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

# Topic 4 — Randomized algorithms

## 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 $k$th smallest number in the array.

The median corresponds to $k = \lceil n/2 \rceil$, while $k = 1$ retrieves the very smallest element. The $p$th 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 $k$th 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 Two types of randomized algorithms

The two algorithms we've seen – for finding percentiles and for sorting – are both guaranteed to return the correct answer. But if you run them multiple times on the same input, their running times will fluctuate, even though the answer will be the same every time. Therefore we are interested in their *expected* running time. We call these *Las Vegas algorithms*.

There's another type of algorithm – called a *Monte Carlo algorithm* – that always has the same running time on any given input, but is not guaranteed to return the correct answer. It merely guarantees that it has some probability $p > 0$ of being correct. Therefore, if you run it multiple times on an input, you'll get many different answers, of which roughly a $p$ fraction will be correct. In many cases, it is possible to look through these answers and figure out which one(s) are right.

In a Monte Carlo algorithm, how much does the probability of success increase if you run it multiple times, say $k$ times?

$$\Pr(\text{wrong every time}) \;=\; (1-p)^k \;\leq\; e^{-pk}.$$

To make this less than some $\delta$, it is enough to run the algorithm $(1/p)\log(1/\delta)$ times. For instance, to make the failure probability less than 1 in a million, just run the algorithm $20/p$ times (a million is roughly $2^{20}$).

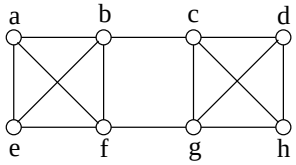
## 4.4 Karger's minimum cut algorithm

### 4.4.1 Clustering via graph cuts

Suppose a mail order company has the resources to prepare two different versions of its catalog, and it wishes to target each version towards a particular sector of its customer base. The data it has is a list of its regular customers, along with their purchase histories. How should this set of customers be partitioned into two coherent groups?

One way to do this is to create a graph with a node for each of the regular customers, and an edge between any two customers whose purchase patterns are similar. The goal is then to divide the nodes into two pieces which have very few edges between them.

More formally, the *minimum cut* of an undirected graph $G = (V, E)$ is a partition of the nodes into two groups $V_1$ and $V_2$ (that is, $V = V_1 \cup V_2$ and, $V_1 \cap V_2 = \emptyset$), so that the number of edges between $V_1$ and $V_2$ is minimized. In the graph below, for instance, the minimum cut has size two and partitions the nodes into $V_1 = \{a, b, e, f\}$ and $V_2 = \{c, d, g, h\}$.




### 4.4.2 Karger's algorithm

Here's a randomized algorithm for finding the minimum cut:

- Repeat until just two nodes remain:
  - Pick an edge of $G$ at random and collapse its two endpoints into a single node
- For the two remaining nodes $u_1$ and $u_2$, set $V_1 = \{\text{nodes that went into } u_1\}$ and $V_2 = \{\text{nodes in } u_2\}$

An example is shown in Figure 4.1. Notice how some nodes end up having multiple edges between them.

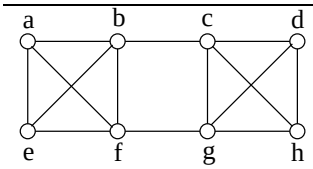
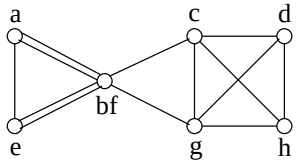
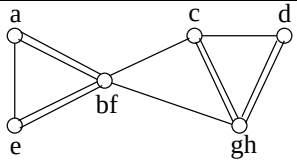
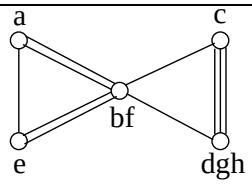
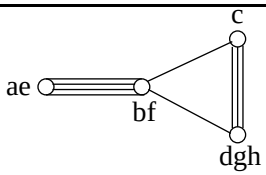
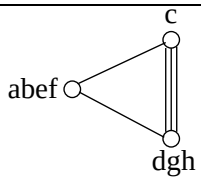
| | |
|---|---|
|  | 14 edges to choose from<br>Pick $b - f$ (probability 1/14) |
|  | 13 edges to choose from<br>Pick $g - h$ (probability 1/13) |
|  | 12 edges to choose from<br>Pick $d - gh$ (probability 1/6) |
|  | 10 edges to choose from<br>Pick $a - e$ (probability 1/10) |
|  | 9 edges to choose from<br>Pick $ab - ef$ (probability 4/9) |
|  | 5 edges to choose from<br>Pick $c - dgh$ (probability 3/5) |
|  | Done: just two nodes remain |

