Name: Alex Book
ID: 108073300
Collaboration:
Varun Narayanswamy, Jacob Reed, Jamie Foster, Athena Parker, Bailee Mulholland

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 11 (60 points) - Extra Credit **Fall 2019, 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: 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.
- If you don't have an account on it, sign up for one using your CU email. You should have gotten an email to sign up. If your name based CU email doesn't work, try the identikey@colorado.edu version.
- Gradescope will only accept **.pdf** files (except for code files that should be submitted separately on Gradescope if a problem set has them) and **try to fit your work in the box provided**.
- You cannot submit a pdf which has less pages than what we provided you as Gradescope won't allow it.

Name: Alex Book
ID: 108073300
Collaboration:
Varun Narayanswamy, Jacob Reed, Jamie Foster, Athena Parker, Bailee Mulholland

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 11 (60 points) - Extra Credit **Fall 2019, CU-Boulder**

Important:
There is no late due date for this extra credit assignment.
This assignment has 1 (Q1) coding question.

- You need to submit 1 python file.
- The .py file should run for you to get points and name the file as following -
  If Q1 asks for a python code, please submit it with the following naming convention - `Lastname-Firstname-PS11-Q1.py`.
- You need to submit the code via Canvas but the table/plot/result should be on the main .pdf.

Name: Alex Book
ID: 108073300
Collaboration:
Varun Narayanswamy, Jacob Reed, Jamie Foster, Athena Parker, Bailee Mulholland

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 11 (60 points) - Extra Credit **Fall 2019, CU-Boulder**

1. (40 pts total) 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. That is, $h(x) = k$ each time x is used as an argument to $h()$. 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.

   Python code - 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 $f(\texttt{Z}) = 26$. Finally, for an m-character string $x \in \Sigma^m$, define $h_1(x) = ([\sum_{i=1}^{m} f(x_i)] \mod \ell)$, where ℓ is the number of buckets in the hash table. For the other hash function, let the function $f_2(x_i, a_i)$ return $f(x_i) * a_i$, where a_i is a uniform random integer, $a_i \in [\texttt{0}, \ldots, \ell\texttt{-1}]$, and define $h_2(x) = ([\sum_{i=1}^{m} f_2(x_i, a_i)] \mod \ell)$. That is, the first hash function sums up the index values of the characters of a string x and maps that value onto one of the ℓ buckets, and the second hash function is a **universal hash function**. Note that for $h_2(x)$, the a_i values are chosen randomly and fixed before you start hashing. Thus $h_2(x)$ is a randomly sampled hash function from the uniform hash function family.

   (Your code should have the code for each of the parts below for you to receive credit for those parts.)

   (a) (16 pts) 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 and then hash them using $h_1(x)$ and $h_2(x)$. Produce a histogram showing the corresponding distribution of hash locations for each hash function when $\ell = 5987$. Label the axes of your figure. Brief description what the figure shows about $h_1(x)$ and $h_2(x)$; justify your results in terms of the behavior of $h(x)$.

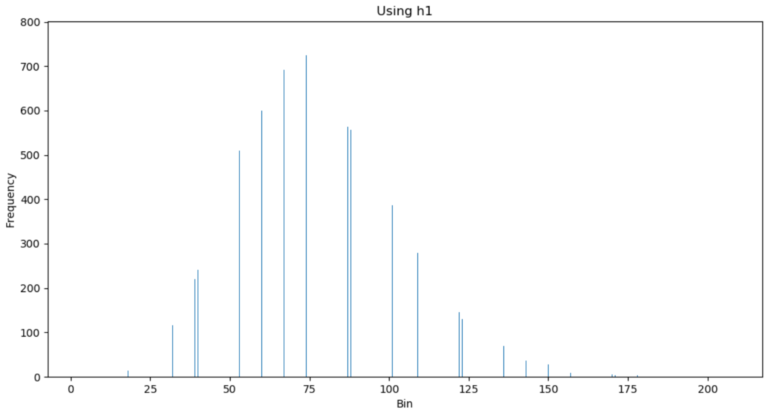
   Hint: the raw file includes information other than the name strings, which will need to be removed; and, think about how you can count hash locations without building or using a real hash table.

