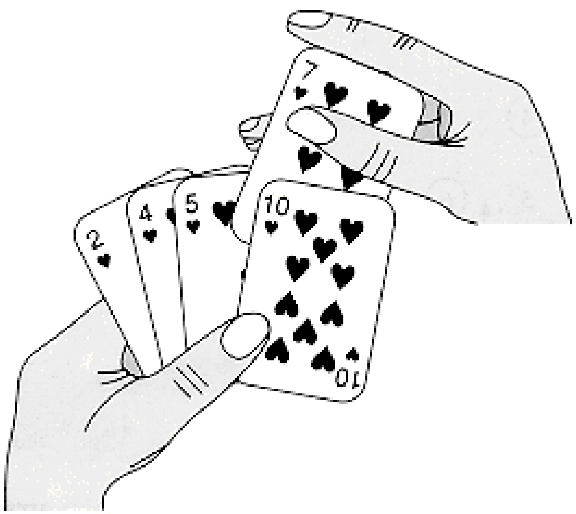
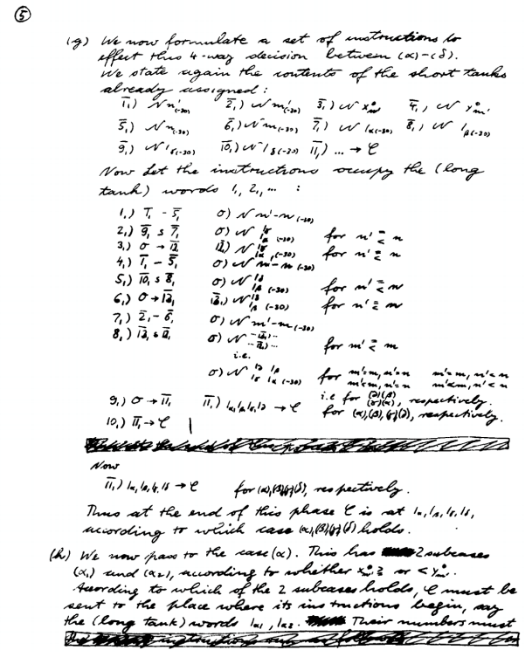
CS3334 Data Structures

Lecture 5: Divide-Conquer & Merge Sort




(g) We now formulate a set of instructions to effect this 4-way decision between (α)-(δ). We state again the contents of the short tanks already assigned:

Now let the instructions occupy the (long tank) words 1, 2, ... :

Thus at the end of this phase C is at ... according to which case (α),(β),(γ),(δ) holds.

(h) We now pass to the case (α). This has 2 subcases (α₁) and (α₂), according to whether ... According to which of the 2 subcases holds, C must be sent to the place where its instructions begin, say the (long tank) words ... Their numbers must

Chee Wei Tan

How to Solve It?

George Pólya

1) First, you have to understand the problem.
2) After understanding, make a plan.
3) Carry out the plan.
4) Look back on your work. How could it be better?

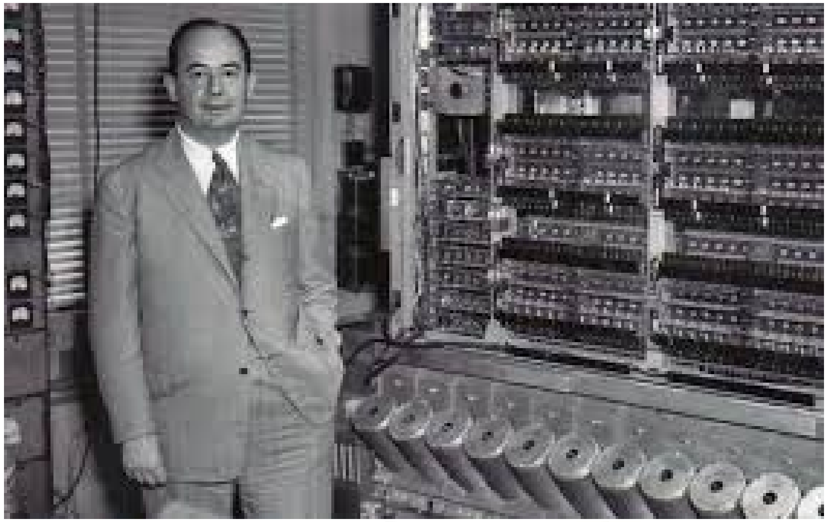
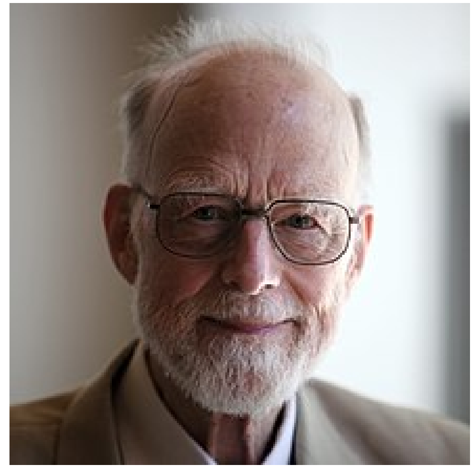
If this fails, Pólya advises: "If you can't solve a problem, then there is an easier problem you can solve: find it."
https://en.wikipedia.org/wiki/How_to_Solve_It

