Sorting (Part II)

CSE 373

Data Structures

Unit 17

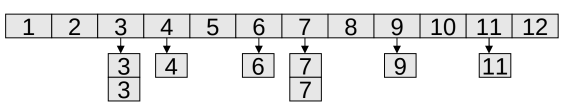
<u>Reading:</u> Section 3.2.6 Radix sort
Section 7.6 Mergesort, Section 7.7, Quicksort,
Sections 7.8 Lower bound

Bucket Sort: Sorting Integers

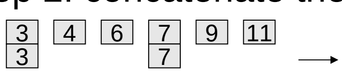
- The goal: sort N numbers, all between 1 to k.
- Example: sort 8 numbers 3,6,7,4,11,3,5,7. All between 1 to 12.
- The method: Use an array of k queues. Queue j (for $1 \leq j \leq k$) keeps the input numbers whose value is j.
- Each queue is denoted 'a bucket'.
- Scan the list and put the elements in the buckets.
- Output the content of the buckets from 1 to k.

Bucket Sort: Sorting Integers

- Example: sort 8 numbers 3,6,7,4,11,3,9,7 all between 1 to 12.
- Step 1: scan the list and put the elements in the queues



- Step 2: concatenate the queues



3,3,4,6,7,7,9,11

- Time complexity: O(n+k).

Radix Sort: Sorting integers

- Historically goes back to the 1890 census.
- Radix sort = multi-pass bucket sort of integers in the range 0 to B^P-1
- Bucket-sort from least significant to most significant "digit" (base B)
- Requires P(B+N) operations where P is the number of passes (the number of base B digits in the largest possible input number).
- If P and B are constants then O(N) time to sort!

Radix Sort Example

Input data

478
537
9
721
3
38
123
67

Bucket sort
by 1's digit

0	1	2	3	4	5	6	7	8	9
	721		3				537	478	9
			123				67	38	

After 1st pass

721
3
123
537
67
478
38
9

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

Radix Sort Example

After 1st pass

721
3
123
537
67
478
38
9

Bucket sort
by 10's
digit

0	1	2	3	4	5	6	7	8	9
03		721	537			67	478		
09		123	38						

After 2nd pass

3
9
721
123
537
38
67
478

Radix Sort Example

After 2nd pass

3
9
721
123
537
38
67
478

Bucket sort
by 100's
digit

0	1	2	3	4	5	6	7	8	9
003	123			478	537		721		
009									
038									
067									

After 3rd pass

3
9
38
67
123
478
537
721

Invariant: after k passes the low order k digits are sorted.

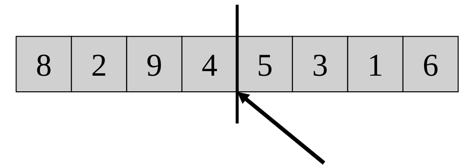
Properties of Radix Sort

- Not in-place
  - › needs lots of auxiliary storage.
- Stable
  - › equal keys always end up in same bucket in the same order.
- Fast
  - › Time to sort N numbers in the range 0 to B^P-1 is O(P(B+N)) (P iterations, B buckets in each)

"Divide and Conquer"

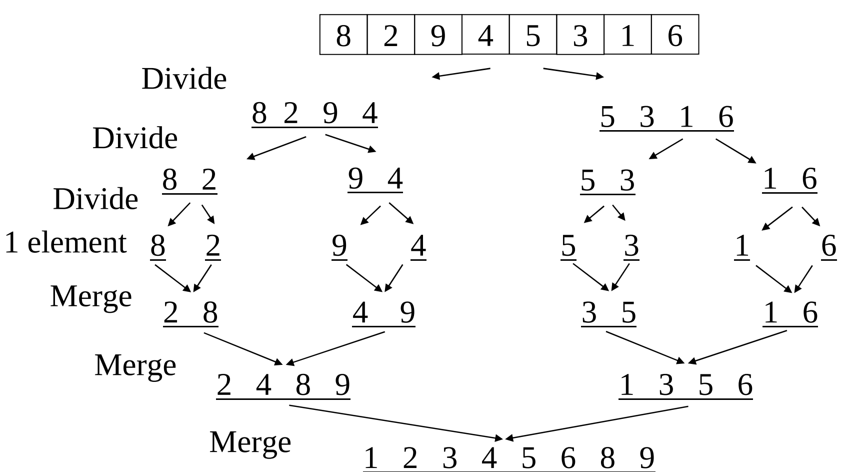
- Very important strategy in computer science:
  - › Divide problem into smaller parts
  - › Independently solve the parts
  - › Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves à Mergesort
- **Idea 2 :** Partition array into items that are "small" and items that are "large", then recursively sort the two sets à Quicksort

