6.6 Conditional Expectations, Recurrences and Algorithms

Probability is a very important tool in algorithm design. We have already seen two important examples in which it is used—primality testing and hashing. In this section we will study several more examples of probabilistic analysis in algorithms. We will focus on computing the running time of various algorithms. When the running time of an algorithm is different for different inputs of the same size, we can think of the running time of the algorithm as a random variable on the sample space of inputs and analyze the expected running time of the algorithm. This us a different understanding from studying just the worst case running time for an input of a given size. We will then consider *randomized algorithms*, algorithms that depend on choosing something randomly, and see how we can use recurrences to give bounds on their expected running times as well.

For randomized algorithms, it will be useful to have access to a function which generates random numbers. We will assume that we have a function `randint(i,j)`, which generates a random integer uniformly between i and j (inclusive) [this means it is equally likely to be any number between i and j] and `rand01()`, which generates a random real number, uniformly between 0 and 1 [this means that given any two pairs of real numbers (r_1, r_2) and (s_1, s_2) with $r_2 - r_1 = s_2 - s_1$ and r_1, r_2, s_1 and s_2 all between 0 and 1, our random number is just as likely to be between r_1 and r_2 as it is to be between s_1 and s_2]. Functions such as `randint` and `rand01` are called *random number generators*. A great deal of number theory goes into the construction of good random number generators.

When Running Times Depend on more than Size of Inputs

Exercise 6.6-1 Let A be an array of length $n-1$ (whose elements are chosen from some ordered set), sorted into increasing order. Let b be another element of that ordered set that we want to insert into A to get a sorted array of length n. Assuming that the elements of A and b are chosen randomly, what is the expected number of elements of A that have to be shifted one place to the right to let us insert b?

Exercise 6.6-2 Let $A(1:n)$ denote the elements in positions 1 to n of the array A. A recursive description of insertion sort is that to sort $A(1:n)$, first we sort $A(1:n-1)$, and then we insert $A(n)$, by shifting the elements greater than $A(n)$ each one place to the right and then inserting the original value of $A(n)$ into the place we have opened up. If $n = 1$ we do nothing. Let $S_j(A(1:j))$ be the time needed to sort the portion of A from place 1 to place j, and let $I_j(A(1:j), b)$ be the time needed to insert the element b into a sorted list originally in the first j positions of A to give a sorted list in the first $j+1$ positions of A. Note that S_j and I_j depend on the actual array A, and not just on the value of j. Use S_j and I_j to describe the time needed to use insertion sort to sort $A(1:n)$ in terms of the time needed to sort $A(1:n-1)$. Don't forget that it is necessary to copy the element in position i of A into a variable b before we move elements of $A(1:i-1)$ to the right to make a place for it, because this moving process will write over $A(i)$. Let $T(n)$ be the expected value of S_n; that is, the expected running time of insertion sort on a list of n items. Write a recurrence for $T(n)$ in terms of $T(n-1)$ by taking expected values in the equation that corresponds to your previous description of the time needed to use insertion sort on a particular array. Solve your recurrence relation in big-Θ terms.

If X is the random variable with $X(A, b)$ equal to the number of items we need to move one place to the right in order to insert b into the resulting empty slot in A, then X takes on the values $0, 1, \ldots, n-1$ with equal probability $1/n$. Thus we have

$$E(x) = \sum_{i=0}^{n-1} i\frac{1}{n} = \frac{1}{n}\sum_{i=0}^{n-1} i = \frac{1}{n}\frac{(n-1)n}{2} = \frac{n-1}{2}.$$

Using $S_j(A(1:j))$ to stand for the time to sort the portion of the array A from places 1 to j by insertion sort, and $I_j(A(1:j), b)$ to stand for the time needed to insert b into a sorted list in the first j positions of the array A, moving all items larger than j to the right one place and putting b into the empty slot, we can write that for insertion sort

$$S_n(A(1:n)) = S_{n-1}(A(1:n-1)) + I_{n-1}(A(1:n-1), A(n)) + c_1.$$

We have included the constant term c_1 for the time it takes to copy the value of $A(n)$ into some variable b, because we will overwrite $A(n)$ in the process of moving items one place to the right. Using the additivity of expected values, we get

$$E(S_n) = E(S_{n-1}) + E(I_{n-1}) + E(c_1).$$

Using $T(n)$ for the expected time to sort $A(1:n)$ by insertion sort, and the result of the previous exercise, we get

