Algorithms running notes

Karl Pichotta

Spring 2013

1 Preamble: Useful identities

- For large n, we have

$$\left(1 + \frac{1}{n}\right) \approx e \tag{1}$$

$$\left(1 - \frac{1}{n}\right) \approx e^{-1} \tag{2}$$

2 1/22/2013: Chernoff Bounds, the Tails of the Binomial

If we have a binomaial RV $X \sim B(n, p)$, then we have the following Chernoff bound, for $0 \leq \delta \leq 1$:

$$P(X \leq (1-\delta)np) \leq \exp\left\{\frac{-\delta^2 np}{2}\right\}$$

Supposing we throw $100n \log n$ balls, by focusing our analysis on a single bin and then applying Union Bound, we get that the probability that some $X_i \leq \log n$ is $\leq n^{-44}$, using the above bound.

Fact: Thinking about Hashing, suppose we throw n balls into n bins, and we want a bound on the max load of any given bin. We might hope it's $O(1)$, but it isn't. Instead, it is

$$\Theta\left(\frac{\log n}{\log \log n}\right)$$

with high probability.

Upper bound: Define $Z = \max_i X_i$, with X_i the number of balls in bin i. Consider bin 1. $E[X_1] = 1$, but this tells us nothing about the tail of the distribution, which is our interest here.

What then, is

$$Pr\left\{X_1 \geq c\frac{\log n}{\log \log n}\right\}$$

Framing it in terms of Chernoff bounds, we will want to use:

$$\delta = c\frac{\log n}{\log\log n} - 1$$

We need to use the Large Deviations bound (bound (2) on handout). This is the following: Suppose $X \sim Binom(n,p)$, then

$$Pr\{X \geq (1+\delta)np\} \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{np}$$

For concreteness, we take $c = 100$. Our $np = 1$, so the RHS of the above chernoff bound is

$$... \leq \left(\frac{e^{\delta+1}}{\left(100\frac{\log n}{\log\log n}\right)^{100\log n/(\log\log n)}}\right) \leq \left(\frac{\log\log n}{\log n}\right)^{100\log n/(\log\log n)} \approx n^{-100}$$

WE inflate the numerator by adding an extra e.

Sidenote: Now, this funny log logn terms come from the fact that solving

$$x^x = n$$

gives you something very like $\log n/\log\log n$. This comes from canceling out lower order approximations to log.

Returning, by union bound, we argue that the probability that some bin gets $\geq 100\log n/\log\log n$ balls is $\leq n^{-99}$.

Lower bound: So that was an upper bound. The lower bound is trickier. Let $k = \varepsilon\log n/\log\log n$, with ε a small positive argument. We won't be able to just reason about bin 1 and then use union bound—there is a constant probability $(1-1/n)^n \approx e^{-1}$ that bin 1 gets 0 bins. Let E_i denote the event that bin i receives at least k balls. We want to show that

$$Pr(\cup E_i) \geq 1 - \frac{1}{n^c}$$

This is, reminder, a lower bound on the max. That is, we're showing that

$$Z = \Omega(\log n/\log\log n).$$

We're showing that $Pr(E_1)$ is small, but it'll be useful.

So consider $Pr(E_1)$. We have

$$Pr(E_1) \geq Pr(X_1 = k)$$

(since RHS entails LHS) This is equal to

$$Pr(X_1 = k) = \binom{n}{k} (1/n)^k (1 - 1/m)^{n-k} \tag{3}$$

$$\geq \binom{n}{k} (1/n)^k (1/e) \tag{4}$$

$$\geq \left(\frac{n}{k}\right)^k (1/n)^k (1/e) \tag{5}$$

$$= \frac{1}{ek^k} \tag{6}$$

$$\approx \frac{1}{n^{\varepsilon'}} \tag{7}$$

We have the third to last step by the following useful lower bound:

$$\binom{n}{k} \geq (n/k)^k$$

which you see by expanding the formula and reasoning about the various values of quantities above and below the line.

