1 Probabilistic analysis and randomized algorithms

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 better candidate comes along, we immediately fire the old one and hire the new one.

```
HIRE-ASSISTANT(n)
1  best = 0          // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          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 Average case analysis

So far this class has mainly focused on the worst case cost of algorithms. 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. (We will use HireAssistant as an example of this.)

Defining the "average" case is tricky, because it requires certain assumptions on how often different types of inputs come up. One possible assumption is that all permutations (i.e. orderings) of candidates are equally likely. But this assumption might not always be true. For example, candidates may be sourced through a recruiter, who sends the strongest candidates first because those are the candidates who are most likely to get him a referral bonus. Without knowing the distribution of inputs, it's hard to perform a rigorous average-case analysis.

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 $1 \cdot 0.5 + 2 \cdot 0.3 + 3 \cdot 0.2$.

In HireAssistant, we can kind of enforce a distribution on inputs by changing the model a bit. Suppose that instead of the candidates coming in on a rolling basis, the recruiter sends us the list of n candidates in advance, and we get to decide which ones to interview first. Then our algorithm can include a randomization step where we choose randomly which candidate to interview on each day.

This is important when analyzing the algorithm, because now that each ordering of candidates is equally likely, it is easier to compute the expected cost. It's also important when designing the algorithm, because now an evil user can't feed the algorithm a deliberately bad input. No particular input elicits the worst case behavior of the algorithm.

1.2 Average case analysis of HireAssistant

1.2.1 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

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

1.2.2 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 Quicksort

2.1 Algorithm

Recall the Partition function from last lecture, which takes as input an array slice and a pivot element, and rearranges the elements in that slice so that all elements less than the

pivot are left of the pivot and all elements greater than the pivot are right of the pivot. Recall that Partition runs in $\Theta(n)$ time, and rearranges the elements in-place, returning the index of the pivot.

We can use Partition to sort an array in-place, as follows. Note that q is the pivot index after the Partition step.

```
Quicksort(array A, int startIndex, int endIndex):
    if startIndex < endIndex:
        choose a suitable pivot
        q = Partition(A, startIndex, endIndex, pivot)
        Quicksort(A, startIndex, q - 1)
        Quicksort(A, q + 1, endIndex)
```

The deterministic version of this algorithm chooses the last element of the array as the pivot. The randomized version chooses a random element as a pivot.

2.1.1 Intuitive performance of quicksort (deterministic case)

The performance of this algorithm depends largely on the choice of pivot.

In one scenario, the pivot always happens to be either the largest or smallest element in the subarray (we'll prove later that this situation results in the worst case runtime). One situation in which this happens is if the array is already sorted. This would produce one subproblem with $n-1$ elements and one with 0 elements.

Then the recurrence would be

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

since the base case is done in constant time.

Summing the costs of each level of the recursion, we get $n + (n-1) + (n-2) + \cdots + 1$, which evaluates to $\Theta(n^2)$.

In another scenario, the pivot always evenly splits the subarray into two halves. So we have two subproblems, one of size $\lfloor n/2 \rfloor$, and one of size $\lceil n/2 \rceil - 1$. In this case (ignoring floors and ceilings), the recurrence relation is

$$T(n) = 2T(n/2) + \Theta(n)$$

which simplifies to $T(n) = \Theta(n \log n)$ by case 2 of the master theorem.

Typically, the performance of quicksort is much closer to the second scenario than the first. To see this, we note two facts:

1. Any constant-proportionality split still gives us a recursion tree of height $O(\log n)$. For instance, if the pivot always splits the subarray into one piece that is 90% of the array

and another piece that is 10% of the array, the maximum cost of a node goes down by a factor of 9/10 on each level, which gives the tree logarithmic height, and the cost at each level is linear. So we can tolerate quite a bit of "unbalancedness" and still get the $O(n \log n)$ runtime.

