Computational Complexity

- Algorithmics vs. Complexity
  - Algorithmics
    - Given a problem, we can prove that the problem can be solved in a time of O(f(n)) by giving and analyzing an algorithm
    - We'd like to reduce f(n) as much as possible
  - Complexity
    - It tells us any algorithm capable of solving our problem correctly takes a time $\Omega(g(n))$
    - Now g(n) is a lower bound on the complexity of the problem
    - If $f(n) \in \Theta(g(n))$, we're satisfied

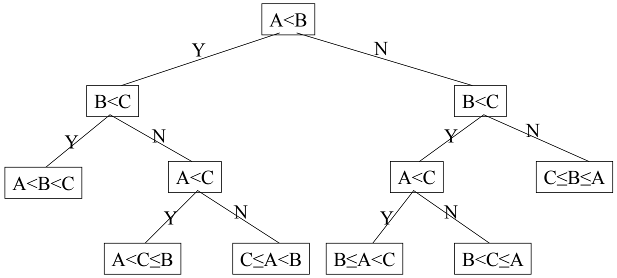
Topics

- Arguing a lower bound
  - Information-theoretic arguments
  - Adversary arguments
- Proof equivalence of complexity or compare complexity
  - Linear reductions
  - NP-completeness

Information-theoretic arguments

- Particularly applies to those problem involving comparisons
- Uses a decision tree to represent the working process of an algorithm on all possible data of a given size
  - A decision tree is a binary tree where
    - Each internal node contains a test on the data
    - Each leaf contains an output, called *verdict*
    - A *trip* through the tree starts from the root and recursively goes to the left subtree or the right subtree depending on whether the answer to the root is "yes" or "no"
    - The trip ends when it reaches a leaf (verdict)

A decision tree for sorting



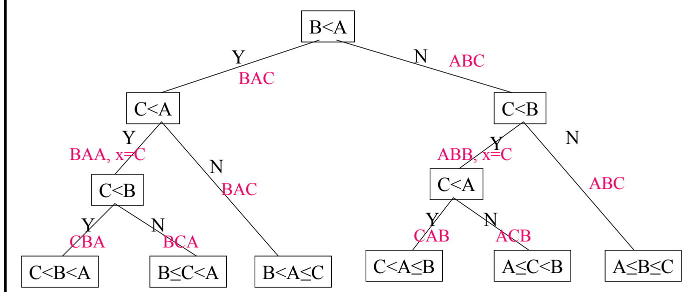
Note that the number of leaves = The number of possible outputs

Insertion Sort

```
void insertionSort(int A[], int n)
{ // array index starts from 1 here
   int i, j,  x;

    for (i=2; i<=n; i++) {
       x =A[i];
       j = i-1;
       while (j>0 && x<A[j]) {
         A[j+1] = A[j];
         j--;
       }
       A[j+1] = x;
    }
}
```

A three-item insertion sort decision tree



Observations

- The number possible outputs = The number of leaves (verdicts)
- The worst-case time is the height of the tree
- The average time is the average depth of leaves assuming equal distribution

Theorem

- Any binary tree with k leaves has an average height of at least $lg\ k$
  - Let $h(k)$ be smallest possible total depths for a tree of k leaves

$$h(k) = \begin{cases} 0 & k \le 1 \\ \min\limits_{1 \le i \le k-1} (h(i) + h(k-i) + k) & otherwise \end{cases}$$

- Any comparison-based algorithm takes a worst case time and average case time $\Omega(n \log n)$

Adversary arguments

- Start the algorithm on an input that is initially unspecified except for its size
- When the algorithm probes the input, the malevolent daemon, the adversary, answers in a way that will force the algorithm to work hard
  - The daemon's goal is to keep the algorithm uncertain of the correct answer as long as possible
  - Constraint: the daemon's answers must be consistent—there always exists at least one input that could cause the algorithm to see exactly the same answers on its probes

Finding the maximum of an array

- A simple algorithm takes a time of $\Theta(n)$
- Using decision tree, we get
  - Any comparison based algorithm to find the maximum must perform at least $\lceil \lg n \rceil$ comparisons in worst case
- Can we find a tighter lower bound?
  - We use adversary arguments to show $n-1$ comparisons are necessary

Find the median

- For a comparison-based algorithm, we can easily argue a lower bound of $\left\lceil \frac{n}{2} \right\rceil$ comparisons
- Can we find a tighter bound?
  - We can prove a lower bound of $\frac{3(n-1)}{2}$ comparisons

Adversary arguments

- Assume n is odd and n ≥ 3.
- Initially, the daemon sets each entry to "uninitialized"
  - As the algorithm makes comparisons, the daemon set values between 1 and n (low) or between 3n+1 to 4n (high).
  - The daemon makes sure #low items = #high items

Comparison

- When T[i] is asked to compare to T[j]
  1. T[i] and T[j] are both unintialized
     – T[i] is sets to i and T[j] is set to 3n+j.
  2. One is uninitialized
     1. If it is the only one uninitialized item left, set its value to 2n which becomes *provisional median*.
     2. If T[i] is low, set T[j] to high value 3n+j. And add one uninitialized item T[k] to low, i.e., set T[k] to k.
     3. If T[j] is low, set T[i] to high value 3n+i. And add one uninitialized item T[k] to low, i.e., set T[k] to k.
     4. If T[i] is high, set T[j] to low value j. And add one uninitialized item T[k] to high, i.e., set T[k] to 3n+k.
     5. If T[j] is high, set T[i] to low value i. And add one uninitialized item T[k] to high, i.e., set T[k] to 3n+k.
  3. Both are initialized

Arguments

- Both are initialized
  - If both are low or one low and one provisional median, then the smaller has *lost* a comparison
  - If both are high or one high and one provisional median, then the larger has *lost* a comparison
  - Otherwise, no one lost comparison
- Arguments: if less than 3(n-1)/2 comparisons
  - (n-1)/2 comparisons needed to initialize all values
  - Less than n-1 comparisons for initialized values
    - Less than n-1 values lost comparisons
    - At least one item in addition to the provisional median never lost a comparison