Name: Vera Duong

ID: 109431166

**CSCI 3104, Algorithms** **Due Feb 19, 2021**

**Problem Set 05 (50 points) Spring 2021, CU-Boulder** Collaborators: Nathan Straub, Matt Hartnett

---

*Advice 1*: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.
*Advice 2*: Verbal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

**Instructions for submitting your solution**:

- The solutions **should be typed** and we cannot accept hand-written solutions. Here's a short intro to Latex.
- You should submit your work through **Gradescope** only.
- The easiest way to access Gradescope is through our Canvas page. There is a Gradescope button in the left menu.
- Gradescope will only accept **.pdf** files.
- It is vital that you match each problem part with your work. Skip to 1:40 to just see the matching info.

---

Name: Vera Duong
ID: 109431166

**CSCI 3104, Algorithms** **Due Feb 19, 2021**
**Problem Set 05 (50 points) Spring 2021, CU-Boulder** Collaborators: Nathan Straub, Matt Hartnett

1. *(16 pts) Suppose in quicksort, we have access to an algorithm which chooses a pivot such that, the ratio of the size of the two subarrays divided by the pivot is a* **constant** *$k$. i.e an array of size $n$ is divided into two arrays, the first array is of size $n_1 = \frac{nk}{k+1}$ and the second array is of size $n_2 = \frac{n}{k+1}$ so that the ratio $\frac{n_1}{n_2} = k$ a constant.*

   (a) *(3 pts) Given an array, what value of $k$ will result in the best partitioning?*

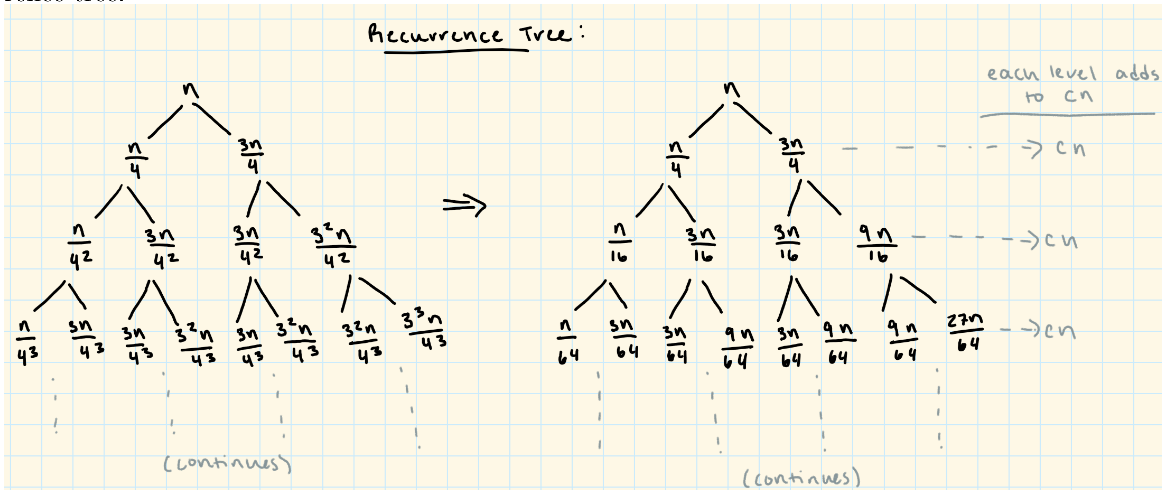
   (b) *(10 pts) Write down a recurrence relation for this version of QuickSort, and solve it asymptotically using* **recursion tree** *method to come up with a big-O notation. For this part of the question assume $k = 3$. Show your work, write down the first few levels of the tree, identify the pattern and solve. Assume that the time it takes to find the pivot is $\Theta(n)$ for lists of length $n$. Note: Remember that a big-O bound is just an upper bound. So come up with an expression and make arguments based on the big-O notation definition.*

   (c) *(3 pts) Does the value of $k$ affect the running time?*

**Solution:**

(a) QuickSort has its best case when the pivot is in the middle, so $n_1$ and $n_2$ have the same size where the size of the first half of the array is the same as the size of the second half of the array. Thus, the value of $k = 1$ will have the best partitioning. In this way, $n_1 = \frac{n}{2}$ and $n_2 = \frac{n}{2}$.

