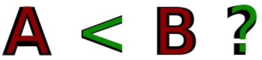
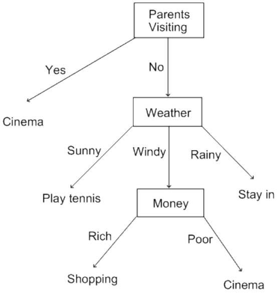
Lower Bounds

Pedro Ribeiro

DCC/FCUP

2018/2019





Upper Bound Problem

- One typical question when designing algorithms is:

  "given some problem X, can we construct an algorithm that runs in time $O(f(n))$ on inputs of size n?"

- This can be seen as an **upper bound** problem, and our goal is to make $f(n)$ as low as possible.

- In order to prove an upper bound we simply should find one algorithm A with that bound:

  In other words, when we give a complexity for our algorithm, what we are really doing, and what many computer scientists spend their career doing, is bragging about how *easy* a problem is...

Lower Bound Problem

- In this lecture, the question is different:

  "given some problem X, what is $g(n)$ such that *any* algorithm must take time $\Omega(g(n))$ on inputs of size n?"

- This can be seen as a **lower bound** problem, and our goal is to make $g(n)$ as high as possible.

- Lower bounds help us understand how *hard* a problem is, i.e., what is its **intrinsic difficulty** and how close we are to the optimal solution.

- This is **much harder** than proving an upper bound, because we must prove that **all** algorithms that solve the problem must be $\Omega(g(n))$, or equivalently, that **no** algorithm has $o(g(n))$ complexity.

Decision Trees

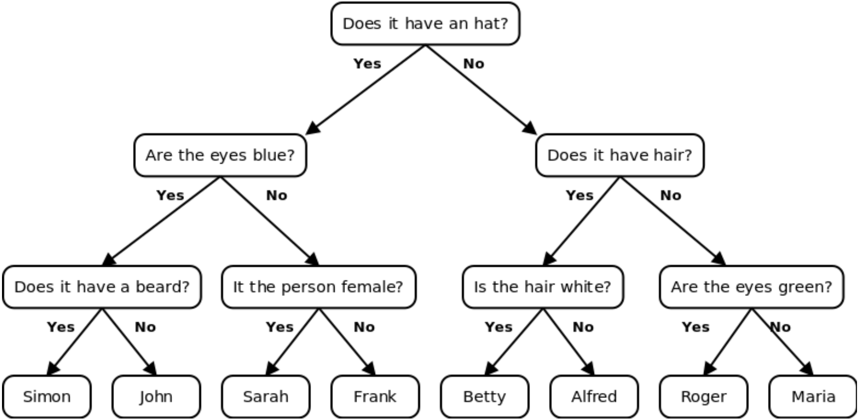
- Unfortunately, there is no formal definition for ***all algorithms***...
- We need to specify *precisely* what kind of algorithms we are considering and *precisely* how to measure their running time. This specification is called a **model of computation**.
- One powerful model of computation (and the only one we are going to talk about here) is the **decision tree** model:
  - As the name suggests, it is a *tree*
  - Each *internal node* is labeled by a *query* (question about the input), and its outgoing edges correspond to the possible answers
  - Each *leaf* of the tree is labeled with a possible *output*
  - To *compute* with a decision tree, we start at the root and simply follow a path to a leaf. The answers at each query tells us which node to visit next, and when we are at a leaf, we output the correspondent result
  - The *cost* will be equal to the number of queries answered (i.e., the length of the path traversed from the root to the leaf)

An Example Decision Tree

Guess who?