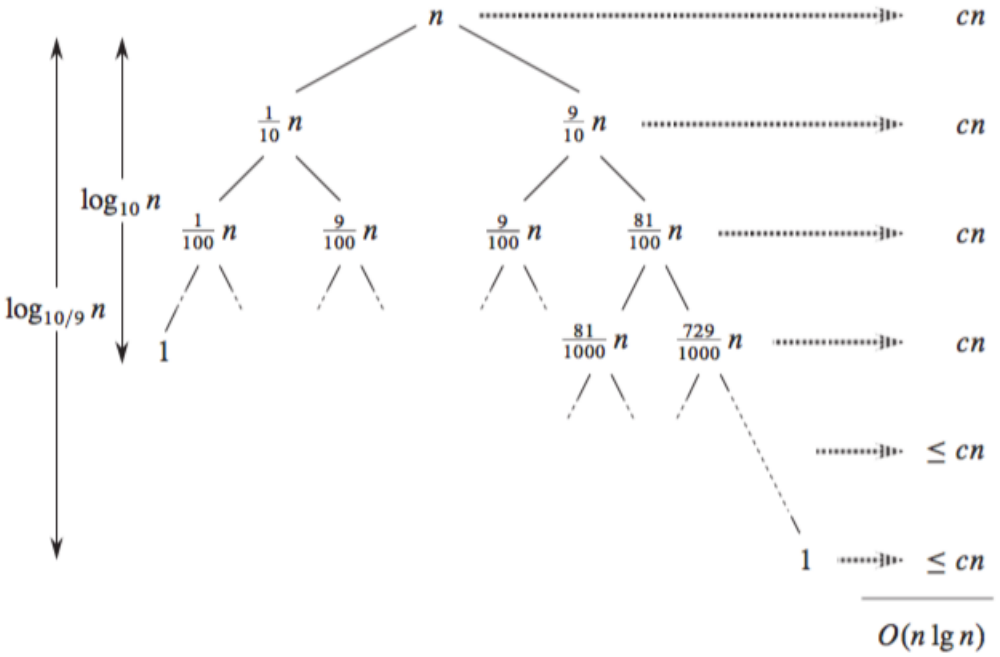

Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

2. We can even tolerate a lot of non-constant-proportionality splits. For instance, if the tree alternates between "good" splits (constant proportionality) and "bad" splits (where the array is split into one subproblem of size $n - 1$ and one subproblem of size 0), we still get essentially the same runtime, just with twice as many levels.

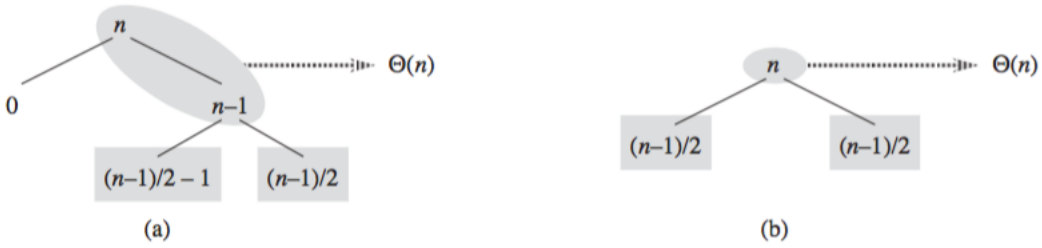

Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n-1$. The partitioning of the subarray of size $n-1$ costs $n-1$ and produces a "good" split: subarrays of size $(n-1)/2-1$ and $(n-1)/2$. **(b)** A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

So we expect that in a typical case, we will get the $O(n \log n)$ runtime.

2.2 Worst case runtime

We will use the substitution method to prove that the worst case runtime is $O(n^2)$. The runtime of quicksort on a given input is described by the recurrence

$$T(n) = T(q) + T(n-q-1) + \Theta(n)$$

where q is a number that is different on each level of the tree.

Similarly, the worst-case runtime of quicksort is described by the recurrence

$$T_{worst}(n) = \max_{0 \le q \le n-1} (T_{worst}(q) + T_{worst}(n-q-1)) + \Theta(n)$$

We guess that $T_{worst}(n) \le cn^2$ for some constant c. Substituting, we get that

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + ((n-1)-q)^2) + \Theta(n) \end{aligned}$$

The highest points of the function $q^2 + (n-1-q)^2$ are at $q = 0$ and $q = n-1$ (in which case the function equals $(n-1)^2$). These are the highest points because (for constant n) this function is an upward-facing parabola, so its highest values are at both endpoints.