Now, by linearity of expectation, we expect $n^{1-\varepsilon'}$ bins to get at least k balls.

What we want, however, is a high probability bound. That is, we want a statement of the form "at least one of bins, whp, will get at least k bounds". We can't use a Chernoff bound, because the E_i's are not independent, so therefore the distribution isn't Binomial.

From a highl-level perspective: we throw n balls into n bins, then ask if E_1 occurred. If it did, then we'll be happy, because all we want is one of the E_is to occur (NB $PR(E_1) \geq 1/(n^{\varepsilon'})$. If E_1 doesn't occur, then note

$$PR(E_2|E_1^c) \geq Pr(E_2) \geq 1/n^{\varepsilon'}$$

and so on with all of the $Pr(E_i)$'s. We can therefore argue that the probbability that noE_i occurs is

$$\leq (1 - 1/n^{\varepsilon'})^n$$

We want to factor this last equation into

$$... = ((1 - 1/n^{\varepsilon'})^{n^{\varepsilon'}})^{n^{1-\varepsilon'}} \tag{8}$$

$$\approx (1/e)n^{1-\varepsilon'} \tag{9}$$

Instead of having $1/e$ to some logarithmic quantity, we have it to some polynomial quantity. So this is much, much less than inverse polynomial bound, and we've more than satisfied our sharp threshold.

An exercise close to the one we did at the begnning—using Chernoff bounds, we can argue that the number of flips of a fair coin to get $\log n$ heads with high probability is

$$\Theta(\log n)$$

(we use Chernoff bound (1), as in the first example). This comes up in at least one way to analyze randomized Quicksort.

In last class, we saw that the expected number of comparisons is $\Theta(n \log n)$. We now argue that, with high probability, the number of comparisons is $O(n \log n)$. Note this is quite different: what we say now is that the probability that you exceed the expected runtime by a factor of, say, 50, is very small. Note these RVs aren't binomial, but we'll be able to bound their behavior with binomial RVs.

First, we'd LIKE to show the number of comparisons involving a specific key is $O(\log n)$ with high probability. This is not true—there is a $1/n$ chance of an element being the first pivot, and it gets compared to n elements.

So what we'll do instead is to use a charging scheme: we charge a comparison to a non-pivot. Whenever we make a comparison, "charge" the comparison to the nonpivot (each comparison involves a pivot and a nonpivot). It IS true that the charge to any key is $O(\log n)$ with high probability. Once we show that, it immediately follows that the total number of comparisons is $O(n \log n)$ WHP. HOw can we use the prvious result to convince ourselves of this?

3 1/24/2013: Hashing

Let's think about Randomized quicksort. We know that randomized quicksort has expected $O(n \log n)$ behavior. (that is, there are no "bad inputs" for it in expectation). Note also that Randomized quicksort has $O(n \log n)$ runtime with high probability (that is, we'll give a nice bound on the probability of the runtime being asymptotically higher).

We can think about quicksort from the perspective of a particular fixed key, call it x. Recall the charging scheme: we "charge" each comparison to the nonpivot (which we can do, since each comparison is between a pivot and another value).

Claim: The key x gets $O(\log n)$ charge with high probability (the failure probability will be $1/n^c$ for arbitrarily large c. (If we prove this, then by Union Bound, we have the total charge to all n keys is $O(n \log n)$ WHP.)

Proof of Claim: We have a "herd" of keys, initially of size n, and select one to be the pivot. The pivot is remarkably lucky—it gets 0 charge. If x is chosen as the pivot, that's nice. If, on the other hand, x is not chosen as pivot, then it will get charge 1. Then the keys will be partitioned into two "herds", one of which will never again be compared with x.

Consider the process where we start with a positive integer n. We flip a coin. If we get heads, we set the number to an integer uniformly drawn from $[0, 3n/4]$ (floored). If we get tails, then we set the value uniformly to between 0 and $n-1$. So at each stage, the value gets smaller (by at least one). The process stops when the number gets to 0.

