CSE 347 Lecture 17

Average-case Analysis

1. Random input model
2. Naive search trees
3. Naive hashing

NP-complete problems are quite common. But, when we are looking to solve a problem and it turns out to be NP-complete, giving up is usually not an option. Instead, we take it as a sign that we need to lower our expectations for our solutions. Our two criteria for algorithms were ① polynomial worst-case running time and ② optimal (correct) results. It's possible to relax either of these criteria. Today, we'll introduce average-case analysis, which is a commonly proposed option. Actually, it isn't very easy to apply average-case analysis to solve NP-complete problems in practice, but the ideas will be useful to help us design simpler and faster randomized algorithms. (We'll relax ② and develop approximation algorithms later on...)

In average-case analysis, we assume that we have a model of the distribution of inputs that our algorithm will be invoked on. Let's consider a simple dictionary problem. Our inputs are integers (say,

in some large range 0-N) and we wish to support two operations: insert, which adds a new integer x to the set S of integers we have seen so far, and lookup, which takes an integer x and returns whether or not $x \in S$. Now we wish to propose a probability distribution over the integers 0-(N-1) for the inputs for the insert and lookup operations. What distribution should we use? Really, it depends on the application domain, and we have no idea. This is the main problem that arises in average-case analysis. One natural choice that is sometimes appropriate is to assume that the inputs are independently chosen from the uniform distribution: each i in the range 0-(N-1) is chosen with probability $\frac{1}{N}$. Nearly equivalently, as long as the number of inserts is small relative to N, we might assume that the numbers are all distinct and simply arrive in a random order. How small is small enough for this to be valid?

Let's calculate: assuming we've inserted n-1 distinct elements so far, the probability that the nth is another distinct element is $\frac{N-(n-1)}{N} = 1 - \frac{n-1}{N}$. Since each draw is another independent event, we find that the probability that all n elements are distinct is given by multiplying these together: $1 \cdot (1-\frac{1}{N}) \cdot (1-\frac{2}{N}) \cdots (1-\frac{n-1}{N})$. How large is this? More concretely, how large can n be before it is less

than $1-\delta$ for small δ (say $\delta = 1\%$)? We can estimate this by taking logs: $\ln(1-\delta) = \ln\left[1 \cdot (1-\frac{1}{N}) \cdots (1-\frac{n-1}{N})\right] = \sum_{i=1}^{n-1} \ln\left(1-\frac{1}{N}\right) \approx \sum_{i=1}^{n-1} -\frac{1}{N}$ so, roughly when $\frac{n(n-1)}{2}$ equals $N \cdot \ln\frac{1}{1-\delta}$, i.e, we need $n \sim \sqrt{N}$ (we stress that for such small $\frac{n}{N}$, the Taylor approximation $\ln(1-x) = -x + O(x^2)$ has error $O((\frac{\sqrt{N}}{N})^2) = O(\frac{1}{N})$ per term, which over $\sqrt{N}$ terms is $O(\frac{1}{\sqrt{N}})$ total, which only needs to be small compared to $\ln(\frac{1}{1-\delta}) \approx \frac{1}{100}$...)

Now, how would you solve the dictionary problem? Make a big table? Make a linked list? Binary search tree? Hashing? (we'll come back to hashing soon enough...) Let's consider a naive search tree, like your first implementation in 131.

```
insert(x, T)
| if T is NULL return a node containing x
| else if the root of T contains an element y > x,
|      | put T's left subtree equal to insert(x, T's left subtree)
|      | return T
| else put T's right subtree equal to insert(x, T's right subtree)
|    | return T
```

```
lookup(x, T)
| if T is NULL, return 'false'
| else if the root contains x, return 'true'
| else if the root contains y>x, return lookup(x, T's left child)
| else return lookup(x, T's right child)
```

You have seen previously that these algorithms maintain a data structure T that solves our dictionary pro-

blem. Of course. What's the worst-case running time for this naive implementation? $O(n)$ after n inserts, e.g., if the inputs are sorted — we construct a tree

$1 \to 2 \to 3 \to \cdots \to n$ so inserting or looking up $(n+1)$ takes $n+1$ recursive calls at $O(1)$ time each.

But now, let's see what the time is when the inputs arrive in a random order. Let's first consider the time to perform a lookup operation when the query x is in the tree:

Theorem The average depth of a lookup over x in the tree is $(1+\frac{1}{n})(2H_{n+1}-3)+\frac{1}{n} \sim O(\log n)$, where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is called the n^{th} harmonic number. (We're using $\sum_{i=1}^{n} \frac{1}{i} \le 1 + \int_1^n \frac{1}{x}\,dx$...)

Proof: First, observe that the number of recursive calls in a lookup is the same as the number of recursive calls made when the element was inserted, so this is actually equal to the average number of recursive calls made when inserting the elements (per element). So, we'll actually calculate the total number of calls, and divide by n.