(b) The recurrence relation for this version of QuickSort is $T(n) = T(\frac{nk}{k+1}) + T(\frac{n}{k+1}) + \Theta(n)$. Since $k = 3$, our recurrence relation would look like this: $T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + \Theta(n)$. Below is the recurrence tree:



For each side of the tree, we have that:
Left: $n/4^k = 1 \Rightarrow n = 4^k \Rightarrow \log_4 n = k$
Right: $n(3/4)^k = 1 \Rightarrow n = (4/3)^k \Rightarrow \log_{(4/3)} n = k$

We are looking for an upper bound in this case, so Right > Left.
Total Cost = (Num of Levels) · (cost per level)
In this case, the number of levels is $\log_{4/3} n + 1$ since the addition of 1 accounts for the root and that $\log_{4/3}(1) = 0$
Total Cost = $(\log_{4/3} n + 1) \cdot (cn) = cn \log_{4/3} n + cn$

Thus, the recurrence $T(n)$ is $\mathcal{O}(n \log n)$.

(c) The value of $k$ does not affect the running time. In our case, for problem b, we used $k = 3$ and plugged that into our recurrence relation. However, if we picked any other $k$ value, it would split differently than how it was done in part b, but in the end each level would still add up to $cn$. We will end up with a log with a different base, but no matter what the recurrence $T(n)$ would still be $\mathcal{O}(n \log n)$ when looking at the upper bound for our QuickSort.

2. *(10 pts) Consider a chaining hash table $A$ with $b$ slots that holds data from a fixed, finite universe $U$.*
   (a) *(3 pts) State the simple uniform hashing assumption.*
   (b) *(7 pts) Consider the worst case analysis of hash tables. Suppose we start with an empty hash table, $A$. A* ***collision*** *occurs when an element is hashed into a slot where there is another element already. Assume that $|U|$ represents the size of the universe and $b$ represents the number of slots in the hash table. Let us assume that $|U| \leq b$. Suppose we intend to insert $n$ elements into $A$* ***Do not assume the simple uniform hashing assumption for this subproblem.***
      i. *What is the worst case for the number of collisions? Express your answer in terms of $n$.*
      ii. *What is the load factor for $A$ in the previous question?*
      iii. *How long will a successful search take, on average? Give a big-Theta bound.*

**Solution:**

(a) The simple uniform hashing assumption is the assumption or goal that any given element has an equal probability to hash into any of the slots $b$ of hash table $A$ from the fixed, finite universe $U$ and will evenly distribute items into the slots, independently of where any other element has placed.

(b$i$) The worst case for the number of collisions for $n$ elements is $n - 1$. According to the definition provided, a collision occurs when an element is hashed into a slot where there is another element already. In the worst case, all elements $n$ will hash to the same slot. So for example, if there were 5 elements, there would be 4 collisions. We would start counting the number of collisions once the second element is inserted into that same spot since the first one is already there.

(b$ii$) The load factor gives an idea of how likely collision resolution is needed when a new element is inserted into the table and also tells us something about how long search will take. The load factor ($\alpha$) is the number of elements divided by the number of buckets. In the worst case, $\alpha = \frac{n}{1}$ because in this instance only one slot is being used for the elements to be inserted into for all values. $\alpha = \frac{n}{b}$ for the best case.

(b$iii$) A successful search will on average take $\Theta(1 + \alpha)$ where $\alpha$ is the load factor. This is all under the assumption of simple uniform hashing.

3. *(12 pts) Consider a hash table of size 100 with slots from 1 to 100. Consider the hash function $h(k) = \lfloor 100k \rfloor$ for all keys $k$ for a table of size 100. You have three applications.*

   - ***Application 1:*** *Keys are generated uniformly at random from the interval* $[0.3, 0.8]$.
   - ***Application 2:*** *Keys are generated uniformly at random from the interval* $[0.1, 0.4] \cup [0.6, 0.9]$.
   - ***Application 3:*** *Keys are generated uniformly at random from the interval* $[0.01, 1.01)$.

   (a) *(3 pts) Suppose you have $n$ keys in total chosen for each application. What is the resulting load factor $\alpha$ for each application?*

   (b) *(3 pts) Which application will yield the worst performance?*

   (c) *(3 pts) Which application will yield the best performance?*

   (d) *(3 pts) Which application will allow the uniform hashing property to apply?*

**Solution:**
(a) The load factor is $\alpha$ = num elements / num slots potentially allocated

Application 1: $\alpha = \frac{n}{51}$ The number of slots potentially allocated is 51 since the range of [0.3, 0.8] has a difference of 0.5, and when you multiply that by 100 we get 50 slots that can possibly be inserted into the hash table. However, since the slots are inclusive, we add one to 50.

Application 2: $\alpha = \frac{n}{62}$ The number of slots potentially allocated is 62 since the range of $[0.1, 0.4] \cup [0.6, 0.9]$ can go into 30 slots for the first range and another 30 for the second, so it can possibly be inserted into 62 buckets in the hash table since it's bounds are inclusive so we add two to 60.

Application 3: $\alpha = \frac{n}{100}$ The number of slots potentially allocated is 100 since the range of [0.01, 1.01) can possibly be inserted into 100 buckets of the hash table.

(b) The application that will yield the worst performance is **Application 1**. This is because the range from [0.3, 0.8] is the smallest compared to the other two applications. The hash function $h(k)$ is written so that 100 random keys $k$ are chosen within the application range and is multiplied by 100, and its slot will be the floor of whatever it returns. By having a smaller range of values, it's more likely that real numbers will be chosen "close" together and hash to the same bucket(s). When this happens, collisions would be created and would then lead to a worse performance the more collisions there are. Thus, there is more limitation to what slots the key will hash to with Application 1.

(c) The application that will yield the best performance is **Application 3**. This is because the range from [0.01, 1.01) is the largest compared to the other 2 applications. Due to the fact that it has a large range, we would have a greater scope of real numbers $k$ that can be randomly chosen. It has a much higher chance to hash with less collisions by having a wider range since the keys will be more likely to be further apart which will then be less likely to hash to the same bucket(s) which causes collisions. The less collisions there are, the better performance we will get.

(d) The application that will allow the uniform hashing property to apply is **Application 3**. This is because its possible slots go from 1 to 100, while Application 1 goes from 30 to 80 and Application 2 goes from 10 to 40, and 60 to 90. This is proven since the UHP states that any given element has an equal probability to hash into any of the slots of the hash table as well as evenly distributing items into the slots independently of where any other element has placed. Application 3 will likely be more evenly distributing of the slots since their possible ones (calculated from the range of [0.01, 1.01)) match with the actual slots of 1 to 100 in the hash table and that each key has an equal chance to hash into any of the buckets.

4. *(12 pts) Median of Medians Algorithm*

   (a) *(4 pts) Illustrate how to apply the QuickSelect algorithm to find the $k = 4$th smallest element in the given array:* `A = [5, 3, 4, 9, 2, 8, 1, 7, 6]` *by showing the recursion call tree. Refer to Sam's Lecture 10 for notes on QuickSelect algorithm works*

   (b) *(4 pt) Explain in 2-3 sentences the purpose of the Median of Medians algorithm.*

   (c) *(4 pts)Consider applying Median of Medians algorithm (A Deterministic QuickSelect algorithm) to find the 4th largest element in the following array:* `A = [6, 10, 80, 18, 20, 82, 33, 35, 0, 31, 99, 22, 56, 3, 32, 73, 85, 29, 60, 68, 99, 23, 57, 72, 25]`*.Illustrate how the algorithm would work for the first two recursive calls and indicate which sub array would the algorithm continue searching following the second recursion. Refer to Rachel's Lecture 8 for notes on Median of Medians Algorithm*

**Solution**:

**(a)** I will be using Sam's QuickSelect algorithm for this problem. According to Sam's lecture 10, $k$ starts at 0, so to find the $4^{th}$ smallest value in array $A$, we need to use $k = 3$ to be inserted as a parameter of QuickSelect(A, k). We can see that the length of the array is not equal to 1, so we don't need to worry about the first if statement. Using Google's random number generator with ranges 1-9, I was given the random value of 5, so we set our pivot equal to 5. Lowers will then be [1, 2, 3, 4] and highers will set to [6, 7, 8, 9]. Next, we compare $k = 3$ to see if it's less than the length of lowers (4), which it is. So we will do a recursive call of QuickSelect(lowers, k) where lowers is [1, 2, 3, 4] and $k$ remains the same which is 3.

In this recursive call, the length of the array is still greater than one, so we can ignore the first if statement. To find the pivot value Google's random number generator gave me a value of 2, so the pivot is equal to 2. Lowers will set to [1] since it's the only value less than 2 in the array, and highers is set to [3, 4]. The first if statement compares $k < length(lowers)$, or if $3 < 2$. This is not the case, so we move on to the else if. This compares if $k < length(lowers) + 1$, or if $3 < 3$, which it is not, so we move on to the else and call QuickSelect(highers, k-1-length(lowers)) where highers is [3, 4] and the second parameter is 1 from doing 3-1-1.

