Quick Sort Notes 18.310, Spring 2010

0.1 Randomized Median Finding

In a previous lecture, we discussed the problem of finding the median of a list of m elements, or more generally the element of rank m. The approach we followed was to select a *pivot* element p, to compute its rank, and partition the list into 2 sublists, one containing the elements smaller than p, the other containing those larger than p. By comparing m, the desired rank, to the size of these lists, we can restrict our search to one of the lists, and proceed recursively. If we select an arbitrary element for p, we might be unlucky and be able to discard very few elements; in the worst-case, this may lead to cn^2 comparisons for some c. By carefully selecting a good candidate as pivot element, we were able to devise a rank m selection algorithm that makes at most cn comparisons in the worst-case. But this was a pretty complicated algorithm and analysis, and instead, we'll propose now a simple probabilistic approach. Just select as pivot p an element selected uniformly at random among all elements in the list. This is very simple to implement. In the worst-case, we may be unlucky at every step and perform roughly cn^2 comparisons, but that is an extremely rare situation (an event with extremely tiny probability). In fact, on average, i.e. in *expection*, this simple *randomized* algorithm performs at most cn comparisons for some appropriate c. We'll show this now.

Let f_n be the number of comparisons this algorithm performs on an input of n elements. This is a random variable as this depends on our random choice of pivot at every level of the recursion. We want to compute, or at least upper bound, $E(f_n)$. We claim that $E(f_n) \leq 4n$.

Describing the sample space precisely and assigning/computing the probability of each sample point does not seem to be trivial. Rather, to get a bound on the expected value of f_n, we will try to isolate the first level of the recursion from subsequent levels by exploiting *independence* and making use of indicator random variables.

Before we proceed, let us observe that when we partition our list into two sublists, which one we keep depends on the rank m we need to select. To make the analysis valid for any value of m, let's assume we always keep the larger of the two sublists. This is as if an adversary would tell us which sublist to keep and which to discard. If the sublist with elements smaller than p has size ℓ and the other sublist has size $n-1-\ell$ (the pivot element p is in neither list), the list we keep has size $\max(\ell, n-1-\ell)$. Observe that ℓ is a uniform random variable taking values $0, 1, \cdots, n-1$, thus

$$P(\ell = i) = \frac{1}{n},$$

for $i = 0, 1, \cdots, n-1$. Let k be a random variable denoting the length of the list we keep after the first pivot. Since both $\ell = 0$ and $\ell = n-1$ would result in keeping a sublist of length $k = n-1$, the probability that the list we keep has length $n-1$ is $P(k = n-1) = \frac{2}{n}$. More generally, we have that

$$P(k = i) = \frac{2}{n}$$

for $i = \frac{n}{2}, \frac{n}{2}+1, \cdots, n-2, n-1$. This is not quite correct if n is odd (e.g. if $n = 5$, we have that $P(\ell = 2) = 1/5, P(\ell = 3) = 2/5, P(\ell = 4) = 2/5$), but this can be fixed easily. For simplicity, we'll do the calculations as if we only deal with even sized lists.

To compute $E(f_n)$, let us introduce the event A_k that the sublist we keep after the first level has size k. Observe that for different values of k, these events are disjoint, that is $A_{k_1} \wedge A_{k_2} = \emptyset$ for $k_1 \neq k_2$, while they partition S, that is $A_{n/2} \vee A_{n/2+1} \vee \cdots \vee A_{n-1} = S$. Thus,

$$f_n = f_n \left(I_{A_{n/2}} + I_{A_{n/2+1}} + \cdots + I_{A_{n-1}} \right).$$

What multiplies f_n in the expression above is always equal to 1, no matter what happens. Therefore,

$$E(f_n) = E(f_n \left(I_{A_{n/2}} + I_{A_{n/2+1}} + \cdots + I_{A_{n-1}} \right),$$

and by linearity of expectations, this is equal to:

$$E(f_n) = E(f_n I_{A_{n/2}}) + E(f_n I_{A_{n/2+1}}) + \cdots + E(f_n I_{A_{n-1}}).$$

Let us consider any such term, say $E(f_n I_{A_k})$ for some fixed value k. Here k is not a random variable but just a given value, say imagine that $k = 857$. This is the expectation of a product, and recall that this is not necessarily the product of the expectations, unless the two random variables are independent. In order to get independence, we write f_n as the sum of the number of comparisons in the first level (which is equal to $n-1$ since we have to compare the pivot to every other element) and the number of comparisons in subsequent levels. But observe that $f_n I_{A_k}$ is the same random variable as $(n-1+f_k) I_{A_k}$ where f_k denotes the number of comparisons we perform on a list of k elements, since both random variables take value 0 on any sample point corresponding to a different value of k. But I_{A_k} depends only on the the choice of our first pivot, while f_k depends only on our choices of subsequent pivots, and thus we have independence and can write:

$$\begin{aligned} E(f_n I_{A_k}) &= E((n-1+f_k) I_{A_k}) = E((n-1) I_{A_K}) + E(f_k I_{A_k}) \\ &= (n-1) E(I_{A_k}) + E(f_k) E(I_{A_k}) \\ &= ((n-1) + E(f_k))\, E(I_{A_k}). \end{aligned}$$

