CHAPTER 5. PROBABILITY

5.6 Conditional Expectations, Recurrences and Algorithms

Probability is a very important tool in algorithm design. We have already seen several 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 in computing the running time of various algorithms, and will introduce the idea of using recurrences to evaluate 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 $r_2 - r_1 = s_2 - s_1$ our random number is just as likely to be between $r_1$ and $r_2$ as it is to be between $s_1$ and $s_2$].

5.6-1 Consider the following algorithm for sorting $n$ items. I flip a coin. If it is heads, I run mergesort and tails I run insertion sort. What is the big-O expected running time of this algorithm?

5.6-2 Consider an algorithm that given a list of $n$ numbers, prints them all out. Then it picks a random number between 1 and 3. If the number is 1 or 2, it stops. If the number is 3 it runs again. What is the expected running time of this algorithm?

5.6-3 Consider the following variant on the previous algorithm:

```python
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 5.6-1, we have a sample space with two items, mergesort and insertion sort, thus we can write the expected running time as

$$P(\text{heads})T(\text{mergesort}) + P(\text{tails})T(\text{insertion sort}) = .5 \cdot O(n \log n) + .5 \cdot O(n^2) = O(n^2).$$

Another view of this problem is a conditional probability view. We could think of our sample space of consisting of all sequences of length $n$. We have computed our expected value by taking the expected value of mergesort times the probability of using mergesort plus the expected value of insertion sort times the probability of using insertion sort. In effect, our sample space now
consists not simply of \( n \) tuples of items to be sorted, but actually of \( n + 1 \)-tuples, whose last entry is either \( M \) for merge sort or \( I \) for insertion sort. Then we divide our new sample space up into one event consisting of the tuples ending in \( M \) and one consisting of those tuples ending in \( I \), compute “conditional expected values” (which we haven’t defined yet) \( E(X|M) \) and \( E(X|I) \), and then we write our expected value as \( E(X) = E(X|M)P(M) + E(X|I)P(I) \). Of course this example is set up perfectly for giving us this intuition; usually the events on which we condition the expected value are not cooked up in this way but rather are intrinsic to our problem.

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 \( \frac{P(x)}{P(F)} \) 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

\[
E(X|F) = \sum_{x:x \in F} \frac{X(x)P(x)}{P(F)}.
\]

**Theorem 5.6.1** Let \( X \) be a random variable defined on a sample space and the \( F_1, F_2, \ldots F_n \) 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.

For Exercise 5.6-2, we have an independent trials process, with success probability \( \frac{1}{3} \) where we stop at the first success, and for each round of the independent trials process we spend \( O(n) \) time.

Letting \( T \) be the running time and \( R \) be the number of rounds, we have that

\[
T = R \cdot O(n)
\]

and so

\[
E(T) = E(R)O(n).
\]

Note that we are applying Theorem 5.4.2 since in this context \( O(n) \) is serving as a constant, since it is independent of \( R \). By Lemma 5.4.4, we have that \( E(R) = 3 \) and so \( E(T) = O(n) \).

In Exercise 5.6-2, 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 \( n \). Note that this is an abuse of notation, but it will simplify our exposition. 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 the same as those for solving normal recurrences. For the problem at hand, we immediately get that with probability \( \frac{1}{2} \) we will be spending \( n \) time (we should really say \( O(n) \) time), and then terminating, and with probability \( \frac{1}{2} \) we will spend \( n \) time and then recurse on a problem of size \( n/2 \). Thus using Theorem 5.6.1, we get that

\[
T(n) = \frac{1}{2}n + \frac{1}{2}(T(n/2) + n)
\]
Simplifying and 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 \( T(n) = O(n) \). Note that the master theorem 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.

It might seem strange that we are defining expected value recursively, but this is no stranger than defining anything else recursively. More precisely, it is well-defined, since we have a base case, and we can build up the case for any \( n \) inductively from the base cases. Note that the difference here is that the rule for defining \( T(n) \) from smaller values involves probability calculations. However, these are just linear equations and thus, because of the linearity of expectation, pose no additional difficulties.

