CS 125 **Course Notes 1** Fall 2014

1.1 Sorting

Today we're going to start with sorting, which is a bit strange, because I usually think of sorting algorithms as a bit boring. But we're going to find that looking at sorting leads us to some basic, interesting questions regarding how we think about computation, both in terms of what is possible and what is not possible. And, of course, sorting is fundamental throughout computer science.

One of the basic questions that we ask about algorithms is: how fast are they? In the context of sorting, we can ask how fast can we sort a list (or array) of n numbers. (For convenience assume we'll sort into increasing order; where convenient we'll assume the n numbers are distinct.)

1.2 Simple Algorithms

1.2.1 BubbleSort

As I imagine many of you know, there are many algorithms that can sort n numbers using a quadratic number of operations. (As we'll see, CS people like to write this as $O(n^2)$ operations; don't worry if you don't know that notation yet.) Here's one such algorithm.

The Bubblesort algorithm is very simple. (You can read more about it on Wikipedia, including various pseudocode.) One way of viewing it is to compare the first and second elements, swapping the order if necessary. Then compare the second and third, then the third and the fourth, and so on, until you get to the end. Now repeat this process n times.

Why does this work? It's easy to check that the first complete pass moves the largest element to the end, and inductively, after k passes, the k largest elements will be at the end. (In fact, you can optimize based on this; you know the k elements are at the end after k passes, so you only have to compare the first $n-k$ elements on the $(k+1)$st pass.) There are at most n^2 comparisons, and therefore at most $O(n^2)$ operations, including swaps, assuming every operation can be done in unit time.

Example:

7 5 9 3 1

7 **5** 9 3 1

5 **7** **9** 3 1

5 7 **9** **3** 1

5 7 3 **9** **1**

5 7 3 1 9

5 **7** 3 1 9

5 **7** **3** 1 9

5 3 **7** **1** 9

5 3 1 7 9

5 **3** 1 7 9

3 **5** **1** 7 9

3 1 5 7 9

3 **1** 5 7 9

1 3 5 7 9

1.2.2 MergeSort

So we can sort in $O(n^2)$ operations. Can we do better? Yes. There are a number of algorithms that can sort in just $O(n \log n)$ operations. (Note that asymptotically $n \log_2 n$ is much much smaller than n^2; the ratio $n \log_2 n / n^2$ goes to 0.) One of simplest is mergesort:

```
function mergesort ($s$)
    list $s$, $s_1$, $s_2$
    if size($s$) $= 1$ then return($s$)
    split($s, s_1, s_2$)
    $s_1 =$ mergesort($s_1$)
    $s_2 =$ mergesort($s_2$)
    return(merge($s_1, s_2$))
end mergesort
```

Break the list into two equal-sized (or as equal as possible) sublists, sort each sublist recursively, and merge the two sorted lists (by going through them together in order). A merge function is given below (using basic list operators push and pop).

```
function merge $(s,t)$
    list $s,t$
    if $s = [\,]$ then return $t$
      else if $t = [\,]$ then return $s$
      else if $s(1) \leq t(1)$ then $u$:= pop($s$)
                              else $u$:= pop($t$)
      return push($u$, merge($s,t$))
end merge
```

Let $T(n)$ be the number of comparisons mergesort performs on lists of length n. Then $T(n)$ satisfies the recurrence relation $T(n) \leq 2T(n/2) + n - 1$. (If n is odd, then really $T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$; not a significant difference.) This follows from the fact that to sort lists of length n we sort two sublists of length $n/2$ and then merge them using (at most) $n - 1$ comparisons. Using the general theory on solutions of recurrence relations, we find that $T(n) = O(n \log n)$.

Exercise: Prove directly that the recurrence $T(n) \leq 2T(n/2) + n$ with $T(2) = 1$ yields $T(n) \leq n \log_2 n$ for n a power of 2 by expanding the recurrence directly.

1.3 Lower Bounds

So far we've been considering what we *can do* – how fast we can sort. Now we want to think about something more challenging – are there limits to how fast we can sort? This is more challenging because when we ask how fast, we just need to consider and analyze one algorithm. When we ask about limits, we seek a lower bound for *all possible algorithms*, or at least all possible algorithms in a certain class of algorithms. Trying to prove a statement about all possible algorithms can be, as you might imagine, difficult.

Let us try to frame the problem in order to obtain a lower bound. The BubbleSort algorithm and the MergeSort algorithm are both based on comparing elements to put them in the right place. How many comparisons are needed to sort n elements correctly every time?

