Lecture 4:
Divide & Conquer

THE UNIVERSITY OF SYDNEY

General techniques in this course

- Greedy algorithms
- **Divide & Conquer algorithms**
- Sweepline algorithms
- Dynamic programming algorithms
- Network flow algorithms

Divide-and-Conquer

- **Divide-and-conquer** [usually 3 parts]
  1. Divide: Break up problem into several parts.
  2. Conquer: Solve each part recursively.
  3. 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.

Searching

Input: A sorted sequence S of n numbers $a_1, a_2, \ldots, a_n$,
stored in an array A[1..n].

Question: Given a number x, is x in S?

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then “Yes”
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example: x=1 (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example: x=1 (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1...n]

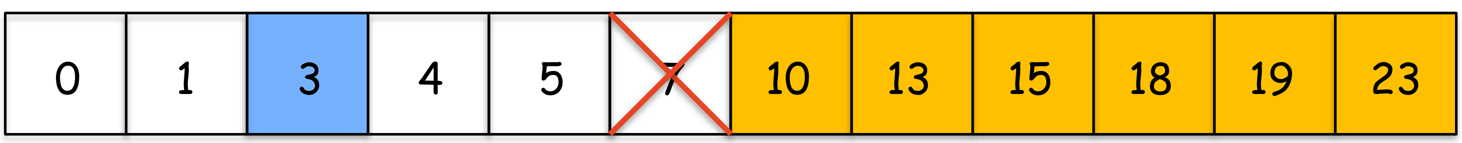
Example: x=1 (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example: x=1 (non-integers are rounded up)



Binary Search

- Compare x to the middle element of the array ($A[n/2]$).
- If $A[n/2] = x$ then "Yes"
- Otherwise, if $A[n/2] > x$ then recursively Search $A[1 \ldots n/2-1]$.
- Otherwise, if $A[n/2] < x$ then recursively Search $A[n/2+1 \ldots n]$

Example: $x=1$ (non-integers are rounded up)

0	1	3	4	5	~~7~~	10	13	15	18	19	23

Binary Search

– Compare x to the middle element of the array (A[n/2]).
– If A[n/2] = x then "Yes"
– Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
– Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example: x=1 (non-integers are rounded up)

0	1	~~3~~	4	5	~~7~~	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example: x=1 (non-integers are rounded up)

~~0~~	1	~~3~~	4	5	~~7~~	10	13	15	18	19	23

Binary Search

– Compare x to the middle element of the array (A[n/2]).
– If A[n/2] = x then "Yes"
– Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
– Otherwise, if A[n/2] < x then recursively Search A[n/2+1...n]

Example: x=1 (non-integers are rounded up)

~~0~~	1	~~3~~	4	5	~~7~~	10	13	15	18	19	23

Analysis: $T(n) = 1 + T(n/2)$

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

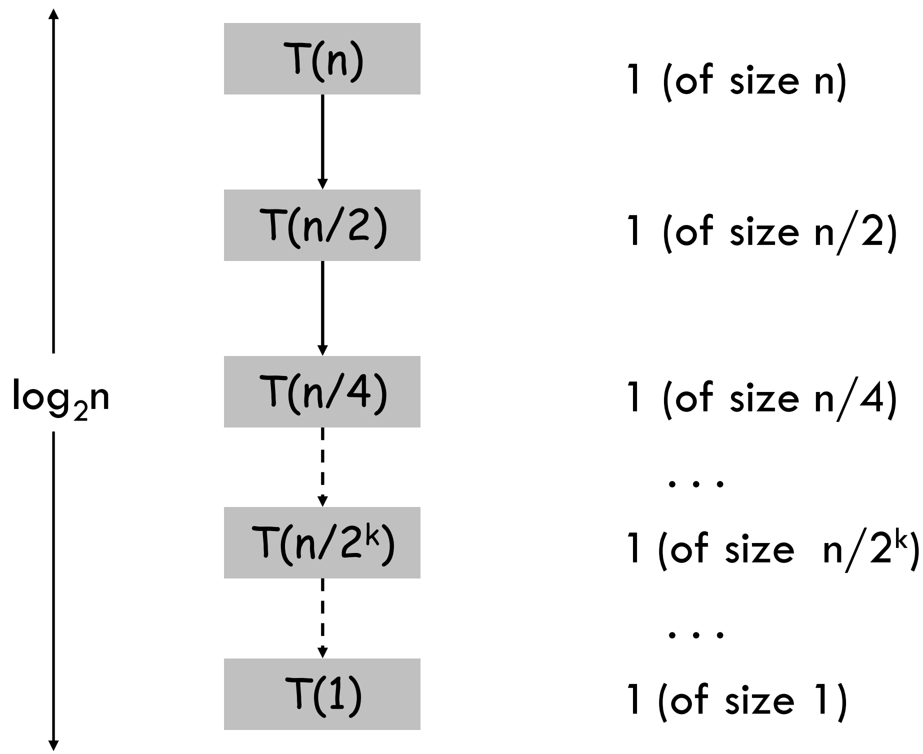
Example: x=1 (non-integers are rounded up)

~~0~~	1	~~3~~	4	5	~~7~~	10	13	15	18	19	23

Analysis: $T(n) = 1 + T(n/2)$

Analyze recursion

$$T(n) = T(n/2) + O(1)$$



Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example: x=1 (non-integers are rounded up)

~~0~~	1	~~3~~	4	5	~~7~~	10	13	15	18	19	23

Analysis: $T(n) = 1 + T(n/2) = O(\log n)$

Sorting

– Sorting. Given n elements, rearrange in ascending order.

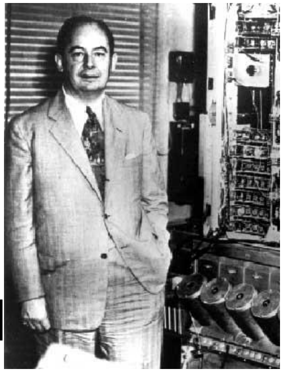
Obvious sorting applications.
- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing.
- ...

Mergesort

1. Divide array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge two halves to make sorted whol



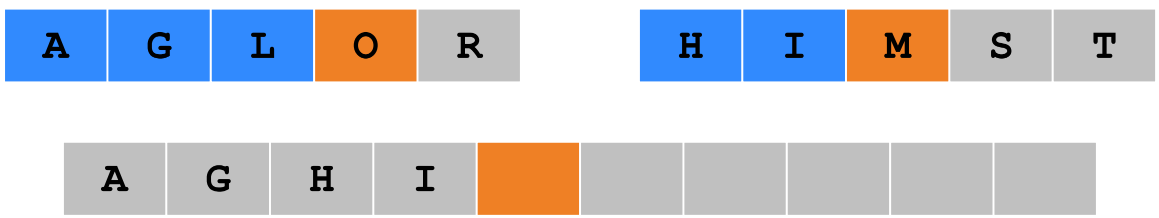
Jon von Neumann (1945)

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

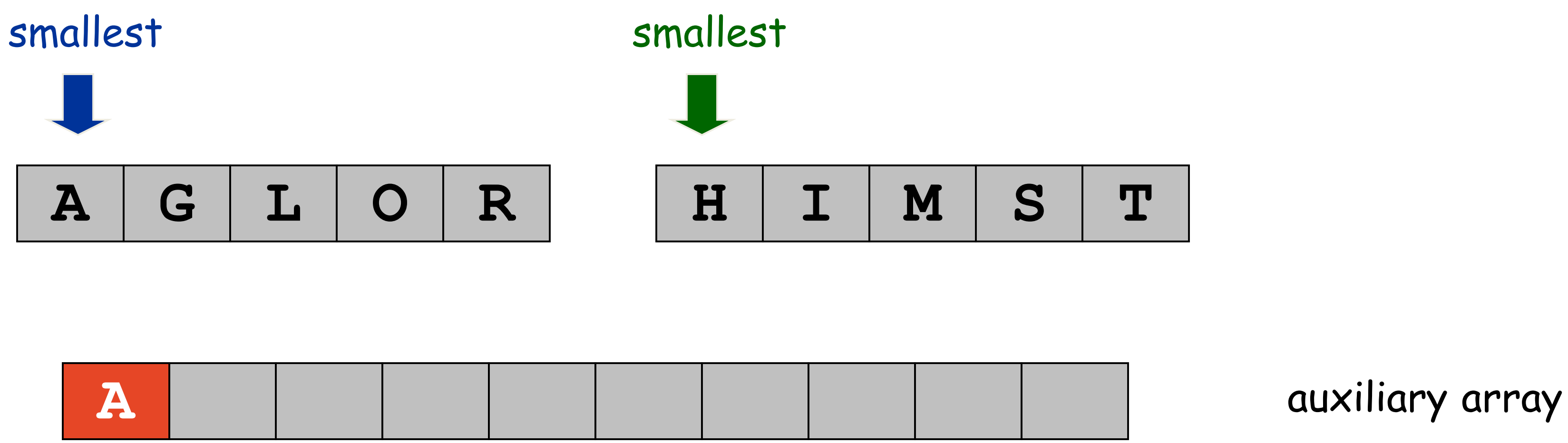
- Merging. Combine two pre-sorted lists into a sorted whole.

- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.



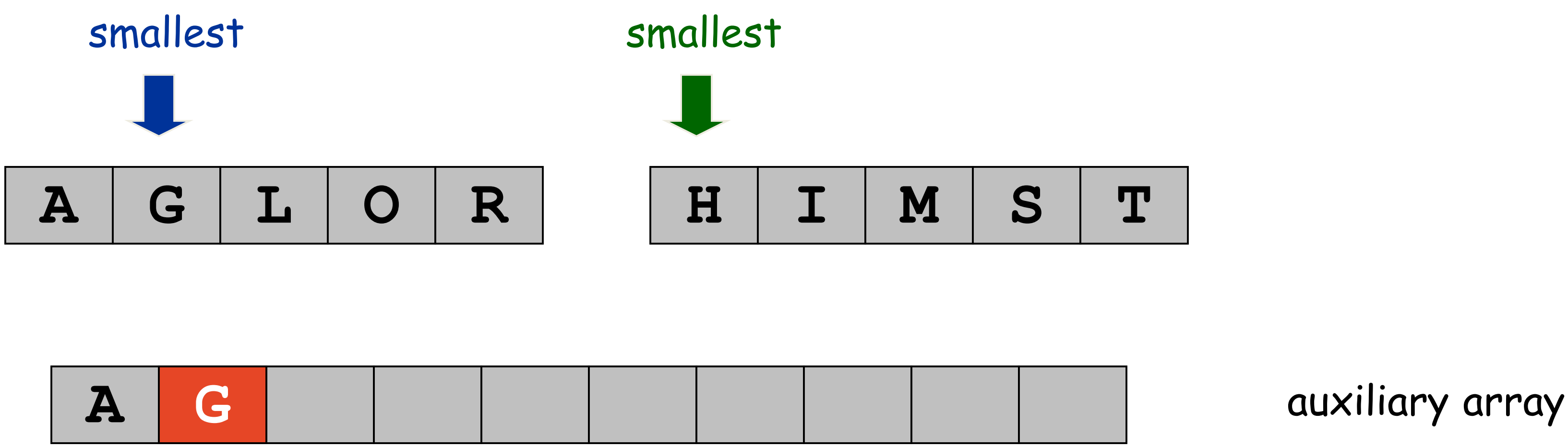
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



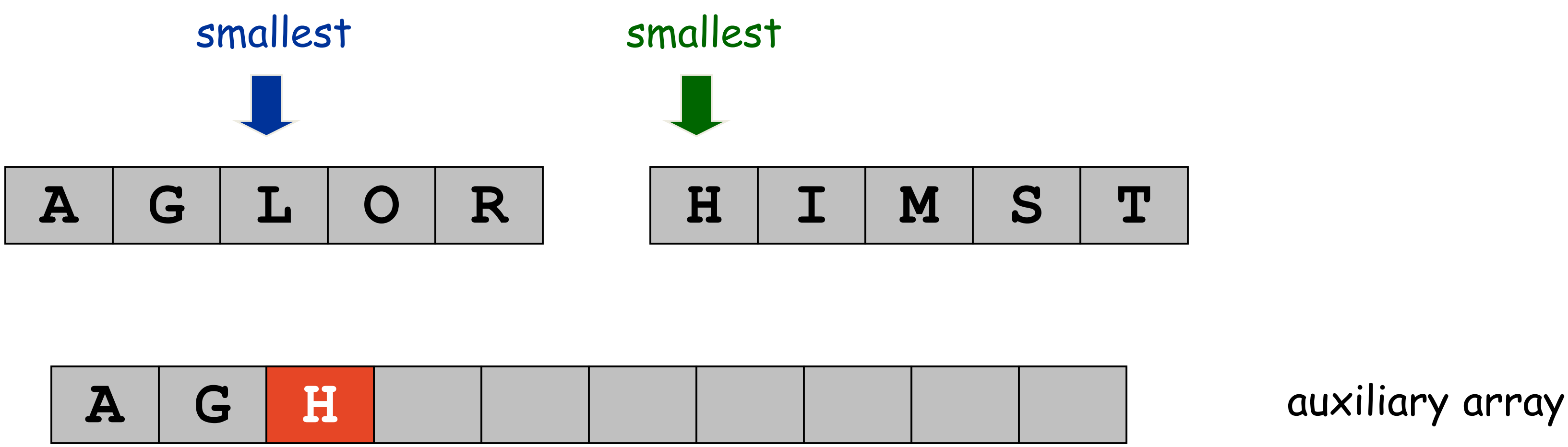
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



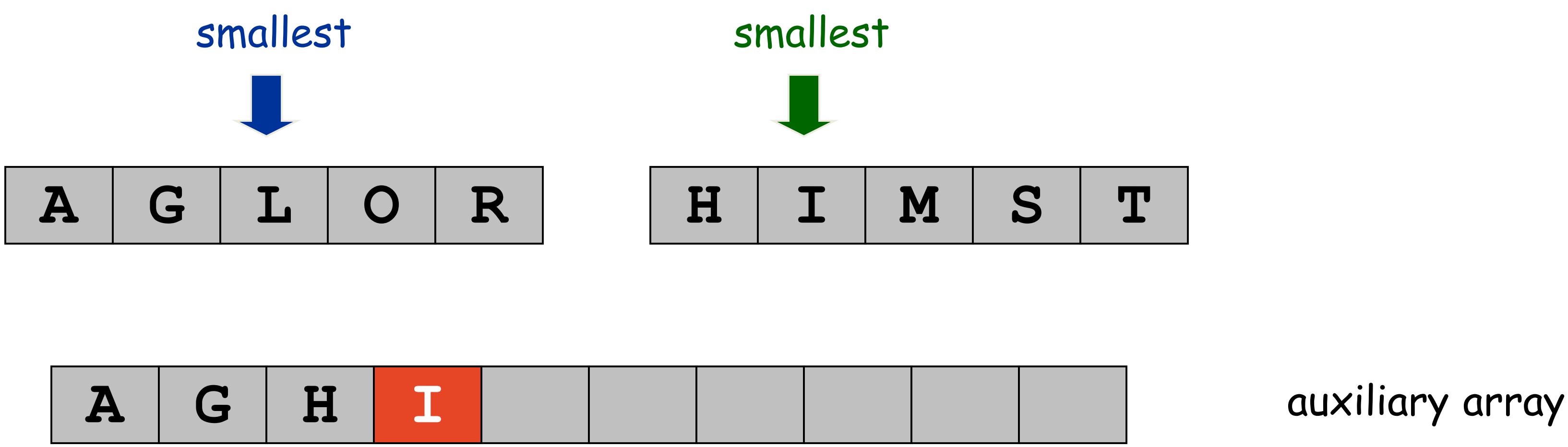
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



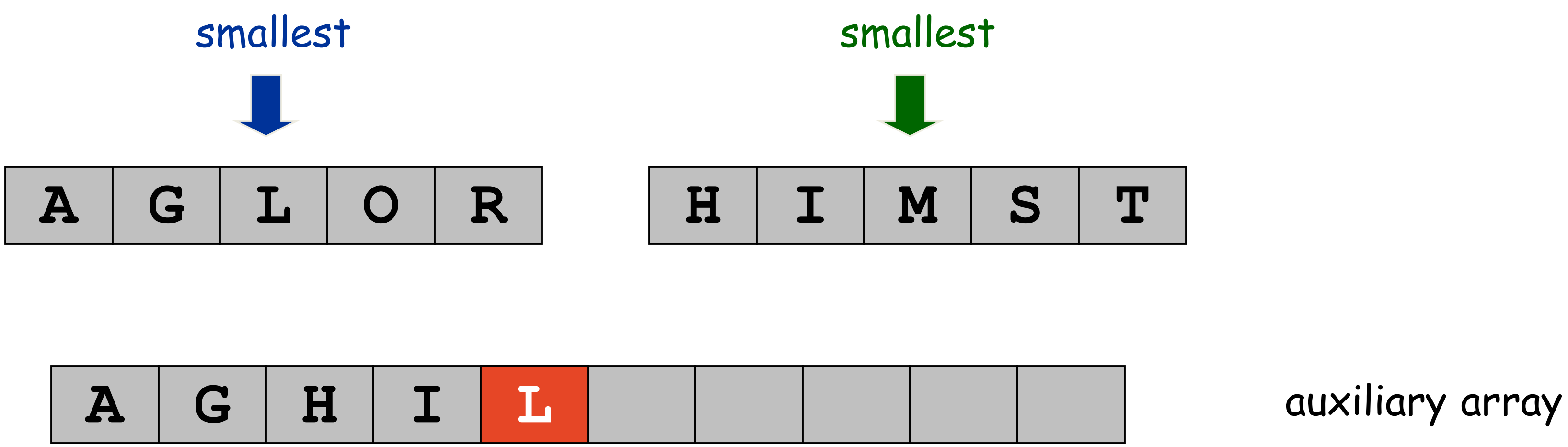
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



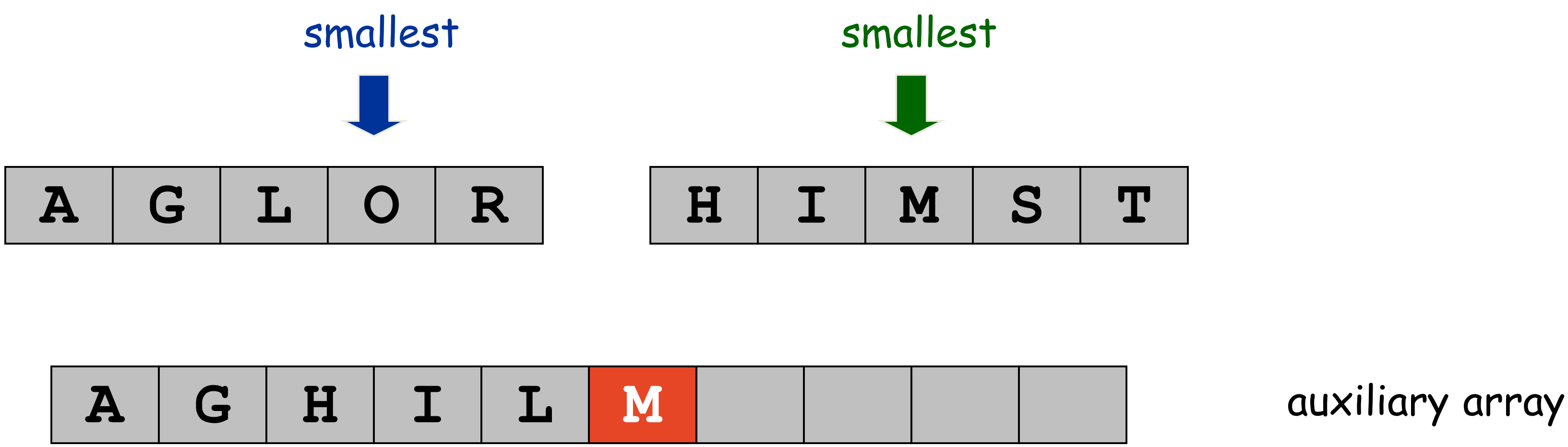
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



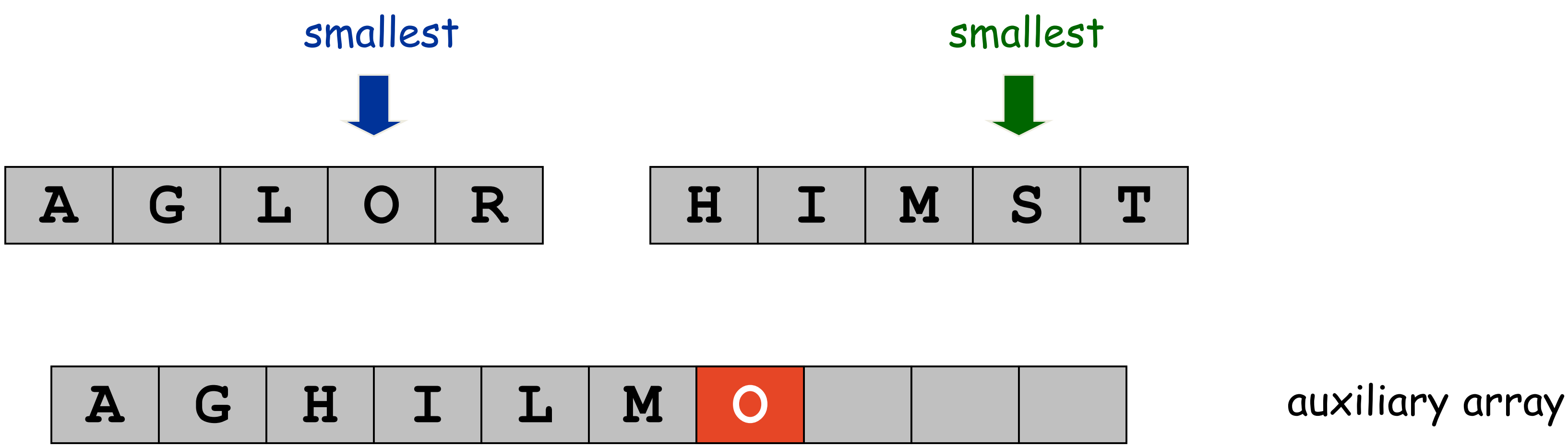
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



