This is new material: if you see errors, please email jtysu at stanford dot edu

1 The secretary problem

We will start by analyzing the expected runtime of an algorithm, as you will be expected to do on your homework.

Consider the problem of hiring an office assistant. We interview candidates on a rolling basis, and at any given point we want to hire the best candidate we've seen so far. If a new candidate comes along, we immediately fire the old one and hire the new one.

```
Hire-Assistant(n):
    Randomly shuffle the candidates (O(n))
    best = 0
    for i = 1 to n:
        interview candidate i
        if candidate i is better than candidate ‘‘best’’
            best = i
            hire candidate i
```

In this model, there is a cost c_i associated with interviewing people, but a much larger cost c_h associated with hiring people. The cost of the algorithm is $O(c_i n + c_h m)$, where n is the total number of applicants, and m is the number of times we hire a new person.

Exercise: What is the worst-case cost of this algorithm?

Answer: In the worst case scenario, the candidates come in order of increasing quality, and we hire every person that we interview. Then the hiring cost is $O(c_h n)$, and the total cost is $O((c_i + c_h)n)$.

1.1 Expected hiring cost of HireAssistant

However, most of the time, we do not have to hire every candidate, so on average the total cost of the algorithm may be substantially less than the worst case cost.

Here we calculate the expected number of times we hire a new candidate.

1.1.1 Indicator random variables

An indicator random variable is a variable that indicates whether an event is happening. If A is an event, then the indicator random variable I_A is defined as

$$I_A = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$$

Example: Suppose we are flipping a coin n times. We let X_i be the indicator random variable associated with the coin coming up heads on the ith coin flip. So

$$X_i = \begin{cases} 1 & \text{if } i\text{th coin is heads} \\ 0 & \text{if } i\text{th coin is tails} \end{cases}$$

By summing the values of X_i, we can get the total number of heads across the n coin flips.

To find the expected number of heads, we first note that

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

by linearity of expectation. Now the computation reduces to computing $E[X_i]$ for a single coin flip, which is

$$\begin{aligned} E[X_i] &= 1 \cdot P(X_i = 1) + 0 \cdot P(X_i = 0) \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \end{aligned}$$

So the expected number of heads is $\sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n}(1/2) = n/2$.

Example: Let A be an event and I_A be the indicator random variable for that event. Then $E[I_A] = P(A)$.

Proof: There are two possibilities: either A occurs, in which case $I_A = 1$, or A does not occur, in which case $I_A = 0$. So

$$E[I_A] = 1 \cdot P(A) + 0 \cdot P(\text{not } A) = P(A)$$

1.1.2 Analysis of HireAssistant

We are interested in the number of times we hire a new candidate.

Let X_i be the indicator random variable that indicates whether candidate i is hired, i.e. let $X_i = 1$ if candidate i is hired, and 0 otherwise.

Let $X = \sum_{i=1}^{n} X_i$ be the number of times we hire a new candidate. We want to find $E[X]$, which by linearity of expectation, is just $\sum_{i=1}^{n} E[X_i]$.

By the previous example, $E[X_i] = P(\text{candidate } i \text{ is hired})$, so we just need to find this probability. To do this, we assume that the candidates are interviewed in a random order. (We can enforce this by including a randomization step at the beginning of our algorithm.)

Candidate i is hired when candidate i is better than all of the candidates 1 through $i-1$. Now consider only the first i candidates, which must appear in a random order. Any one of them is equally likely to be the best-qualified thus far. So the probability that candidate i is better than candidates 1 through $i-1$ is just $1/i$.

Therefore $E[X_i] = 1/i$, and $E[X] = \sum_{i=1}^{n} 1/i = \ln n + O(1)$ (by equation A.7 in the textbook).

So the expected number of candidates hired is $O(\ln n)$, and the expected hiring cost is $O(c_h \ln n)$.

2 Lower bound on randomized comparison sorting algorithms

In lecture, we talked about how any sorting algorithm that relies only on comparisons between array elements for information must make at least $\Omega(n \log n)$ comparisons, which gives us a lower bound on the runtime. Today we will prove that this lower bound still holds, even if you use **randomized** comparison sorting algorithms.