Mergesort



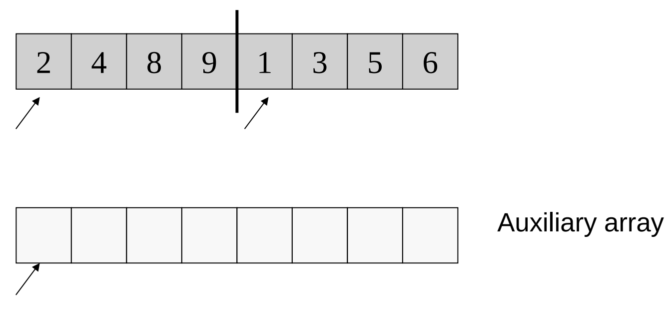
- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Mergesort Example



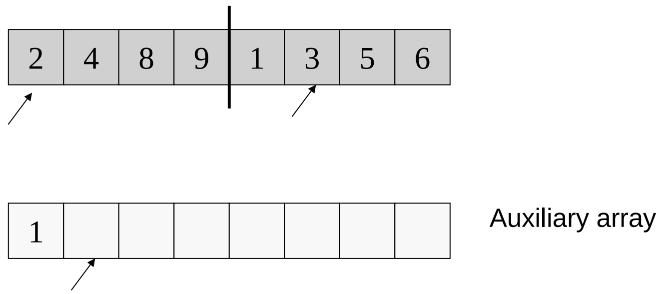
Auxiliary Array

- The merging requires an auxiliary array.



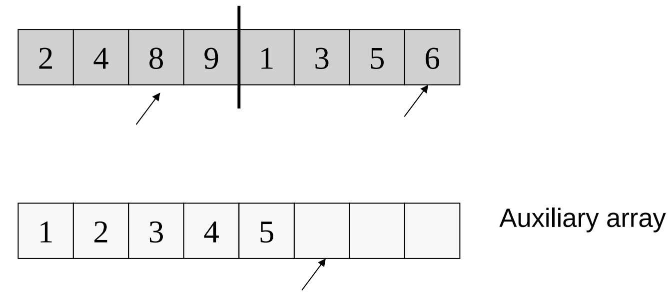
Auxiliary Array

- The merging requires an auxiliary array.