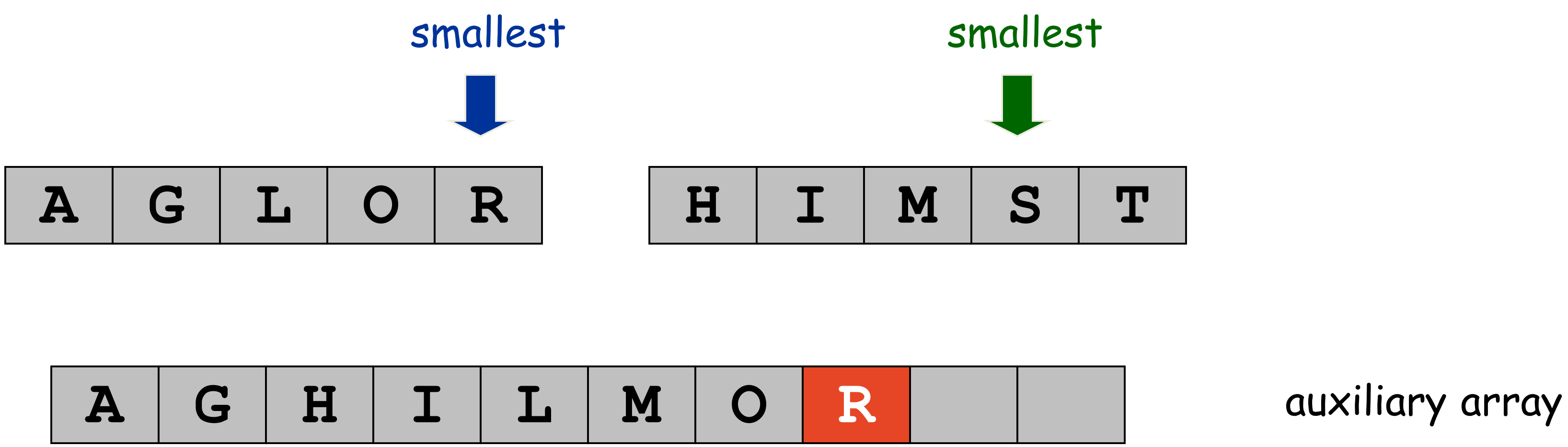
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



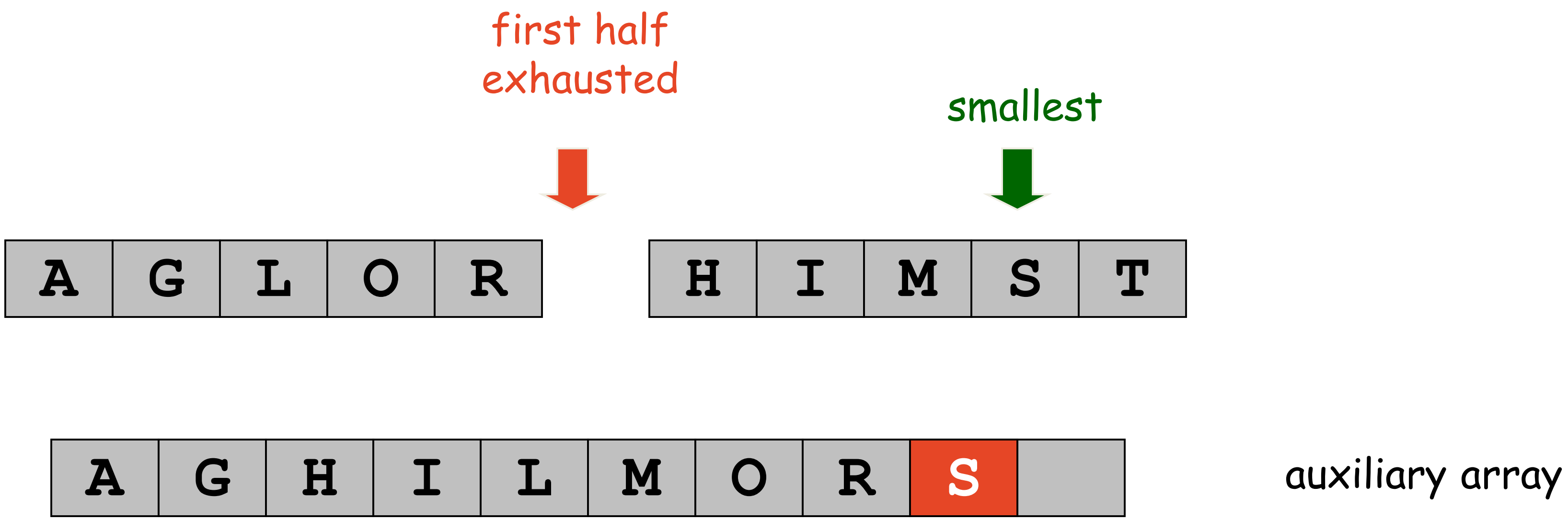
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



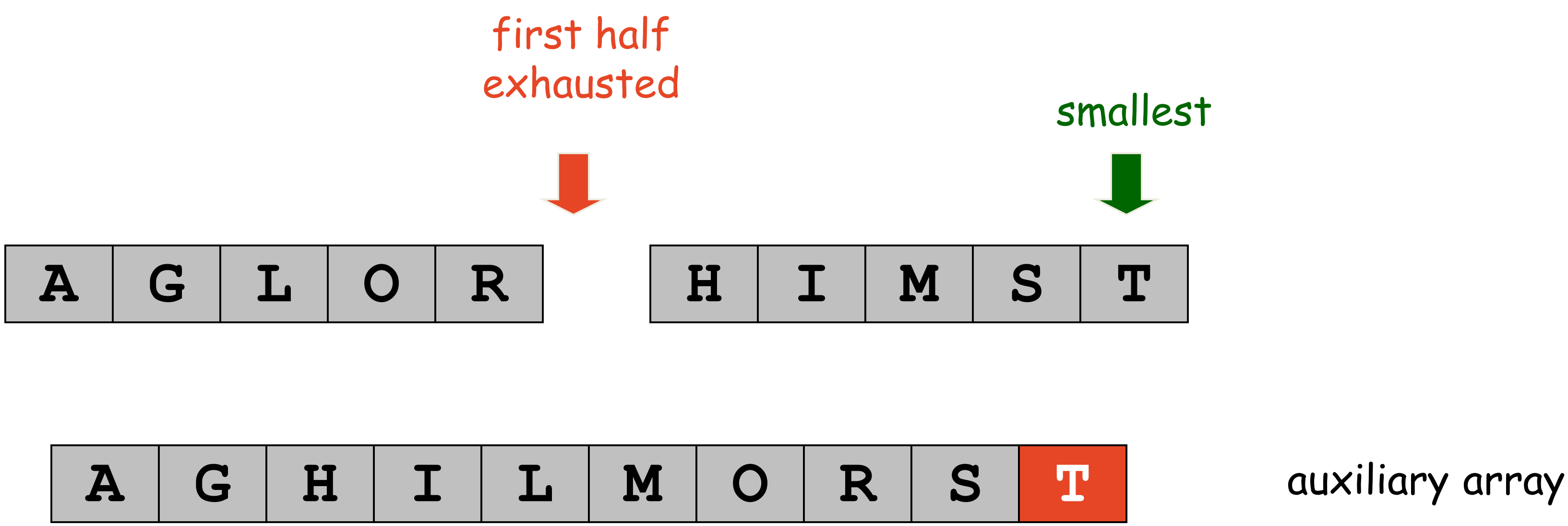
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



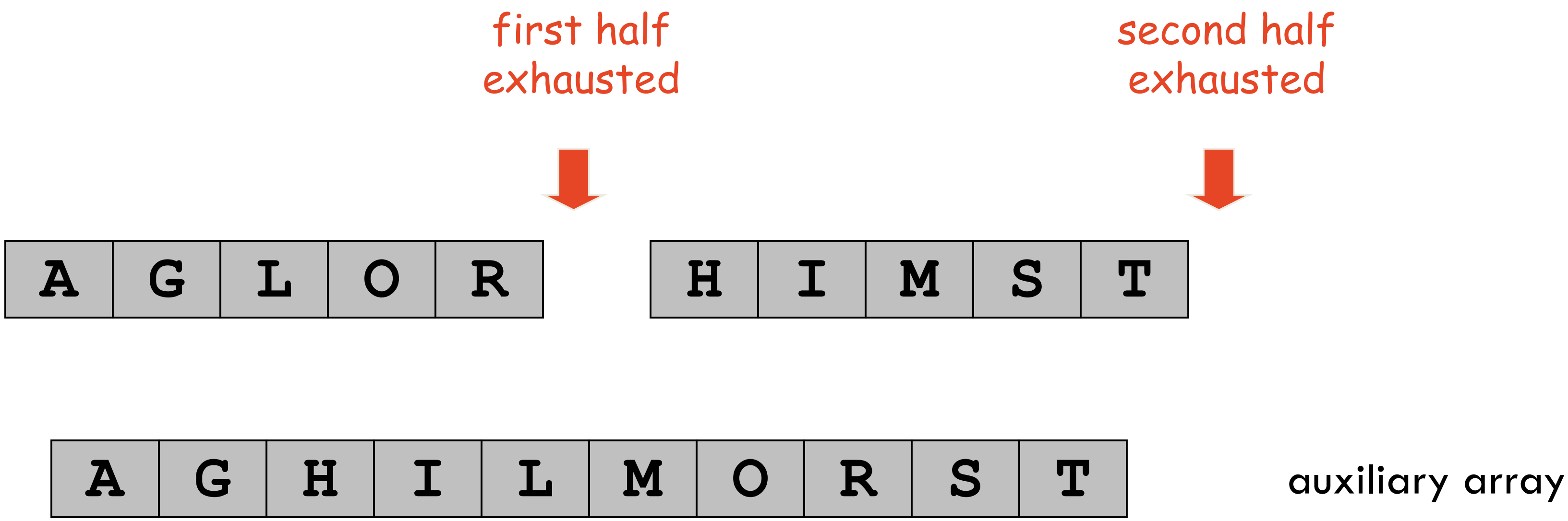
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



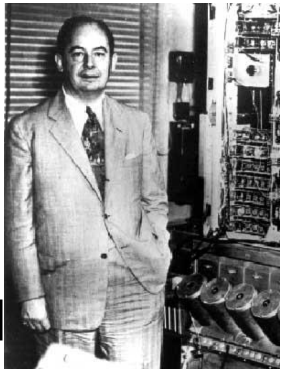
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Mergesort

1. Divide array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge two halves to make sorted whol



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)

A Useful Recurrence Relation

- **Definition:** $T(n)$ = number of comparisons to mergesort an input of size n.

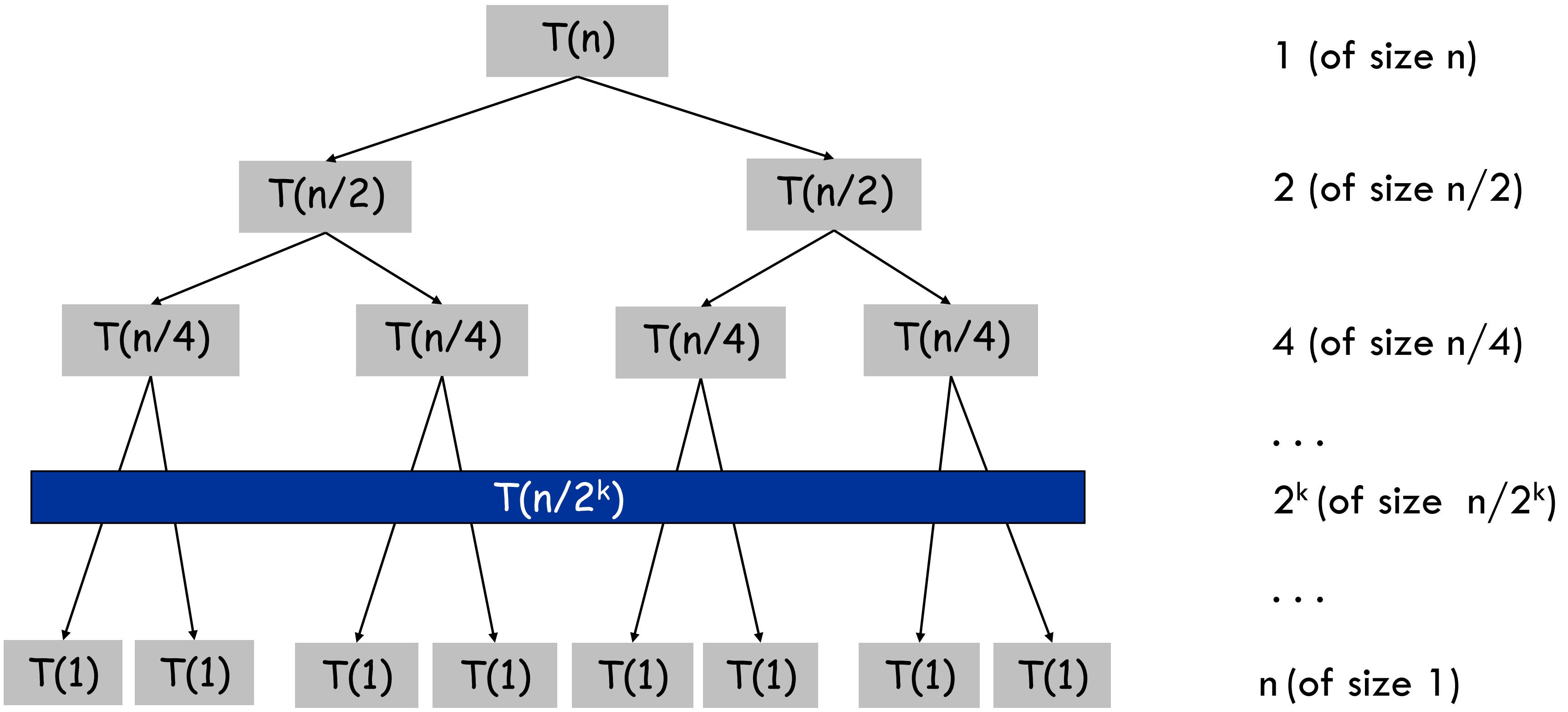
- **Mergesort recurrence.**

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & \text{otherwise} \end{cases}$$

- **Solution:** $T(n) = ?$

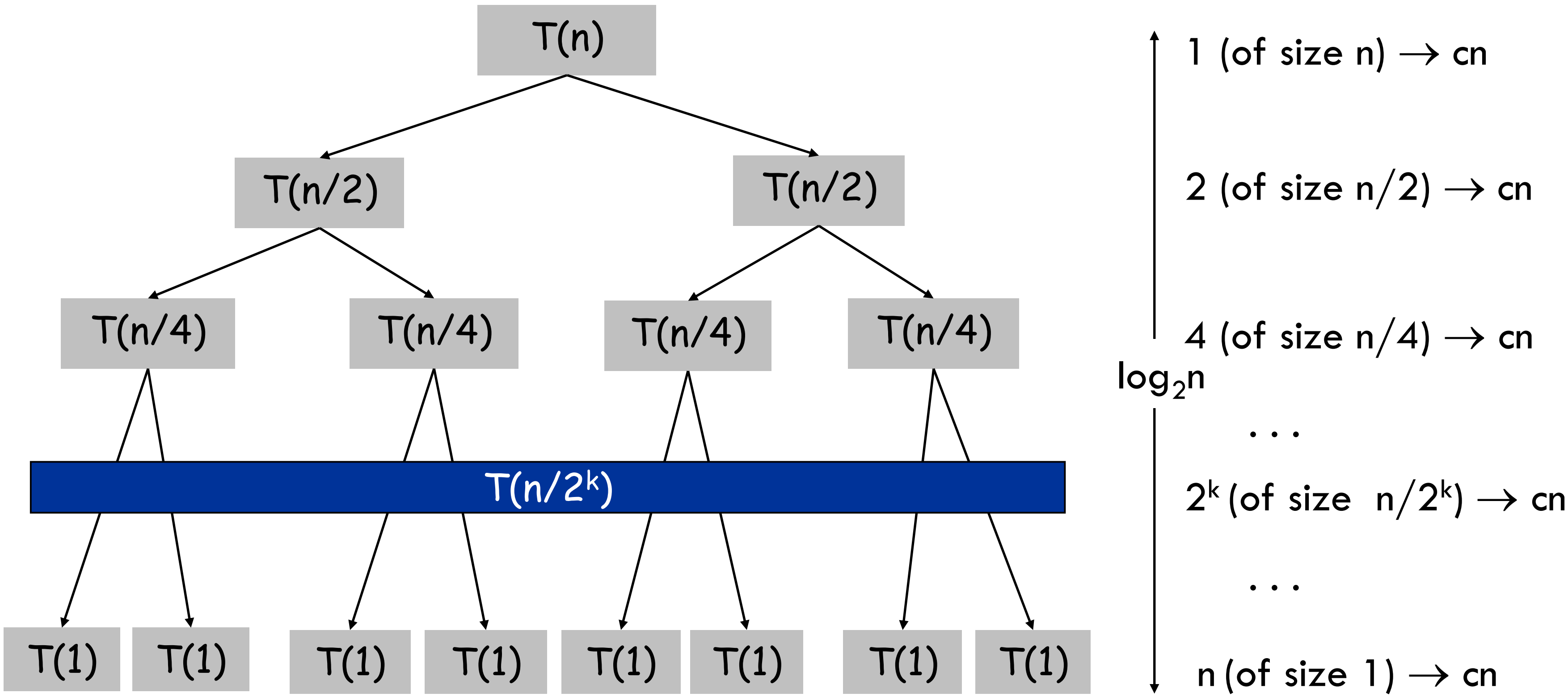
Proof by unrolling

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



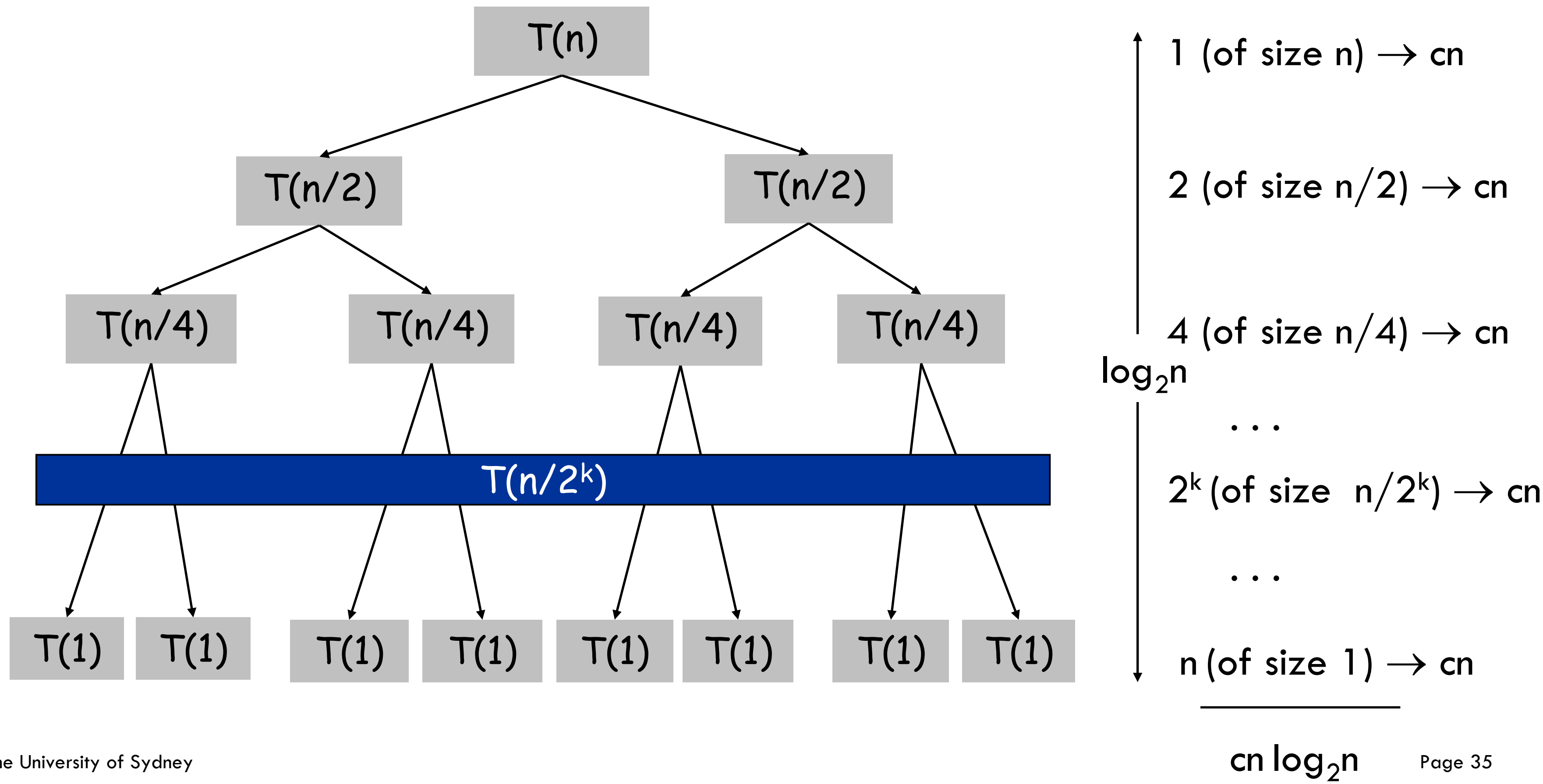
Proof by unrolling

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



Proof by unrolling

$$T(n)=\begin{cases} c & \text{if } n=1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



A Useful Recurrence Relation

- **Definition:** T(n) = number of comparisons to mergesort an input of size n.

