Algorithms Notes

Karl Pichotta

Spring 2013

Abstract

These are notes I've been taking for a graduate algorithms course taught by Greg Plaxton at UT Austin. They're taken primarily in-class, and probably only useful for my own personal reference.

1 Preamble: Useful identities

- For nonzero a, b, c, we have

$$a^{\log_b c} = c^{\log_b a} \tag{1}$$

- For large n, we have

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

$$\left(1 - \frac{1}{n}\right) \approx e^{-1} \tag{3}$$

- For nonzero integer n, we have

$$n^2 = n + 2\binom{n}{2}. \tag{4}$$

  Both the algebraic and intuitive justifications for this identity should be pretty clear.

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

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

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 $\log \log n$ 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

$$Pr(X_1 = k) = \binom{n}{k} (1/n)^k (1 - 1/m)^{n-k} \tag{5}$$

$$\geq \binom{n}{k} (1/n)^k (1/e) \tag{6}$$

$$\geq \left(\frac{n}{k}\right)^k (1/n)^k (1/e) \tag{7}$$

$$= \frac{1}{ek^k} \tag{8}$$

$$\approx \frac{1}{n^{\varepsilon'}} \tag{9}$$

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

$$... = ((1 - 1/n^{\varepsilon'})^{n^{\varepsilon'}})^{n^{1-\varepsilon'}} \tag{10}$$

$$\approx (1/e)n^{1-\varepsilon'} \tag{11}$$

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

$$E[X] = \sum_{i,j} E[Z_{ij}] \tag{12}$$

$$= \sum_{i,j} 1/n \tag{13}$$

$$= \binom{n}{2}/n \tag{14}$$

$$= \frac{n-1}{2} \tag{15}$$

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:

$$a_{ij} = \begin{cases} 0 & \text{if } (i,j) \text{ entry is F} \\ 1 + \min(a_{i-1,j}, a_{i,j-1}, a_{i-1,j-1}) & \text{otherwise} \end{cases} \tag{16}$$

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

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

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

$$\begin{aligned} a_{0j} &= 0 \\ a_{i,0} &= 0 \end{aligned}$$

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

$$a_{ij} = \begin{cases} a_{i-1,j-1} + 1 & \text{if} x_i = y_j \\ \max\{a_{i,j-1}, a_{i-1,j}\} & \text{if} x_i \neq y_j. \end{cases} \tag{17}$$

4.1.4 The Partition Problem (a pseudopolynomial algorithm)

This problem is NP-hard. Suppose we have n positive integers $x_1, \ldots, x_n$. We want to know whether the x_i's can be partitioned into two multisets of equal sum.

We give a DP algorithm. First, let $\sum x_i = 2s$. (Note if the sum is odd, we return that there is no such partition). To determine s, it' a natural thing to consider computing

$$a_{i,j} = \begin{cases} T & \text{if there's a subseq. of } x_1, \ldots, x_i \text{ summing to exactly } j \\ F & o.w. \end{cases} \tag{18}$$

Note $\forall i$, $a_{i,0} = T$. Further, $\forall j > 0$ $a_{0,j} = F$. When $i, j > 0$, we have

$$a_{i,j} = \begin{cases} a_{i-1,j} & \text{if } j < x_i \\ a_{i-1,j} \vee a_{i-1,j-x_i} & o.w. \end{cases} \tag{19}$$

NB we only really need the first term for the technicality of the subscript on the RHS disjunct being negative.

Now, the number of table entries is $O(nS)$. Is this polynomial time? Intuitively, no—S can be very large. We argue that the algorithm is not polynomial; to do so, we need to find one family of inputs where the runtime is not upper-bounded by a polynomial function of the input in bits.

What do we mean by polynomial time? Well, we upper-bound the runtime of an algorithm according to a polynomial function of its input in bits. What's the input of this like? Well, consider an input where each x_i is an n-bit integer. The input size for this problem is $\Theta(n^2)$ bits (we have n n-bit numbers). For us to claim that we have a polynomial runtime, we need that, for such inputs, the runtime is polynomial in n. S, however, is $O(n2^n)$, since an n-bit integer can be as large as 2^n in value (and we have n of them). So, more importantly, S can be $\Omega(n2^n)$. So this algorithm, which kinda looks like it's polynomial, actually isn't polynomial in the input size.

