CSE 103: Probability and statistics **Fall 2010**

Topic 4 — Randomized algorithms, I

4.1 Finding percentiles

4.1.1 The mean as a summary statistic

Suppose UCSD tracks this year's graduating class in computer science and finds out everyone's salary ten years down the line. What might these numbers look like? Well, if there are (say) 100 students, the spread might be roughly:

- A few people with zero salary (unemployed)
- A few grad students with salary around 20K
- A few part-timers with salary around 50K
- A whole lot of software engineers with salaries between 100K and 200K

The *mean* salary would then be something like 100K, which UCSD could report with pride in its brochure for prospective students.

Now suppose that one student managed to strike it rich and become a billionaire. Accordingly, take the spread of salaries above and convert one of the 200K salaries to 1000000K. What would be the new mean salary? Answer: at least 10 million dollars! (Do you see why?) If UCSD were to report this number, nobody would take it seriously, despite its being perfectly truthful.

The problem is that the mean is extremely sensitive to *outliers* – it is very easily thrown off by a single number that is unusually small or unusually large. In many circumstances, therefore, the preferred summary statistic is the *median*, or 50th percentile. For the salary data, for instance, the median would remain unchanged (at around 100K) even if a few people were to become billionaires, or if a few more people were to lose their jobs.

We're also interested in other percentiles – the 25th, 75th, and so on. How can we compute these for a *very large* data set (for instance, a data set giving the salary of everyone in the US)?

4.1.2 Selection

Here the problem, formally.

> SELECTION
>
> *Input:* An array $S[1 \cdots n]$ of n numbers; an integer k between 1 and n
>
> *Output:* The kth smallest number in the array.

The median corresponds to $k = \lceil n/2 \rceil$, while $k = 1$ retrieves the very smallest element. The pth percentile ($0 \leq p \leq 100$) can be obtained with $k = \lceil pn/100 \rceil$.

The most natural algorithm for this problem is:

> Sort S and return $S[k]$

The running time here is dominated by that of sorting, which is $O(n \log n)$. This is pretty good, but we'd like something faster since we often need to compute percentiles of enormous data sets.

4.1.3 A randomized algorithm

Here's a randomized (and recursive) procedure for selection.

For any number v, imagine splitting array S into three categories: elements smaller than v, those equal to v (there might be duplicates), and those greater than v. Call these S_L, S_v, and S_R respectively. For instance, if the array

S :

2	36	5	21	8	13	11	20	5	4	1

is split on $v = 5$, the three subarrays generated are

S_L :

2	4	1

S_v :

5	5

S_R :

36	21	8	13	11	20

The search can instantly be narrowed down to one of these sublists. If we want, say, the *eighth*-smallest element of S, we know it must be the *third*-smallest element of S_R since $|S_L| + |S_v| = 5$. That is, $\text{SELECTION}(S, 8) = \text{SELECTION}(S_R, 3)$. More generally, by checking k against the sizes of the subarrays, we can quickly determine which of them holds the desired element:

$$\text{SELECTION}(S, k) = \begin{cases} \text{SELECTION}(S_L, k) & \text{if } k \leq |S_L| \\ v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\ \text{SELECTION}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

The three sublists S_L, S_v, S_R can be computed from S in *linear* time, scanning left-to-right. We then recurse on the appropriate sublist.

The effect of the split is thus to shrink the number of elements from $|S|$ to at most $\max\{|S_L|, |S_R|\}$. How much of an improvement is this, and what is the final running time?

4.1.4 Running time analysis

By how much does a single split reduce the size of the array? Well, this depends on the choice of v.

1. *Worst case.* When v is either the smallest or largest element in the array, the array shrinks by just one element. If we keep getting unlucky in this way, then

   (time to process array of n elements) = (time to split) + (time to process array of $n - 1$ elements).

   Since the time to split is linear, $O(n)$, this works out to a total running time of $n+(n-1)+(n-2)+\cdots = O(n^2)$, which is really bad.

   Fortunately, this case is unlikely to occur. The probability of consistently picking an element v which is the smallest or largest, is miniscule:

   $$\frac{2}{n} \cdot \frac{2}{n-1} \cdot \frac{2}{n-2} \cdots \frac{2}{3} \approx \frac{2^n}{n!}$$

   (do you see where this expression comes from?).

2. *Best case.* The best possible case is that v just happens to be the element we are looking for, that is, the kth smallest element in the array. In this case, the running time is $O(n)$, the time for a single split.

   This case is also unlikely, but it is certainly a lot more likely than the worst case. In fact, the probability of it occurring is at least $1/n$. (Why? When might it be more than $1/n$?)

Neither the best case nor worst case is a particularly good way to quantify the running time of this algorithm. A more sensible measure is the *expected* running time. Let $T(n)$ be (an upper bound on) the expected time to process an array of n (or fewer) elements. We will now derive an expression for it.