Thus, we get $T(n) \le c(n-1)^2 + \Theta(n) = cn^2 - c(2n-1) + \Theta(n)$, which is less than or equal to cn^2 if we choose a c that is large enough for the $c(2n-1)$ term to dominate the $\Theta(n)$ term.

2.3 Average case runtime (formal treatment)

Here we will assume we are using the randomized version of Quicksort, where we choose the pivot randomly each time.

```
Partition(A, leftEdge, rightEdge, pivot):
    x = A[pivot]
    exchange A[pivot] with A[rightEdge]
    boundary = leftEdge - 1
    for j = leftEdge to rightEdge - 1:
        if A[j] <= x:
            // Move element A[j] into the left side of the partition
            boundary = boundary + 1
            exchange A[boundary] with A[j]
    exchange A[boundary + 1] with A[rightEdge]
    return boundary + 1
```

The running time of Quicksort is determined by the number of times array elements are compared to each other.

Why? The running time is dominated by the time spent in the Partition procedure. Each call to Partition takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the for loop. On every iteration of the for loop, an array element is compared to the pivot.

(Note that not every element in the array is compared to every other element in the array. Once the pivot splits the array into left and right pieces, no element in the left piece is compared to any element in the right piece, which means the total number of comparisons is not necessarily $\Theta(n^2)$.)

In addition to the time spent on comparisons, each call to Partition does additional $O(1)$ work. There are at most n calls to Partition, because each time the Partition procedure is called, it selects a pivot element, and that element is never included in any future recursive calls to Quicksort or Partition. Therefore, the total runtime of Quicksort is $O(n+X)$, where X is the total number of comparisons performed over the entire execution of Quicksort.

2.3.1 Computing X

Let $z_1, z_2, \ldots, z_n$ be the elements of the array in sorted order. Let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ be the set of elements between z_i and z_j.

Observe that each pair of elements is compared at most once. Elements are compared only to the pivot element, and after Partition finishes, the pivot element used in that call is never again compared to any other element. So we can define an indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \text{ at any point during the algorithm} \\ 0 & \text{otherwise} \end{cases}$$

The total number of comparisons performed by the algorithm is

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij},$$

and

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(z_i \text{ is compared to } z_j).$$

So we just need to compute the probability that z_i is compared to z_j. (Here we assume that all elements are distinct.)

Let us think about when two elements are not compared. Consider an input to quicksort of the numbers 1 through 10 (in any order), and suppose that the first pivot element is 7. Then the first call to Partition separates the numbers into two sets: $\{1, 2, 3, 4, 5, 6\}$ and $\{8, 9, 10\}$. In doing so, the pivot 7 is compared to all other elements, but no number from the first set will ever be compared to any number from the second set.

In general, once a pivot x is chosen with $z_i < x < z_j$, we know that z_i and z_j will never be compared. However, if z_i is chosen as the pivot before any other element between z_i and z_j, then z_i gets compared to every element in that range, including z_j. Similarly, if z_j is chosen as the pivot first, then it gets compared to z_i. In our example, 7 is compared with 9 because 7 is the first item from $\{z_7, z_8, z_9\}$ to be chosen as a pivot. However, 2 will never be compared with 9 because 7 was chosen as a pivot before any other element in $\{z_2, \ldots, z_9\}$. So z_i and z_j are compared if and only if the first element to be chosen as a pivot from $\{z_i, \ldots, z_j\}$ is either z_i or z_j.

Before any element from $\{z_i, \ldots, z_j\}$ has been chosen as a pivot, the whole set is together in the same partition, so any element of that set is equally likely to be the first one chosen as a pivot (since pivots are chosen at random). This set has $j - i + 1$ elements, so the probability that either z_i or z_j is chosen first is $\frac{2}{j-i+1}$. Therefore,

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

We evaluate this using some summation trickery.

$$\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \quad (\text{let } k = j-i) \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \quad (\text{remember we only need an upper bound here}) \\
&= \sum_{i=1}^{n-1} O(\log n) \quad (\text{by the sum of a harmonic series}) \\
&= O(n \log n)
\end{aligned}$$

Therefore, the expected running time of quicksort is $O(n \log n)$ when all the elements are distinct.