princeton univ. F'18 cos 521: Advanced Algorithm Design

Homework 1: Random Variables and Algorithms (**60pts**)

Due: **Monday, October 8th, 2018, 11:59pm**

Collaboration is allowed on this problem set, but solutions must be written-up individually. Please list collaborators for each problem separately, or write "No Collaborators" if you worked alone. Collaboration is not allowed on bonus problems.

Please prepare your problem sets in LaTeX and compile to a PDF for your final submission. A LaTeX template is available on the course webpage.

§1 **(8 pts)** In class, we saw that, when hashing m items into a hash table of size $O(m^2)$, the expected number of collisions was < 1. In particular, this meant we could easily find a "perfect" hash function of the table that has no collisions.

Consider the following alternative scheme: build two tables, each of size $O(m^{1.5})$ and choose a separate random hash function for each table. To insert an item, hash it to one bucket in each table and place it in the emptier bucket.

(a) Show that, if we're hashing m items, with probability $1/2$, there will be no collisions in either table. You may assume a fully random hash function, although this assumption is not necessary (2-universal hashing is fine).

(b) Modify the above scheme to use $O(\log m)$ tables. Prove that this approach yields a collision-free hashing scheme with space $O(m \log m)$. Again, you may assume a fully random hash function.

§2 **(8 pts)** In modern systems, hashing is often used to distribute data items or computational tasks to a collection of servers. What happens when a server is added or removed from a system? For most hash functions, including those discussed in class, the hash function is tailored to the number of servers, n, and would change completely if n changes. This would require rehashing and moving all of our m data items.

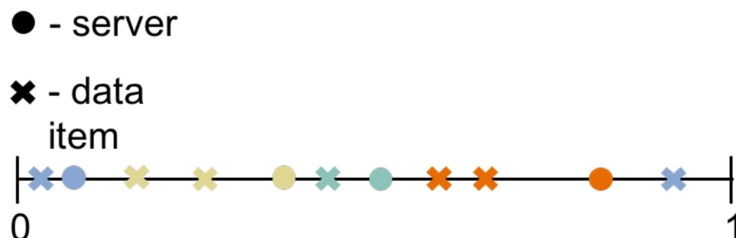

Figure 1: Each data item is stored on the server with matching color.

Here we consider an approach to avoid this problem. Assume we have access to a completely random hash function that maps any value x to a real value $h(x) \in$

$[0,1]$ (this is a simplification – the scheme can also be implemented with real, low-independence hash functions). Use the hash function to map *both* data items and servers randomly to $[0,1]$. Each data item is stored on the first server to its right on the number line (with wrap around). When a new server is added to the system, we hash it to $[0,1]$ and move data items accordingly.

(a) When a new server is added to the system, what is the expected number of data items that need to be relocated?

(b) Show that, with high probability, no server "owns" more than an $O(\log n/n)$ fraction of the interval $[0,1]$.

When analyzing randomized algorithms, "with high probability" typically refers to probability $1-1/n^c$ for some constant c. For this homework, you can prove the result for $c=1$, but you might find it holds with even better than probability.

(c) Show that if we have n servers and m items, the maximum load on any server is $O(\frac{m}{n}\log(n))$ with high probability.

§3 **(8 pts)** A cut is said to be a *B-approximate min cut* if the number of edges in it is at most B times that of the minimum cut. Show that a graph has at most $(2n)^{2B}$ cuts that are B-approximate. (Hint: Run Karger's algorithm until it has $2B+1$ supernodes. What is the chance that a particular B-approximate cut is still available? How many possible cuts does this collapsed graph have?)

§4 **(8 pts)**

(a) Show that given n numbers in $[0,1]$, it is impossible to estimate the *value* of the median with $o(n)$ *uniformly drawn* samples from that set, even if we just want to get within a constant approximation factor, and succeed with constant probability. (Hint: to show an impossibility result, show two different sets of n numbers that have very different medians but, with high probability, generate identical samples of size $o(n)$.)

We do not care about what specific constants your result holds for, as long as they are truly constants that do not depend on n. For example, it would be fine to show that it's impossible to estimate the median with $n/10$ samples, to within a 1.1 factor, with success probability 9/10.

(b) In contrast, show that it is possible to use $o(n)$ uniform samples to find an approximate median in the following sense: with success probability $1-\delta$, find some point x so that there are at least $(1-\epsilon)\frac{n}{2}$ numbers in our original set that are greater than or equal to x and at least $(1-\epsilon)\frac{n}{2}$ numbers that are less than or equal to x.

How many samples are required to find x (as a function of ϵ, n, δ, etc.).

§5 **(10 pts)** In class we saw a hash to estimate the size of a set. Change it to estimate frequencies. Thus there is a stream of packets each containing a *key* and you wish to maintain a data structure which allows us to give an estimate at the end of the *number of times* each key appeared in the stream. The size of the data structure

should not depend upon the number of distinct keys in the stream but can depend upon the success probability, approximation error etc. Just shoot for the following kind of approximation: if a_k is the true number of times that key k appeared in the stream then your estimate should be $a_k \pm \epsilon(\sum_k a_k)$. In other words, the estimate is going to be accurate only for keys that appear frequently in the stream.

This problem requires creativity! We will give some credit to any solution using $o(n)$ space. Hint: Think in terms of maintaining $m_1 \times m_2$ counts using as many independent hash functions, where each key updates m_2 of them.

§6 **(10 pts)** Consider the following process for matching n jobs to n processors. In each step, every job picks a processor at random. The jobs that have no contention on the processors they picked get executed, and all the other jobs back off and then try again. Jobs only take one round of time to execute, so in every round all the processors are available. Show that all the jobs finish executing after O(log log n) steps, with high probability.

§7 **(8 pts)** In MATLAB, Python, or any other language, implement the random hash family described in Lecture 1.

(a) Write code to experimentally test whether or not the family is 2-universal and include a plot and/or other evidence demonstrating what you find. You can run your experiments for some fixed, small value of the table size n (e.g. $n = 20$), but should test various values of the universe size $|\mathcal{U}|$ (e.g. $|\mathcal{U}|$= n, 2n, 4n).

(b) Based on your experiments, is the hash function also 2-independent? (Note the difference in definition between 2-independent and 2-universal).

(c) Run the same experiments on the hash function:

$$h(x) = (ax \mod p) \mod n.$$

I.e. what we saw in class, but with $b = 0$. Is the hash function still 2-universal? Provide evidence for your yes or no answer.

(d) Finally, invent a random hash function of your own that, at least experimentally, appears to be 2-universal. Your function could involve crazy stuff like taking XOR of bits, etc.