Data Structures and Algorithms for Information Processing

Lecture 12: Sorting

Outline

- Correctness proof digression
- Consider various sorts, analyze
- Insertion, Selection, Merge, Radix
- Upper & Lower Bounds
- Indexing

What Does This Method Compute?

A proof of termination is needed.

```
int doubleTheNumber(int m) {
      int n = m;
      while(n > 1) {
            if (n % 2 == 0) n = n / 2;
            else n = 3 * n + 1;
      }
      return 2 * m;
}
```

The Jar Game

A jar contains n >= 1 marbles. Each is of color red or of blue. Also we have an unlimited supply of red marbles.

Will the following algorithm terminate?

From http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html

The Jar Game

```
 while (# of marbles in the jar > 1)  {
    choose (any) two marbles from the jar;
    if (the two marbles are of the same color)
    { toss them aside;
      place a RED marble into the jar;
    }
    else {
      toss the chosen RED marble aside;
      place the chosen BLUE marble back
      into the jar;
}     }
```

http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html

Find A Loop Invarient

- Can we find a loop invariant that will help us to prove the following theorem:

  The last remaining ball will be blue if the initial number of blue balls was odd and red otherwise.

From http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html

Intuitive Introduction

[Main's slides from Chapter 12]

Insertion Sort

Consider each item once, insert into growing sorted section.

```
void insertionSort(int A[]) {
   for(int i=1; i<A.length; i++)
     for(int j=i; j>0 && A[j]<A[j-1]; j--)
          swap(A[j],A[j-1]);
}
```

Insertion Sort

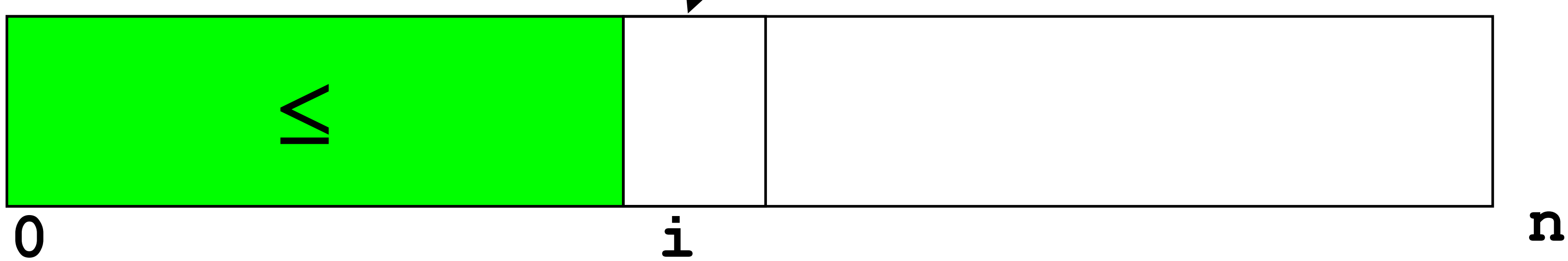
```
void insertionSort(int A[]) {
   for(int i=1; i<A.length; i++)
      for(int j=i; j>0 && A[j]<A[j-1]; j--)
            swap(A[j],A[j-1]);
}
```

- runs in $\Theta(n^2)$, where n = A.length.
- If A is sorted already, runs in Θ (n).
- Use if you're in a hurry to code it , and speed is not an issue.