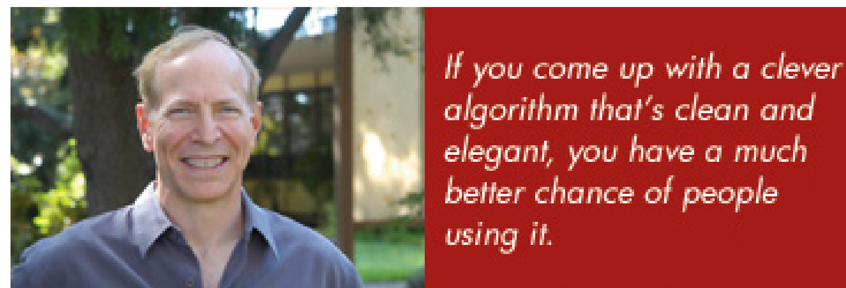
Clearly expound the idea of Divide-and-Conquer

Break down a problem into smaller ones you can handle/play with

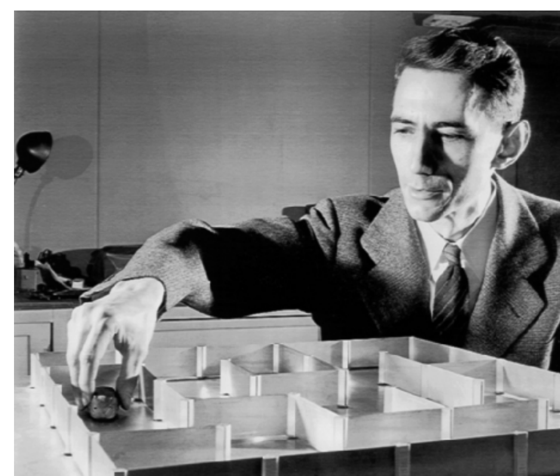
"Johnny (von Neumann) was the only student I was ever afraid of," George Pólya