We use this process as follows. Picking a pivot at random, there is a 0.5 probability that the pivot sits in between $n/4$ and $3n/4$ in the sorted list (it sits

in the middle half). There's also a .5 chance it's not in the middle. The number in the process is just the size of the "herd" x is in.

If we get a good pivot, then the herd value is at most $3n/4$ (imagine getting the rightmost pivot in the middle half, say). This is like flipping a head. In the worst case (flipping a tail), then the size of the herd decreases by 1.

The process can't involve more than some constant times $\log n$ heads: each head decreases the herd size by at least 3/4. (Prove this?) So x sits there and hopes for good pivot choices from its perspective. (This is basically the end of the proof.)

We don't have a binomial variable, but we can relate it to this simpler process defined above, and use a Chernoff bound on that.

3.1 Hashing with chained overflows

We have a hashtable, visualized as an array of buckets, numbered $0, \ldots, k-1$. We have a has function

$$h(x) \in \{0, \ldots, k-1\}.$$

In order to handle collisions, we can use a linked list to chain them together in the bucket. Searching for an item in the table later involves computing the hash and then traversing the linked list.

Suppose you put k things in. You're generally hoping that each bucket tends to have $O(1)$ elements. In the worst case, you traverse a linked list (and have $O(k)$ lookup). In the best case, we get $O(1)$ lookup.

What's the runtime of h? Supposing we have n keys, each of which is $10 \lg n$ bits. (Note we're using the RAM model: the word size of the machine we're programming is logarithmic in the input size in bits of the problem. So if we have a million-bit instance, we assume that we can manipulate words of log 1m in constant time.) So that means that our word size is $\Theta(\log n)$ bits. That is, these keys will fit into a constant number of words. We'll be relying on this later to assume various operations are $O(1)$.

So for now, we think of h as taking any key and mapping uniformly random from $0, k-1$. This is of course not quite right: h must be deterministic. However, we think of the hash for our purposes as choosing uniformly at random.

This relates directly to the bin/bucket problems we discussed previously.

Claim: The average time for a search is $O(1)$, assuming n buckets (with n keys).

No proof (prove this?). Basically, the vast majority of buckets will get very few elements.

On the other hand, what's the expected max search time? That is, has n keys into n buckets with our idealized hash. The worst-case search time is the longest linked-list we get. We looked at this last time: The max load is

$$\Theta\left(\frac{\log n}{\log \log n}\right) \ w.h.p.$$

So it's only marginally better than using a red-black tree.

3.2 Perfect Hashing

At a high-level, we use *Perfect Hashing* to obviate this non-constant load problem. That is the following:

We assume we're dealing with a *static* set of n keys. That is, we construct a hash-table structure that is specific for the particular n keys that we have at hand. As before, assume each key is $10 \log_2 n$ bits.

Desiderata:

- We want to construct a hash table with $O(1)$ worst case search time. NOTE this isn't just a matter of expectation: we want guarantees about the worst case.
- We also want to use $O(n)$ space (that is, our solution can't be "use $O(2^n)$ keys").
- Further, we want "fast" construction

A naive approach is the following: Repeatedly pick a new hash function until you find one inducing max load of $O(1)$. We can do this because we have static keys (we know them in advance). Eventually we'll find one with $O(1)$ worst case. However, this won't give us fast construction: we'll have to run this an exponential number of times in order to find this (I don't quite get the proof, but it involves one of the results we showed last time involving $\log n$).

We'll use a twist on this: we'll follow a similar approach, but our criterion for selecting a hash function will be looser. we'll use a two-tiered approach: What we'll do is count "collisions", and as long as we don't have more than n collisions, we'll call it good. So we have n buckets, but instead of a linked list in each bucket, we have a hash table at each of the n buckets. That is, we have one primary hash table, and n secondary hash tables.

