Name: Kevina Wong
ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 4 – Due Thurs Feb 13 11:55pm **Spring 2020, CU-Boulder**

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: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- All submissions must be easily legible.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please allot at least as many pages per problem (or subproblem) as are allotted in this template.
- For Problem 4 **you must submit your code in a separate file on Canvas** (*not* in a zip file). **Solutions must have both code in a separate file and answers in the PDF, and your plots must match up with your code; solutions which don't will receive a 0 for Problem 4.**

Quicklinks: 1 2 3a 3b 3c 3d 4 4a 4b 4c 4d

---

1. Provide a one-sentence description of each of the components of a divide and conquer algorithm.

   - **Divide:** This is the step where one must break down the original problem into smaller instances of the same problem.
   - **Conquer:** This is the step where, if the problem is broken down to the point where the solution is trivial, solve it; otherwise, if it is not trivial, divide it into smaller parts again.
   - **Combine:** This is the step where all of the smaller parts are regroluped into a larger instaance and thus is the solution to the problem.

Name: Kevina Wong
ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 4 – Due Thurs Feb 13 11:55pm **Spring 2020, CU-Boulder**

2. For the following problems, **you must** use the pseudocode for QuickSort and Partition in Section 1.1 of the course note "Week 4: QuickSort" on canvas and the array $A = [2, 4, 7, 5, 1, 9, 6]$.

   (a) What is the value of the pivot in the call $partition(A, 1, 7)$?

   Answer: 6. This is becuase pivot = A[7] = 6.

   (b) What is the index of that pivot value at the end of that call to $partition()$?

   Answer: 5. This is because values 4 and 4 swap (so nothing occurs), 7 and 5 swap, 7 and 1 swap, and 7 and 6 swap. After all of those swaps, the value of pivot = 6 ends up in A[5].

   (c) On the next recursive call to Quicksort, what sub-array does $partition()$ evaluate? (Give the indices specifying the subarray.)

   Answer: The indices of the subarray are A[1:4]. This is because the index of the partition is now at 5.

Name: Kevina Wong
ID: 109179049

3. Suppose that we modify the *Partition* algorithm in QuickSort in such a way that on alternating levels of the recursion tree, *Partition* either chooses the best possible pivot or the worst possible pivot.

   (a) Given an array, what choice of pivot will result in the best partitioning, and which one will result in the worst partitioning?

   Answer: Given an array, the worse case will occur if the largest or the smallest value is selected to be its pivot. This is because it would cause the partitioning to be very unbalanced. Choosing an array's median value as its pivot will result in the best and balanced partitioning.

(b) Write down a recurrence relation for this version of QuickSort, and solve it asymptotically. Show your work. Assume that the time it takes to find the pivot (either best or worst, depending on the level of recursion) is $\Theta(n)$ for lists of length n.

Since there are alternating levels of recursion tree, to solve this, we would be plugging in best case into worse case.

Worst Case Recurrence Relation: $T_n = T_{n-1} + \Theta(n)$
Best Case Recurrence Relation: $T_n = 2T_{\frac{n}{2}} + \Theta(n)$

Here is how you plug it in:
Best Case $= T_{n-1} = 2T_{\frac{n-1}{2}} + \Theta(n-1)$
Recall that Worse Case $= T_n = T_{n-1} + \Theta(n)$
$T_n = (2T_{\frac{n-1}{2}} + \Theta(n-1)) + \Theta(n)$
Recurrence Relation Answer: $T_n = 2T_{\frac{n-1}{2}} + \Theta(n-1) + \Theta(n)$

Then, we can apply Master's Method:

Master's Method says that recurrence relations of the form $T_n = aT\frac{n}{b} + f(n)$. *There are three cases of solution that can occur when using masters method:*

If $f(n) = \Theta(n^c)$...

- $c < Log_b a$ *then* $T_n = \Theta(n^{Log_b a})$
- $c = Log_b a$ *then* $T_n = \Theta(n^c Logn)$
- $c > Log_b a$ *then* $T_n = \Theta(f(n))$

For $T_n = 2T_{\frac{n-1}{2}} + \Theta(n-1) + \Theta(n)$
Note that $\Theta(n-1), \Theta(n), and \Theta(n-1) + \Theta(1)$ *are asymptotically the same*

- *a=2*
- *b=2*
- *c=1*

Apply Master's Method:
$Log_2 2 = 1 = c$
Therefore, $T_n = \Theta(n^1 Logn)$
Answer: $T_n = \Theta(n^1 Logn)$

(c) Provide a verbal explanation of how this *Partition* algorithm affects the running time of QuickSort, in comparison with the case where the best possible pivot is always used.