**Selection revisited**

We now return to the selection algorithm from Section ???. Recall that in this algorithm, we tried to pick an item from the middle half, used this to partition the items 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:

\[
\text{RandomSelect}(A, i, n) \\
(\text{selects the } i\text{th smallest element in set } A, \text{ where } n = |A|) \\
\text{if } (n = 1) \\
\quad \text{return the one item in } A \\
\text{else} \\
\quad p = \text{randomElement}(A) \\
\quad \text{Let } H \text{ be the set of elements greater than } p \\
\quad \text{Let } L \text{ be the set of elements less than or equal to } p \\
\quad \text{If } H \text{ is empty} \\
\quad \quad \text{put } p \text{ in } H \\
\quad \text{if } (i \leq |L|) \\
\quad \quad \text{Return Select1}(L, i, |L|) \\
\text{else} \\
\quad \quad \text{Return Select1}(H, i, i - |L|)
\]

Here \( \text{randomElement}(A) \) returns one element from \( A \) uniformly at random. We add the special case when \( H \) is empty, to ensure that both recursive problems have size strictly less than \( n \). This will simplify the analysis, but is not strictly necessary. In order to analyze this algorithm, we will give a recurrence to describe the running time.

5.6-4 Explain why, if we pick the \( k \)th element as the random element \( (k \neq n) \), our recursive problem is of size no more than \( \max\{k, n - k\} \).
If we pick the $k$th 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 $n$th 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 5.6.1 and the solution to Exercise 5.6-4, we can write that

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

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

$$T(n) \leq \sum_{k=1}^{n-1} \frac{1}{n}(T(\max\{k, n - k\}) + O(n)) + \frac{1}{n}(T(\max\{1, n - 1\}) + O(n))$$

$$= \frac{1}{n} \left( \sum_{k=1}^{n-1} T(\max\{k, n - k\}) \right) + O(n) + \frac{1}{n}(T(\max\{1, n - 1\}) + O(n))$$

Now if we write out $\sum_{k=1}^{n-1} T(\max\{k, n - k\})$, we get $T(n - 1) + T(n - 2) + \cdots + T(n/2) + T(n/2) + \cdots + T(n - 2) + T(n - 1)$, which is just $2 \sum_{k=n/2}^{n-1} T(k)$, so we can simplify our recurrence to

$$T(n) \leq \begin{cases} \frac{2}{n} \left( \sum_{k=n/2}^{n-1} T(k) \right) + \frac{1}{n}T(n - 1) + O(n) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases}.$$ (5.27)

5.6-5 Show that the recurrence in Equation 5.27 has solution $T(n) = O(n)$.

We can prove this by induction. We let $b$ be the constant implicit in the $O(n)$ term in the recurrence, and try to prove that $T(n) \leq cn$. By induction, we get that

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

$$= \left( \sum_{k=1}^{n-1} ck - \sum_{k=1}^{\lfloor n/2 \rfloor} ck \right) + \frac{1}{n}c(n - 1) + bn$$

$$\leq \frac{2c}{n} \left( \frac{(n - 1)n}{2} - \frac{\left(\frac{n}{2} - 1\right)\frac{n}{2}}{2} \right) + c + bn$$

$$= \frac{2c}{n} \left( \frac{3n^2 - 1}{2} - \frac{2}{2} \right) + c + bn$$

$$\leq \frac{3}{4}cn + \frac{c}{2} + bn$$

$$= cn - (\frac{1}{4}cn - bn - \frac{c}{2})$$

By choosing $c$ so that $\frac{1}{4}cn - bn - \frac{c}{2}$ is nonnegative (for example $c = 8b$ makes this term $bn - 2b$ which is nonnegative for $n \geq 2$), we conclude the proof, and have shown that
Theorem 5.6.2 Algorithm RandomSelect has expected running time \( O(n) \).

This kind of careful analysis arises when we are trying to get an estimate of the constant in a big Oh bound (which we decided not to do in this case). There is another point of view that leads to a “sloppier” upper bound on the time but gives the same big Oh result. 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

\[
T(n) \leq \begin{cases} 
\frac{1}{2}T(\frac{3}{4}n) + \frac{1}{2}T(n) + O(n) & \text{if } n > 1 \\
O(1) & \text{if } n = 1 
\end{cases}.
\]

We may rewrite the recursive term of the recurrence as

\[
\frac{1}{2}T(n) \leq \frac{1}{2}T(\frac{3}{4}n) + O(n),
\]

or

\[
T(n) \leq T(\frac{3}{4}n) + 2O(n) = T(\frac{3}{4}n) + O(n).
\]

By the master theorem we know that any solution to this recurrence is \( O(n) \), giving another proof of Theorem 5.6.2.

Quicksort

There are many algorithms that will efficiently sort a list of \( n \) numbers. If we restrict ourselves to sorting algorithms that are guaranteed to run in \( O(n \log n) \) time, then the two most common algorithms 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. In fact 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 divide 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” and so the concatenation just happens.) We can write the code for 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 \ell = |L|
```
If $H$ is empty
   put $p$ in $H$
$A_1 = \text{QuickSort}(H,h)$
$A_2 = \text{QuickSort}(L,\ell)$
return the concatenation of $A_1$ and $A_2$

We could analyze quicksort similarly to the way we analyzed Recursive Select, and this is an exercise at the end of the section. Instead, we will think about modifying the algorithm a bit in order to make the analysis easier. First, consider what would happen if each time, the random element was the median. 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.

5.6-6 Suppose you had a recurrence of the form

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

where $a$ is between $1/4$ and $3/4$. Show that any recurrence of this form has a solution of $T(n) = O(n \log n)$.

The above can easily be proved by induction, or via a recursion tree, noting that there are $O(\log n)$ levels, and each level has at most $O(n)$ work.

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**

```plaintext
if ($n = 1$)
   return the one item in $A$
else
   Repeat
      $p = \text{randomElement}(A)$
      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 $\ell = |L|$
   Until ($|H| \geq n/4$) and ($|L| \geq n/4$)
   $A_1 = \text{QuickSort}(H,h)$
   $A_2 = \text{QuickSort}(L,\ell)$
   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 \( an \) be the position of the pivot element, then
\[
T(n) = E(r)O(n) + T(an) + T((1 - a)n),
\]
since each time through the loop is \( O(n) \). Note that we take the expectation of \( r \), because of our convention that \( T(n) \) now 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
\[
T(n) = \begin{cases} 
T(an) + T((1 - a)n) + O(n) & \text{if } n > 1 \\
O(1) & \text{if } n = 1
\end{cases},
\]
where \( a \) is between \( 1/4 \) and \( 3/4 \). Thus by Exercise 5.6-6 the running time of this algorithm is \( O(n \log n) \).

Further, it is easy to see that the running time of Slower Quicksort is no faster than that of Quicksort and so we have shown:

**Theorem 5.6.3** \( \text{Quicksort has expected running time } O(n \log n) \).

**Exercises**

E5.6-1 Prove Theorem 5.6.1.

E5.6-2 A tighter (up to constant factors) analysis of quicksort is possible by using ideas very similar to those that we used for the randomized median algorithm. More precisely, we use can Theorem 5.6.1, similarly to the way we used it for median. 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 \( c_2 \).

E5.6-3 It is also possible to write a version of Selection analogous to Slower Quicksort. That is, when we pick out random pivot element, we check if it is in the middle half and discard it if it is not. Write this modified median algorithm, give a recurrence for its running time, and show that this recurrence has solution \( O(n) \).

E5.6-4 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?

E5.6-5 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 \( \frac{1}{3} n \) and \( \frac{2}{3} n \) keys. If \( T(n) \) is the expected height of a random binary search tree on \( n \) keys, explain why
5.6. CONDITIONAL EXPECTATIONS, RECURRENCES AND ALGORITHMS

\[ T(n) \leq \frac{1}{2} T(n) + \frac{1}{2} T\left(\frac{3}{4}n\right) + 1. \] (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 \( O(\log n) \).

E5.6-6 The expected time for an unsuccessful search in a random binary search tree on \( n \) keys (see Exercise 5.6-5 for a definition) is the expected depth of a leaf node. Arguing as in Exercise 5.6-5 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.

E5.6-7 The expected time for a successful search in a random binary search tree on \( n \) nodes (see Exercise 5.6-5 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 the expected depth of one of its children. Argue as in Exercise 5.6-5 and the second proof of 5.6.2 to show that if \( T(n) \) is the expected depth of a node in a binary search tree, then \( T(n) \leq \frac{n-1}{n} \left(\frac{1}{2} T(n) + \frac{1}{2} T\left(\frac{3}{4}n\right)\right) + 1 \). 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?

E5.6-8 Consider the following code for searching an array \( A \) for the maximum item:

\[
m = -\infty
\]

\[
\text{for } i = 1 \text{ to } n
\]

\[
\text{if } (A[i] > max)
\]

\[
max = A[i]
\]

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 \( i \)th iteration of the loop.)