1 Introduction (January 21)

1.1 Deterministic Complexity

Consider a monotonically nondecreasing function $f : \{0, 1, \ldots, n\} \to \{0, 1\}$, where $f(0) = 0$ and $f(n) = 1$. We call f a *step function*. There must be an integer i between 0 and $n - 1$, called the *break point*, such that $f(i) = 0$ and $f(i + 1) = 1$. Given a step function as input, how efficiently can we find its break point? Specifically, how many times do we need to evaluate the function f?

The algorithms that we consider or this problem are *adaptive*: the point where we evaluate the function at each step depends (at least in principle) on the results of all previous evaluations. Since we want the fastest possible algorithm, we can clearly consider only *irredundant* algorithms, which never evaluate the function more than once at the same location.

Let $T(A, f)$ denote the number of function evaluations performed by some algorithm A given the step function f as input. The *time complexity* of an algorithm A is its worst-case running time over all possible inputs:

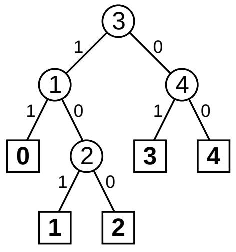
$$T(A) = \max_{|f|=n} T(A, f).$$

Finally, the time complexity of the *problem* is the time complexity of the fastest algorithm that correctly solves it.

$$T(n) = \min_{A} T(A) = \min_{A} \max_{|f|=n} T(A, f).$$

Note that the time complexity of a problem is a function of n, the size of the input function f.

We can represent any algorithm for this problem as a *decision tree*. A decision tree is a rooted, ordered, binary tree. Each internal node v is labeled with an integer n_v and has two outgoing edges labeled 0 and 1. To execute the algorithm, we start at the root node v, follow the edge labeled $f(n_v)$, and continue recursively. Thus, the execution of the algorithm gives us a path from the root to some leaf. Each leaf has an integer label; when the execution reaches a leaf, its label is returned as the algorithm's output. For example, the following decision tree describes a working algorithm for our problem when $n = 5$.



A decision tree.

With this model in mind, we can redefine $T(A, f)$ as the length of the root-to-leaf path in A traversed when the input is f. The complexity $T(A)$ of any decision tree algorithm A is simply its depth, and the complexity of the problem is the depth of the shallowest decision tree. For this problem, we can easily derive the exact value of $T(n)$:

$$T(n) = \min_{A} \max_{f} T(A, f) = \min_{\text{trees } A} \max_{\text{leaves } \ell \in T} \text{depth}(\ell) = \lceil \log_2 n \rceil$$

In particular, the algorithm illustrated above is optimal for $n = 5$.

1.2 Randomized Complexity

We can also consider *randomized* algorithms, which can base some of their decisions on random coin flips. Without loss of generality, we can assume that all the coin flips are made at the beginning of the algorithm. Thus, we can think of any randomized algorithm as a probability distribution over a set of deterministic algorithms, or in this case, as a probability distribution over decision trees.

Given a randomized algorithm P, let $\Pr_P[A]$ denote the probability of choosing the deterministic algorithm A. The *expected* running time of a randomized algorithm P on input f is just the weighted average of $T(A, f)$ over all deterministic algorithms A:

$$\overline{T}(P, f) = \mathrm{E}_P[T(A, f)] = \sum_A \Pr_P[A] \cdot T(A, f).$$

The time complexity of a randomized algorithm is its worst-case expected running time:

$$\overline{T}(P) = \max_f \sum_A \Pr_P[A] \cdot T(A, f)$$

Finally, the randomized (time) complexity of the problem is the time complexity of the fastest randomized algorithm that solves it:

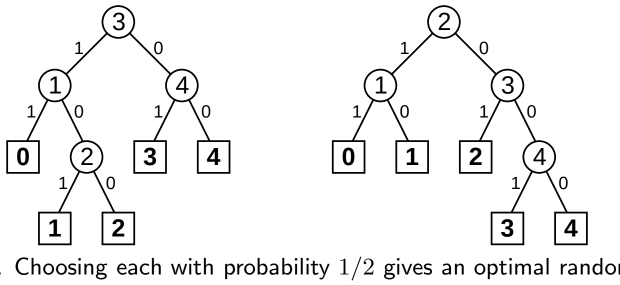
$$\overline{T}(P) = \min_P \max_f \sum_A \Pr_P[A] \cdot T(A, f)$$

It is crucial to note that we are not assuming anything about the distribution of possible *inputs* here. The expectation is taken entirely over random bits generated by the algorithm. This distinguishes randomized complexity analysis from "average case" analysis of algorithms, which typically (and unrealistically!) assumes that all inputs are equally likely.