Answer:
Knowing that the best case of the regular partition algorithm is $T_n = nlogn$, we can confirm that they are asymptotically the same. This is because, as we saw in part B with the modified algorithm, it is $T_n = \Theta(nlogn)$. While we can say that the modified algorithm is likely to be 2 times slower than the normal running time since the worse case would be chosen about half the time, they would be still be asymptotically the same since constants would not change it's asymptotic running time.

(d) Suppose now we modify the *Partition* algorithm in QuickSort in such a way that the algorithm chooses the worst possible pivot for two levels of recursion, then the best possible pivot at the next level, then the worst possible pivot for the next two levels, then the best possible pivot, and so on. What is your estimate of the asymptotic running time? Provide a verbal justification of your estimate. From there can conclude that when the partition algorithm is implemented, it will be more likely that the running time of Quicksort will be at its best case running time.

Answer:
If the partition algorithm was modified in quicksort, the running time of quicksort would be split up between two running times: the worst case running time, and the best case running time. Since the worse case running time occurs twice (because the worse possible pivot is chosen twice before the best case is chosen), we can say the this is represented by $T_{(\frac{2n}{3})}$. Similarly, we can say that the best case running time in the quicksort function can be represented as $T_{(\frac{n}{3})}$. Thus, we can conclude that the overall running of quicksort using this partition algorithm is represented by $T_n = T_{(\frac{2n}{3})} + T_{(\frac{n}{3})}$. Since the running time is split between three "nodes", we can argue that the running tim is $T_n = 3nlogn$. Asymptotically, however, $T_n = 3nlogn$ is the same as $T_n = nlogn$ because constants don't change a function's asymptotic running time. Therefore, we can say that using this partition algorithm, the aymptotic tunning time would be $T_n = 3nlogn$.

Name: Kevina Wong
ID: 109179049

4. A good hash function $h(x)$ behaves in practice very close to the simple uniform hashing assumption analyzed in class, but is a deterministic function. Designing good hash functions is hard, and a bad hash function can cause a hash table to quickly exit the sparse loading regime by overloading some buckets and under-loading others. Good hash functions often rely on beautiful and complicated insights from number theory, and have deep connections to pseudorandom number generators and cryptographic functions. In practice, most hash functions are moderate to poor approximations of uniform hashing.

   Consider the following two hash functions. Let U be the universe of strings composed of the characters from the alphabet $\Sigma = [\texttt{A}, \ldots, \texttt{Z}]$, and let the function $f(x_i)$ return the index of a letter $x_i \in \Sigma$, e.g., $f(\texttt{A}) = 1$ and $f(\texttt{Z}) = 26$. Let x be a string of length m.

   (1) The first hash function we consider is $h_1(x) = [\sum_{i=1}^{m} f(x_i)] \mod \ell$, where ℓ is the number of buckets in the hash table.

   (2) For the second hash function, first—globally, external to the hash function—choose uniformly random integers a_i (one for each $x_i \in \Sigma$) from $\{0, \ldots, 10{,}000\}$, and then define $h_2(x) = [\sum_{i=1}^{m} a_i \cdot f(x_i)] \mod \ell$.

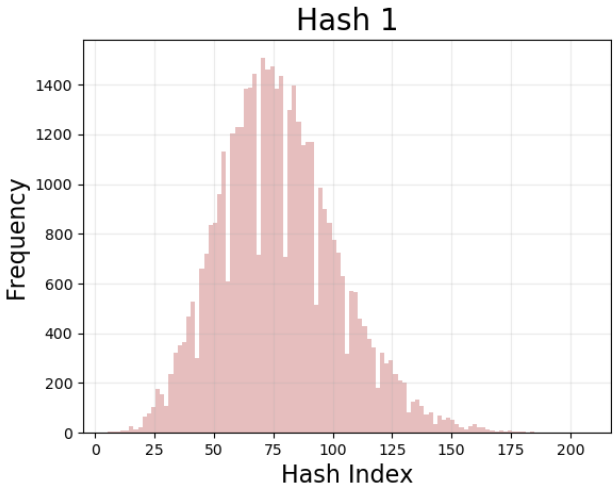
   List your values of a_i here: (and please use consistent values of a_i throughout this question)

   a_i values: 4053, 5251, 9073, 9315, 9714, 3389, 7362, 7905, 8265, 7518, 7087, 4587, 3306, 3377, 6746, 5766, 2026, 5106, 1543, 8923, 3036, 8862, 4416, 6561, 1763, 5037

Name: Kevina Wong
ID: 109179049

(a) There is a txt file on Canvas that contains US Census derived last names. Using these names as input strings, first choose a uniformly random 50% of these name strings. Let $\ell = 5851$ be the number of buckets. For each of the two hash functions (separately), produce a histogram showing the distribution of hash locations for the names you chose. Label the axes of your figures. Give a brief description of what the figure shows about $h_1(x)$ and $h_2(x)$; justify your results in terms of the behavior of these hash functions.