**Figure 4.1.** Karger's algorithm at work.

### 4.4.3 Analysis

Karger's algorithm returns the minimum cut with a certain probability. To analyze it, let's go through a succession of key facts.

**Fact 1.** *If degree($u$) denotes the number of edges touching node $u$, then*

$$\sum_{u \in V} \textit{degree}(u) = 2|E|.$$

To see this, imagine the following experiment: for each node, list all the edges touching it. The number of edges in this list is exactly the left-hand sum. But each edge appears exactly twice in it, once for each endpoint.

**Fact 2.** *If there are $n$ nodes, then the average degree of a node is $2|E|/n$.*

This is a straightforward calculation: when you pick a node $X$ at random,

$$\mathbb{E}[\text{degree}(X)] \;=\; \sum_{u \in V} \Pr(X = u)\text{degree}(u) \;=\; \frac{1}{n}\sum_{u} \text{degree}(u) \;=\; \frac{2|E|}{n}$$

where the last step uses the first Fact.

**Fact 3.** *The size of the minimum cut is at most $2|E|/n$.*

Consider the partition of $V$ into two pieces, one containing a single node $u$, and the other containing the remaining $n-1$ nodes. The size of this cut is degree($u$). Since this is a valid cut, the minimum cut cannot be bigger than this. In other words, for all nodes $u$,

$$(\text{size of minimum cut}) \leq \text{degree}(u).$$

This means that the size of the minimum cut is also $\leq$ the average degree, which we've seen is $2|E|/n$.

**Fact 4.** *If an edge is picked at random, the probability that it lies across the minimum cut is at most $2/n$.*

This is because there are $|E|$ edges to choose from, and at most $2|E|/n$ of them are in the minimum cut.

Now we have all the information we need to analyze Karger's algorithm. It returns the right answer *as long as it never picks an edge across the minimum cut.* If it always picks a non-cut edge, then this edge will connect two nodes on the same side of the cut, and so it is okay to collapse them together.

Each time an edge is collapsed, the number of nodes decreases by 1. Therefore,

$$\begin{aligned}
\Pr(\text{final cut is the minimum cut}) &= \Pr(\text{first selected edge is not in mincut}) \times \\
&\quad \Pr(\text{second selected edge is not in mincut}) \times \cdots \\
&\geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) \\
&= \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdots\frac{2}{4}\cdot\frac{1}{3} \\
&= \frac{2}{n(n-1)}.
\end{aligned}$$

The last equation comes from noticing that almost every numerator cancels with the denominator two fractions down the line.

Karger's algorithm succeeds with probabililty $p \geq 2/n^2$. Therefore, it should be run at least $n^2/2$ times, after which the smallest cut found should be chosen.

# 4.5 Hashing

## 4.5.1 The Google problem

When you give a search phrase to Google (or any other search engine), you *immediately* get back a list of documents containing that phrase. But there are billions of documents on the web – how does Google look through all of them so quickly? The answer is, it uses *hashing*.

