Name: Chirag Telang

ID: 109410601

Due February 12, 2021

CSCI 3104, Algorithms **Spring 2021, CU-Boulder**

Problem Set 4 (50 points) **Collaborators: Peers on Discord**

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: Chirag Telang
ID: 109410601

Due February 12, 2021

CSCI 3104, Algorithms **Spring 2021, CU-Boulder**

Problem Set 4 (50 points) **Collaborators: Peers on Discord**

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.*

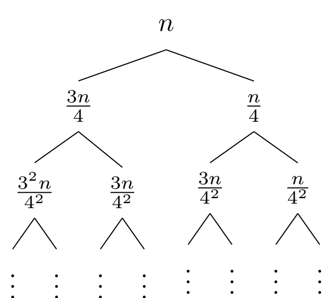
   (c) *(3 pts) Does the value of k affect the running time?*

Solution:

(a) The best partitioning will occur when $k = 1$. This is the case because when $k = 1$, the two subarrays would be evenly divided – so, $n_1 = \frac{n}{2}$ and $n_2 = \frac{n}{2}$.

(b) $T(n) = \begin{cases} T(\frac{nk}{k+1}) + T(\frac{n}{k+1}) + c(n) & \text{when } n > 1, \\ 1 & \text{when } n = 1 \end{cases}$

For $k = 3$,



When $\frac{3^k n}{4^k} = 1$, the first subarray bottoms out. Therefore, solving for k, $k = \frac{log(1/n)}{log(3/4)} = log_{\frac{3}{4}}(1/n)$

When $\frac{n}{4^k} = 1$, the second subarray bottoms out. Therefore, solving for k, $k = \frac{log(n)}{log(4)} = log_4(n)$

(c) Since the time complexity will be asymptotically equal, the value of k does not affect the running time

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 states that a hypothetical hashing function will evenly distribute items into the slots of a hash table – so that every slot has an equal chance of being "picked" by the key. In this case, the probability of any slot getting picked is $1/b$.

(b) i. The worst case for the number of collisions is $n - 1$ collisions.

   ii. The load factor for A is $n/1$ which is n – since, in the worst case scenario, all the insertions are going into the same slot.

   iii. On average, a successful search will be $\theta(n)$ – since, in the worst case scenario, the search time is proportional to the length of the list.

Name: Chirag Telang
ID: 109410601

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* $[0, 1]$.

(a) *(3 pts) Suppose you have* n *keys in total chosen for each application. What is the resulting load factor* α *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) **Application 1:** From the interval $[0.3, 0.8]$, $b = 51$ since there can be a total of 51 different keys that can be generated. As a result, the load factor is $\alpha = n/b = n/51$.

Application 2: From the intervals $[0.1, 0.4]$ and $[0.6, 0.9]$, $b = 62$ since there can be a total of 62 different keys that can be generated between the two. As a result, the load factor is $\alpha = n/b = n/62$.

Application 3: From the interval $[0, 1]$, $b = 100$ since there can be a total of 100 different keys that can be generated and, from the interval [0.01, 1.01), keys are generated uniformly at random. As a result, the load factor is $\alpha = n/b = n/100$.

(b) Due to it having the largest load factor, Application 1 will yield the worst performance.

(c) Due to it having the smallest load factor, Application 3 will yield the best performance.

(d) Since all slots in the hash table have a chance to be chosen, Application 3 will allow the uniform hashing property to apply. In regards to the other applications, some of the slots are not available to be chosen – as they can only fill 51 and 62 unique slots respectively.

Name: Chirag Telang
ID: 109410601

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

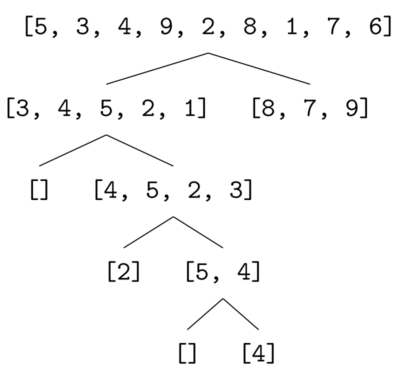
   (a) *(4 pts) Illustrate how to apply the QuickSelect algorithm to find the $k = 4$th smallest element in the given array:* `A = [5, 3, 4, 9, 2, 8, 1, 7, 6]` *by showing the recursion call tree. Refer to* Sam's Lecture 10 *for notes on QuickSelect algorithm works*

   (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) **Using Michael Levet's Method**:



- Invoke QuickSelect(A, 4), using the Partition algorithm, the pivot is the last element of the array where left = `[3, 4, 5, 2, 1]` and right = `[8, 7, 9]`. Now, as $k = 4$, we recurse again: QuickSelect(left, 4).
- Noting that A = `[3, 4, 5, 2, 1]`, using the Partition algorithm, the pivot is the last element of the array where pivot = 1, left = [ ], and right = `[4, 5, 2, 3]`.
  Now, since $k = 4 > 1 + \text{len(left)} = 1 + 0 = 1$, we have that the 4th smallest element of A must be the 3rd smallest element of right. So, we recurse again: QuickSelect(right, 3).
- Noting that A = `[4, 5, 2, 3]`, using the Partition algorithm, the pivot is the last element of the array where pivot = 3, left = `[2]` and right = `[4, 5]`
  Now, since $k = 3 > 1 + \text{len(left)} = 1 + 1 = 2$, we recurse again: QuickSelect(right, 2).
- Noting that A = `[4, 5]`, using the Partition algorithm, the pivot is the last element of the array where pivot = 4, and left = [5]. Note that $k = 2 = \text{len(left)} + 1$. So we return pivot = 4.

(b) The purpose of the Median of Medians algorithm is to provide a good pivot for selection algorithms, such as QuickSelect. Additionally, the Median of Medians algorithm is faster than quickSort since it has a worst case runtime of $O(n * logn)$, while quickSort has a worst case runtime of $O(n^2)$.

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

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

median = [18, 32, 33, 57, 68]
pivot = 33 (median of medians)
pivot = p
The median of medians is calculated for each subarray at each level of recursion.

[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] where p = 33, k = 21

[0, 3, 6, 10, 18, 20, 22, 23, 25, 29, 31, 32] [35, 56, 57, 60, 68, 72, 73, 80, 82, 85, 99, 99] where p = 80, k = 8

[35, 56, 57, 60, 68, 72, 73] [82, 85, 99, 99] where p = 92, k = 0

[82, 85] [99, 99]

⋮

After the second recursion gets called, the algorithm searches the lower values subarray (values < pivot) – in this situation, the subarray would be [82, 85]. Once the algorithm completes all of its iterations, the 4th largest element would be 82.