Proving Insertion Sort Correct

What is the invariant?

```
void insertionSort(int A[]) {
   for(int i=1; i < A.length; i++)
      for(int j=i; j>0 && A[j]<A[j-1]; j--)
          swap(A[j],A[j-1]);
}
```



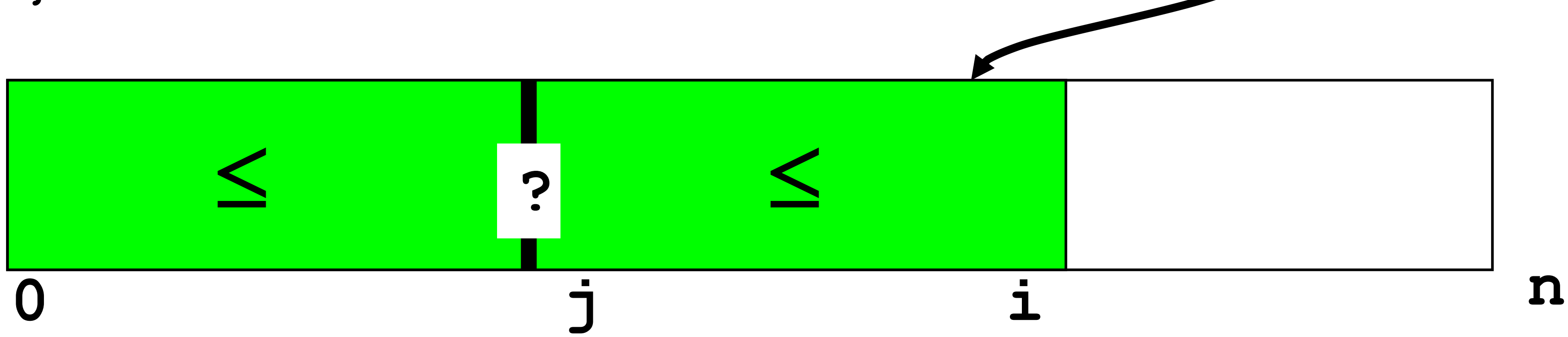
$$\left(\forall 0 \le t < u < i\right)\left[A[t] \le A[u]\right]$$

i=1, it's trivially true, When when i=n, array is sorted.

Now consider inner loop

```
void insertionSort(int A[]) {
   for(int i=1; i < A.length; i++)
       for(int j=i; j>0 && A[j]<A[j-1]; j--)
             swap(A[j],A[j-1]);

}
```



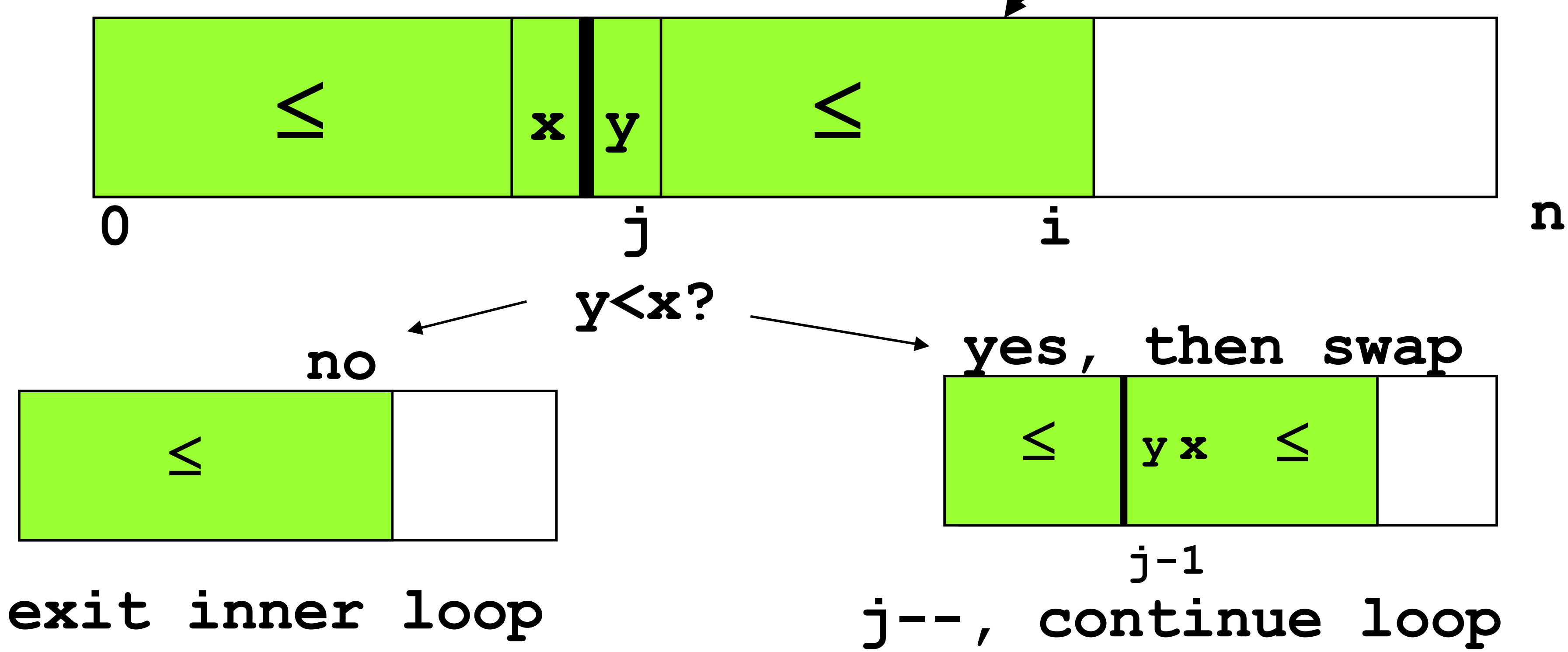
$$(\forall 0 \le t < u < j)[A[t] \le A[u]] \wedge (\forall j \le v < w \le i)[A[v] \le A[w]]$$