Any deterministic algorithm can also be considered a trivial randomized algorithm, where all the probability is concentrated on one decision tree. Thus, the randomized complexity of a problem can never be larger than its deterministic complexity.

$$\boxed{\overline{T}(n) \leq T(n)}$$

However, it can be strictly smaller. Consider the break point problem again. Let P be the randomized algorithm that flips a fair coin, and chooses between the two trees below depending on the outcome. We have $\overline{T}(P) = 5/2$, which implies that $\overline{T}(5) \leq 5/2$, even though $T(5) = 3$.



Two decision trees. Choosing each with probability $1/2$ gives an optimal randomized algorithm.

In fact, this randomized algorithm is optimal. Brute-force case analysis implies that in any binary tree with five leaves labeled 0 to 4 in left-to-right order, $\mathrm{depth}(1) + \mathrm{depth}(3) \geq 5$. In other words, for any deterministic algorithm A, we have $T(A, f_1) + T(A, f_3) \geq 5$, where f_i denotes the step function with break point i. Thus, for any randomized algorithm P, we have

$$\overline{T}(P, f_1) + \overline{T}(P, f_3) = \sum_A \Pr_P[A](T(A, f_1) + T(A, f_3)) \geq 5.$$

Either $\overline{T}(P, f_1)$ or $\overline{T}(P, f_3)$ must be at least $5/2$, since their sum is 5, and in either case, we have $\overline{T}(P) \geq \max\{\overline{T}(P, f_1), \overline{T}(P, f_3)\} \geq 5/2$. Since we already have a matching upper bound, we conclude that $\overline{T}(P) = 5/2$.

Exercise: What is the *exact* randomized time complexity of the break point problem, as a function of n? *[Hint: An asymptotic bound of $\Theta(\log n)$ is easy.]*

1.3 Methodology

There are several differences between the type of complexity analysis we will consider in this course and the more familiar "complexity theory" encountered in courses like CS 375 and CS 479.

- We do not consider the behavior of a "universal" computing device such as a Turing machine or RAM. Rather, we develop specialized ad hoc models tailored to the problem at hand. Typically these models consider only a single basic operation—for example, memory accesses, comparisons, arithmetic, pointer traversals—with the implicit assumption that the number of basic operations dominates the running time of any "reasonable" algorithm.
- We consider the problem size n to be fixed. In typical algorithm analysis, we consider arbitrarily large inputs, but developing lower bounds in this setting is extremely hard. An algorithm will be represented by some finite structure—for example, decision tree, boolean circuit, directed graph, transition matrix—with an associated measure of complexity. We are interested in the asymptotic growth rate of this complexity as a function of the input size.
- A consequence of fixing the input size is that we allow *nonuniform* algorithms. That is, the algorithms for different input sizes need not be described by a single uniform algorithm; they could be completely unrelated. Lower bounds for nonuniform algorithms clearly apply to uniform algorithms as well.
- Finally, we will attempt to exhibit upper bounds that match our lower bounds, usually be developing a single *uniform* algorithms that works for all possible input sizes. Needless to say, we will not always succeed. There are a few problems whose nonuniform complexity is (as far as we know) exponentially smaller than their uniform complexity!