However, sometimes these sorts of algorithms are useful. If we know, for example, that each of the numbers is at most n^3, then we have that S is at most n^4, and we have a polytime algorithm. We call algorithms like this *pseudopolynomial*: they are polynomial in the input size if the input integers are represented in unary.

5 1/31/2013: Greedy Algorithms

5.1 A Scheduling Problem

Suppose we have n tasks, each with a positive integer deadline and an execution requirement. So an input may look like

task	Deadline	execution requirement
A	8	5
B	6	2
C	9	2

We want to know if we can meet all the deadlines in a nonpreemptive schedule? So, in the above, suppose we run B at time 0 and C at time 2; both of these take time 2, and have met their deadlines. If we run A next, though, it'll miss its deadline. However, if we run in the order BAC, we get a feasible schedule.

We may want to try to get at this by Dynamic Programming, but we don't need it: a greedy solution ends up sufficing. We use *earliest deadline*, breaking ties arbitrarily. This will yield a feasible schedule if one exists.

We want to argue that the answer to the greedy algorithm is right iff there's a feasible schedule. Clearly, Left to Right is easy. The argument is R to L, then: If there's a feasible schedule, then we need to show the greedy algorithm finds it.

So take a feasible schedule. Assume the feasible schedule has no "spaces": there's no advantage to idle time, so everything is "squeezed" as far as possible to the left. Suppose that we have a schedule $ABCD$ of four elements, in that order. Suppose that C has an earlier deadline than B; earliest deadline would have chosen the order $ACBD$, rather than $ABCD$.

We can modify the schedule by leaving everything the same except for B and C, swapping the latter two. Note the new schedule is clearly the same length. We want to show that $ACBD$ is also feasible. Clearly C meets it deadline still—it's moved earlier. We need to show that B is meeting its deadline. Since, by assumption, C had a more stringent deadline than B, it must be the case that if B finishes when C did before, it certainly meets its deadline, since it meets C's deadline. In other words, before, with $ABCD$, C was meeting its deadline; we know that B's deadline is later than C's, so in $ACBD$, B definitely meets its deadline.

We apply the same logic to the new problem instance to argue that it's feasible (eventually we will get to the earliest-first schedule).

5.2 A variation of the scheduling problem

Suppose that, in addition to the deadline and execution requirements, each task has a positive profit $p_i > 0$ associated with it. Our goal is to find a max-profit feasible subset (that is, we won't in general use the entire set of tasks). What we'll do is combine the greedy approach with dynamic programming.

We first order tasks by deadline (in nondecreasing order). We then consider all prefixes with respect to that particular ordering. The DP's subproblem is $a_{k,t}$, defined as the max profit we can obtain with a schedule using only the first k tasks and time $\leq t$. We can write a recurrence fairly straightforwardly (left as an exercise).

The runtime of the above will be polynomial if the execution requirements are polynomially-bounded. That is, if every task has a task that is $O(n^c)$ for some c. Without such an assumption, we could get an exponentially-sized table in our DP

5.3 Matroids

Captures a large class of greedy algorithms. Consider the MInimal Spanning Tree problem, in particular, Kruskal's greedy algorithm. One way to convince ourselves that Kruskal's algorithm is correct is to reason directly about minimal spanning trees. Another is to frame the algorithm as a matroid problem, and use a general property about Matroids.

A Matroid is a pair

$$M = (S, \mathcal{I})$$

with S analogous to the vertices in a graph: it's just a set. $\mathcal{I}$ is called the *independent sets* of S: $\mathcal{I} \subseteq 2^S$ (elements are subsets of S). We need two more properties:

1. **Hereditary Property**: if $A \in \mathcal{I}$ and $B \subseteq A$, then $B \in \mathcal{I}$.
2. **Exchange Property**: If $A, B \in \mathcal{I}$ and $|A| > |B|$, then there is some $x \in A \setminus B$ such that $B \cup \{x\} \in \mathcal{I}$.

