Lecture 15: Expected running time

Plan/outline

Last class, we saw algorithms that used randomization in order to achieve interesting speed-ups. Today, we consider algorithms in which the running time is a random variable, and study the notion of the "expected" running time.

Las Vegas algorithms

Most of the algorithms we saw in the last class had the following behavior: they run in time $O(r \times f(n))$, where n is the input size and r is some parameter, and they have a *failure probability* (i.e., probability of returning an incorrect answer) of $\exp(-r)$. These algorithms are often called Monte Carlo algorithms (although we refrain from doing this, as it invokes different meanings in applied domains).

Another kind of algorithms, known as **Las Vegas** algorithms, have the following behavior: their output is **always correct**, but the running time is a random variable (and in principle, the run time can be unbounded).

Sometimes, we can convert a Monte Carlo algorithm into a Las Vegas one. For instance, consider the toy problem we considered last time (where an array A is promised to have $n/3$ indices i for which $A[i] = 0$ and the goal was to find one such index). Now consider the following modification of the algorithm:

```
procedure find_Vegas(A):
  pick random index i in [0, ..., N-1]
  while (A[i] != 0):
    pick another random index i in [0, ..., N-1]
  end while
  return i
```

Whenever the procedure terminates, we get an i that satisfies $A[i] = 0$. But in principle, the algorithm can go on for ever.

Another example of a Las Vegas algorithm is *quick sort*, which is a simple procedure for sorting an array.

Quick sort

Suppose we are given an unsorted array A with n elements. Suppose for simplicity that all the elements of A are **distinct**. The quicksort procedure works as follows:

```
procedure quickSort(A):
  if (|A|=1), break and return A;
  let i be a random index in [0, ..., |A|-1]
  create new arrays B, C, where B contains all elements < A[i], and C contains all elements >
A[i]
  recursively run quickSort(B) and quickSort(C)
  return the sorted array B, followed by A[i], followed by sorted array C
```

The random index i chosen in the first step is often called the *pivot* element. Suppose for a moment that the pivot always turned out to be the median (i.e. the $n/2$th smallest element). Then we have the recurrence of $T(n) = 2T(n/2) + O(n)$, which results in a bound of $T(n) = O(n \log n)$, which matches the best algorithm we've seen for sorting (merge sort).

But on the other hand, suppose we were unlucky, and the pivot always turns out to be the largest element of the array. Then, B would have size $|A| - 1$, and it is easy to see that this results in an overall running time of $\Theta(n^2)$. Note that for this to happen, we need to be unlucky in each of the recursive steps; indeed, a little bit of analysis shows that the probability of this happening is $\approx \exp(-n)$.

Can we reason about the "typical" running time? Can we say that the running time is not $> n^{1.5}$ with a pretty good probability? We will see how to answer these kinds of questions.

Expected running time

One key observation in the algorithms above is that the running time depends on the random choices, so in other words, it is a **random variable (r.v.)**. Thus, the questions we asked above about the running time are, in effect, properties about the distribution of this random variable.

Recall that the *expected value* of a random variable X with probability density function p is, by definition, $\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot p(x) dx$.

For a discrete random variable (one that takes values in a discrete set S), we have $\mathbb{E}[X] = \sum_{x \in S} x \Pr[X = x]$.

We can try using this definition to compute the running time of the algorithms above. Let us start with the procedure find_Vegas(): by definition, if the while loop executes r times (each execution takes const time, assuming that finding a random index takes const time), it means that the first $r-1$ times resulted in $A[i] \neq 0$ and the rth time resulted in $A[i] = 0$. This happens with probability $(2/3)^{r-1}(1/3)$.

Thus, the expected running time (for simplicity, assume that each execution of the loop takes 1 time step) is: $\sum_{r=1}^{\infty} r \cdot \left(\frac{2}{3}\right)^{r-1} \frac{1}{3}$.

Note that this is precisely like asking the expected number of tosses needed to see heads when tossing a coin whose probability of heads is $1/3$. The expectation turns out to be 3 -- but note that this involves summing an appropriate infinite series.

What about QuickSort? Note that it is now totally unclear how one estimates $\Pr[X = r]$, where X is the r.v. denoting the running time.

Thus, we need a nicer way of computing the expected value. The key turns out to be to come up with **recurrences for the expected value**. Let us illustrate it first with the case of find_Vegas().

We will need one key property of the expected value:

Law of conditional expectation (also called law of total expectation, and other names). Let X be a random variable, and let F be *any* event. Then we have $\mathbb{E}[X] = \Pr[F] \cdot \mathbb{E}[X|F] + \Pr[\overline{F}] \cdot \mathbb{E}[X|\overline{F}]$.

In the above, $\overline{F}$ is the event that F *does not* occur. Also, $\mathbb{E}[X|F]$ is the conditional expectation (i.e., the expectation of the distribution of X conditioned on F).