Trivially true when j=i, and implies outer loop invariant when it exits.

What happens inside inner loop?

```
void insertionSort(int A[]) {
   for(int i=1; i < A.length; i++)
       for(int j=i; j>0 && A[j]<A[j-1]; j--)
           swap(A[j],A[j-1]);

}
```



What is the Average Time for Insertion Sort?

(Best is Θ (n), Worst is Θ (n^2))

- Running time is proportional to number of swaps.
- Each swap of adjacent items decreases *disorder* by one unit where

  disorder = number of i<j such that A[i]>A[j]

- Therefore running time is proportional to disorder and average running time is proportional to average disorder.

Average disorder

Sequence	disorder	Reversed Sequence	disorder
1234	0	4321	6
1243	1	3421	5
1324	1	4231	5
1342	2	2431	4
1423	2	3241	4
1432	3	2341	3
2134	1	4312	5
2143	2	3412	4
2314	2	4132	4
2413	3	3142	3
3124	2	4213	4
3214	3	4123	3
	22		**50**

for n=4 Average disorder = 72/24 = 3

What is the Average Disorder?

Theorem: The average disorder for a sequence of n items is n(n-1)/4

Proof: Assume all permutations of array A equally likely. If A^R is the reverse of A, then disorder(A) + disorder(A^R) = n(n-1)/2 because A[i]<A[j] iff A^R[i]>A^R[j]. Thus the average disorder over all permutations is n(n-1)/4. □

Corollary: The average running time of *any* sorting program that swaps only adjacent elements is $\Omega(n^2)$.

Proof: It will have to do n(n-1)/4 swaps and may waste time in other ways. □

To better $\Theta(n^2)$ we must compare non-adjacent elements

Shell Sort: Swap elements n/2, n/4, … apart

Heap Sort: Swap A[i] with A[i/2]

QuickSort: Swap around "median"

