CS 240 — Lecture 6

Bartosz Antczak *Instructor: Eric Schost* *January 19, 2017*

Today's Plan

Today we'll observe the average case analysis of two algorithms, **quick-select** and **quick-sort**. Note that this will probably be the hardest algorithm analysis in this course (according to Prof. Schost).

Recall

The **selection problem** states

> *Given an array A of n numbers, and $0 \leq k < n$, find the element in position k of the sorted array*

We'll look at the two previously mentioned algorithms to solve this problem

6.1 Solving the Selection Problem

6.1.1 Partition Algorithm

Algorithm Description

The **partition** algorithm takes an input of one array `A` and an arbitrary integer `p`. It rearranges `A` such that all of the elements smaller than the element at index `p` are to the left of it and the elements larger than it are to the right.
The algorithm is structured as:

```
// A - array of size n
// p - arbitrary integer that represents our pivot
partition(A, p) {
  swap(A[0], A[p])
  i = 1
  j = n-1
  loop
    while i < n and A[i] <= A[0] do
      i = i + 1
    while j >= 1 and A[j] > A[0] do
      j = j - 1
    if (j < i) then break
    else swap(A[i], A[j])
  end loop
  swap(A[0], A[j])
  return j // the new pivot
}
```

Let's analyse this algorithm through an example. Let $A = [7, 3, 6, 9, 2, 1, 5]$ and $p = 0$. After each iteration of the big loop, we have

1. $[7, 3, 5, 9, 2, 1, 6]$, $i = 2, j = 6$
2. $[7, 3, 5, 6, 2, 1, 9]$, $i = 3, j = 6$

3. $[7, 3, 5, 9, 1, 2, 6]$, $i = 4, j = 5$
4. $[7, 3, 5, 6, 1, 2, 9]$, $i = 5, j = 5$
5. $[2, 3, 5, 6, 1, 7, 9]$, $i = 6, j = 5$ and then exit

Observe that the value 7 was our pivot. After the algorithm, 7 is now in its correct position.

6.1.2 QuickSelect Algorithm

Algorithm Description

The **quick select** algorithm returns the kth smallest element an array. The structure is as follows:

```
// A - array of size n
// k - arbitrary integer such that 0 <= k < n
quick-select1(A, k) {
  p = choosePivot(A)
  i = partition(A, p)
  if (i = k) then
    return A[i]
  else if i > k then
    return quick-select1(A[0, 1, ..., i-1], k)
  else if i < k then
    return quick-select1(A[i+1, i+2, ..., n-1], k-i-1)
}
```

Algorithm Analysis

The recurrence relation for this algorithm is:

$$T(n) = \begin{cases} T(n-1) + cn & n \geq 2 \\ d & n = 1 \end{cases}$$

(cn represents the time to partition the array, which takes linear time)
The **worst-case** analysis will have a recursive call that will always have size $n - 1$. We calculate it to be

$$T(n) = cn + c(n-1) + \cdots + c \cdot 2 + d \in \Theta(n^2)$$

The **best-case** analysis will have the first chosen pivot being the kth element. This means there are no recursive calls and the runtime is

$$\Theta(n)$$

.

Average Case Analysis

We'll set some preconditions first:

- Running the algorithm on array A, assume all entries in A are unique

- The algorithm does not depend on the actual values in A; rather, it depends on their *order*. For instance $[2, 3, 10, 5]$ and $[1, 2, 4, 3]$ are identical for the quick-select algorithm (i.e., the array is sorted up to the second last index)

To calculate the average case, we must consider the runtime on *every* permutation of A, which is $n!$, and then we divide that by the number of permutations $n!$. We call $T(A, k)$ the runtime of quick-select on input array A and integer k (k is index of the median, A is the array). Additionally, we define

$$T(n, k) = \frac{1}{n!} \sum_{i=0}^{n!} T(A, k)$$

(Note, it's written a little differently in the slides, but this definition is easier to understand)
We want to show that $T(n, k) \in O(n)$: First, for simplicity, let $T(n) = \max T(n, k)$, $0 \leq k < n$.
Claim: if a function F satisfies a "sloppy" recurrence then $F(n) \in O(n)$.
"**Proof**": for $\alpha = 1/2$ and n being a power of 2, then

$$\begin{aligned} F(n) &\leq kn + F\left(\frac{1}{2}n\right) \\ &\leq kn + k\frac{n}{2} + F\left(\frac{1}{4}n\right) \\ &\leq kn + kn/2 + kn/4 + \cdots + d \\ &\leq kn(1 + 1/2 + 1/4 + 1/8 \cdots) + d \\ &\implies F(n) \in O(n) \end{aligned}$$

We will assume that after the partition, the elements less than the pivot are in the same order as before the partition, and the same being for the elements greater than the pivot. For instance, consider $A = [2, 5, 3, 1, 6, 4]$. If we let the pivot p equal 2 ($A[p] = 3$), then we expect the array after the partition to look like this:

$$A = [2, 1, 3, 5, 6, 4]$$

Where 2 still has an index value smaller than 1; and 4's index number $>$ 6's index number $>$ 5's index number.

(From this moment on, I was pretty lost during the lecture. Prof. Schost wrote just a ton of equations, please refer to his notes he posted on the CS 240 website for the conclusion of this lecture).