Giants of Computer Science





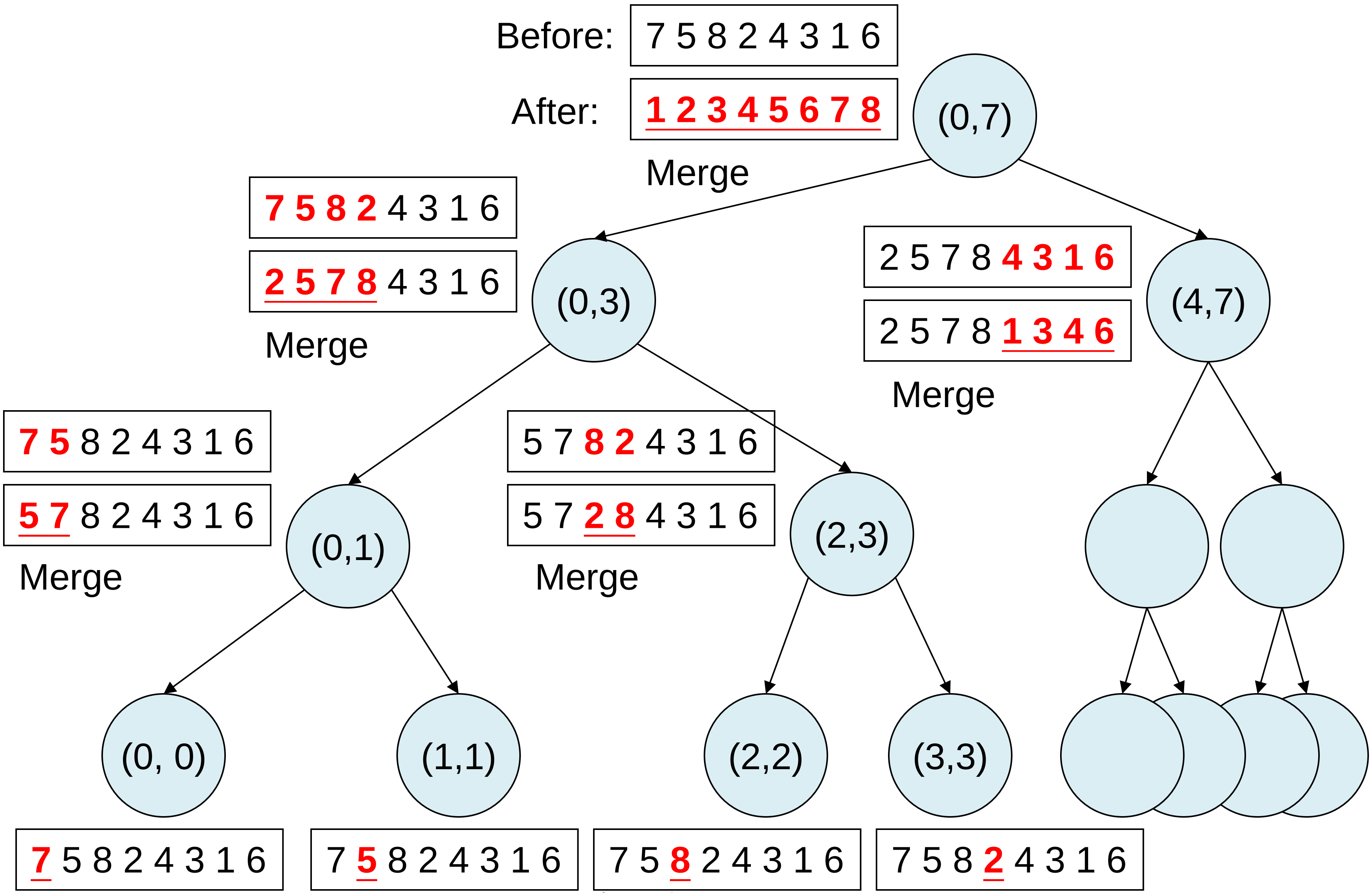
Idea: Divide and Conquer

- **Divide**: Divide a problem into smaller and simpler subproblems
- **Conquer**: Solve recursively each subproblem
- **Merge**: Combine solutions to subproblems to get the solution to original problem
- Examples of historical algorithms:
  - Gauss Fast Fourier Transform (1805)
    - Gauss discovered the algorithm at age 28
  - Merge Sort algorithm (1945)
    - John von Neumann may have discovered it by playing poker
  - Karatsuba algorithm (1960)
    - Karatsuba discovered the algorithm at age 23