Call a split *lucky* if the resulting S_L and S_R both have size less than $3n/4$; call it *unlucky* otherwise. A split is lucky if v lies somewhere between the 25th and 75th percentile of S, which happens with probability exactly $1/2$.

Therefore,

$$\begin{aligned} T(n) &\leq (\text{time to split}) + (\text{expected time taken to process the larger of } S_L, S_R) \\ &\leq n + \Pr(\text{lucky split})(\text{time for array of size} \leq 3n/4) + \Pr(\text{unlucky split})(\text{time for array of size } n) \\ &\leq n + \frac{1}{2}T(3n/4) + \frac{1}{2}T(n) \end{aligned}$$

Rearranging, we get $T(n) \leq 2n + T(3n/4)$. Expanding it out, and using the formula for the sum of a geometric series, we get

$$T(n) \;\leq\; 2n + 2 \cdot \frac{3n}{4} + 2 \cdot \frac{9n}{16} + \cdots \;\leq\; 2n\left(1 + \frac{3}{4} + \frac{9}{16} + \cdots\right) \;=\; 8n.$$

Our randomized algorithm for selection has an expected *linear* running time!

4.2 A randomized sorting algorithm

There's a very popular algorithm for sorting that operates on similar principles. It's called *quicksort*:

- Given an array $S[1 \cdots n]$, pick an element v from it at random.
- Split S into three pieces:

S_L	elements less than v
S_v	elements equal to v
S_R	elements greater than v

- Now return quicksort$(S_L) \circ S_v \circ$ quicksort(S_R), where $\circ$ denotes concatenation.

Letting $T(n)$ be the expected running time on an array of n elements, we have

$$\begin{aligned} T(n) &= (\text{time to split}) + (\text{expected time to sort } S_L \text{ and } S_R) \\ &= n + \sum_{i=1}^{n} \Pr(v \text{ is the } i\text{th smallest element in } S)(T(i-1) + T(n-i)) \\ &= n + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)). \end{aligned}$$

This is tricky to solve, but works out to $T(n) = O(n \log n)$.

4.3 Sorting in expected linear time

Suppose we want to sort an array of numbers $S[1 \cdots n]$ that we expect to be distributed uniformly in some range $[\min, \max]$. Here's a *bucket sort* approach:

- Divide $[\min, \max]$ into n equal-sized intervals. These are the *buckets* $B_1, B_2, \ldots, B_n$.
- Now scan array S from left to right, putting each element $S[i]$ in its appropriate bucket.
- Return $\text{sort}(B_1) \circ \text{sort}(B_2) \circ \cdots \circ \text{sort}(B_n)$, where "sort" is a standard sorting algorithm (say mergesort).

Notice that there is no randomization in the algorithm. However, we can talk about the expected running time if the elements of S are generated from a uniform distribution over $[\min, \max]$. In that case, each element is equally likely to fall into any of the buckets B_i.

Let N_i be the number of array elements that fall into B_i. Assuming we use a standard sorting procedure for each bucket, we get a total running time of

$$T = N_1 \log N_1 + N_2 \log N_2 + \cdots + N_n \log N_n \ \leq \ N_1^2 + N_2^2 + \cdots + N_n^2.$$

What is $\mathbb{E}(N_i^2)$? The easiest way to compute this is to write N_i as a sum:

$$N_i = X_1 + X_2 + \cdots + X_n$$

where X_j is 1 if the array element $S[j]$ falls into bin i, and 0 otherwise. Notice that $X_j^2 = X_j$, and that X_j is independent of $X_{j'}$ whenever $j \neq j'$. Therefore,

$$\begin{aligned} \mathbb{E}(X_j) &= \frac{1}{n} \\ \mathbb{E}(X_j^2) &= \frac{1}{n} \\ \mathbb{E}(X_j X_j') &= \mathbb{E}(X_j)\mathbb{E}(X_{j'}) = \frac{1}{n^2} \quad \text{if } j \neq j' \end{aligned}$$

By linearity of expectation, we then have

$$\begin{aligned} \mathbb{E}(N_i^2) &= \mathbb{E}\left((X_1 + \cdots + X_n)^2\right) \\ &= \mathbb{E}\left(\sum_j X_j^2 + \sum_{j \neq j'} X_j X_{j'}\right) \\ &= \sum_j \mathbb{E}(X_j^2) + \sum_{j \neq j'} \mathbb{E}(X_j X_{j'}) \\ &= n \cdot \frac{1}{n} + n(n-1)\frac{1}{n^2} \ \leq \ 2. \end{aligned}$$

So the expected running time of the sorting algorithm, once again invoking linearity, is

$$\mathbb{E}(T) \ \leq \ \mathbb{E}(N_1^2) + \mathbb{E}(N_2^2) + \cdots + \mathbb{E}(N_n^2) \ \leq \ 2n.$$

It is linear!