Hint: the raw file includes information other than the name strings, which will need to be removed; and, think about how you can count hash locations without building or using a real hash table.



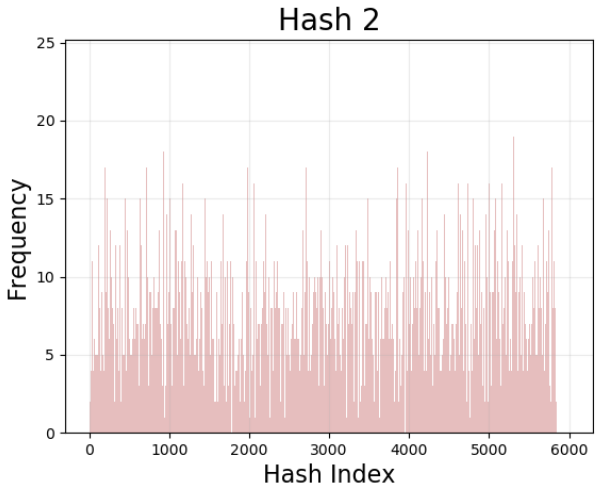


Figure 1: Left: Histogram of $h_1(x)$ Right: Histogram of $h_2(x)$

Based on the histograms for $h_1(x)$ and $h_2(x)$, it shows that $h_1(x)$ would be a more efficient hash table. This is because the performance of a hash table depends on how well the hash function can **evenly** distribute the set of elements. Because the histogram for $h_1(x)$ has a clear peak, it shows that the hash function for $h_1(x)$ unevenly distributes the set of elements; whereas, the histogram for $h_2(x)$ is relatively flat which shows that the hash function can evenly distribute the set of elements, and thus, have a better performed hash table.

(b) State at least 4 reasons why $h_1(x)$ is a bad hash function relative to the ideal behavior of uniform hashing.

- Because values 1...26 are so small scaled, it is extremely easy to have collisions with the hash keys.
- Because this hash function does not take take into account the order in which the values take place, and using such small values, strings with the same letters (in other words, anagrams) will lead to a collision (ie. "SMALL" and "MALLS"), and in addition to the small values added together, collisions are significantly more frequent. and build up quckly at certain hash keys.
- The hash function $h_1(x)$ distributes hash keys unevenly as shown by the peak in the histogram. This makes it a bad hash function because in the behavior of ideal hashing, a historgram of the hash keys would ideally be flat.
- Much of the hash table is left unused because looking at the histograms, the hash keys are at most about 200, whereas in $h_2(x)$, it uses hash keys up to 6000. This shows that a large majority of this hash table is left unused.

(c) Produce two plots—one for each hash function h_1, h_2—showing the length of the longest chain (were we to use chaining for resolving collisions) as a function of the number n of these strings that we hash into a table with $\ell = 5851$ buckets. That is, you may use the 50% of names from part (a), and as you hash them one by one, show how the length of the longest chain is growing.

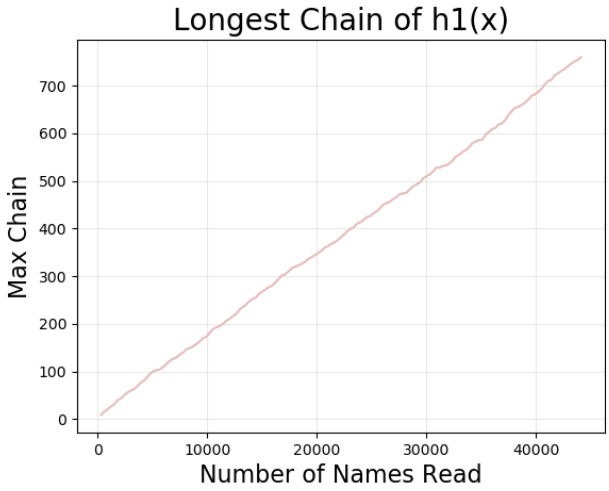
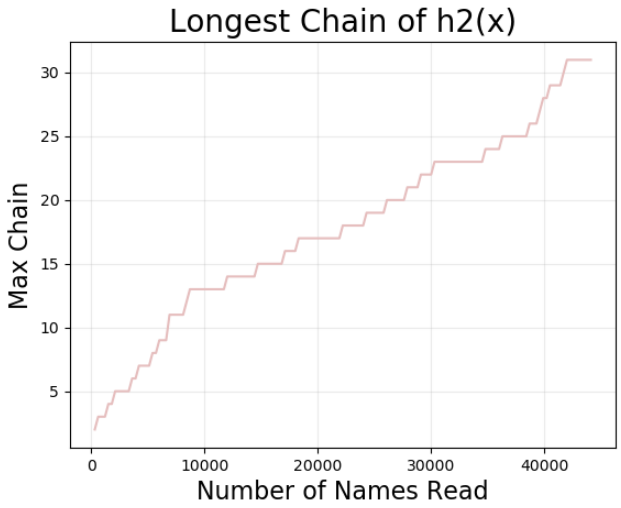