- **Mergesort recurrence.**

$$
T(n) \le \begin{cases} 0 & \text{if } n=1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}
$$

- **Solution:** $T(n) = O(n \log_2 n)$.

Counting Inversions

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 k inverted if $i < k$, but $a_i > a_k$.

Songs

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

Inversions

3-2, 4-2

- Brute force: check all $\Theta(n^2)$ pairs i and k.

Applications

- 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: 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.

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

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-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

6	9	12	11	3	7

5 blue-blue inversions

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

5 blue-blue inversions

2	11	16	17	23	25
6	3	2	2	0	0

8 green-green inversions

How many blue-green inversions?

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 6

3	7	10	14	18	19

2	11	16	17	23	25

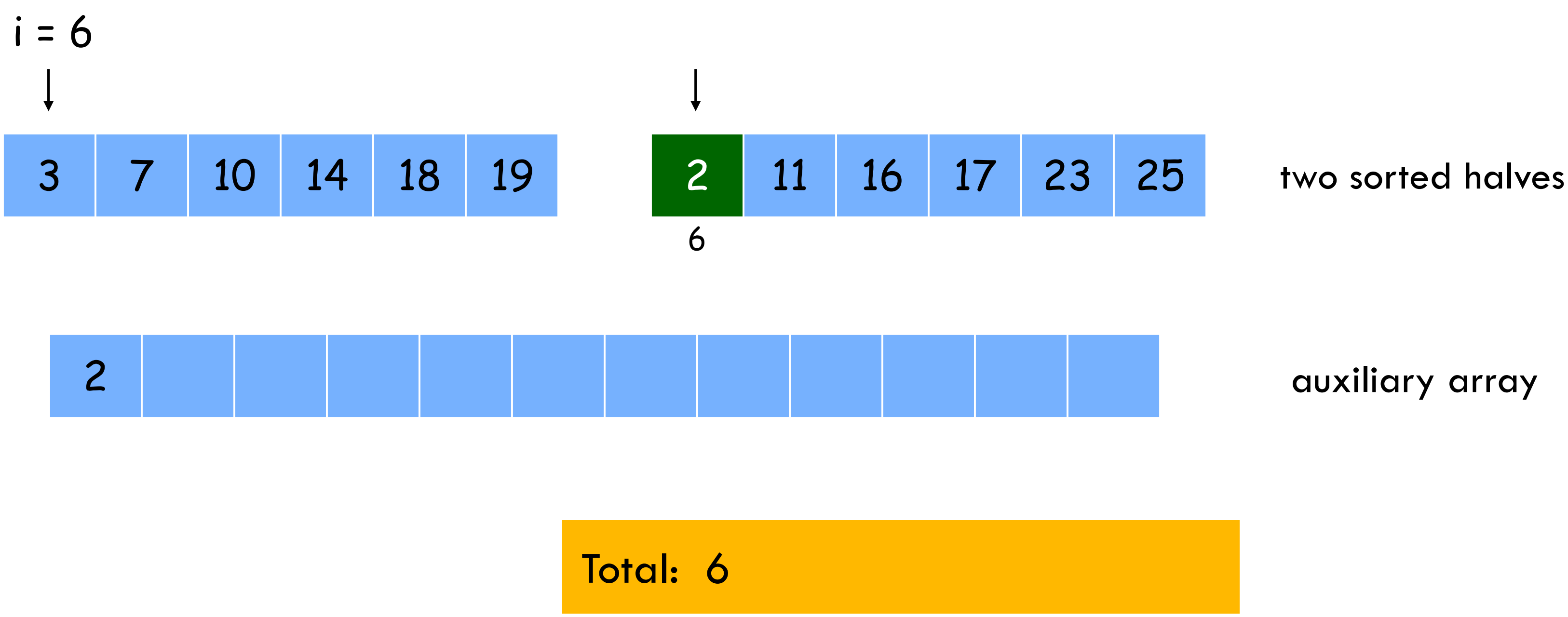
two sorted halves

auxiliary array

Total:

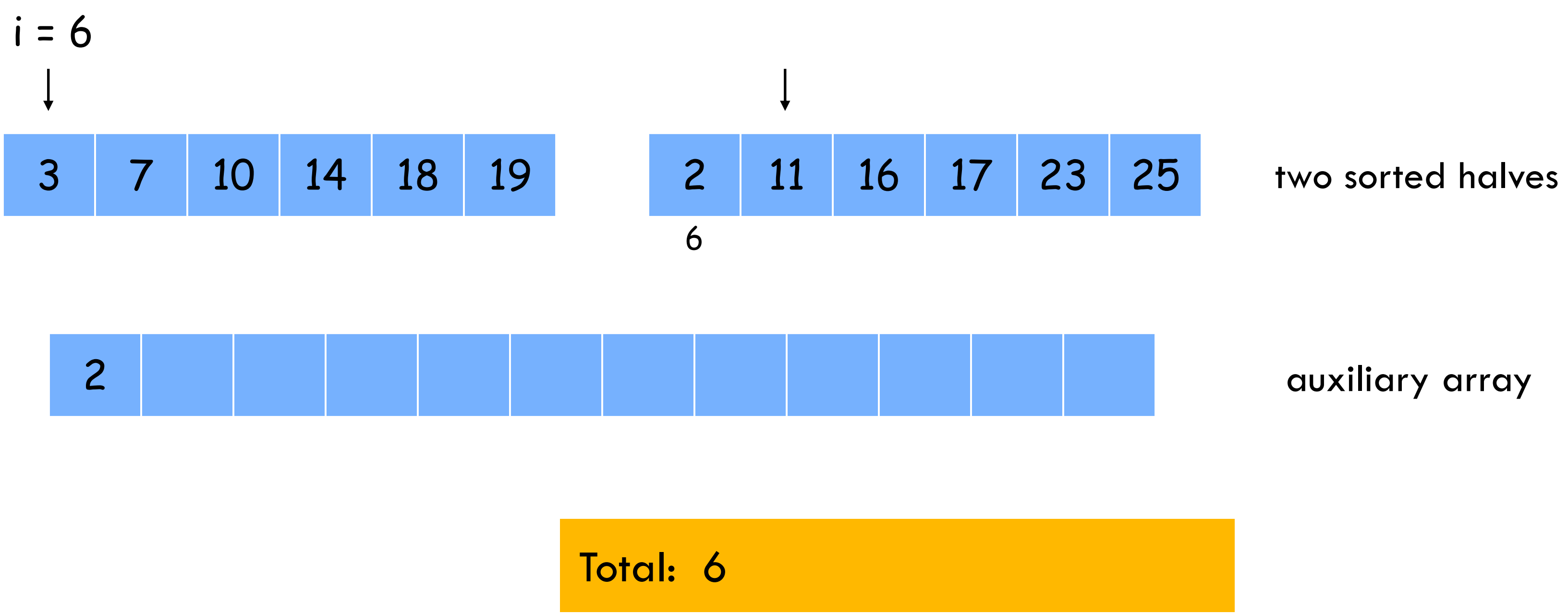
Merge and Count

– Merge and count step.
  – Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  – Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 6

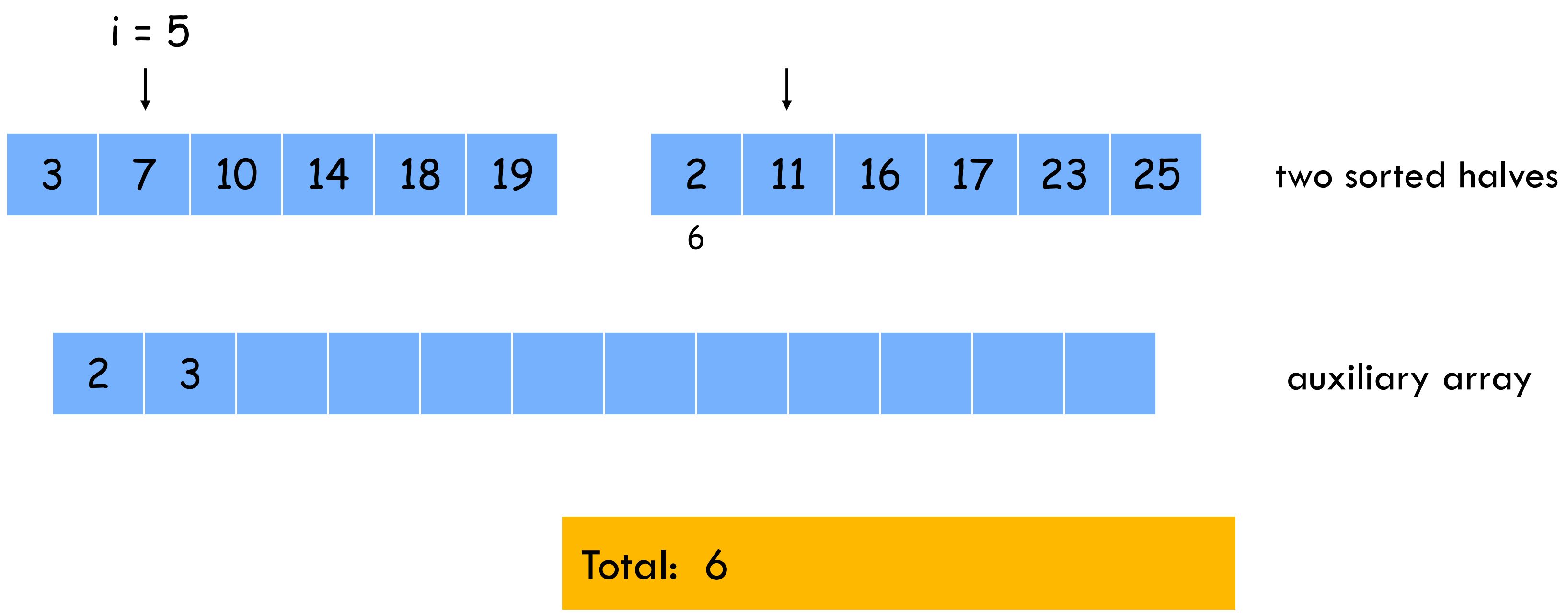
3	7	10	14	18	19	2	11	16	17	23	25	two sorted halves
						6						

2	3											auxiliary array

Total: 6

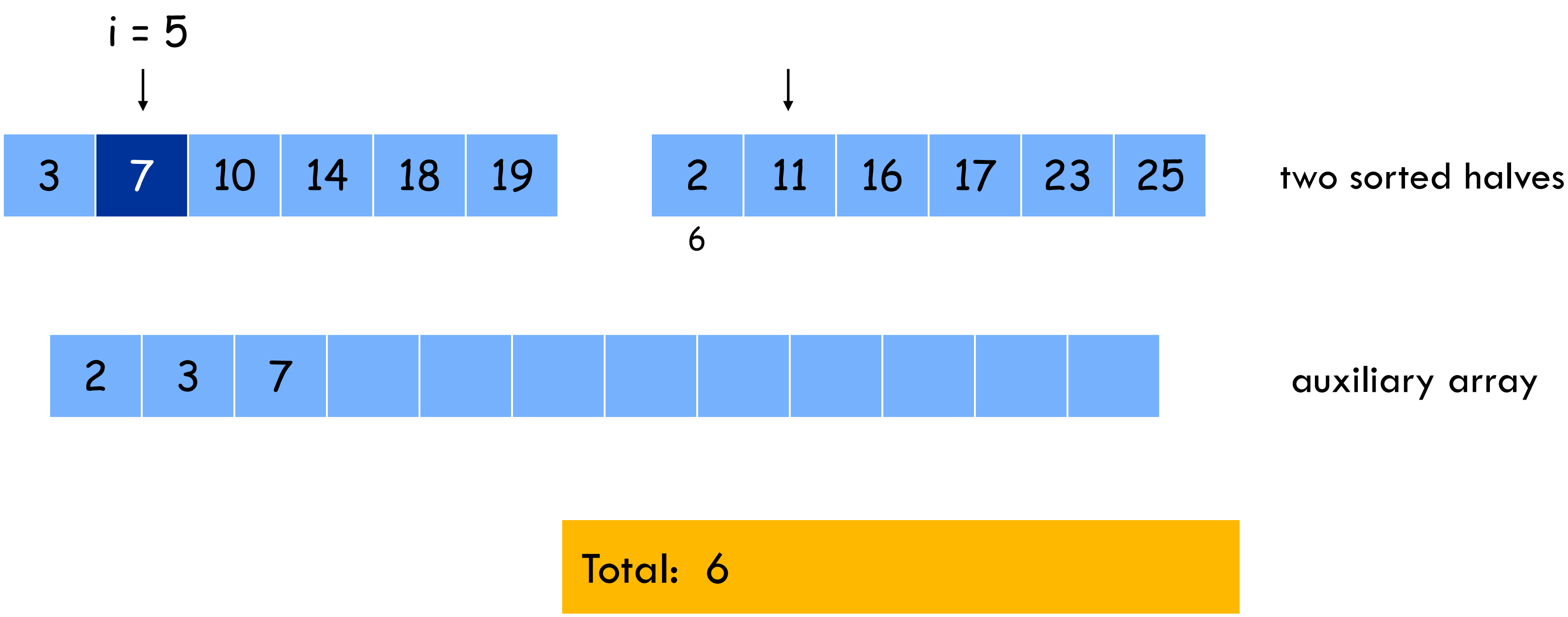
Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



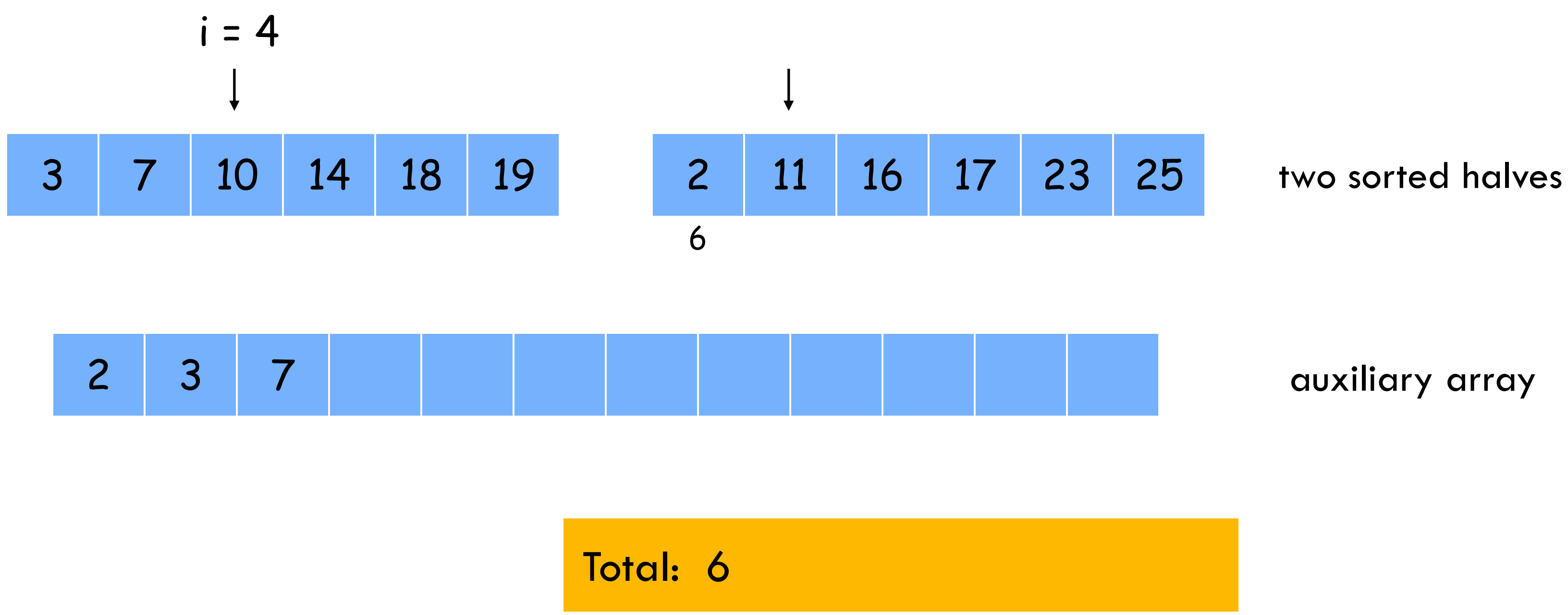
Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 4

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

two sorted halves

2	3	7	10								

auxiliary array

Total: 6

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 3

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

two sorted halves

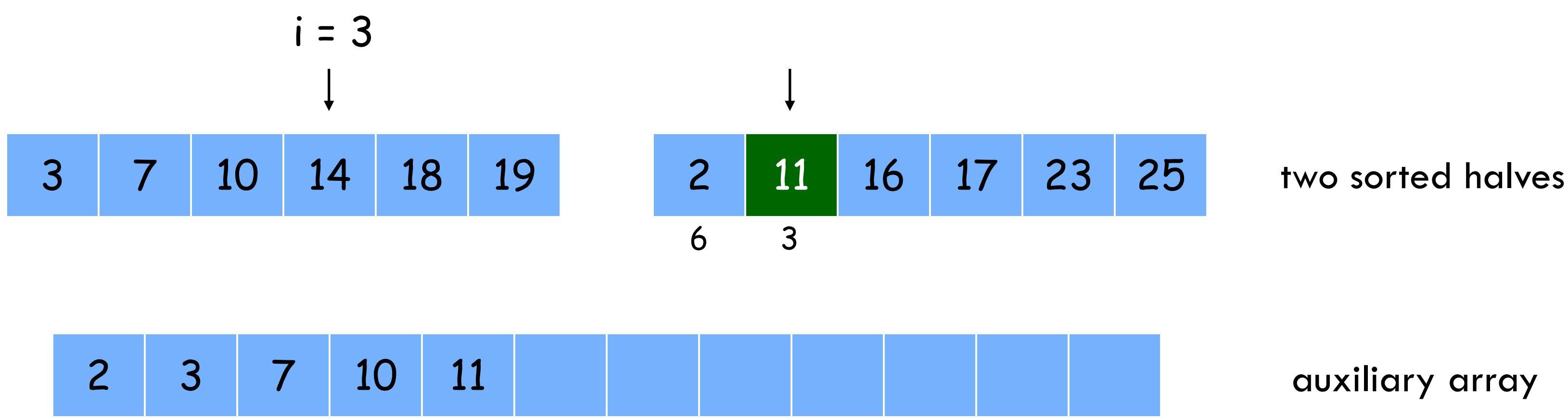
2	3	7	10								

auxiliary array

Total: 6

Merge and Count

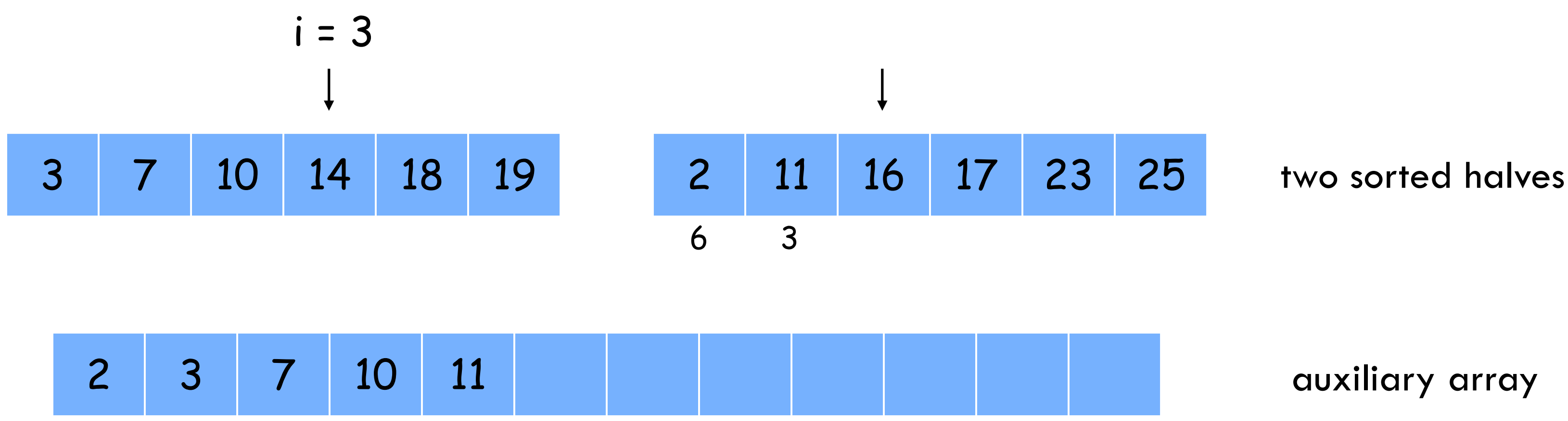
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3

Merge and Count

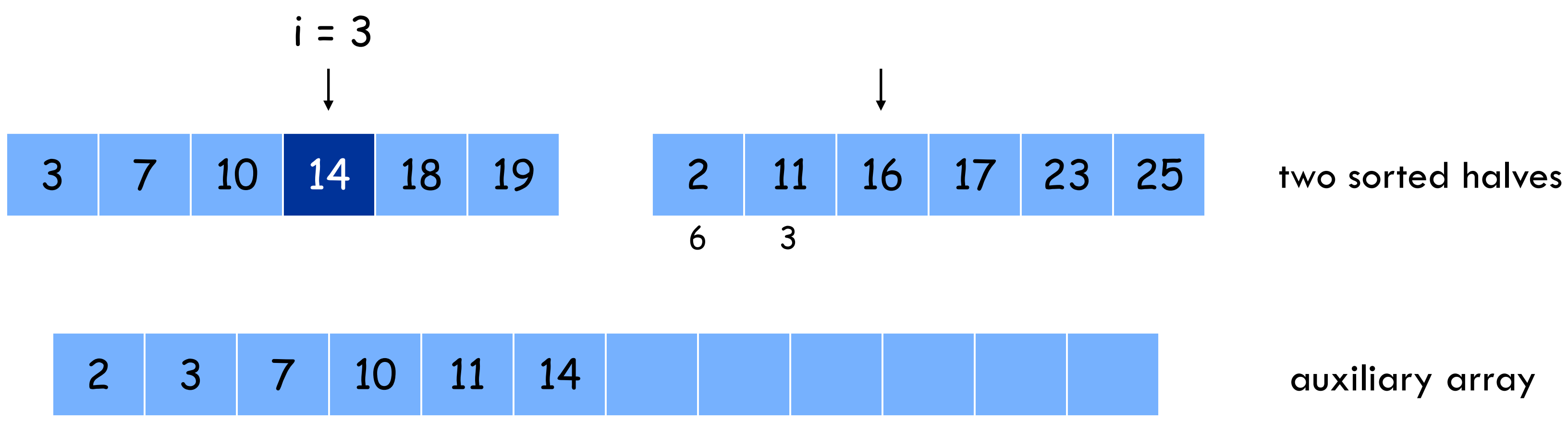
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3

