CSE 431/531: Analysis of Algorithms

Divide-and-Conquer

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

Outline

- Greedy algorithm: design efficient algorithms

- Greedy algorithm: design efficient algorithms
- Divide-and-conquer: design more efficient algorithms

Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

merge-sort(A, n)

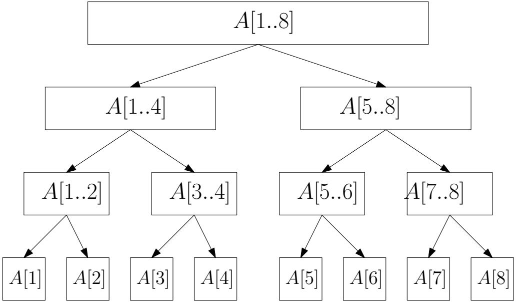
1. if $n = 1$ then
2. return A
3. else
4. $B \leftarrow$ merge-sort$\Big(A\big[1..\lfloor n/2 \rfloor\big], \lfloor n/2 \rfloor\Big)$
5. $C \leftarrow$ merge-sort$\Big(A\big[\lfloor n/2 \rfloor + 1..n\big], \lceil n/2 \rceil\Big)$
6. return merge$(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$

merge-sort(A, n)

1. if $n = 1$ then
2.   return A
3. else
4. $B \leftarrow$ merge-sort$\Big(A\big[1..\lfloor n/2\rfloor\big], \lfloor n/2\rfloor\Big)$
5. $C \leftarrow$ merge-sort$\Big(A\big[\lfloor n/2\rfloor + 1..n\big], \lceil n/2\rceil\Big)$
6. return merge$(B, C, \lfloor n/2\rfloor, \lceil n/2\rceil)$

- Divide: trivial
- Conquer: 4, 5
- Combine: 6

Running Time for Merge-Sort



- Each level takes running time $O(n)$
- There are $O(\lg n)$ levels
- Running time $= O(n \lg n)$
- Better than insertion sort

Running Time for Merge-Sort Using Recurrence

- $T(n)$ = running time for sorting n numbers,then

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

Running Time for Merge-Sort Using Recurrence

- $T(n)$ = running time for sorting n numbers,then

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

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

Running Time for Merge-Sort Using Recurrence

- $T(n)$ = running time for sorting n numbers,then

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

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

- Even simpler: $T(n) = 2T(n/2) + O(n)$. (Implicit assumption: $T(n) = O(1)$ if n is at most some constant.)

Running Time for Merge-Sort Using Recurrence

- $T(n)$ = running time for sorting n numbers,then

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

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

- Even simpler: $T(n) = 2T(n/2) + O(n)$. (Implicit assumption: $T(n) = O(1)$ if n is at most some constant.)
- Solving this recurrence, we have $T(n) = O(n \lg n)$ (we shall show how later)

Outline

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:

$10 \quad 8 \quad 15 \quad 9 \quad 12$

Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:

10	8	15	9	12
8	9	10	12	15

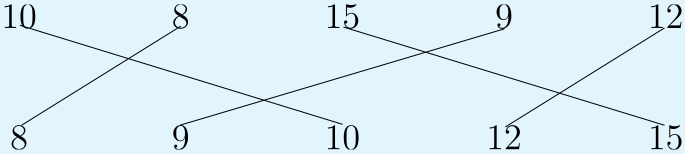
Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:



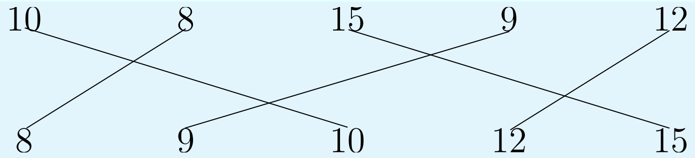
Def. Given an array A of n integers, an inversion in A is a pair (i, j) of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

Input: an sequence A of n numbers

Output: number of inversions in A

Example:



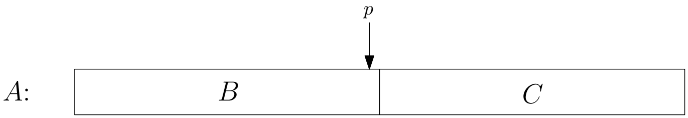
- 4 inversions (for convenience, using numbers, not indices): $(10, 8), (10, 9), (15, 9), (15, 12)$

Naive Algorithm for Counting Inversions

count-inversions(A, n)

```
c ← 0
for every i ← 1 to n − 1
    for every j ← i + 1 to n
        if A[i] > A[j] then c ← c + 1
return c
```

Divide-and-Conquer



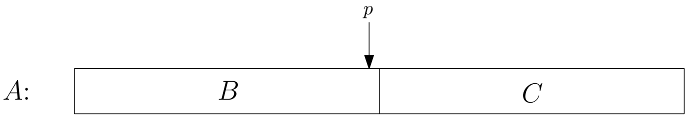
- $p = \lfloor n/2 \rfloor, B = A[1..p], C = A[p+1..n]$
- $$\#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m$$
  $$m = \Big|\big\{(i,j) : B[i] > C[j]\big\}\Big|$$

Q: How fast can we compute m, via trivial algorithm?

A: $O(n^2)$

- Can not improve the $O(n^2)$ time for counting inversions.

Divide-and-Conquer



- $p = \lfloor n/2 \rfloor, B = A[1..p], C = A[p+1..n]$
- $$\#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m$$
  $$m = \left|\left\{(i,j) : B[i] > C[j]\right\}\right|$$

Lemma If both B and C are sorted, then we can compute m in $O(n)$ time!

Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B:

3	8	12	20	32	48

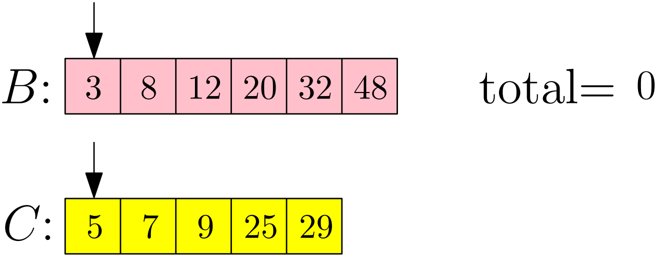
total $= 0$

C:

5	7	9	25	29

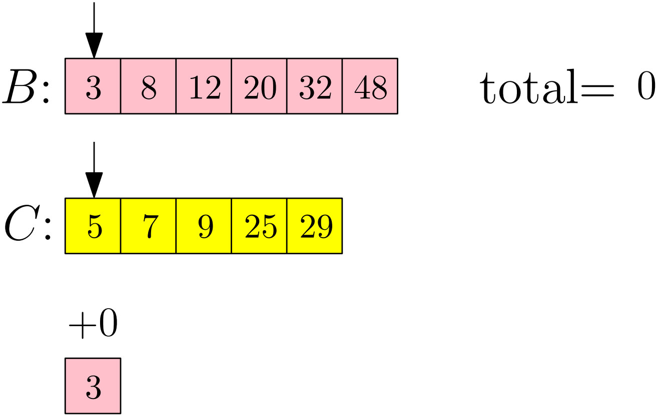
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B: | 3 | 8 | 12 | 20 | 32 | 48 |

total$=$ 0

C: | 5 | 7 | 9 | 25 | 29 |

$+0$

| 3 |

Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B: | 3 | 8 | 12 | 20 | 32 | 48 |