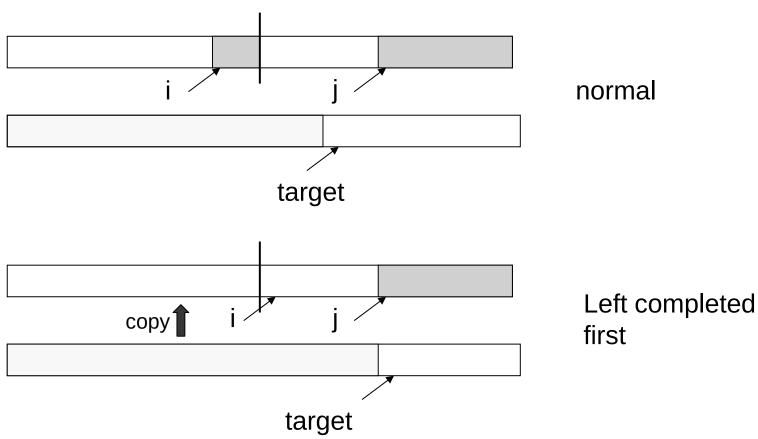


Auxiliary Array

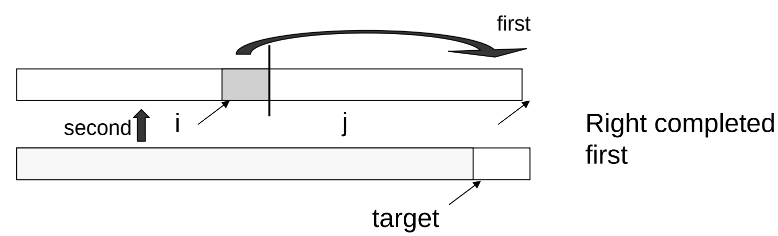
- The merging requires an auxiliary array.



Merging



Merging



Merging

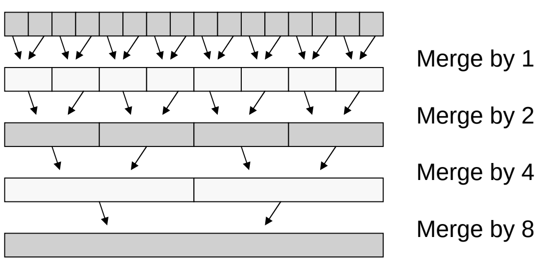
```
Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i ≤ mid and j ≤ right do
    if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;
      else T[target] := A[j]; j := j + 1;
    target := target + 1;
  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k : = mid; l := right;
    while k ≥ i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}
```


Recursive Mergesort

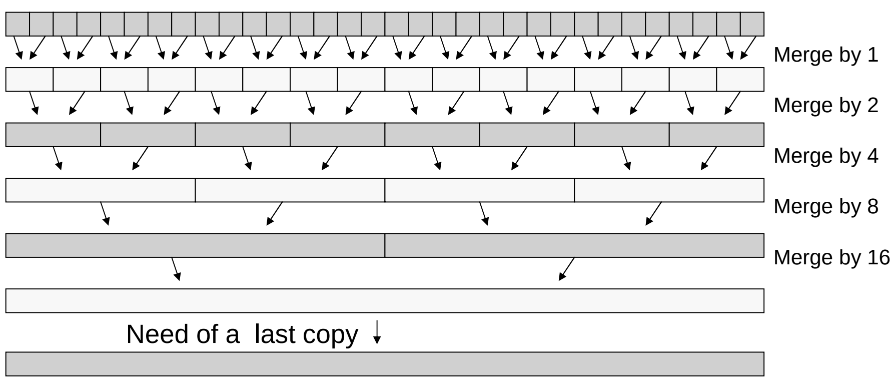
```
Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort[A,T,1,n];
}
```


Iterative Mergesort



Iterative Mergesort



Iterative Mergesort

```
IterativeMergesort(A[1..n]: integer array, n : integer) : {
//precondition: n is a power of 2//
  i, m, parity : integer;
  T[1..n]: integer array;
  m := 2; parity := 0;
  while m ≤ n do
    for i = 1 to n – m + 1 by m do
       if parity = 0 then Merge(A,T,i,i+m-1);
         else Merge(T,A,i,i+m-1);
    parity := 1 – parity;
    m := 2*m;
  if parity = 1 then
    for i = 1 to n do A[i] := T[i];
}
```

How do you handle non-powers of 2?
How can the final copy be avoided?

Mergesort Analysis

- Let T(N) be the running time for an array of N elements
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
- Each recursive call takes T(N/2) and merging takes O(N)

Mergesort Recurrence Relation

- The recurrence relation for T(N) is:
  - › $T(1) \le a$
    - base case: 1 element array à constant time
  - › $T(N) \le 2T(N/2) + bN$
    - Sorting N elements takes
      - – the time to sort the left half
      - – plus the time to sort the right half
      - – plus an O(N) time to merge the two halves
- T(N) = O(n log n) (see Lecture 5 Slide17)

Properties of Mergesort

- Not in-place
  - › Requires an auxiliary array (O(n) extra space)
- Stable
  - › Make sure that left is sent to target on equal values.
- Iterative Mergesort reduces copying.