$$T(n) = T(n-1) + c_2\frac{n-1}{2} + c_1.$$

where we include the constant c_2 because the time needed to do the insertion is going to be proportional to the number of items we have to move plus the time needed to copy the value of $A(n)$ into the appropriate slot (which we will assume we have included in c_1). We can say that $T(1) = 1$ (or some third constant) because with a list of size 1 we have to realize it has size 1, and then do nothing. It might be more realistic to write

$$T(n) \leq T(n-1) + cn$$

and

$$T(n) \geq T(n-1) + c'n,$$

because the time needed to do the insertion may not be exactly proportional to the number of items we need to move, but might depend on implementation details. By iterating the recurrence or drawing a recursion tree, we see that $T(n) = \Theta(n^2)$. (We could also give an inductive proof.) Since the best-case time of insertion sort is $\Theta(n)$ and the worst-case time is $\Theta(n^2)$, it is interesting to know that the expected case is much closer to the worst-case than the best case.

Conditional Expected Values

Our next example is cooked up to introduce an idea that we often use in analyzing the expected running times of algorithms, especially randomized algorithms.

Exercise 6.6-3 I have two nickels and two quarters in my left pocket and 4 dimes in my right pocket. Suppose I flip a penny and take two coins from my left pocket if it is heads, and two coins from my right pocket if it is tails. Assuming I am equally likely to choose any coin in my pocket at any time, what is the expected amount of money that I draw from my pocket?

You could do this problem by drawing a tree diagram or by observing that the outcomes can be modeled by three tuples in which the first entry is heads or tails, and the second and third entries are coins. Thus our sample space is HNQ, HQN, HQQ, HNN, TDD The probabilities of these outcomes are $\frac{1}{6}$, $\frac{1}{6}$, $\frac{1}{12}$, $\frac{1}{12}$, and $\frac{1}{2}$ respectively. Thus our expected value is

$$30\frac{1}{6} + 30\frac{1}{6} + 50\frac{1}{12} + 10\frac{1}{12} + 20\frac{1}{2} = 25.$$

Here is a method that seems even simpler. If the coin comes up heads, I have an expected value of 15 cents on each draw, so with probability 1/2, our expected value is 30 cents. If the coin comes up tails, I have an expected value of ten cents on each draw, so with probability 1/2 our expected value is 20 cents. Thus it is natural to expect that our expected value is $\frac{1}{2}30 + \frac{1}{2}20 = 25$ cents. In fact, if we group together the 4 outcomes with an H first, we see that their contribution to the expected value is 15 cents, which is 1/2 times 30, and if we look at the single element which has a T first, then its contribution to the sum is 10 cents, which is half of 20 cents.

In this second view of the problem, we took the probability of heads times the expected value of our draws, given that the penny came up heads, plus the probability of tails times the expected value of our draws, given that the penny came came up tails. In particular, we were using a new (and as yet undefined) idea of *conditional expected value*. To get the conditional expected value if our penny comes up heads, we could create a new sample space with four outcomes, NQ, QN, NN, QQ, with probabilities $\frac{1}{3}$, $\frac{1}{3}$, $\frac{1}{6}$, and $\frac{1}{6}$. In this sample space the expected amount of money we draw in two draws is 30 cents (15 cents for the first draw plus 15 cents for the second), so we would say the conditional expected value of our draws, given that the penny came up heads, was 30 cents. With a one-element sample space $\{DD\}$, we see that we would say that the conditional expected value of our draws, given that the penny came up tails, is 20 cents.

How do we define conditional expected value? Rather than create a new sample space as we did above, we use the idea of a new sample space (as we did in discovering a good definition for conditional probability) to lead us to a good definition for conditional expected value. Namely, to get the conditional expected value of X given that an event F has happened we use our conditional probability weights for the elements of F, namely $P(x)/P(F)$ is the weight for the element x of F, and pretend F is our sample space. Thus we define the **conditional expected value** of X given F by

$$E(X|F) = \sum_{x:x\in F} X(x)\frac{P(x)}{P(F)}. \tag{6.40}$$

Remember that we defined the expected value of a random variable X with values $x_1, x_2, \ldots x_k$ by

$$E(X) = \sum_{i=1}^{k} x_i P(X = x_i),$$

where $X = x_i$ stands for the event that X has the value x_i. Using our standard notation for conditional probabilities, $P(X = x_i|F)$ stands for the conditional probability of the event $X = x_i$