We want the total size of all these tables to be $O(n)$. We do our primary hashing; based on how many elements hash into a location, each bucket has a hash table of that size. We'll have the size of a secondary hash table being quadratic in the number of elements there. So if we have 20 elements in a bucket, we'll have a hash table of size approximately 20^2. Why? We want to totally avoid collisions at the second level of hashing. The thing about quadratic size is that this is the threshold at which we'll suffer 0 collisions in the secondary hash tables (we can repick secondary hashes).

One concern is that since we're wasting space in the secondary tables, when we add the size of the secondary tables up, it'll be too large. So we need to show that the sum of the table sizes will be linear (intuitively, we'll show the vast majority of buckets get only a constant number of keys).

So OK getting to details. What is a "collision"? Mapping keys to buckets, we'll let Y_i denote the RV corresponding to the number of keys mapping to bucket i at the top level (for $1 \leq i \leq n$). Let X be the RV denoting the number of collisions. Then

$$X = \sum_{i=1}^{n} \binom{Y_i}{2}.$$

That is, if 10 keys are mapped to a bucket, that gives 10 choose 2 pairwise collisions. Note that

$$X = \Theta\left(\sum_{i=1}^{n} Y_i^2\right)$$

which is, magically, the total size of the secondary hash tables. So X is constant-factor-related to the space required for the secondary hash tables. (Note that it's good enough, then, to pick a hash function that induces at most $100n$ or $1000n$ collisions.)

Now, to get at $E[X]$, we express X as a sum of indicator variables. Define Z_{ij} to be the indicator variable taking 1 if the i, jth keys collide, and 0 otherwise. Note there are n choose 2 such vars. So

$$E[X] = \sum_{i,j} E[Z_{ij}] \tag{10}$$

$$= \sum_{i,j} 1/n \tag{11}$$

$$= \binom{n}{2}/n \tag{12}$$

$$= \frac{n-1}{2} \tag{13}$$

where we appeal to the fact that we have an idealized hash function. This is less than $1/2$ of our target of getting $\leq n$ collisions. (It is important that we choose a constant $> 1/2$ when defining our criterion for accepting a hash functions.)

Call a hash function "good" if it induces $\leq n$ collisions; "bad" otherwise. On average, the number of collisions is $n/2$. We can therefore bound the percentage of all hash functions that are "bad". If, for example, 90% of the has functions are bad, then the expected number of collisions would be at least $0.9n$. So we have that at most half of the hash functions are bad. Note that this is a huge overestimate: the only way this can happen (that is, we get the expected value we had before) is if all the bad hash functions have exactly n collisions and the good ones have 0.

So in our first phrase, picking a primary hash function, This is, in the worst case, like flipping a fair coin until you get a heads. So the expected number of trials is $O(1)$ (it is Geometric(0.5)). So, with high probability, the number of trials is $O(\log n)$ (proof?).

Once we have the top-level hash function, then, the Y_i values are determined. In bucket i, we use a hash table of size Y_i^2. We repeatedly pick hash functions for bucket i until we find one that gives no collisions at all.

That is, the secondary hash table at buekct i has Y_i^2 size, Y_i keys. The expected number of collisions is calculated using a similar approach before. We define $\binom{Y_i}{2}$ indicator variables, and the probability that one of those is 1 is $1/Y_i^2$. So defining C_i th enum of collisions at i, we have

$$E[C_i] = \binom{Y_i}{2}(1/Y_i^2) = \frac{Y_i - 1}{2Y_i} \leq \frac{1}{2}.$$

The analysis is identical for every bucket. So cool! We're done, this scheme works.

3.3 Realistic Hash Functions

The tricky thing here is that we've been assuming that we have these idealized has functions that distribute uniformly, and we can generate them easily (and they're independent). What is it like in real life?

When we were talking about idealized hash functions, the associated family corresponds to the family of

$$n^{n^{10}}$$

hash functions (this is the number of functions from $10 \lg n$ strings to n things, I think?). Note if we represent these naively, then we use base-2 log num bits to specify an element. This is probematic, because this description of a function is of length $n^{10} \lg 10$ bits.