Formally, let us consider algorithms that start with an array $A = (a_i)$ and *only* do two kinds of operations; they compare elements ("is $a_i > a_j$"), and they move elements.

Our goal will be to prove the following theorem:

Theorem 1.1 *Any deterministic comparison-based sorting algorithm must perform* $\Omega(n\log n)$ *comparisons to sort n elements in the worst case. More specifically, for any such algorithm, there exists some input with n elements that requires at least* $\log_2(n!)$ *comparisons.*

Proof: We'll describe two ways of thinking about this proof (they're equivalent, but one might be easier to think about than another for you). For convenience, let us assume our input is always just a permutation of the numbers from 1 to n. (Notice that since our algorithm just does comparisons, the "size" of the numbers in the input doesn't really matter; as long as we have distinct numbers, this argument holds.)

Let us define the "state" of our algorithm as a list of all the comparisons that have been done along with their outcomes. We do not have to worry about the current order of the elements for our "state", since we are only considering comparisons in our lower bound; the algorithm could always move any of the elements at any time "for free" based on the state given by the comparisons, so we can without loss of generality save the moves to the end.

The algorithm cannot terminate with the same state for two different permutations, because if it did, it would necessarily give the wrong answer on one of them. But if the algorithm does at most k comparisons, there are at most 2^k different possible states the deterministic algorithm could end up in, because there are 2 outcomes for each comparison. And there are $n!$ different permutations. Hence if k is the largest number of comparisons done for any possible input we must have

$$2^k \geq n!.$$

This gives the theorem.

Here's an alternative view, which might seem easier if you like the game 20 Questions. At the beginning, there are $n!$ possible input permutations. Each time the algorithm does a comparison, it determines information that makes some of those inputs impossible. Let X_j be the set of initial permutations consistent with the information that the algorithm has determined after j comparisons. For the algorithm to work correctly, the algorithm must ask enough questions so that $|X_j| = 1$. Initially, $|X_0| = n!$.

Each comparison splits X_j into two subsets, call them Y_j and and Z_j, depending on the outcome of the comparison. Now suppose an adversary just tries to make the algorithm take as long as possible, by returning the answer to the comparison that corresponds to the biggest set of Y_j and Z_j. In that case,

$$|X_{j+1}| = \max|Y_j|,|Z_j| \geq |X_j|/2.$$

It follows that there's some permutation the adversary can use to ensure that at least $k = \lceil \log_2(n!) \rceil$ comparisons are used by the algorithm before it can get to a point where $|X_k| = 1$. ■

Usually, this result is interpreted as meaning that we have tight bounds for sorting. There are algorithms that take $O(n\log n)$ comparisons, and that many comparisons are required. But this is a limited point of view. The result has shown that algorithms based on comparisons in the worst case can require $\Omega(n\log n)$ comparisons. So if we want to do better, we will have to *change the problem* in some way. One way to change the problem is to assume something about the input so that it is not worst case, and we will give some examples of that. But another way to change the problem is to consider operations other than comparisons. That is, we change and enrich our model of computation.

1.4 Counting Sort

Suppose that we restrict our input so that we are sorting n numbers that are integer values in the range from 1 to k. Then we can sort the numbers in $O(n+k)$ space and $O(n+k)$ time. Notice that if k is "smaller than" $n\log n$ – in notation we'll discuss, if k is $o(n\log n)$, then this is a sorting algorithm that breaks the $\Omega(n\log n)$ barrier. Naturally, this means counting sort is not a comparison based sorting algorithm.

Instead, we build an array *count* keeping track of what numbers are in our input. (More generally, if we can have multiple copies of the same number, we build a histogram of the frequencies of the numbers in our input. See the Wikipedia page for more details – here we'll assume the elements are distinct.) If our input is $a_1, a_2, \ldots, a_n$, we keep track of counts; for each a_i, we increase it's count:

$$\text{count}[a_i] += 1.$$

Now we walk through the count array, updating it via what is called a prefix sum operation, so that the count$[j]$ contains the number of inputs that are less than or equal to j.

Finally, we walk back through the inputs; for each a_i, it's sorted position is now count$[a_i]$. The two passes through the input take $O(n)$ operations; the pass through the count array to compute the prefix sum takes $O(k)$ operations.

Here we are making assumptions about the input, and expanding our model of computation. Instead of just comparisons, we're accessing a non-trivially sized array. In particular, we are assuming that we can maninpulate values with $\lceil \log_2 k \rceil$ bits in one unit of time, and use these values to do things like access arrays. If k for example is n, this means we are assuming we can work with number of size $\log_2 n$ bits.

