CS 624
Lecture 4: Quicksort

1 The quicksort algorithm

This is a divide-and-conquer algorithm—one of the first, and probably the most famous. Certainly it's one of the most useful.

The interface is this: QUICKSORT takes an array A, and indexes $p < q$ in the array, and sorts the elements in $A[p\,..\,q]$ into ascending order *in place*[1].

Recall that MERGE-SORT started out by dividing the array into two equal parts. This was done completely arbitrarily—we just took the first half and the second half. QUICKSORT, on the other hand, is rather more clever. It divides the array into two parts using a special procedure called PARTITION. We'll show below how PARTITION works, but the key point is that PARTITION takes as input an array A, and indexes $p < r$ in A, and returns an index q between p and r—that is, $p \le q \le r$—such that all the elements to the left of q are $\le A[q]$, and all the elements to the right of q are $\ge A[q]$.

The pseudo-code for QUICKSORT then looks like this:

```
QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

The initial call to quicksort an entire array $A[1\,..\,n]$ is QUICKSORT$(A, 1, n)$.

The key thing about the PARTITION procedure (and we have mentioned some of this already above) is that just after the line $q \leftarrow$ PARTITION(A, p, r) has executed, the following are true:

1. $p \le q \le r$.
2. The number $A[q]$ is in its final position. It will never be moved again.
3. If $i < q$, then $A[i] < A[q]$, and if $i > q$, then $A[i] > A[q]$.

The partition procedure is not all that easy to get right. (Almost everyone makes a mistake the first time they write code for it. That certainly happened to me.) There are actually many different

[1] This is what makes QUICKSORT better than MERGE-SORT. MERGE-SORT needs to allocate extra memory to hold the partially sorted arrays which are then merged back into place in the original array.

versions of the partition procedure. Conceptually, they are all simple. The difficulty comes because we are trying to do all this in place, without allocating temporary storage. This code is the one the book uses:

```
PARTITION(A, p, r)
    x ← A[r] // x is the value at the "pivot" element.
    i ← p − 1 // i maintains the "left-right boundary".
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            exchange A[i] ↔ A[j]
    exchange A[i + 1] ↔ A[r]
    return i + 1
```

This code is fairly tricky. Figure 1 shows how it works in one particular case. But clearly we really need a formal proof of correctness here.