The key thing to note here (that'll get us around this)is that pairwise independence is sufficient for the family of hash functions we use. What do we mean? We mean the following: A family of hash functions $\mathcal{H}$ is pairwise independent if, for any distinct keys x, y, and any (possibly equal) bucket indices i, j, if the hash function h is drawn uniformly at random from $\mathcal{H}$, then

$$Pr(h(x) = i \ \& \ h(y) = j) = \frac{1}{B}$$

with B the number of buckets.

This is sufficient because, in our analysis, we were only interested in pairwise indicator variables. We were never interested in any more complicated conditions.

We're interested in designing a family of functions mapping from $(10 \log_2 n)$-bit strings to $(\log_2 n)$-bit strings.

We saw taht for analysis, we didn't need full independence, but just pairwise independence. This is because our analysis cared only about the number of pairwise collisions, which we can write as a sum of indicator variables X_{ij}, which is 1 iff i and j collide. Recall that $P\{X_{ij} = 1\} = 1/p$, with p the number of buckets.

In our analysis, in fact, it's fine to have approximate pairwise independence; that can give us OK bounds. So we want a function h such that, for two distinct keys $x \neq y$ (and any i, j, not necessarily distinct), then

$$Pr\{h(x) = i \wedge h(y) = j\} = O\left(\frac{1}{n^2}\right).$$

IN the above, i and j are bins; x and y are keys.

First, we'll pick a prime p a bit bigger than n^{10} (why?). The prime number theorem tells us that about, picking things around the neighborhood, we only have to try a logarithmic number of keys (in n^{10}) in expectation before finding

a prime. (There is also a theorem indicating that there is definitely a prime between k and $2k$; in particular, this is n^{10} and $2n^{10}$).

So let's hash from $\mathbb{Z}_p$ to $\mathbb{Z}_n$. Consider

$$h(x) = [ax + b \mod p] \mod n$$

with a and b chosen from $\mathbb{Z}_p$. (Note that we have a and b because we want it to be the case that, under our analysis, there will be no "bad input"—if we have an adversary picking keys, they won't be able to pick bad keys if they know what the family looks like; if there were no a and b, then they would be able to do so. In other words, our family needs to have more than one hash function in it if we want to argue a small number of expected collisions.) This isn't quite uniform between 0 and $n-1$, but we're quite close. The inner hash

$$ax + b \mod p$$

is, I think, actually uniform over $\mathbb{Z}_p$ (proof?). The outer hash, then, will be approximately uniform over $\mathbb{Z}_m$ (proof?). This satisfies the condition for approximate pairwise independence that we defined earlier.

So at the top level, we pick a p, then we repeatedly choose a and b; for each a and b, we check how the particular hash function works. We repeatedly do that until we get a good a and b; we store them, as they fully parametrize a hash function. We do this for each primary hash function and each secondary hash function.

4 1/29/2013: Dynamic Programming

4.1 Examples

4.1.1 contiguous subarrays of booleans

Suppose we're given an $n \times n$ boolean array A. We want to find the side-length of a largest all-true contiguous square subarray of A. So if there's a 3×3 subarray of Trues, then the desired answer is at least 3.

Now, certainly this problem is solvable in polynomial time. For each side length k, there are only n^2 different places where the $k \times k$ subarrays top-left corner could be; we could just exhaustively check this for all k and compute the answer.

We can get a better polynomial bound using Dynamic Programming. What DP does is, instead of solving a single problem instance, find a bunch of problem instances to solve, and do so in such a way that the larger problem instances can use the computations performed for the smaller instances.

So let a_{ij} be the size of the largest all-true contiguous square subarray with lower right corner at location (i, j). This is a family of n^2 subproblems. Our answer to the original problem is just

$$\max_{i,j} a_{ij}.$$

Now, what is a good order to solve these n^2 subproblems? What we'd like is that, whenever we get to a particular problem in this ordering, we can very easily solve it in terms of the instances we've already solved. In other words, we want to be able to write a suitable reccurrence for the a_{ij}'s. Thinking about it, it's not too tough to see the following works:

$$a_{ij} = \begin{cases} 0 & \text{if } (i,j) \text{ entry is F} \\ 1 + \min(a_{i-1,j}, a_{i,j-1}, a_{i-1,j-1}) & \text{otherwise} \end{cases} \tag{14}$$

So filling in left-to-right row by row (or top-down column by column) should work. The important thing is that, when we get to a position, the spot above it, to the left of it, above and to the left of it, are filled in. (A small note is that we need to interpret out-of-bounds indices as 0 above.)