Merge and Count

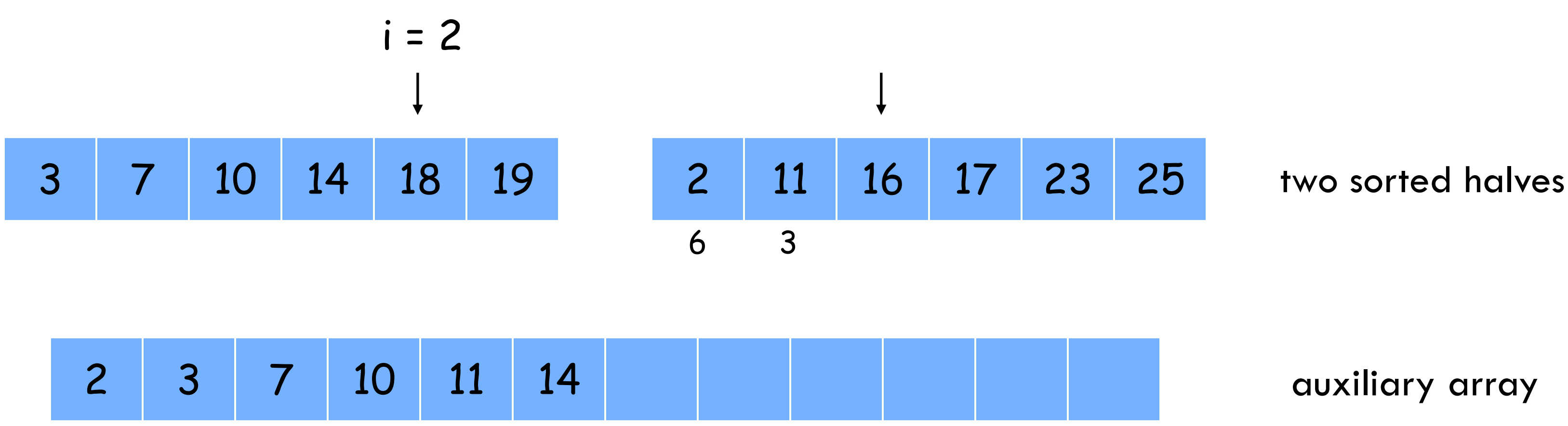
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3

Merge and Count

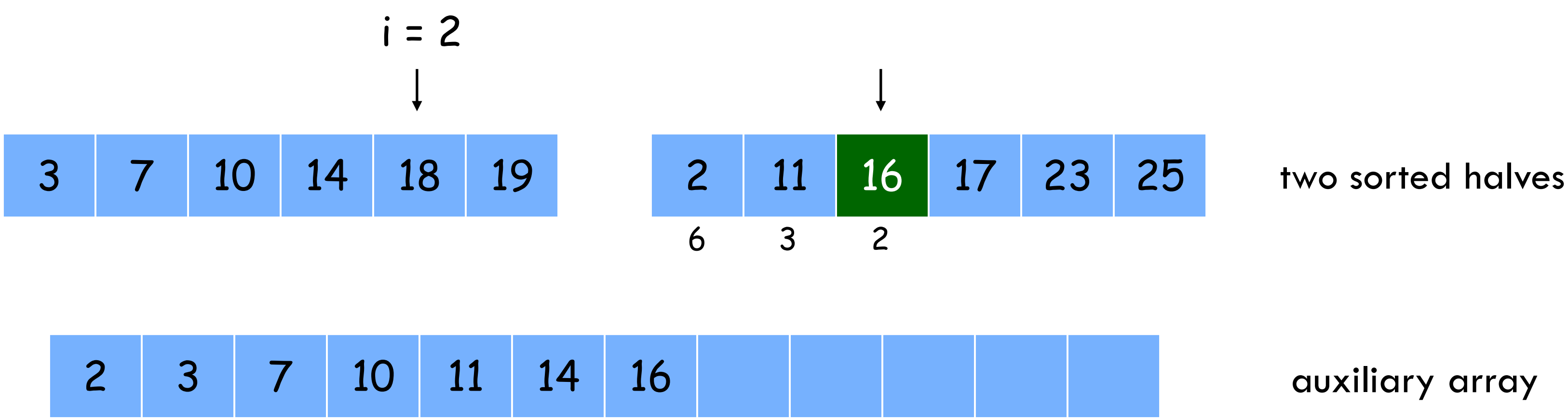
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3

Merge and Count

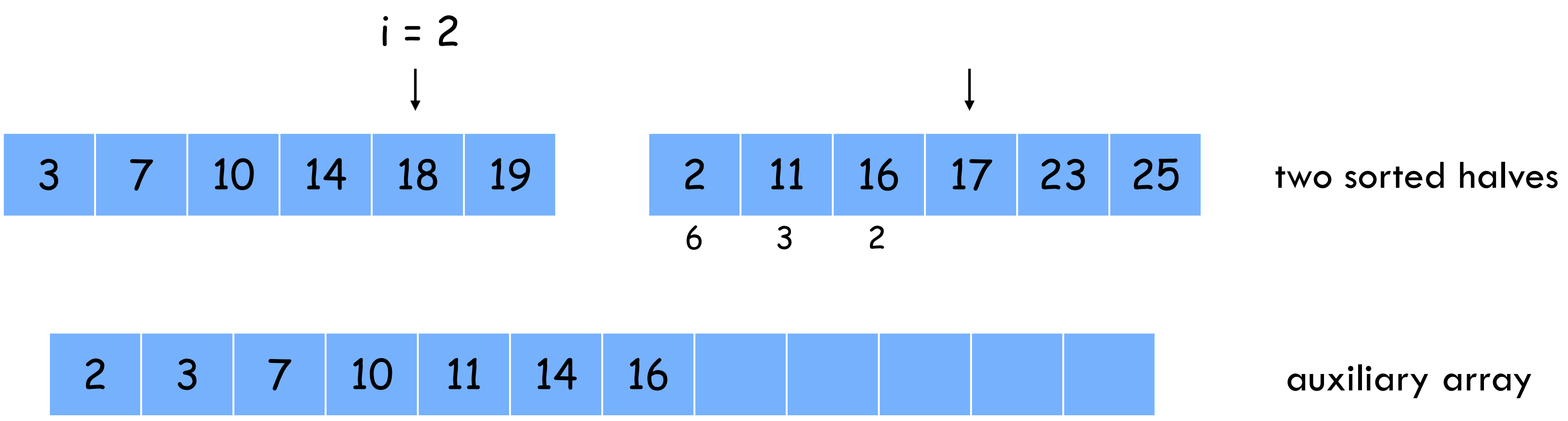
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3 + 2

Merge and Count

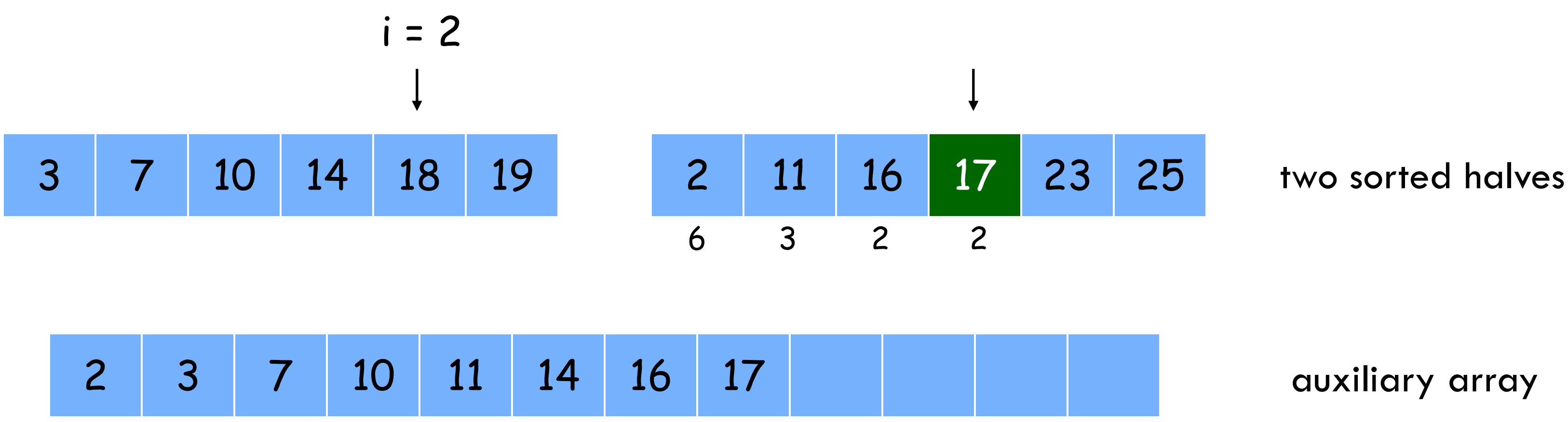
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3 + 2

Merge and Count

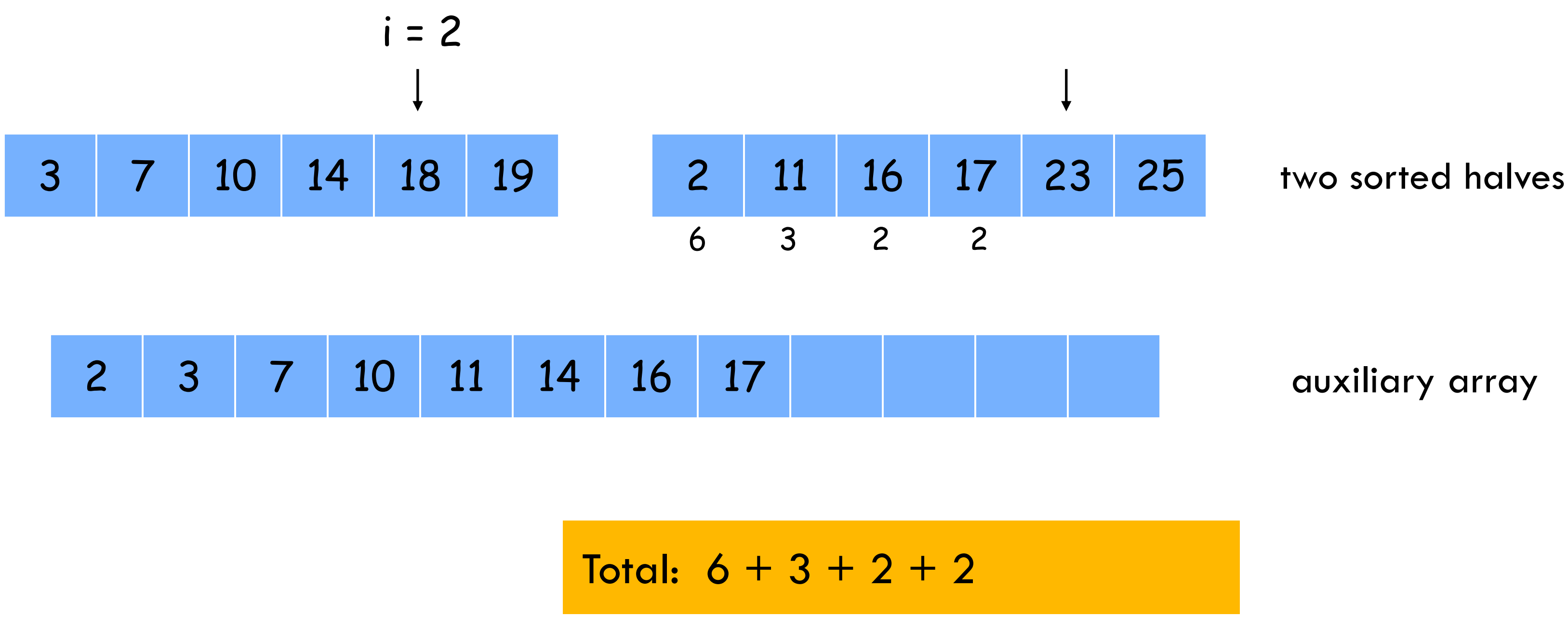
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

– Merge and count step.
  – Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  – Combine two sorted halves into sorted whole.

i = 2

two sorted halves

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

auxiliary array

2	3	7	10	11	14	16	17	18			

Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 1

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

two sorted halves

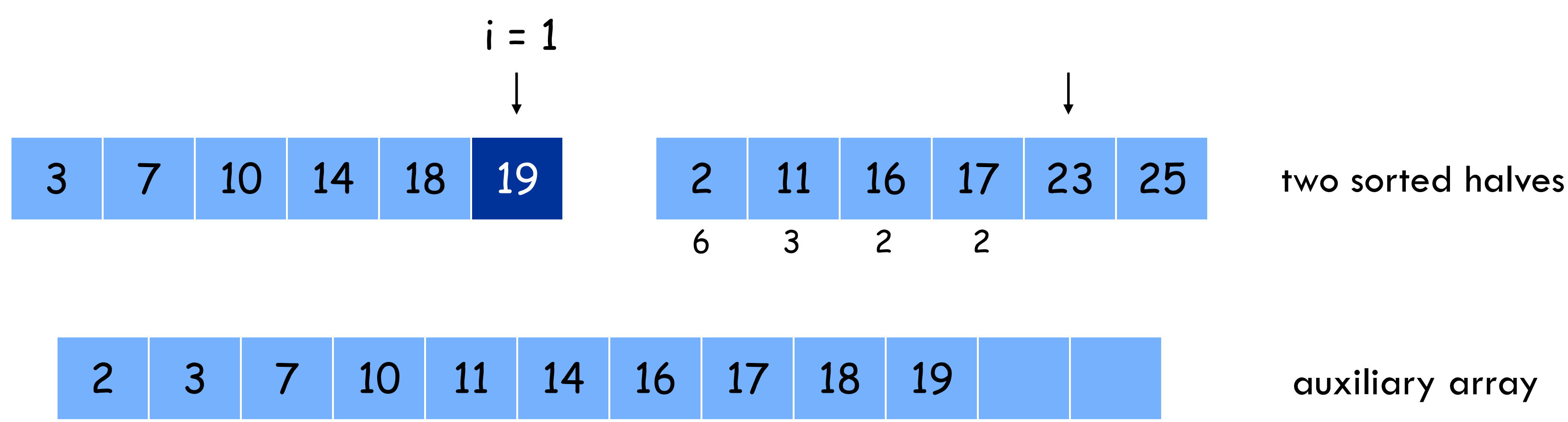
2	3	7	10	11	14	16	17	18			

auxiliary array

Total: 6 + 3 + 2 + 2

Merge and Count

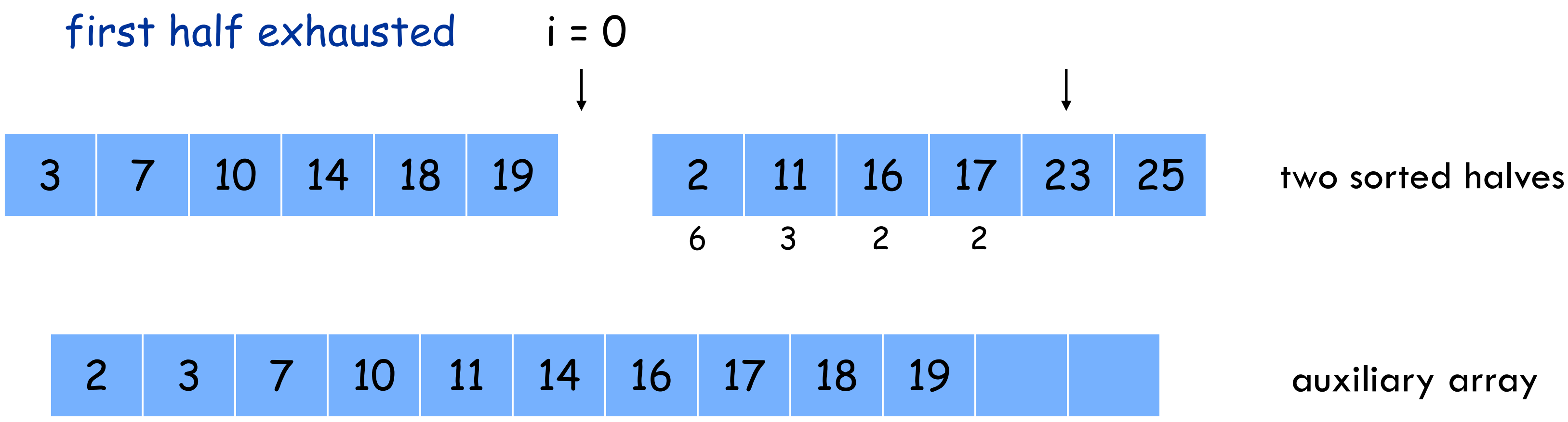
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3 + 2 + 2

Merge and Count

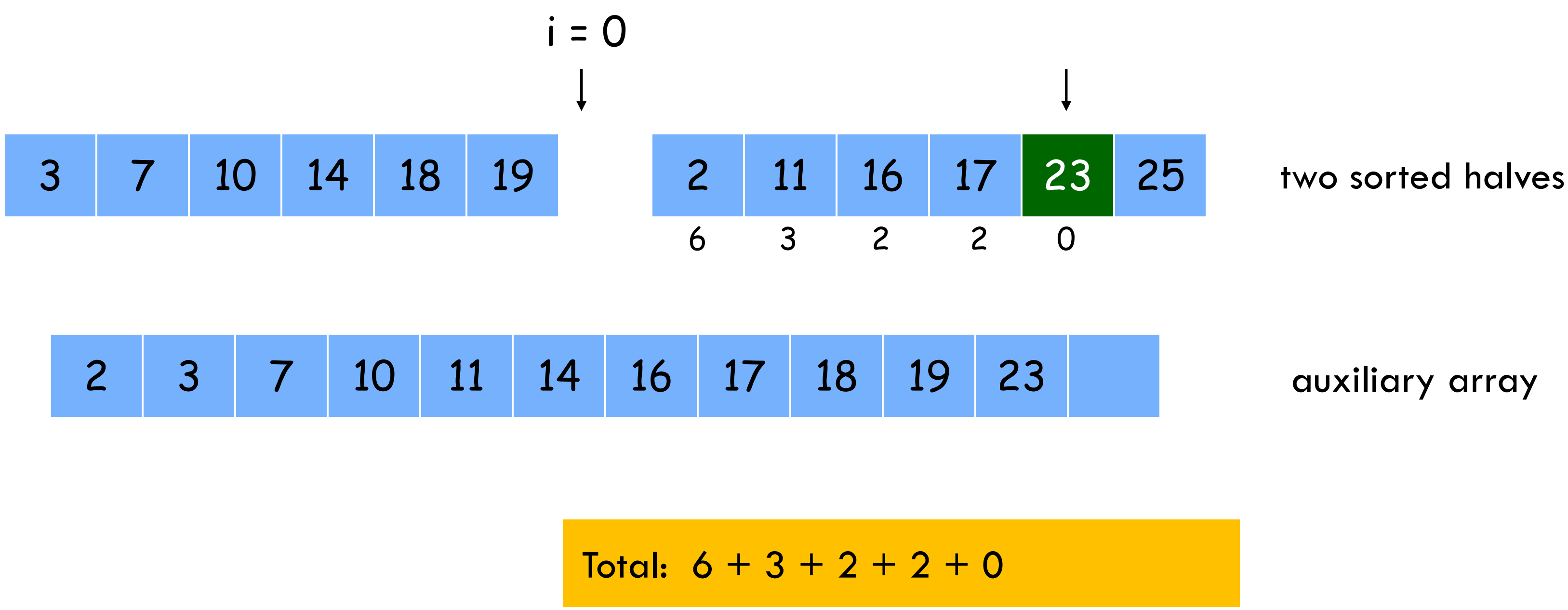
- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 0

two sorted halves

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	

auxiliary array

2	3	7	10	11	14	16	17	18	19	23	

Total: 6 + 3 + 2 + 2 + 0

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 0

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

two sorted halves

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

auxiliary array

Total: 6 + 3 + 2 + 2 + 0 + 0

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
  - Combine two sorted halves into sorted whole.

i = 0

two sorted halves

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

auxiliary array

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

Total: 6 + 3 + 2 + 2 + 0 + 0 = 13

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)

Time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Counting Inversions: 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
   (r_A, A) ← Sort-and-Count(A)
   (r_B, B) ← Sort-and-Count(B)
   (r_B, L) ← Merge-and-Count(A, B)

   return r = r_A + r_B + r and the sorted list L
}
```

Closest Pair of Points

Closest Pair of Points

- **Closest pair.** Given n points in the plane, find a pair with smallest Euclidean distance between them.

- **Fundamental geometric primitive.**
  - Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
  - Special case of nearest neighbor, Euclidean MST, Voronoi diagram...

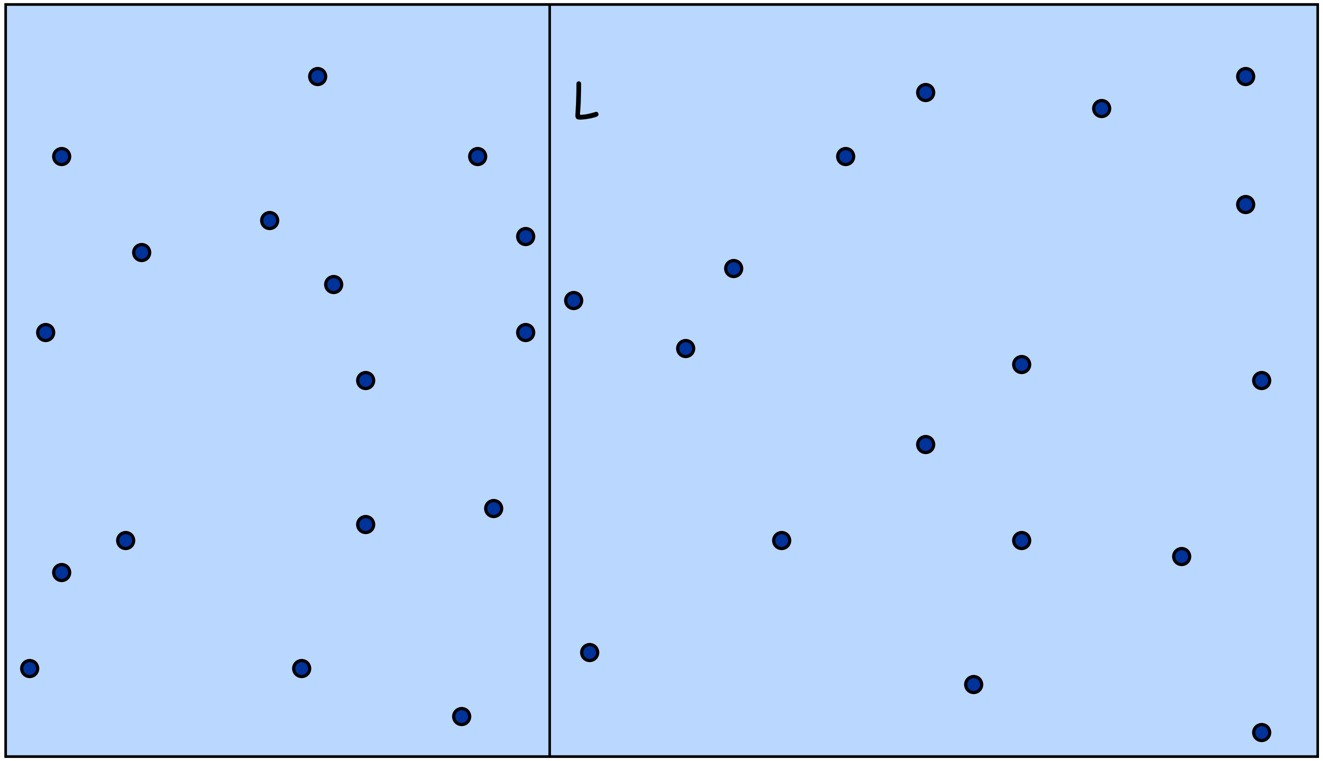
- **Brute force.** Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

- **1-D version.** O(n log n) easy if points are on a line.

- **Assumption.** No two points have same x coordinate.

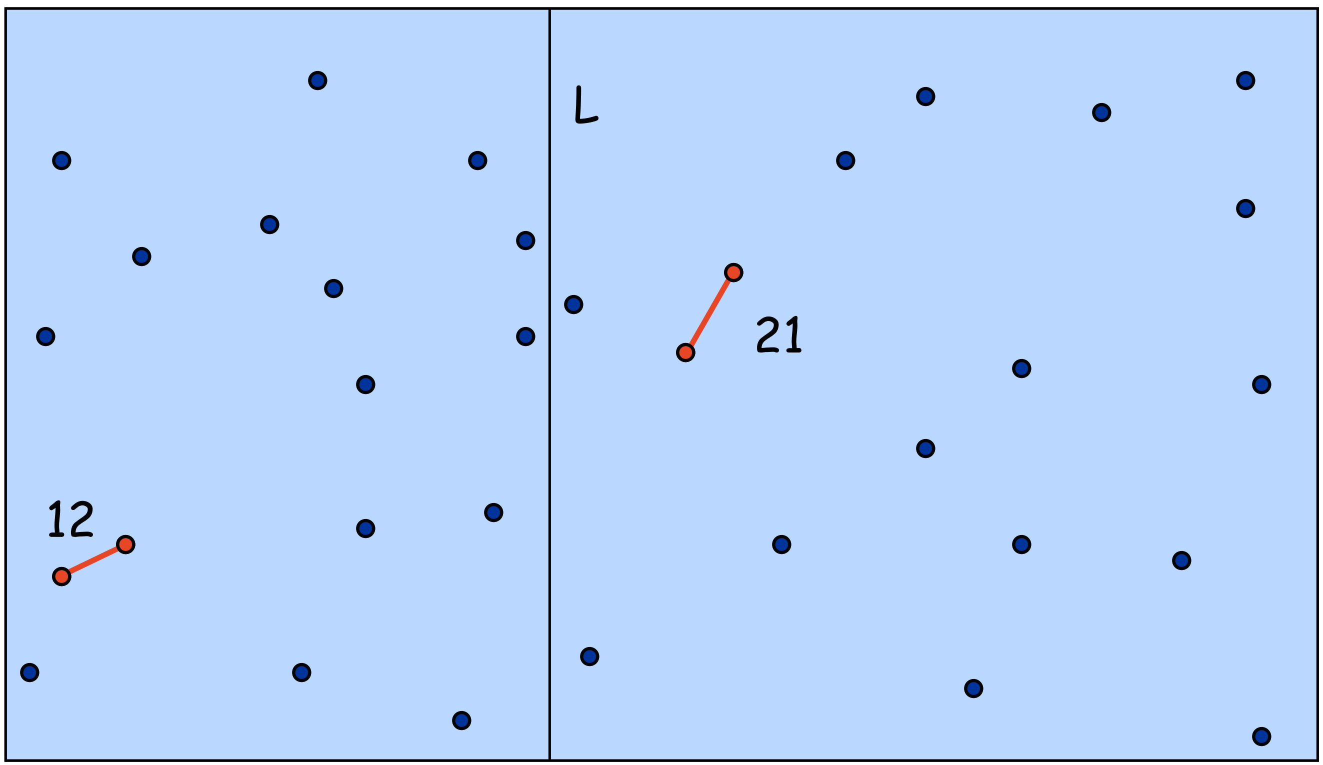
Closest Pair of Points

- **Algorithm.**
  - Divide: draw vertical line L so that roughly n/2 points on each side.



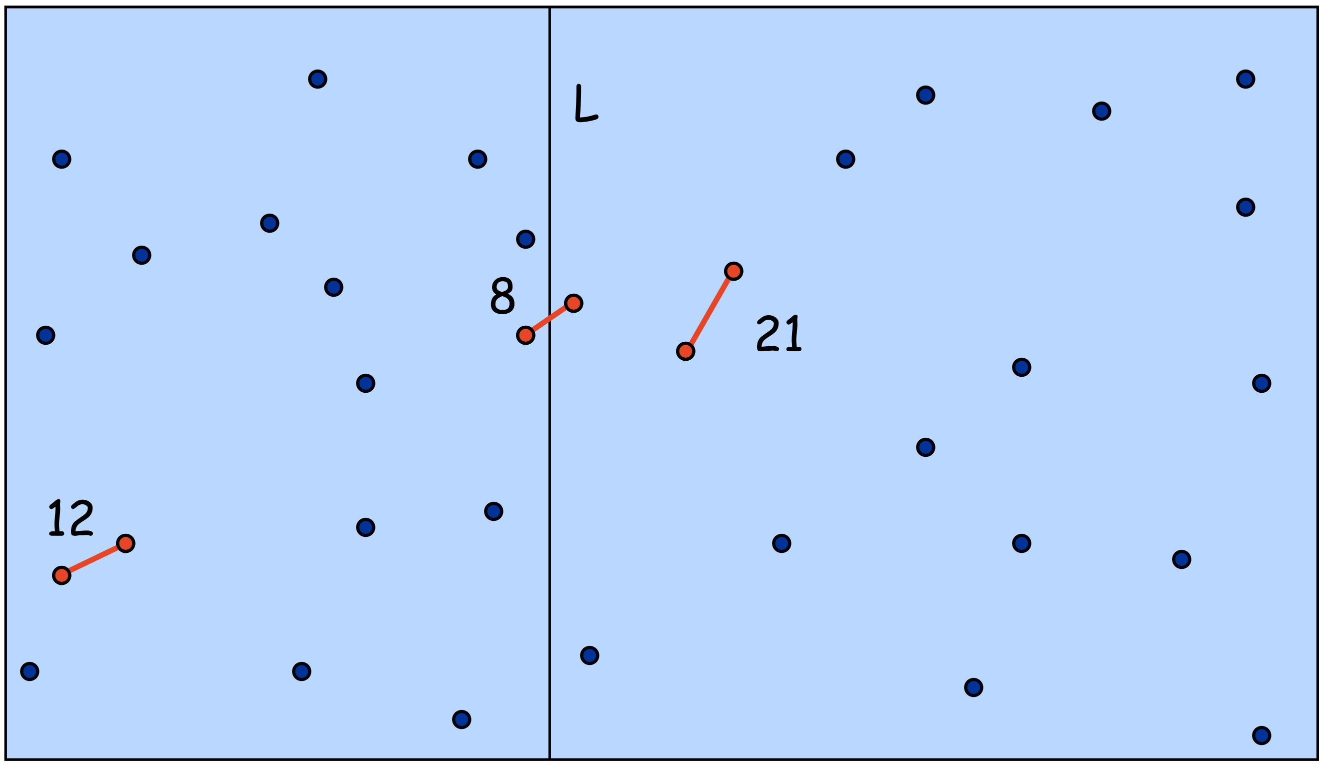
Closest Pair of Points

- **Algorithm.**
  - Divide: draw vertical line L so that roughly n/2 points on each side.
  - Conquer: find closest pair in each side recursively.



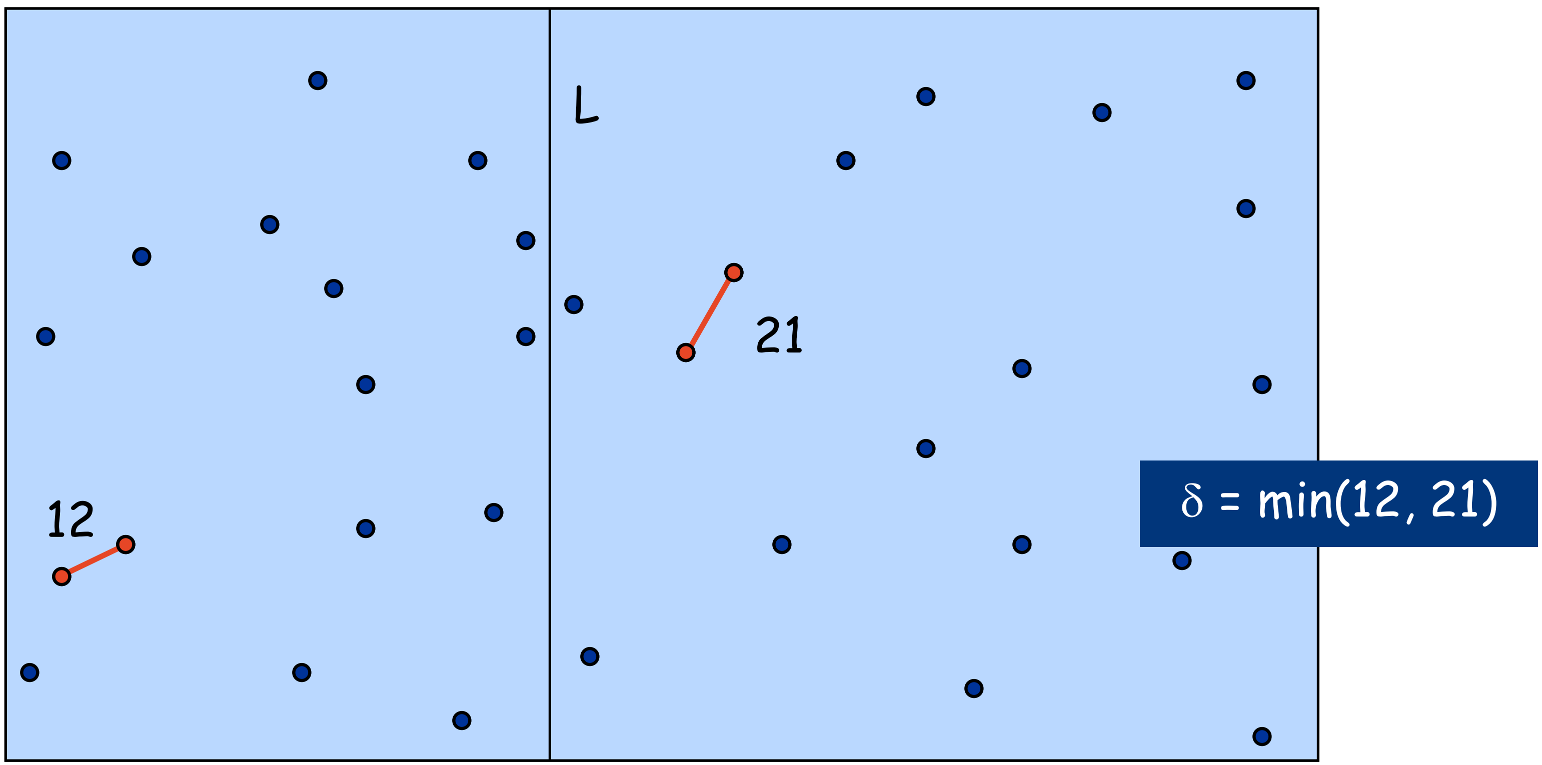
Closest Pair of Points

- **Algorithm.**
  - Divide: draw vertical line L so that roughly n/2 points on each side.
  - Conquer: find closest pair in each side recursively.
  - Combine: find closest pair with one point in each side.
  - Return best of 3 solutions.



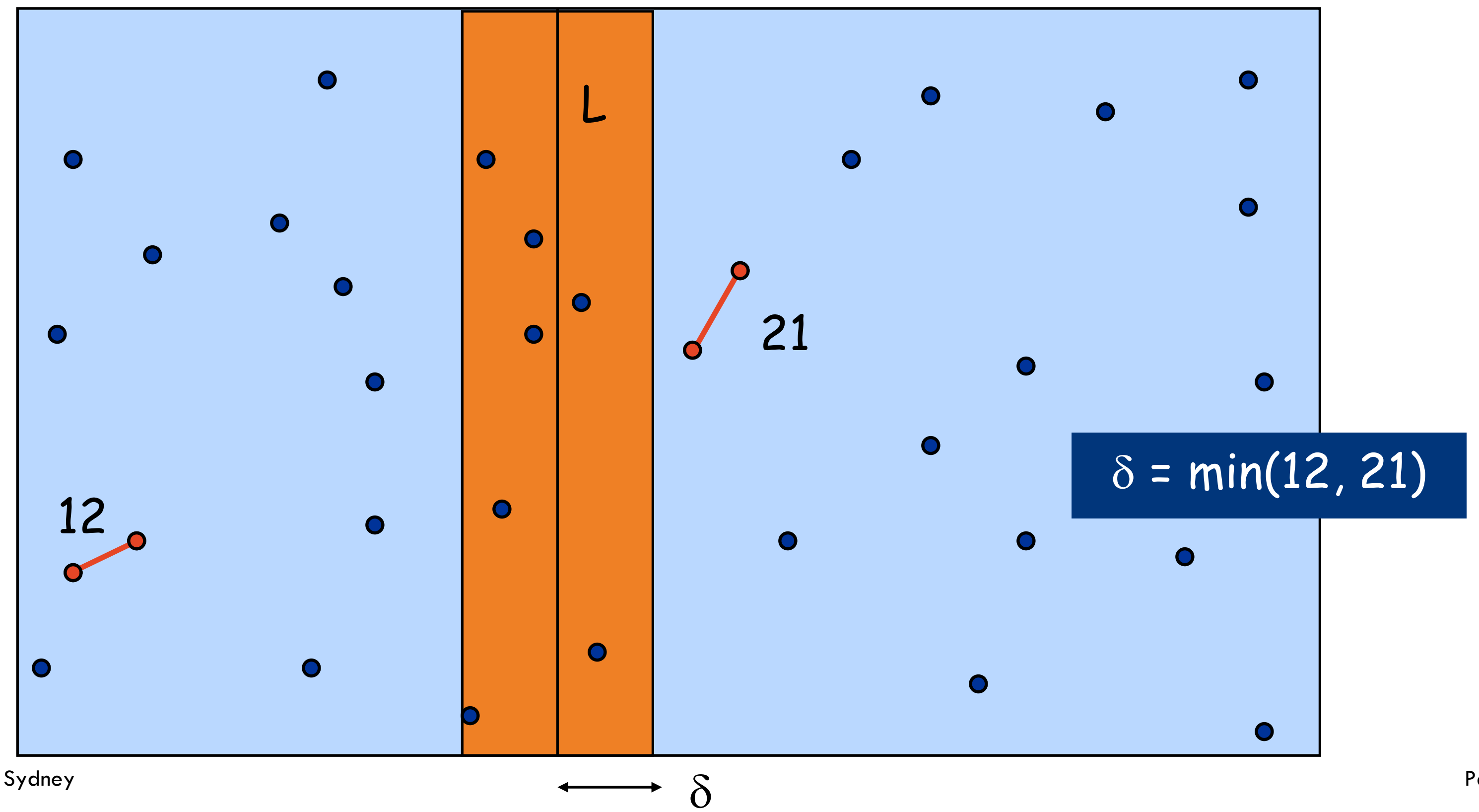
Closest Pair of Points

– Find closest pair with one point in each side, assuming that distance $< \delta$.



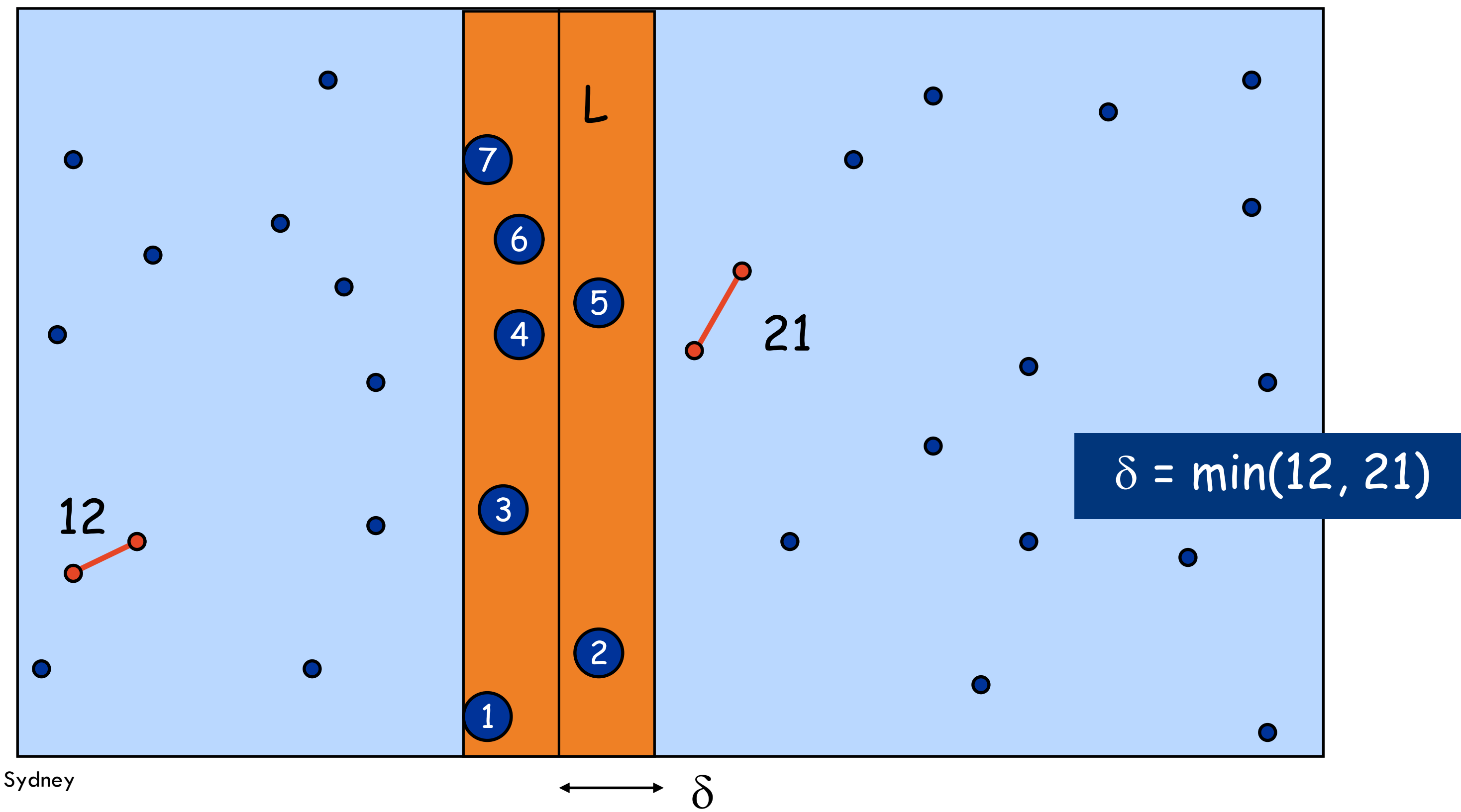
Closest Pair of Points

– Find closest pair with one point in each side, assuming that distance < δ.