given the event F. This lets us rewrite Equation 6.40 as

$$E(X|F) = \sum_{i=1}^{k} x_i P(X = x_i|F).$$

Theorem 6.22 *Let X be a random variable defined on a sample space S and let F_1, F_2, ... F_n be disjoint events whose union is S (i.e. a partition of S). Then*

$$E(X) = \sum_{i=1}^{n} E(X|F_i)P(F_i).$$

Proof: The proof is simply an exercise in applying definitions. ■

Randomized algorithms

Exercise 6.6-4 Consider an algorithm that, given a list of n numbers, prints them all out. Then it picks a random integer between 1 and 3. If the number is 1 or 2, it stops. If the number is 3 it starts again from the beginning. What is the expected running time of this algorithm?

Exercise 6.6-5 Consider the following variant on the previous algorithm:

```
funnyprint(n)
if (n == 1)
     return
for i = 1 to n
     print i
x = randint(1,n)
if (x > n/2)
     funnyprint(n/2)
else
     return
```

What is the expected running time of this algorithm?

For Exercise 6.6-4, with probability 2/3 we will print out the numbers and quit, and with probability 1/3 we will run the algorithm again. Using Theorem 6.22, we see that if $T(n)$ is the expected running time on a list of length n, then there is a constant c such that

$$T(n) = \frac{2}{3}cn + \frac{1}{3}(cn + T(n)),$$

which gives us $\frac{2}{3}T(n) = cn$. This simplifies to $T(n) = \frac{3}{2}cn$.

Another view is that we have an independent trials process, with success probability 2/3 where we stop at the first success, and for each round of the independent trials process we spend

$\Theta(n)$ time. Letting T be the running time (note that T is a random variable on the sample space $1, 2, 3$ with probabilities $\frac{1}{3}$ for each member) and R be the number of rounds, we have that

$$T = R \cdot \Theta(n)$$

and so

$$E(T) = E(R)\Theta(n).$$

Note that we are applying Theorem 6.10 since in this context $\Theta(n)$ behaves as if it were a constant[7], since n does not depend on R. By Lemma 6.12, we have that $E(R) = 3/2$ and so $E(T) = \Theta(n)$.

In Exercise 6.6-5, we have a recursive algorithm, and so it is appropriate to write down a recurrence. We can let $T(n)$ stand for the *expected* running time of the algorithm on an input of size n. Notice how we are changing back and forth between letting T stand for the running time of an algorithm and the expected running time of an algorithm. Usually we use T to stand for the quantity of most interest to us, either running time if that makes sense, or expected running time (or maybe worst-case running time) if the actual running time might vary over different inputs of size n. The nice thing will be that once we write down a recurrence for the expected running time of an algorithm, the methods for solving it will be those for we have already learned for solving recurrences. For the problem at hand, we immediately get that with probability $1/2$ we will be spending n units of time (we should really say $\Theta(n)$ time), and then terminating, and with probability $1/2$ we will spend n units of time and then recurse on a problem of size $n/2$. Thus using Theorem 6.22, we get that

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

Including a base case of $T(1) = 1$, we get that

$$T(n) = \begin{cases} \frac{1}{2}T(n/2) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}.$$

A simple proof by induction shows that $T(n) = \Theta(n)$. Note that the Master Theorem (as we originally stated it) doesn't apply here, since $a < 1$. However, one could also observe that the solution to this recurrence is no more than the solution to the recurrence $T(n) = T(n/2) + n$, and then apply the Master Theorem.

Selection revisited

We now return to the selection algorithm from Section 5.4. The purpose of the algorithm is to select the ith smallest element in a set with some underlying order. Recall that in this algorithm, we first picked an an element p in the middle half of the set, that is, one whose value was simultaneously larger than at least $1/4$ of the items and smaller than at least $1/4$ of the items. We used p to partition the items into two sets and then recursed on one of the two sets. If you recall, we worked very hard to find an item in the middle half, so that our partitioning would work well. It is natural to try instead to just pick a partition element at random, because, with probability $1/2$, this element will be in the middle half. We can extend this idea to the following algorithm:

[7]What we mean here is that $T \geq Rc_1 n$ for some constant c_1 and $T \leq Rc_2 n$ for some other constant c_2. Then we apply Theorem 6.10 to both these inequalities, using the fact that if $X > Y$, then $E(X) > E(Y)$ as well.

```
RandomSelect(A,i,n)
(selects the ith smallest element in set A, where n = |A| )
if (n = 1)
     return the one item in A
else
     p = randomElement(A)
     Let H be the set of elements greater than p
     Let L be the set of elements less than or equal to p
     If H is empty
          put p in H
     if (i ≤ |L|)
          Return RandomSelect(L, i, |L|)
     else
          Return RandomSelect(H, i − |L|, |H|)
```

Here randomElement(A) returns one element from A uniformly at random. We use this element as our partition element; that is, we use it to divide A into sets L and H with every element less than the partition element in L and every element greater than it in H. We add the special case when H is empty, to ensure that both recursive problems have size strictly less than n. This simplifies a detailed analysis, but is not strictly necessary. At the end of this section we will show how to get a recurrence that describes fairly precisely the time needed to carry out this algorithm. However, by being a bit less precise, we can still get the same big-O upper bound with less work.

When we choose our partition element, half the time it will be between $\frac{1}{4}n$ and $\frac{3}{4}n$. Then when we partition our set into H and L, each of these sets will have no more than $\frac{3}{4}n$ elements. The other half of the time each of H and L will have no more than n elements. In any case, the time to partition our set into H and L is $O(n)$. Thus we may write

$$T(n) \leq \begin{cases} \frac{1}{2}T(\frac{3}{4}n) + \frac{1}{2}T(n) + bn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

We may rewrite the recursive part of the recurrence as

$$\frac{1}{2}T(n) \leq \frac{1}{2}T\left(\frac{3}{4}n\right) + bn,$$

or

$$T(n) \leq T\left(\frac{3}{4}n\right) + 2bn = T\left(\frac{3}{4}n\right) + b'n.$$

Notice that it is possible (but unlikely) that each time our algorithm chooses a pivot element, it chooses the worst one possible, in which case the selection process could take n rounds, and thus take time $\Theta(n^2)$. Why, then, is it of interest? If involves far less computation than finding the median of medians, and its expected running time is still $\Theta(n)$. Thus it is reasonable to suspect that on the average, it would be significantly faster than the deterministic process. In fact, with good implementations of both algorithms, this will be the case.

Exercise 6.6-6 Why does every solution to the recurrence

$$T(n) \leq T\left(\frac{3}{4}n\right) + b'n.$$

have $T(n) = O(n)$?

By the master theorem we know that any solution to this recurrence is $O(n)$, giving a proof of our next Theorem.

Theorem 6.23 *Algorithm RandomSelect has expected running time $O(n)$.*

Quicksort

There are many algorithms that will efficiently sort a list of n numbers. The two most common sorting algorithms that are guaranteed to run in $O(n \log n)$ time are MergeSort and HeapSort. However, there is another algorithm, Quicksort, which, while having a worst-case running time of $O(n^2)$, has an expected running time of $O(n \log n)$. Moreover, when implemented well, it tends to have a faster running time than MergeSort or HeapSort. Since many computer operating systems and programs come with quicksort built in, it has become the sort of choice in many applications. In this section, we will see why it has expected running time $O(n \log n)$. We will not concern ourselves with the low-level implementation issues that make this algorithm the fastest one, but just with a high-level description.

Quicksort actually works similarly to the RecursiveSelect algorithm of the previous subsection. We pick a random element, and then use it to partition the set of items into two sets L and H. In this case, we don't recurse on one or the other, but recurse on both, sorting each one. After both L and H have been sorted, we just concatenate them to get a sorted list. (In fact, quicksort is usually done "in place" by pointer manipulation and so the concatenation just happens.) Here is a pseudocode description of quicksort.

```
Quicksort(A,n)
if (n = 1)
     return the one item in A
else
     p = randomElement(A)
     Let H be the set of elements greater than p; Let h = |H|
     Let L be the set of elements less than or equal to p; Let ℓ = |L|
     If H is empty
          put p in H
     A_1 = QuickSort(H,h)
     A_2 = QuickSort(L,ℓ)
     return the concatenation of A_1 and A_2
```

There is an analysis of quicksort similar to the detailed analysis of RecursiveSelect at the end of the section, and this analysis is a problem at the end of the section. Instead, based on the preceding analysis of RandomSelect we will think about modifying the algorithm a bit in order to make the analysis easier. First, consider what would happen if the random element was the median each time. Then we would be solving two subproblems of size $n/2$, and would have the recurrence

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases}$$

and we know by the master theorem that all solutions to this recurrence have $T(n) = O(n \log n)$. In fact, we don't need such an even division to guarantee such performance.