So the above algorithm runs in $O(n^2)$ time, which is a nice, much-faster polynomial-time algorithm than the brute-force poly-time algorithm.

4.1.2 Rod-cutting problem

Commonly, Dynamic programming yields polynomial-time algorithms for problems where the brute-force technique is exponential. For example, suppose we have a rod of length n inches ($n \in \mathbb{Z}^+$). We can cut it into any integer-length pieces (such that they sum to n). For every i, we have a price p_i that we can sell a rod of length i for. The problem, then, is to cut up the rod into pieces in such a way that we maximize the total prices for the pieces.

When we cut the number up, we get a multiset of positive integers; in the language of number theory, this is a *partition* of n. If the number of partitions of n were sufficiently slow, we could just generate them and calculate their value; however, the number of partitions grows superpolynomially. Consider the numbers

$$1, 2, \ldots, \lceil 2\sqrt{n} \rceil - 1, \lceil 2\sqrt{n} \rceil.$$

Pair up the first two numbers and the last two numbers:

$$1, 2, \ldots, \lceil 2\sqrt{n} \rceil - 1, \lceil 2\sqrt{n} \rceil.$$

For the first $\lceil 2\sqrt{n} \rceil + 1$ elements, you can either partition them into the two outermost pieces or the next innermost pieces.

We repeat for $3, 4$ and the next two inner numbers on the right. For each of these, we eat up $\sqrt{n}$ of the rod. We'll have $\sqrt{n}/2$ such decisions for what to do with the pieces of length $\sqrt{n}$, and we'll have about

$$2^{\Omega(\sqrt{n})}$$

ways to partition the rods up. (In fact, it ends up being $2^{O(\sqrt{n})}$ too.) So the number of partitions of n is pretty clearly superexponential.

We can solve this, however, with a simple one-dimensional dynamic program. For each i, let v_i be the maximum value of a rod of length i. We will use the

following recurrence for v_i in terms of the smaller v_j's. For a partition of i, there will be some length ℓ of the last partition. So

$$v_0 = 0$$
$$v_i = \max_{1 \le \ell \le i} v_{i-\ell} + p_\ell$$

We get $O(n^2)$ time of this: we have n subproblems, but the ith subproblem takes $O(i)$ time to solve (summing over these subproblems uses the familiar $\sum_i i^2 = O(n^2)$ identity).

4.1.3 Longest Common Subsequence

We have two sequences X, Y of symbols over some finite alphabet. Writing X as

$$X = x_1 x_2 \dots x_n,$$

we define a subsequence of X as a subset of entries x_i, concatenated in order. We want to find the largest integer k such that there are subsequences in X and Y of length k such that both subsequences are the same.

The brute-force approach here is exponential: it's pretty easy to see (there are 2^n subsets of n sets).

We use the familiar two-dimensional DP solution. Let X_i denote the length-i prefix of X:

$$X_i = x_1 \dots x_i,$$

and similarly with Y_j. Now, let a_{ij} be the length of the LCS of X_i and Y_j.

Supposing $|X| = m$ and $|Y| = n$, we have $m \times n$ subproblems. The final answer to the question is a_{mn}. So we have the base cases

$$a_{0j} = 0$$
$$a_{i,0} = 0$$

for all i, j. For the recursive case, we have

$$a_{ij} = \begin{cases} a_{i-1,j-1} + 1 & \text{if} x_i = y_j \\ \max\{a_{i,j-1}, a_{i-1,j}\} & \text{if} x_i \neq y_j. \end{cases} \tag{15}$$