Indeed, a more general version of the law also holds: let $F_1, F_2, \ldots, F_k$ be some set of disjoint events whose union is the entire probability (i.e., precisely one of these events always occurs). Then we have $\mathbb{E}[X] = \Pr[F_1] \cdot \mathbb{E}[X|F_1] + \Pr[F_2] \cdot \mathbb{E}[X|F_2] + \cdots + \Pr[F_k] \cdot \mathbb{E}[X|F_k]$.

Clearly, if we only have two events F and $\overline{F}$ we recover the earlier formulation.

Let us see how this is useful to analyze find_Vegas(). Let us denote by α the expected running time, i.e., $\alpha = \mathbb{E}[X]$. Now, let F be the event that in the *first* iteration, we find index i such that $A[i] = 0$. Clearly, $\Pr[F] = 1/3$ and $\Pr[\overline{F}] = 2/3$.

Plugging this into the law of conditional expectation, we get $\alpha = \frac{1}{3}\mathbb{E}[X|F] + \frac{2}{3}\mathbb{E}[X|\overline{F}]$.

Now, the first expectation is clearly 1 (if the first index satisfies $A[i] = 0$, the running time is 1).

We next observe that the second expectation is precisely $1 + \alpha$. This is because we *know* that the first iteration was unsuccessful, and given that the process is history-independent, the situation after the first iteration is precisely the same as the original one.

Plugging these values in, we get $\alpha = \frac{1}{3} + \frac{2}{3}(1 + \alpha)$. Solving, we get $\alpha = 3$ (no infinite summations!)

Let us now see if this viewpoint lets us do something more non-trivial, like quickSort. To do this, let us define

$f(n) :=$ expected running time of quickSort() on an array of size n.

Implicit in this definition is the observation that the expected running time is independent of the array itself, and it depends only on the size. This makes sense since we're dealing with arrays with all distinct elements, and the procedure never uses anything about the original order of the elements.

We would now like to use the law of conditional expectation. For this, let us define F_i, for $1 \leq i \leq n$ to be the event that the ith smallest element is chosen as the pivot. Clearly, the events partition the probability space (in every run, precisely one element is chosen as the pivot). Also, because we choose a random element, we have $\Pr[F_i] = 1/n$ for all i.

Thus we have $f(n) = \mathbb{E}[X] = \sum_{i=1}^{n} \frac{1}{n}\mathbb{E}[X|F_i]$.

Now, for any i, $\mathbb{E}[X|F_i] = f(i-1) + f(n-i-1) + O(n)$, because $(i-1)$ and $(n-i-1)$ are the sizes of the recursive sub-problems, and we do $O(n)$ additional work. [Here we use the convention that $f(0) = 0$; let us also denote the $O(n)$ term going forward by cn.]

This ends up with the recurrence $f(n) = cn + \frac{1}{n}\sum_{i=1}^{n}\left(f(n-i-1) + f(i-1)\right)$.

A bit of simplification yields $f(n) = cn + \frac{2}{n}\sum_{i=1}^{n-1} f(i)$.

Before simplifying, note what we have done: by defining $f(n)$ to be the expected running time on an array of size n, we have obtained a recurrence for $f(n)$, which we'll use to bound $f(n)$.

We claim by induction that $f(n) \leq 2cn\log n$. ('log' refers to the natural logarithm here)

Suppose the inequality holds for all integers $< n$. Then we need to prove that: $cn + \frac{2}{n}\sum_{i=1}^{n-1} 2ci\log i \leq 2cn\log n$. (**)

The "tricky" term is obviously $\frac{1}{n}\sum_{i=1}^{n} i\log i$. One way to simplify it is as follows:

$\sum_{i=1}^{n} \frac{i}{n}\log i = \sum_{i=1}^{n} \frac{i}{n}\log \frac{i}{n} + \sum_{i=1}^{n} \frac{i}{n}\log n$

Now, using the standard trick of estimating a summation using an integral, this is asymptotically equal to:

$n \cdot \left(\int_0^1 x\log x dx + \log n \cdot \int_0^1 x dx\right) = -n/4 + \frac{n}{2}\log n$. [You can evaluate the integrals using Wolfram Alpha if you're not comfortable with integration.]

Plugging this in, the LHS of (**) is less or equal to $cn + \frac{4c}{n}\left[-\frac{n}{4} + \frac{n}{2}\log n\right] = 2cn\log n$, as desired.

Comparing to merge sort. Quick sort's expected running time is asymptotically the same as the running time of merge sort. But in many implementations (e.g., Unix sort), quick sort is the preferred sort procedure. It turns out to be easy to implement, and one can easily sort "in place" (using $O(1)$ extra memory ---- although the way we stated it formed new arrays B and C, which takes $O(n)$ additional memory; it's a nice exercise to avoid doing this).