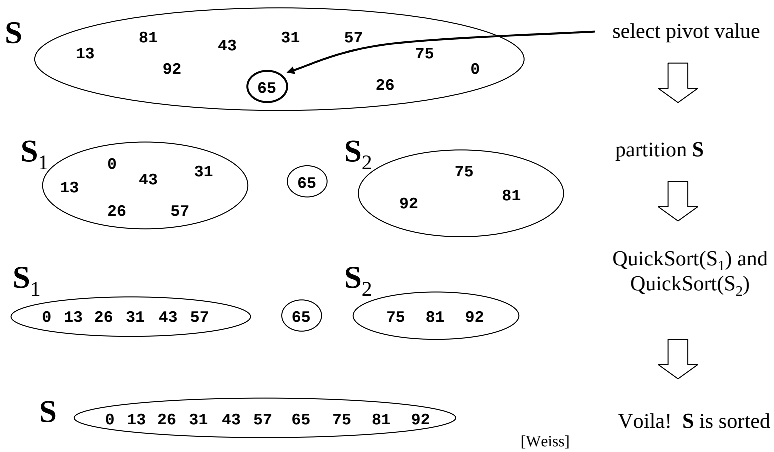
Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - › Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - › Recursively sort left and right sub-arrays
  - › Concatenate left and right sub-arrays in O(1) time

"Four easy steps"

- To sort an array **S**
  1. If the number of elements in **S** is 0 or 1, then return. The array is sorted.
  2. Pick an element *v* in **S**. This is the *pivot* value.
  3. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ = {all values $x \leq v$}, and $\mathbf{S}_2$ = {all values $x \geq v$}.
  4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

The steps of QuickSort



Details, details

- Implementing the actual partitioning
- Picking the pivot
  - › want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Dealing with cases where an element equals the pivot

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - › the elements in left sub-array are ≤ pivot
  - › elements in right sub-array are ≥ pivot
- How do the elements get to the correct partition?
  - › Choose an element from the array as the pivot
  - › Make one pass through the rest of the array and swap as needed to put elements in partitions

Partitioning:Choosing the pivot

- One implementation (there are others)
  - › median3 finds pivot and sorts left, center, right
    - Median3 takes the median of leftmost, middle, and rightmost elements
    - An alternative is to choose the pivot randomly (need a random number generator; “expensive”)
    - Another alternative is to choose the first element (but can be very bad. Why?)
  - › Swap pivot with next to last element

Partitioning in-place

- › Set pointers i and j to start and end of array
- › Increment i until you hit element A[i] > pivot
- › Decrement j until you hit element A[j] < pivot
- › Swap A[i] and A[j]
- › Repeat until i and j cross
- › Swap pivot (at A[N-2]) with A[i]

Example

Choose the pivot as the median of three

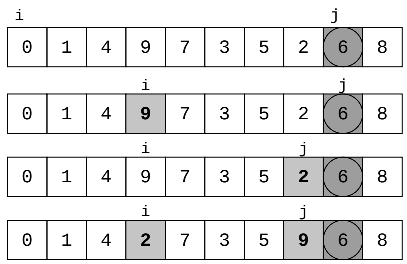
0	1	2	3	4	5	6	7	8	9
8	1	4	9	**0**	3	5	2	7	**6**

Median of 0, 6, 8 is 6. Pivot is 6

0	1	4	9	7	3	5	2	6	8
i								j	

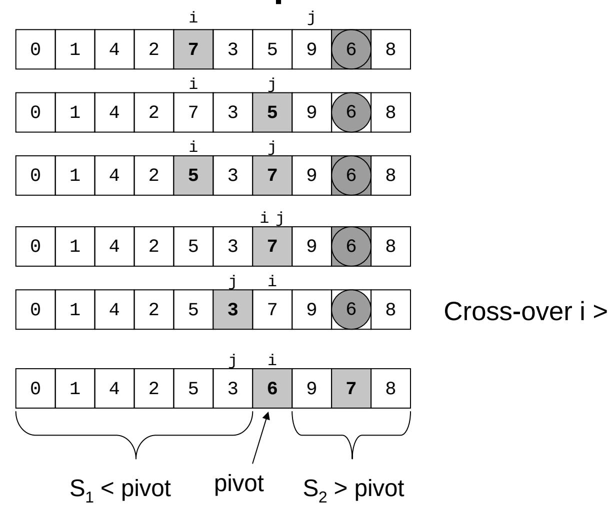
Place the largest at the right
and the smallest at the left.
Swap pivot with next to last element.

