: 104666391

CSCI 3104, Algorithms **Profs. Grochow & Layer**

Problem Set 3 **Spring 2019, CU-Boulder**

(c) *(10 points) Although Flitwick's algorithm is a natural way to solve the problem—after all, it just iterates through the relevant elements of B, filling in a value for each—it contains some highly unnecessary sources of inefficiency. Give an algorithm that solves this problem in time $O(f(n)/n)$ (asymptotically faster than before) and prove its correctness. (Recall the class conventions on what is required when we ask you to "give an algorithm.")*

So $F(n) = O(n^3)$ and we now want to perform it with $O(n^2)$ now instead. To make this happen we have to rewrite the algorithm to:

```
FlitwickSolve (A){
    For i = 1 to n {
        For j = i+1 to n {
            if (j - i == 1) {
                B[i, j] = A[j-1] + A[j]
            }
            else {
                B[i, j] = B[i, j-1] + A[j]
            }
        }
    }
}
```

Idk how to prove the correctness for a nested for loop

Name: _____

ID: _____

CSCI 3104, Algorithms **Profs. Grochow & Layer**
Problem Set 3 **Spring 2019, CU-Boulder**

3. *(30 pts total) The Dementors have captured n prisoners, of dubious morals and dubious reliability. The Ministry of Magic needs your help to identify which prisoners can be relied on to tell the truth, and which cannot. To this end, they have developed a spell that can be cast on a pair of prisoners at a time. When the spell is cast, the two prisoners each say whether the other prisoner is truthful or a liar. A truthful prisoner always gives the correct answer—that is, correctly identifies whether the* other *prisoner is truthful or a liar—but anything one of the lying prisoners says cannot be trusted. That is, lying prisoners can lie whenever they want, but they* don't *necessarily always make false statements. Furthermore, the spell has the useful property that if it is cast on the same two prisoners repeatedly, they will always give the same answers they gave initially (hence why a spell is needed).*

You quickly notice that there are four possible outcomes to the spell, when cast on prisoners i and j:

prisoner i says	*prisoner j says*		
"j is a liar"	*"i is a liar"*	$\Longrightarrow$	*at least one is a liar*
"j is a liar"	*"i is truthful"*	$\Longrightarrow$	*at least one is a liar*
"j is truthful"	*"i is a liar"*	$\Longrightarrow$	*at least one is a liar*
"j is truthful"	*"i is truthful"*	$\Longrightarrow$	*both truthful, or both liars*

(a) *(10 points) Prove that if n/2 or more prisoners are liars, the Ministry cannot necessarily determine which prisoners are truthful using* any *strategy based on this kind of pairwise test. Assume a worst-case scenario in which the lying prisoners contain a psychic link that they can use to collectively conspire to fool the Ministry.*

T T T T

T F T F

T F F T

F F T T

I dk

(b) *(12 points) Suppose the Ministry knows that strictly more than $n/2$ of the prisoners are truthful, but not which ones. Prove that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.*

(c) *(8 points) Now, under the same assumptions as part* (3b)*, prove that all of the truthful prisoners can be identified with* $\Theta(n)$ *pairwise tests. Give and solve the recurrence (as always, in* O *or* Θ *notation) that describes the number of tests.*

IDK