2 Proof of correctness

The PARTITION function works by dividing the array A into four sections, as you can see in Figure 1. The inductive hypothesis (which we will show to be a loop invariant) can be expressed in terms of these four sections.

2.1 Theorem *At the end of the* PARTITION *procedure, the elements in the array have been rearranged so that each element less than the pivot is to the left of it, and each element greater than the pivot is to the right of it.*

Remark *Note that we are not saying that the elements are in sorted order. This is a much weaker condition. That's why* PARTITION *can be implemented efficiently. And even though the condition is weaker than complete sorting, it turns out to be just what we need for the complete* QUICKSORT *algorithm.*

We will prove this theorem by induction. We will encapsulate the inductive step in a lemma:

2.2 Lemma *At the beginning of each iteration of the* ***for*** *loop,*

1. *All entries in* $A[p\,..\,i]$ *are* $\leq$ *pivot.*
2. *All entries in* $A[i+1\,..\,j-1]$ *are* $>$ *pivot.*
3. $A[j\,..\,r-1]$ *consists of elements whose contents have not yet been examined, and so we don't know how they compare to the pivot.*
4. $A[r] =$ *pivot.*

Remarks

Actually, item 3 is not actually needed for the proof of correctness of the algorithm, but we include it here anyway because it makes things a bit clearer. Figure 2 annotates one of the iterations of the ***for*** *loop.*

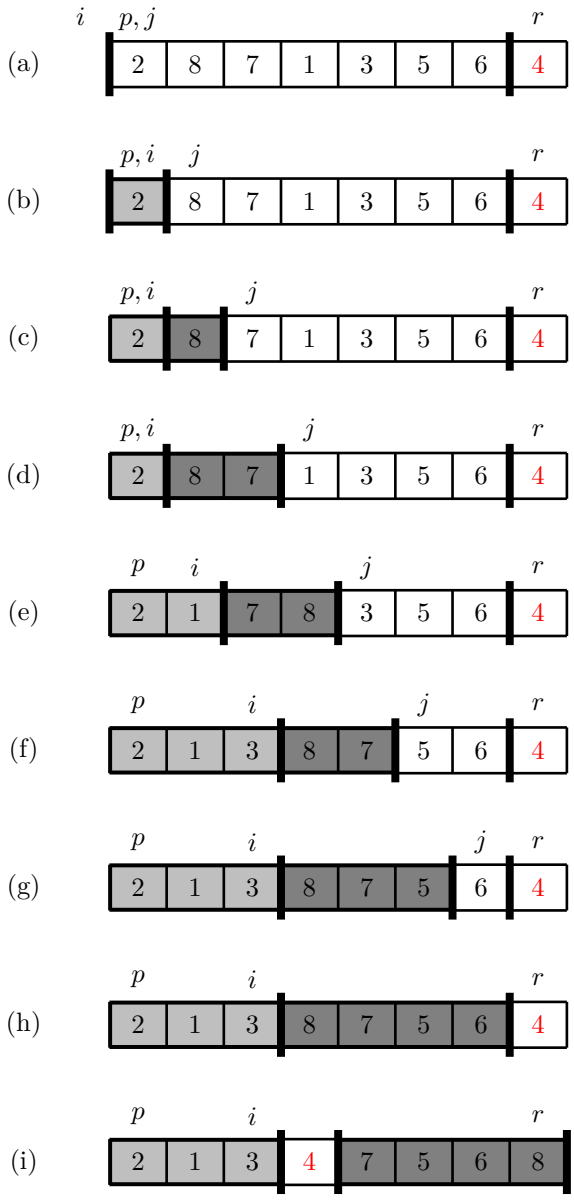

Figure 1: The workings of PARTITION. Each line shows the situation at the beginning of successive iterations of the **for** loop.

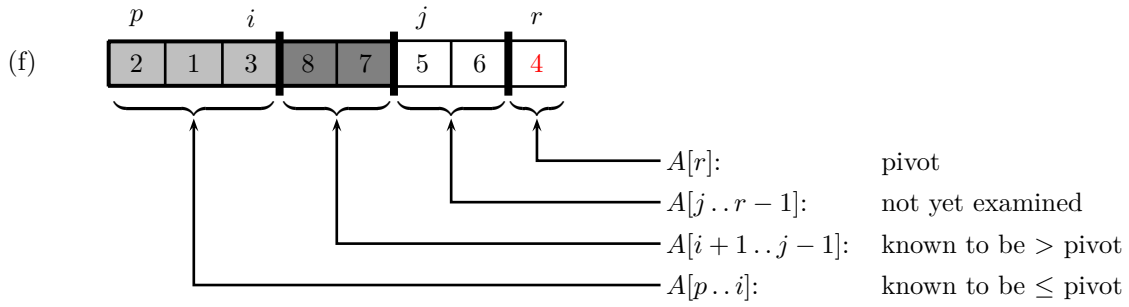

Figure 2: The situation at the start of one iteration of the **for** loop of the PARTITION function.

What we are really doing here is stating an inductive hypothesis—it is just the four statements of the lemma. The lemma shows how the inductive step works (that is, it shows how we get from one iteration of the loop to the next, preserving the truth of the inductive hypothesis.

And as usual, this inductive hypothesis is really a sequence of assertions (each assertion composed of the four statements of the lemma). There is one assertion for each value of j, and j runs from p up to $r-1$. We have to show that all the assertions are true.

PROOF. When we start out, $j = p$, i is $p-1$, and the four statements in the lemma are trivially true. Thus, the assertion corresponding to $j = p$ is true.