In this last recursive call, the given array is of length 2, so we ignore the first if statement. To find the pivot value Google's random number generator gave me a value of 4, so the pivot is equal to 4. Lowers will set to [3] since it's the only value less than 4 in the array, and highers is set to [] since there is no number higher than 4 in this call. The first if statement compares $k < length(lowers)$, or if $1 < 1$. This is not the case, so we move on to the else if. This compares if $k < length(lowers) + 1$, or if $1 < 2$, which is true. Since it's true, we will return the pivot value, 4. Thus, we've shown how to apply Sam's QuickSelect Algorithm to find the $4^{th}$ smallest value using $k = 3$ in array $A$, which is 4.

**(b)** The purpose of the Median of Medians algorithm is that it's a pivot selection strategy for selection algorithms, such as QuickSort or QuickSelect which deterministically picks a better pivot rather than picking one randomly or the last element of an array. This allows for the worst case complexity of QuickSelect to reduce significantly because it's worst case complexity becomes linear rather than it's worst case quadratic time due to poor pivot choices.

**(c)** For the Median of Medians algorithm we first section the array in groups of 5:

$$A = [6, 10, 80, 18, 20|82, 33, 35, 0, 31|99, 22, 56, 3, 32|73, 85, 29, 60, 68|99, 23, 57, 72, 25]$$

Next, we sort each section from least to greatest:

$$A = [6, 10, 18, 20|0, 31, 33, 35, 80|3, 22, 32, 56, 99|29, 60, 68, 73, 85, |23, 25, 57, 72, 99]$$

Next, we find the median within each section, or the middle element:

$$A = [6, 10, \boxed{18}, 20|0, 31, \boxed{33}, 35, 80|3, 22, \boxed{32}, 56, 99|29, 60, \boxed{68}, 73, 85, |23, 25, \boxed{57}, 72, 99]$$

Out of the five medians we found 18, 32, 33, 57, and 68, the median of those is 33.

When applying this to Sam's QuickSelect algorithm, since we are looking for the 4th largest element, our $k = 21$. This is because this is also known as the 22nd smallest element. We can see that the length of the array is not equal to 1, so we don't need to worry about the first if statement. When finding the Median of Medians, we are using this as our initial pivot which is 33. Lowers is [0, 3, 6, 10, 18, 20, 22, 23, 25, 29, 31, 32] and highers is [35, 56, 57, 60, 68, 72, 73, 80, 82, 85, 99, 99]. The next if statement that applies is the else return QuickSelect(highers, k-1-length(lowers)). The second parameter is 8 from the calculation of 21-1-12.

In the first recursive call, we can see that the length of the array is not equal to 1, so we don't need to worry about the first if statement. We next do the Median of Medians algorithm to find the pivot value on highers which was passed in as a parameter. Splitting the highers array we passed in as the parameter into 3 sections we get:

$$[35, 56, 57|60, 68, 72|73, 80, 82|85, 99, 99]$$

Next, we find the middle element in each section:

$$[35, \boxed{56}, 57|60, \boxed{68}, 72|73, \boxed{80}, 82|85, \boxed{99}, 99]$$

Since there's an even number of medians, we find the average between the 2 in the middle by doing $\frac{68+80}{2} = 74$. Our pivot in this call will be 74. Lowers will be [35, 56, 57, 60, 68, 72, 73] and highers is [80, 82, 85, 99, 99]. The if statement that we take into account is else return QuickSelect(highers, k-1-length(lowers)). The second parameter is 0 from the calculation of 8-1-7.

In the second recursive call, we can see that the length of the array is not equal to 1, so we don't need to worry about the first if statement. We next do the Median of Medians algorithm to find the pivot value on highers which was passed in as a parameter. We can see that the highers array is an odd number with 5 elements, so we can find the median which is 85 from [80, 82, $\boxed{85}$, 99, 99]. We set our pivot to 85. Lowers is [80, 82] and highers is [99, 99]. We can see that $k < length(lowers)$ which is $0 < 2$, so we would do another call to QuickSelect(lowers, k). If we were to continue the recursion, we would get that the 4th greatest element is 80 in the array $A$.

I've thus shown 2 recursive calls to Sam's Quicksort utilizing Median of Medians to find the pivots as well as which subarray we would use to continue searching.