Limitations of Algorithm Power
Lower bound finding

Dr. Bibhudatta Sahoo
Communication & Computing Group
# CS215, Department of CSE, NIT Rourkela
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Why learn about lower bounds

- **Know your limit**

◆ we always try to make algorithms faster, but if there is a limit that you cannot exceed, you want to know .

- **Approach the limit**

◆ Once you have an understanding about of limit of the algorithm’s performance, you get insights about how to approach that limit.

Lower bound : Introduction

- A lower bound for a problem is the worst-case running time of the best possible algorithm for that problem.
- A lower bound on an *algorithm* is just a big-Omega bound on its worst-case running time.
- A lower bound on a *problem* is a big-Omega bound on the worst-case running time of *any* algorithm that solves the problem.
- Lower bound: an estimate on a minimum amount of work needed to solve a given problem.
- Lower bound can be an exact count an efficiency class (**Ω**).
- Tight lower bound: there exists an algorithm with the same efficiency as the lower bound.
- A tight bound implies that both the lower and the upper bound for the computational complexity of an algorithm are the same.

Lower Bounds

Lower bound: an estimate on a minimum amount of work needed to solve a given problem

Examples:

- number of comparisons needed to find the largest element in a set of *n* numbers
- number of comparisons needed to sort an array of size *n*
- number of comparisons necessary for searching in a sorted array
- number of multiplications needed to multiply two *n*-by-*n* matrices

Lower Bounds

Lower bound can be

- an exact count
- an efficiency class (Ω)

- **Tight lower bound**: there exists an algorithm with the same efficiency as the lower bound

Problem	Lower bound	Tightness
sorting (comparison-based)	$\Omega(n\log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n\log n)$	yes
n-digit integer multiplication	$\Omega(n)$	unknown
multiplication of n-by-n matrices	$\Omega(n^2)$	unknown

- Can we do better? Why not?
- Lower bounds prove that we cannot hope for a better algorithm, no matter how smart we are.
- Only very few lower bound proofs are known
- Most notorious open problems in Theoretical Computer Science are related to **proving lower bounds** for very important problems

Methods for Establishing Lower Bounds

- trivial lower bounds
- information-theoretic arguments (decision trees)
- adversary arguments
- problem reduction

1: Trivial Lower Bounds

Trivial lower bounds: based on counting the number of items that must be processed in input and generated as output

Examples

- finding max element - n steps or n/2 comparisons
- polynomial evaluation
- sorting
- element uniqueness
- Hamiltonian circuit existence

Conclusions

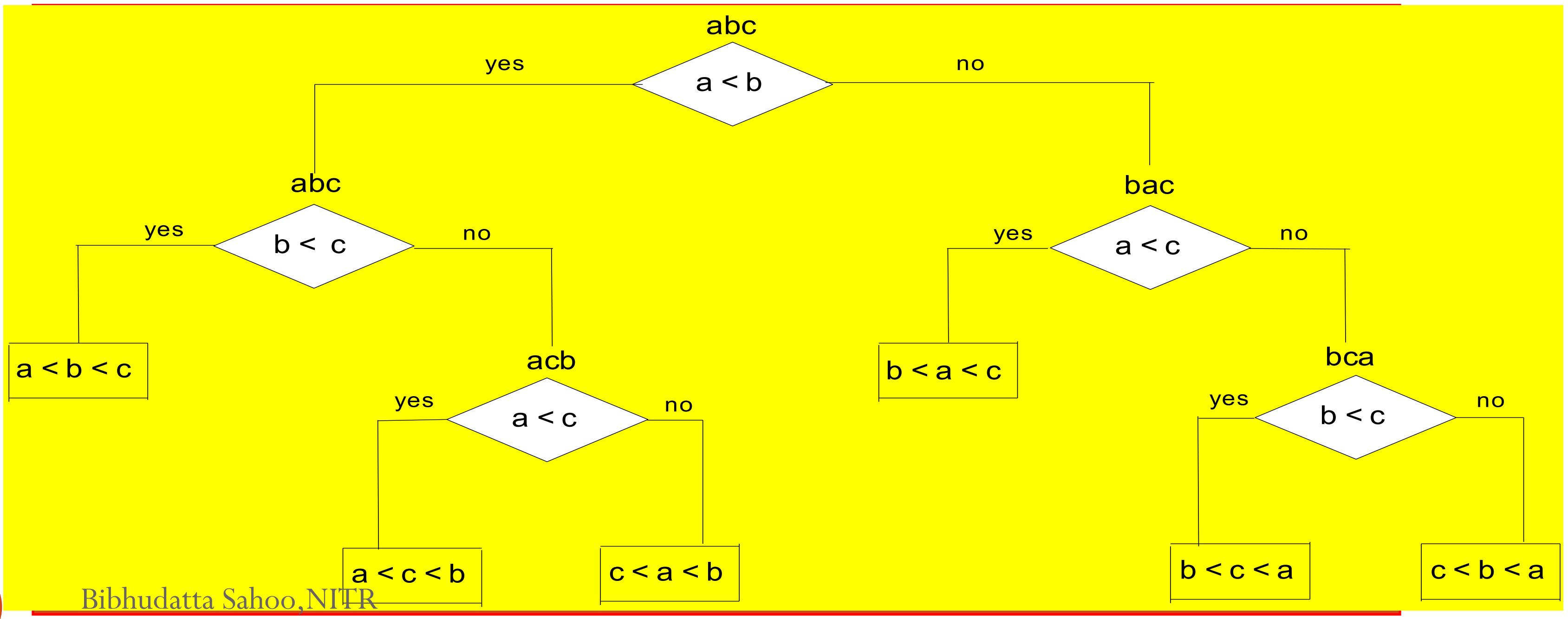
- may and may not be useful
- be careful in deciding how many elements must be processed

2: Decision Trees

Decision tree — a convenient model of algorithms involving comparisons in which:

- internal nodes represent comparisons
- leaves represent outcomes (or input cases)

Decision tree for 3-element insertion sort



Decision Trees and Sorting Algorithms

- Any comparison-based sorting algorithm can be represented by a decision tree (for each fixed n)
- Number of leaves (outcomes) $\geq n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$ for any comparison-based sorting algorithm, since the longest path represents the worst case and its length is the height
- $\lceil \log_2 n! \rceil \approx n \log_2 n$ (by Sterling approximation)
- This lower bound is tight (mergesort or heapsort)

Ex. Prove that 5 (or 7) comparisons are necessary and sufficient for sorting 4 keys (or 5 keys, respectively).

3: Adversary Arguments

Adversary argument:

- It's a game between the adversary and the (unknown) algorithm.
- The adversary has the input and the algorithm asks questions to the adversary about the input.
- The adversary tries to make the algorithm work the hardest by adjusting the input (consistently).
- It wins the "game" after the lower bound time (lower bound proven) if it is able to come up with two different inputs.

Example 1: "Guessing" a number between 1 and n using yes/no questions (Is it larger than x?)

Adversary: Puts the number in a larger of the two subsets generated by last question

3: Adversary Arguments

Example 2: Merging two sorted lists of size n

$$a_1 < a_2 < \ldots < a_n \text{ and } b_1 < b_2 < \ldots < b_n$$

Adversary: Keep the ordering $\boldsymbol{b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n}$ in mind and answer comparisons consistently

Claim: Any algorithm requires at least $2n$-1 comparisons to output the above ordering (because it has to compare each pair of adjacent elements in the ordering)

- **Ex: Design an adversary to prove that finding the smallest element in a set of n elements requires at least n-1 comparisons**

4: Lower Bounds by Problem Reduction

Fact: If problem *Q* can be "reduced" to problem *P*, then *Q* is at least as easy as *P*, or equivalent, *P* is at least as hard as *Q*.

Reduction from *Q* to *P*: Design an algorithm for *Q* using an algorithm for *P* as a subroutine.

Idea: If problem *P* is at least as hard as problem *Q*, then a lower bound for *Q* is also a lower bound for *P*.

Hence, find problem *Q* with a known lower bound that can be *reduced to* problem *P* in question.

Example: *P* is finding MST for *n* points in Cartesian plane, and *Q* is element uniqueness problem (known to be in $\Omega(n\log n)$)

4: Lower Bounds by Problem Reduction

- **Example:** P is finding MST for n points in Cartesian plane, and Q is element uniqueness problem (known to be in $\Omega(n\log n)$)
- **Reduction from Q to P: Given a set $X = \{x1, \ldots, xn\}$ of numbers (*i.e.* an instance of the uniqueness problem), we form an instance of MST in the Cartesian plane: $Y = \{(0,x1), \ldots, (0,xn)\}$. Then, from an MST for Y we can easily (*i.e.* in linear time) determine if the elements in X are unique.**

Lower bound

- A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.