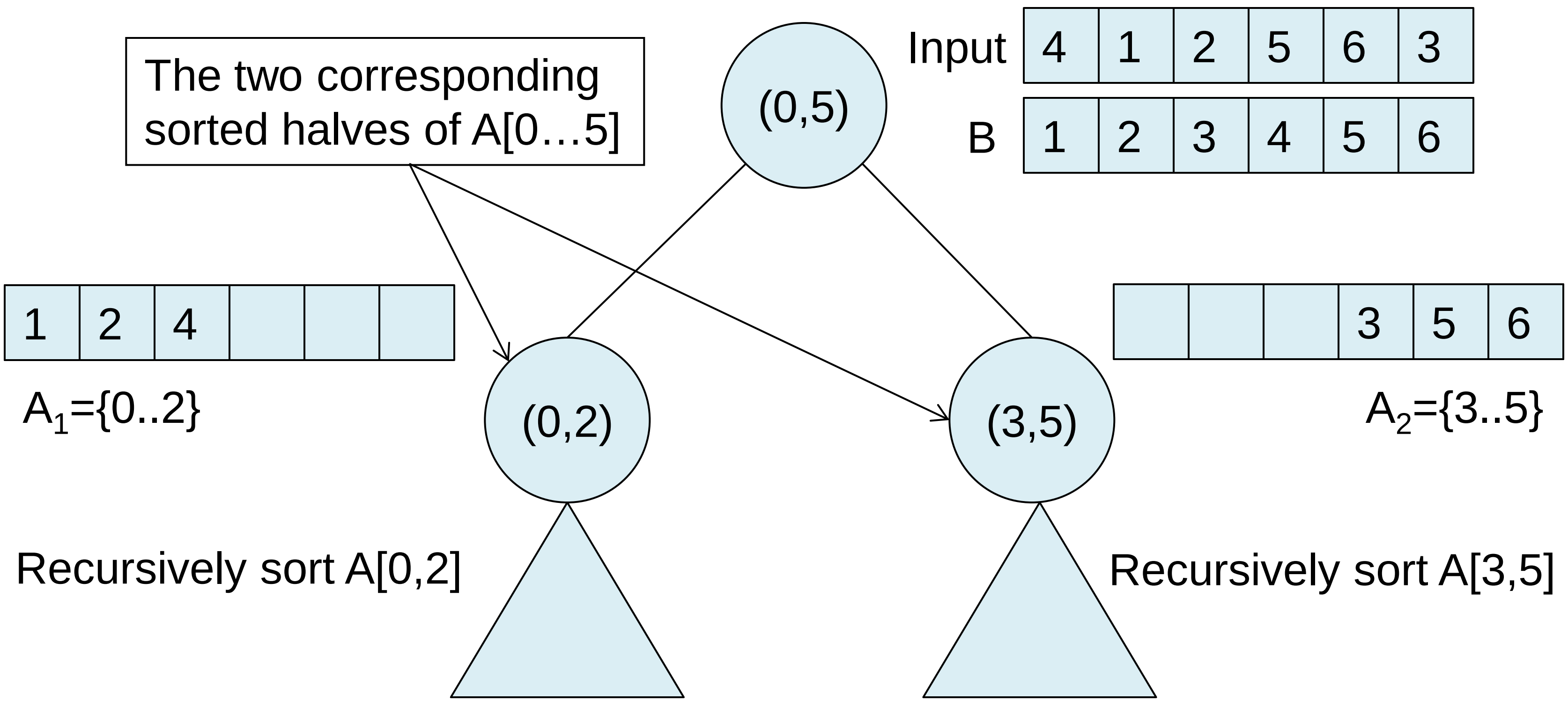
Merge Sort

- Given:
  - A[0..n-1]: array to be sorted
  - B[0..n-1]: buffer array (global variable)
  - Array of type `item`
- Apply idea of divide-and-conquer:
  - Recursively divide A[0..n-1] into two halves until each half has one element
  - Sort each half by merging its two corresponding sorted halves into one sorted list

An Illustration



Zoom-in to the Merge Step



"1" is the smallest element, so insert "1" to B[ ]

"2" is the smallest element, so insert "2" to B[ ]

"3" is the smallest element, so insert "3" to B[ ]

"4" is the smallest element, so insert "4" to B[ ]

A_1 becomes empty, so insert the remaining elements in A_2 to B[ ] one by one

Merge Sort (Code) (1/2)

```
void mergesort(item A[], int low, int up)
{
   if (low < up)
   {
      int m = (low+up)/2; //Integer division
      //Recursively divide A[0..n-1] into two halves
      //until each half has one element
      mergesort(A,low,m);
      mergesort(A,m+1,up);

      //Sort each half by merging its two corresponding
      //sorted halves into one sorted list
      int i=low, j=m+1, k=low;       //A1={i..m}; A2={j..up}
      while (i<=m && j<=up)          //Repeatedly append the
         if (A[i] < A[j])            //smallest element x in
            B[k++]=A[i++];           //A1 and A2 to B[] until
         else                        //A1 or A2 becomes empty
            B[k++]=A[j++];
```

Merge Sort (Code) (2/2)

```
      //A1 or A2 becomes empty
      while (i<=m)
         B[k++]=A[i++];
      while (j<=up)
         B[k++]=A[j++];

      for (k=low; k<=up; k++)
         A[k]=B[k];  //Copy back from B[] to A[]

   } //If low<up
}
//In main program
   mergesort(A,0,n-1);
```

Space Analysis

- Array B[0..n-1] requires O(n) space (in terms of words)
- Each recursive call requires constant number of variables for bookkeeping
- Let S'(n) = space complexity, excluding B[0..n-1]
- $S'(n) \leq S'(n/2) + c \leq \ldots = O(\log n)$
- Total space $S(n) = O(n) + S'(n)$
  $= O(n + \log n) = O(n)$

Time Analysis

Assume $n = 2^k$

$$T(n) \le 2T(n/2) + cn \quad \text{for some constant } c$$

$$2T(n/2) \le 2^2T(n/2^2) + cn$$

$$2^2T(n/2^2) \le 2^3T(n/2^3) + cn$$

$$\dots$$

$$+)\quad 2^{k-1}T(n/2^{k-1}) \le 2^kT(n/2^k) + cn$$

$$T(n) \le 2^kT(n/2^k) + k(cn)$$

$$= nT(1) + \log_2 n(cn)$$

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

Each recursion
(1) selects elements to B[] and (2) copy elements from B[] to A[]

$2[2T(n/2^2)+c(n/2)]$
$=2[2T(n/2^2)]+2[c(n/2)]$
$=2^2T(n/2^2)+cn$

$2^2[2T(n/2^3)+c(n/2^2)]$
$=2^2[2T(n/2^3)]+2^2[c(n/2^2)]$
$=2^3T(n/2^3)+cn$

$n = 2^k$
$\log_2 n = \log_2 2^k$
$\log_2 n = k\log_2 2$
$\log_2 n = k$

Merge Sort Theorem

Suppose n is a power of 2. Consider the recurrence relation of the form

$T(n) = 2\,T(n/2) + n$ if $n>1$ and $T(1)=0$.

Then $T(n) = n \log_2 n$.

[Quiz] Give a proof by induction.

Recall the Tower of Hanoi recurrence relation we saw earlier.

Fundamental Bound

- Theorem: Any *comparison-based* sorting algorithm requires $\Omega(n\log n)$ time
- Examples of comparison-based sorting algorithms:
  - Bubble Sort, Insertion Sort: $O(n^2)$ time
  - Merge Sort, Heap Sort (next lecture): $O(n\log n)$ time
  - Quick Sort (next lecture): $O(n^2)$ worst case, $O(n\log n)$ average case
- What are their common features?

Comparison-based Sorting Algorithms

- Variables and values are classified into 2 types:
  - Key type: `item`

    Only comparison or assignment allowed, e.g.,
    - `if (A[mid]==x)` [binary search]
    - `if (A[i]<A[j])` [merge sort]
    - `swap(A[j-1],A[j])` [bubble & insertion sort]

    Cannot create new values of key type, e.g.,
    `pivot=(A[i]+A[i+1])/2` not allowed

    Array index cannot be value of key type, e.g.,
    `B[A[i]]=A[i]` or `C[A[i]]++` not allowed
  - Other types

    No restrictions on the operations

Decision Tree Model

- Every comparison-based sorting algorithm can be abstracted as a *decision tree*
- *Comparison* of the order of two items constitutes *a single bit (binary digit)*
- A decision tree captures the comparisons of input values while ignoring all other computations (e.g., updating a loop counter, swapping a pair of input values, etc.)

An Example of Decision Tree

- Next page shows a decision tree for sorting 3 elements, X, Y and Z
- Each internal node is labeled by a pair of input values:

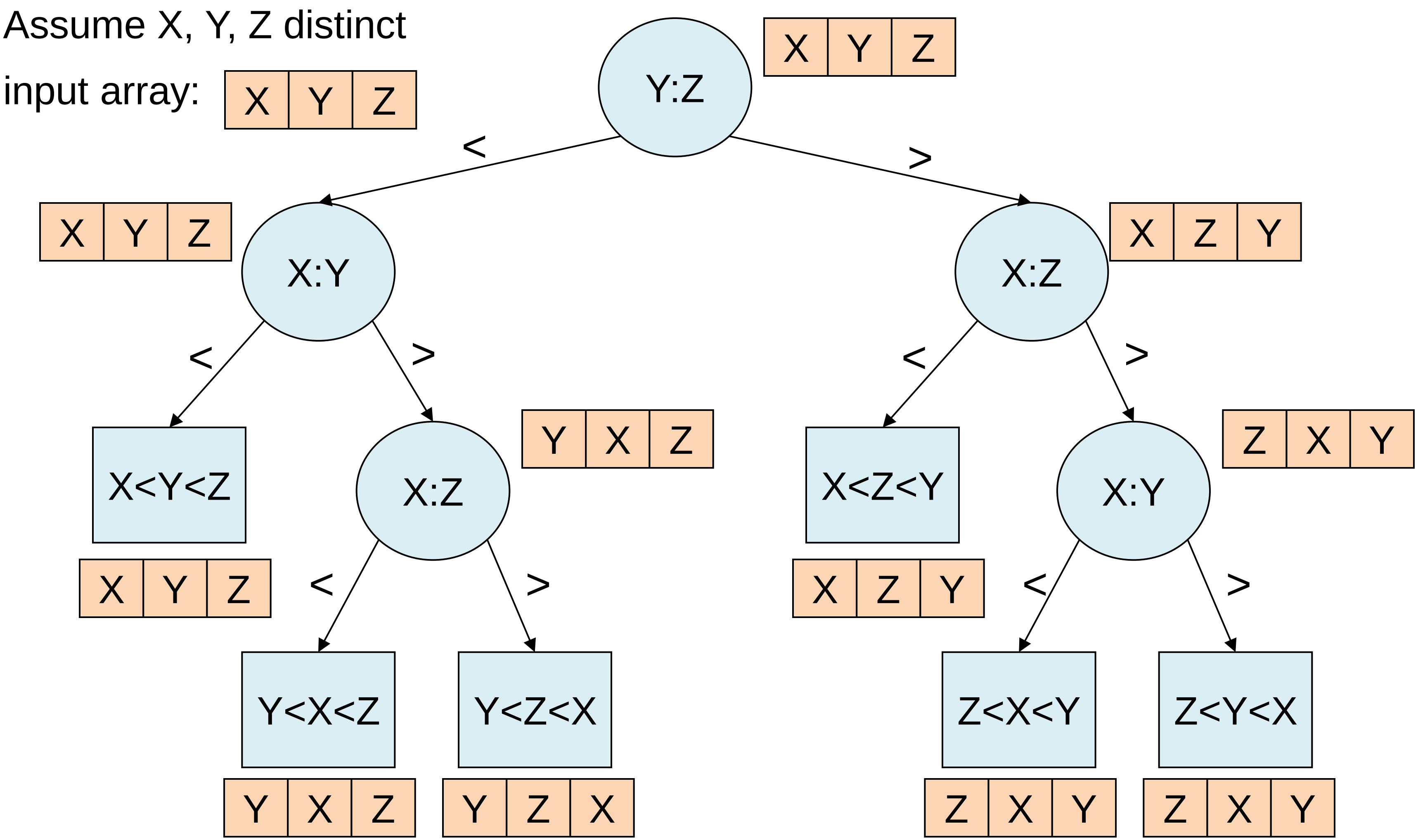
  a:b

- The two edges coming out of an internal node represent the 2 possible outcomes:

  a < b or a > b

- Each leave node is labeled by a permutation of the input values

A Decision Tree for n=3



How to Run a Decision Tree?

- Start from the root
- Repeat
  - Compare the two values mentioned in the current node (e.g., X:Y)
  - Follow the path according to outcome of the comparison to arrive at the next node (e.g., X<Y or Y>X)
- Until we arrive at a leaf node
- Report the permutation of the leaf as the answer

Worst-Case Running Time (1/4)

- How do we count the running time of a decision tree?
  - We only count the number of comparisons
  - E.g.: If the input is X<Z<Y, the decision tree in previous slide performs 2 comparisons
- What is the worst case running time of a decision tree?
  - **It is equal to the *height* of the tree, where *height* is the number of hops along the longest root-to-leaf path**

Worst-Case Running Time (2/4)

- Proof of Lower Bound:
  - Assume inputs are permutations of {1,…,n}
    - There are n! permutations.
  - Consider an arbitrary decision tree and calculate the *height* of the binary tree
    - Each tree has n! leaves, one for each permutation.
    - If the binary tree has height h, then it has ≤ 2^h leaves.
    - Thus, $2^h \geq n!$.

Worst-Case Running Time (3/4)

- So the tree must have height h large enough so that

$$2^h \geq n!$$

i.e., $h \geq \log_2(n!)$

$\geq \log_2[(n)(n\text{-}1)\ldots(n/2+1)(n/2)(n/2\text{-}1)\ldots(1)]$

$\geq \log_2[\underbrace{(n)(n\text{-}1)\ldots(n/2+1)}]$

(only keep the first n/2 terms)

$\geq \log_2[(n/2)(n/2)\ldots(n/2)]$

Reduce each term to n/2

$\geq \log_2(n/2)^{n/2}$

$\geq (n/2)\log_2(n/2)$

$\geq (n/2)(\log_2 n\text{-}\log_2 2)$

$\geq (n/2)(\log_2 n)\text{-}(n/2)(\log_2 2)$

$\geq (n/2)(\log_2 n)\text{-}n/2$

$= \Omega(n\log n)$

– Worst case time complexity

Worst-Case Running Time (4/4)

- Another proof using Stirling's Approximation (as n tends to infinity):

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

How many trailing zeros does this number have?

(first stated in 1733 by Abraham de Moivre and later refined by James Stirling)

- So the tree must have height h large enough so that

$$\begin{aligned}\log_2(2^h) &\geq \log_2(n!)\\ &\geq \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)\\ &= \log_2(\sqrt{2\pi n}) + n\log_2 n - n\log_2 e\\ &= \Omega(n\log n)\end{aligned}$$

– What is the physical meaning of $\log_2(n!)$?

Counting Inversions

- Music site tries to match your song preferences with others.
  - You rank n songs.
  - Site consults database to find people with similar tastes → Recommender System

- Similarity metric: number of inversions between two rankings.
  - My rank: 1, 2, …, n.
  - Your rank: $a_1, a_2, \ldots, a_n$.
  - Songs i and j inverted if $i < j$, but $a_i > a_j$.

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

<u>Inversions</u>

3-2, 4-2

- Brute force: check all $\Theta(n^2)$ pairs i and j. [Quiz] Write down this algorithm

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.

1	5	4	8	10	2	6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - Divide: separate list into two pieces.

1	5	4	8	10	2	6	9	12	11	3	7

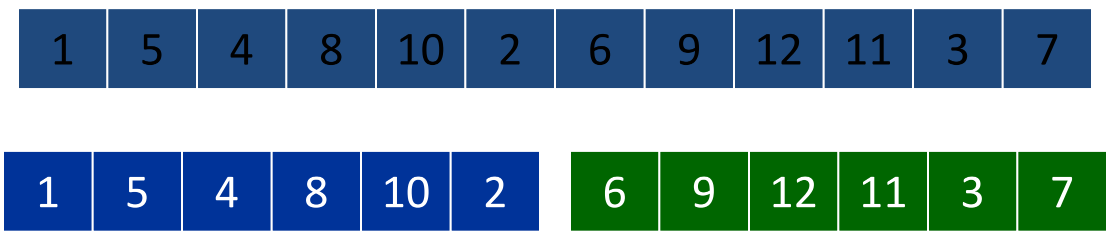
Divide: O(1).

1	5	4	8	10	2

6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.



Divide: O(1).

Conquer: 2T(n / 2)

5 blue-blue inversions

5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

– Divide: separate list into two pieces.

– Conquer: recursively count inversions in each half.

– Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2

5 blue-blue inversions

6	9	12	11	3	7

8 green-green inversions

Conquer: 2T(n / 2)

9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole.

3	7	10	14	18	19	2	11	16	17	23	25
						6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: O(n)

2	3	7	10	11	14	16	17	18	19	23	25

Merge: O(n)