Example



Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap

Example



Recursive Quicksort

```
Quicksort(A[]: integer array, left,right : integer): {
pivotindex : integer;
if left + CUTOFF ≤ right then
  pivot := median3(A,left,right);
  pivotindex := Partition(A,left,right-1,pivot);
  Quicksort(A, left, pivotindex – 1);
  Quicksort(A, pivotindex + 1, right);
else
  Insertionsort(A,left,right);
}
```

Don't use quicksort for small arrays.
CUTOFF = 10 is reasonable.

Quicksort Best Case Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - › T(0) = T(1) = O(1)
    - constant time if 0 or 1 element
  - › For N > 1, 2 recursive calls plus linear time for partitioning
  - › T(N) = 2T(N/2) + O(N)
    - Same recurrence relation as Mergesort
  - › T(N) = <u>O(N log N)</u>

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - › $T(N) \le a$ for $N \le C$
  - › $T(N) \le T(N-1) + bN$
  - › $\le T(N-2) + b(N-1) + bN$
  - › $\le T(C) + b(C+1) + \ldots + bN$
  - › $\le a + b(C + (C+1) + (C+2) + \ldots + N)$
  - › $T(N) = O(N^2)$
- Fortunately, *average case performance* is O(N log N) (see text for proof)

Properties of Quicksort

- Not stable because of long distance swapping.
- No iterative version (without using a stack).
- Pure quicksort not good for small arrays.
- "In-place", but uses auxiliary storage because of recursive call (O(logn) space).
- O(n log n) average case performance, but $O(n^2)$ worst case performance.

Folklore

- "Quicksort is the best in-memory sorting algorithm."
- Truth
  - › Quicksort uses very few comparisons on average.
  - › Quicksort does have good performance in the memory hierarchy.
    - Small footprint
    - Good locality

How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in O(N log N) best case running time
- Can we do any better?
- No, if sorting is comparison-based.
- We saw that radix sort is O(N) but it is only for integers from bounded-range.

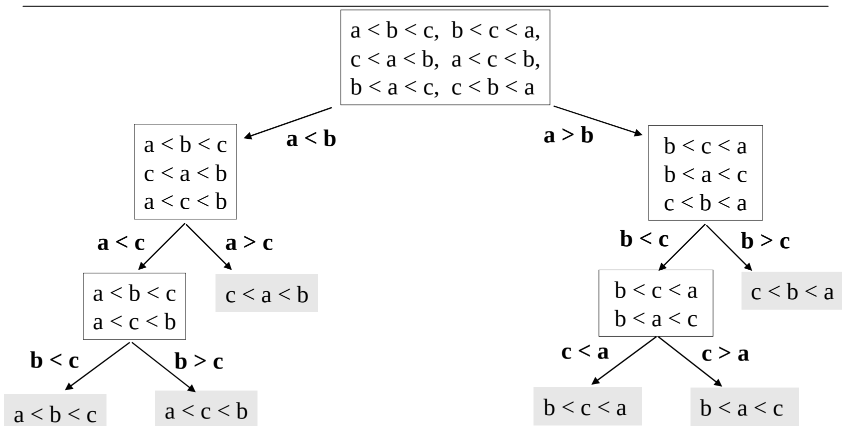
Sorting Model

- Recall the basic assumption: we can only compare two elements at a time
  - › we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - › Assume no duplicates
- How many possible orderings can you get?
  - › Example: a, b, c (N = 3)

Permutations

- How many possible orderings can you get?
  - › Example: a, b, c (N = 3)
  - › (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - › 6 orderings = 3•2•1 = 3! (i.e., “3 factorial”)
  - › All the possible permutations of a set of 3 elements
- For N elements
  - › N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  - › N(N-1)(N-2)⋯(2)(1)= N! possible orderings