☆ **worst case lower bound**

☆ **average case lower bound**

- The lower bound for a problem is not unique.
  - e.g. Ω(1), Ω(n), Ω(n log n) are all lower bounds for sorting.
  - (Ω(1), Ω(n) are trivial)

Lower bound of sorting

- At present, if the highest lower bound of a problem is Ω(n log n) and the time complexity of the best algorithm is O(n^2).
  - We may try to find a higher lower bound.
  - We may try to find a better algorithm.
  - Both of the lower bound and the algorithm may be improved.
- If the present lower bound is Ω(n log n) and there is an algorithm with time complexity O(n log n), then the algorithm is optimal.

The worst case lower bound of sorting

6 permutations for 3 data elements

a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Straight insertion sort: Finding lowerbound

- input data: (2, 3, 1)

  (1) $a_1 : a_2$

  (2) $a_2 : a_3$, $a_2 \leftrightarrow a_3$

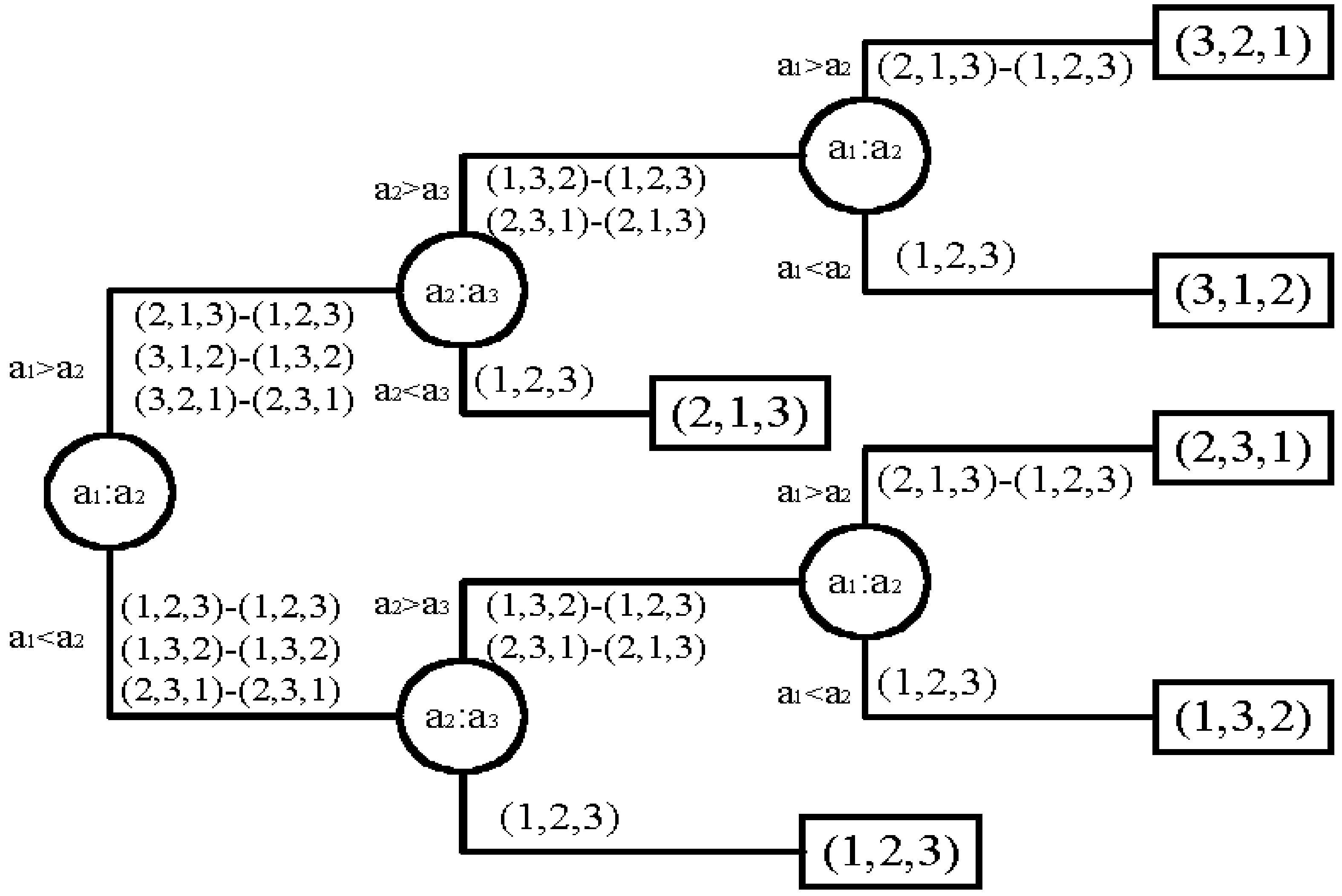
  (3) $a_1 : a_2$, $a_1 \leftrightarrow a_2$

- input data: (2, 1, 3)

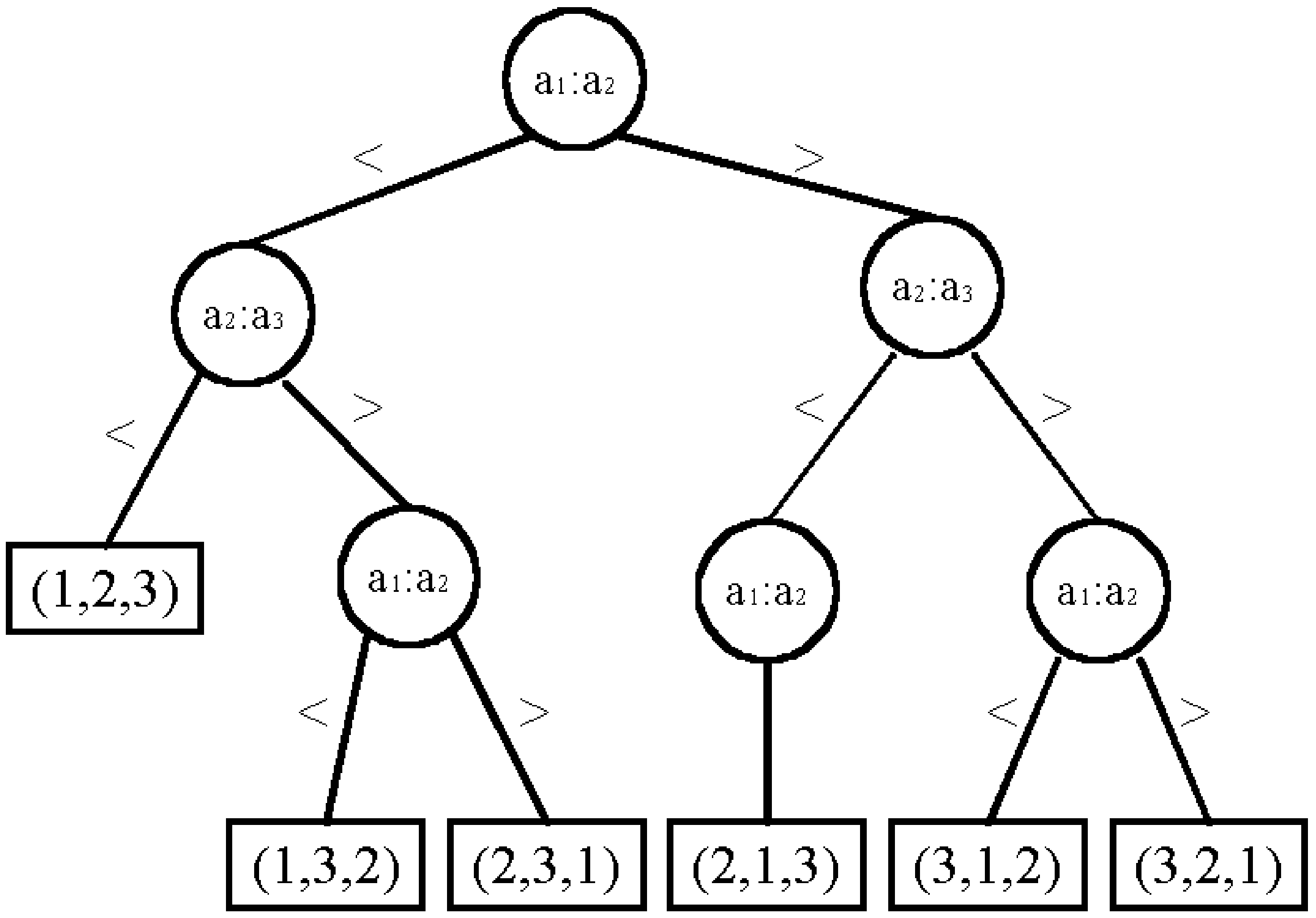
  (1)$a_1 : a_2$, $a_1 \leftrightarrow a_2$