Exercise 6.6-7 Suppose you had a recurrence of the form

$$T(n) = \begin{cases} T(a_n n) + T((1-a_n)n) + O(n) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases},$$

where a_n is between 1/4 and 3/4. Show that all solutions of a recurrence of this form have $T(n) = O(n \log n)$. What do we really need to assume about a_n in order to prove this upper bound?

We can prove that $T(n) = O(n \log n)$ by induction, or via a recursion tree, noting that there are $O(\log n)$ levels, and each level has at most $O(n)$ work. (The details of the recursion tree are complicated somewhat by the fact that a_n varies with n, while the details of an inductive proof simply use the fact that a_n and $1 - a_n$ are both no more than 3/4.) So long as we know there is some positive number $a < 1$ such that $a_n < a$ for every n, then we know we have at most $\log_{(1/a)} n$ levels in a recursion tree, with at most cn units of work per level for some constant c, and thus we have the same upper bound in big-O terms.

What does this tell us? As long as our problem splits into two pieces, each having size at least 1/4 of the items, quicksort will run in $O(n \log n)$ time. Given this, we will modify our algorithm to enforce this condition. That is, if we choose a pivot element p that is not in the middle half, we will just pick another one. This leads to the following algorithm:

```
Slower Quicksort(A,n)
if (n = 1)
     return the one item in A
else
     Repeat
          p = randomElement(A)
          Let H be the set of elements greater than p; Let h = |H|
          Let L be the set of elements less than or equal to p; Let ℓ = |L|
     Until (|H| ≥ n/4) and (|L| ≥ n/4)
     A_1 = QuickSort(H,h)
     A_2 = QuickSort(L,ℓ)
     return the concatenation of A_1 and A_2
```

Now let's analyze this algorithm. Let r be the number of times we execute the loop to pick p, and let $a_n n$ be the position of the pivot element. Then if $T(n)$ is the expected running time for a list of length n, then for some constant b

$$T(n) \leq E(r)bn + T(a_n n) + T((1-a_n)n),$$

since each iteration of the loop takes $O(n)$ time. Note that we take the expectation of r, because $T(n)$ stands for the expected running time on a problem of size n. Fortunately, $E(r)$ is simple to compute, it is just the expected time until the first success in an independent trials process with

success probability 1/2. This is 2. So we get that the running time of Slower Quicksort satisfies the recurrence

$$T(n) \leq \begin{cases} T(a_n n) + T((1-a_n))n + b'n & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases},$$

where a_n is between 1/4 and 3/4. Thus by Exercise 6.6-7 the running time of this algorithm is $O(n \log n)$.

As another variant on the same theme, observe that looping until we have($|H| \geq n/4$ and $|L| \geq n/4$, is effectively the same as choosing p, finding H and L and then calling Slower Quicksort(A,n) once again if either H or L has size less than $n/4$. Then since with probability 1/2 the element p is between $n/4$ and $3n/4$, we can write

$$T(n) \leq \frac{1}{2}T(n) + \frac{1}{2}(T(a_n n) + T((1-a_n)n) + bn),$$

which simplifies to

$$T(n) \leq T(a_n n) + T((1-a_n)n) + 2bn,$$

or

$$T(n) \leq T(a_n n) + T((1-a_n)n) + b'n.$$

Again by Exercise 6.6-7 the running time of this algorithm is $O(n \log n)$.

Further, it is straightforward to see that the expected running time of Slower Quicksort is no less than half that of Quicksort (and, incidentally, no more than twice that of quicksort) and so we have shown:

Theorem 6.24 *Quicksort has expected running time $O(n \log n)$.*

A more exact analysis of RandomSelect

Recall that our analysis of the RandomSelect was based on using $T(n)$ as an upper bound for $T(|H|)$ or $T(|L|)$ if either the set H or the set L had more than $3n/4$ elements. Here we show how one can avoid this assumption. The kinds of computations we do here are the kind we would need to do if we wanted to try to actually get bounds on the constants implicit in our big-O bounds.