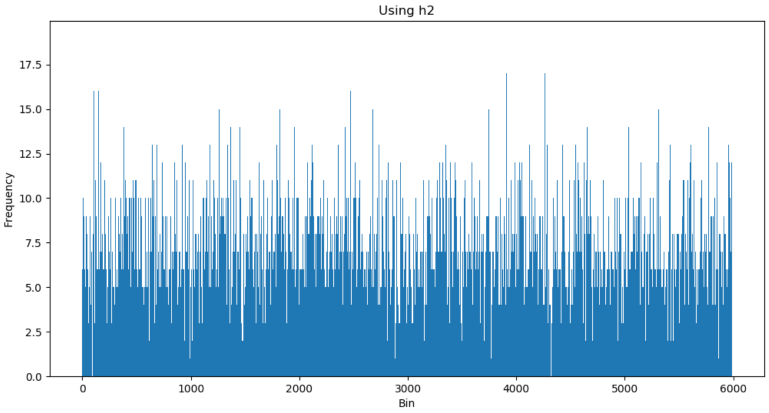
   Solution.

   The figures show the following about $h_1(x)$ and $h_2(x)$:

   $h_1(x)$ has very high concentration (high sum) in certain buckets, while having nothing (or very little) in others (literally none past 200, as shown in the figure). This means that $h_1(x)$ does a poor job of hashing the names.

   $h_2(x)$ has a nice distribution of concentration among the buckets, using nearly every one. This means that $h_2(x)$ does a much better job of hashing the names.


Using h1
Frequency
0
100
200
300
400
500
600
700
800
Bin
0
25
50
75
100
125
150
175
200


Using h2
Frequency
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Bin
0
1000
2000
3000
4000
5000
6000

(b) (4 pts) Enumerate at least 3 reasons why $h_1(x)$ is a bad hash function relative to the ideal behavior of uniform hashing.

Solution.

1. All of the names are somewhat similar in length and character content, and are thus not likely to differ too much from each other in “codeSum” (especially not on a scale of almost 6,000).

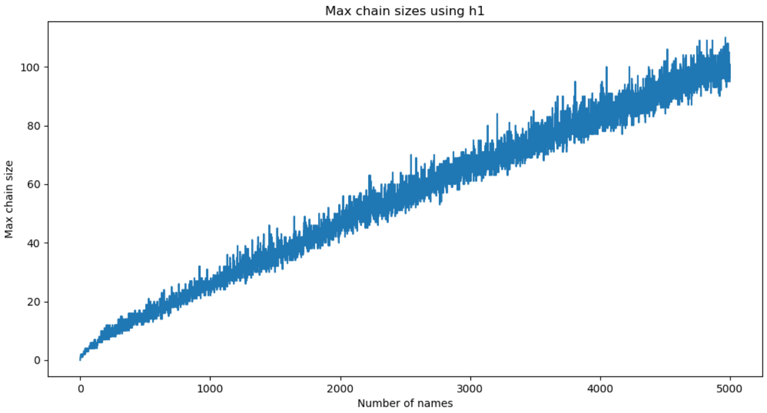
2. Taking such a large sample of names makes it very likely that some names have the same length and characters (just in different orders), which would cause them to have equal “codeSum” values.

3. Certain letters (such as a, e, s, t, n, etc.) are much more common than others (such as u, v, w, x, y, z, etc.), causing the average “codeSum” value to tend to be toward those letters (tend to have lesser “codeSum” values because the larger code values happen to be lesser-used letters).

(c) (10 pts) For $h_1(x)$, produce a plot showing (i) 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 $\ell = 5987$ buckets. (ii) Produce another plot showing the number of collisions as a function of ℓ. Choose prime numbers for ℓ and comment on how collisions decrease as ℓ increases.

Solution.

(i):



(ii):

I'm choosing to save time during the busy pre-finals week and not re-structure the code for many different numbers of ℓ, but here's the explanation for (hopefully) partial credit: Collisions decrease as ℓ decreases because having more buckets allows more possible values for "codeSum *mod* ℓ", therefore placing different names in different buckets.

