CS 240 — Lecture 7

Bartosz Antczak *Instructor: Eric Schost* *January 24, 2017*

7.1 Randomized Algorithms

A **randomized algorithm** is an algorithm with such an implementation that relies on generating random numbers in addition to the input.
Note that computers *can't actually generate* random numbers. The generated number always depends on some external factors, such as the system clock most notably. Using these external sources is expensive, so instead we use a *pseudo-random number generator* (PRNG), which is a deterministic program that uses a true-random initial value or seed. This approach is faster than the other method.

7.1.1 Expected Running Time

The expected running time, denoted $T^{(\exp)}(I)$, of a randomized algorithm for a particular input I and a sequence of random numbers R is

$$T^{(\exp)}(I) = E[T(I,R)] = \sum_R T(I,R) \cdot \Pr[R]$$

($\Pr[R]$ denotes is the probability of getting the particular value in R). Note that the expected runtime is defined as *expected* because it considers the statistical probability of R.
For randomized algorithms, the best-, worst-case scenario are at about the same running time.

7.1.2 Randomized QuickSelect

How would we want to randomize the quick select algorithm?

First Method

Randomly permute the input (an array `A`) first using shuffle.

```
shuffle(A) {
  for i = 0 to n - 2 do
    swap(A[i], A[i + random(n-i)]) // random(n) returns an integer from 0 to n-1
}
```

(Note: we loop until the second last element in the array to avoid swapping the last element with itself, which is redundant) The expected cost becomes the same as the average cost, which is $\Theta(n)$, since we assume that the probability of getting any integer from 0 to $n-1$ is uniform for all values. Because the algorithm calls the `random` function n times, this algorithm is too costly. Let's see a different approach.

Second Method

Let's just change the pivot selection.

```
choose-pivot(A) {
  return random(n)
}
```

And in our main quick select algorithm

```
quick-select(A, k) {
  p = choose-pivot(A)
  .... // Rest of the original algorithm
}
```

With an equal probability for every pivot, there is at least a 50% change the pivot has position $\frac{n}{4} \leq i < \frac{3n}{4}$. This means that half the time, the runtime is $\frac{3n}{4}$, and the other time is at most n (kind of a stretch when the pivot is on the left of the array, but it suffices to say that for this example). So our expected runtime for this randomized algorithm is:

$$
\begin{aligned}
T^{(\exp)}(n) &\leq cn + \frac{1}{2}T^{(\exp)}(n) + \frac{1}{2}T^{(\exp)}\left(\frac{3n}{4}\right) \\
\frac{1}{2}T^{(\exp)}(n) &\leq cn + \frac{1}{2}T^{(\exp)}\left(\frac{3n}{4}\right) && \text{(Rearrange)} \\
T^{(\exp)}(n) &\leq 2cn + T^{(\exp)}\left(\frac{3n}{4}\right) && \text{(Multiply by 2)}
\end{aligned}
$$

The result implies that $T^{(\exp)} \in O(n)$, just like the first method. **This is generally the fastest quick-select implementation**. This implementation is less costly because we only call `random` $\log n$ times, rather than n times.

Worst-case Linear Time

In 1973, the "medians-of-five" algorithm was created for pivot selection:

1. Split thr array (of length n) into $\frac{n}{5}$ subsets. This causes each subset to have a length of 5.
2. Find the median of each subset (i.e., if the subset is $A_s = [3, 10, 17, 1, 20]$, then the median is 10). This step has a runtime of $\frac{n}{5} \times c \in O(n)$, where c is the time to calculate the median, which is constant since each array has length 5.
3. Return the median out of the $\frac{n}{5}$ calculated medians.

For instance, for the array $A = [3, 10, 17, 1, 20 \mid 1, 2, 4, 20, 10 \mid 12, 22, 13, 1, 4]$ (which is already partitioned into $\frac{n}{5} = 3$ subsets), the respective medians are 10, 4, and 12. From that, the median of those three is 10. So the algorithm returns 10.
This algorithm has a runtime of $\Theta(n)$. The proof showing this is a little beyond the scope of this course.

7.2 The QuickSort Algorithm

The quick sort algorithm is based on a sorting method developed by Hoare in 1960. On input array `A` of size `n`:

```
quick-sort1(A) {
  if (n <= 1) return
  p = choose-pivot(A)
  i = partition
  quick-sort1(A[0, 1, ..., i-1])
  quick-sort1(A[i+1, ..., n-1])
}
```

Analysing the algorithm, the worst case runtime occurs when one array always has size $n-1$ and the other has size 0:

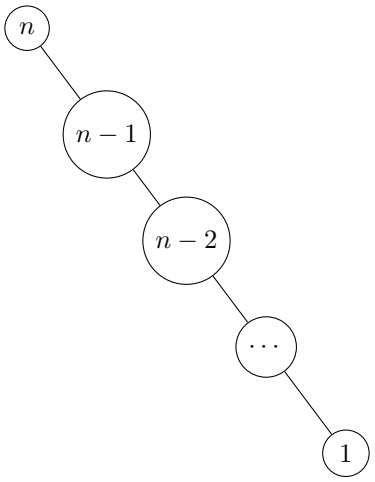
$$T^{(\text{worst})}(n) = T^{(\text{worst})}(n-1) + \Theta(n) \in \Theta(n^2)$$

The best case runtime occurs when both arrays are split evenly on each recursive call

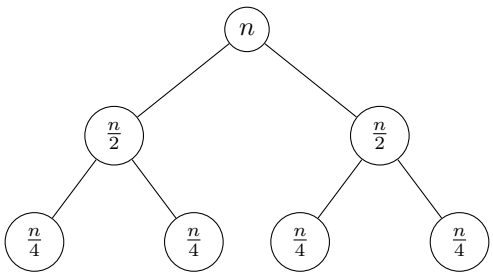
$$T^{(\text{best})}(n) = T^{(\text{best})}\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T^{(\text{best})}\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + \Theta(n) \in \Theta(n \log n)$$

We can visualize these runtime using a tree (a the height of the tree shows the how deep the recursive calls go).

The worst-case runtime splits the array into sizes $n-1$ and 0 (i.e., it has a height of n):



The best case runtime splits the arrays evenly, or off by at most 1 (i.e., has a height of $\log n$)



Average-case Analysis of Quick Sort

By our previous remark, we observe that the runtime is determined by the height H of the tree that defines the depth of the recursive calls. This means that $T(n) \in O(nH(n))$. In the average case, $H(n) \in O(\log n)$, so the average-case runtime for this algorithm is $O(n \log n)$.
QuickSort is often the most efficient algorithm in practice.