1 So What If You Can’t Solve Your Problem?

What do you do if you really want to solve a problem, but it appears to be too hard?

- Let’s think about what NP-completeness means.
- If problem \( P \) is NP-complete, or even if we personally cannot find an “efficient algorithm,” then we cannot...
- ... find an algorithm that \textit{correctly} solves \textit{arbitrary} instances of \( P \) in \textit{worst-case} time \textit{polynomial} in the input size.
- Each of the four qualifiers above could be relaxed to make the problem easier.
- \textit{correctly} – find an efficient algorithm that occasionally (but rarely) gives the wrong answer. [e.g. primality testing]
- \textit{arbitrary} – find an efficient algorithm that solves only a restricted version of the problem. [e.g. 2SAT]
- \textit{worst-case} – find an algorithm that is efficient “most of the time” but can be slow in the worst-case. [e.g. simplex algorithm for LP]
- \textit{polynomial} – find an algorithm whose running time is super-polynomial but still acceptable for small inputs. [e.g. integer factoring]
- Today, we’ll start to look at the third option – alternatives to worst-case execution time for measuring efficiency.

2 Average-case Analysis

Instead of worst-case efficiency, let’s consider what it means for an algorithm to be efficient “on average.”

- What are we averaging over?
- For a given input size, there are many different inputs possible.
- Some inputs (or input properties) may be “typical” – they occur more often, i.e. with higher probability.
• For example, if our input consists of the integers 1..n, and their order in the input is arbitrary, then we might expect that they will typically not be sorted.

• More formally, if we fix an input size n, assume that each possible input x of this size has an associated probability Pr(x).

• If the running time of our algorithm on input x is T(x), then its average running time T(n) is given by

\[ T(n) = \sum_{x} T(x) Pr(x). \]

Where does the distribution Pr(x) come from?

• It depends on your application! If you tell me what you expect, I will tell you how fast your algorithm is.

• But your expectations may be wrong – especially, but not exclusively, if someone is trying to “break” your algorithm.

• (But even wrong expectations can sometimes give you a good ballpark estimate of real-world performance.)

• Example: in biological sequence analysis, the performance of string matching algorithms on DNA is usually estimated assuming that a DNA sequence contains equal numbers of the four bases A, C, G, and T, and that each base occurs independently of its neighbors.

• This distribution is patently false for real DNA, but the performance estimates that it gives are order-of-magnitude correct in practice.

3 Naive Trees – an Average-Case Analysis

Let’s do a complete example of analysis around binary trees.

• Suppose you need to keep track of numbers in the range 0..N – 1.

• As each number shows up, you must remember that you saw it by keeping a dictionary of numbers seen before.

• You may recall dictionary structures from 247, e.g. trees, hash tables, etc.

• We will put n numbers into our dictionary, and then answer queries of the form “is number x in there?"

• For our dictionary, we will use a binary search tree, with the input numbers acting as keys.
• We will use no special balancing tricks – just build the tree in the order in which inputs arrive.

• How long does it take to insert a number, and how long does it take to check if one is present?

• What is the worst-case depth for this tree?

• If the inputs arrive in sorted order, the depth becomes $O(n)$.

• Hence, worst-case insertion and search time is also $O(n)$.

• This is no better than a linked list!

Let’s see if average case analysis can give us a less terrible result.

• Assume that each time a number shows up, it is equally likely to be any value in the range 0..$N - 1$, independent of all previous inputs.

• Hence, $Pr(x) = 1/N$ for all $x$.

• First, we argue that, if $n$ is quite small relative to $N$, we can pretend that our inputs cannot contain duplicates.

• What is the chance that in $n$ inputs, we see the same value twice?

• If we’ve seen $k$ distinct inputs, then the chance that the next input is distinct from all of them is $1 - k/N$.

• Hence, the probability $P_d$ that $n$ successive inputs are distinct is

$$P_d = 1 \cdot \left(1 - \frac{1}{N}\right) \cdot \left(1 - \frac{2}{N}\right) \cdot \ldots \cdot \left(1 - \frac{n - 1}{N}\right).$$

• We will use the fact that, when $z \ll 1$, $1 - z \approx e^{-z}$.

• (This approximation is obtained via Taylor series expansion of the log, which leaves error $O(z^2)$ in the log, which is fine if $z$ is tiny.)

• Plugging in the approximation, we have $P_d \approx e^{-\sum_{i=1}^{n-1} i/N}$.

• Put another way,

$$\log P_d \approx -\frac{1}{N} \sum_{i=1}^{n-1} i = -\frac{n(n - 1)}{2N}.$$

• Let’s say we want to keep $n$ small enough that the chance of seeing a duplicate is at most some small $\delta$, e.g. 1%.

• Then we want $\log(1 - \delta) \geq \log P_d$. 

• Solving for \( n \) in this inequality gives us \( n \lesssim \sqrt{N} \).

• (More precisely, \( n(n - 1) \leq 2N \log(1/(1 - \delta)) \) – the threshold depends on \( \delta \) as well as \( N \).)

• Conclude that if \( n \) is about \( \sqrt{N} \), then with high probability, duplicates do not occur.

• If duplicates do happen, do something expensive to handle them – it doesn’t impact the running time much in practice.

OK, back to the trees.

• Suppose we put \( n \) inputs into our tree in the order in which they arrive, then search for some \( x \) that is in the tree.

• What is the expected time to perform this search?

• We start at the root and walk down the tree until we find \( x \), so we need to know the average depth at which \( x \) occurs.