  (2)$a_2 : a_3$

Decision tree for straight insertion sort



Decision tree for bubble sort



Lower bound of sorting

- To find the lower bound, we have to find the smallest depth of a binary tree.
- n! distinct permutations

  n! leaf nodes in the binary decision tree.
- balanced tree has the smallest depth:

  $\lceil \log(n!) \rceil = \Omega(n \log n)$

  <u>lower bound for sorting: $\Omega(n \log n)$</u>

Method 1:

$$\log(n!) = \log(n(n-1)\cdots 1)$$

$$= \log 2 + \log 3 + \cdots + \log n$$

$$> \int_1^n \log x\,dx$$

$$= \log e \int_1^n \ln x\,dx$$

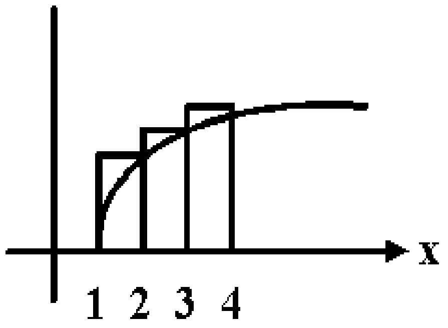
$$= \log e[x \ln x - x]_1^n$$

$$= \log e(n \ln n - n + 1)$$

$$= n \log n - n \log e + 1.44$$

$$\geq n \log n - 1.44n$$

$$= \Omega(n \log n)$$



Method 2:

- Stirling approximation:
  - $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$
  - $\log n! \approx \log \sqrt{2\pi} + \frac{1}{2}\log n + n \log \frac{n}{e} \approx n \log n \approx \Omega(n \log n)$

n	n!	S_n
1	1	0.922
2	2	1.919
3	6	5.825
4	24	23.447
5	120	118.02
6	720	707.39
10	3,628,800	3,598,600
20	2.433×10^{18}	2.423×10^{18}
100	9.333×10^{157}	9.328×10^{157}

Sorting Lower Bound in the Comparison Model

Theorem: Any decision tree sorting n elements has height Ω(n log n).

Proof:

– Assume elements are the (distinct) numbers 1 through n

– There must be n! leaves (one for each of the n! permutations of n elements)

– Tree of height h has at most 2^h leaves

$$
\begin{aligned}
2^h \geq n! \Rightarrow h &\geq \log(n!) \\
&= \log(n(n-1)(n-2)\cdots(2)) \\
&= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2
\end{aligned}
$$

Sorting Lower Bound in the Comparison Model

$$
\begin{aligned}
2^h \geq n! \Rightarrow h &\geq \log(n!) \\
&= \log(n(n-1)(n-2)\cdots(2)) \\
&= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 \\
&= \sum_{i=2}^{n} \log i \\
&= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq 0 + \sum_{i=n/2}^{n} \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \log \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}
$$

Algorithm 2-7 Heapsort

Input : $A(1), A(2), \ldots, A(n)$

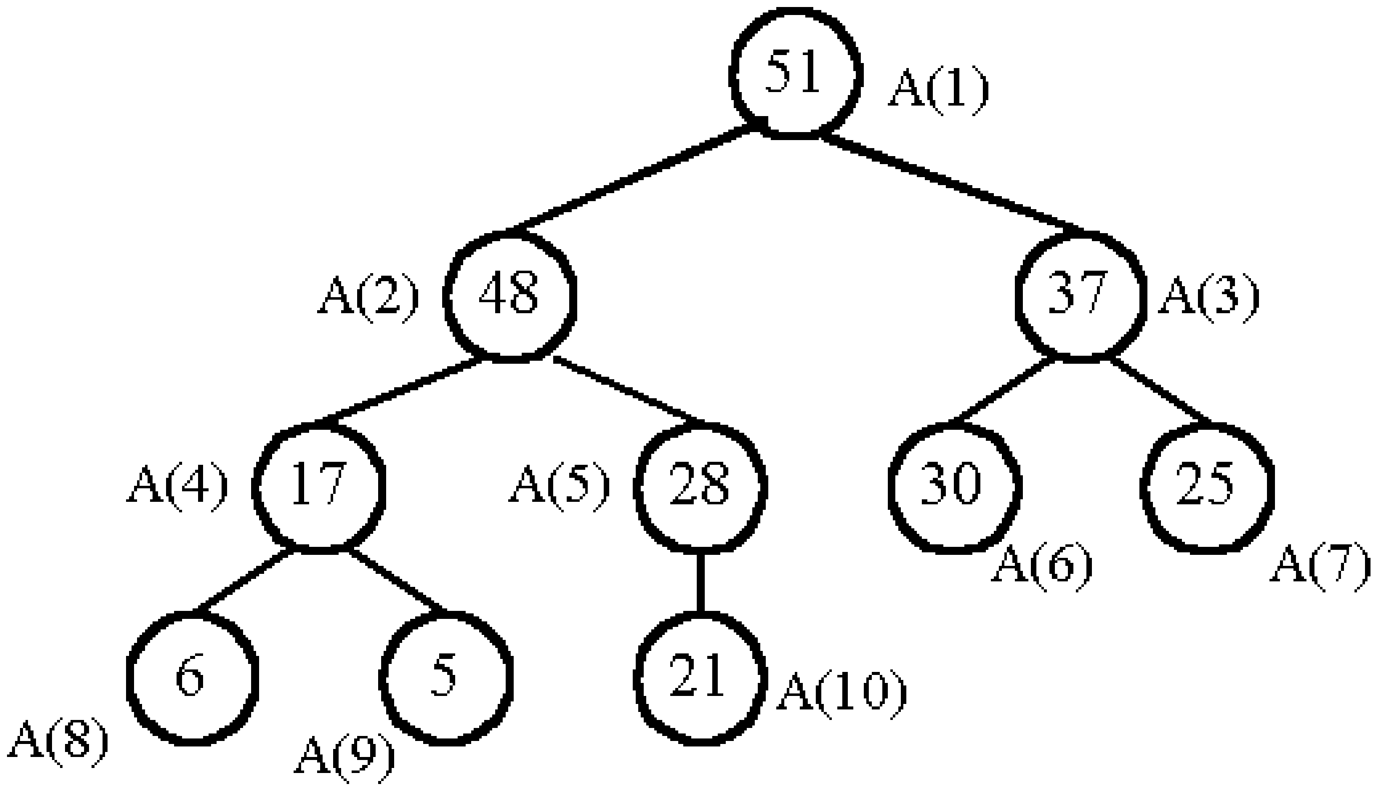
where each $A(i)$ is a node of a heap already constructed.

Ouput : The sorted sequence of $A(i)$'s.

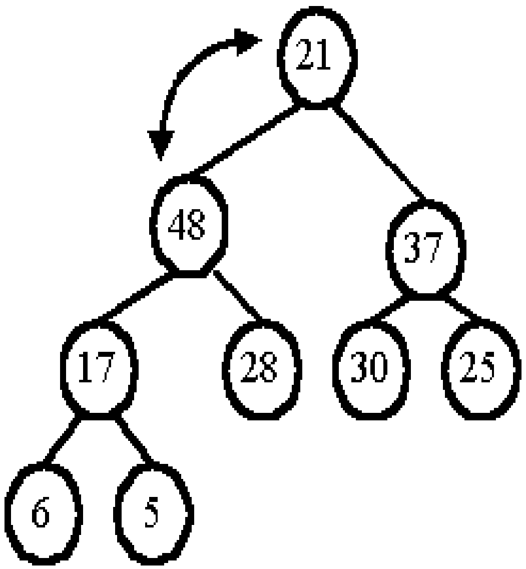
```
For $i := n$ down to 2 do
Begin
    Output $A(1)$
    $A(1) := A(i)$
    Delete $A(i)$
    Restore$(1, i\text{-}1)$
End
Output $A(1)$
```