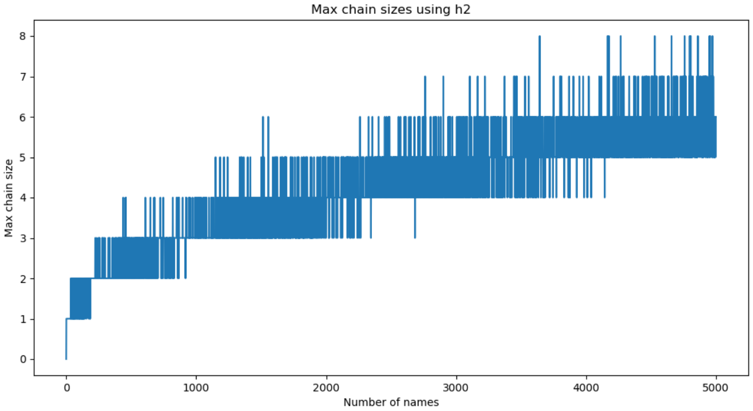
Name: Alex Book
ID: 108073300
Collaboration:
Varun Narayanswamy, Jacob Reed, Jamie Foster, Athena Parker, Bailee Mulholland

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 11 (60 points) - Extra Credit **Fall 2019, CU-Boulder**

(d) (10 pts) Repeat the exercise in part (c) for $h_2(x)$ and also comment on the difference in the plots produced with respect to $h_1(x)$.

Solution.

(i):



(ii):

I'm choosing to save time during the busy pre-finals week and not re-structure the code for many different numbers of ℓ, but here's the explanation for (hopefully) partial credit: Collisions decrease as ℓ decreases because having more buckets allows more possible values for "codeSum *mod* ℓ", therefore placing different names in different buckets.

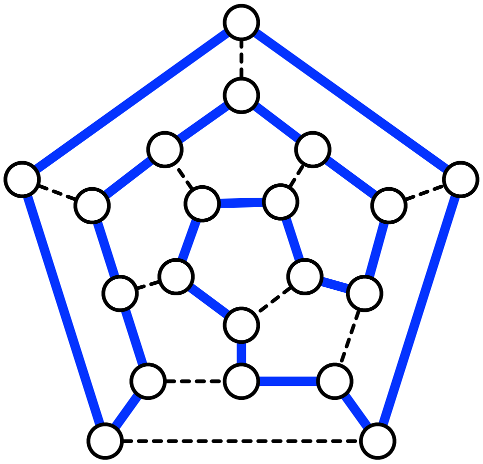
(iii):

The plot produced by $h_2(x)$ increases more slowly than the one produced by $h_1(x)$ because the way in which it hashes names is more effective (using a randomized number) than simply summing the characters in the name.

Name: Alex Book
ID: 108073300
Collaboration:
Varun Narayanswamy, Jacob Reed, Jamie Foster, Athena Parker, Bailee Mulholland

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 11 (60 points) - Extra Credit **Fall 2019, CU-Boulder**

2. (20 pts total) Every young wizard learns the classic NP-complete problem of determining whether some unweighted, undirected graph $G = (V, E)$ contains a Hamiltonian cycle, i.e., a cycle that visits each vertex exactly once (except for the starting/ending vertex, which is visited exactly twice). The following figure illustrates a graph and a Hamiltonian cycle on it.



(a) (10 pts) Ginny Weasley is working on a particularly tricky instance of this problem for her Witchcraft and Algorithms class, and she believes she has written down a "witness" for a particular graph G in the form of a path P on its vertices. Explain how she should verify in polynomial time whether P is or is not a Hamiltonian cycle. (And hence, demonstrate that the problem of Hamiltonian Cycle is in the complexity class NP.)

Solution.

(b) (10 pts) For the final exam in Ginny's class, each student must visit the Oracle's Well in the Forbidden Forest. For every bronze Knut a young wizard tosses into the Well, the Oracle will give a yes or no response as to whether a particular graph contains a Hamiltonian cycle. Ginny is given an arbitrary graph G, and she must find a Hamiltonian cycle on it to pass her exam.

Describe an algorithm that will allow Ginny to use the Oracle to solve this problem by asking it a series of questions, each involving a modified version of the original graph G. Her solution must not cost more Knuts than a number that grows polynomially as a function of the number of vertices in G. (Hence, prove that if we can solve the Hamiltonian cycle *decision* problem in polynomial time, we can solve its *search* problem as well.)

Solution.