Figure 2: Left: Longest Chain of $h_1(x)$



Right: Longest Chain of $h_2(x)$

(d) Produce another pair of plots—one for each of h_1, h_2—showing the number of collisions as a function of ℓ. Comment on how collisions decrease as ℓ increases. Aside from size, do you notice any particular kinds of values for ℓ that seem better than others? (e.g. odd/even, prime, etc.) Discuss briefly.

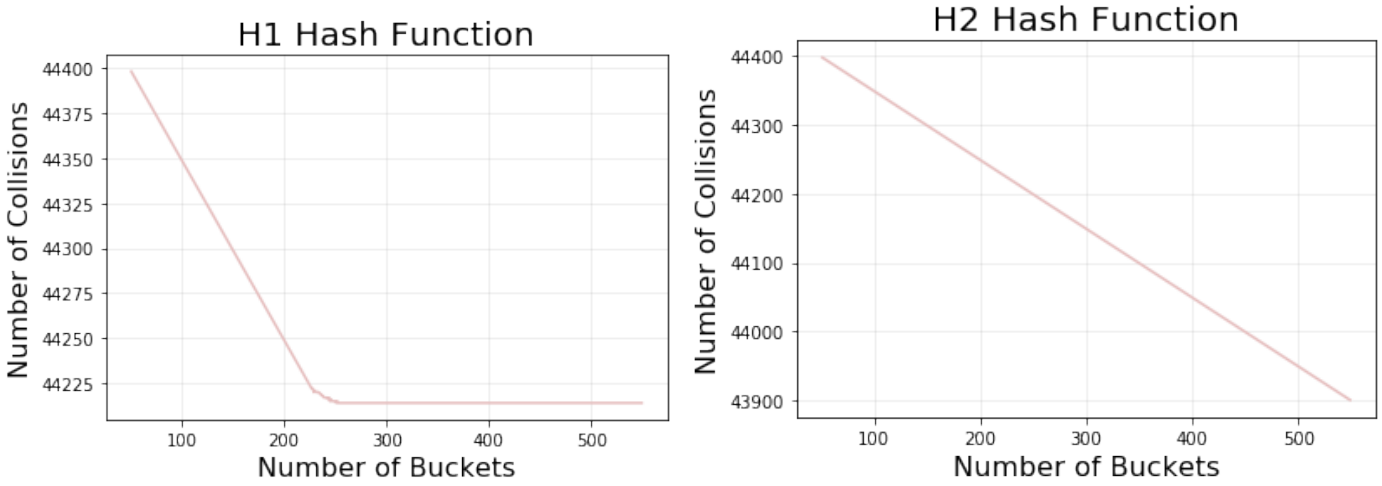


Figure 3: Left: Collisions of $h_1(x)$ Right: Collisions of $h_2(x)$

For both of the plots, it shows that as the number of buckets, l, increases, the number of collisions decreases. h_1 drecreases exponentially, while h_2 appears to decrease linearly. Furthermore, prime numbers would be a better. This is becuase prime numbers in nature have no factors other than 1 and itself. Therefore, making it the number of buckets would make it less likely for collision to occur. Along that similar idea, powers of 2 would be more likely to have higher counts of collision because powers of 2 have a very large number of factors.

Name: Kevina Wong
ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 4 – Due Thurs Feb 13 11:55pm **Spring 2020, CU-Boulder**

Collaboration and References

- Worked with other students at office hours to solve problems together.
- Helped many other students plot histograms
- GeeksForGeeks algorithm Analysis Page for information on Master's Method.