Let us say that when we are at the top of iteration j_0 of the **for** loop, i has the value i_0. Then the inductive hypothesis says that at the top of that iteration of the **for** loop,

1. All entries in $A[p\,..\,i_0]$ are $\le$ pivot.
2. All entries in $A[i_0+1\,..\,j_0-1]$ are $>$ pivot.
3. $A[j_0\,..\,r-1]$ consists of elements whose contents have not yet been examined, and so we don't know how they compare to the pivot.
4. $A[r]$ = pivot

Now there are two cases to consider when we enter this iteration of the **for** loop:

- If $A[j_0] \le$ the pivot value, then the following two things happen:
  - $A[j_0]$ and $A[i_0+1]$ are interchanged.
  - i_0 is incremented. Let us call its new value i_1. So $i_1 = i_0 + 1$, and i_1 will be the value of i at the top of the next iteration of the **for** loop.

  and this brings us to the next iteration of the **for** loop, in which j has the value $j_1 = j_0 + 1$. Thus, at the top of the next iteration of the **for** loop, since the original $A[j_0]$ was $\le$ the pivot and since we interchanged $A[j_0]$ and $A[i_0+1]$, we see that now

1. All entries in $A[p\,..\,i_1]$ are $\leq$ pivot.
2. All entries in $A[i_1+1\,..\,j_1-1]$ are $>$ pivot. (These are the same elements that were originally in $A[i_0+1\,..\,j_0-1]$. The first one has been moved up to the end.)
3. $A[j_1\,..\,r-1]$ consists of elements whose contents have not yet been examined, and so we don't know how they compare to the pivot.
4. $A[r]$ = pivot.

And this is just the statement of the inductive hypothesis at the top of the j_0+1 $(=j_1)$ iteration of the **for** loop.

- Alternatively, we must have $A[j_0] >$ the pivot value. In this case, nothing is done. At the top of the next iteration of the **for** loop, we have
  - $i_1 = i_0$ (because we didn't increment i).
  - $j_1 = j_0 + 1$ (because we always increment j when we go to the next iteration).
  - And no change was made to the elements of the array A.

  Thus, we have
  1. All entries in $A[p\,..\,i_1]$ continue to be $\leq$ pivot.
  2. All entries in $A[i_1+1\,..\,j_1-1]$ are $>$ the pivot (these are just the original elements $A[i_0+1\,..\,j_0-1]$ together with $A[j_1-1] = A[j_0]$, which is $>$ the pivot by assumption in this case.)
  3. $A[j_1\,..\,r-1]$ consists of the elements whose contents have not yet been examined, and so we don't know how they compare to the pivot.
  4. $A[r]$ = pivot

  And this is just the statement of the inductive hypothesis at the top of the j_0+1 $(=j_1)$ iteration of the **for** loop.

So by induction the four statements are true at the start of each loop iteration. □

Now we can complete the proof of the theorem:

Proof of theorem. At the conclusion of the **for** loop, element r (which is the pivot element) is exchanged with element $i+1$ (which is the left-most element that is greater than the pivot element). This ensures that all the elements to the left of the pivot element have values $\leq$ the pivot, and all the elements to the right of the pivot element have values $>$ the pivot.

And that concludes the proof of correctness. □

3 First (non-rigorous) steps towards understanding the running time

In this section we will look intuitively at the running time. What we do will not be rigorous, but it will be helpful in understanding the actual proofs we will come up with later.

First, the time to execute the Partition function on an array of n elements is clearly $\Theta(n)$. Based on this, we can reason roughly about some special cases:

3.1 The best case

We expect the best case running time occurs when the partition element is the median element at each step—that is, it divides the array being partitioned into two equal parts. (Of course, they won't be exactly equal, but as we'll see below, the algorithm is actually fairly insensitive to this.)

In fact, this is true. But it's important to realize that we haven't really proved it. We will later. But for the moment, let's go on:

In this case, we get the recurrence

$$T(n) = 2T(n/2) + \Theta(n)$$

and so by the "master theorem", we have $T(n) = \Theta(n \log n)$.