1.4 A slightly harder problem

One of the very first problems for which lower bounds were developed was considered by a statistician named Kiefer in the early 1950s.[1] We call a function $f : \{0, 1, \ldots, n-1\} \to \mathbb{R}$ *unimodal* if it has a unique maximum value and no two successive function values are equal.[2] In other words, for some optimal index $0 \leq x^* \leq n-1$, we have

$$f(0) < f(1) < \cdots < f(x^*-1) < f(x^*) > f(x^*+1) > \cdots > f(n-1).$$

In particular, if $x^* = 0$, the function is monotonically decreasing, and if $x^* = n-1$, the function is monotonically increasing. Our task is to determine, given a unimodal function f as input, the index x^* such that $f(x^*)$ is maximized, evaluating the function f at as few points as possible.

Clearly we need at least two function evaluations to learn anything at all. Suppose we evaluate $f(x)$ and $f(y)$ for some indices $x < y$. If $f(x) < f(y)$, the optimal index x^* must be strictly greater

[1] J. Kiefer, Sequential minimax search for a maximum, *Proc. AMS* 4:502–506, 1953.

[2] Using $\{0, 1, \ldots, n-1\}$ instead of the more "natural" range $\{1, 2, \ldots, n\}$ will simplify our analysis slightly.

than x. Similarly, if $f(x) > f(y)$, we must have $x^* < y$. Finally, if $f(x) = f(y)$, the optimal index must lie strictly between x and y.

In general, after we evaluate the function at several points, there are two cases. If there is a unique maximum among the evaluated points, the true maximum must lie strictly between its neighbors. Otherwise, there must be two evaluation points with the same maximum value, and the true maximum lies strictly between them. To put the top-level problem into this context, we can pretend that $f(-1) = f(n) = -\infty$, and that these values are known to the algorithm in advance.

In either case, we can maintain a *pinning interval* that contains the optimal index x^*. Initially, the pinning interval is the range $[0 \,..\, n]$; each function evaluation potentially reduces this interval; and when the pinning interval contains only one index, it must be the maximum, so we return it. It clearly suffices to maintain only the left and right endpoints of the pinning interval and (if there is one) the single known function value within that interval.

As usual, we want to determine the complexity $T(n)$ of this problem as a function of the input size n. A simple variant of binary search implies that $T(n) \le 2\lceil \log_2 n \rceil$, and a leaf-counting argument[3] implies that $T(n) \ge \lceil \log_2 n \rceil + 1$, but we want an *exact* answer!

Our task will be drastically simplified by the following simple but powerful observation.

The Little Birdie Principle. *Extra information cannot increase the complexity of the problem.*

This principle implies, among other things, that decreasing the size of the pinning interval cannot increase the complexity of our problem. In other words, the function $T(n)$ is nondecreasing in the input size n.

Perhaps the easiest method to derive tight bounds on $T(n)$ is to turn the problem on its head. Let $N(t)$ be the size of the largest pinning interval such that we can determine the maximum value in at most t function evaluations. A few minutes of farting around tells us that $N(1) = 1$ and $N(2) = 2$.

Theorem 1. *$N(t) = F_{t+1} - 1$, where F_k is the kth Fibonacci number.*[4]

Proof: I will show that $N(t) = N(t-1) + N(t-2) + 1$ for all $t > 2$, from which the theorem follows immediately. We prove a lower bound (that is, an upper bound on $T(n)$) by describing an algorithm and an upper bound (that is, a lower bound on $T(n)$) using an adversary argument.

Suppose $n = N(t-1) + N(t-2) + 1$. To find x^* in t steps, our algorithm begins by evaluating the function at two locations: $f(N(t-2))$ and $f(N(t-1))$. These two values split the n-element range into three smaller ranges:

$$\begin{array}{|c|c|c|c|c|}\hline \longleftarrow N(t-2) \longrightarrow & \bullet & \longleftarrow N(t-3) \rightarrow & \bullet & \longleftarrow N(t-2) \longrightarrow \\ \hline \end{array}$$

If $f(N(t-2)) = f(N(t-1))$, the algorithm recurses in the middle interval, which has length $N(t-3)$. By the inductive hypothesis, the algorithm finishes in only $t-3$ more steps, so in this case, the total number of steps is only $t-1$.