• **Thm**: the average depth of \( x \) is given by

\[
\left(1 + \frac{1}{n}\right)(2H_{n+1} - 3) + \frac{1}{n} \approx O(\log n)
\]

where \( H_n = \sum_{i=1}^{n} 1/i \) is the \( n \)th harmonic number, which grows asymptotically as \( O(\log n) \).

• **Pf**: first, observe that the number of steps needed to find \( x \) in the tree is equal to the number of steps needed to insert it in the first place.

• We will compute \( C(n) \), the average total cost to insert all \( n \) elements, then divide by \( n \) to obtain the average cost to insert \( x \), which is also the average cost to find \( x \).

• When we insert an element into the tree, it always touches the root node. If the root already exists, it then goes left or right depending on its value relative to the root.

• With probability \( 1/n \), the element that is the \( j \)th smallest arrives first and becomes the root.

• In that case, \( j - 1 \) of the remaining elements go left, and the other \( n - j \) of them go right.

• The elements that go in each direction recursively form a subtree by the same random process as the whole.

• Hence, the average time \( C(n) \) to insert all \( n \) elements is given by the recurrence

\[
C(n) = n + \frac{1}{n} \sum_{j=1}^{n} (C(j - 1) + C(n - j)).
\]

• How can we solve this recurrence?
• Observe that

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

because most of the terms of the two summations cancel out.

• It follows after some algebra that

\[
\frac{C(n)}{n+1} = \frac{2}{n+1} - \frac{1}{n(n+1)} + \frac{C(n-1)}{n} \\
= \frac{2}{n+1} - \frac{1}{n+1} + \frac{1}{n} + \frac{C(n-1)}{n},
\]

which is a much easier recurrence – we can solve it iteratively.

• The explicit solution is then

\[
\frac{C(n)}{n+1} = 2 \sum_{j=1}^{n} \frac{1}{j+1} + \sum_{j=1}^{n} \left( \frac{1}{j+1} - \frac{1}{j} \right) \\
= 2H_{n+1} - 2 + \frac{1}{n+1} - 1 \\
= 2H_{n+1} + \frac{1}{n+1} - 3
\]

• Conclude that the average time to insert (equivalently, to look up) one element in the tree is given by

\[
\frac{C(n)}{n} = \left( \frac{n+1}{n} \right) \cdot \frac{C(n)}{n+1} \\
= \left( 1 + \frac{1}{n} \right) (2H_{n+1} - 3) + \frac{1}{n}
\]

as claimed. QED

OK, but what if we look up a random element that isn’t in the tree?

• Thm: the average number of steps to look up an \( x \) not in the tree is given by

\[
2H_{n+1} - 1 \approx O(\log n).
\]

• Pf: an element not in the tree will proceed all the way to some “null child” at the bottom the tree before failing.

• We start from the assumption that \( x \) is random and so equally likely to wind up at each null child.
• There are $n+1$ null children in a binary tree with $n$ nodes.

• Moreover, with probability $1/n$ (i.e. depending on the order in which the tree was built), exactly $j$ of these null children are to the left of the root, and $n+1-j$ are to the right.

• Let $S(n+1)$ be the total cost to insert an element at each of the $n+1$ null children.

• As before, we pay 1 step for each element to pass the root, then recursively pay for the elements that go to the left null children and the right null children according to how many there are of each.

• Conclude that

$$S(n+1) = n + 1 + \frac{1}{n} \sum_{j=1}^{n} (S(j) + S(n+1-j)).$$

• We can simplify this recurrence similarly to the one above to get

$$nS(n+1) - (n-1)S(n) = 2n + 2S(n),$$

and hence

$$\frac{S(n+1)}{n+1} = \frac{2}{n+1} + \frac{S(n)}{n}.$$

The LHS of this last equation is the average cost to look up one $x$ not in the tree, over random tree construction and random choice of $x$.

• Again solving this simpler recurrence iteratively, we get

$$\frac{S(n+1)}{n+1} = \sum_{j=1}^{n} \left( \frac{2}{j+1} \right) + 1$$

$$= 2H_{n+1} - 1$$

(taking advantage of the fact that $S(1) = 1$). QED

4 Naive Hashing

Another way to keep elements in a dictionary is as a hash table.

• Suppose for simplicity that the range of possible values $N$ is a power of 2.

• We’ll use a hash table of size $m \leq N$ that is also a power of 2.

• Our hash function will be $h(x) = x \mod m$.

• Observe that exactly $N/m$ possible values hash to each slot in the table.

• Hence, for a randomly chosen $x$, $\Pr(h(x) = i) = 1/N \times N/m = 1/m$.

Let’s say we insert $n$ randomly chosen elements into the table. What is the chance that no insertion causes a collision?
• **Thm**: Suppose we insert $n$ random elements from $0..N - 1$ into a table of size $m$ with $h(x) = x \mod m$.

• For any $\delta > 0$, if we set $1/m = O \left( \log^2 \left( \frac{1}{1-\delta} \right) \right)$...

• ... then with probability $1 - \delta$, we will observe no collisions after $n = O \left( \sqrt{m \log \left( \frac{1}{1-\delta} \right)} \right)$ insertions.

• (Proof omitted, but it is basically the same analysis of the chance of seeing $n$ unique elements that we did above, applied to the $m$ possible hash bins.)

• Conclude that if we set $m = O(n^2)$, we will rarely have collisions, and so lookups will almost surely take time $O(1)$.

• We can do something slow (e.g. chaining) to resolve the rare collision.