3.2 The worst case

The worst case (well, it's certainly a very bad case, anyway; we'll prove it really is the worst case later) occurs when the array is already sorted. In this case, the PARTITION function applied to an array of size n produces two sub-arrays, of size $n-1$ and 0. Thus, we have

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

and it is easy then to see that

$$T(n) = \sum_{j=0}^{n} \Theta(j) = \Theta\Big(\frac{n(n+1)}{2}\Big) = \Theta(n^2)$$

So the worst case of QUICKSORT is no better than the naive and very inefficient sorts such as insertion sort.

3.3 The average case

It turns out, however, that in general, QUICKSORT runs in $O(n \log n)$ time. This is a remarkable fact. Often in algorithms, the average case is no better than the worst case. But for QUICKSORT, the best case really is pretty much the average case. This will take some serious proving, and we'll do that later. But first, let's try to see why this remarkable fact might be true.

As we've already noted, it is intuitively clear that QUICKSORT will be most efficient when the pivot element divides the array into equal parts. But suppose it doesn't? Suppose, for instance, we are unlucky enough that the pivot element always divides the array into two parts such that the first part is 1/10 the size of the original array and the second part is 9/10 the size of the original array?

> Of course this can't be exactly true, but we're just doing this by way of illustration, and actually the analysis would hold just as well if we did things exactly. And we'll perform an exact calculation later, anyway.

We get a recursion tree that looks like the tree in Figure 3.

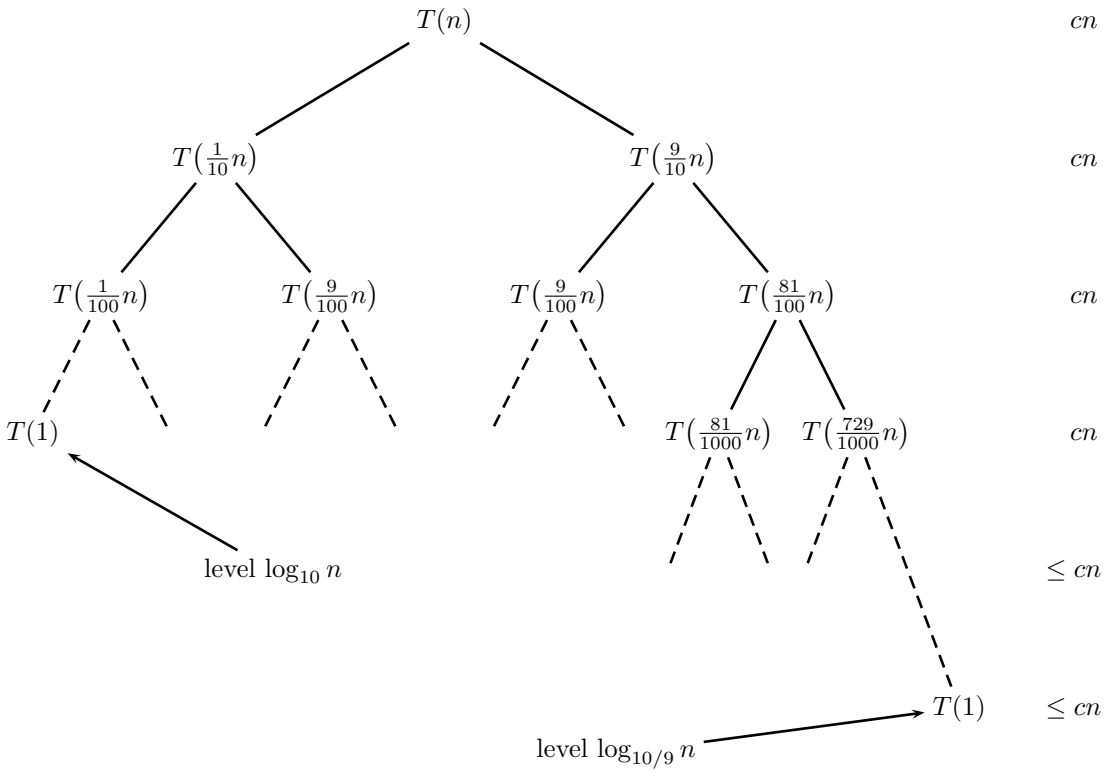

Figure 3: Recursion tree for QUICKSORT in the case where each pivot divides its sub-array in the ratio 1:9. The tree is shortest on the left—the leftmost leaf is at level $\log_{10} n$, and the rightmost leaf is at level $\log_{10/9} n$.