Exercise 6.6-8 Explain why, if we pick the kth element as the random element in RandomSelect ($k \neq n$), our recursive problem is of size no more than $\max\{k, n-k\}$.

If we pick the kth element, then we recurse either on the set L, which has size k, or on the set H which has size $n-k$. Both of these sizes are at most $\max\{k, n-k\}$. (If we pick the nth element, then $k = n$ and thus L actually has size $k-1$ and H has size $n-k+1$.)

Now let X be the random variable equal to the rank of the chosen random element (e.g. if the random element is the third smallest, $X = 3$.) Using Theorem 6.22 and the solution to Exercise 6.6-8, we can write that

$$T(n) \leq \begin{cases} \sum_{k=1}^{n-1} P(X=k)(T(\max\{k, n-k\}) + bn) + P(X=n)(T(\max\{1, n-1\} + bn) & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

Since X is chosen uniformly between 1 and n, $P(X = k) = 1/n$ for all k. Ignoring the base case for a minute, we get that

$$\begin{aligned} T(n) &\leq \sum_{k=1}^{n-1} \frac{1}{n}(T(\max\{k, n-k\}) + bn) + \frac{1}{n}(T(n-1) + bn) \\ &= \frac{1}{n}\left(\sum_{k=1}^{n-1} T(\max\{k, n-k\})\right) + bn + \frac{1}{n}(T(n-1) + bn). \end{aligned}$$

Now if n is odd and we write out $\sum_{k=1}^{n-1} T(\max\{k, n-k\})$, we get

$$T(n-1) + T(n-2) + \cdots + T(\lceil n/2 \rceil) + T\lceil n/2 \rceil) + \cdots + T(n-2) + T(n-1),$$

which is just $2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)$. If n is even we write out $\sum_{k=1}^{n-1} T(\max\{k, n-k\})$, we get

$$T(n-1) + T(n-2) + \cdots + T(n/2) + T(1 + n/2) + \cdots + T(n-2) + T(n-1),$$

which is less than $2\sum_{k=n/2}^{n-1} T(k)$. Thus we can replace our recurrence by

$$T(n) \leq \begin{cases} \frac{2}{n}\left(\sum_{k=n/2}^{n-1} T(k)\right) + \frac{1}{n}T(n-1) + bn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases} \tag{6.41}$$

If n is odd, the lower limit of the sum is a half-integer, so the possible integer values of the dummy variable k run from $\lceil n/2 \rceil$ to $n-1$. Since this is the natural way to interpret a fractional lower limit, and since it corresponds to what we wrote in both the n even and n odd case above, we adopt this convention.

Exercise 6.6-9 Show that every solution to the recurrence in Equation 6.41 has $T(n) = O(n)$.

We can prove this by induction. We try to prove that $T(n) \leq cn$ for some constant c. By the natural inductive hypothesis, we get that

$$\begin{aligned} T(n) &\leq \frac{2}{n}\left(\sum_{k=n/2}^{n-1} ck\right) + \frac{1}{n}c(n-1) + bn \\ &= \frac{2}{n}\left(\sum_{k=1}^{n-1} ck - \sum_{k=1}^{n/2-1} ck\right) + \frac{1}{n}c(n-1) + bn \\ &\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\frac{n}{2}-1)\frac{n}{2}}{2}\right) + c + bn \\ &= \frac{2c}{n}\frac{\frac{3n^2}{4} - \frac{n}{2}}{2} + c + bn \\ &= \frac{3}{4}cn + \frac{c}{2} + bn \\ &= cn - (\frac{1}{4}cn - bn - \frac{c}{2}) \end{aligned}$$

Notice that so far, we have only assumed that there is some constant c such that $T(k) < ck$ for $k < n$. We can choose a larger c than the one given to us by this assumption without changing

the inequality $T(k) < ck$. By choosing c so that $\frac{1}{4}cn - bn - \frac{c}{2}$ is nonnegative (for example $c \geq 8b$ makes this term at least $bn - 2b$ which is nonnegative for $n \geq 2$), we conclude the proof, and have another proof of Theorem 6.23.

This kind of careful analysis arises when we are trying to get an estimate of the constant in a big-O bound (which we decided not to do in this case).

Important Concepts, Formulas, and Theorems

1. *Expected Running Time.* When the running time of an algorithm is different for different inputs of the same size, we can think of the running time of the algorithm as a random variable on the sample space of inputs and analyze the expected running time of the algorithm. This us a different understanding from studying just the worst case running time.