Decision Tree



The leaves contain all the possible orderings of a, b, c

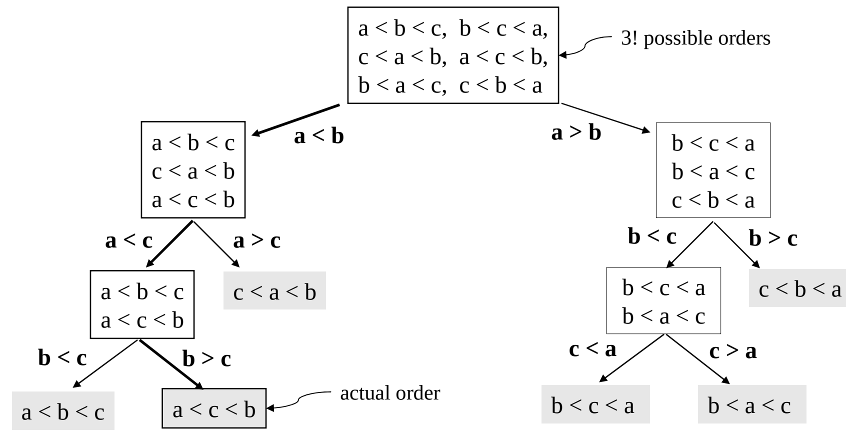
Decision Trees

- A Decision Tree is a Binary Tree such that:
  - › Each node = a set of orderings
    - i.e., the remaining solution space
  - › Each edge = 1 comparison
  - › Each leaf = 1 unique ordering
  - › How many leaves for N distinct elements?
    - N!, i.e., a leaf for each possible ordering
- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

Decision Trees and Sorting

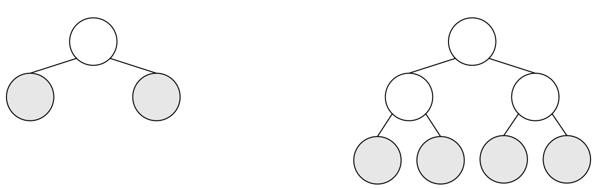
- Every comparison-based sorting algorithm corresponds to a decision tree
  - › Finds correct leaf by choosing edges to follow
    - i.e., by making comparisons
  - › Each decision reduces the possible solution space by one half
- Run time is ≥ maximum no. of comparisons
  - › maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

Decision Tree Example



How many leaves on a tree?

- Suppose you have a binary tree of height d . How many leaves can the tree have?
  - › d = 1 à at most 2 leaves,
  - › d = 2 à at most 4 leaves, etc.



Lower bound on Height

- A binary tree of height d has at most **2^d** leaves
  - › depth d = 1 à 2 leaves, d = 2 à 4 leaves, etc.
  - › Can prove by induction
- Number of leaves, $L \leq 2^d$
- Height $d \geq \log_2 L$
- The decision tree has N! leaves
- So the decision tree has height $d \geq \log_2(N!)$

log(*N*!) is Ω(*N*log*N*)

$$\log(N!) = \log\left(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)\right)$$

$$= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1$$

select just the first N/2 terms

$$\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2}$$

each of the selected terms is ≥ logN/2

$$\geq \frac{N}{2} \log \frac{N}{2}$$

$$\geq \frac{N}{2}(\log N - \log 2) = \frac{N}{2}\log N - \frac{N}{2}$$

$$= \Omega(N \log N)$$

$n! \approx \sqrt{2\pi n}(n/e)^n$

Sterling's formula

Summary of Sorting

- Sorting choices:
  - $O(N^2)$ – Bubblesort, Insertion Sort
  - O(N log N) average case running time:
    - Heapsort: In-place, not stable.
    - Mergesort: O(N) extra space, stable.
    - Quicksort: claimed fastest in practice but, $O(N^2)$ worst case. Needs extra storage for recursion. Not stable.
  - Run time of any comparison-based sorting algorithm is Ω(N log N)
  - O(N) – Radix Sort: fast and stable. Not comparison based. Not in-place.