  – **Observation:** only need to consider points within δ of line L.



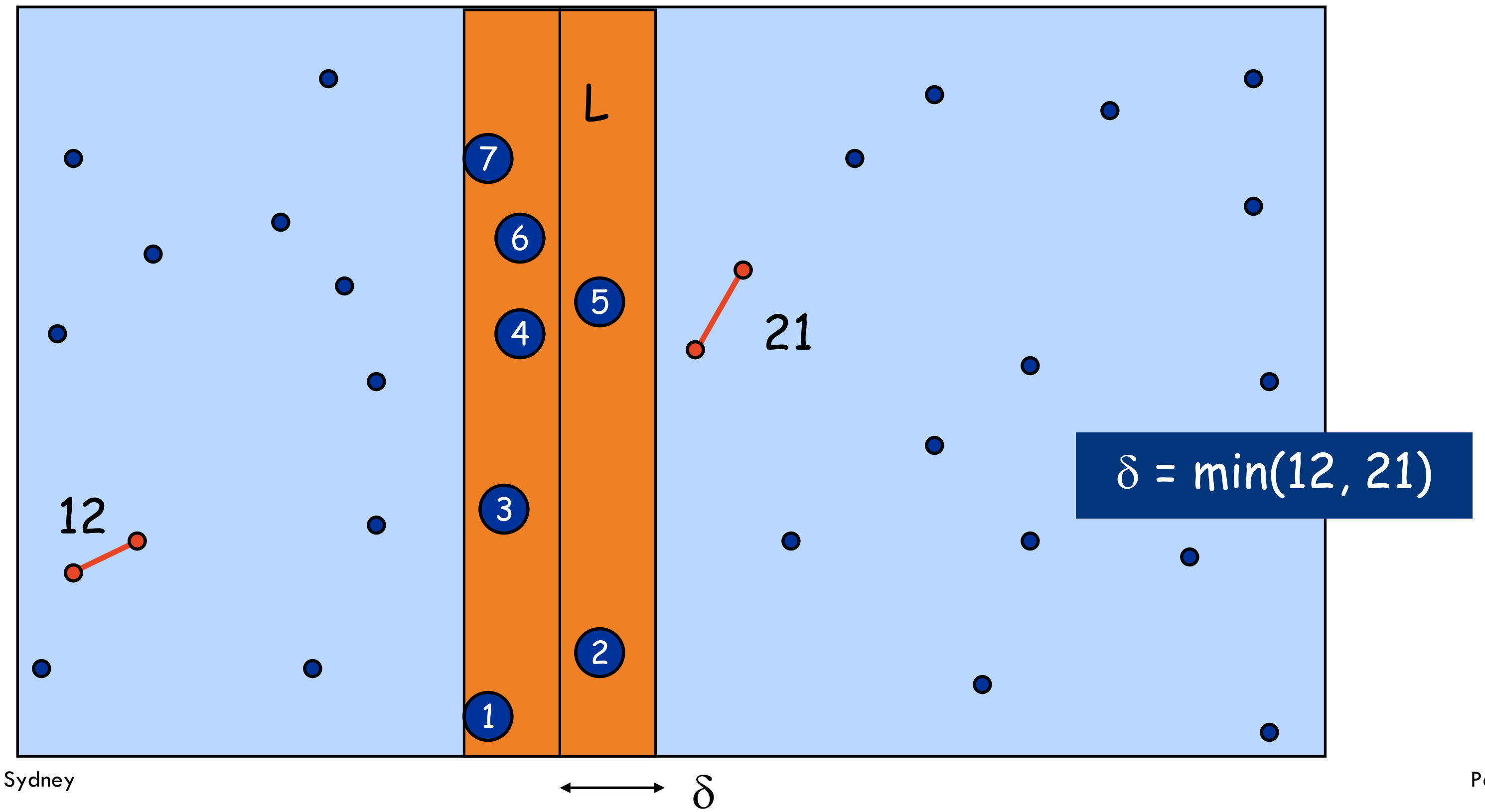
Closest Pair of Points

– Find closest pair with one point in each side, assuming that distance < δ.
  – **Observation:** only need to consider points within δ of line L.
  – Sort points in 2δ-strip by their y-coordinate.



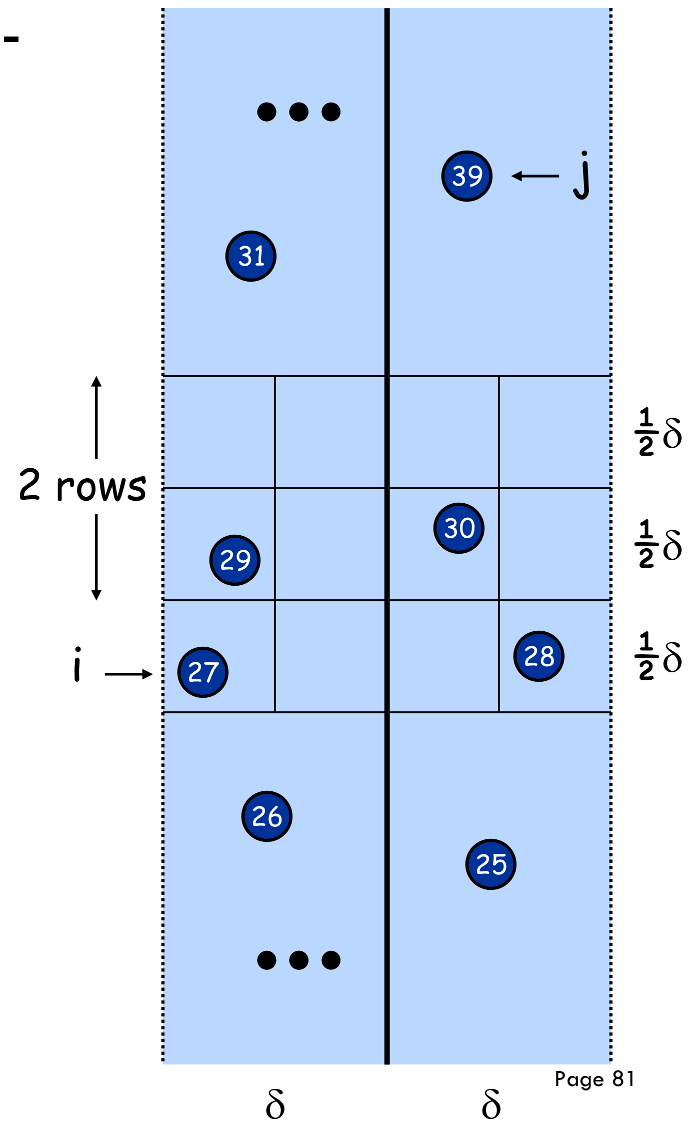
Closest Pair of Points

- Find closest pair with one point in each side, assuming that distance < δ.
  - **Observation:** only need to consider points within δ of line L.
  - Sort points in 2δ-strip by their y coordinate.
  - Only check distances of those within 11 positions in sorted list!



Closest Pair of Points

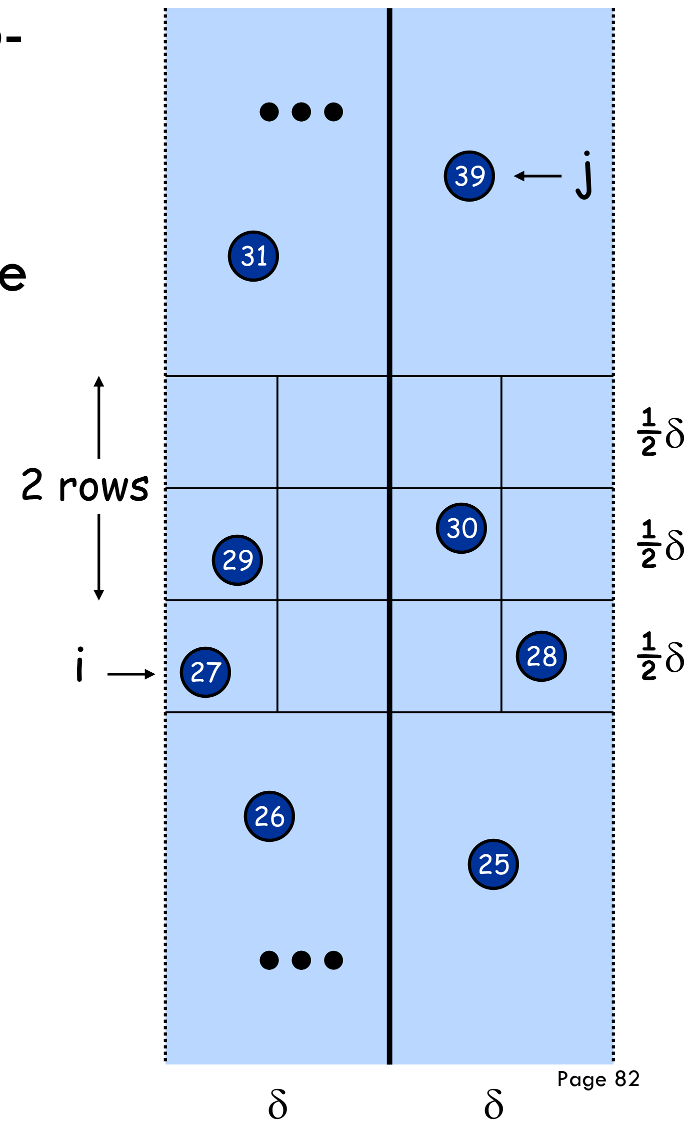
– **Definition:** Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.



Closest Pair of Points

– **Definition:** Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

– **Claim:** If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.

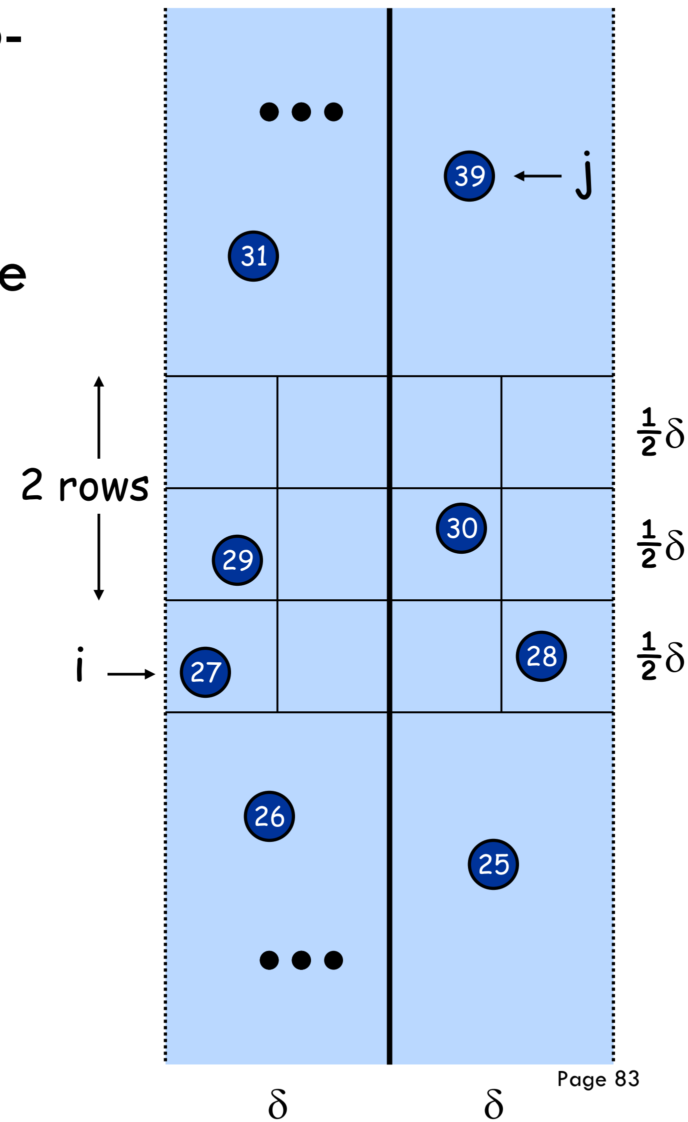


Closest Pair of Points

– **Definition:** Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

– **Claim:** If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.

– **Proof:**
  – No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
  – Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta) = \delta$. ▪



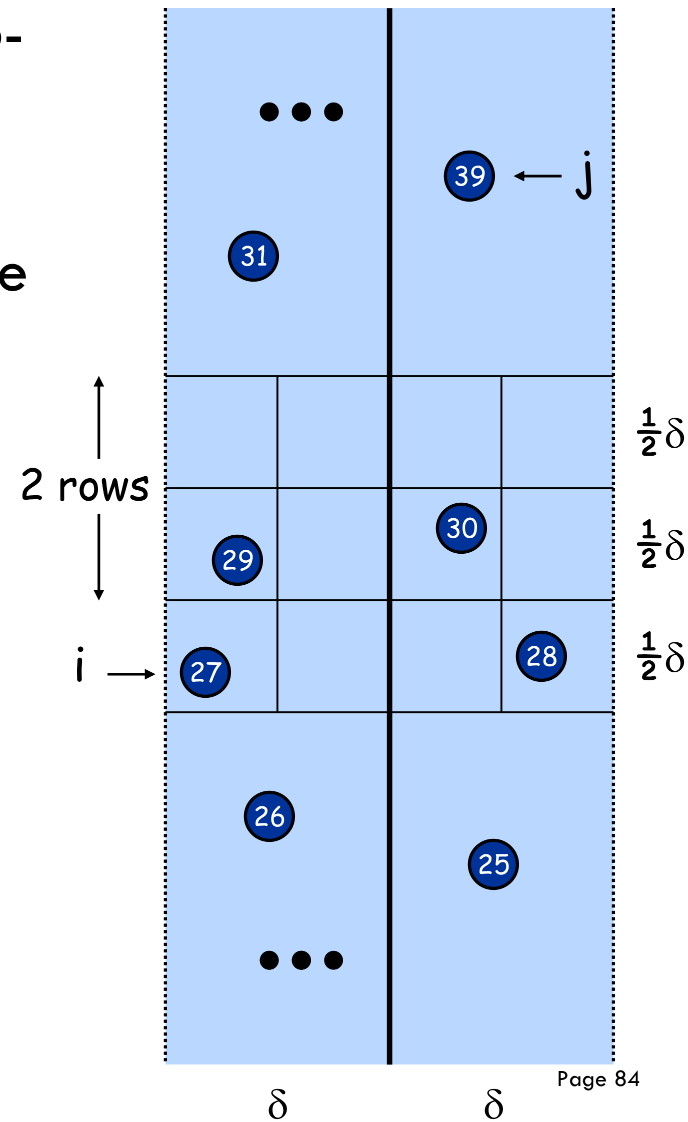
Closest Pair of Points

- **Definition:** Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

- **Claim:** If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.

- **Proof:**
  - No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
  - Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta) = \delta$. ▪

Fact: Still true if we replace 12 with 7.



Closest Pair Algorithm

```
Closest-Pair(p₁, …, pₙ) {

 If |P|≤ 3 then compute closest-pair brute force
 else
   Compute separation line L such that half the points
   are on one side and half on the other side.

   δ₁ = Closest-Pair(left half)
   δ₂ = Closest-Pair(right half)
   δ  = min(δ₁, δ₂)

   Delete all points further than δ from separation line L

   Sort remaining points by y-coordinate.

   Scan points in y-order and compare distance between
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.

 return δ.
}
```

- Compute separation line: $O(n \log n)$
- Recursive calls: $2T(n / 2)$
- Delete points: $O(n)$
- Sort: $O(n \log n)$
- Scan: $O(n)$

Closest Pair of Points: Analysis

– **Running time**

$$T(n) \leq 2T(n/2) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)$$

– **Question:** Can we achieve O(n log n)?

– **Answer:** Yes. Don't sort points in strip from scratch each time.
  – Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
  – Sort by merging two pre-sorted lists.

$$T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$

Sort P by x-coordinates ⇒ P_x
Sort P by y-coordinates ⇒ P_y — $O(n \log n)$

Closest-Pair(P_x,P_y) {

If |P|≤ 3 then compute closest-pair brute force
else
Compute separation line L — $O(n)$
$P_{x,left}$ = points to the left of L sorted by x-coordinate
$P_{y,left}$ = points to the left of L sorted by y-coordinate
$P_{x,right}$ = points to the right of L sorted by x-coordinate — $O(n)$
$P_{y,right}$ = points to the right of L sorted by y-coordinate

δ_1 = Closest-Pair($P_{x,left}$,$P_{y,left}$)
δ_2 = Closest-Pair($P_{x,right}$,$P_{y,right}$) — $2T(n / 2)$
δ = min(δ_1, δ_2)

Delete all points further than δ from separation line L — $O(n)$

Scan points in y-order and compare distance between
each point and next 11 neighbors. If any of these — $O(n)$
distances is less than δ, update δ.

return δ.
}

Closest Pair of Points (improved): Analysis

– **Running time**

Preprocessing: $O(n \log n)$

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

Total running time: $O(n \log n)$

Solving recursions

Unrolling and the Master method

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

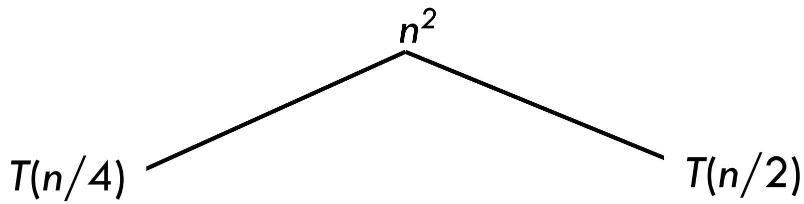
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$T(n)$$

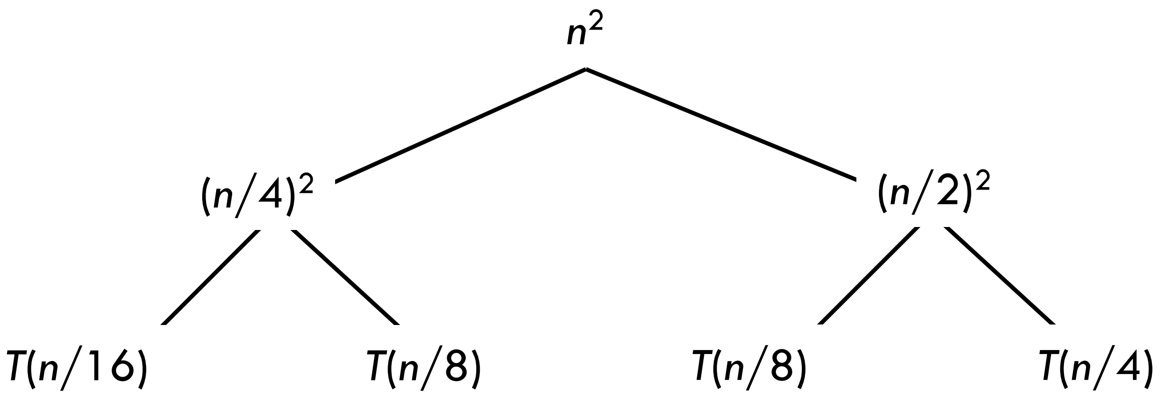
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



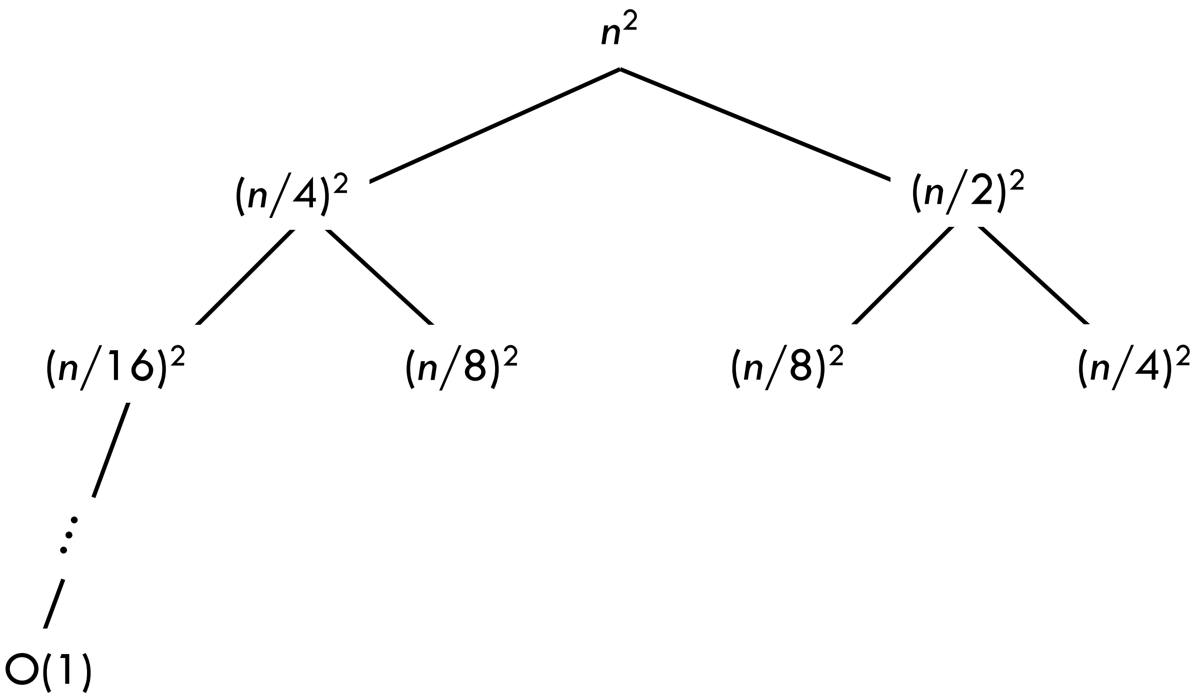
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



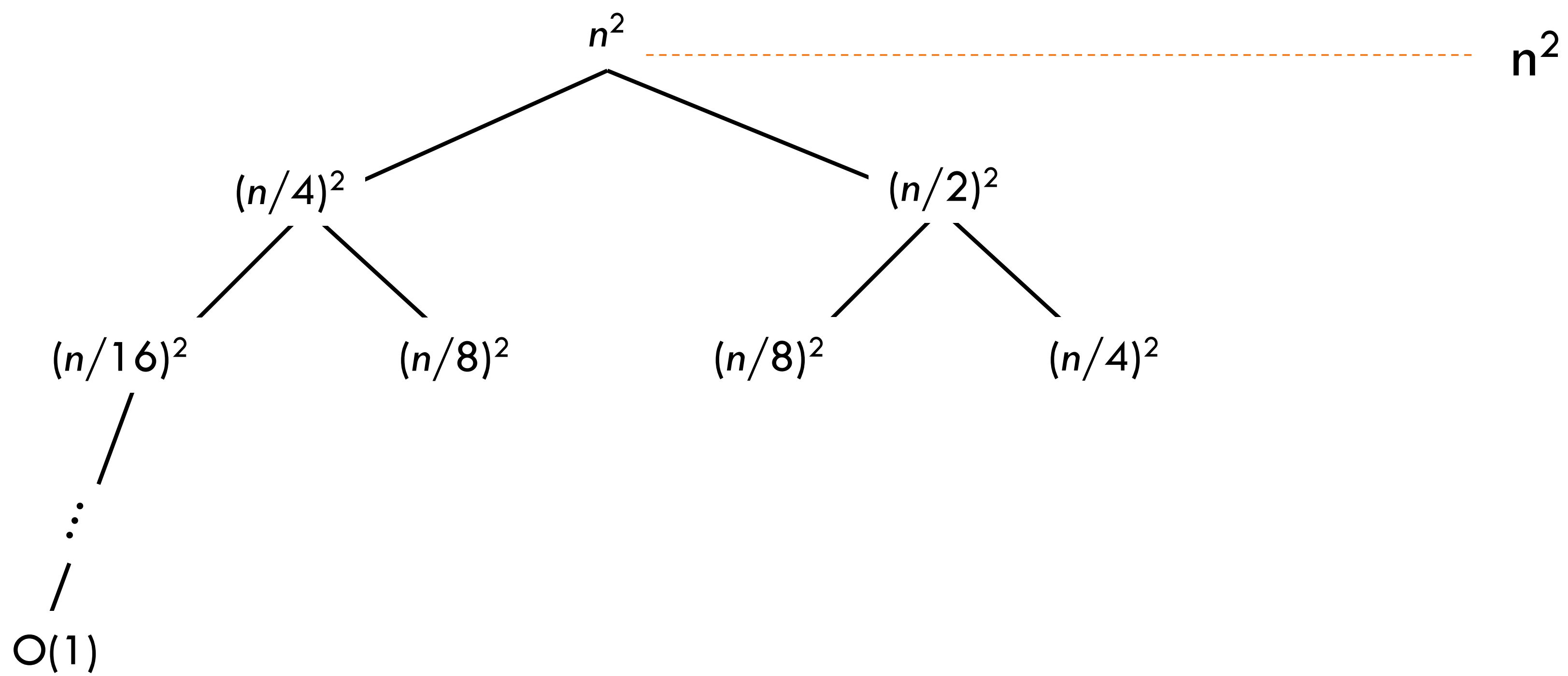
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



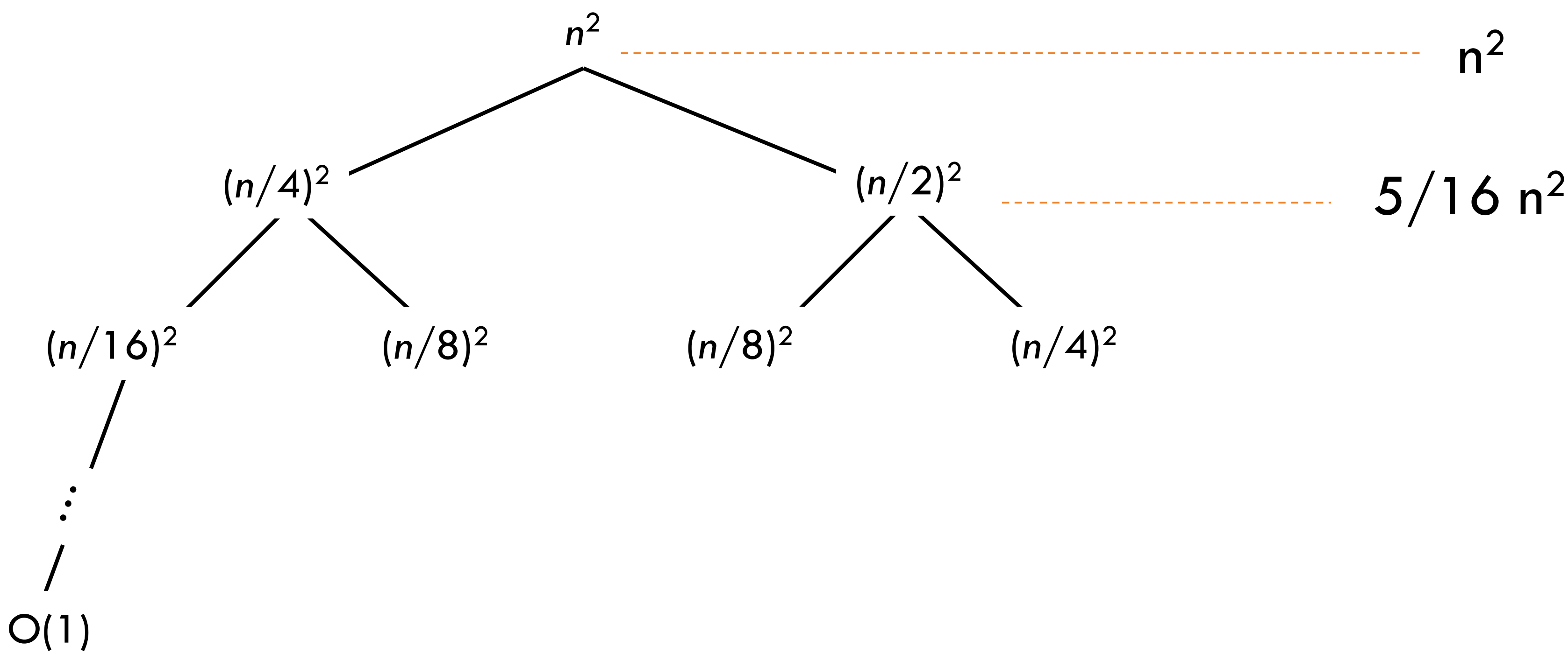
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



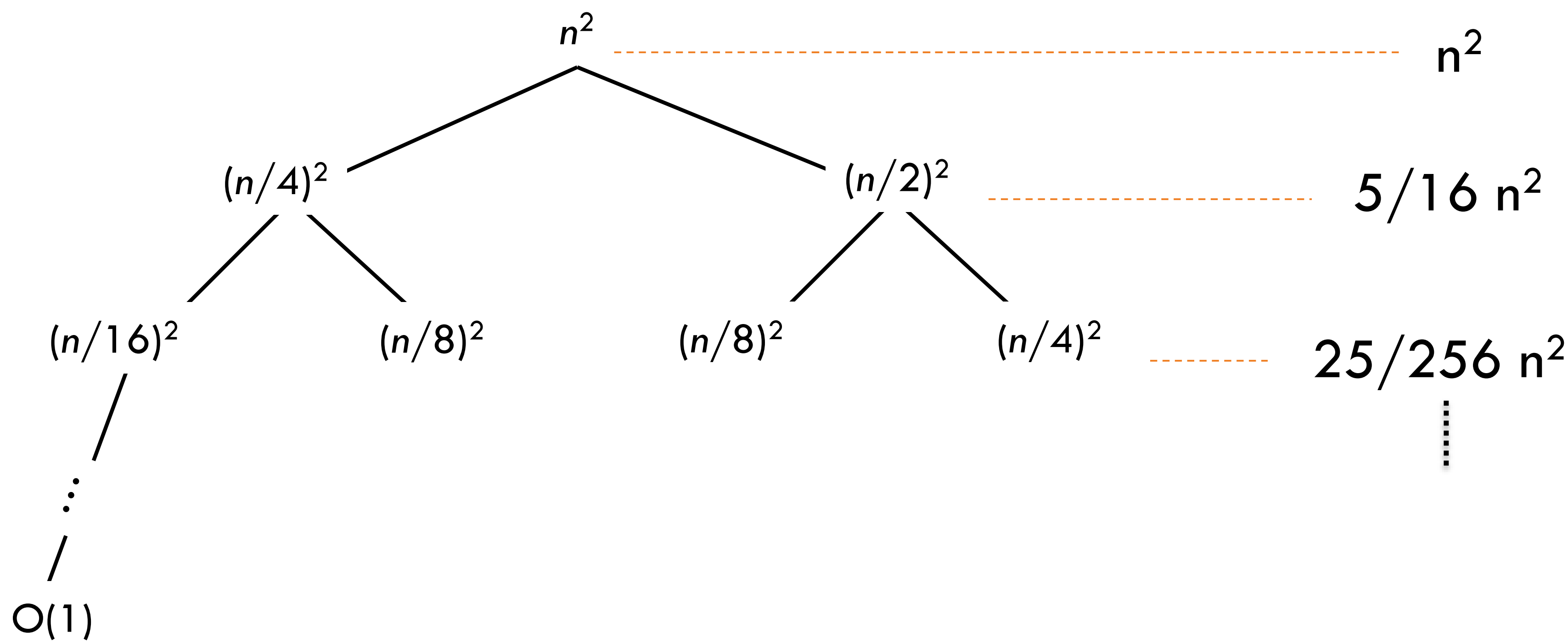
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



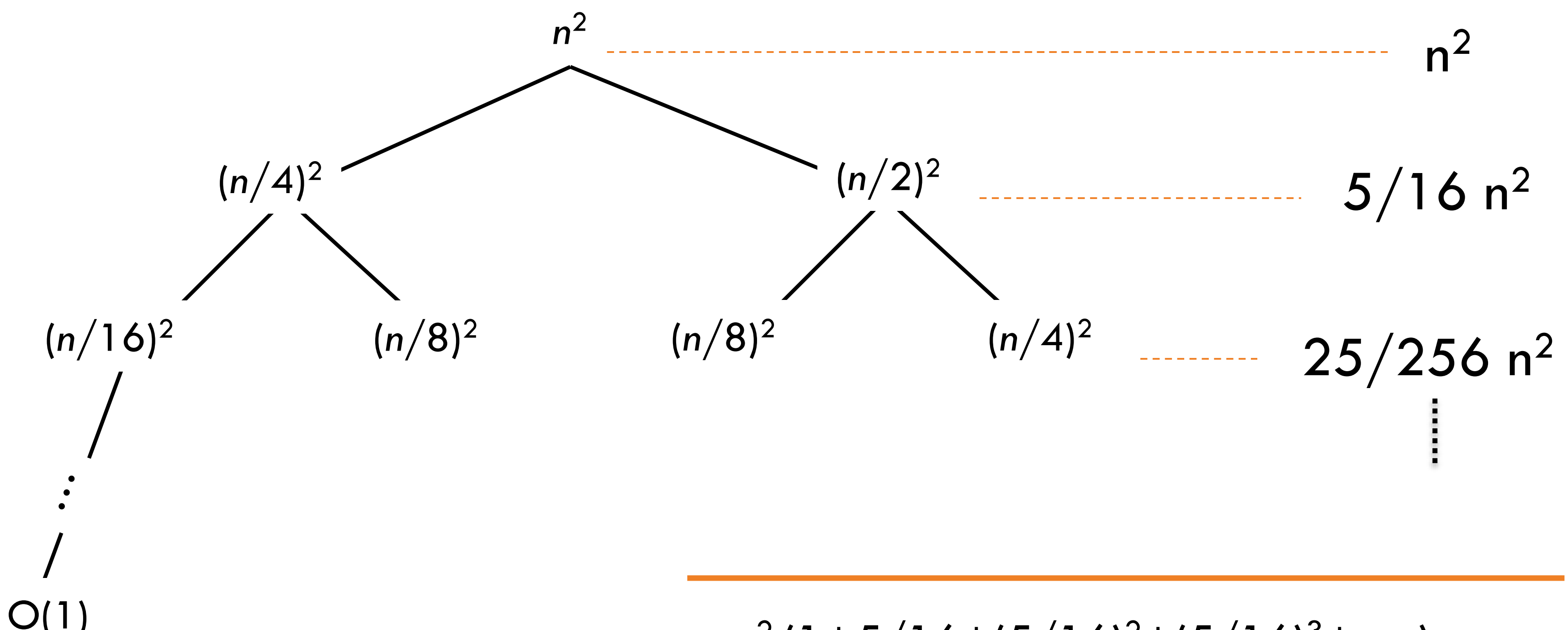
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + 5/16 + (5/16)^2 + (5/16)^3 + \dots\right)$$

geometric series

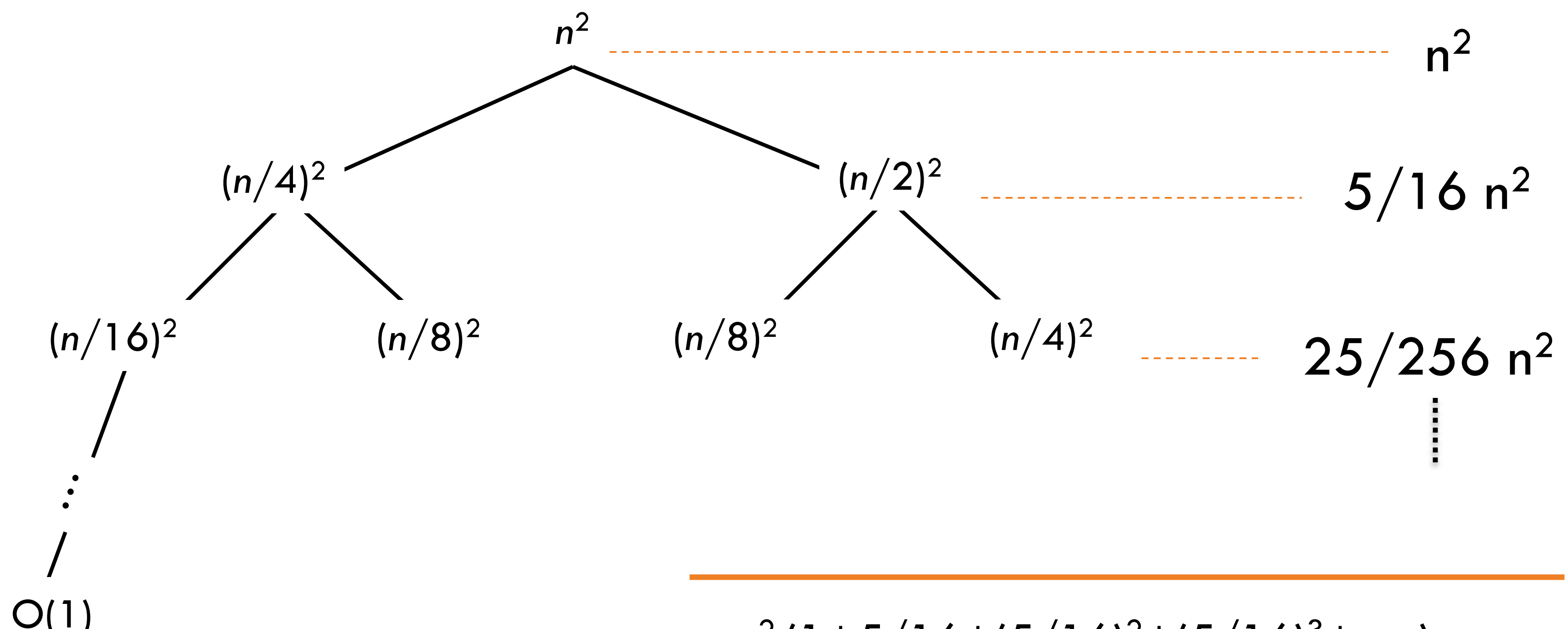
Appendix: geometric series

$$1 + x + x^2 + \ldots + x^n = \frac{1-x^{n+1}}{1-x} \quad \text{for } x \neq 1$$

$$1 + x + x^2 + \ldots = \frac{1}{1-x} \quad \text{for } x < 1$$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\,(1+5/16+(5/16)^2+(5/16)^3+\ldots)$$

$$= O(n^2)$$ *geometric series*

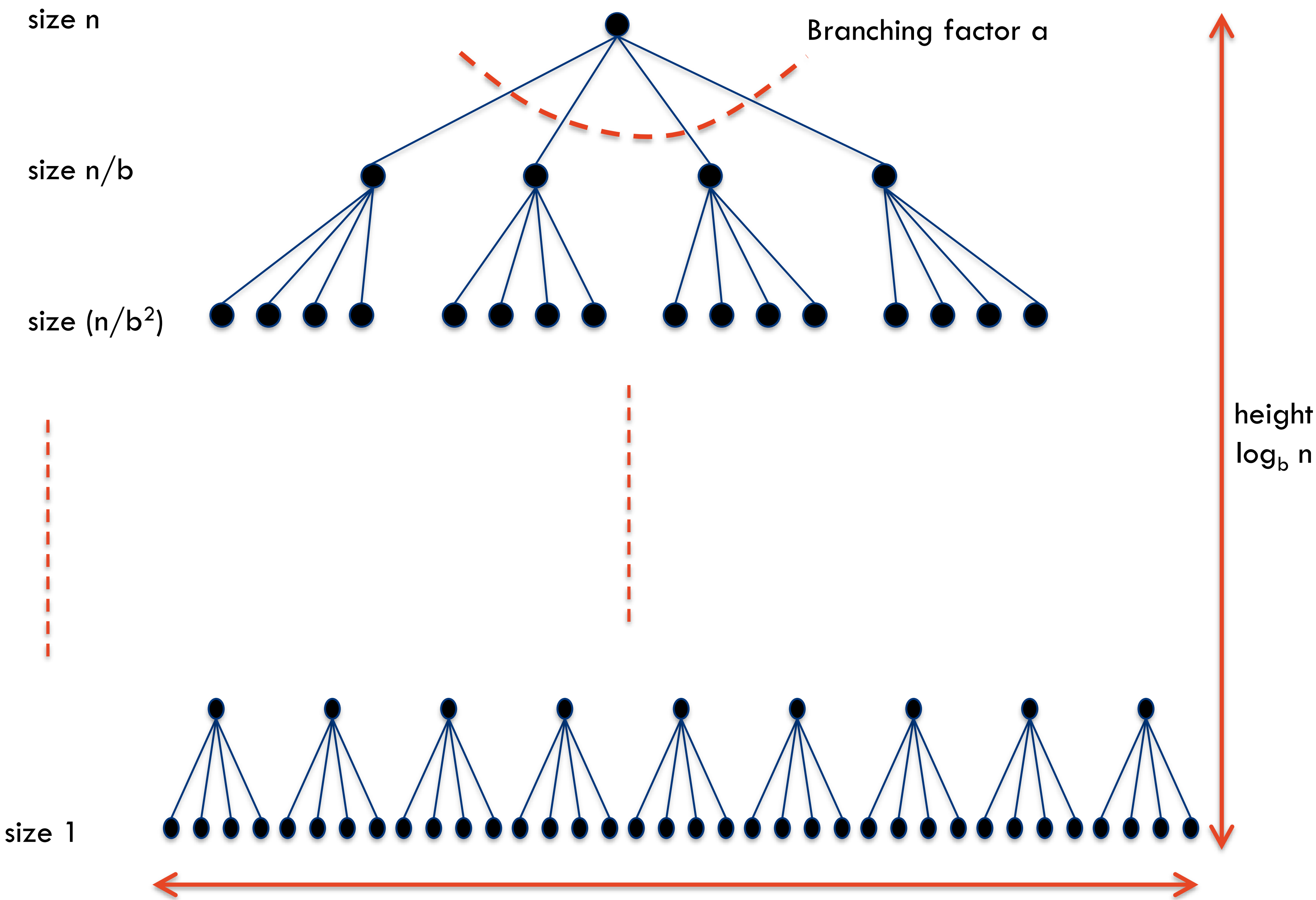
The master method

The master method applies to recurrences of the form

$$T(n) = a \cdot T(n/b) + f(n) \ ,$$

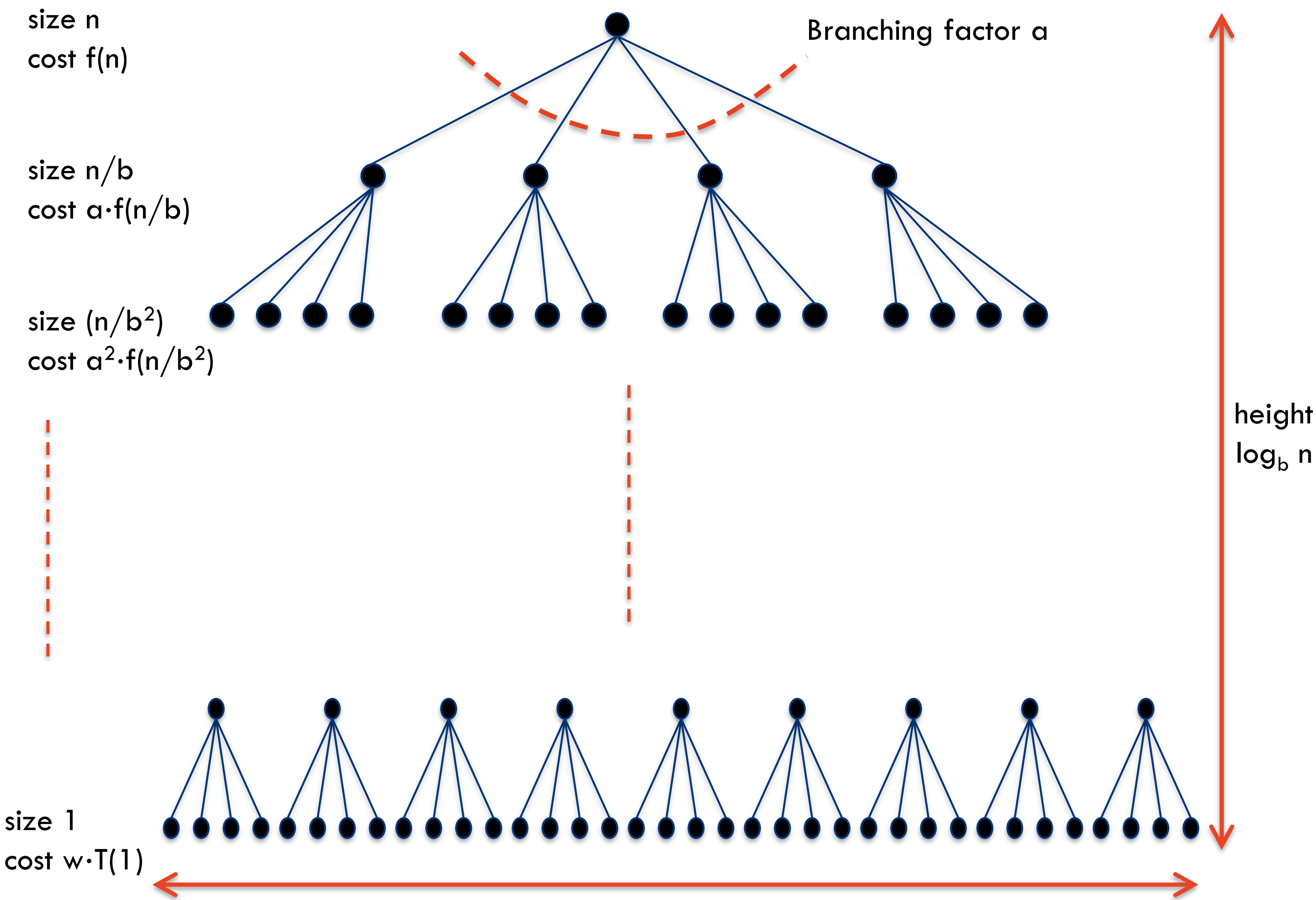
where $a \geq 1$, $b > 1$, and f is asymptotically positive.

$T(n) = a \cdot T(n/b) + f(n)$



width $w = a^{\log_b n} = n^{\log_b a}$

$T(n) = a \cdot T(n/b) + f(n)$



$\text{width } w = a^{\log_b n} = n^{\log_b a}$

Three common cases

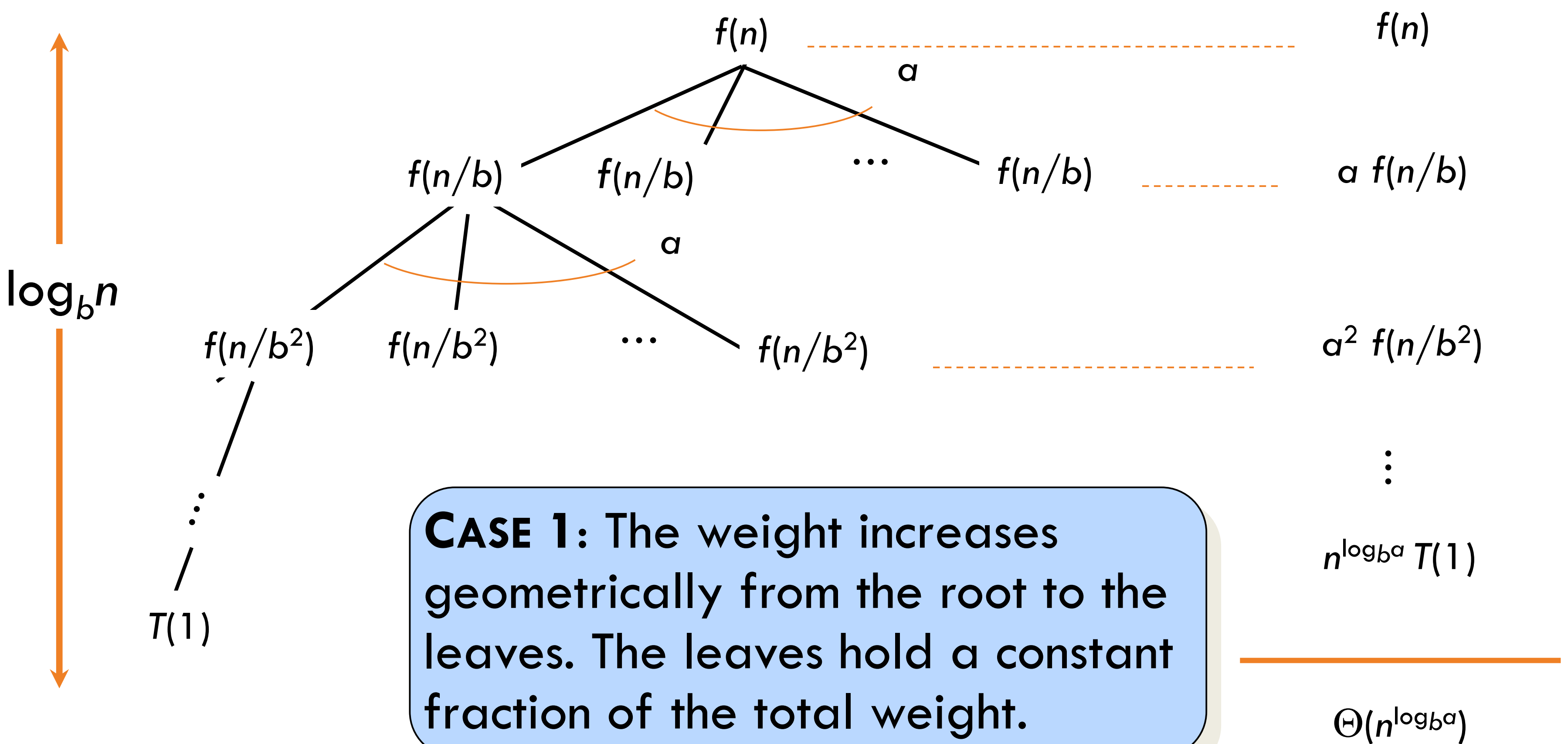
Compare f(n) with $n^{\log_b a}$:

Case 1:

If $f(n) = O(n^{\log_b a - \varepsilon})$ for any constant $\varepsilon>0$ then f(n) grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

Solution: $T(n) = \Theta(n^{\log_b a})$.

Idea of master theorem: Case 1



CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

Three common cases [Compare f(n) with $n^{\log_b a - \varepsilon}$]

Case 1: Example

$T(n) = 8T(n/2) + 10n^2$

$\Rightarrow$ a= 8, b=2 and $f(n)=10n^2$

$f(n) = 10n^2$

$n^{\log_b a - \varepsilon} = n^{\log_2 8 - \varepsilon} = O(n^{3 - \varepsilon})$ for $\varepsilon=1>0$.

$\Rightarrow f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow$ Case 1 holds.

Solution: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$

Three common cases (cont'd) [Compare $f(n)$ with $n^{\log_b a - \varepsilon}$]

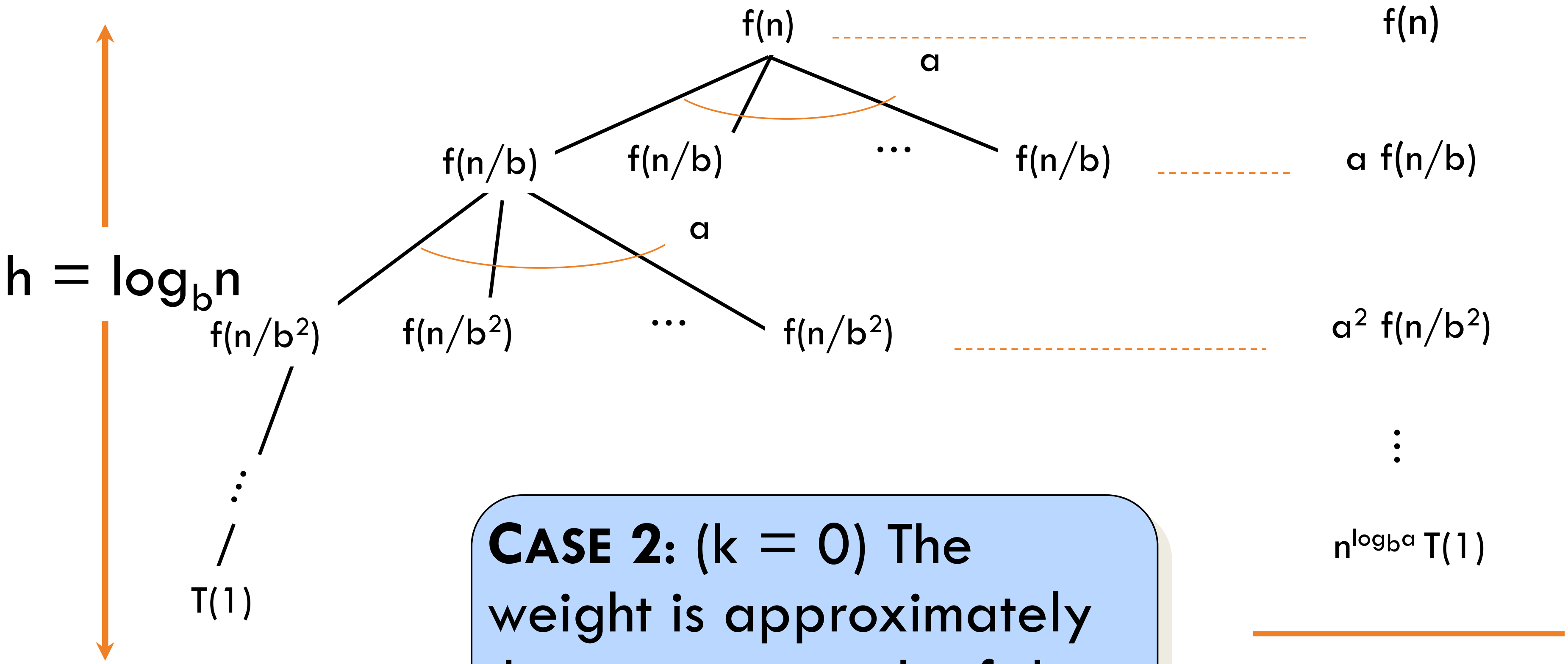
Case 2:

If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$ then $f(n)$ and $n^{\log_b a}$ grow at similar rates.

Solution: $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Idea of master theorem

Recursion tree:



CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

Three common cases [Compare f(n) with $n^{\log_b a - \varepsilon}$]

Case 2: Example

$T(n) = 2T(n/2) + n \log n$

$\Rightarrow$ a= 2, b=2 and f(n)=n log n

$f(n) = n \log n = \Theta(n^{\log_b a} \log^k n) = \Theta(n \log n)$ for k=1

$\Rightarrow$ Case 2 holds.

Solution: $T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n \log^2 n)$

Three common cases (cont.) [Compare $f(n)$ with $n^{\log_b a - \varepsilon}$]

Case 3:

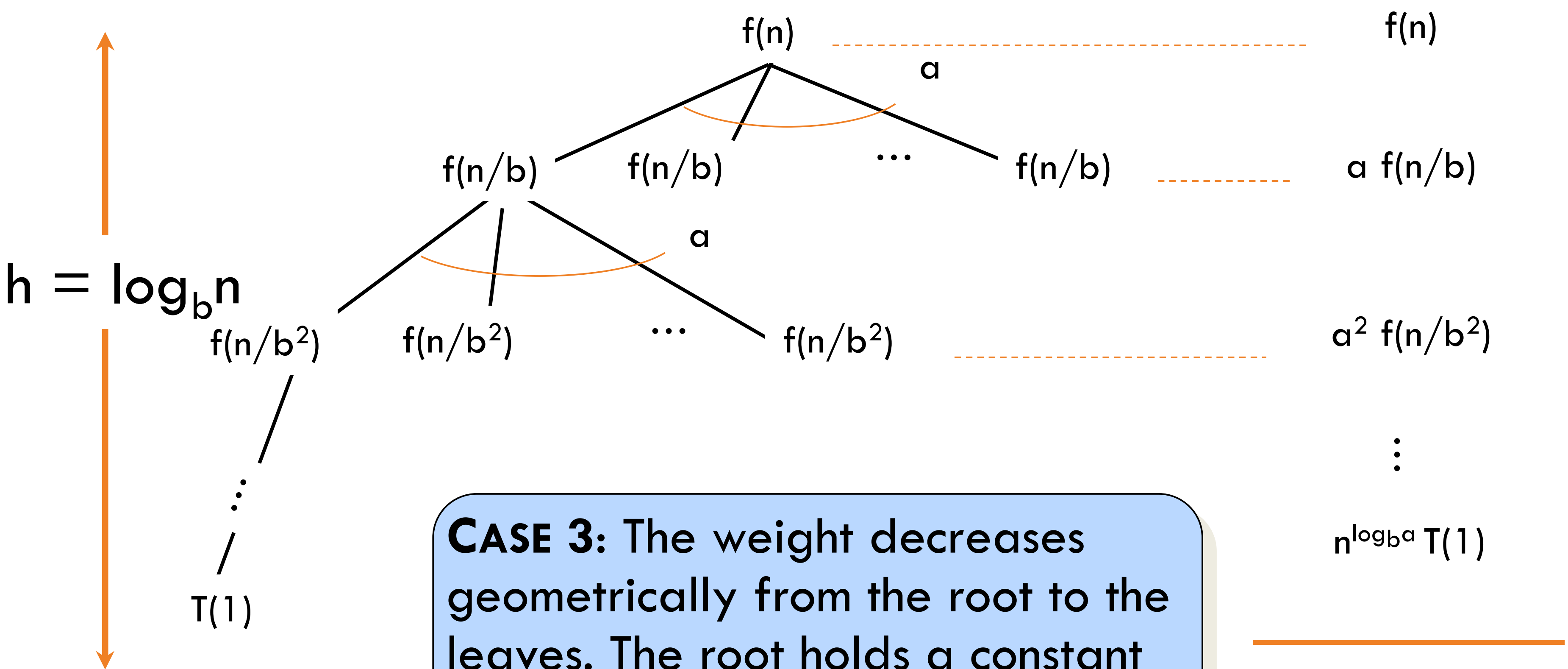
If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.

- $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor), and
- $f(n)$ satisfies the **regularity condition** that $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$.

Solution: $T(n) = \Theta(f(n))$.

Idea of master theorem

Recursion tree:



CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

Three common cases [Compare f(n) with $n^{\log_b a - \varepsilon}$]

$T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \quad \Rightarrow \quad n^{\log_b a} = n^2$ and $f(n) = n^3$.

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and

$4(cn/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.

Solution: $T(n) = \Theta(n^3)$

Integer Multiplication

Integer Arithmetic

- **Add.** Given two n-digit integers a and b, compute a + b.
  - $O(n)$ bit operations.

- **Multiply.** Given two n-digit integers a and b, compute a × b.
  - Brute force solution: $\Theta(n^2)$ bit operations.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Add

Multiply

									1	1	0	1	0	1	0	1
								*	0	1	1	1	1	1	0	1
								1	1	0	1	0	1	0	1	0
							0	0	0	0	0	0	0	0	0	
						1	1	0	1	0	1	0	1	0		
					1	1	0	1	0	1	0	1	0			
				1	1	0	1	0	1	0	1	0				
			1	1	0	1	0	1	0	1	0					
		1	1	0	1	0	1	0	1	0						
	0	0	0	0	0	0	0	0	0							
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0

Divide-and-Conquer Multiplication: Warmup

- To multiply two n-digit integers:
  - Given two numbers A and B with n bits each.
  - Partition the n bits into the n/2 “high” bits and the n/2 “low” bits.

A	A_H	A_L
B	B_H	B_L

$$A = A_H \cdot 2^{n/2} + A_L$$

$$B = B_H \cdot 2^{n/2} + B_L$$

Divide-and-Conquer Multiplication: Warmup

- To multiply two n-digit integers:
  - Given two numbers A and B with n bits each.
  - Partition the n bits into the n/2 “high” bits and the n/2 “low” bits.

A	A_H	A_L
B	B_H	B_L

$$A = A_H \cdot 2^{n/2} + A_L$$

$$B = B_H \cdot 2^{n/2} + B_L$$

$$\begin{aligned} \rightarrow\ A \cdot B &= (A_H \cdot 2^{n/2} + A_L) \cdot (B_H \cdot 2^{n/2} + B_L) \\ &= A_H\, B_H \cdot 2^n + A_H\, B_L \cdot 2^{n/2} + A_L\, B_H \cdot 2^{n/2} + A_L\, B_L \end{aligned}$$

- 4 multiplications of n/2-bit numbers: A_HB_H, A_HB_L, A_LB_H, A_LB_L, additions and shifts. Multiplications by powers of 2 are just shifts.

Divide-and-Conquer Multiplication: Warmup

- To multiply two n-digit integers:
  - Given two numbers A and B with n bits each.
  - Partition the n bits into the n/2 "high" bits and the n/2 "low" bits.

A	A_H	A_L
B	B_H	B_L

$$A = A_H \cdot 2^{n/2} + A_L$$

$$B = B_H \cdot 2^{n/2} + B_L$$

$$\rightarrow A \cdot B = (A_H \cdot 2^{n/2} + A_L) \cdot (B_H \cdot 2^{n/2} + B_L)$$

$$= A_H B_H \cdot 2^n + A_H B_L \cdot 2^{n/2} + A_L B_H \cdot 2^{n/2} + A_L B_L$$

$$\mathrm{T}(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow \mathrm{T}(n) = \Theta(n^2)$$

Karatsuba Multiplication [1960]

– Multiply two n-digit integers

A	A_H	A_L
B	B_H	B_L

$$A = A_H \cdot 2^{n/2} + A_L$$

$$B = B_H \cdot 2^{n/2} + B_L$$

Observation:

$$A{\cdot}B = A_H B_H \cdot 2^n + [(A_H + A_L){\cdot}(B_H + B_L) - A_H B_H - A_L B_L\,] \cdot 2^{n/2} + A_L B_L$$

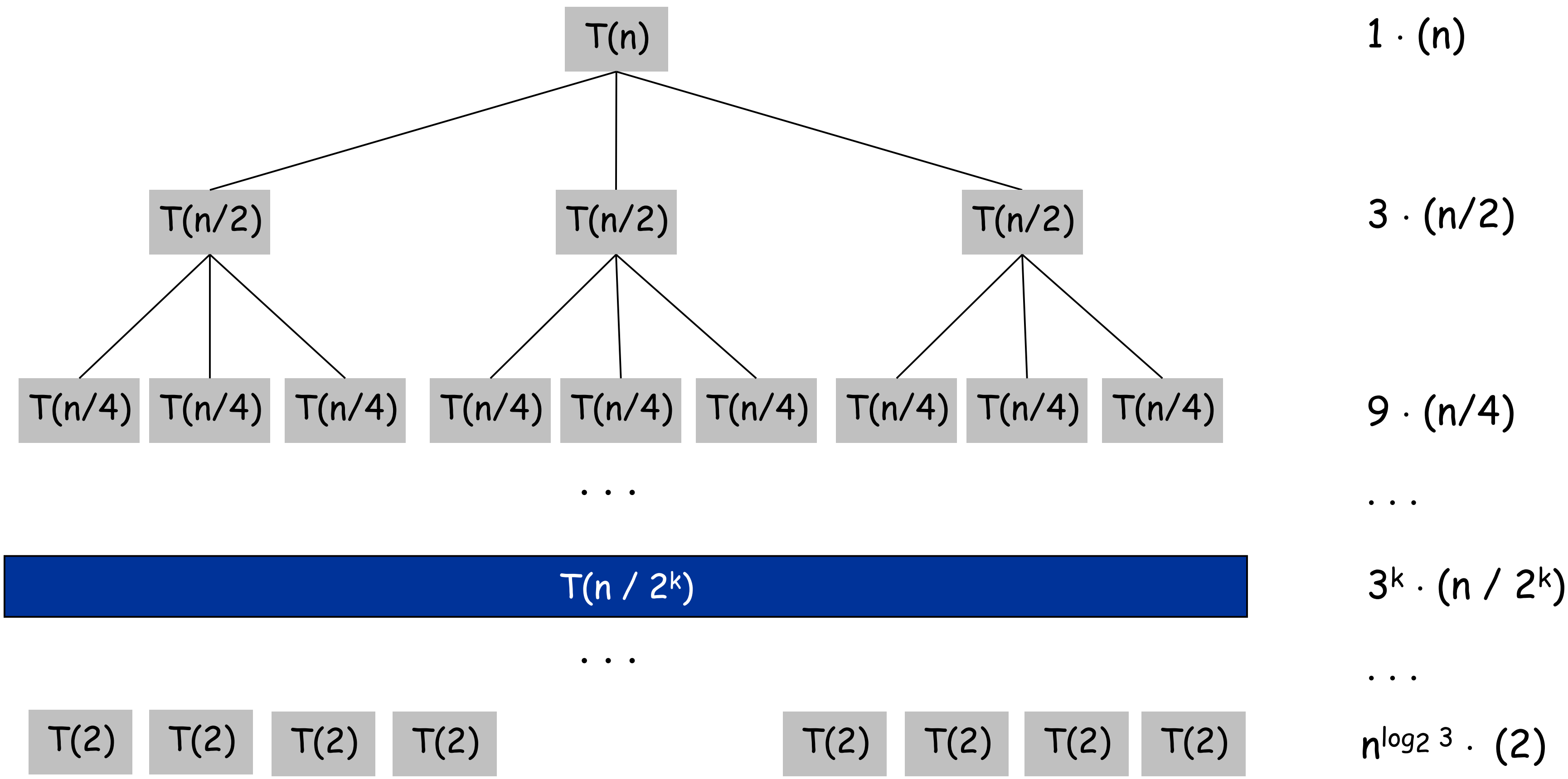
Theorem: [Karatsuba-Ofman, 1962]

3 multiplications of n/2-bit numbers + additions, subtractions and shifts.

Karatsuba: Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\log_2 n} n\left(\tfrac{3}{2}\right)^k = \frac{\left(\tfrac{3}{2}\right)^{1+\log_2 n} - 1}{\tfrac{3}{2} - 1} = 3n^{\log_2 3} - 2$$
$$= O(n^{1.59})$$



Summary: Divide-and-Conquer

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

- **Master theorem**

- **Problems**
  - Merge Sort
  - Closest pair
  - Multiplication

This weeks quiz is all about solving recursions!