C: | 5 | 7 | 9 | 25 | 29 |

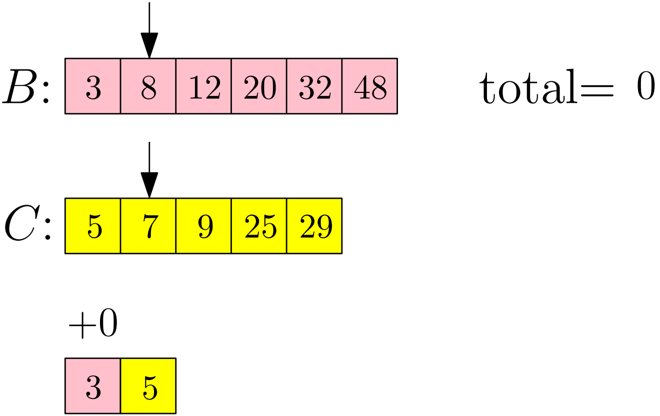
total$=$ 0

$+0$

| 3 | 5 |

Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

B:

3	8	12	20	32	48

total$=$ 0

C:

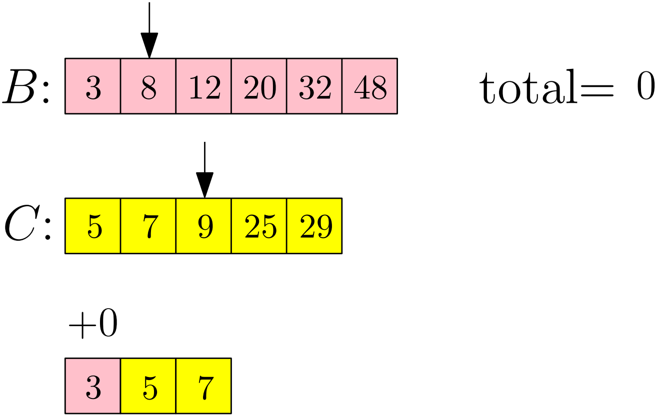
5	7	9	25	29

$+0$

3	5	7

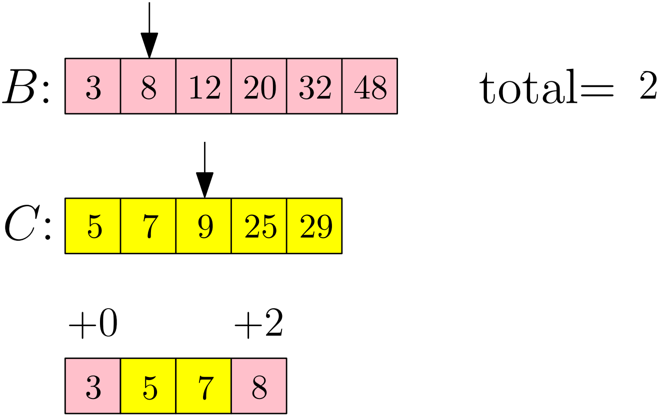
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



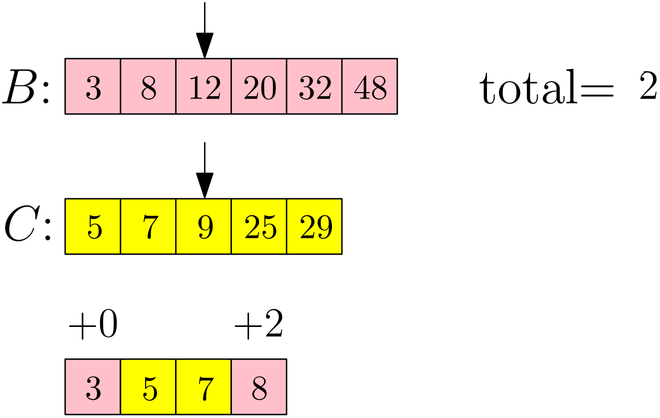
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



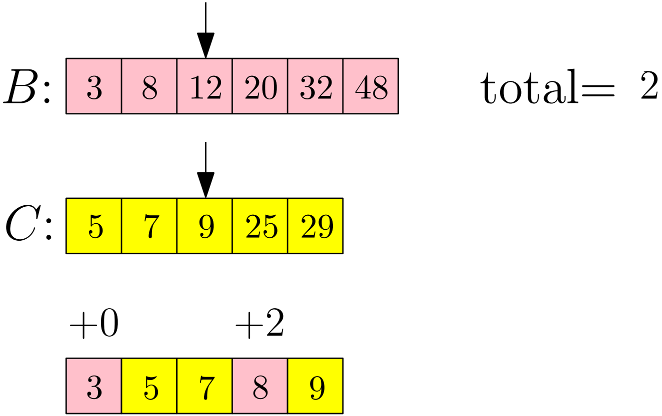
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



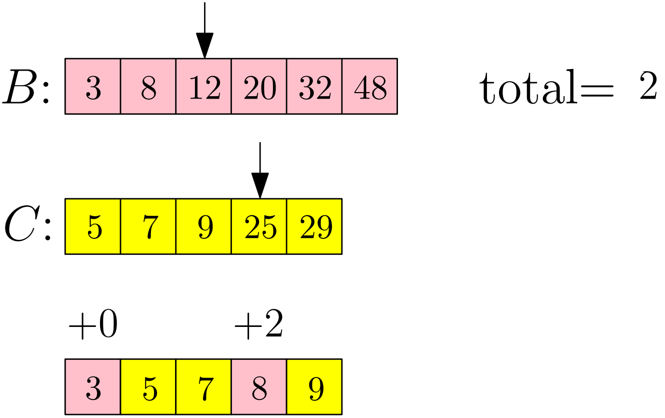
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

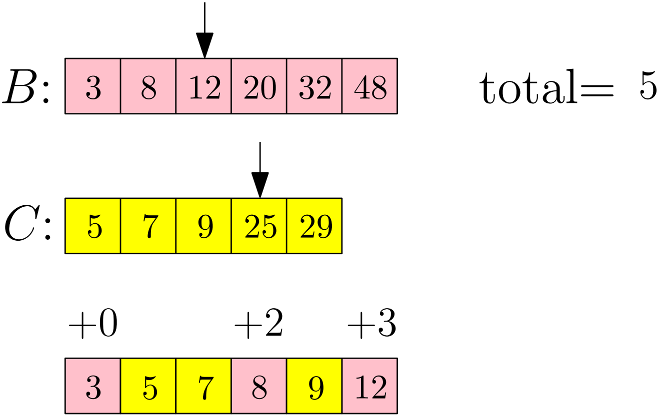


Counting Inversions between B and C

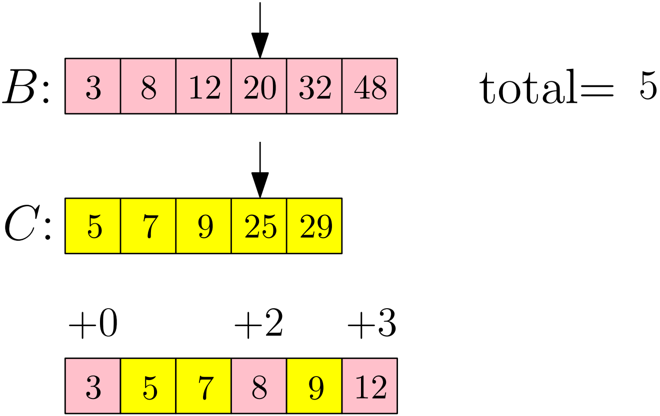
Count pairs i, j such that $B[i] > C[j]$:



Count pairs i, j such that $B[i] > C[j]$:

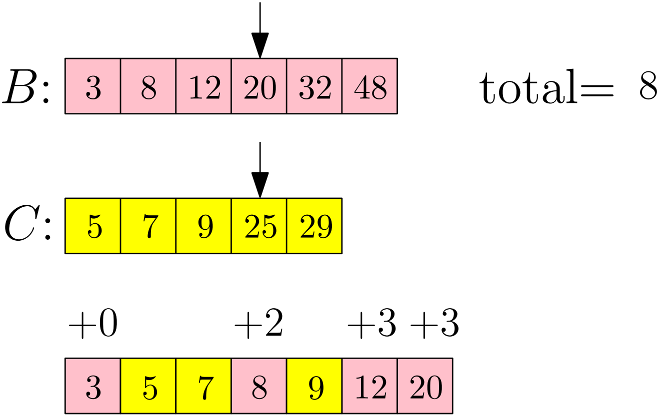


Count pairs i, j such that $B[i] > C[j]$:



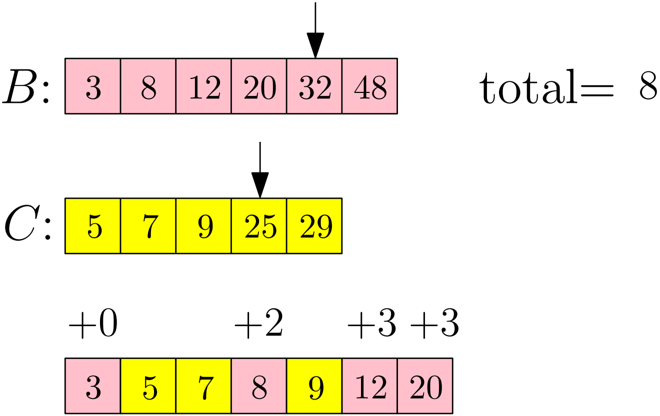
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:



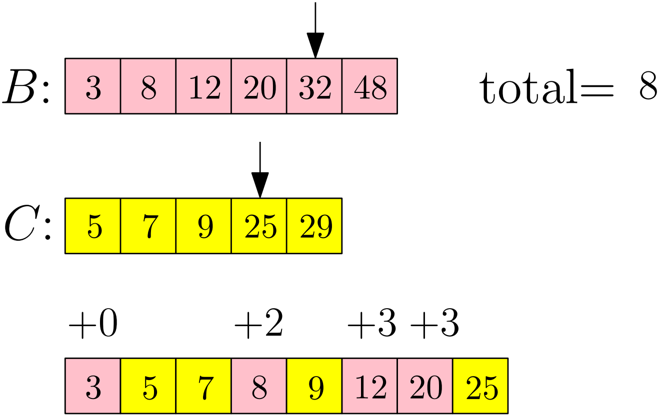
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

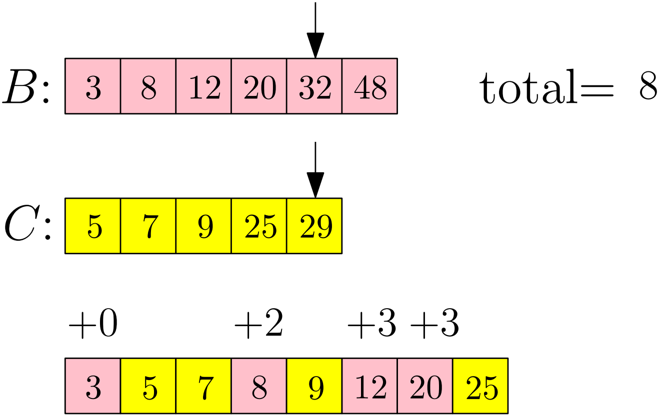


Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

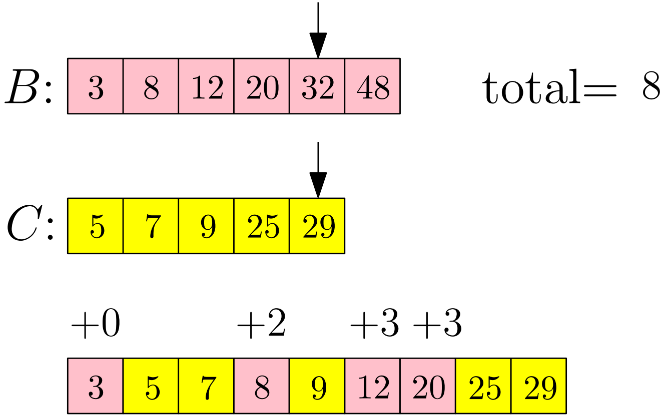


Count pairs i, j such that $B[i] > C[j]$:



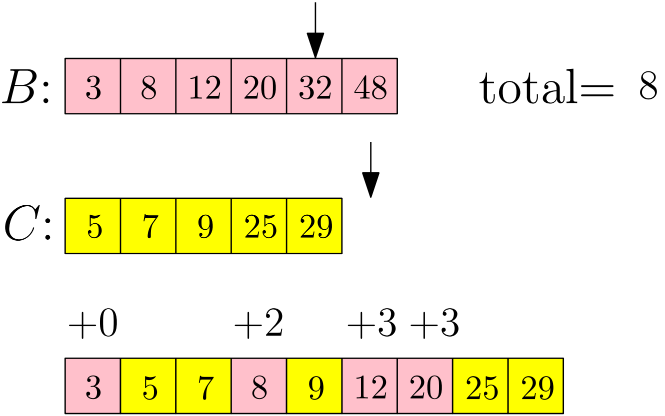
Counting Inversions between B and C

Count pairs i, j such that $B[i] > C[j]$:

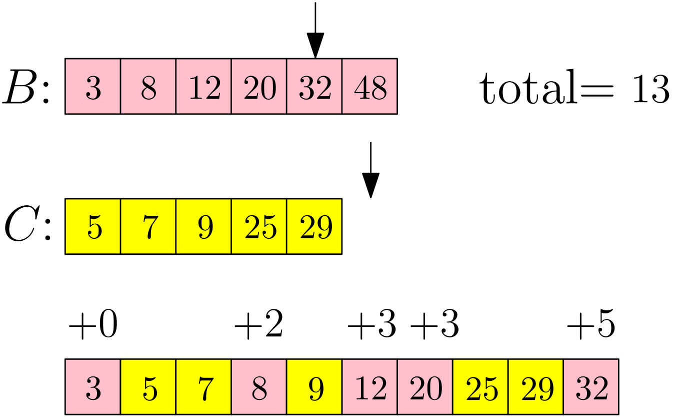


Counting Inversions between B and C

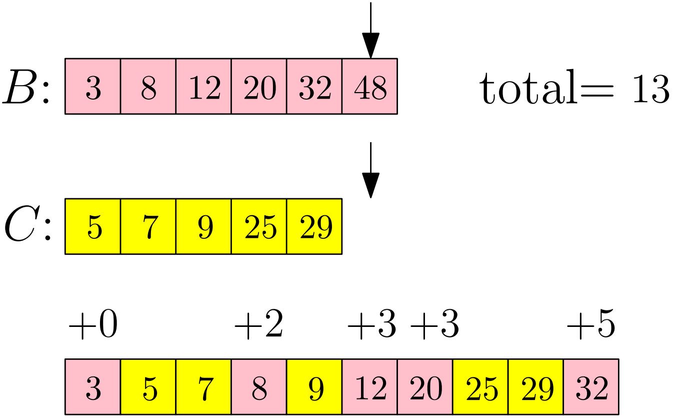
Count pairs i, j such that $B[i] > C[j]$:



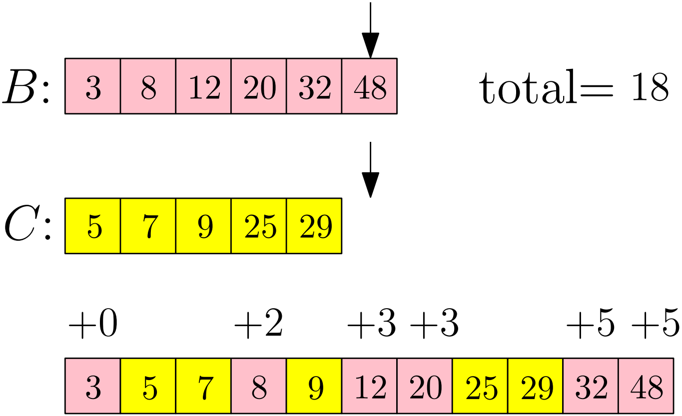
Count pairs i, j such that $B[i] > C[j]$:



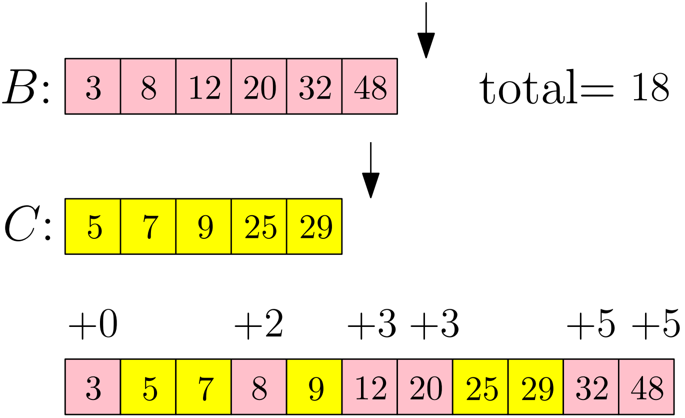
Count pairs i, j such that $B[i] > C[j]$:



Count pairs i, j such that $B[i] > C[j]$:



Count pairs i, j such that $B[i] > C[j]$:



Count Inversions between B and C

- Procedure that merges B and C and counts inversions between B and C at the same time

merge-and-count(B, C, n_1, n_2)

```
count ← 0;
A ← []; i ← 1; j ← 1
while i ≤ n₁ or j ≤ n₂
    if j > n₂ or (i ≤ n₁ and B[i] ≤ C[j]) then
        append B[i] to A; i ← i + 1
        count ← count + (j − 1)
    else
        append C[j] to A; j ← j + 1
return (A, count)
```

Sort and Count Inversions in A

- A procedure that returns the sorted array of A and counts the number of inversions in A:

sort-and-count(A, n)

1. if $n = 1$ then
2. return $(A, 0)$
3. else
4. $(B, m_1) \leftarrow$ sort-and-count$\Big(A\big[1..\lfloor n/2 \rfloor\big], \lfloor n/2 \rfloor\Big)$
5. $(C, m_2) \leftarrow$ sort-and-count$\Big(A\big[\lfloor n/2 \rfloor + 1..n\big], \lceil n/2 \rceil\Big)$
6. $(A, m_3) \leftarrow$ merge-and-count$(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
7. return $(A, m_1 + m_2 + m_3)$

Sort and Count Inversions in A

- A procedure that returns the sorted array of A and counts the number of inversions in A:

sort-and-count(A, n)

1. if $n = 1$ then
2.   return $(A, 0)$
3. else
4. $(B, m_1) \leftarrow$ sort-and-count$\Big(A\big[1..\lfloor n/2\rfloor\big], \lfloor n/2\rfloor\Big)$
5. $(C, m_2) \leftarrow$ sort-and-count$\Big(A\big[\lfloor n/2\rfloor + 1..n\big], \lceil n/2\rceil\Big)$
6. $(A, m_3) \leftarrow$ merge-and-count$(B, C, \lfloor n/2\rfloor, \lceil n/2\rceil)$
7.   return $(A, m_1 + m_2 + m_3)$

- Divide: trivial
- Conquer: 4, 5
- Combine: 6, 7

sort-and-count(A, n)

```
if n = 1 then
    return (A, 0)
else
    (B, m_1) ← sort-and-count(A[1..⌊n/2⌋], ⌊n/2⌋)
    (C, m_2) ← sort-and-count(A[⌊n/2⌋ + 1..n], ⌈n/2⌉)
    (A, m_3) ← merge-and-count(B, C, ⌊n/2⌋, ⌈n/2⌉)
    return (A, m_1 + m_2 + m_3)
```

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$

sort-and-count(A, n)

1. if $n = 1$ then
2. return $(A, 0)$
3. else
4. $(B, m_1) \leftarrow$ sort-and-count$\Big(A\big[1..\lfloor n/2 \rfloor\big], \lfloor n/2 \rfloor\Big)$
5. $(C, m_2) \leftarrow$ sort-and-count$\Big(A\big[\lfloor n/2 \rfloor + 1..n\big], \lceil n/2 \rceil\Big)$
6. $(A, m_3) \leftarrow$ merge-and-count$(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
7. return $(A, m_1 + m_2 + m_3)$

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$
- Running time $= O(n \lg n)$

Outline

Outline

Quicksort vs Merge-Sort

	Merge Sort	**Quicksort**
Divide	Trivial	Separate small and big numbers
Conquer	Recurse	Recurse
Combine	Merge 2 sorted arrays	Trivial

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

29	38	45	25	15	37	17	64	82	75	94	92	69	76	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

29	38	45	25	15	37	17	64	82	75	94	92	69	76	85

Quicksort Example

Assumption We can choose median of an array of size n in $O(n)$ time.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

29	38	45	25	15	37	17	64	82	75	94	92	69	76	85

25	15	17	29	38	45	37	64	82	75	94	92	69	76	85

Quicksort

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ lower median of A
3. $A_L \leftarrow$ elements in A that are less than x \\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L.\text{size})$ \\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R.\text{size})$ \\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

Quicksort

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ lower median of A
3. $A_L \leftarrow$ elements in A that are less than x \\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L$.size$)$ \\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R$.size$)$ \\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

- Recurrence $T(n) \leq 2T(n/2) + O(n)$

Quicksort

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ lower median of A
3. $A_L \leftarrow$ elements in A that are less than x \\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L$.size$)$ \\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R$.size$)$ \\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

- Recurrence $T(n) \leq 2T(n/2) + O(n)$
- Running time $= O(n \lg n)$

Assumption We can choose median of an array of size n in $O(n)$ time.

Q: How to remove this assumption?

Assumption We can choose median of an array of size n in $O(n)$ time.

Q: How to remove this assumption?

A:

1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)

Assumption We can choose median of an array of size n in $O(n)$ time.

Q: How to remove this assumption?

A:

1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)
2. Choose a pivot randomly and pretend it is the median (it is practical)

Quicksort Using A Random Pivot

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ a random element of A (x is called a pivot)
3. $A_L \leftarrow$ elements in A that are less than x \\\\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\\\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L$.size$)$ \\\\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R$.size$)$ \\\\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0,1]$.

Q: Can computers really produce random numbers?

A: No! The execution of a computer programs is deterministic!

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?

A: No! The execution of a computer programs is deterministic!

- In practice: use pseudo-random-generator, a deterministic algorithm returning numbers that "look like" random

Randomized Algorithm Model

Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?

A: No! The execution of a computer programs is deterministic!

- In practice: use pseudo-random-generator, a deterministic algorithm returning numbers that "look like" random
- In theory: make the assumption

Quicksort Using A Random Pivot

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ a random element of A (x is called a pivot)
3. $A_L \leftarrow$ elements in A that are less than x \\\\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\\\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L.\text{size})$ \\\\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R.\text{size})$ \\\\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

- When we talk about randomized algorithm in the future, we show that the expected running time of the algorithm is $O(n \lg n)$.

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

29	82	75	64	38	45	94	69	25	76	15	92	37	17	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

64	82	75	29	38	45	94	69	25	76	15	92	37	17	85

Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

i														j
64	82	75	29	38	45	94	69	25	76	15	92	37	17	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

i													j	
64	82	75	29	38	45	94	69	25	76	15	92	37	17	85

Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

i													j	
17	82	75	29	38	45	94	69	25	76	15	92	37	64	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

	i												j	
17	82	75	29	38	45	94	69	25	76	15	92	37	64	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

17	64	75	29	38	45	94	69	25	76	15	92	37	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

	i											j		
17	64	75	29	38	45	94	69	25	76	15	92	37	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

	i											j		
17	37	75	29	38	45	94	69	25	76	15	92	64	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

		i										j		
17	37	75	29	38	45	94	69	25	76	15	92	64	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

		i										j		
17	37	64	29	38	45	94	69	25	76	15	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

		i								j				
17	37	64	29	38	45	94	69	25	76	15	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

		i								j				
17	37	15	29	38	45	94	69	25	76	64	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

						i				j				
17	37	15	29	38	45	94	69	25	76	64	92	75	82	85

Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

						i				j				
17	37	15	29	38	45	64	69	25	76	94	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

						i		j						
17	37	15	29	38	45	64	69	25	76	94	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

						i		j						
17	37	15	29	38	45	25	69	64	76	94	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

							i	j						
17	37	15	29	38	45	25	69	64	76	94	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

							i	j						
17	37	15	29	38	45	25	64	69	76	94	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

i j

17	37	15	29	38	45	25	64	69	76	94	92	75	82	85

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

i j

17	37	15	29	38	45	25	64	69	76	94	92	75	82	85

- To partition the array into two parts, we only need $O(1)$ extra space.