The hereditary property tells us that removing elements from an independent set yields an independent set. In particular, this tells us that $\emptyset \in \mathcal{I}$.

Define a maximal independent set as an independent set such that, if you add any additional elements to it, it will no longer be independent. The exchange property tells us that all maximal independent sets will have the same cardinality. A maximal independent set is sometimes called a **basis**

In a **weighted matroid**, each element $x \in S$ has an associated weight $w(x)$. In applications, we often want to compute a maximum (or minimum) weight maximal independent set. For example, eventually we'll have a minimal spanning tree representing a minimum weight maximal independent set (the maximal independent sets will correspond to the spanning trees).

5.4 Matroid Greedy Algorithm

Suppose we want to do maximization.

- First, sort the elements of S in nonincreasing order of weight (if we are minimizing, we'll sort in the other direction).
- Initialize $A := \emptyset$
- For each $x \in S$, in the order determined in the first step:
  - If $A \cup \{x\} \in \mathcal{I}$, then $A := A \cup \{x\}$.

And that's it. Thinking about Kruskal's algorithm, the set $\mathcal{I}$ will be the set of acyclic graphs—the conditional above corresponds to the conditional in that algorithm.

How do we show this algorithm is correct? First, index the elements of S $1, \ldots, n$ according to the ordering in the first step. Call the weight for element i in this ordering w_i. For each i, calculate some max-weight maximial independent

set B. Mark down if element i is in B or not. Also, for each i, consider A, the set computed at that step by the algorithm. What we need to do is prove, for each step, A is a maximal weight independent step. Similarly, look at whether elem i is put in A.

Look at the first part where the sets A and B differ. It can't be the case that the elem i is in B but not in A (proof? It has somethign to do with the hereditary property, not sure what). So the first place where the two differ is that an element i must be in A and not in B. WE note the first place i where A and B differ. Construct $A' \subset A$ from all the before i in the ordering, and also elem i. Now, we know $|B| \geq A'$, since $|A'| \leq |A|$, and $|B| = |A|$, by the exchange property.

We repeatedly apply the exchange property to add elements from B to A'. The exchange property tells us that we can add an element from B to A' and get an independent set, so long as $|A'| < |B|$. This process terminates exactly when $|A'| = |B|$.

When we're done growing A', B gave A' all its elements from $i+1$ onwards except for one of them. In other words,

$$A' = (B + i) = j$$

for some j. That is A' has elem i while B doesn't, and A' deosn't get some $j > i$. However, since the elements are ordered by weight, and $w_j \leq w_i$, we can conclude that

$$w(A') \geq w(B).$$

Now we play the whole game over, but with B replaced by A'. This gets us "one step" in the right direction, and we iterate this process until we end up with $A' = A$, at which point we can argue that A was a max-weight maximal independent set.

5.5 Some examples of Matroids

- **Uniform matroid**: Let $|S| = n$. Fix $0 \leq k \leq n$. $\mathcal{I}$ is the set of all subsets of S of size $\leq k$. This is a matroid. The hereditary property is pretty obvious. The exchange property is also pretty clear: if we have A and B such that $|A| < |B|$, then there's clearly an element in B that's not in A such that adding it to A will result in an element of cardinality $\leq k$.

- **Partition matroid**: Fix a partition of S into $\{S_1, \ldots, S_k\}$. Define $\mathcal{I}$ to be the set of all subsets $A \subseteq S$ such that $\forall i |S_i \cap A| \leq 1$.

  A basis (maximal independent set) for this matroid would be a set that chooses exactly one element from each S_i. The number of such bases is

  $$\prod_{i=1}^{k} |S_i|.$$

  The hereditary property is pretty clear. The exchange property is a bit trickier, but also pretty clear.

- **Scheduling problem**: we have n tasks numbered from 1 to n. Task i has a positive integer deadline d_i, a positive weight w_i (the profit of the task, say), and an execution requirement of 1 (unit tasks). We're scheduling these tasks on a shared resource.