Heapsort: an optimal sorting algorithm

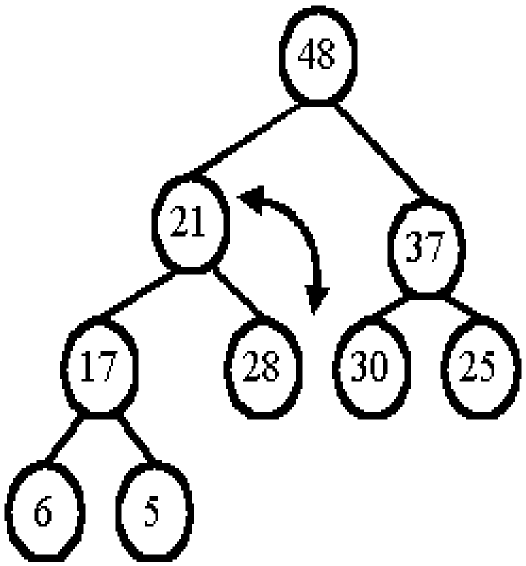
- **A heap :** parent ≥ son



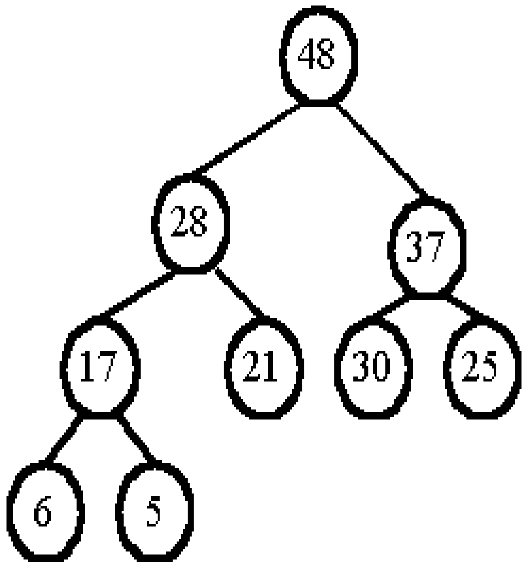
- output the maximum and restore:



(a)



(b)

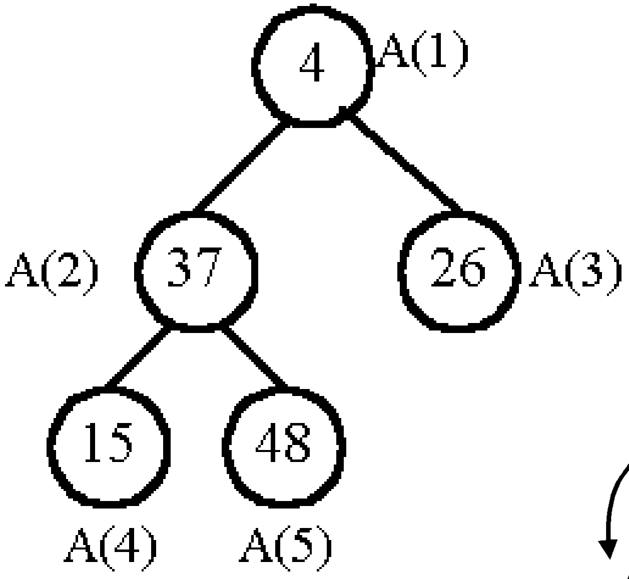


(c)

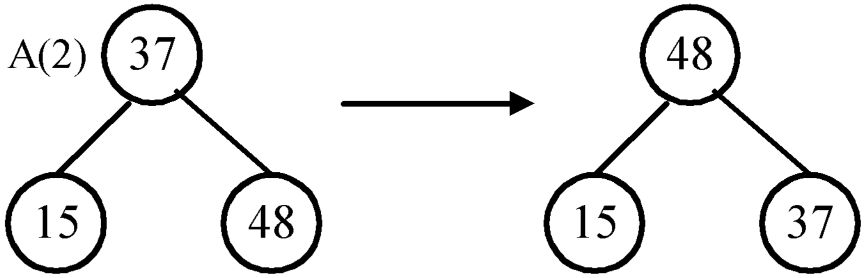
{

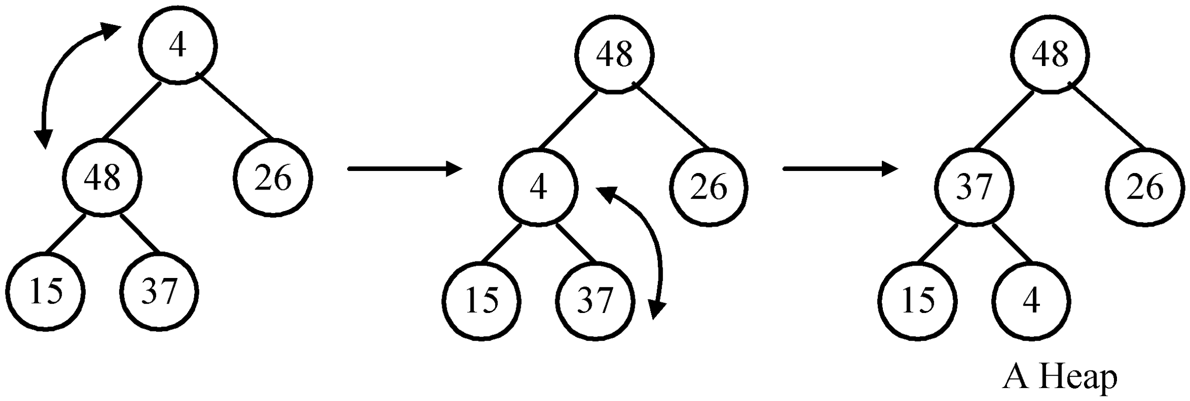
Phase 1: construction

- input data: 4, 37, 26, 15, 48

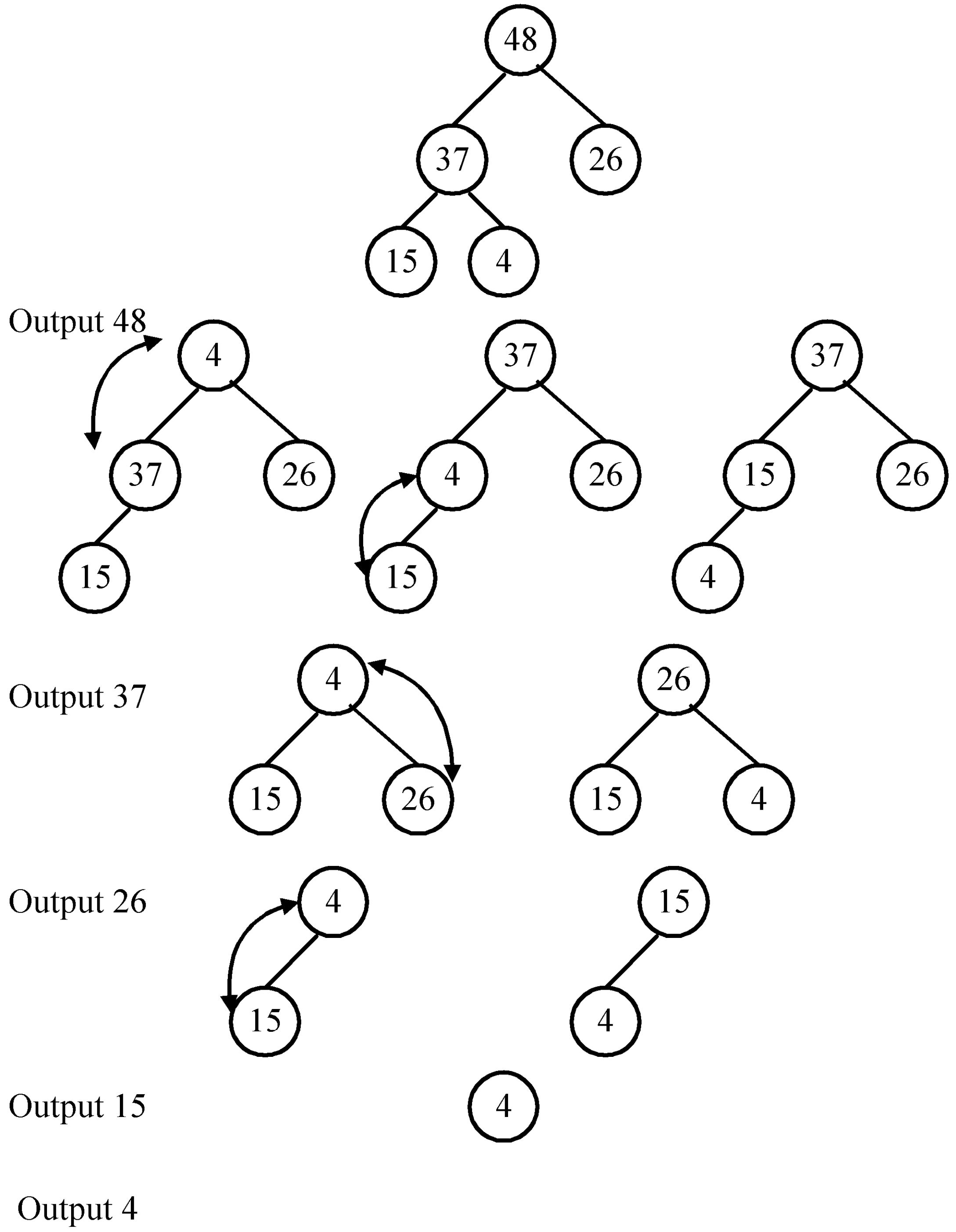


- restore the subtree rooted at A(2):