Idea of Merge Sort

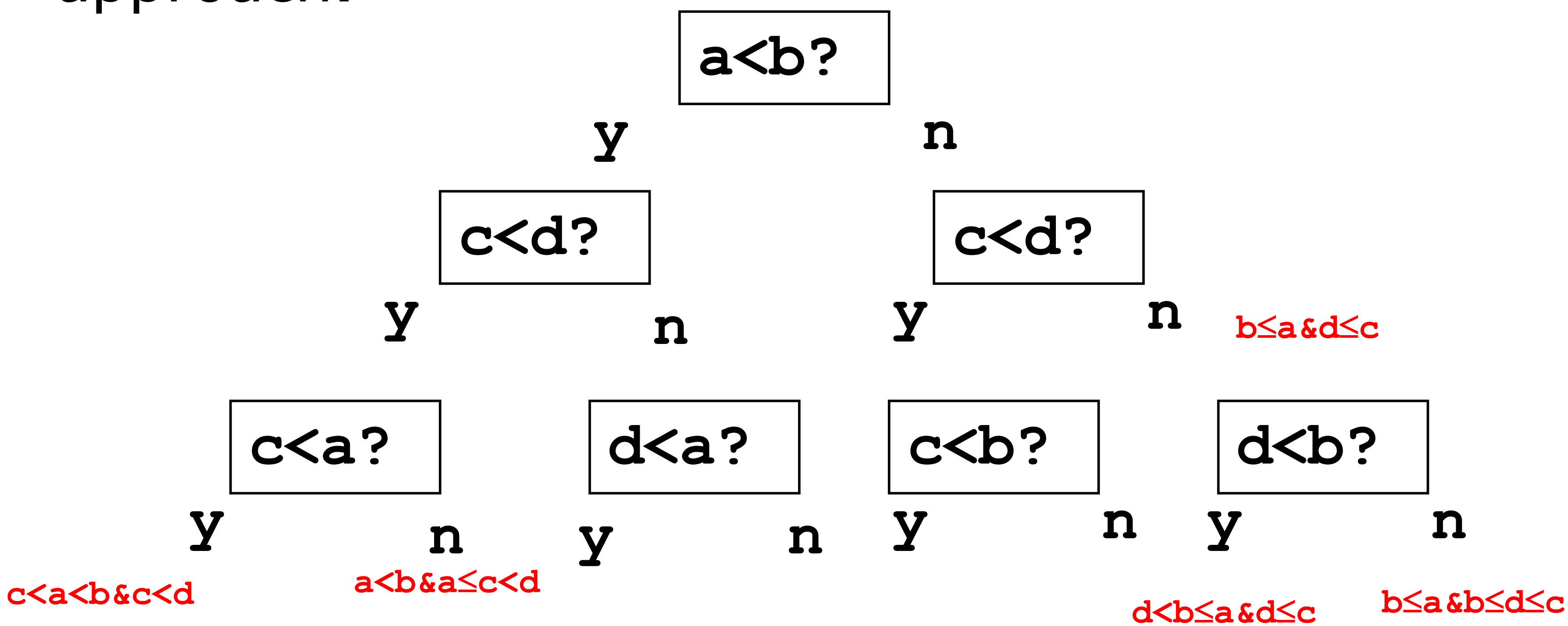
- Divide elements to be sorted into two groups of equal size
- Sort each half
- Merge the results using a simultaneous pass through each

