Comparison-based algorithms

Finished looking at comparison-based sorts.

Crucial observation: All the sorts work for any set of elements – numbers, records, objects,……

Only require a comparator for two elements.

#include <stdlib.h>

void qsort(void *base, size_t *nmemb*, size_t *size*, int(**compar*)(const void *, const void *));

DESCRIPTION: The qsort() function sorts an array with *nmemb* elements of *size* size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

Comparison-based algorithms

Finished looking at comparison-based sorts.

Crucial observation: All the sorts work for any set of elements – numbers, records, objects,……

Only require a comparator for two elements.

#include <stdlib.h>

void qsort(void *base, size_t *nmemb*, size_t *size*, int(**compar*)(const void *, const void *));

DESCRIPTION: The qsort() function sorts an array with *nmemb* elements of *size* size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

Comparison-based algorithms

- The algorithm only uses the results of comparisons, not values of elements (*).
- Very general – does not assume much about what type of data is being sorted.
- However, other kinds of algorithms are possible!
- In this model, it is reasonable to count #comparisons.
- Note that the #comparisons is a **lower bound** on the running time of an algorithm.

(*) If values are used, lower bounds proved in this model are not lower bounds on the running time.

Lower bound for a simpler problem

Let's start with a simple problem.

Minimum of n numbers

```
Minimum (A)
1. min = A[1]
2. for i = 2 to length[A]
3.     do if min >= A[i]
4.           then min = A[i]
5. return min
```

Can we do this with fewer comparisons?

We have seen very different algorithms for this problem. How can we show that we cannot do better by being smarter?

Lower bounds for the minimum

Claim: Any comparison-based algorithm for finding the minimum of n keys must use at least n-1 comparisons.

Proof: If x,y are compared and x > y, call x the winner.

Any key that is not the minimum must have won at least one comparison. WHY?

Each comparison produces exactly one winner and at most one NEW winner.

$\Rightarrow$at least n-1 comparisons have to be made.

Points to note

Crucial observations: We proved a claim about ANY algorithm that only uses comparisons to find the minimum. Specifically, we made no assumptions about

1. Nature of algorithm.
2. Order or number of comparisons.
3. Optimality of algorithm
4. Whether the algorithm is reasonable – e.g. it could be a very wasteful algorithm, repeating the same comparisons.

On lower bound techniques

Unfortunate facts:

Lower bounds are usually hard to prove.

Virtually no known general techniques – must try ad hoc methods for each problem.

Lower bounds for comparison-based sorting

- Trivial: $\Omega(n)$ – every element must take part in a comparison.
- Best possible result – $\Omega(n \log n)$ comparisons, since we already know several O(n log n) sorting algorithms.
- Proof is non-trivial: how do we reason about all possible comparison-based sorting algorithms?