  What is the maximum-weight schedulable subset of the tasks (in general, we might not be able to meet all tasks of the schedule)? This is a bit complex: Suppose there are 2 tasks with deadline 1, and two tasks with deadline 2. We can schedule at most one of the former, but two of the latter (but only if we schedule neither of the first two).

  This problem ends up having a useful matroid structure, allowing us to use the matroid greedy algorithm.

  Define the set S as the set of n tasks. $\mathcal{I}$ will be the schedulable subsets of the tasks. If this works, our optimal solution will certainly be a maximal independent set (this isn't necessarily true if we have negative weights I don't think).

  Now, to show $(S, \mathcal{I})$ is a matroid. The hereditary property is pretty clear. The exchange property is a bit trickier. Suppose $A, B \in \mathcal{I}$, with $|A| > |B|$. We proved earlier that A is schedule iff any ED schedule (???) of A is feasible (wtf does this mean?). Define a "canonical" ED schedule as one of the ED schedules: break ties in a particular manner. That is, we're constructing a schedule of a task A; we list tasks in increasing order of their index or something like that, guaranteeing a unique tie-breaking schedule. The schedule, then, will be some ordering

  $$A = x_{i_1}, x_{i_2}, \ldots, x_{i_{|A|}}$$

  and

  $$B = x_{i_1}, x_{i_2}, \ldots x_{i_{|B|}}.$$

  Note that there will be no "empty spaces" in the schedule.

  Let's focus on the last task appearing in A that isn't a part of B. Define z to be this tasks: the last task in the canonical ED schedule of A that does not belong to B. To show the exchange property, we need to show there's something in A that can be added to B. We'll claim that z is such an element; that is, $B + z \in \mathcal{I}$ (that is, $B + z$ is schedulable).

  - **Case 1**: z is the last task in the ED schedule of A. Because B is shorter, we can just tack z onto the end of B; it will surely meet its deadline, because it will start at least as early as it started in A. This isn't necessarily an ED schedule, but is feasible.
  - **Case 2**: The ED schedule of A ends with $z, y_1, \ldots, y_\ell$ with $\ell \geq 1$ (that is, z isn't the last element of A's ED schedule). What we do is jimmy around with the schedule of B to arrive at a feasible schedule for $B + z$. Leave the schedule before y_1 alone; put z in the spot where y_1 is. Find y_2 in B's schedule, and put y_1 in it; similarly, put

$y_{\ell-1}$ into the schedule where y_ℓ was, and tack y_ℓ onto the end (recall, these are the occurrences of $y_1, \ldots, y_\ell$ in B's ED schedule. Now, y_ℓ in the new schedule for B is no further to the right than it is in A's schedule. Similarly, the furthest to the right that $y_{\ell-1}$ could possibly be is where it is in A's schedule. So this is a feasible schedule (KBP: i don't see why $y_1, \ldots, y_\ell$ are in B's schedule. Maybe they don't have to be?).

- **Scheduling with release times**: a more complicated version of the above. Assume each task also has a positive integer **release time**. Previously, we assumed all tasks were available at time 0; we restrict that assumption now: tasks become available to executable at a certain point in time, and can only be scheduled past that time.

  Even with this additional constraint, we can construct a matroid structure (we don't here).

- **Transversal matroids**: even more general. Say we have a bipartite graph (U, V, E) (with U, V the two sets of vertices such that $\forall (u, v) \in E$, u and b aren't both in U or V). Consider the matroid $(U, \mathcal{I})$, with every $S \in \mathcal{I}$ a "matchable" subset of U, defined below. A **matching** is a subsets of the edges E that induces degree at most 1 on every vertex. That is, in a matching has no vertex incident on more than 1 edge (that is, every vertex has degree ≤ 1). Given a matching, call a vertex **matched** if it's incident on one of the edges in the matching. A subset U of vertices is **matchable** if there exists some matching matching every $u \in U$.

  So our matroid is $(U, \mathcal{I})$, with U the same set in our bipartite graph $(U, V, \mathcal{I})$ (proof that this is a matroid?). Relating to the last scheduling problem: Consider U the set of tasks, and V the set of times during which a task can start. So supposing task i has deadline of 5 and a release time of 2 (and takes 2 time units), we'll add edges from $i \in U$ to $2, 3, 4$ in V. Given this setup, every matchable subset corresponds to a valid scheduling.