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

partition(A, ℓ, r)

```
p ← random integer between ℓ and r
swap A[p] and A[ℓ]
i ← ℓ, j ← r
while i < j do
    while i < j and A[i] ≤ A[j] do j ← j − 1
    swap A[i] and A[j]
    while i < j and A[i] ≤ A[j] do i ← i + 1
    swap A[i] and A[j]
return i
```

Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

quicksort(A, ℓ, r)

1. if $\ell \geq r$ return
2. $p \leftarrow$ patition(A, ℓ, r)
3. $q \leftarrow p - 1$; while $A[q] = A[p]$ and $q \geq \ell$ do: $q \leftarrow q - 1$
4. quicksort(A, ℓ, q)
5. $q \leftarrow p + 1$; while $A[q] = A[p]$ and $q \leq r$ do: $q \leftarrow q + 1$
6. quicksort(A, q, r)

- To sort an array A of size n, call quicksort$(A, 1, n)$.

Note: We pass the array A by reference, instead of by copying.

Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

Merge-Sort is Not In-Place

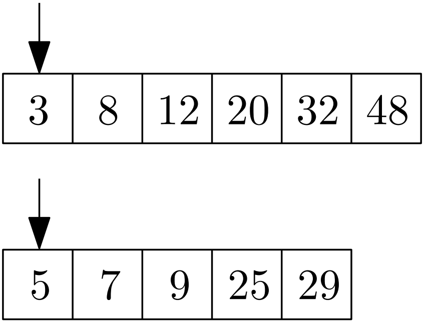
- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

3

Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

3

Merge-Sort is Not In-Place

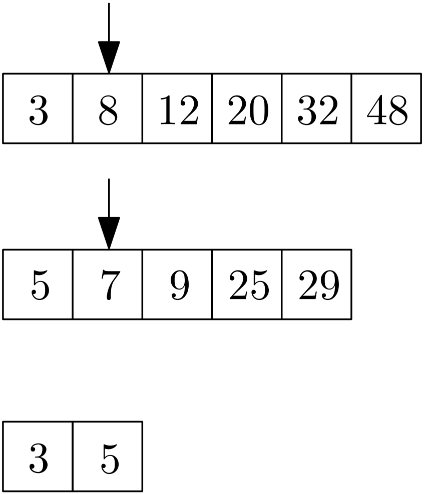
- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

3	5

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

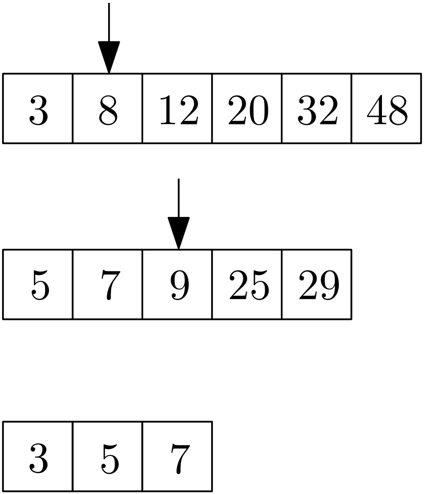
3	8	12	20	32	48

5	7	9	25	29

3	5	7

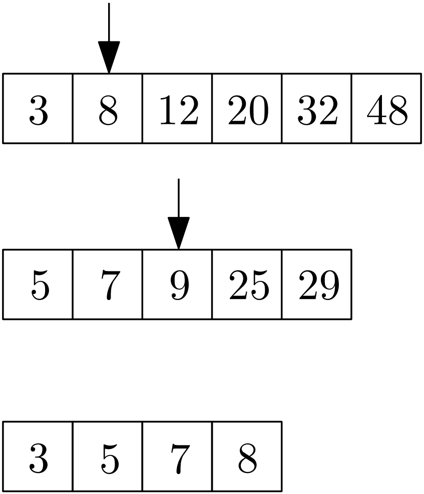
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



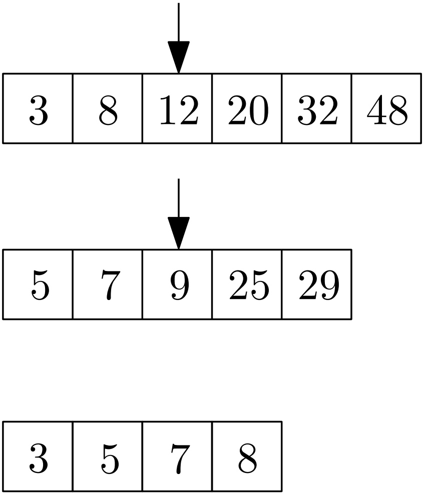
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

3	5	7	8	9	12	20	25	29

Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

3	5	7	8	9	12	20	25	29	32	48

Outline

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

- To sort, we are only allowed to compare two elements
- We can not use "internal structures" of the elements

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Q: How many questions do you need to ask Bob in order to know x?

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Q: How many questions do you need to ask Bob in order to know x?

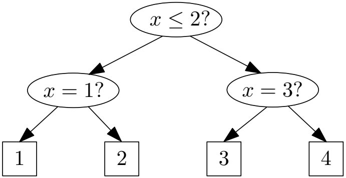
A: $\lceil \log_2 N \rceil$.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Q: How many questions do you need to ask Bob in order to know x?

A: $\lceil \log_2 N \rceil$.



Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob "yes/no" questions about π.

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob "yes/no" questions about π.

Q: How many questions do you need to ask in order to get the permutation π?

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob "yes/no" questions about π.

Q: How many questions do you need to ask in order to get the permutation π?

A: $\log_2 n! = \Theta(n \lg n)$

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form "does i appear before j in π?"

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form "does i appear before j in π?"

Q: How many questions do you need to ask in order to get the permutation π?

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form "does i appear before j in π?"

Q: How many questions do you need to ask in order to get the permutation π?

A: At least $\log_2 n! = \Theta(n \lg n)$

Outline

Selection Problem

Input: a set A of n numbers, and $1 \le i \le n$

Output: the i-th smallest number in A

Selection Problem

Input: a set A of n numbers, and $1 \leq i \leq n$

Output: the i-th smallest number in A

- Sorting solves the problem in time $O(n \lg n)$.