Psuedocode for Merge Sort

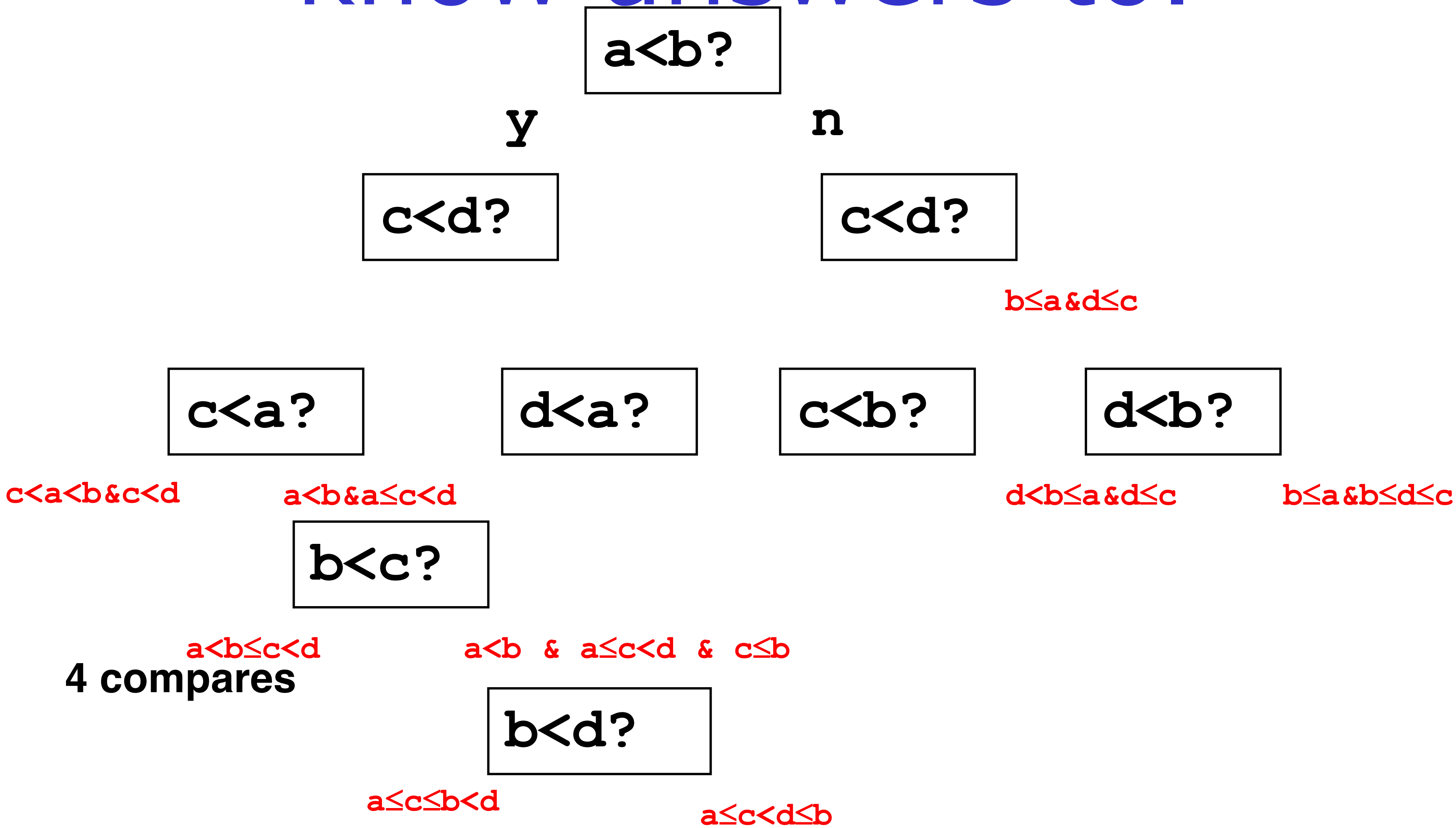
```
void mergesort(int data[], int first, int n) {
  if (n > 1) {
     int n1 = n/2;
     int n2 = n - n1;
     mergesort(data, first, n1);
     mergesort(data, first+n1, n2);
     merge(data, first, n1, n2);
  }
}
```

How fast could a sort that uses binary comparisons run?

Consider 4 numbers, a, b, c, d. Merge Sort approach:



Ask only questions you don’t know answers to.



5 compares

A different strategy, insertion sorts, may get lucky.

a<b?

y n

b<c?

c<d?

a<b<c<d

3 compares

But it may be unlucky.

a<b?

y n

b<c?

a<b & c≤b

c<d?

a<c?

a<b<c<d

c≤a<b

b<d?

d≤b & c≤a<b

a<d?