2. *Randomized Algorithm.* A*randomized algorithm* is an algorithm that depends on choosing something randomly.

3. *Random Number Generator.* A random number generator is a procedure that generates a number that appears to be chosen at random. Usually the designer of a random number generator tries to generate numbers that appear to be uniformly distributed.

4. *Insertion Sort.* A recursive description of insertion sort is that to sort $A(1 : n)$, first we sort $A(1 : n-1)$, and then we insert $A(n)$, by shifting the elements greater than $A(n)$ each one place to the right and then inserting the original value of $A(n)$ into the place we have opened up. If $n = 1$ we do nothing.

5. *Expected Running Time of Insertion Sort.* If $T(n)$ is the expected time to use insertion sort on a list of length n, then there are constants c and c' such that $T(n) \leq T(n-1) + cn$ and $T(n) \geq T(n-1) + c'n$,. This means that $T(n) = \Theta(n^2)$. However the best case running time of insertion sort is $\Theta(n)$.

6. *Conditional Expected Value.* We define the *conditional expected value* of X given F by $E(X|F) = \sum_{x:x\in F} X(x)\frac{P(x)}{P(F)}$. This is equivalent to $E(X|F) = \sum_{i=1}^{k} x_i P(X = x_i|F)$.

7. *Randomized Selection Algorithm.* In the randomized selection algorithm to select the ith smallest element of a set A, we randomly choose a pivot element p in A, divide the rest of A into those elements that come before p (in the underlying order of A) and those that come after, put the pivot into the smaller set, and then recursively apply the randomized selection algorithm to find the appropriate element of the appropriate set.

8. *Running Time of Randomized Select.* Algorithm RandomSelect has expected running time $O(n)$. Because it does less computation than the deterministic selection algorithm, on the average a good implementation will run faster than a good implementation of the deterministic algorithm, but the worst case behavior is $\Theta(n^2)$.

9. *Quicksort.* Quicksort is a sorting algorithm in which we randomly choose a pivot element p in A, divide the rest of A into those elements that come before p (in the underlying order of A) and those that come after, put the pivot into the smaller set, and then recursively apply the Quicksort algorithm to sort each of the smaller sets, and concatenate the two sorted lists. We do nothing if a set has size one.

10. *Running Time of Quicksort.* Quicksort has expected running time $O(n \log n)$. It has worst case running time $\Theta(n^2)$. Good implementations of Quicksort have proved to be faster on the average than good implementations of other sorting algorithms.

Problems

1. Given an array A of length n (chosen from some set that has an underlying ordering), we can select the largest element of the array by starting out setting $L = A(1)$, and then comparing L to the remaining elements of the array one at a time, replacing L by $A(i)$ if $A(i)$ is larger than L. Assume that the elements of A are randomly chosen. For $i > 1$, let X_i be 1 if element i of A is larger than any element of $A(1 : i-1)$. Let $X_1 = 1$. Then what does $X_1 + X_2 + \cdots + X_n$ have to do with the number of times we assign a value to L? What is the expected number of times we assign a value to L?

2. Let $A(i : j)$ denote the array of items in positions i through j of the Array A. In selection sort, we use the method of Exercise 6.6-1 to find the largest element of the array A and its position k in the array, then we exchange the elements in position k and n of Array A, and we apply the same procedure recursively to the array $A(1 : n-1)$. (Actually we do this if $n > 1$; if $n = 1$ we do nothing.) What is the expected total number of times we assign a value to L in the algorithm selection sort?

3. Show that if H_n stands for the nth harmonic number, then

$$H_n + H_{n-1} + \cdots + H_2 = \Theta(n \log n).$$

4. In a card game, we remove the Jacks, Queens, Kings, and Aces from a deck of ordinary cards and shuffle them. You draw a card. If it is an Ace, you are paid a dollar and the game is repeated. If it is a Jack, you are paid two dollars and the game ends; if it is a Queen, you are paid three dollars and the game ends; and if it is a King, you are paid four dollars and the game ends. What is the maximum amount of money a rational person would pay to play this game?