Phase 2: output



Implementation

- using a linear array

  not a binary tree.
  - The sons of A(h) are A(2h) and A(2h+1).
- time complexity: O(n log n)

Time complexity
Phase 1: construction

$d = \lfloor \log n \rfloor$: depth

# of comparisons is at most:

$$\sum_{L=0}^{d-1} 2(d-L)2^L$$

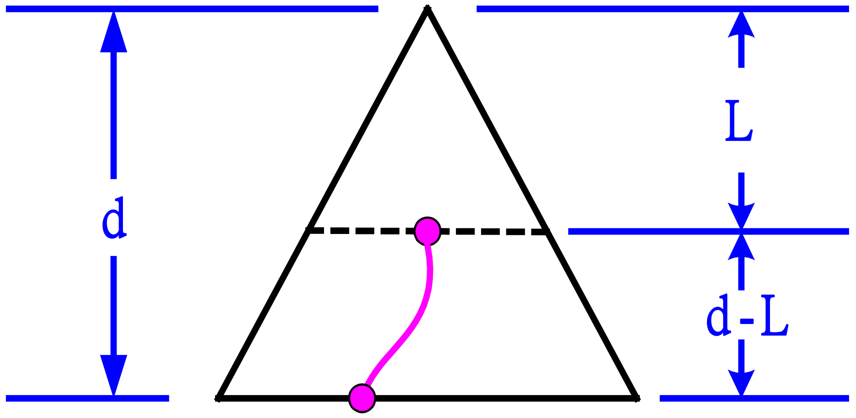
$$=2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L2^{L-1}$$

$$\left(\sum_{L=0}^{k} L2^{L-1} = 2^k(k-1)+1 \right)$$

$$=2d(2^d-1) - 4(2^{d-1}(d-1-1)+1)$$

$$\vdots$$

$$= cn - 2\lfloor \log n \rfloor - 4, \quad 2 \le c \le 4$$



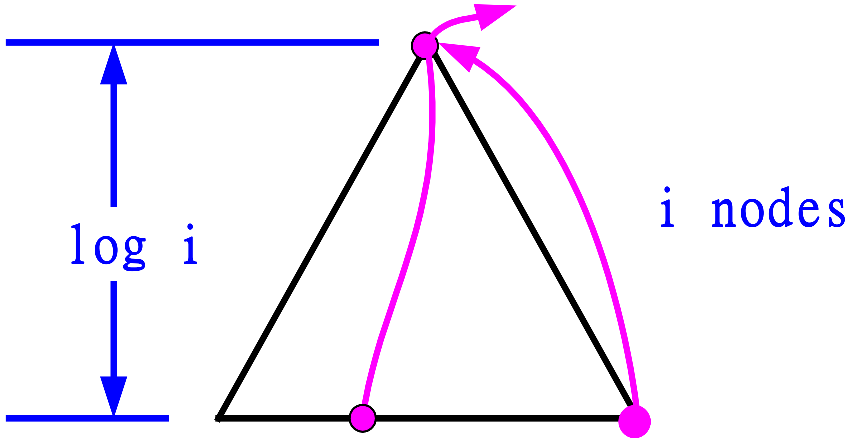
Phase 2: output, Time complexity

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor$$

$$= \quad :$$

$$=2n\lfloor \log n \rfloor - 4cn + 4, \quad 2 \le c \le 4$$

$$=O(n \log n)$$



Average case lower bound of sorting

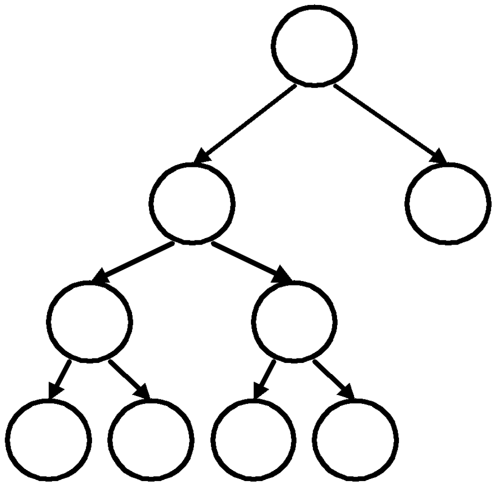
- By binary decision tree
- The average time complexity of a sorting algorithm:

$$\frac{\text{the external path length of the binary tree}}{n!}$$

- The external path length is minimized if the tree is balanced.

  (all leaf nodes on level d or level d−1)

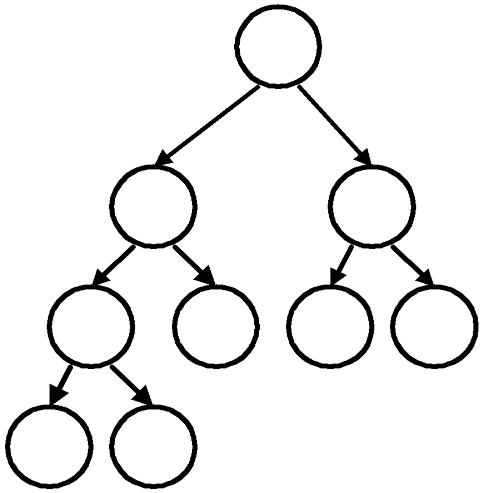
Average case lower bound of sorting



unbalanced

external path length

= 4·3 + 1 = 13



balanced

external path length

= 2·3+3·2 = 12

Compute the min external path length

1. Depth of balanced binary tree with c leaf nodes:
   $d = \lceil \log c \rceil$
   Leaf nodes can appear only on level d or d–1.
2. x_1 leaf nodes on level d–1
   x_2 leaf nodes on level d
   - $x_1 + x_2 = c$
   - $x_1 + \frac{x_2}{2} = 2^{d-1}$

$$\Rightarrow \quad x_1 = 2^d - c$$
$$x_2 = 2(c - 2^{d-1})$$

3. External path length:

$$M = x_1(d-1) + x_2 d$$
$$= (2^d - 1)(d-1) + 2(c - 2^{d-1})d$$
$$= c(d-1) + 2(c - 2^{d-1}),\ \ d - 1 = \lfloor \log c \rfloor$$
$$= c\lfloor \log c \rfloor + 2(c - 2^{\lfloor \log c \rfloor})$$

4. $c = n!$

$$M = n!\lfloor \log n! \rfloor + 2(n! - 2^{\lfloor \log n! \rfloor})$$
$$M/n! = \lfloor \log n! \rfloor + 2$$
$$= \lfloor \log n! \rfloor + c,\ \ 0 \le c \le 1$$
$$= \Omega(n \log n)$$

Average case lower bound of sorting: $\Omega(n \log n)$

Quicksort & Heapsort

- Quicksort is optimal in the average case.
  (Θ(n log n) in average)
- (i)worst case time complexity of heapsort is
  Θ(n log n)
  (ii)average case lower bound: Ω(n log n)
  - average case time complexity of heapsort is Θ(n log n)
  - Heapsort is optimal in the average case.

Improving a lower bound through oracles

- Problem P: merge two sorted sequences A and B with lengths m and n.

(1) <u>Binary decision tree:</u>

There are $\binom{m+n}{n}$ ways !

$\binom{m+n}{n}$ leaf nodes in the binary tree.