6 compares

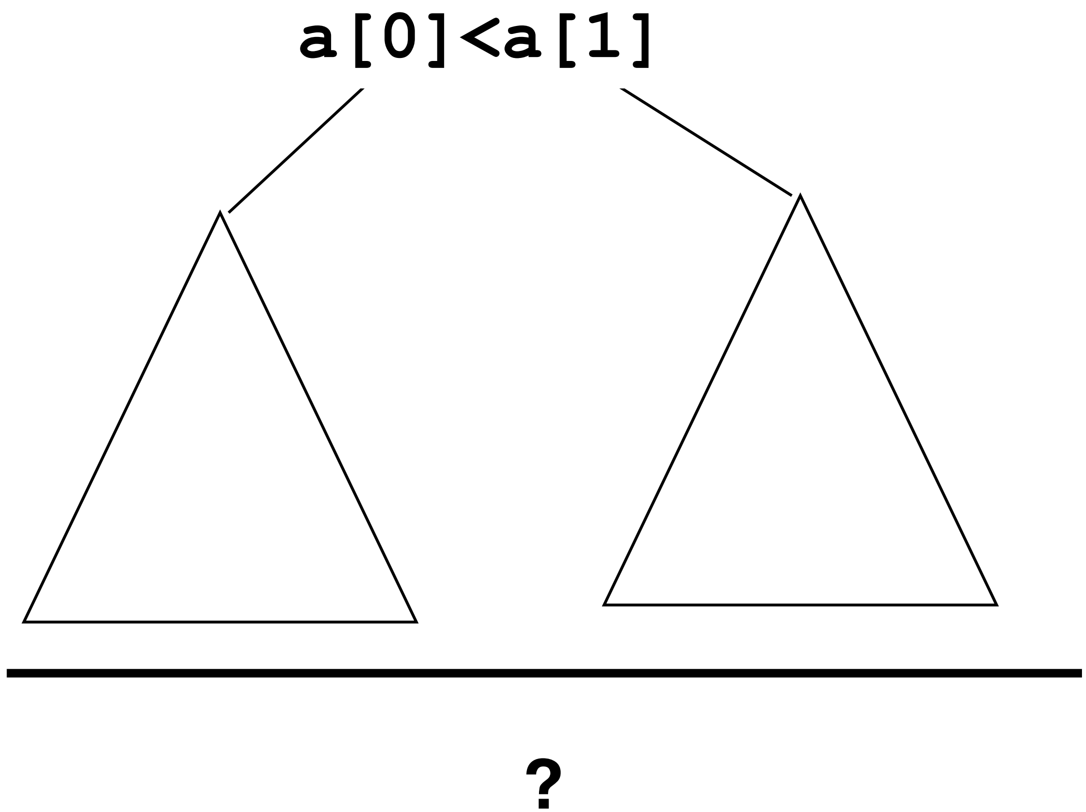
d≤a & c≤a<b

c<d?