A **randomized algorithm** is just a program that uses randomness. In Python, you would import the "random" library. Formally, a randomized algorithm is a program that has access to a special source of "random" bits, and every time it needs to make a random decision, it just pulls the next bit from its source.

A **deterministic algorithm** is an algorithm that does not use randomness (although technically you could say that a deterministic algorithm is just a randomized algorithm that ignores its source of random bits).

In this class, all the randomized algorithms we see will produce correct output 100% of the time, but the runtime of the algorithms might vary depending on the results of the random coin flips.

2.1 Worst case, deterministic

Theorem: Any deterministic comparison-based sorting algorithm must make at least $\Omega(n \log n)$ comparisons, in the worst case.

We can view the algorithm as a decision tree, where nodes correspond to comparisons between array elements, and each leaf corresponds to an ordering of the array elements.

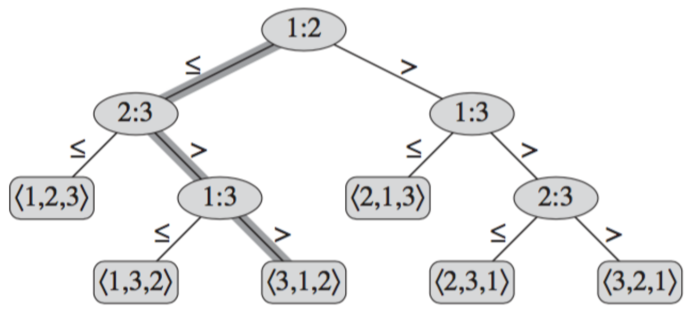

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

Now in order for a comparison sort to be correct, the leaves of the decision tree must contain all possible permutations of the input array. This is because the input can be in any possible order, so the algorithm must be able to output all possible rankings of the input elements.

Indeed, each possible ordering must be represented exactly once. That is because in order for a leaf to appear on two different decision tree branches, one of the comparisons would have to give different results for two inputs, and if the two inputs are ordered in the same way, that is impossible.

Therefore, there must be exactly $n!$ leaves of the decision tree (since there are $n!$ ways to order an array of n elements).

The worst case runtime is just the length of the longest path through the decision tree. This is just $\Omega(\log(n!))$. By Stirling's formula, this is $\Omega(n \log n)$.

2.1.1 Why the height of a binary tree with m leaves is $\Omega(\log m)$

If a tree has height h, it must have at most 2^h leaves. This is because each leaf corresponds to a unique path from the root of the tree to the bottom of the tree. At each junction, there are at most two directions the path can take. Since there are at most h choices to be made, and each choice increases the number of possible paths by (at most) a factor of 2, there are at most 2^h paths to the bottom of the tree, so the tree has at most 2^h leaves.

Therefore, if the tree has m leaves, its height is at least $\log m$.

2.2 Average case, deterministic

2.2.1 Average-case analysis

So far this class has mainly focused on the worst case cost of algorithms, i.e. how fast the algorithm is when run on the worst possible input. Worst case analysis is very important, but sometimes the typical case is a lot better than the worst case, so algorithms with poor worst-case performance may be useful in practice.

Defining the "average" case is tricky, because it requires certain assumptions on how often different types of inputs are fed to the algorithm.

If we know the distribution of inputs, we can compute the average-case cost of an algorithm by finding the cost on each input, and then averaging the costs together in accordance with how likely the input is to come up.

Example: Suppose the algorithm's costs are as follows:

Input	Probability that the input happens	Cost of the algorithm when run on that input
A	0.5	1
B	0.3	2
C	0.2	3

Then the average-case cost of the algorithm is just the expected value of the cost, which is $1 \cdot 0.5 + 2 \cdot 0.3 + 3 \cdot 0.2$.

Here, it is worth discussing the difference between expected runtime and average-case runtime. When we compute the average-case cost of an algorithm, we are averaging over the different types of inputs that might be fed to the algorithm. However, when we compute the expected runtime of a randomized algorithm, we are averaging over the inherent randomness in the algorithm. (For example, if our function calls `random.randint(1, 6)`, we would run the algorithm once for every possible output of `randint()`, and average the runtimes to find the expected cost).