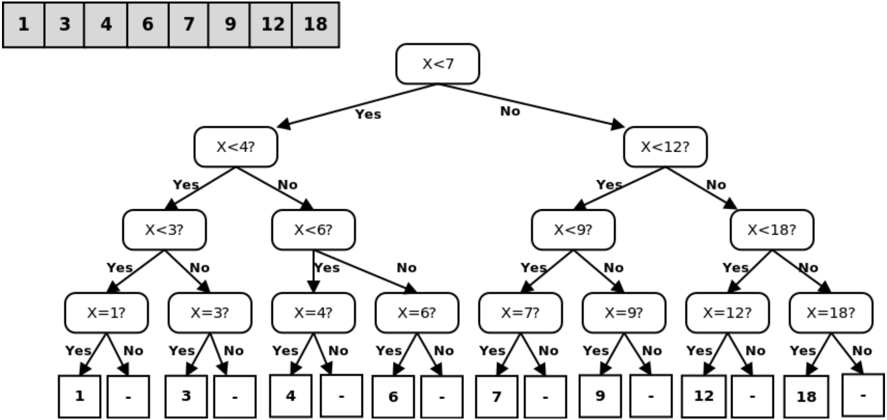
An Example Decision Tree

Guess who?



Another example with a known algorithm

Binary search



Decision Trees

Degree of the nodes

- Both examples used ***binary*** decision trees: each query has only two possible answers
- We might want to have trees with ***higher degree***: for example, we might ask 'is x smaller, equal or greater than y?', or 'are these 3 points in clockwise order, collinear or counterclockwise order?'
- A **k-ary decision tree** is one where each query has at most k possible answers. For the purposes of this lecture we will consider **decision trees with a constant k**.
- Note that the **worst case** cost is equal to the maximum depth of the decision tree

Lower Bounds - Information Theory View

- Most lower bound for decision trees follow a simple observation:

  the answers to the queries must give you enough information to specify any possible output

- This implies that **if a problem has at least N outputs, then the decision tree must have at least N leaves**!

- This is a very *powerful* implication, that can be used in many problems!

Lower Bounds - A first example

Searching for an element

- Imagine you have an array of n elements (ex: numbers)
- You want to implement a **search** function for any given x, returning the position of x of the array, or '-' if the element is not there
- There are $n+1$ possible *outputs*
- This implies that the decision tree must have $n+1$ *leaves*
- If we use a query capable of producing k answers (ex: making comparisons of type $x < y$? implies $k = 2$), then any decision tree must have *maximum depth* at least $\lceil \log_k(n+1) \rceil = \Omega(\log n)$
- A lower bound on the *cost* (runtime) is therefore $\Omega(\log n)$
- Under this computation model (decision trees), our well known *binary search* algorithm is ***optimal***! (no other algorithm can be faster)

- Wait a minute... what about **hashing** solving the searching problem in $O(1)$? Isn't this *inconsistent* with the $\Omega(\log n)$ lower bound?
- Not really, because an hash function involves a query with *more than a constant number of answers*: 'what is the hash value of x?'
- If we don't *restrict the degree* of the decision tree, even without hashing, we could easily get constant time runtime by asking the clearly *unreasonable* query: 'what is the position of x in the array?' (this is not cheating: the decision tree model allows us to ask any question about the input)
- This illustrates the **crucial importance of choosing the right model of computation**. A too powerful model may make the problem completely trivial, and a very restrictive model may even make the problem impossible

- Let's now turn our attention to the classical **sorting** problem
- Let's phrase it as: **given a sequence** $\langle x_1, x_2, \ldots, x_n \rangle$ **of** n **numbers, find a permutation** π **such that** $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$
  (without loss of generality, we may assume all numbers are different)
- We will consider a *binary* decision tree ($k = 2$) based around the following **model of computation**:

Comparison-Based Sorting Algorithm

A comparison-based sorting algorithm can only gain information about the items by comparing pairs of them. Each comparison ('is $x_i < x_j$?') returns `YES` or `NO`

Ex: Quicksort, Mergesort, Heapsort, Insertionsort, Selectionsort or Bubblesort are all comparison-based sorting algorithms.