$\Rightarrow$ The lower bound for merging:

$$\left\lceil \log \binom{m+n}{n} \right\rceil \leq m + n - 1 \text{ (conventional merging)}$$

- When m = n

$$\log\binom{m+n}{n} = \log\frac{(2m)!}{(m!)^2} = \log((2m)!) - 2\log m!$$

- Using Stirling approximation

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$\log\binom{m+n}{n} \approx 2m - \frac{1}{2}\log m + O(1)$$

- Optimal algorithm: 2m – 1 comparisons

$$\log\binom{m+n}{n} < 2m - 1$$

(2) Oracle:

- The oracle tries its best to cause the algorithm to work as hard as it might. (to give a very hard data set)
- Sorted sequences:
  - A: $a_1 < a_2 < \ldots < a_m$
  - B: $b_1 < b_2 < \ldots < b_m$
- The very hard case:
  - $a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$

- We must compare:

  $a_1 : b_1$
  $b_1 : a_2$
  $a_2 : b_2$
  $\vdots$
  $b_{m-1} : a_{m-1}$
  $a_m : b_m$

- Otherwise, we may get a wrong result for some input data.

  e.g. If b_1 and a_2 are not compared, we can not distinguish

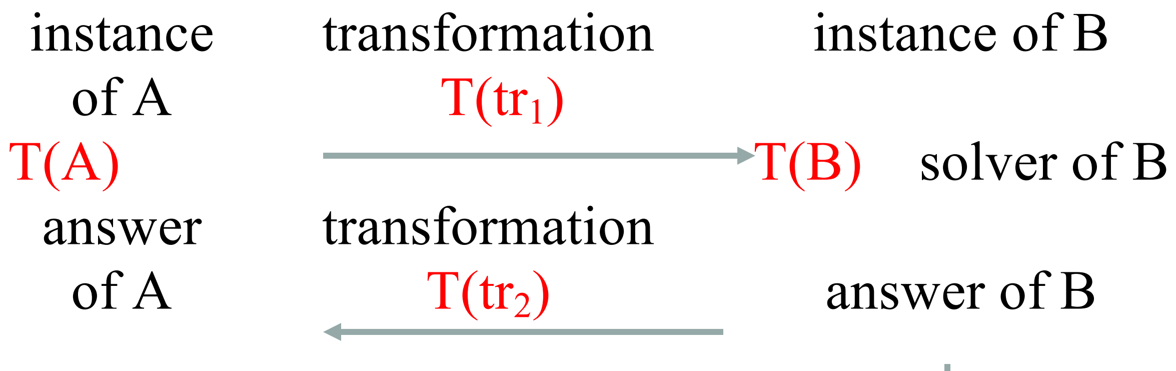
  $a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$ and

  $a_1 < a_2 < b_1 < b_2 < \ldots < a_m < b_m$

- Thus, at least $2m-1$ comparisons are required.
- The conventional merging algorithm is optimal for $m = n$.

Finding lower bound by problem transformation

- Problem A reduces to problem B ($A \propto B$)
  - iff A can be solved by using any algorithm which solves B.
  - If $A \propto B$, B is more difficult.



- Note: $T(tr_1) + T(tr_2) < T(B)$

$$T(A) \le T(tr_1) + T(tr_2) + T(B) \sim O(T(B))$$

The lower bound of the convex hull problem

- sorting $\propto$ convex hull

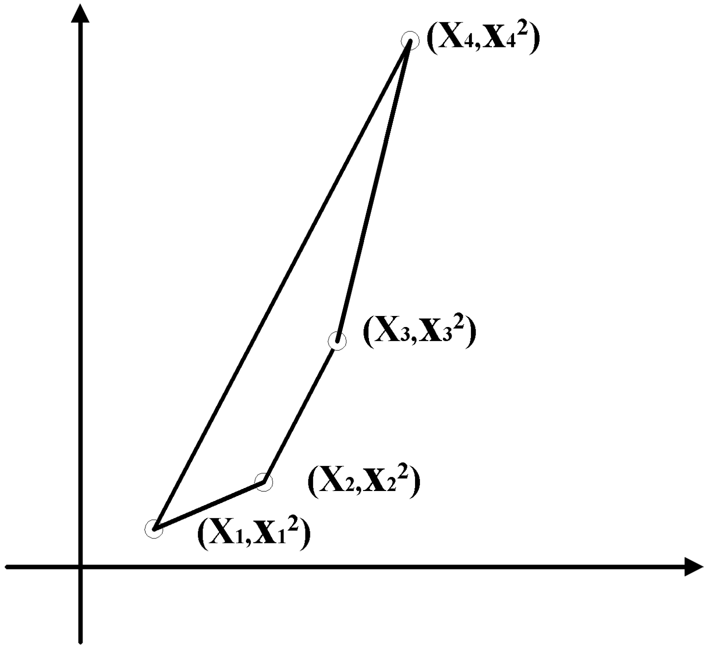
  A B

- an instance of A: $(x_1, x_2, \ldots, x_n)$

  $\downarrow$transformation

  an instance of B: $\{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$

  assume: $x_1 < x_2 < \ldots < x_n$



- If the convex hull problem can be solved, we can also solve the sorting problem.
    - The lower bound of sorting: $\Omega(n \log n)$
- The lower bound of the convex hull problem: $\Omega(n \log n)$

The lower bound of the Euclidean minimal spanning tree (MST) problem

- sorting $\propto$ Euclidean MST

  A   B

- an instance of A: $(x_1, x_2, \ldots, x_n)$

  $\downarrow$ transformation

  an instance of B: $\{(x_1, 0), (x_2, 0), \ldots, (x_n, 0)\}$

  - Assume $x_1 < x_2 < x_3 < \ldots < x_n$
  - $\Leftrightarrow$ there is an edge between $(x_i, 0)$ and $(x_{i+1}, 0)$ in the MST, where $1 \le i \le n-1$

- If the Euclidean MST problem can be solved, we can also solve the sorting problem.
  - The lower bound of sorting: Ω(n log n)
- The lower bound of the Euclidean MST problem: Ω(n log n)

LOWER BOUND THEORY

Searching ordered lists
with Comparison Based Algorithms

Comparison-Based Algorithms

- **Comparison-Based Algorithms**: Information can be gained only by comparing key–to–element, or element–to–element (in some problems).

- **Given:** An integer n, a key, and an ordered list of n values.
- **Question:** Is the key in the list and, if so, at what index?
- We have an algorithm. We don't know what it is, or how it works. It accepts n, a key and a list on n values. That's it.

1) It must calculate an index for the first compare based solely upon n since it has not yet compared the key against anything, i.e., it has not yet obtained any additional information. Notice, this means for a fixed value of n, the position of the first compare is fixed for all data sets (of size n).

2) The following is repeated until the key is found, or until it is determined that no location contains the key:

The key is compared against the item at the specified index.

a)If they are equal, the algorithm halts.

b)If the key is less, then it incorporates this information and computes a new index.

c)If the key is greater, then it incorporates this information and computes a new index

Observations

- There are no rules about how this must be done. In fact we want to leave it wide open so that we are not eliminating any possible algorithm.

- **Observation 0**: Every comparison–based search algorithm has it's own set of decision tree's (one for each value of n) – even if we don't know what or how it does its task, we know it has one for each n and are pretty much like the one described above.
- **Observation 1:** For any decision tree and any root–leaf path, there is a set of date which will force the algorithm to take that path. The number of compares with a given data set (key and n values) is the number of nodes in the "forced" root–leaf path.
- **Observation 2:** The longest root–leaf path is the "worst case" running time of the algorithm.
- **Observation 3.** For any position i of {1, 2, . . . , n}, some data set contains the key in that position. So every algorithm must have a compare for every index, that is, the decision tree must have at least one node for each position.

- Therefore, all decision trees–for the search problem–must have at least n nodes in them
- All binary trees with n nodes have a root–leaf path with at least $\lceil \log_2(n+1) \rceil$ nodes (you can verify this by induction).
- Thus, all decision trees defined by search algorithms on **n** items, have a path requiring $\lceil \log_2(n+1) \rceil$ compares.
- Therefore, **the best any comparison–based search algorithm can hope to do** is $\log_2 n \approx \lceil \log_2(n+1) \rceil$.
- This is the *comparison–based lower bound for the problem of searching an ordered list* of n items for a given key

Comparison based sorting

- Typically, in comparison–based sorting, we will compare values in two locations and, depending upon which is greater, we might

  1) do nothing,

  2) exchange the two values,

  3) move one of the values to a third position,

  4) leave a reference (pointer) at one of the positions,

  5) etc.

Observation

- Each sorting algorithm has its own decision tree.
- Some differences occur: Leaf nodes are the locations which indicate "the list is now sorted."
- Internal nodes simply represent comparisons on the path to a leaf node.

examples relate to proving lower bounds for comparison-based algorithms

Using both decision

trees and an adversary style proof

Example 1: Question

- Suppose we are given a sorted list of n numbers, $X = x_1 \leq x_2 \leq . . . \leq x_n$, and we are asked to check whether or not there are any duplicates in the list. We are limited to algorithms which compare pairs of list elements (with a procedure Compare(i, j) which returns $<$, $>$ or $=$, depending on the values of x_i and x_j).
- Assume the algorithm returns either distinct (if there are no duplicates) or ”i-th element equals j-th element” if $x_i = x_j$.
- Note that if there are multiple ties, any one can be reported. Prove **a good lower bound** for the number of calls to Compare to solve this problem.

Lower bound, finding using an adversary argument

Solution:

- Before selecting a proposed lower bound, and using an adversary argument to prove it, it is useful to determine the lower bound on the simplest and most straightforward algorithm.
- The naive approach simply walks the list and compares the current element to the next element, looking for duplicates.
- This takes **$(n - 1)$** comparisons. Thus, we will try to prove (using an adversary argument) that **$(n - 1)$** is the best (largest) lower bound possible.