The # of calls to insert n elements is given by the following recurrence: $C(n) = n + \frac{1}{n}\sum_{j=1}^{n} (C(j-1)+C(n-j))$, since all n elements touch the root, which contains the j^{th} smallest element out of n with probability $\frac{1}{n}$ — in which case, we pay the cost of inserting the $j-1$ subsequent smaller elements into the left subtree, $C(j-1)$, and the $n-j$ larger elements into

the right subtree, $C(n-j)$.

We can easily solve the recurrence by computing

$$nC(n) - (n-1)C(n-1) = n^2 + \sum_{j=1}^{n}(C(j-1)+C(n-j)) - (n-1)^2 - \sum_{j=1}^{n-1}(C(j-1)+C(n-1-j))$$

$$= 2n - 1 + 2C(n-1)$$ and dividing by $n(n+1)$.

$$\frac{C(n)}{n+1} = \frac{2}{n+1} - \underbrace{\frac{1}{n(n+1)}}_{\frac{1}{n+1}-\frac{1}{n}} + \frac{C(n-1)}{n}$$ so iterating, we find

$$\frac{C(n)}{n+1} = 2\sum_{j=1}^{n}\frac{1}{j+1} + \sum_{j=1}^{n}\left[\frac{1}{j+1} - \frac{1}{j}\right] + 0, \text{ i.e.,}$$

($j=1$ in $\frac{1}{j+1}$)

$$\frac{C(n)}{n+1} = 2H_{n+1} - 2 + \frac{1}{n+1} - 1 \text{ but, we want } \frac{C(n)}{n} = \left(1+\frac{1}{n}\right)\frac{C(n)}{n+1}$$

which is indeed $\left(1+\frac{1}{n}\right)(2H_{n+1}-3)+\frac{1}{n} \sim O(\log n)$ ▨

What about the cost of a lookup when x <u>isn't</u> in the tree? Again, we assume that x appears in a random position in the ordered elements, i.e., that it reaches a random null child in the tree. Thus.

<u>Theorem</u> The average number of recursive calls for lookup <u>of x</u> not in the tree is $2H_{n+1} - 1 \sim O(\log n)$.

<u>Proof</u>: Again, the <u>total</u> number of calls summed over all n+1 null children satisfies a recurrence (what?) $S(n+1) = n+1+\frac{1}{n}\sum_{j=1}^{n}(S(j)+S(n+1-j))$ since all n+1 null children make a call at the root, which appears at a random position <u>in between</u> the null children. We similarly solve it: $nS(n+1) - (n-1)S(n) = 2n + 2S(n)$ so $\frac{S(n+1)}{n+1} = \frac{2}{n+1} + \frac{S(n)}{n}$, so iterating thus, $\frac{S(n+1)}{n+1} = \sum_{j=1}^{n}\frac{2}{j+1} + \underbrace{\frac{S(1)}{1}}_{=1=\frac{2}{1}-1} = 2H_{n+1} - 1 \sim O(\log n)$ ▨

Actually, now observe that as long as the elements are all distinct, an insert uses the same number of recursive calls as a lookup of an element not in the tree — thus,

<u>Corollary</u> The average number of recursive calls for insert

when the tree contains n elements is $O(\log n)$.
So this is great — we don't have to do anything clever, and lookup and insert operations only take $O(\log n)$ time!

Hashing

An even simpler solution is given by hashing. Let's suppose for simplicity that we use a hash table of size m that divides N, our range (e.g., both are powers of 2). Then, if we use $h(x) = x \bmod m$ as our hash function, observe that for every index i in 0-(m-1), exactly $\frac{N}{m}$ x in 0-(N-1) hash to i, i.e., $h(x)=i$ w.p. $\frac{N}{m}\cdot\frac{1}{N}=\frac{1}{m}$. So, each $h(x)$ is an independent, uniformly distributed element in the range 0-(m-1).
We actually already calculated the probability that we get no collisions when we have n elements stored —

Theorem For any $\delta > 0$, if $\frac{1}{m} = O\left(\log^2\left(\frac{1}{1-\delta}\right)\right)$, then for $n = O\left(\sqrt{m \log\frac{1}{1-\delta}}\right)$, there are no collisions with probability $1-\delta$ after inserting n random integers from 0-(N-1) in a hash table of size m using $h(x) = x \bmod m$.

So, we can create a table of size $m = O(n^2)$, and use a simple linked list to store the elements with $h(x)=i$ in index i, and since with high probability every list contains at most one element, insert and lookup take $O(1)$ time (w.p. $1-\delta$).

There is, of course, more to say about hashing — e.g., how does performance scale once we insert more than $\sqrt{m}$ elements? But the more serious issue is that all of this

depends on the assumption that the inputs are independent, uniformly distributed integers, which is a big assumption.