3.1 Exercise *In Figure 3, where do the costs in the right-hand column come from? Show that they make sense. In particular, show why some of the costs are exact, and some are upper bounds (with $\leq$ signs).*

Since there are $1 + \log_{10/9} n$ levels, and each level has cost $O(n)$, we see that the total cost is $O(n \log n)$. So even though our pivot choice was not great at all, we really didn't lose much. In other words, it appears that QUICKSORT is relatively insensitive to the choice of the pivot node.

Well, that's nice, but of course we have assumed here that the pivot choice, while certainly not optimal, is still "bounded away from the edges". What happens when it occasionally is as bad as it can be? For instance, suppose it happens that on every other iteration the pivot is the largest element of its sub-array (this is the worst possible case), but on the rest, the pivot divides the sub-array equally. What happens then?

Well, each step looks about like that in Figure 4.

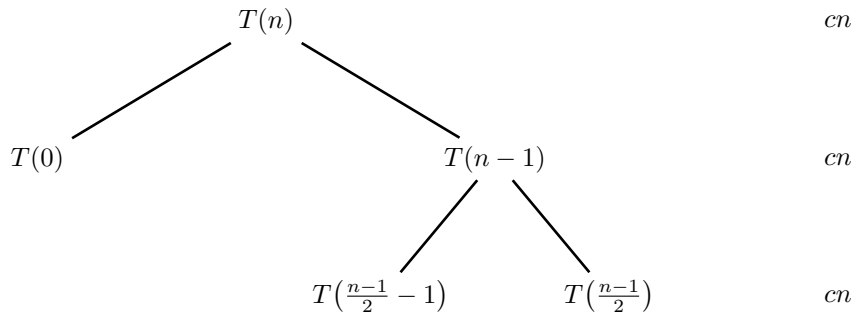

Figure 4: Fragment of the recursion tree where on every other iteration the pivot is the largest element of its subarray.

We can see immediately that the only real difference between this tree and the best tree (in which every level gets divided equally) is that there are now twice as many levels. So the total cost is still $O(n \log n)$.

Now in a typical case, we would expect the pivot to be sometimes "in the middle", and sometimes "toward the edge". The two cases we have just considered give us some hope that unless we are incredibly unlucky and the pivot is overwhelmingly at the edge—i.e., really almost all of the time—that we will still get $O(n \log n)$ behavior.

This turns out to be true, and we'll prove this below.

4 Randomized Quicksort

In the Partition algorithm, we have always let the pivot element be the last element of the array. Of course we could have picked it in any way. One way that is widely used is to pick it "at random"—that is, to introduce a procedure Random(A, p, r) which returns one of the values $\{A[p], A[p+1], \ldots, A[r]\}$, each value being equally probable. (Exactly what this means is that if we call this function a number of times, it will return different values in general, and the fraction of times it returns any particular value tends to $\frac{1}{r-p+1}$ as the number of calls grows.) The construction of such functions is a topic in itself, but we'll assume that we have such a function. Then we can define our variant of Quicksort as follows:

First we define a Randomized-Partition function which starts by picking a pivot at random and moving it into the last position in the array.

```
Randomized-Partition(A, p, r)
    i ← Random(p, r)
    exchange A[r] ↔ A[i]
    return Partition(A, p, r)
```

Then we can define RANDOMIZED-QUICKSORT just as before, but this time using RANDOMIZED-PARTITION.

RANDOMIZED-QUICKSORT(A, p, r)
  if $p < r$ **then**
    $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT$(A, p, q-1)$
    RANDOMIZED-QUICKSORT$(Q, q+1, r)$

5 Rigorous analysis of QUICKSORT

5.1 Worst-case analysis

Let $T(n)$ be the worst-case running time for QUICKSORT (or for RANDOMIZED-QUICKSORT; the reasoning is just the same in this case). We know that there is some constant $a > 0$ such that the following recursive inequality is satisfied:

$$T(n) \leq \max_{0 \leq q \leq n-1} \big(T(q) + T(n-q-1)\big) + an$$

We have seen that we have reason to believe that $T(n) = O(n^2)$. So let us try to prove this. Certainly there is some constant $c > 0$ such that $T(k) \leq ck^2$ for $k = 1$. Suppose this inequality (with some fixed constant c) holds true for all $k < n$. Then we have

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} \big(T(q) + T(n-q-1)\big) + an \\ &\leq c \max_{0 \leq q \leq n-1} \big(q^2 + (n-q-1)^2\big) + an \end{aligned}$$

How big is the max? Since the expression $\big(q^2 + (n-q-1)^2\big)$ is a convex function, it attains its maximum at one of the two endpoints (0 or $n-1$). Therefore, it is at most $(n-1)^2 = n^2 - (2n-1)$

Thus, (provided only that $n \geq 1$—we'll need that at one point below) we have

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} \big(T(q) + T(n-q-1)\big) + an \\ &\leq c \max_{0 \leq q \leq n-1} \big(q^2 + (n-q-1)^2\big) + an \\ &\leq cn^2 - c(2n-1) + an \\ &= cn^2 - (2c-a)n + c \\ &\leq cn^2 - (2c-a)n + cn \qquad \text{(here's where we use the assumption that } n \geq 1) \\ &= cn^2 - (c-a)n \end{aligned}$$

and if we just pick c large enough so that $c \geq a$, this all ends up being $\leq cn^2$, and thus we have proved the inductive step. (It's important to notice that we can make this choice of c once and for all—it doesn't change with every n.)

Thus, in the worst case, $T(n) = O(n^2)$. Of course this analysis doesn't actually prove that there *is* a case with behavior that is this bad—it just shows that there isn't an case with behavior worse

than this. But in fact we have already seen that there is a case with behavior this bad—the case in which the array is already sorted. (That is, we have previously shown that the worst-case running time is $\Omega(n^2)$.) Therefore the worst-case running time is in fact $\Theta(n^2)$.

5.1 Exercise *The analysis for the best-case running time can be done in a somewhat similar fashion, but is technically a bit more complicated mathematically. You might be interested in trying it out, however.*

5.2 Average-case analysis: first method

It turns out that the easiest way of analyzing the average-case behavior of QUICKSORT is actually to analyze the average-case behavior of RANDOMIZED-QUICKSORT. So that is what we will do. (And actually, RANDOMIZED-QUICKSORT is what is often used in practice in any case.)

Let $T(n)$ be the average running time for RANDOMIZED-QUICKSORT on an array of size n. We have

$$T(n) = \frac{1}{n}\sum_{q=0}^{n-1}\big(T(q) + T(n-q-1)\big) + cn + \Theta(1)$$

since each of the n positions in the array can be chosen (with equal probability) as the pivot.

Also note (and this is one of the tricky points here) that we wrote $cn + \Theta(1)$ rather than $\Theta(n)$. Actually the cost of the PARTITION function on an array of size n is $\Theta(n)$. However, we don't lose much by assuming the worst case—that on every iteration of the loop in PARTITION we "do everything". In that case, the cost really is of the form $cn + b$ for some fixed constant c. By doing this, we have possibly given away something in our analysis, but as we will see, we haven't given away anything important. And by doing this, we can perform a really neat mathematical derivation.