  The hereditary property is easy: if a set of tasks is matchable, of course any subset is matchable.

  The exchange property is tricky: say we have $A, B \in \mathcal{I}$, $|A| > |B|$. Let M_A be a matching matching exactly A (which exists by assumption), and let M_B be a matching for B. That is, the set of nodes matching in M_A is exactly A (so $|M_A| = |A|$), ditto B. We need to show that there is some M matching $B + x$, for some $x \in A \setminus B$. Think of these matchings as sets of edges.

  Consider the graph $G \equiv (U, V, M_A \Delta M_B)$ (taht is, the symmetric difference of M_A and M_B: edges in exactly one of M_A, M_B). Because M_A is such that each node is of degree at most 1, we have that $M_A \Delta M_B$ induces degree of at most 2. G consists of vertex-disjoint paths and cycles (the cycles are of even length; each cycle would alternate between an edge in

M_A and and edge in M_B—you can't have two consecutive edges from M_A, because this would give us a node of degree 2).

We cannot have only cycles: since we know that $|M_A| > |M_B|$, we must have some paths (since every cycle draws an equal number of edges from M_A and M_B, I think). In particular, there must be at least one odd-length path that begins and ends with an edge in M_A. Note that the paths, like the edges, alternate from edges in M_A and edges in M_B. So assume WLOG that this path starts in U and ends in V (it's odd-length, and a bipartite graph, so an odd-length path can't begin and end in U or V). Call the node in U that the path starts at x.

We can obtain a matching for $B+x$ by making a slight hack on M_B. Take the odd-length path above; get rid of the original edges from M_B, and add the edges from M_A that were touching the same nodes. This doesn't lose matchability (every one of the nodes on the path is still connected to exactly one edge, this time the edge from M_A), and also now x is matched.

It's worth noting that the computational problem to solve the matroid greedy algorithm is trickier for this formulation: we have to determine matchability for arbitrary subsets, which is a little tricky.

- **Graphic matroids**: Fix an undirected graph $G = (V, E)$. Define $M = (S, \mathcal{I})$, with $S = E$ and $\mathcal{I}$ the set of all acylic subsets of E. The hereditary property is pretty obvious: removing edges won't add cycles.

  We don't have time for the exchange property in class.

6 2/7/2013: Minimal Spanning Trees and Matroids

Recall that, given an undirected graph $G = (V, E)$, we construct a graphic matroid from it by the set $(E, \mathcal{I})$, with $A \in \mathcal{I}$ iff A is acyclic. Here, the Maximally independent set are exactly the spanning forests; if G is connected, then it is a spanning tree.

Note that a **minimal dependent set** of a matroid $(S, \mathcal{I})$ is a dependent set (that is, a subset of S not in $\mathcal{I}$) such that if we remove any element from it, we get an independent set. If we have a graphic matroid, then a minimal dependent set will be a cycle (we can't throw any edge out of the set without getting something that's acyclic).

6.1 A theorem and some corollaries about Matroids

Lemma: let $M = (S\mathcal{I})$ be a weighted matroid. Let A, B be distinct max-weight bases of M. Then there exists a sequence $A_1, \ldots, A_k$ of max-weight bases of M such that $A = A_1, B = A_k$, and for every i with $1 \leq i < k$, we have

$$|A_{i+1} \setminus A_i| = 1.$$

That is, A_{i+1} and A_i differ by a "swap":

$$A_{i+1} = A_i + x - y$$

with x and y having equal weight (since they're all max-weight).

How do we prove this? One way is to use something similar to our proof of correctness for the matroid greedy algorithm. Well, we order the elements (x_i, w_i) by decreasing order of weight, and then we line up A and B, marking whether $x_i \in A$ and $x_i \in B$. We look at the first element such that $x_i \in A$ but $x_i \notin B$, and, we obtain B' by growing an A^* or something, starting with A^*, and growing to a basis via repeated application of the exchange property with B. So we'll end up with B', which will differ from B by a single swap. There'll be one element that B does have but that B' doesn't have; other than that, B' (which is, I think, what you get after constructing as many A^*s as you can) and B will be identical.