1.5 Interlude on Models of Computation, Complexity, and Computability

As the discussion above suggests, in order to precisely specify an algorithm, we need to precisely specify a *model of computation* — what kind of operations are allowed, and on what kind of data? Having a precise model is particularly important if we want to be able to understand the limitations of algorithms — what is impossible for algorithms to achieve. An example is the above proof that any comparison-based algorithm for sorting requires $\Omega(n\log n)$ comparisons. But the fact that we can beat this lower bound by working in a more general, but still very reasonable, model of computation, suggests that the choice of model might be very important.

Can there be a robust theory of computing, that is not sensitive to arbitrary details of our model of computation? Remarkably, the answer turns out to be yes. In a few weeks, we will see that there is a simple model of computation, known as the Turing machine, that can compute the same functions as any reasonable model of computation. It was formulated in the 1930's by Alan Turing, motivated by questions in the foundations of mathematics (related to Godel's Incompleteness Theorem). It inspired the development of actual physical computers, and its relevance has not diminished despite all the changes in computing technology. Using the Turing machine model, we will prove that there are some well-defined computational problems that cannot be solved by any algorithm whatsoever. For example:

- Given a computer program (in your favorite language), can it be made to go into an infinite loop?
- Given a multivariate polynomial equation $p(x_1, \ldots, x_n) = 0$, does there exist an integer solution?

Now, as we all know when we purchase a new computer, the choice of the computational model (= computing hardware) does make a difference, at least to how *fast* we can compute. But, it turns out that it does not make a very big difference. In particular, while it may make the difference between $O(n\log n)$ and $O(n)$ time (as we saw above), it will not make a difference between polynomial and exponential time. This leads to a robust theory of *computational complexity*, which we will also study. Here many questions are open (notably, the famous P vs. NP question, widely considered one of the most important open problems in computer science and in mathematics) — there are many problems conjectured to require exponential time, but for which we still have no proof. But the beautiful theory of NP-completeness allows us to show that a vast collection of important problems all have the same complexity: if one has a polynomial-time algorithm, all do, and if one requires exponential time, all do.

1.6 Bucket Sort

For bucket sort, we again make assumptions about the input – this time, we assume the inputs are randomly selected according to some distribution. Because of this "randomness" in the input, we consider the "expected time" to sort instead of worst case time. Here we assume that the n input numbers are uniform over a collection of numbers in the range $(0,M]$, and show that the expected time to sort is $O(n)$. (More general results are possible.)

We put the numbers into n buckets of size M/n, with the numbers 1 to m/n going in the first bucket, the next M/n numbers going in the second bucket, and so on. We then sort by using BubbleSort on each bucket, and then concatenating the results from each bucket in order. If X_i is the number of elements that land in the ith bucket, the expected number of comparison operations to sort T is given by:

$$\begin{aligned} E[T] &\leq E[\sum_{i=1}^{n} X_i^2] \\ &= \sum_{i=1}^{n} E[X_i^2] \\ &= nE[X_1^2]. \end{aligned}$$

The first line follows from the number of comparisons for BubbleSort-ing j items being bounded above by j^2. The second line follows from linearity of expectations, and the last line is from symmetry. Now the probability any given element goes in the first bucket is $1/n$. Hence

$$E[X_1^2] = \sum_{k=0}^{\infty} \binom{n}{k} \left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k} k^2 \leq \sum_{k=0}^{\infty} \frac{k^2}{k!}.$$

This last term is bounded by a constant. (One can check it equals $2e$.)

Exercise: Show that $E[X_1^2] < 2$. Note: this requires a more sophisticated approach to bounding $E[X_1^2]$.

1.7 Van Emde Boas trees

A Van Emde Boas tree (vEB tree) implements a data structure that acts like an associative array, keeping track of a set of numbers. That means you can search for an element, insert an element, or delete an element. In fact, you can ask queries where you given an element that may not be in set, and the structure will return the smallest element in the set that is larger than the query element, or you can ask for the largest element in the set that is smaller than the query element. Given such a structure, it is easy to sort a collection of numbers – insert them all and then walk through the structure repeatedly outputting the next larger element.

Amazingly, if your numbers are in the range $(0,M]$, all operations on a vEB tree can be done in $O(\log\log M)$ operations. It follows that sorting n numbers in this range can be done with $O(n\log\log M)$ operations. If M is for example polynomial in n, this means we have again broken the $\Omega(n\log n)$ lower bound barrier for comparison-based algorithms. (We won't focus on space, but part of the cost of Van Emde Boas tree require a lot of space – typically $O(M)$.)