- Our information theory argument allow us to obtain an almost immediate **lower bound**:
  - There are $n!$ possible *outputs* (all possible permutations π)
  - Any decision tree must have $n!$ *leaves*
  - This means the tree must have *depth* $\Omega(\log_2(n!))$ Let's simplify this expression:
    $log_2(n!) = log_2(n) + log_2(n-1) + log_2(n-2) + \ldots + log_2(2) = \Omega(n \log n)$
    One way to show this is to consider the first $n/2$ elements:
    $log_2(n!) \geq (n/2)\log_2(n/2) = \Omega(n \log n)$
    (we could also have used **Stirling's Approximation** to prove this)

A comparison-based sorting algorithm
must perform $\Omega(n \log n)$ comparisons in the worst case!

  - We already know optimal algorithms that are $O(n \log n)$
    Mergesort, Quicksort, Heapsort, ...

Maximum Problem and Adversarial View

- Let's first consider the **maximum** problem: **given a sequence** $\langle x_1, x_2, \ldots, x_n \rangle$ **of** n **distinct numbers, find the index** m **such that** x_m **is the largest element in the sequence**
- What algorithm would you use?
  - A simple $O(n)$ algorithm is to simply do a *linear scan* on the sequence, maintaining the current maximum. We would spend $n - 1$ comparisons.
- Intuitively, this *seems like the best possible* (we need to consider all elements?). But can we prove this is the actual *lower bound*?
- A first try using the *information theory argument* would only give us a $\Omega(\log n)$ bound because we have n possible outputs
  - This is indeed the real information-theoretical bound
  - We could ask *unreasonable* questions such as 'is the position of the maximum odd?', gaining one bit of information
  - Remember the importance of the model of computation!
- We need something more to push the lower bound to $\Omega(n)$..

Maximum Problem and Adversarial View

- Let's use the same comparison-based model as before, with $k = 2$
- We will also use an **adversary argument**:
  - Your goal is to determine the maximum of n elements (that you do not know in advance)
  - Imagine you can ask me questions about the result of comparing a pair of elements
  - I'm answering this questions with the goal of delaying as much as possible your final answer
  - How many questions do you need to ask?
- The *adversary* answers in way that is consistent with the queries, but that makes the algorithm do as much work as possible

Maximum Problem and Adversarial View

- Consider the following adversarial strategy:
  - Initially the adversary pretends that $x_i = i$ for all i and answers queries accordingly
  - Each time a question $x_i < x_j$? is asked, he *marks* x_i as an item that the algorithms knows it should not be the maximum element
  - After each comparison, *at most one element* x_i is marked
    (note that x_n is never marked)
  - If the algorithm asked less than $n-1$ questions before terminating, the adversary must have at least one other unmarked element $x_k \neq x_n$.
  - The adversary can change the value of x_k to $n+1$, making it the new maximum, without being inconsistent with the queries answered.
  - This means the algorithm *cannot be correct in both cases*: x_n is the maximum in the original input, and x_k in the modified output (both consistent with all the answers)
  - This implies **any** algorithm must make at least $n-1$ comparisons!

A comparison-based maximum algorithm must perform $\Omega(n)$ comparisons in the worst case!

Adversarial View

- The **adversarial argument** we described is very powerful and has two very important properties:
  - No algorithm can distinguish between a malicious adversary and an honest opponent that fixes an input and answers all queries truthfully
  - Most importantly, **the adversary makes *absolutely no assumptions* on about the order in which the algorithm performs comparisons**
- The adversary therefore *forces* any comparison-based algorithm to either do at least $n-1$ comparisons, or to give the wrong answer for at least one input sequence
(we are assuming it is a deterministic algorithm)
- This lower bound for the maximum problem also shows that we can be **more specific** than a simple asymptotic class. At least $n-1$ comparisons is even more tight than $\Omega(n)$ comparisons
(ex: would you be happy with $5n+3$ comparisons, when $n-1$) would suffice?