Otherwise, the algorithm tosses out (say) the right segment of the range and recurses on the remainder. The length of the remaining interval is $N(t-1)$, so the algorithm will terminate after $t-1$ more steps. Naively, this appears to give a total of $t+1$ steps, but observe that the first recursive call should evaluate $f(N(t-3))$ and $f(N(t-2))$. Since the latter value is already known, we should not charge for that evaluation a second time.

[3]More about this in later lectures!

[4]Recall that $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for all integers $k \ge 2$.

To prove that this is optimal, we describe an inductive adversary that immediately exploits any deviation from the algorithm just described. Suppose some algorithm's first two probes are at $f(i)$ and $f(j)$, where $i < j$.

- If the algorithm follows the strategy outlined earlier, the adversary chooses values such that $f(i) \neq f(j)$. By our earlier analysis, the algorithm requires t steps to terminate.
- If $i < N(t-2)$, the adversary chooses values such that $f(i) < f(j)$, and also fixes $f(x)$ for all $i < x$ so that the fucntion is strictly increasing in that range. The algorithm is left with a pinning interval of length $n - i > N(t-1)$ with one known value. Thus, by the inductive hypothesis, it requires more than $t-2$ more steps to determine x^* ($t-1$ steps to solve the problem, minus one step for the known value), for a total of more than t steps.
- On the other hand, if $i > N(t-2)$, the adversary chooses values such that $f(i) > f(j)$, and then whispers in the algorithm's ear, "Psst! Hey buddy! Don't tell anybody I told you, but $x^* \leq N(t-2)$." The adversary also fixes $f(x)$ for all $x > N(t-2)$ so that the function is decreasing in that range. By the Little Birdie Principle, this extra information cannot *hurt* the algorithm's performance. On the other hand, if the algorithm knows that $x^* < i$, it's left with an interval of size $N(t-2)+1$ without a single known value. Thus, by the inductive hypothesis, the algorithm requires more than $t-2$ more steps to determine x^*, or more than t steps overall.
- Finally, if $j \neq N(t-1)$, we behave symmetrically to one of the two previous cases.

We conclude that *any* algorithm other than the one we described earlier will require at least $t+1$ steps when $n \geq N(t)$. □

Exercise: Let $f : \{0, 1, \ldots, n\} \to \{0, 1, \ldots, m\}$ be a nondecreasing function such that $f(0) = 0$ and $f(n) = m$. How many times do we need to evaluate f to learn enough information to describe the function completely? In other words, for each value $0 \leq y \leq m$, we want to know the largest and smallest values of x (if any) such that $f(x) = m$. Let $T(n,m) = \min_A \max_f T(A,f)$, where $T(A,f)$ is the running time of a deterministic algoorithm that learns the function f. We can immediately deduce a few facts:

- $T(n,0) = 0$: If $m = 0$, then $f(x) = 0$ for all x.
- $T(n,1) = \lceil \lg n \rceil$: This is just the break point problem.
- $T(1,m) = 0$: We already know that $f(0) = 0$ and $f(1) = m$.
- $T(n, \lfloor n/2 \rfloor) = n-1$: Consider the function $f(x) = \lfloor x/2 \rfloor$.

1. Prove that $T(n,m) = \max_x \min_y \big(1 + T(x,y) + T(n-x, m-y)\big)$.
2. Prove that $T(n,m) = \lceil n/2^{\lfloor \lg(n/m) \rfloor} \rceil + m \lfloor \lg(n/m) \rfloor - 1$ for all $2 \leq n < 2m$.

Exercise: UIUC has just finished constructing the new Reingold Building, the tallest structure on campus. In order to determine how much money it will have to pay in insurance, the university administration needs to determine the highest floor from which a student can jump without dying. The only way to determine whether a floor is safe is for a student "volunteer" to jump from a window on that floor. If the student survives, that floor is considered safe; if the student dies, the floor is declared unsafe. For obvious reasons, the University wants to minimize the number of tests. However, there are only a handful of student "volunteers" available. To simplify things, let's assume that each floor is either completely safe or immediately deadly to every person who jumps.

Suppose the building is n stories tall, and there are v volunteers. What is the exact minimum number of tests necessary and sufficient to locate the highest safe floor, in the worst case?