Selection Problem

Input: a set A of n numbers, and $1 \leq i \leq n$

Output: the i-th smallest number in A

- Sorting solves the problem in time $O(n \lg n)$.
- Our goal: $O(n)$ running time

Recall: Quicksort with Median Finder

quicksort(A, n)

1. if $n \leq 1$ then return A
2. $x \leftarrow$ lower median of A
3. $A_L \leftarrow$ elements in A that are less than x \\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\ Divide
5. $B_L \leftarrow$ quicksort$(A_L, A_L$.size$)$ \\ Conquer
6. $B_R \leftarrow$ quicksort$(A_R, A_R$.size$)$ \\ Conquer
7. $t \leftarrow$ number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

Selection Algorithm with Median Finder

selection(A, n, i)

if $n = 1$ then return A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ elements in A that are less than x \\ Divide
$A_R \leftarrow$ elements in A that are greater than x \\ Divide
if $i \leq A_L$.size then
    return selection$(A_L, A_L.\text{size}, i)$ \\ Conquer
elseif $i > n - A_R$.size then
    return select$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ \\ Conquer
else return x

Selection Algorithm with Median Finder

selection(A, n, i)

1. if $n = 1$ then return A
2. $x \leftarrow$ lower median of A
3. $A_L \leftarrow$ elements in A that are less than x \\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\ Divide
5. if $i \leq A_L$.size then
6. return selection$(A_L, A_L.\text{size}, i)$ \\ Conquer
7. elseif $i > n - A_R$.size then
8. return select$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ \\ Conquer
9. else return x

- Recurrence for selection: $T(n) = T(n/2) + O(n)$

Selection Algorithm with Median Finder

selection(A, n, i)

1. if $n = 1$ then return A
2. $x \leftarrow$ lower median of A
3. $A_L \leftarrow$ elements in A that are less than x \\\\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\\\ Divide
5. if $i \leq A_L$.size then
6.   return selection$(A_L, A_L.\text{size}, i)$ \\\\ Conquer
7. elseif $i > n - A_R$.size then
8.   return select$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ \\\\ Conquer
9. else return x

- Recurrence for selection: $T(n) = T(n/2) + O(n)$
- Solving recurrence: $T(n) = O(n)$

Randomized Selection Algorithm

selection(A, n, i)

1. if $n = 1$ then return A
2. $x \leftarrow$ random element of A (called pivot)
3. $A_L \leftarrow$ elements in A that are less than x \\\\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\\\ Divide
5. if $i \leq A_L$.size then
6.   return selection$(A_L, A_L.\text{size}, i)$ \\\\ Conquer
7. elseif $i > n - A_R$.size then
8.   return select$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ \\\\ Conquer
9. else return x

Randomized Selection Algorithm

selection(A, n, i)

1. if $n = 1$ then return A
2. $x \leftarrow$ random element of A (called pivot)
3. $A_L \leftarrow$ elements in A that are less than x \\\\ Divide
4. $A_R \leftarrow$ elements in A that are greater than x \\\\ Divide
5. if $i \leq A_L$.size then
6.   return selection$(A_L, A_L.\text{size}, i)$ \\\\ Conquer
7. elseif $i > n - A_R$.size then
8.   return select$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ \\\\ Conquer
9. else return x

- expected running time $= O(n)$

Outline

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Example:

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Example:

$$
\begin{aligned}
&(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5) \\
= {}& 6x^6 - 9x^5 + 18x^4 - 15x^3 \\
&+ 4x^5 - 6x^4 + 12x^3 - 10x^2 \\
&- 10x^4 + 15x^3 - 30x^2 + 25x \\
&+ 8x^3 - 12x^2 + 24x - 20 \\
= {}& 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20
\end{aligned}
$$

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Example:

$$
\begin{aligned}
&(3x^3+2x^2-5x+4)\times(2x^3-3x^2+6x-5)\\
=\ &6x^6-9x^5+18x^4-15x^3\\
&+4x^5-6x^4+12x^3-10x^2\\
&-10x^4+15x^3-30x^2+25x\\
&+8x^3-12x^2+24x-20\\
=\ &6x^6-5x^5+2x^4+20x^3-52x^2+49x-20
\end{aligned}
$$

- **Input**: $(4,-5,2,3),(-5,6,-3,2)$
- **Output**: $(-20,49,-52,20,2,-5,6)$

Naïve Algorithm

polynomial-multiplication(A, B, n)

1. let $C[k] = 0$ for every $k = 0, 1, 2, \cdots, 2n - 2$
2. for $i \leftarrow 0$ to $n - 1$
3.   for $j \leftarrow 0$ to $n - 1$
4. $C[i + j] \leftarrow C[i + j] + A[i] \times B[j]$
5. return C

Naïve Algorithm

polynomial-multiplication(A, B, n)

1. let $C[k] = 0$ for every $k = 0, 1, 2, \cdots, 2n - 2$
2. for $i \leftarrow 0$ to $n - 1$
3. for $j \leftarrow 0$ to $n - 1$
4. $C[i + j] \leftarrow C[i + j] + A[i] \times B[j]$
5. return C

Running time: $O(n^2)$

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x+2)x^2 + (-5x+4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x-3)x^2 + (6x-5)$$

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5)$$

- $p(x)$: degree of $n - 1$ (assume n is even)
- $p(x) = p_H(x)x^{n/2} + p_L(x)$,
- $p_H(x), p_L(x)$: polynomials of degree $n/2 - 1$.

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5)$$

- $p(x)$: degree of $n - 1$ (assume n is even)
- $p(x) = p_H(x)x^{n/2} + p_L(x)$,
- $p_H(x), p_L(x)$: polynomials of degree $n/2 - 1$.

$$pq = \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right)$$

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5)$$

- $p(x)$: degree of $n - 1$ (assume n is even)
- $p(x) = p_H(x)x^{n/2} + p_L(x)$,
- $p_H(x), p_L(x)$: polynomials of degree $n/2 - 1$.

$$\begin{aligned} pq &= \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right) \\ &= p_H q_H x^n + \left(p_H q_L + p_L q_H\right)x^{n/2} + p_L q_L \end{aligned}$$

Divide-and-Conquer for Polynomial Multiplication

$$\begin{aligned} pq &= \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right) \\ &= p_H q_H x^n + \left(p_H q_L + p_L q_H\right) x^{n/2} + p_L q_L \end{aligned}$$

Divide-and-Conquer for Polynomial Multiplication

$$
\begin{aligned}
pq &= \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right) \\
&= p_H q_H x^n + \left(p_H q_L + p_L q_H\right) x^{n/2} + p_L q_L
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{multiply}(p, q) = \; & \mathsf{multiply}(p_H, q_H) \times x^n \\
& + \left(\mathsf{multiply}(p_H, q_L) + \mathsf{multiply}(p_L, q_H)\right) \times x^{n/2} \\
& + \mathsf{multiply}(p_L, q_L)
\end{aligned}
$$

Divide-and-Conquer for Polynomial Multiplication

$$\begin{aligned} pq &= \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right) \\ &= p_H q_H x^n + \left(p_H q_L + p_L q_H\right) x^{n/2} + p_L q_L \end{aligned}$$

$$\begin{aligned} \text{multiply}(p, q) = {} & \text{multiply}(p_H, q_H) \times x^n \\ & + \left(\text{multiply}(p_H, q_L) + \text{multiply}(p_L, q_H)\right) \times x^{n/2} \\ & + \text{multiply}(p_L, q_L) \end{aligned}$$

- Recurrence: $T(n) = 4T(n/2) + O(n)$

Divide-and-Conquer for Polynomial Multiplication

$$\begin{aligned}
pq &= \big(p_H x^{n/2} + p_L\big)\big(q_H x^{n/2} + q_L\big) \\
&= p_H q_H x^n + \big(p_H q_L + p_L q_H\big) x^{n/2} + p_L q_L
\end{aligned}$$

$$\begin{aligned}
\mathsf{multiply}(p, q) = {} & \mathsf{multiply}(p_H, q_H) \times x^n \\
& + \big(\mathsf{multiply}(p_H, q_L) + \mathsf{multiply}(p_L, q_H)\big) \times x^{n/2} \\
& + \mathsf{multiply}(p_L, q_L)
\end{aligned}$$

- Recurrence: $T(n) = 4T(n/2) + O(n)$
- $T(n) = O(n^2)$

Reduce Number from 4 to 3

Reduce Number from 4 to 3

$$
\begin{aligned}
pq &= \big(p_H x^{n/2} + p_L\big)\big(q_H x^{n/2} + q_L\big) \\
&= p_H q_H x^n + \big(p_H q_L + p_L q_H\big) x^{n/2} + p_L q_L
\end{aligned}
$$