Now that we have done this, we can simplify this equation somewhat. It is the same as

$$T(n) = \frac{2}{n}\sum_{q=0}^{n-1} T(q) + cn + \Theta(1)$$

Now this still looks very complex. But look what we can do: First, let us multiply through by n:

$$nT(n) = 2\sum_{q=0}^{n-1} T(q) + cn^2 + \Theta(n) \tag{1}$$

Replacing n by $n+1$, we get a similar equation:

$$(n+1)T(n+1) = 2\sum_{q=0}^{n} T(q) + c(n+1)^2 + \Theta(n) \tag{2}$$

Now if we subtract (1) from (2), a lot of terms cancel, and we get

$$(n+1)T(n+1) - nT(n) = 2T(n) + \Theta(n)$$

Now we can see the reason we really needed to give an upper estimate of the cost of the partition as $cn + \Theta(1)$, rather than $\Theta(n)$. If we had used $\Theta(n)$, then it would have

been $\Theta(n^2)$ in the two equations we were subtracting, and $\Theta(n^2) - \Theta(n^2)$ could still be—and often *will* be—$\Theta(n^2)$, which we really don't want. But the way we did it, we have $(n+1)^2 - n^2 = \Theta(n)$ which is exactly what we need.

Collecting the $T(n)$ terms, this is

$$(n+1)T(n+1) = (n+2)T(n) + \Theta(n)$$

We can get this in yet a simpler form by dividing both sides by $(n+1)(n+2)$. We get

$$\frac{T(n+1)}{n+2} = \frac{T(n)}{n+1} + \Theta\Big(\frac{1}{n}\Big)$$

Thus, if we define a function $g(n)$ by

$$g(n) = \frac{T(n)}{n+1}$$

then we have

$$g(n+1) = g(n) + \Theta\Big(\frac{1}{n}\Big)$$

and thus

$$g(n) = \Theta\Bigg(\sum_{k=1}^{n-1} \frac{1}{k}\Bigg) = \Theta(\log n)$$

and so finally

$$T(n) = (n+1)g(n) = \Theta(n \log n)$$

5.3 Average-case analysis: second method

If we look again at the Quicksort (or Randomized-Quicksort) algorithm, we can see that the cost is the sum of the costs of all the calls to Partition (or Randomized-Partition). And—because of the simplification that we made at the beginning of the last section—the cost of a call to Randomized-Partition is

$$O(\langle \text{the number of times its } \textbf{for} \text{ loop is executed} \rangle)$$

This in turn is just

$$O(\langle \text{the number of comparisons it makes} \rangle)$$

So the cost of Randomized-Quicksort is

$$O(\text{the average number of comparisons that are made in the course of executing the algorithm})$$

And so that's what we have to compute—the average number of comparisons that are made in the course of executing the algorithm.

Now the thing that makes QUICKSORT more efficient in general is that not all pairs of elements are compared. If they were, we would have an $O(n^2)$ algorithm. But in fact, if we let

$$\{x_1, x_2, \ldots, x_n\}$$

be the set of keys *in sorted order*, then we know that once a key (x_k, say) has been used as the pivot of a partition, the keys to the left of it will never be compared with the keys to the right of it.

Continuing this line of reasoning, consider the set of keys $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$. Since keys are only compared with the current pivot, the only way for any two of these keys to be compared with each other is for one of them to be chosen as the pivot. And as soon as any one of these keys is chosen as the pivot, x_i and x_j will no longer be in the same partition and can no longer be compared. The last moment at which all these keys of the set occur in the same partition is the moment at which one of them is chosen as a pivot. Any one of them is equally likely to be chosen.

So we see that the only way that x_i and x_j can ever be compared is if one of them was the pivot. And this can only happen if one of them is the first pivot chosen from the set $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$.