Solution 1: cont…

1. Assume there exists an algorithm A which runs in $(n-2)$ comparisons which correctly finds duplicates in an ordered list of size n.
2. Let X be a list such that $x_i = 2i$ for $i = 1$ to n. **Note that all elements are unique.**
3. Run algorithm A on input X. Since it only takes $(n-2)$ comparisons, there is at least 1 element which is not compared to its next element.
4. Find that element $xi = 2i$ and set $x_{i+1} = 2i$. Remember that previously, $x_{i+1} = 2(i + 1)$.
5. Rerun the algorithm. The algorithm will report no duplicates, as it did before, but it will be wrong.

   Thus, there cannot exist any correct comparison-based algorithm that finds duplicates in a sorted list in **less than $(n - 1)$ comparisons**. $\square$

Example 2

- Give a $(2n - 1)$ lower bound on the number of comparisons to merge two sorted lists $A_1 < A_2 < \ldots < A_n$ and $B_1 < B_2 < \ldots < B_n$ into a single sorted list.
- **Solution**: It is useful to notice that with **(2n − 1)** comparisons, every element in the final merged list of 2n elements has been compared to its next element (except the last element).

Thus, our **adversarial argument** is as follows:

1. Assume there exists an algorithm A which runs in **(2n−2)** comparisons or less which correctly merges two sorted lists of size **n**.

Solution 2: cont…

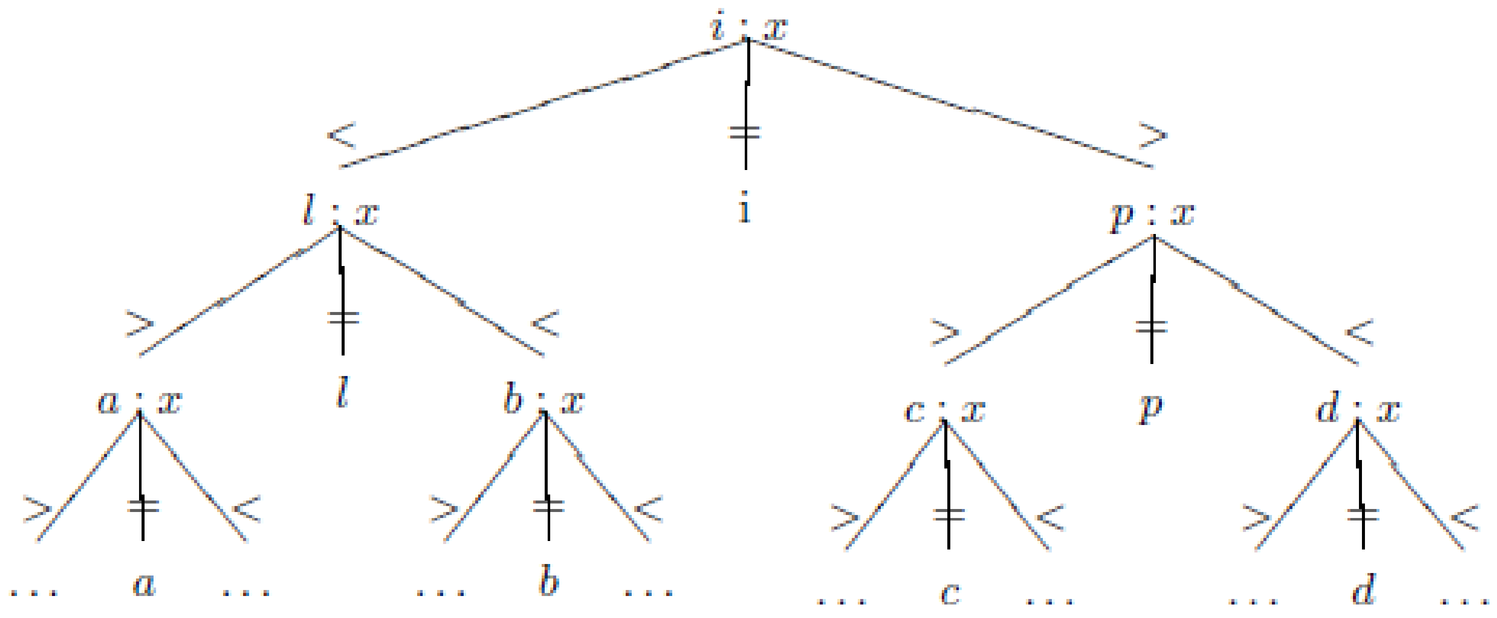
2. Let X be a list such that $x_i = 2i - 1$ for $i = 1$ to n. Let Y be a list such that $y_i = 2i$ for $i = 1$ to n. Note that all elements in both lists are unique.
3. Run algorithm A on input X and Y . Since it only takes $(2n - 2)$ comparisons, there must be at least 1 element x_i in the final merged list which has not been compared to both y_i and y_{i+1}.
4. There are two cases. Either x_i has not been compared to y_i or it has not been compared to y_{i+1}. We will show the case where xi has not been compared to y_i.

   The other case is similar. Find that element $x_i = 2i - 1$ and set $x_i = 2i$ and $y_i = 2i - 1$. This does not change the order of either list.
5. Rerun the algorithm. Since y_i and x_i were not compared, and the order is now different, the algorithm will sort the output in the same way as before, but now it will be wrong.

- Thus, there cannot exist any correct comparison-based algorithm that merges two sorted lists of size n in **less than $(2n - 1)$** comparisons.☐

Example 3

- Suppose we are given a sorted list A1 < A2 < . . . < An, and we are asked to find a lower-bound on a comparison-based algorithm for finding an arbitrary value x in A. Assume that the algorithm outputs i if x = Ai and zero otherwise. Consider the following decision tree:



Solution 3: cont…

- Thus, we see that every "=" branch leads to a single leaf. Thus, the decision tree must have one node which compares x to A_i for i = 1 to n.
- If not, we could easily come up with an adversarial-type argument where the algorithm outputs an incorrect value.
- Since every internal node produces at most two non-leaf nodes, there are at most 2k comparison nodes at level k.
- Since there are no comparisons performed on the last level of the decision tree (there are only leaf nodes), we can sum the number of comparisons performed at k levels as:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}$$

Solution 3: cont…

$$2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i \tag{1}$$

$$= \frac{1-2^k}{1-2} \tag{2}$$

$$= 2^k - 1 \tag{3}$$

Furthermore, since there must be at least n comparison nodes in the tree:

$$2^k - 1 \geq n \tag{4}$$

$$2^k \geq n + 1 \tag{5}$$

$$\log 2^k \geq \log(n+1) \tag{6}$$

$$k \geq \log(n+1) \tag{7}$$

Thus, we have shown using a decision tree that the number of comparisons must be greater than or equal to $\log(n+1)$.