Using $E(I_{A_k}) = P(A_k)$ and summing over k, we get:

$$\begin{aligned} E(f_n) &= \sum_{k=n/2}^{n-1} (n-1+E(f_k))\, P(A_k) \\ &= \left((n-1) \sum_{k=n/2}^{n-1} P(A_k) \right) + \sum_{k=n/2}^{n-1} E(f_k) P(A_k) \\ &= (n-1) + \frac{2}{n} \sum_{k=n/2}^{n-1} E(f_k). \end{aligned}$$

We have thus obtained a recurrence relation for $E(f_n)$ and we simply need to solve it. Let's show by induction that $E(f_n) \leq cn$ for some value of n. We check the base case and also assume as inductive hypothesis that $E(f_m) \leq cm$ for all $m < n$. Thus we get that

$$\begin{aligned} E(f_n) &= (n-1) + \frac{2}{n} \sum_{k=n/2}^{n-1} E(f_k) \\ &\leq (n-1) + \frac{2}{n} c \sum_{k=n/2}^{n-1} k \\ &< (n-1) + c\frac{3}{4} n \\ &< (1 + \frac{3c}{4}) n \\ &\leq cn, \end{aligned}$$

provided that we choose $c \geq 4$. This proves by induction that $E(f_n) \leq 4n$. This randomized rank m selection algorithm therefore makes an expected number of comparisons linear in n (even though in the worst-case it is quadratic).

0.2 quicksort

We now know enough probability theory that we can analyze the expected number of comparisons taken by one of the most widely used sorting algorithms, quicksort. We mentioned quicksort briefly in Lecture 2, but we will describe it in detail now. Quicksort is an algorithm that can be implemented in place; that is, we do not need more storage than is required to store the number of keys we are sorting.

How does quicksort work? The first step is to choose a random key. We call this key the *pivot*. We then divide all the other keys into ones larger and smaller than

the pivot. Now, we put the keys less than the pivot before it, and the keys greater than the pivot after it. This gives us two lists which we still need to sort: the list of keys smaller than the pivot and those larger than the pivot. Let's give an example. Assume that we have the following array to sort.

`6 9 3 7 1 2 8 4 5 10`

Suppose we choose 7 as our random key. Putting those keys smaller than 7 before it and those larger than 7 after it, we get the following array.

`6 3 1 2 4 5` **`7`** `9 8 10`

Now, we have two new lists to sort: the first consisting of six numbers (6 3 1 2 4 5), and the second of three. We sort these lists recursively, applying quicksort to each list. Although in this example, we have kept the relative order within the two lists, there is in general no reason an implementation of quicksort needs to do this (and it does make it harder to program). This first step takes $n-1$ comparisons, as we have to compare the pivot to every other key.

How many comparisons does quicksort take? First, let's look at the two extremes, that is, the worst-case and the bast-case performance. Take k to be the number of keys that are less than the pivot. Then $n-k-1$ are the number of keys that are larger than the pivot, and we use recurision on lists of size k and size $n-k-1$.

If we let $f(j)$ be the number of comparison quicksort takes to sort j items, we thus get the recursion

$$f(n) = n - 1 + f(k) + f(n-k-1)$$

The best-case running time for quicksort occurs when the pivot is always in the exact middle of the list. If we were very lucky, and the pivot always divides the list into two nearly equally sized lists at each step, we get the recursion equation

$$f(n) \leq 2f(n/2) + n - 1.$$

(Actually, this is slightly different depending on whether n is even or odd, but this only make a very small difference to the analysis.) This recursion is very similar to an equaqtion we'e seen before, and solves to $f(n) \approx n \log_2 n$.

If, on the other hand, we are really unlucky in the choice of the pivot, and always chose a pivot that was either the smallest or largest key in the list, then we get the equation

$$f(n) = n - 1 + f(n-1)$$

which gives

$$f(n) = (n-1) + (n-2) + (n-3) + \ldots + 1,$$

so $f(n) = n(n-1)/2 \approx n^2/2$. This case happens if you always choose the first key in your list to be sorted, and you start with a list that is already sorted. Since in

practice, many lists which need to be sorted are already nearly sorted, choosing the first key as the pivot is not a good idea.

If we have some deterministic algorithm for picking the pivot, then we can arrange the input so the pivot is always the first element of the sublist, and quicksort will take around $n^2/2$ comparisons on this input. How can we do better? What we can do is always choose a random key in the list. If we do this, we can show that the expected number of comparisons taken by quicksort is $cn \log n$.

If we let f be the random variable which gives the amount of time taken by the algorithm on an input of size n, then we have by linearity of expectation,

$$\mathrm{E}f(n) = n - 1 + \mathrm{E}f(k) + \mathrm{E}f(n - k - 1)$$

where k is a random variable which is uniformly distributed between 0 and $n - 1$. One way to analyze quicksort is to solve this equation. However, there's another very clever way which illustrates the use of indicator variables and linearity of expectations, and we will explain this now.