Since A and B are both max-weight bases, what can we conclude about the basis B'? We have that $w(B') \geq w(B)$; we can get B' from B by exchanging the elements that differ, and the one in B' is at least as heavy as that in B (why?!). We cannot have $w(B') > w(B)$, since B is a max-weight basis.

So we will have obtained a third max-weight basis that agrees on a longer prefix with A than B did. We can do this again, getting a max-weight basis B'' differing from B' by a single edge swap; eventually, we will end up with something equal to A. We will therefore have exhibited a sequence of the desired form: Something starting at B and ending up at A, where each sequence is a max-weight basis differing from the previous by a single element.

Corollary to above: all max-weight bases have the same distribution of weights (that is, taking just the element weights and sorting it will yield the same list). (You might think that, e.g., you can have two minimal spanning trees such that one has an edge of weight 10 and 5, and the other has instead two edges of weight 8 and 7; this corollary rules out that possibility.)

Another corollary: Every max-weight basis is a possible output of the matroid greedy algorithm (that is, will be produced if ties in weight are resolved appropriately). (You might think that, in Kruskal's algorithm, for example, that you can have a graph with a couple MSTs such that one of them will never be produced; this isn't possible, by this corollary. There will be a tie-breaking that will cause every MST to be produced.) It's pretty easy to see what tie-breaking is appropriate: break ties by giving priority to elements in our desired max-weight basis.

A third corollary: if the weights are all distinct, then there is a unique max-weight basis. Cool!

6.2 Matric Matroids

There is another class of matroids called **"matric matroids"**, defined in terms of a given matrix A. Define such a matroid $M = (S, \mathcal{I})$, with S the columns of A, and $\mathcal{I}$ the sets of linearly independent columns of A. You can't model all

matroids as matric matroids, but a bunch of the classes of matroids discussed above can be modeled as matric matroids.

It's useful, in general, to consider these special classes of matroids, rather than considering matroids in full generality, because we can often get faster algorithms for specific matroids.

7 2/7/2013 (cont): Amortized Analysis

Amortized analysis is typically used in settings where the cost of operations is nonuniform: we ask, if we spend k operations, what sort of bounds can we put on performance. So if we have some operations that are much more expensive than others, but the overall cost is less than the pessimistic estimate (wherein we upper-bound every single operation by the worst-case performance).

There are different methods of amortized analysis:

1. **Aggregate method**: Simply sum up the cost of all the individual operations to bound the total cost of a sequence of operations. This is the most basic approach, but can get kind of messy.

2. **Accounting method**: Suppose we're trying to prove a theorem that any sequence of k operations will take $O(k)$; then, when an operation is performed, we'll want to give an operation a constant amount of dollars in order to do it (we'll give an operation \$10 to fund its execution, if we're trying to bound the operations at amortized \$10. The actual semantics are that an operation uses the money we give it; if it uses less than we give it, it puts the rest of the money into different piles of money in the data structure. When an expensive operation takes a lot, then it uses the leftover piles of money from the preceding operations.

   If the money we inject into a system is enough to pay for all the operations, then we can get bounds on the running time.

   Note that it'll frequently make sense to convert this approach to the potential approach, described below (where the potential corresponds to "amount of money left in data structure").

3. **Potential method**: (sometimes, potential function method): A more general method, emphasized more in this course. We define a potential function mapping the state of the data structure to a (usually nonnegative) number. The amortized cost of an operation will be the actual cost plus the change in potential.

   So if an operation changes the state of the data structure from D to D', then the change in potential is $\phi(D') - \phi(D)$, with ϕ the potential.

   We'll use the potential function to make sure the amortized costs all look similar, making it easy to add up the amortized cost.