2.2.2 Average case bound on deterministic comparison-based sorting algorithms

So far, we have shown that for any deterministic comparison-based sorting algorithm, we can find an input that forces it to make $\Omega(n \log n)$ comparisons. It turns out that we also have to make $\Omega(n \log n)$ comparisons in the average case (so it's not just a statement about super-pathological inputs).

Remember that this statement is meaningless unless we know how often different types of inputs are fed to the algorithm. When we say the average case is $\Omega(n \log n)$, we assume that **each ordering of an input array is equally likely to occur.** Under other distributions of inputs, this lower bound does not necessarily hold.

Formally, we have

Theorem: For any deterministic comparison-based sorting algorithm $\mathcal{A}$, the average-case number of comparisons (the number of comparisons on average on a randomly chosen permutation of n distinct elements) is at least $\Omega(\log(n!)) = \Omega(n \log n)$.

Proof: The average-case number of comparisons is simply the average length of the paths in the decision tree from the root to the leaves. This is because each possible permutation corresponds to exactly one leaf, which corresponds to a unique path in the decision tree.

We find that (1) trees that are more unbalanced have a longer average path length (2) trees that are perfectly balanced have an average path length of $\lceil \log(n!) \rceil - 1$. Together, these facts show that the average path length will always be at least $\lceil \log(n!) \rceil - 1$.

2.2.3 Balanced binary trees

In a perfectly balanced binary tree with $n!$ leaves, the height of each path from the root to a leaf is either the height of the tree, or the height of the tree minus 1. (Let's define it that way for now.) Recall that we proved that the height of a tree with $n!$ leaves must be at least $\log(n!)$. (This means it must be at least $\lceil \log(n!) \rceil$.) So each path from the root to a leaf must have length $\geq \lceil \log(n!) \rceil - 1$. Since each input results in a trip from the root to a leaf, each input must result in $\Omega(\log(n!))$ comparisons, so the average-case number of comparisons is $\Omega(\log(n!))$.

2.2.4 Unbalanced binary trees

In an unbalanced binary tree, the highest leaf x must be at least two levels higher than the lowest leaf y. In fact, y must be one of two "sibling leaves", since we are dealing with a decision tree and both decisions must correspond to an output. Moving y and y's sibling to be the child of x decreases the average height of the leaves. If the smaller depth is d and the larger depth is D, then we have removed two leaves of depth D (y and y's sibling) and one leaf of depth d (namely x), and we have added two leaves of depth $d + 1$ (y and y's sibling) and one leaf of depth $D - 1$ (y's parent).

Therefore, any unbalanced decision tree can be modified to have smaller average depth, so the decision tree with the lowest average depth must be balanced. Therefore, the average-case number of comparisons is still $\Omega(\log(n!))$.

2.3 Average case, randomized

You may think of a randomized algorithm as a collection of deterministic algorithms, each with the random number choices hardwired in. For example, if your algorithm $\mathcal{A}$ involves choosing a random number between 1 and 6, you can actually represent it as six different deterministic algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_{\mathcal{6}}$ where algorithm $\mathcal{A}_{\mathcal{i}}$ is the version of $\mathcal{A}$ where we chose i as our random number.

Then the expected number of comparisons made by randomized algorithm $\mathcal{A}$ on an input I is just

$$\sum_s Pr(s)(\text{Number of comparisons of } \mathcal{A}_f \text{ on } I)$$

Now to find the average case number of comparisons, we must average this runtime over all possible inputs I:

$$\sum_I Pr(I) \sum_s Pr(s)(\text{Number of comparisons of } \mathcal{A}_f \text{ on } I)$$

We can reverse the order of summations:

$$\sum_s Pr(s) \sum_I Pr(I)(\text{Number of comparisons of } \mathcal{A}_f \text{ on } I)$$

And since any deterministic algorithm makes at least $\lceil \log(n!) \rceil - 1$ comparisons in the average case, our current expression must be at least

$$\sum_s Pr(s)(\lceil \log(n!) \rceil - 1)$$

And since $\sum_s Pr(s) = 1$, this is just $\lceil \log(n!) \rceil - 1 = \Omega(n \log n)$.

2.4 Worst case, randomized

Worst case lower bounds are always at least as large as average case lower bounds, so in the worst case, we also get $\Omega(n \log n)$ comparisons.

3 References

Comparison-based Lower Bounds For Sorting
`https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf`