CMP 2202

LECTURE 4

Divide and Conquer

- Merge Sort
- Counting Inversions
- Binary Search
- Exponentiation

Solving Recurrences

- Recursion Tree Method

Divide and Conquer

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

- Most common usage.
  - Break up problem of size n into **two** equal parts of size n/2.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in **linear time**.
- Consequence.
  - Brute force: $\Theta(n^2)$.
  - Divide-and-conquer: Θ (n log n).

Sorting

- Given n elements, rearrange in ascending order.
- Applications.
  - Sort a list of names.
  - Display Google PageRank results.

obvious applications

  - Find the median.
  - Find the closest pair.
  - Binary search in a database.
  - Find duplicates in a mailing list.

problems become easy once items are in sorted order

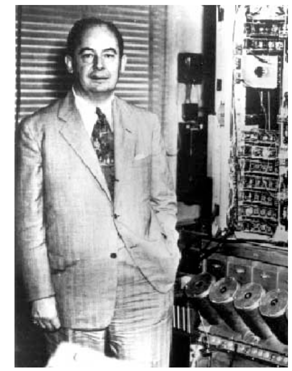
  - Data compression
  - Computer graphics
  - Computational biology.
  - Load balancing on a parallel computer.

  . . .

non-obvious applications

Mergesort

– Divide array into two halves.
– Recursively sort each half.
– Merge two halves to make sorted whole.



Jon von Neumann (1945)

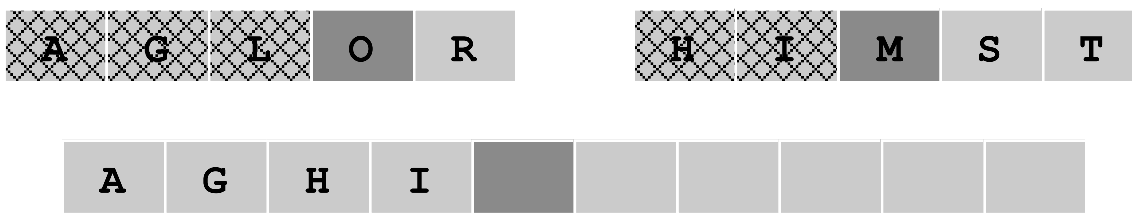
A	L	G	O	R	I	T	H	M	S

A	L	G	O	R		I	T	H	M	S	divide	$O(1)$
A	G	L	O	R		H	I	M	S	T	sort	$2T(n/2)$

A	G	H	I	L	M	O	R	S	T	merge	$O(n)$

Merging

- Combine two pre-sorted lists into a sorted whole.
- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.



- Challenge for the bored: in-place merge [Kronrud, 1969]

using only a constant amount of extra storage

Recurrence for Mergesort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

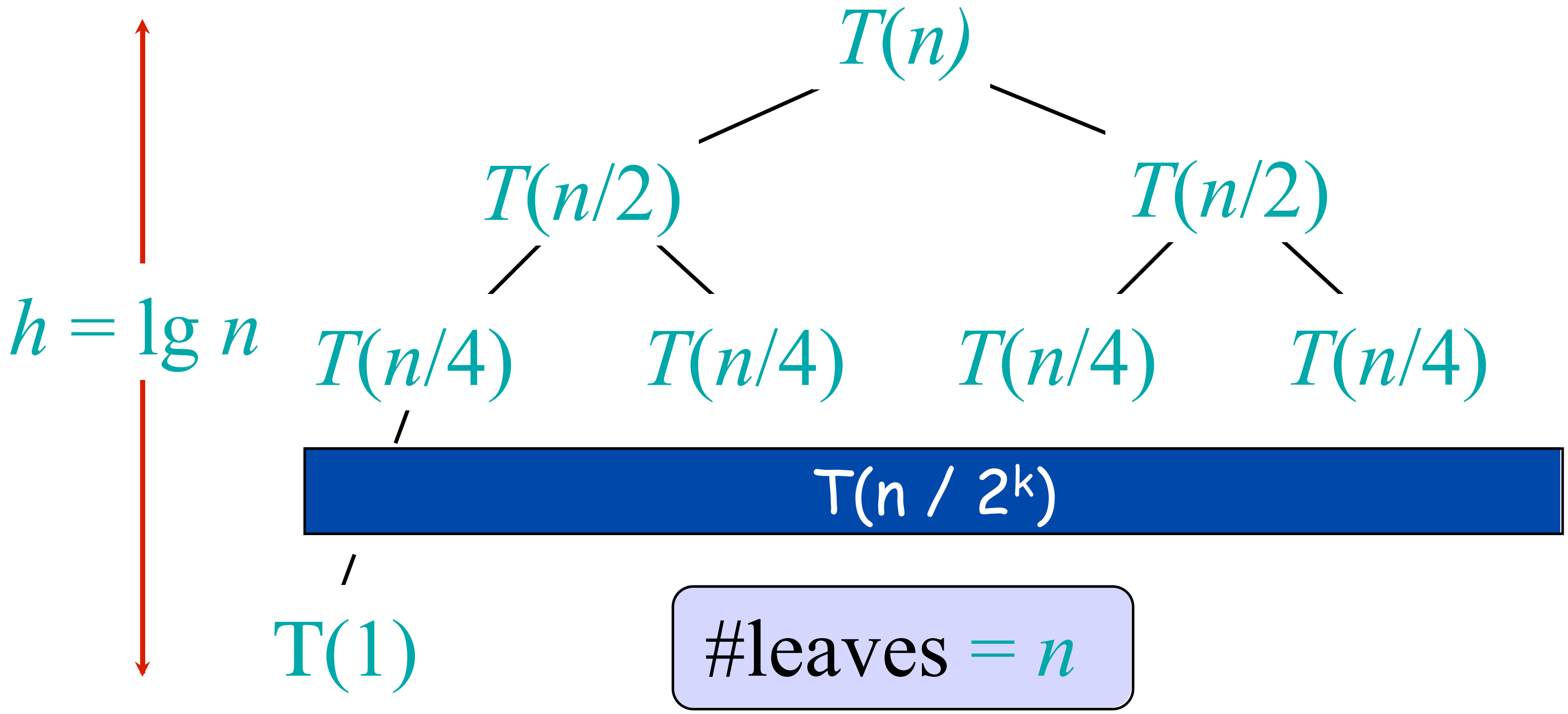
- T(n) = worst case running time of Mergesort on an input of size n.
- Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.
- Usually omit the base case because our algorithms always run in time $\Theta(1)$ when n is a small constant.
- Several methods to find an upper bound on $T(n)$.

Recursion Tree Method

- Technique for guessing solutions to recurrences
  - Write out tree of recursive calls
  - Each node gets assigned the work done during that call to the procedure (dividing and combining)
  - Total work is **sum** of work at all nodes
- After guessing the answer, can prove by induction that it works.

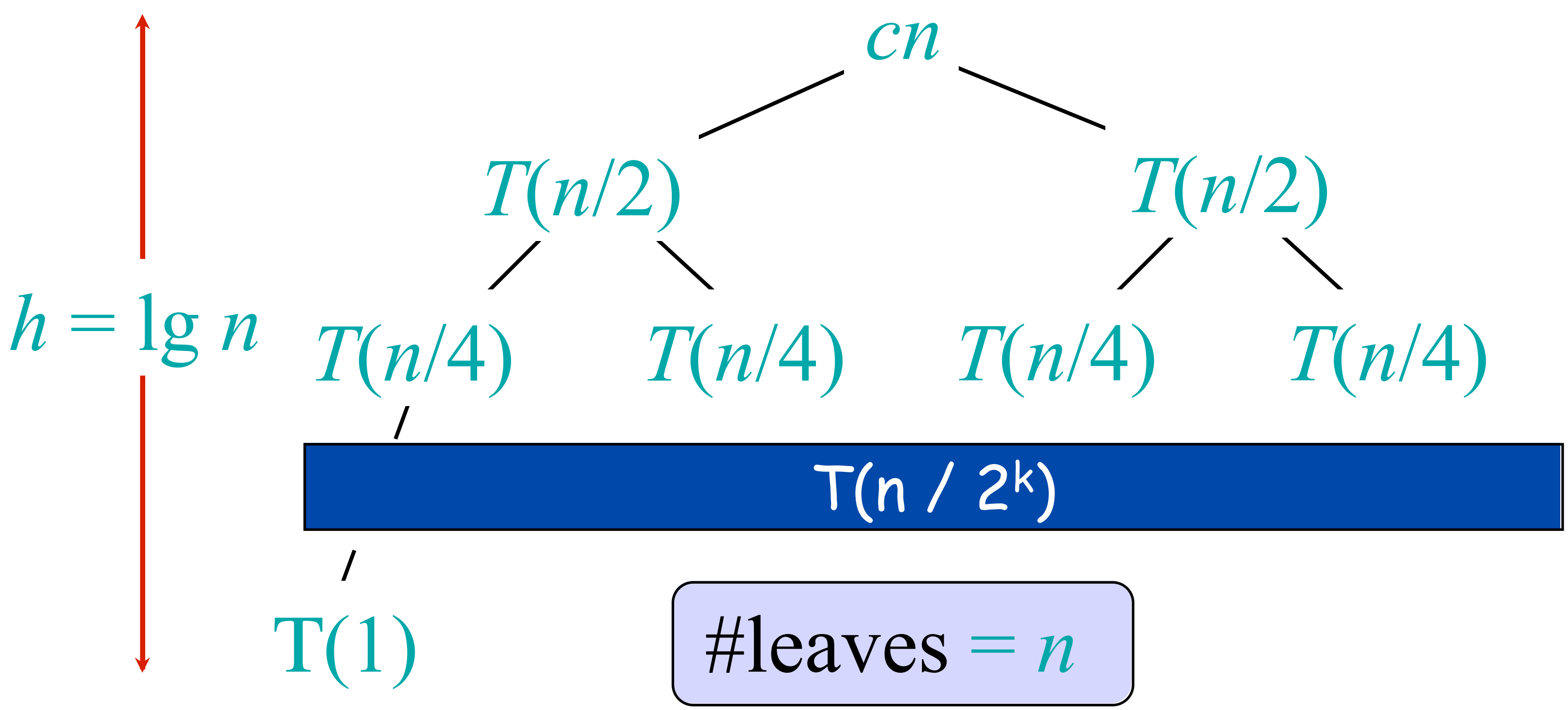
Recursion Tree for Mergesort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



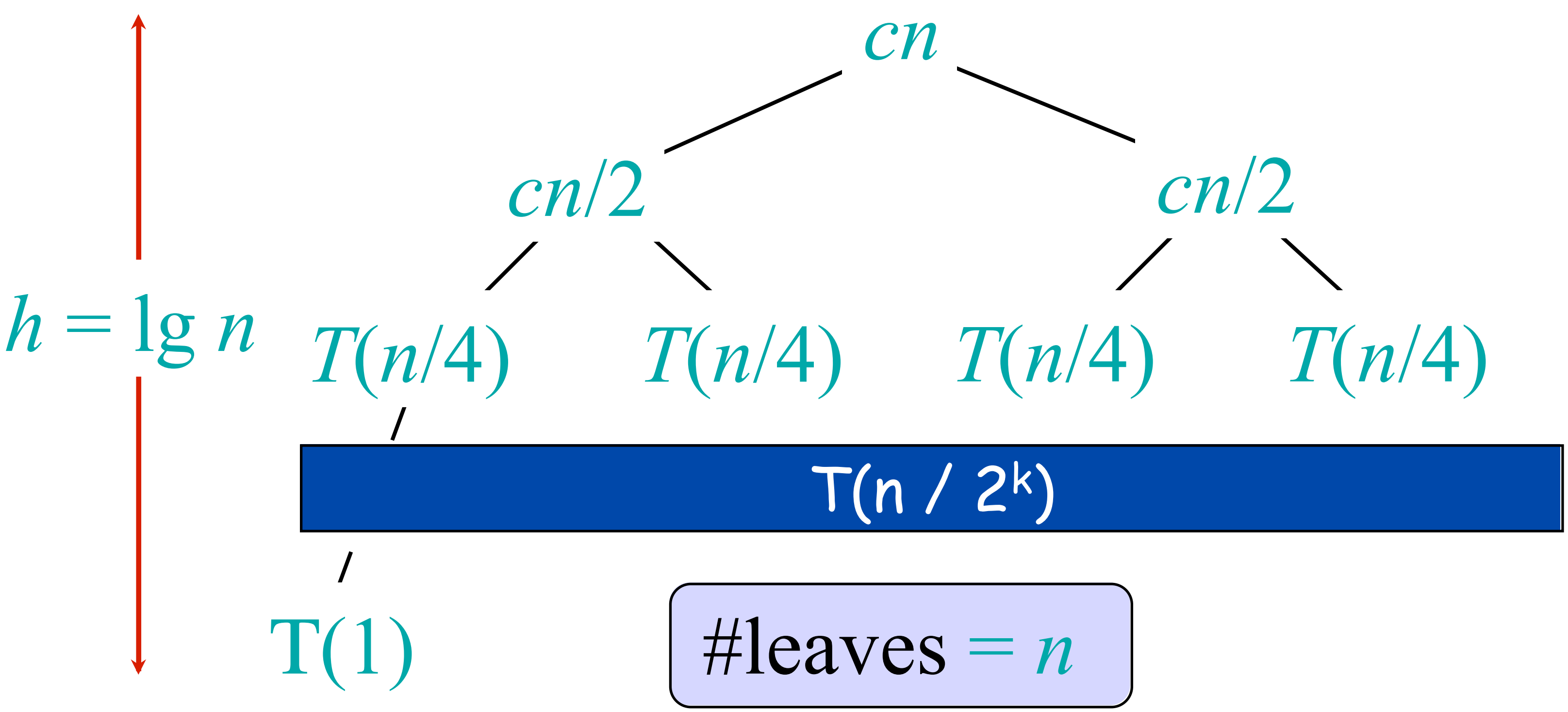
Recursion Tree for Mergesort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



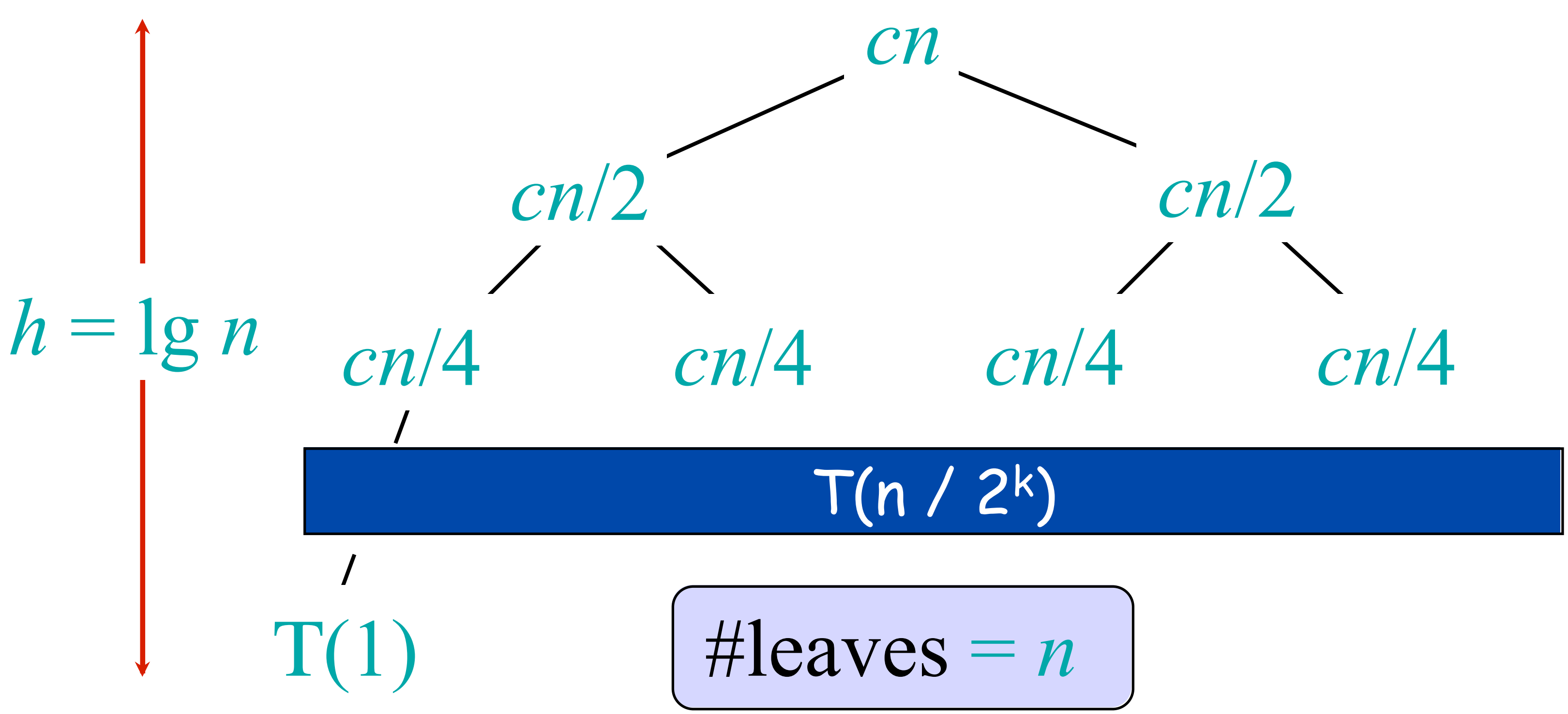
Recursion Tree for Mergesort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



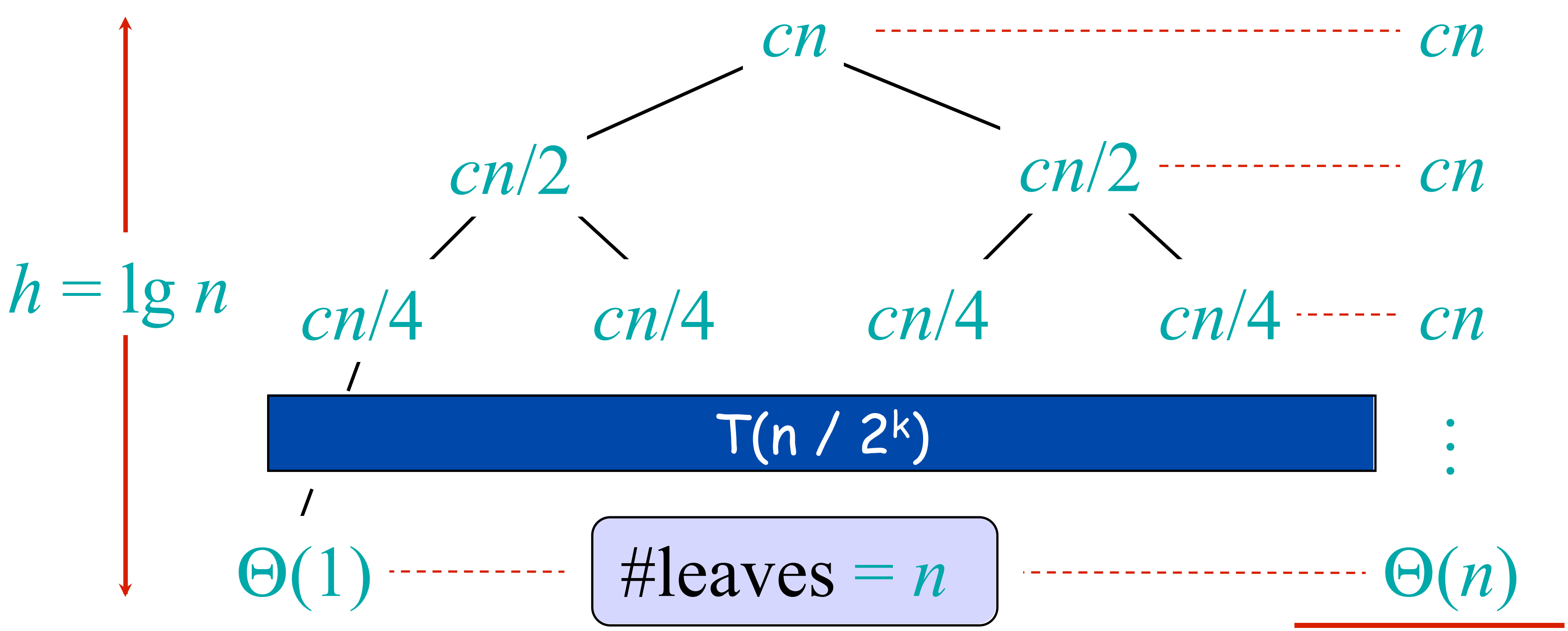
Recursion Tree for Mergesort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion Tree for Mergesort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Total $= \Theta(n \lg n)$

Counting Inversions

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

Applications

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).

Counting Inversions: Algorithm

- Divide-and-conquer

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

Counting Inversions: Algorithm

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

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

1	5	4	8	10	2

6	9	12	11	3	7

Divide: $\Theta(1)$.

Counting Inversions: Algorithm

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

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

Divide: $\Theta(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: $2T(n / 2)$

5 blue-blue inversions

8 green-green inversions

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

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

Counting Inversions: Algorithm

- 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: $\Theta(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: $2T(n / 2)$

5 blue-blue inversions

8 green-green inversions

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.

to maintain sorted invariant

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: $\Theta(n)$

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

Merge: $\Theta(n)$

$T(n) = 2T(n/2) + \Theta(n)$. Solution: $T(n) = \Theta(n \log n)$.

Implementation

- Pre-condition. [Merge-and-Count] A and B are sorted.
- Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (rA, A) ← Sort-and-Count(A)
   (rB, B) ← Sort-and-Count(B)
   (r , L) ← Merge-and-Count(A, B)

   return r = rA + rB + r and the sorted list L
}
```

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1.Divide: Check middle element.

2.Conquer: Recursively search 1 subarray.

3.Combine: Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary Search

BINARYSEARCH(b, $A[1\ ..\ n]$) $\triangleright$ find b in **sorted** array A

1. If n=0 then return "not found"
2. If $A[\lceil n/2 \rceil] = b$ then return $\lceil n/2 \rceil$
3. If $A[\lceil n/2 \rceil] < b$ then
4.   return BINARYSEARCH($A[1\ ..\ \lceil n/2 \rceil]$)
5. Else
6.   return $\lceil n/2 \rceil$ + BINARYSEARCH($A[\lceil n/2 \rceil+1\ \ ..\ \ n]$)

Recurrence for binary search

$$T(n) = 1\,T(n/2) + \Theta(1)$$

 # subproblems

subproblem size

work dividing and combining

Recurrence for binary search

$$T(n) = 1\,T(n/2) + \Theta(1)$$

 # subproblems

subproblem size

work dividing and combining

$$\Rightarrow \quad T(n) = T(n/2) + c = T(n/4) + 2c$$

$$\cdots$$

$$= c\lceil \log n \rceil = \Theta(\lg n)\ .$$

Exponentiation

Problem: Compute a^b, where $b \in \mathbb{N}$ is n bits long

Question: How many multiplications?

Naive algorithm: $\Theta(b) = \Theta(2^n)$ (exponential in the input length!)

Divide-and-conquer algorithm:

$$a^b = \begin{cases} a^{b/2} \cdot a^{b/2} & \text{if } b \text{ is even;} \\ a^{(b-1)/2} \cdot a^{(b-1)/2} \cdot a & \text{if } b \text{ is odd.} \end{cases}$$

$$T(b) = T(b/2) + \Theta(1) \Rightarrow T(b) = \Theta(\log b) = \Theta(\text{n}) .$$

So far: 2 recurrences

- Mergesort; Counting Inversions

  $T(n) = 2\,T(n/2) + \Theta(n) \qquad = \Theta(n \log n)$

- Binary Search; Exponentiation

  $T(n) = 1\,T(n/2) + \Theta(1) \qquad = \Theta(\log n)$

Master Theorem: method for solving recurrences.