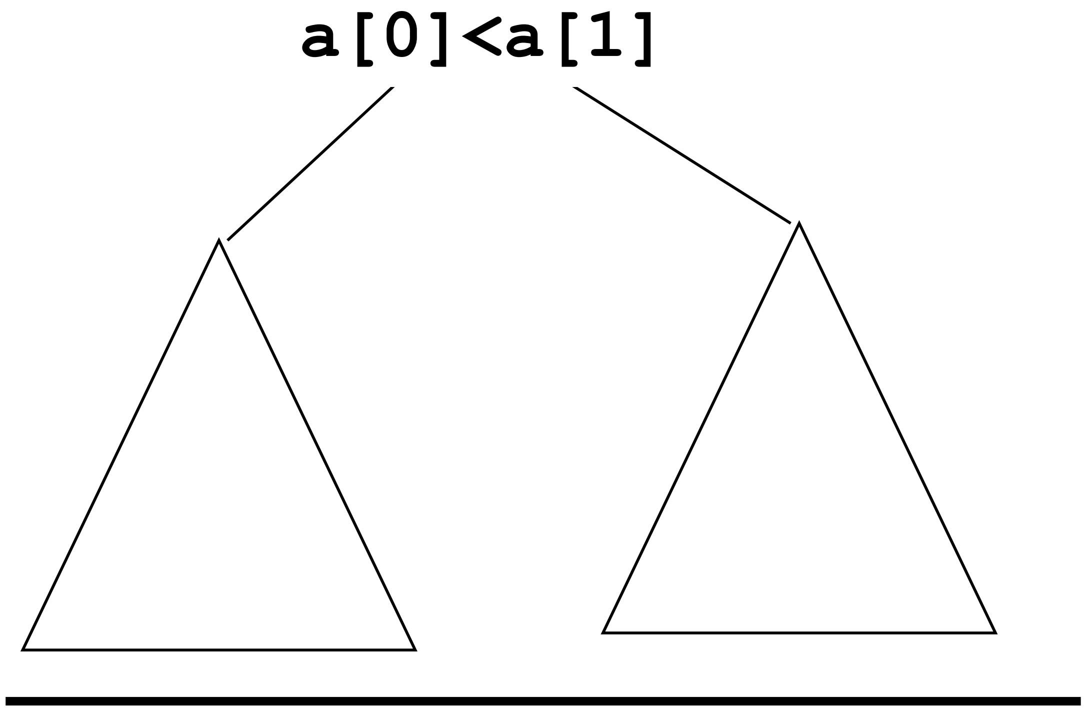
d≤c≤a<b

Consider all possible sorting trees.

How many leaves must a sorting tree have to distinguish all possible orderings of n items?



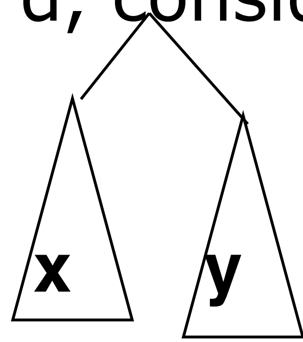
How many leaves must there for a sorting tree for n items?



n!, the number of different permutations.

Theorem: A binary tree with K leaves must have depth at least $\lceil \log_2 K \rceil$. In other words, a BT with k leaves and depth d has d >= $\lceil \log_2 K \rceil$ or $K <= 2^d$

Proof: Prove by induction that a tree of depth d can have at most, 2^d leaves.
Base: for d=0, there is 1 leaf.
Suppose true for d, consider tree of depth d+1.



BIH: x and y have at most 2^d leaves so whole tree has at most $2*2^d = 2^{d+1}$ leaves.

Now the shortest trees with K leaves must be “perfect” and their depth will be $\lceil \log_2 K \rceil$

So a tree with n! leaves has depth at least lg n!.

Notice that depth = the maximum number of tests one might have to perform.

$$
\begin{aligned}
\lg n! &= \lg n(n-1)(n-2)\ldots 1 \\
&= \lg n + \lg n-1 + \lg n-2 + \ldots + \lg 1 \\
&\geq \lg n + \ldots + \lg(n/2) \\
&\geq (n/2) \lg(n/2) \\
&\geq (n/2) \lg n - n/2 \\
&= \Omega(n \lg n)
\end{aligned}
$$

So *any* sort algorithm takes $\Omega(n \lg n)$ comparisons.

Is there a way to sort without using binary comparisons?

Ternary comparisons, K-way comparisons.

The basic $\Omega(n \log n)$ result will still be true, because $\Omega(\log_2 x) = \Omega(\log_k x)$.

Useful speed-up heuristic: use your data as an index of an array.

Consider sorting tray of letters

```
int counts[26];
int j = 0;
for(int i=0; i<26; i++) counts[i]=0;
for(j=0; j<tray.length; j++)
     count[tray[j]-'a']++;
j=0;
for(int i=0; i<26; i++)
  while(count[i]-- > 0) tray[j++]=i+'a';
```

Sorting tray of letters

```
int counts[26];
int j = 0;
for(int i=0; i<26; i++) counts[i]=0;
for(j=0; j<tray.length; j++)
        count[tray[j]-'a']++;
j=0;
for(int i=0; i<26; i++)
  while(count[i]-- > 0) tray[j++]=i+'a';
```

if tray = "abbcabbdaf"

count = {3,4,1,1,0,1,0, …, 0}

and new tray = "aaabbbbcdf"

Running time is Θ(26+tray.size()), i.e. ***linear***!