To put it slightly differently, think of the pivots as being chosen at random, one after another. Every element is eventually chosen as a pivot. Whether x_i and x_j are ever compared with each other will not be known until a pivot is chosen from the set $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$. At that point, we know for sure what happens: x_i and x_j are compared $\iff$ that pivot is either x_i or x_j. They could not have been compared before that time, and they can't be compared afterwards either. They can only be compared the first time a pivot is chosen in $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$, and then only if the pivot is x_i or x_j.

The probability of that happening is $\frac{2}{j-i+1}$.

So we have just shown that the probability of x_i being compared with x_j is $\frac{2}{j-i+1}$.

The expected (or "average") number of comparisons is then

$$\begin{aligned}
\sum_{\substack{\text{all pairs } (i,j) \\ \text{with } i<j}} 1 \cdot \text{probability that } x_i \text{ and } x_j \text{ are compared} &= \sum_{\substack{\text{all pairs } (i,j) \\ \text{with } i<j}} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \qquad (\text{letting } k = j-i) \\
&\le \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= 2(n-1)H_n \\
&= O(n \log n)
\end{aligned}$$

5.4 A little more explanation of the last proof

Some people are not convinced by this argument. They say, "Well, suppose that we are not dealing with the elements $\{x_i, x_{i+1}, \ldots, x_j\}$. Suppose we are dealing with a larger interval, say $\{x_a, x_{a+1}, \ldots, x_b\}$ with $a \leq i < j \leq b$ (and either $a < i$ or $j < b$ or both). Don't we still want to know if x_i or x_j is chosen as a pivot before all the other numbers in $\{x_i, x_{i+1}, \ldots, x_j\}$? And doesn't the fact that we are dealing with a larger interval change the computation?

Actually, it doesn't, although this is an example of just how hard it is to reason probabilistically without a certain amount of experience.

Here is one way to look at this. Let's make it simple: let's assume we are dealing with the interval $\{x_i, x_{i+1}, \ldots x_b\}$ where $i < j < b$. And let us look at what happens when the first pivot is chosen in that interval.

Further, let us define

$$P(i,j;b) \quad = \quad \begin{array}{l}\text{the probability that when elements are picked from the interval } \{x_i, \ldots, x_j, \ldots, x_b\}, \text{ the first element that falls in the sub-interval } \{x_i, \ldots, x_j\} \text{ is either } x_i \text{ or } x_j.\end{array}$$

We have already seen that $P(i,j;j) = \frac{2}{j-i+1}$.

Now let us see what happens when a number is picked from the interval $\{x_i, \ldots, x_j, \ldots, x_b\}$. There are two cases:

Case 1: The number is x_b. This happens with probability $\frac{1}{b-i+1}$. Of course this number is not in the interval $\{x_i, \ldots, , x_j\}$, but it still might be true that the first number chosen after this which is in the interval $\{x_i, \ldots, x_j\}$ is either x_i or x_j. Now the numbers remaining from the interval $\{x_i, \ldots, x_j, \ldots, x_b\}$ (that is the numbers from this interval that might be chosen subsequently) constitute the interval $\{x_i, \ldots, x_j, \ldots, x_{b-1}\}$. The probability that either x_i or x_j is the first number chosen after this which is in the interval $\{x_i, \ldots, x_j\}$ is thus $P(i,j;b-1)$.

Case 2: The number is not x_b. This happens with probability $1 - \frac{1}{b-i+1}$. We know that this number must be in the interval $\{x_i, \ldots, x_j, \ldots, x_{b-1}\}$, and the probabiliity that this number is either x_i or x_j is therefore just $P(i,j;b-1)$.

Thus, if $b > j$, we have

$$P(i,j;b) = \frac{1}{b-i+1}P(i,j;b-1) + \Big(1 - \frac{1}{b-i+1}\Big)P(i,j;b-1) = P(i,j;b-1)$$

and clearly we can keep on doing this computation[2] to get

$$P(i,j;b) = P(i,j;b-1) = P(i,j;b-2) = \ldots = P(i,j;j) = \frac{2}{j-1+1}$$

and we are done.

[2]This is really an example of mathematical induction, right?