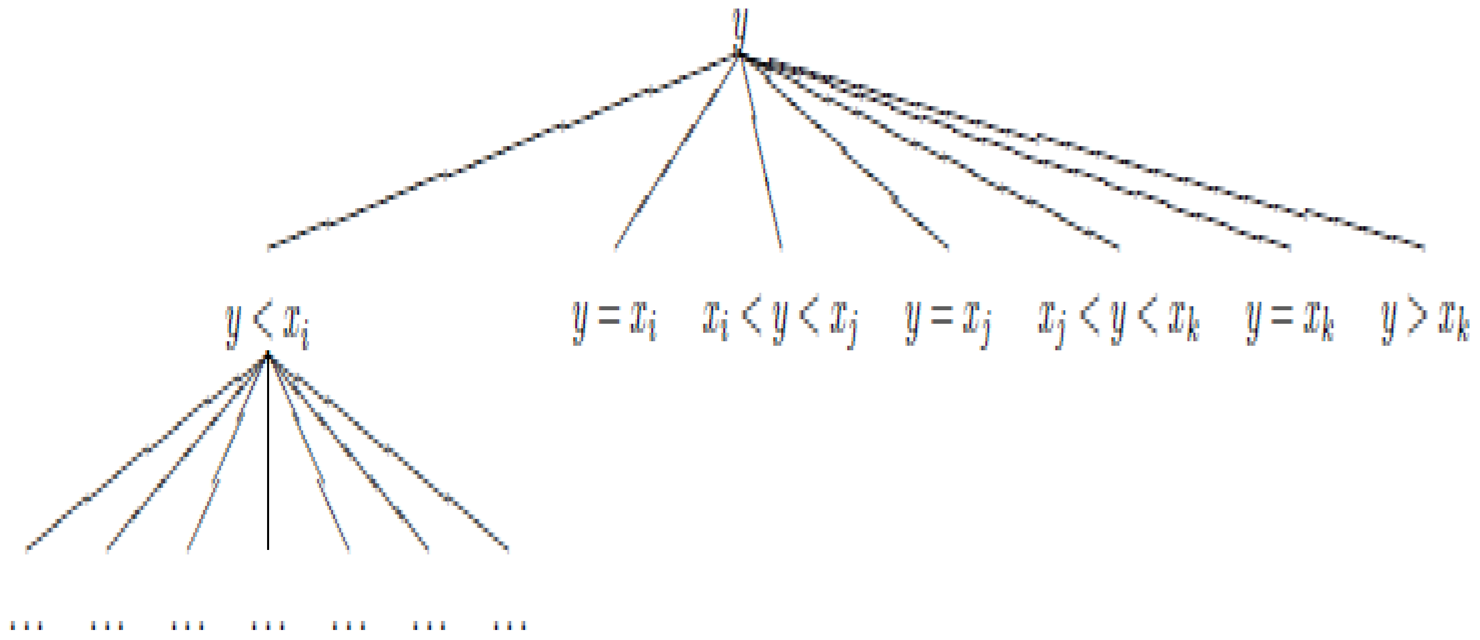
Example 4: Question

- Suppose we are given a sorted list $X = x1 < x2 < \ldots < x_n$. We want to search the list for a value y, and output 0 if y is not present and i if $y = x_i$ otherwise. However, the only way we can access X is using a procedure called **Test(i, j, i, k, y)**, which, when $0 < i < j < k \;\; n$, returns one of seven answers:

  1. $y < x_i$
  2. $y = x_i$
  3. $x_i < y < x_j$
  4. $y = x_j$
  5. $x_j < y < x_k$
  6. $y = x_k$
  7. $y > x_k$

Solution 4: Contd….

- Use a decision tree to prove a lower bound on the number of calls to Test required to determine if y is in X, and return an index if yes.
- Consider the following decision tree



Solution 4: cont…

- This tree is clearly not drawn with all of the branches, but every "=" branch leads to a single leaf as before. But in this case, each call to Test has three "=" leaf nodes and 4 real branches.
- Since the decision tree must have one node which compares x to Ai for i = 1 to n, there must be at least n/3 internal nodes (each of which represents a call to Test).
- Since every internal node produces at most 4 non-leaf nodes, there are at most 4k comparison nodes at level k.
- Since there are no comparisons performed on the last level of the decision tree (there are only leaf nodes), we can sum the number of comparisons performed at k levels as:

$$4^0 + 4^1 + 4^2 + \ldots + 4^{k-1}$$

Solution 4: cont...

$$4^0 + 4^1 + 4^2 + \ldots + 4^{k-1} = \sum_{i=0}^{k-1} 4^i \tag{8}$$

$$= \frac{1-4^k}{1-4} \tag{9}$$

$$= \frac{4^k - 1}{3} \tag{10}$$

Furthermore, since there must be at least $n/3$ comparison nodes in the tree:

$$\frac{4^k - 1}{3} \geq \frac{n}{3} \tag{11}$$

$$4^k - 1 \geq n \tag{12}$$

$$4^k \geq n + 1 \tag{13}$$

$$\log_4 4^k \geq \log_4 (n+1) \tag{14}$$

$$k \geq \log_4 (n+1) \tag{15}$$

$$k \geq \log(n+1)/2 \tag{16}$$

The last line follows from using the change of basis formula. Thus, we have shown using a decision tree, that the number of comparisons must be greater than or equal to $\log(n+1)/2$.

Thank you for your attention

Exercises

Exercises

- Prove that finding the second largest element in an n-element array requires *exactly* $n - 2 + \lceil \log n \rceil$ comparisons in the worst case. Prove the upper bound by describing and analyzing an algorithm; prove the lower bound using an adversary argument.
- Suppose we are given the adjacency matrix of a *directed* graph G with n vertices. Describe an algorithm that determines whether G has a *sink* by probing only $O(n)$ bits in the input matrix. A sink is a vertex that has an incoming edge from every other vertex, but no outgoing edges.
- We seek to sort a sequence S of n integers with many duplications, such that the number of distinct integers in S is O(log n).

(a) Give an O(n log log n) worst-case time algorithm to sort such subsequences.

(b) Why doesn't this violate the (n log n) lower bound for sorting?

Adversary Lower Bound Technique

The lower bound for the problem is the tightest (highest) lower bound that we can prove for all possible algorithms that solve the problem.

Adversary Lower Bound Technique

- You can think of the adversary lower bound technique as devising a strategy to construct a worst case input for an unknown correct algorithm to solve problem P.
- We will view this process as a game between an algorithm **A** and an adversary (or devil) **D**.
- We assume that D has unlimited computational power. In each round of this game, algorithm A asks D if element i is less than element j (for a choice of i and j made by A).
- The adversary D must answer "yes" or "no".
- This technique can be easily extended to when there are the three possible answers of <, =, and >.

Adversary Lower Bound Technique

- The goal of algorithm A is to minimize the number of rounds until its computation to solve P is completed.
- Observe, that A can do any computation it wants between rounds without any cost to it.
- The goal of the adversary D is to maximize the number of rounds until A could be done. However, D has no control over what comparison A will make.
- However, as long as there are at least two possible answers to P that are consistent with all answers given by D, A cannot be done.

Example:

- let's consider playing the game of "20 questions" against the adversary D.
- In this game, D picks an integer x between 1 and n, and the algorithm A must determine x by asking questions (to D) of the form "Is the number you have picked less than y?"
- We now argue that D can force A to ask at least $\lceil \log_2 n \rceil$ questions before A can be certain about the value for x.
- We call the way in which the adversary D plays this game the adversary strategy to be sure we do not confuse it with the algorithm A is using to determine x.
- The adversary is allowed to keep changing his mind about x but must answer in a consistent manner. That is, at the end of the game, the adversary must be able to give a value for x that is consistent with all answers given throughout the game. So it is possible that D did indeed have x in mind from the start.

An adversary strategy that leads to the stated lower bound

- The adversary maintains a list L of all possible values that are still legal for x. So initially the n integers $\{1, 2, 3, \ldots, n-1, n\}$ are placed in L.
- Each time A asks D if $x < y$, D counts how many of of the integers in L are greater than x.
- If at least half of the numbers in L are greater than x then D will respond "yes," and otherwise D will respond "no."
- If D responds "yes" (i.e., $x < y$) then all elements in the list that are $\geq y$ must be removed from L (or otherwise the adversary would be lying).
- Likewise, if D responds "no" (i.e., $x \geq y$) then all elements in the list that are $< y$ must be removed from L.
- Observe that a correct **algorithm A** cannot know the value of x (and thus not finish executing) until L contains only a single item.

- Observe that a correct algorithm A cannot know the value of x (and thus not finish executing) until L contains only a single item.
- (If there are two or more items in L then whatever A outputs could be the wrong one.)
- We must now determine how many rounds the adversary D can force before L could possibly reach size 1. Initially $|L| = n$.
- Now let's consider any single round of this game and let $|L| = s$. Since D responds in a way that least half of the items are consistent with the response, at the next round $|L|$ ds/2e.
- Let Li be the list after round i where L0 is the initial list. Then we have that $|L_0| = n$, and • $|L_{i+1}| \geq |L_i|/2e$ for $i > 0$
- It follows that $i = \lceil \log2\ n \rceil$ is the smallest possible value for i for which $|Li| = 1$. (This could be proven by induction.)

Example to illustrate this where n = 100

- $|L0| = 100$, $|L1| \geq 50$, $|L2| \geq 25$, $|L3| \geq 13$,
  $|L4| \geq 7$, $|L5| \geq 4$, $|L6| \geq 2$, $|L7| \geq 1$
- Thus after 6 rounds A could not be done since there must be at least two values for x that are consistent with all answers given so far.
- Thus for this problem when n = 100, the best any algorithm A can do is to make 7 questions.
- We have just shown that regardless of the algorithm A, the adversary strategy described above guarantees that A could not possibly be done until at least 7 rounds.
- Observe that $\lceil \log 2\ 100 \rceil = 7$. In general, any algorithm to solve this problem requires at least $\lceil \log 2\ n \rceil$ questions.
- Note that we have not shown that there exists an algorithm that could achieve this

Adversarial Lower Bounds Proofs

- A simple algorithm for finding the second largest is to first find the maximum (in n-1 comparisons), discard it, and then find the maximum of the remaining elements (in n-2 comparisons) for a total cost of 2n-3 comparisons.
- Is this optimal?

Theorem: The lower bound for finding the second largest value is 2n - 3.