Why does this beat n log n?

- The operation `count[tray[j]]++` is like a 26-way test; the outcome depends directly on the data.
- This is "cheating" because it won't work if the data range grows from 26 to 2^{32}.
- Technique can still be useful — can break up range into "buckets" and use mergesort on each bucket

Radix Sort

A way to exploit the data-driven idea for large data spaces.

Idea: Sort the numbers by their *lowest* digit. Then sort them by the next lowest digit, being careful to break ties properly. Continue to highest digit.

4567	3480	1908	2009	109	
2132	9241	109	109	456	
456	8721	2009	2132	1908	
1908	3521	8721	9241	2009	
3456	2132	3521	3297	2132	
9241	456	2132	456	3297	
109	3456	9241	3456	3456	
5789	4567	456	3480	3480	
3297	3297	3456	3521	3521	
2009	1908	4567	4567	4567	
8721	109	3480	8721	5789	
3521	5789	5789	5789	8721	
3480	2009	3297	1908	9241	

Radix Sort

- Each sort must be ***stable***
  The relative order of equal keys is preserved
- In this way, the work done for earlier bits is not "undone"

Radix Sort

Informal Algorithm:

To sort items A[i] with value $0 \ldots 2^{32}-1$ (= INT_MAX)

- Create a table of 256 buckets.
- {For every A[i] put it in bucket A[i] mod 256.
- Take all the items from the buckets 0,…, 255 in a FIFO manner, re-packing them into A.}
- Repeat using A[i]/256 mod 256
- Repeat using A[i]/256^2 mod 256
- Repeat using A[i]/256^3 mod 256
- This takes O(4*(256+A.length))

Radix Sort using Counts

The Queues can be avoided by using counts:

Let N= number of elements in array a

Array a is indexed from 1 to N

Let w = the number of bits in a[i]

Let m = number of bits examined per pass

Let M = 2^m patterns to count

Radix Sort using Counts

The Queues can be avoided by using counts:

```
void RadixSort(int a[], int b[], int N) {
   int i, j, pass, count[M];
   for (pass=0; pass < (w/m); pass++) {
      for (j=0; j < M; j++) count[j] = 0;
      for (i=1; i <= N; i++)
         count[a[i].bits(pass*m, m)]++;
      for (j=1; j < M; j++)
         count[j] = count[j-1] + count[j];
      for (i=N; i >= 1; i--)
         b[count[a[i].bits(pass*m,m)]--] = a[i];
      for (i=1; i <= N; i++) a[i] = b[i];
   }
}
```

Radix Sort using Queues

```
const int BucketCount = 256;
void RadixSort(vector<int> &A) {
 vector<queue<int> > Table(BucketCount);
 int passes = ceil(log(INT_MAX)/log(BucketCount));
 int power = 1;
 for(int p=0; p<passes;p++) {
    int i;
    for(i=0; i<A.size(); i++) {
         int item = A[i];
         int bucket = (item/power) % BucketCount;
         Table[bucket].push(item);
    }
    i =0;
    for(int b=0; b<BucketCount; b++)
      while(!Table[b].empty()) {
         A[i++] = Table[b].front(); Table[b].pop();
      }
    power *= BucketCount;
 }}
```

Radix Sort

In general it takes time

Θ(Passes*(NBuckets+A.length))

where Passes=
$\lceil$log(INT_MAX)/log(NBuckets)$\rceil$

Suppose we have n 4 digit numbers to sort and 1 bucket for each digit.

Passes = ceil($\log_{10}$(9999)/$\log_{10}$(10)) = 4

Θ(4 * (10 + n))

It needs Θ(A.length) in extra space.

Next Time

- The next topic will be ***Quicksort***, a very fast, practical, and widely used algorithm