5. Why does every solution to $T(n) \leq T(\frac{2}{3}n) + bn$ have $T(n) = O(n)$?

6. Show that if in Algorithm Random Select we remove the instruction

```
If H is empty
     put p in H,
```

then if $T(n)$ is the expected running time of the algorithm, there is a constant b such that $T(n)$ satisfies the recurrence

$$T(n) \leq \frac{2}{n-1} \sum_{k=n/2}^{n-1} T(k) + bn.$$

Show that if $T(n)$ satisfies this recurrence, then $T(n) = O(n)$.

7. Suppose you have a recurrence of the form

$$T(n) \leq T(a_n n) + T((1 - a_n)n) + bn \text{ if } n > 1,$$

where a_n is between $\frac{1}{5}$ and $\frac{4}{5}$. Show that all solutions to this recurrence are of the form $T(n) = O(n \log n)$.

8. Prove Theorem 6.22.

9. A tighter (up to constant factors) analysis of quicksort is possible by using ideas very similar to those that we used for the randomized selection algorithm. More precisely, we use Theorem 5.6.1, similarly to the way we used it for select. Write down the recurrence you get when you do this. Show that this recurrence has solution $O(n \log n)$. In order to do this, you will probably want to prove that $T(n) \leq c_1 n \log n - c_2 n$ for some constants c_1 and c_2.

10. It is also possible to write a version of rhe randomized Selection algorithm analogous to Slower Quicksort. That is, when we pick out the random pivot element, we check if it is in the middle half and discard it if it is not. Write this modified selection algorithm, give a recurrence for its running time, and show that this recurrence has solution $O(n)$.

11. One idea that is often used in selection is that instead of choosing a random pivot element, we choose three random pivot elements and then use the median of these three as our pivot. What is the probability that a randomly chosen pivot element is in the middle half? What is the probability that the median of three randomly chosen pivot elements is in the middle half? Does this justify the choice of using the median of three as pivot?

12. Is the expected running time of Quicksort $\Omega(n \log n)$?

13. A random binary search tree on n keys is formed by first randomly ordering the keys, and then inserting them in that order. Explain why in at least half the random binary search trees, both subtrees of the root have between $\frac{1}{4}n$ and $\frac{3}{4}n$ keys. If $T(n)$ is the expected height of a random binary search tree on n keys, explain why $T(n) \leq \frac{1}{2}T(n) + \frac{1}{2}T(\frac{3}{4}n) + 1$. (Think about the **definition** of a binary tree. It has a root, and the root has two subtrees! What did we say about the possible sizes of those subtrees?) What is the expected height of a one node binary search tree? Show that the expected height of a random binary search tree is $O(\log n)$.

14. The expected time for an unsuccessful search in a random binary search tree on n keys (see Problem 13 for a definition) is the expected depth of a leaf node. Arguing as in Problem 13 and the second proof of Theorem 5.6.2, find a recurrence that gives an upper bound on the expected depth of a leaf node in a binary search tree and use it to find a big Oh upper bound on the expected depth of a leaf node.

15. The expected time for a successful search in a random binary search tree on n nodes (see problem 13 for a definition) is the expected depth of a node of the tree. With probability $\frac{1}{n}$ the node is the root, which has depth 0; otherwise the expected depth is one plus the expected depth of one of its subtrees. Argue as in Problem 13 and the first proof of Theorem 6.23 to show that if $T(n)$ is the expected depth of a node in a binary search tree, then $T(n) \leq \frac{n-1}{n}(\frac{1}{2}T(n) + \frac{1}{2}T(\frac{3}{4}n)) + 1$. What big Oh upper bound does this give you on the expected depth of a node in a random binary search tree on n nodes?

16. Consider the following code for searching an array A for the maximum item:

```
max = -∞
for i = 1 to n
    if (A[i] > max)
        max = A[i]
```

If A initially consists of n nodes in a random order, what is the expected number of times that the line $max = A[i]$ is executed? (Hint: Let X_i be the number of times that $max = A[i]$ is executed in the ith iteration of the loop.)

17. You are a contestant in the game show "Let's make a Deal." In this game show, there are three curtains. Behind one of the curtains is a new car, and behind the other two are cans of spam. You get to pick one of the curtains. After you pick that curtain, the emcee, Monte Hall, who we assume knows where the car is, reveals what is behind one of the curtains that you did not pick, showing you some cans of spam. He then asks you if you would like to switch your choice of curtain. Should you switch? Why or why not? Please answer this question carefully. You have all the tools needed to answer it, but several math Ph.D.'s are on record (in Parade Magazine) giving the wrong answer.