For sorting, all we need is "Insert" and "FindNext" operations, to insert an element and find the next larger element. So we explain those operations. For simplicity, let $M = 2^{2^z}$ for some z. A vEB tree T for numbers in the range $[0,M)$ has a root node that stores an array $T.c$ (where here c is short for children) of length $\sqrt{M}$. The entry $T.c[i]$ is itself a pointer (recursively) to a vEB tree that is responsible for the values $[i\sqrt{M},\ldots(i+1)\sqrt{M})$. Additionally, T stores two values $T.\min$ and $T.\max$ as well as an auxiliary vEB tree $T.a$.

Before doing an analysis, let's highlight what powers vEB trees have that will lead to its efficiency. First, given a number x to search on (or insert), we can jump to the child it should be in by computing its child number $i = \lfloor x/\sqrt{M} \rfloor$. As long as we can divide and compute square roots, we're fine.

You might notice that, because of the way we've chosen M, the value of i should correspond to the high order bits of x. This means we don't actually have to do any complicated division, just bit shifting. Also, note that the child doesn't have to store the whole value of x; just the remainder, which corresponds to the low order bits. That is, the child nodes are (recursively) storing numbers in the range $[0,\sqrt{M})$.

What about for FindNext? You might notice that there could be a problem if there are a lot of empty children in our structure. If that's the case, when we do a FindNext, if we look sequentially through the structure, we might have to check a lot of empty children before getting to a child structure that actually contains the next number. However, we can avoid this – that's the purpose of the "auxiliary" vEB tree. The auxiliary tree $T.a$ will contain the indices for all the children of the root T that actually contain at least one set element. Again, notice that these indices are just numbers in a range from $[0,\sqrt{M})$. This means that using the auxiliary tree, we can (recursively) find the next child that contains an item easily (in $O(\log\log M)$ operations).

More formally, to insert a number x we do the following steps. (If you want pseudocode, you can check the Wikipedia page.)

- There are edge cases to handle if the element being inserted is the minimum or the maximum; these are left to the reader. Otherwise...
- Compute $i = \lfloor x/\sqrt{M} \rfloor$.
- Recursively insert $x \bmod \sqrt{M}$ into $T.c[i]$.
- If $T.c[i]$ was previously empty, then insert i into $T.a$.

It looks like we may recursively have to do 2 insertions into structures for numbers of size $\sqrt{M}$. This would give us a recursion for the number of operations S given by $S(M) = 2S(\sqrt{M}) + O(1)$. You might check that this recursion would not gives us the desired $O(\log\log M)$ bound on the computation. We have to be a little more careful. Notice that the second insertion into $T.a$ only is needed if $T.c[i]$ was empty. If $T.c[i]$ was empty, then only a constant amount of work is needed to set up the child with a single element. So in fact there's really only one recursive call that does any work in either case. Hence we have $S(M) = S(\sqrt{M}) + O(1)$. You can check that this recursion gives $S(M) = O(\log\log M)$. (This can be a little clearer if you let $M = 2^m$ and look at the recursion as $S(m) = S(m/2) + O(1)$.)

To do a FindNext for a number x, we do the following steps.

- Check if x is less than or equal to the minimum or greater than the maximum; in these cases to the right thing. Otherwise...
- Compute $i = \lfloor x/\sqrt{M} \rfloor$ and $j = x \bmod \sqrt{M}$. (Note $x - j = i\sqrt{M}$.)
- If $T.c[i]$ is not empty and $T.c[i].\max \geq j$ then return $(x-j) + \mathrm{F}indNext(T.c[i], j)$.
- Otherwise, find the first non-empty child after x and return its minimum value by computing $y = \mathrm{F}indNext(T.a[i])$ and returning $y\sqrt{M} + T.c[y].\min$.

Again, we find the number of operations S given by $S(M) = S(\sqrt{M}) + O(1)$, and we conclude FindNext is $O(\log\log M)$ work.

1.8 Conclusion

Sorting remains an active area for research; the best sorting algorithm might still not be known, in part because the right model for machine operations is open to interpretation. One often hears of an $\Omega(n\log n)$ lower bound for sorting. That lower bound is correct in the most general setting for data where all one can do with two elements is compare them, but taking advantage of even simple aspects of the data, such as that they are numbers within some fixed range, can lead to sorting algorithms that break the lower bound.