## 4.5.2 The hashing framework

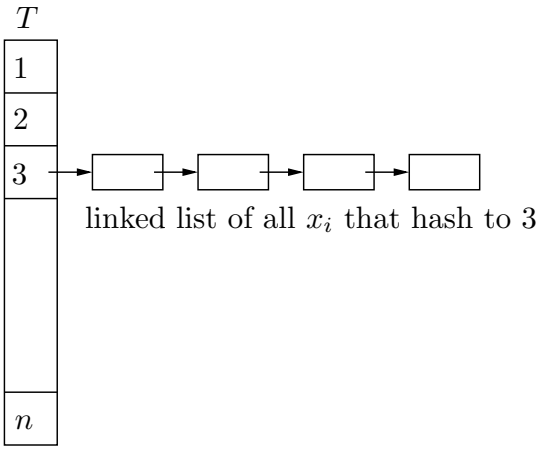
Suppose you have a large collection of items $x_1, \ldots, x_n$ that you want to store (for instance, all the documents on the web), where these items are drawn from some set $\mathcal{U}$ (for instance, the set of all conceivable documents). The requirements are:

1. The total storage space used should be $O(n)$.
2. Given a query $q \in \mathcal{U}$, it should be possible to *very rapidly* determine whether $q$ is one of the stored items $x_i$.

## 4.5.3 A simple solution using randomization

1. Pick a completely random function $h : \mathcal{U} \to \{1, 2, \ldots, n\}$.
   This is the *hash function*.
2. Create a table $T$ of size $n$, each of whose entries is a pointer to a linked list, initialized to null.
3. Store each $x_i$ in the linked list at $T[h(x_i)]$.
   We say $x_i$ *hashes to* location $h(x_i)$.
4. Given a query $q$, look through the linked list at $T[h(q)]$ to see if it's there.

Here's a picture of the data structure.




The storage used is $O(n)$. What about the query time?

### 4.5.4 Average query time

Suppose query $q$ is picked at random, so that it is equally likely to hash to any of the locations $1, 2, \ldots, n$. What is the expected query time?

$$\begin{aligned}\text{Expected query time} &= \sum_{i=1}^{n} \Pr(q \text{ hashes to location } i) \cdot (\text{length of list at } T[i]) \\ &= \frac{1}{n}\sum_{i} (\text{length of list at } T[i]) \\ &= \frac{1}{n} \cdot n \;=\; 1\end{aligned}$$

So the average query time is constant!

### 4.5.5 Worst case query time, and a balls-in-bins problem

What is the worst case query time; that is, what is the length of the longest linked list in $T$? Equivalently, when you throw $n$ balls in $n$ bins, what is the size of the largest bin? We'll see that with very high probability, no bin gets $\geq \log n$ balls.

For any bin $i$, let $E_i$ be the event that it gets $\geq \log n$ balls.

$$\Pr(E_i) \;\leq\; \binom{n}{\log n}\left(\frac{1}{n}\right)^{\log n}.$$

(Do you see why?) It turns out, using all sorts of calculations, that this is at most $1/n^2$.

Therefore,

$$\Pr(\text{some bin gets} \geq \log n \text{ balls}) \;=\; \Pr(E_1 \cup E_2 \cup \cdots \cup E_n) \;\leq\; \Pr(E_1) + \cdots + \Pr(E_n) \;\leq\; \frac{1}{n}.$$

For instance, if you throw a million balls into a million bins, then the chance that there is a bin with $\geq 20$ balls is at most 1 in a million.

Getting back to hashing, this means that the worst case query time is (with high probability) $O(\log n)$.

### 4.5.6 The power of two choices

Here's a variant on the balls and bins setup. As usual, you have before you a row of $n$ bins, along with a collection of $n$ identical balls. But now, when throwing each ball, *you pick two bins at random and you put the ball in whichever of them is less full.*

It turns out, using an analysis that is too complicated to get into here, that under this small change, the maximum bin size will be just $O(\log \log n)$ instead of $O(\log n)$.

This inspires an alternative hashing scheme:

1. Pick *two* completely random functions $h_1, h_2 : \mathcal{U} \to \{1, 2, \ldots, n\}$.
2. Create a table $T$ of size $n$, each of whose entries is a pointer to a linked list, initialized to null.
3. For each $x_i$, store it in either the linked list at $T[h_1(x_i)]$ or $T[h_2(x_i)]$, whichever is shorter.
4. Given a query $q$, look through *both* the linked list at $T[h_1(q)]$ and at $T[h_2(q)]$ to see if it's there.

The storage requirement is still $O(n)$, the average query time is still $O(1)$, but now the worst case query time drops to $O(\log \log n)$.

## 4.6 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!