Brief Intro to Randomized Algorithms

CS130A Coakley

Sometimes Randomization is Bad

```
void BogoSort( int a[] ) {
        while (! sorted(a) ) {
                randomly_shuffle(a);
} }
```

random_shuffle

```
// O(n) shuffle

void random_shuffle(int a[]) {

        for (int i = 0; i < a.length - 1; ++i) {

                int j = uniform_random_number(i, a.length - 1);

                swap(a, i, j);

}}
```

Analysis of BogoSort

sorted(a) can be performed in O(n) time where n is the number of elements in a

randomly_shuffle(a) can be performed in O(n) provided there exists an O(1) random number generator.

If the array comes pre-sorted, we return in O(n) time. Best case.

We always take O(n) space.

We *can* randomly guess the sort in O(n) time. Best case.

How likely is that?

Random Selection

The odds of getting a[0] right is 1/n

The odds of now getting a[1] right is 1/(n-1)

...

The odds of getting a sorted array are 1/n!

So we expect to do O(n!) shuffles, and each shuffle is O(n)

O((n+1)!) expected runtime.

When is random good?

In BogoSort, we were penalized for randomness because we had a low probability of guessing right.

What happens if most answers are good, only a few answers are bad, and we have a low probability of guessing wrong?

What if you have to guess wrong each step of a recursive algorithm to be very bad?

Quicksort

1. Choose an element called the *pivot*.
2. Partition: reorder the array so that elements less than the pivot are before, elements greater are after.
3. Recurse to 1 on the left and right sub-arrays.

This can be done in-place on the array. Otherwise, the return step is to return sorted_left + pivot + sorted_right.

Partition

The idea is to use two indices to loop from both ends of the array.

If an item is in the correct partition, skip it.

When the two indices cross, return.

Otherwise, each time they both stop on an element (so both indexes point to something in the wrong partition), swap the cells.

Partition

```
for(;;) {
        while( a[ ++i ] < pivot ) {}
        while( pivot < a[ --j ] ) {}
        if ( i < j ) std::swap( a[i], a[j] );
        else break;
}
```

Correctness of Quicksort

This is a divide and conquer algorithm. At each step, the pivot is placed in its final spot in the array. Each recursive step also has 1 or 2 subproblems that are smaller than the original problem.

The algorithm terminates.

For any element, some subproblem chose it as the pivot and placed it in the correct location (base case can just be a single element).

Efficiency of Quicksort

The partition step is O(n). Both indices advance at least one position each outer loop, even when encountering equal values.

We choose the pivot in O(1).

So we break into 2 subproblems (minus 1) and have a cN scan for the partition + pivot

$T(N) = T(i) + T(N - i - 1) + cN$

But what is i?

Subproblem Size (Best)

$T(N) = T(i) + T(N - i - 1) + cN$

If we choose the median element (just going to ignore the extra - 1, so this is an upper bound):

$T(N) = T(N/2) + T(N/2) + cN$

Mergesort's recurrence!

$O(n \log n)$

Subproblem Size (Worst)

$T(N) = T(i) + T(N - i - 1) + cN$

If we choose the greatest element (or least). The partition is completely lopsided. $T(0) = 1$ for one of the subproblems

$T(N) = T(N-1) + cN$

$O(N^2)$

Pivot Selection

Calculating the median isn't really an option if we want an O(1) pivot choice.

We could just choose the first element in the array as the pivot, but this gives us worst-case performance on sorted (or almost-sorted) arrays.

The story is the same for choosing the last element of an array.

What if we just choose a pivot at random?

Analysis 1: Average Case Analysis

The expected case for a random pivot is actually the same analysis as the average case analysis. Each pivot is equally likely:

$T(N) = T(i) + T(N - i - 1) + cN$

$T(N) = cN + (1/N) * \Sigma\ (T(i) + T(N - i - 1))$ (sum is from i=0 to N-1)

$T(N) \approx 2N \ln N \approx 1.39N \log_2 N$

$O(n \log n)$

Analysis 2: Percentile Analysis

If we choose a pivot in the middle 50% percentile (i.e. somewhat close to the median), we require $\log_{4/3} n$ recursion steps. But we only hit that 50% of the time, so the expected value is twice that (you expect to need only 2n coin flips to hit n heads).

$2 * \log_{4/3} n$ recursion steps is still O(n log n)

What We Ignore

Your book goes into more analysis of quicksort using a deterministic pivot selection criteria called the median-of-three (left, right, middle values).

The book also covers choices on partition implementations for how to handle multiple equal values. Imagine sorting an array of a billion 2-byte words... at some point your subproblems are mostly duplicates.

Other Algorithms

Find a convex hull for a set of points in 2-space in O(n log n) expected time.

Quickhull: https://en.wikipedia.org/wiki/Quickhull

Find the k smallest elements in a set in O(n) expected time:

Quickselect: https://en.wikipedia.org/wiki/Quickselect

Quickselect

Quickselect is sort of fun because it is like half of quicksort.

1. Select a pivot
2. Partition
3. Recurse on the left subproblem if pivot ends < k, right subproblem if pivot > k, and stop if pivot = k.