The Decision Tree Model

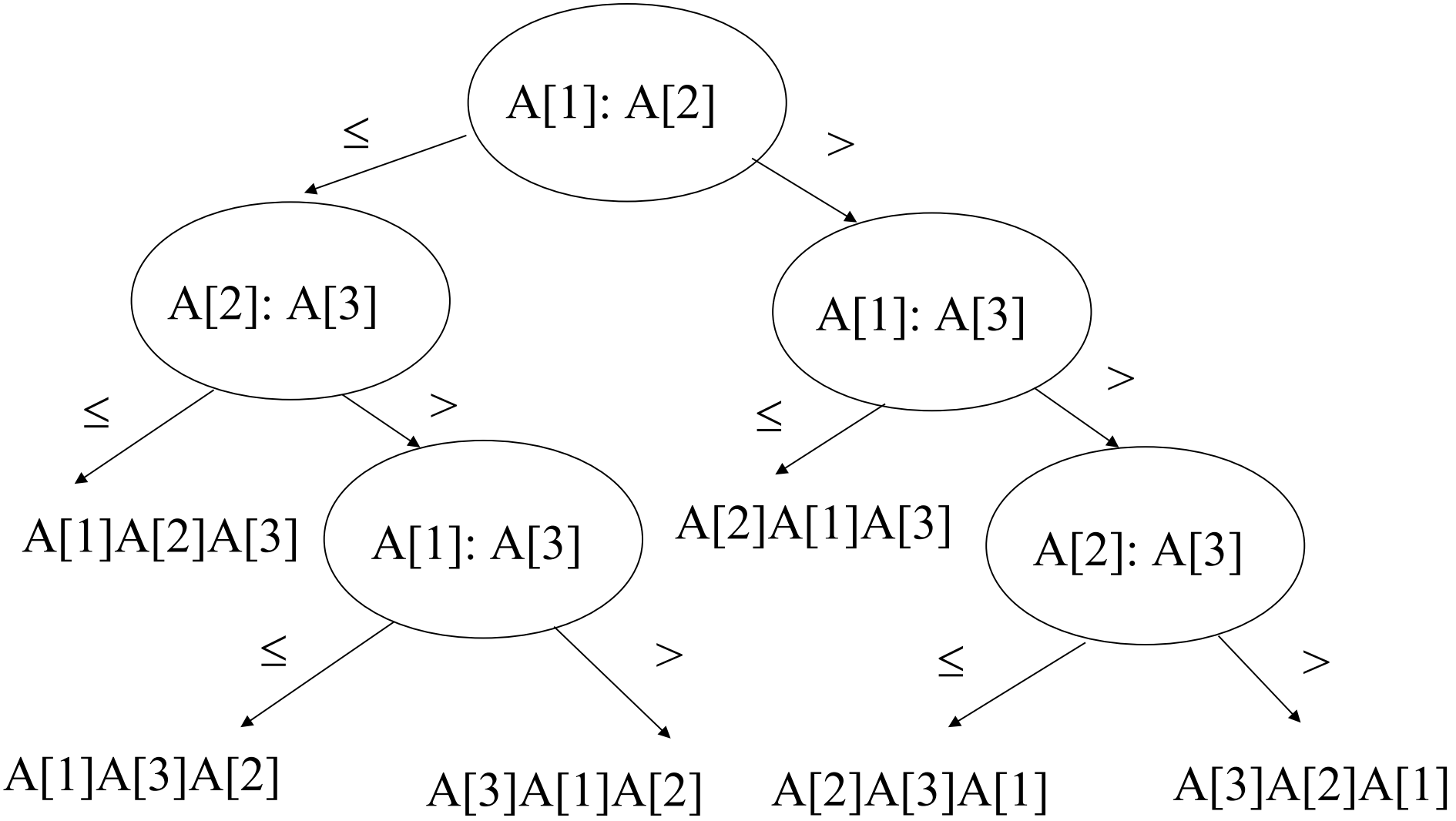
- Assumptions:
  - All numbers are distinct (so no use for $a_i = a_j$)
  - All comparisons have form $a_i \leq a_j$ (since $a_i \leq a_j$, $a_i \geq a_j$, $a_i < a_j$, $a_i > a_j$ are equivalent).
- Decision tree model
  - Full binary tree
  - Ignore control, movement, and all other operations, just use comparisons.
  - suppose three elements $< a_1, a_2, a_3>$ with instance <6,8,5>.

The Decision Tree Model - contd

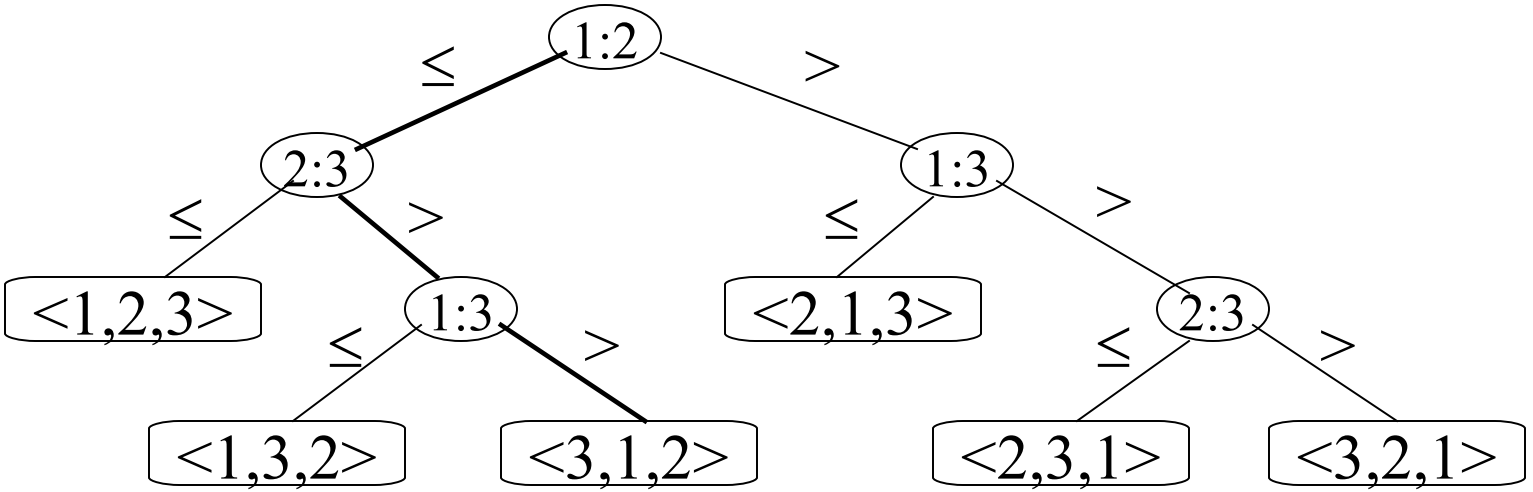
- Consider Insertion sort again

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

Example: insertion sort (n=3)



The Decision Tree Model



Internal node i:j indicates comparison between a_i and a_j.

Leaf node $<\pi(1), \pi(2), \pi(3)>$ indicates ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq a_{\pi(3)}$.

Path of bold lines indicates sorting path for <6,8,5>.

There are total 3!=6 possible permutations (paths).

Summary

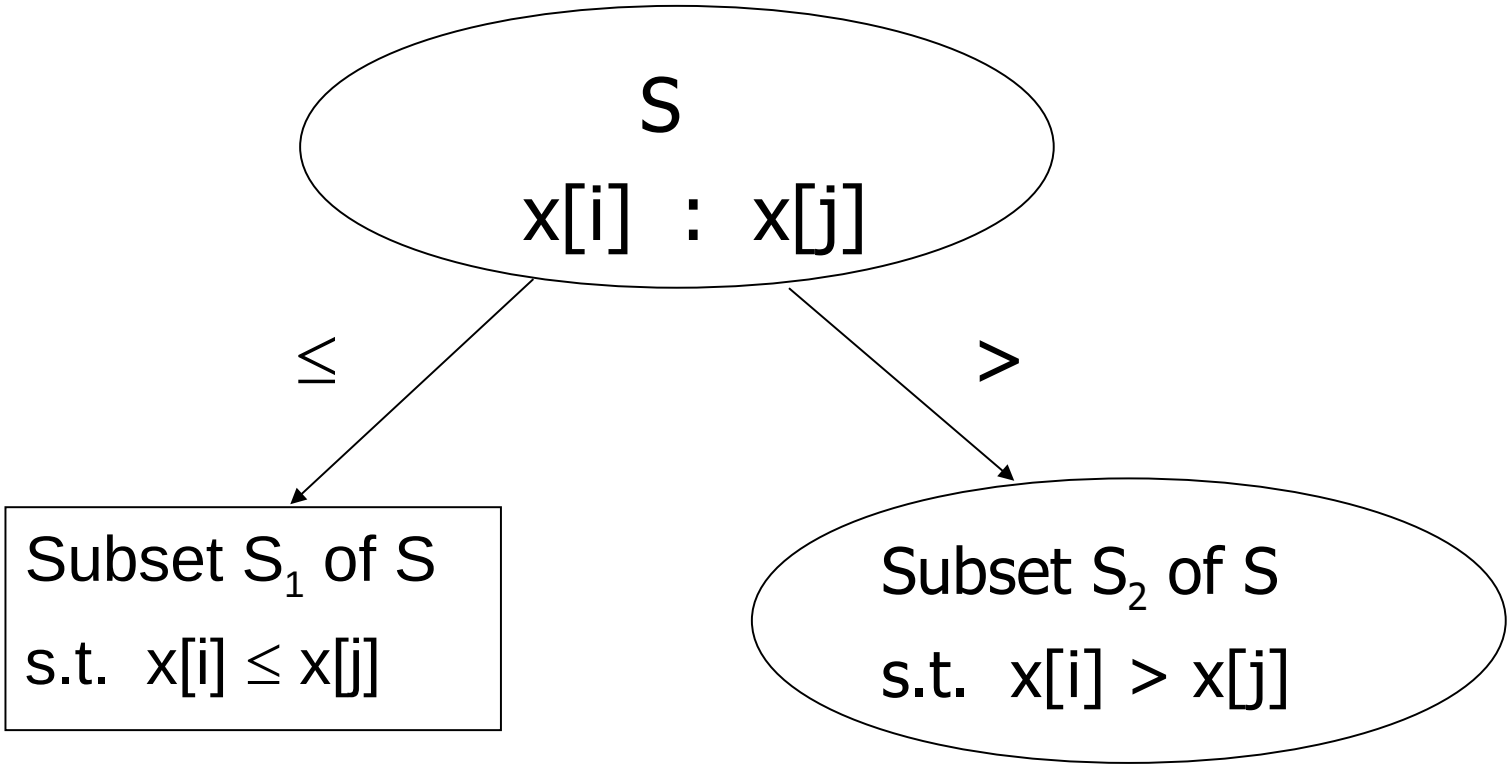
- Only consider comparisons
- Each internal node = 1 comparison
- Start at root, make the first comparison

  - if the outcome is $\leq$ take the LEFT branch

  - if the outcome is > - take the RIGHT branch
- Repeat at each internal node
- Each LEAF represents ONE correct ordering

Intuitive idea

S is a set of permutations



Lower bound for the worst case

- Claim: The decision tree must have at least n! leaves. WHY?
- worst case number of comparisons= the height of the decision tree.
- Claim: Any comparison sort in the worst case needs Ω(n log n) comparisons.
- Suppose height of a decision tree is h, number of paths (i,e,, permutations) is n!.
- Since a binary tree of height h has at most 2^h leaves,

$$n! \leq 2^h \text{ , so } h \geq \lg(n!) \geq \Omega(n \lg n)$$

Lower bounds: check your understanding

Can you prove that any algorithm that searches for an element in a sorted array of size n must have running time $\Omega(\lg n)$?