We will compute the probability that the rank j and rank k keys are compared in quicksort. To get some intuition into this, let's look at the extreme cases first. If $j = i + 1$, then quicksort (and in fact, any sorting algorithm) must compare these keys, as otherwise there would be no way to tell which was larger. In the quicksort algorithm, what happens is that they remain in the same sublist until one of them is chosen as a pivot, at which point they are compared. If $i = 1$ and $j = n$, then the first key chosen as a pivot will separate them into two sublists, except in the case when one of them is chosen as the first pivot; in this case they will be compared on the first step. Thus, the probability that these keys are compared is $2/n$.

Let us consider the probability the the i'th rank key and the j'th rank key get compared for arbitrary i and j. There are $j - i + 1$ keys strictly between these two keys. What happens when we run quicksort? As long as these two keys remain in the same sublist, the possibility exists that these keys will be compared later in the algorithm. When we process this sublist, if we do not pick either of these two keys, or any of the $j - i + 1$ keys between them as a pivot, they will remain in the same sublist. If we pick one of these two keys as a pivot, then they will be compared. If we pick one of the keys between them, then the i'th and j'th rank keys will be placed in different sublists, and so will never be compared. Thus, there is exactly one critical step that determines whehter the i'th and j'th rank keys will be compared. This is the first pivot that picks either one of these keys of one of the keys between them. Given that this pivot step picks one of these $j - i + 1$ keys, the conditional probability that it picks any particular one of these keys is $1/(j - i + 1)$. Thus, the probability that key i and key j are compared is the probability that one of these is picked on this step, which is $2/(j = i + 1)$.

Now, let $I_{i,j}$ be the indicator variable which is 1 if keys i and j are compared by our algorithm and 0 otherwise. The number of comparisons needed by quicksort is

$$C = \sum_{i<j} I_{i,j}$$

so by linearity of expectation, we can take the expectation on both sides and get

$$\mathrm{E}C = \sum_{1\le i<j\le n} \mathrm{E}I_{i,j}.$$

What is $\mathrm{E}I_{i,j}$? It is 1 if we compare them, and 0 if we don't, so the expectation of $I_(i,j)$ is exactly the probability that we compare the rank i and rank j keys, or $1/(j-i+1)$. Thus, we have

$$\mathrm{E}C = \sum_{1\le i<j\le n} \frac{2}{j-i+1}.$$

We can count that there are $n-k$ pairs of keys whose difference is exactly k, so

$$\mathrm{E}C = \sum_{k=1}^{n-1}(n-k)\frac{2}{k+1} \le 2n\sum_{h=2}^{n}\frac{1}{h}$$

The harmonic series $\sum_{h=1}^{n}\frac{1}{h}$ is approximately $\ln n$. Thus, we have an upper bound of $2n\ln n$ for $\mathrm{E}C$. Looking at the sum above more carefully, it is not hard to check that to first order this bound is correct, and $\mathrm{E}C \approx 2n\ln n$.

There is one last thing to do: we claimed that quicksort was an algorithm that sorts "in place," that is, without using any extra workspace, and we haven't showed how to do this. The only hard part of doing this is the first step: namely, rearranging the list to put the keys smaller than the pivot first and the keys larger than the pivot last? There are several ways of doing this, and the one I presented in class is different from the one in these notes. Let's first put the pivot at the end where it will be out of the way. Now let us first assume that we know where the pivot goes. We'll put an imaginary dividing line there. Doing this with our example,

`6 9 3 7 1 2 8 4 5 10`

with 7 as the pivot, gives

`6 9 3 10 1 2| 8 4 5|`**7**

Now, the correct number of keys are on each side of this dividing line. This means that the number of out-of-place keys on the right is the same as the number of out-of-place keys on the left. [This isn't hard to prove, but we'll leave it as an exercise for you.]

Now, we can go through the list on the right of the dividing line and the list on the left, one key at a time, and when we find two out-of-place keys we can swap them. Once we've swapped all the out-of-place keys, the correct number will be on each side of the line, since there are an equal number of out-of-place keys on either side. Let's look for the misplaced keys by working from the outside in, and comparing each key to the pivot. The first misplaced keys we find are 9 (on the left) and 5 (on the right). Swapping them gives

`6 5 3 10 1 2| 8 4 9|`**`7`**

The next two are 10 and 4. Swapping them gives

`6 5 3 4 1 2|8 10 9|`**`7`** ,

and now all the keys are on the correct size of the line. Now, we can swap the pivot and the first key to the right of the dividing line, to get

`6 5 3 4 1 2|`**`7`**`|10 9 8` .

This places the keys smaller and larger than the pivot on the correct side of the pivot, and we're ready for the next step of quicksort.

But how do we know where to put the dividing line? In fact, we don't need to know where the dividing line is to run the algorithm. Suppose we don't know where the dividing line is. We can still work our way in from the two ends of the array, and whenever we find two out-of-place keys, we swap them. The place where we meet in the middle is the dividing line.