This is tricky—what we really care about is the total cost, rather than the total amortized cost (which is this funny function of the actual cost that we made up). However, summing things up makes it make sense:

Suppose the state of our data structure before/after each of our operations is $D_0, \ldots, D_k$. When we sum the total cost, we get the potential telescoping: the sum of the potentials is

$$\phi(D_1) - \phi(D_0) + \phi(D_2) - \phi(D_1) + \ldots$$

which gives us that the total amortized cost is the total cost plus the final potential ($\phi(D_k)$ minus the initial potential ($\phi(D_0)$).

Usually, the initial potential is 0, and the final potential is typically non-negative; this means that the total amortized cost, according to this method, will actually be an overestimate of the total cost (since total amortized cost = total cost + final potential).

7.1 Three Examples

7.1.1 Push/multipop stack

We have a cheap operation, pop (push a single item from the stack), and a potentially expensive one, multipop, which pops k times (taking time proportional to k).

So a push has actual cost 1, and multipop(k) will have cost k; multipop(k) can only be performed on a stack with at least k elements. What is the cost of a sequence of n operations starting with the empty stack?

- **Naive approach**: At all times, we cannot have more then n elements on the stack. No single operation costs more than n: at most, we can pop n elems at any time. Since we have n operations, the total cost is $\leq n^2$.

  This is a very weak upper bound: we'll see that our handful of expensive operations must be offset by a bunch of cheap operations.

- **Aggregate Method**: The total cost of all multipops must be $\leq$ the total cost of all pushes. If we ever multipop off 10 items, we can only do that if, previously, we'd pushed all of them on, at a total cost of n. Since the total cost of all pushes is $\leq n$ (we perform at most n pushes of cost 1), we have that the total cost is at most $2n$.

- **Accounting method**: Give the system \$2 for each push, and \$0 for each multipop. We'll want to show that the pushes have amortized cost 2 and the multipops have 0 amortized cost; this would imply that the total amortized cost is at most $2n$.

  When a push happens, it has to take one dollar; it puts the extra dollar onto the pile. At any point, a stack of m elements will have \$$m$ sitting around. In fact, we can think of each element keeping its dollar, and so we can easily prove taht each element has one dollar associated with it, and so we'll be able to fund every operations.

- **Potential method**: Define ϕ of a stack to be the number of elements on the stack (notice that this is basically the same as the accounting argument). Then it's easy to show, using the argument we just gave, that we can bound the cost of operations linearly.

7.1.2 Incrementing a binary counter

We have a counter that has the number 0 initially, and we have only one operation: increment counter. We increment in the most elementary way: start at the LSB, flip and possibly carry, and continue until you hit a 0. So the increment operations are occasionally expensive (if you have to flip a lot of 1's to 0's), but they're usually cheap. We'd hope that, over a large sequence of k increments, our total cost would be $O(k)$.

Concretely, we imagine our counter having an infinite number of bits going off to the left. The first increment will take cost 1; the next will take 2 (we have to flip 2 bits); the third will flip 1 bit; the next 3; and so on.

- **Aggregate method**: the total cost of n operations will give us taht roughly 1/2 the operations will have cost 1 (every other number is even); 1/4 of the ops will cost 2; 1/8 will cost 3, and so on. So the total cost is

$$(n/2) + 2(n/4) + 3(n/8) + \ldots \leq \sum_{i \geq 0} \frac{i}{2^i} = 2$$

  (the last equivalence is somewhat nontrivial)

- **Accounting method**: \$2 / operation. We'll need to maintain a different invariant from the stack: we might think that we'd have a dollar per bit, but that won't end up working. We instad have \$1 for each 1-bit in the counter: if our number is 001011, then we have \$3. It's not tough to see this works: incrementing the counter, if we have k bit flips, we had to convert $k-1$ 1's to 0's; we grab that dollar going around, and then deposit our extra dollar on the 1.

- **Potential method**: ϕ will is the number of 1s present in the counter.

7.1.3 Dynamic Array

The canonical example: suppose we're using an STL vector or a Java ArrayList. The underlying implementation doesn't know how much space is needed; we'll reserve space, but from time to time, the application will exceed the bounds, and a large cost will be incurred. So sometimes a single push can be very expensive, but we'll want to determine that the total cost of a sequence of operations is, say, $O(k)$.