Reduce Number from 4 to 3

$$
\begin{aligned}
pq &= \big(p_H x^{n/2} + p_L\big)\big(q_H x^{n/2} + q_L\big) \\
&= p_H q_H x^n + \big(p_H q_L + p_L q_H\big) x^{n/2} + p_L q_L
\end{aligned}
$$

- $p_H q_L + p_L q_H = (p_H + p_L)(q_H + q_L) - p_H q_H - p_L q_L$

Divide-and-Conquer for Polynomial Multiplication

Divide-and-Conquer for Polynomial Multiplication

$$r_H = \mathsf{multiply}(p_H, q_H)$$
$$r_L = \mathsf{multiply}(p_L, q_L)$$

Divide-and-Conquer for Polynomial Multiplication

$$r_H = \mathsf{multiply}(p_H, q_H)$$
$$r_L = \mathsf{multiply}(p_L, q_L)$$

$$\begin{aligned}\mathsf{multiply}(p, q) = {} & r_H \times x^n \\ & + \big(\mathsf{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L\big) \times x^{n/2} \\ & + r_L\end{aligned}$$

Divide-and-Conquer for Polynomial Multiplication

$$
\begin{aligned}
r_H &= \mathsf{multiply}(p_H, q_H) \\
r_L &= \mathsf{multiply}(p_L, q_L)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{multiply}(p, q) = {} & r_H \times x^n \\
& + \big(\mathsf{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L\big) \times x^{n/2} \\
& + r_L
\end{aligned}
$$

- Solving Recurrence: $T(n) = 3T(n/2) + O(n)$

Divide-and-Conquer for Polynomial Multiplication

$$
\begin{aligned}
r_H &= \mathsf{multiply}(p_H, q_H) \\
r_L &= \mathsf{multiply}(p_L, q_L)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{multiply}(p, q) = \; & r_H \times x^n \\
& + \big(\mathsf{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L\big) \times x^{n/2} \\
& + r_L
\end{aligned}
$$

- Solving Recurrence: $T(n) = 3T(n/2) + O(n)$
- $T(n) = O(n^{\lg_2 3}) = O(n^{1.585})$

Assumption n is a power of 2. Arrays are 0-indexed.

multiply(A, B, n)

```
if n = 1 then return (A[0]B[0])
A_L ← A[0 .. n/2 − 1], A_H ← A[n/2 .. n − 1]
B_L ← B[0 .. n/2 − 1], B_H ← B[n/2 .. n − 1]
C_L ← multiply(A_L, B_L, n/2)
C_H ← multiply(A_H, B_H, n/2)
C_M ← multiply(A_L + A_H, B_L + B_H, n/2)
C ← array of (2n − 1) 0's
for i ← 0 to n − 2 do
    C[i] ← C[i] + C_L[i]
    C[i + n] ← C[i + n] + C_H[i]
    C[i + n/2] ← C[i + n/2] + C_M[i] − C_L[i] − C_H[i]
return C
```

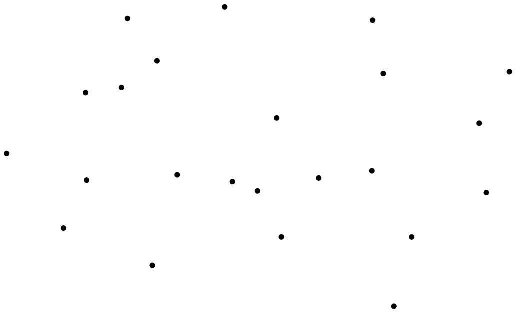
Outline

- Closest pair
- Convex hull
- Matrix multiplication
- FFT(Fast Fourier Transform): polynomial multiplication in $O(n \lg n)$ time

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

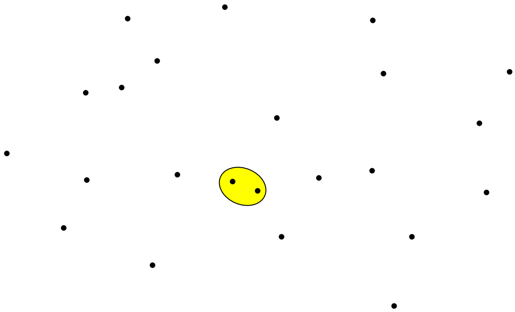
Output: the pair of points that are closest



Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

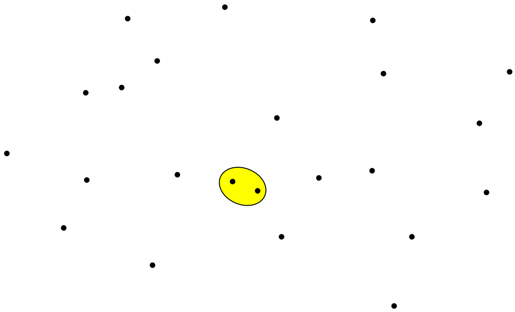
Output: the pair of points that are closest



Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

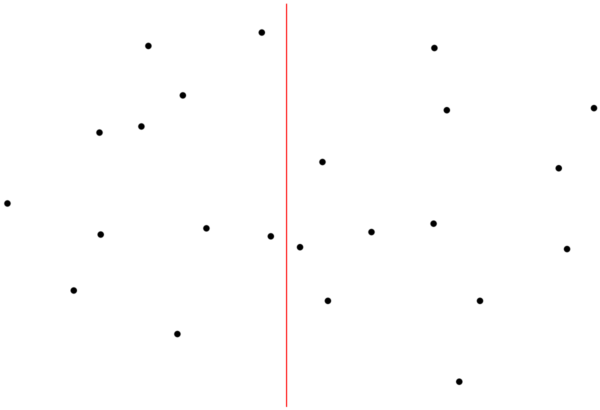
Output: the pair of points that are closest



- Trivial algorithm: $O(n^2)$ running time

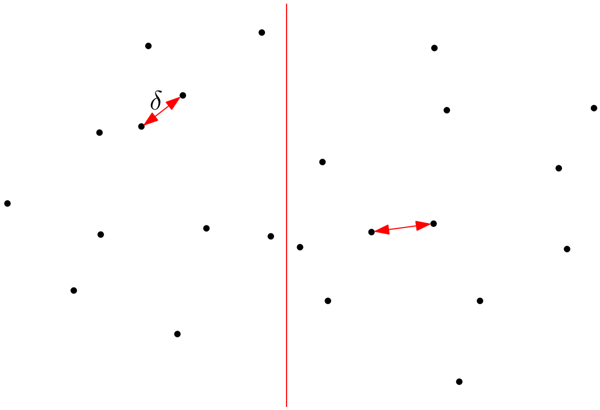
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line



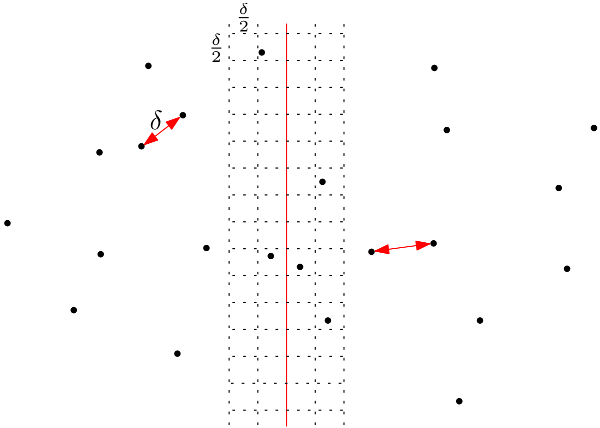
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively

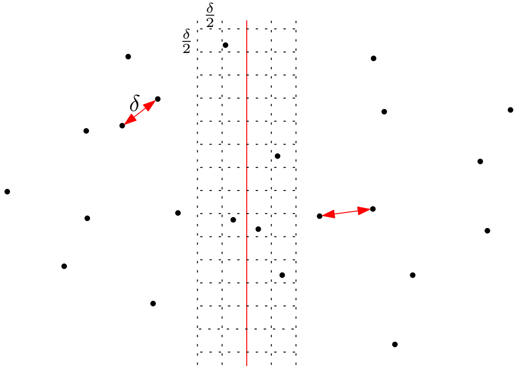


Divide-and-Conquer Algorithm for Closest Pair

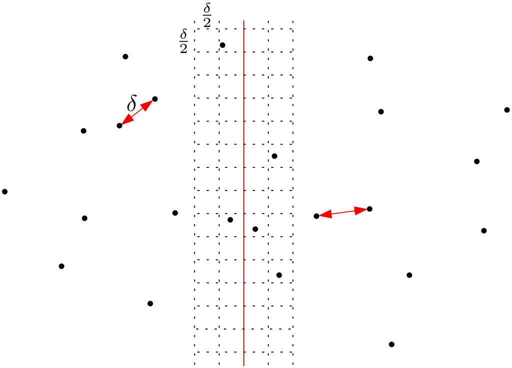
- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half



Divide-and-Conquer Algorithm for Closest Pair

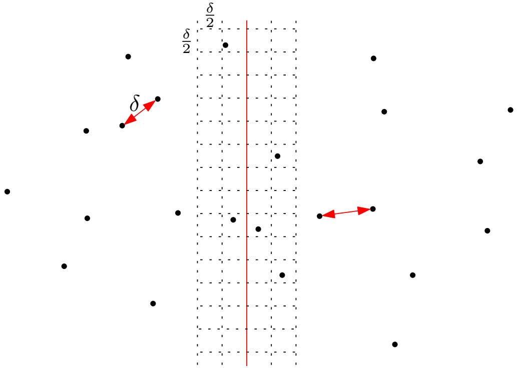


Divide-and-Conquer Algorithm for Closest Pair



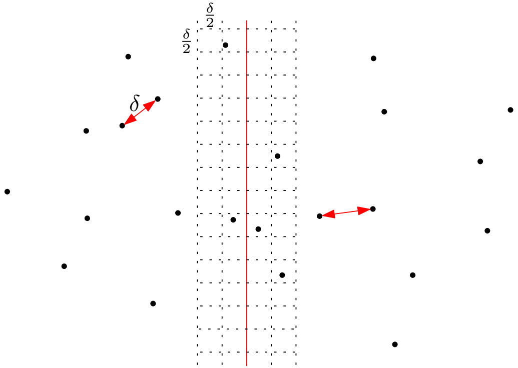
- Each box contains at most one pair

Divide-and-Conquer Algorithm for Closest Pair



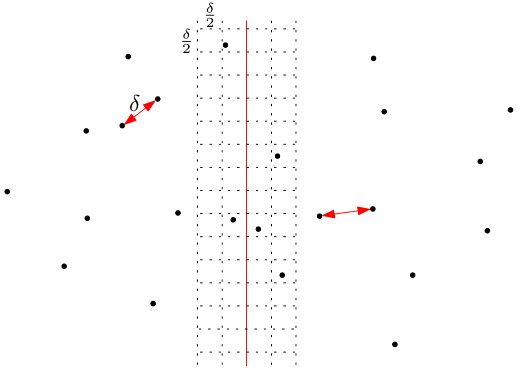
- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby

Divide-and-Conquer Algorithm for Closest Pair



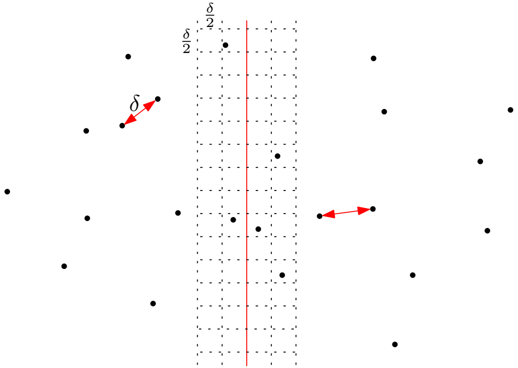
- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine $= O(n)$ (many technicalities omitted)

Divide-and-Conquer Algorithm for Closest Pair



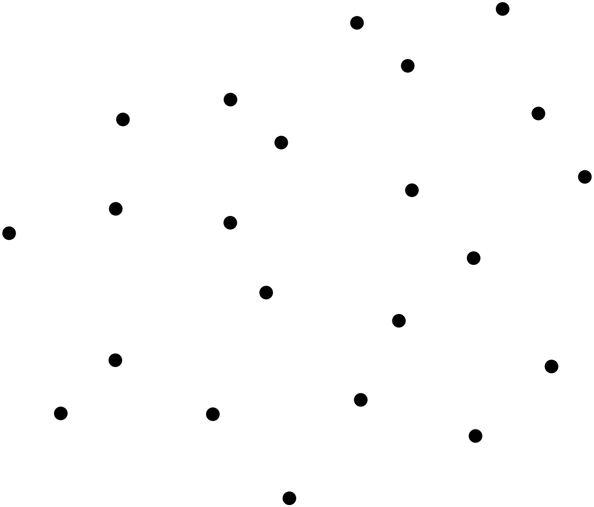
- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine $= O(n)$ (many technicalities omitted)
- Recurrence: $T(n) = 2T(n/2) + O(n)$

Divide-and-Conquer Algorithm for Closest Pair

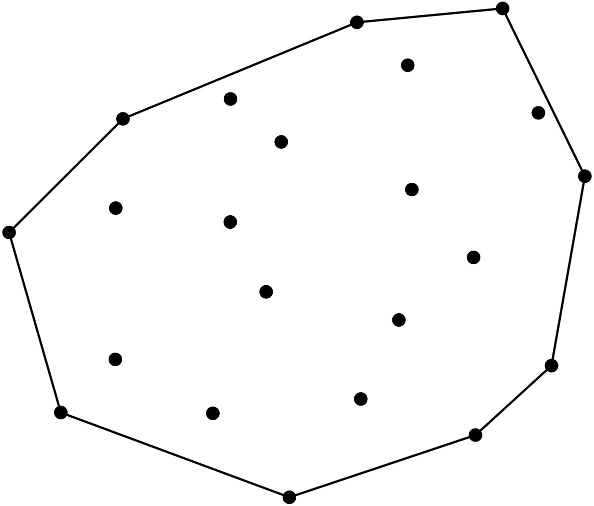


- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine $= O(n)$ (many technicalities omitted)
- Recurrence: $T(n) = 2T(n/2) + O(n)$
- Running time: $O(n \lg n)$

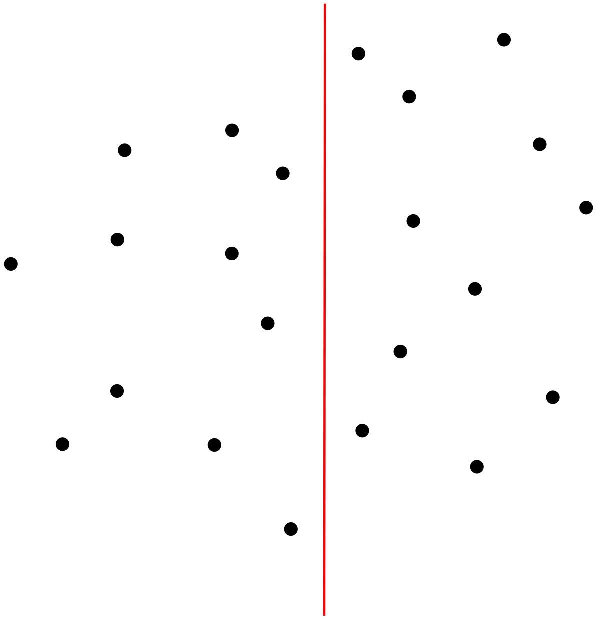
$O(n \lg n)$-Time Algorithm for Convex Hull



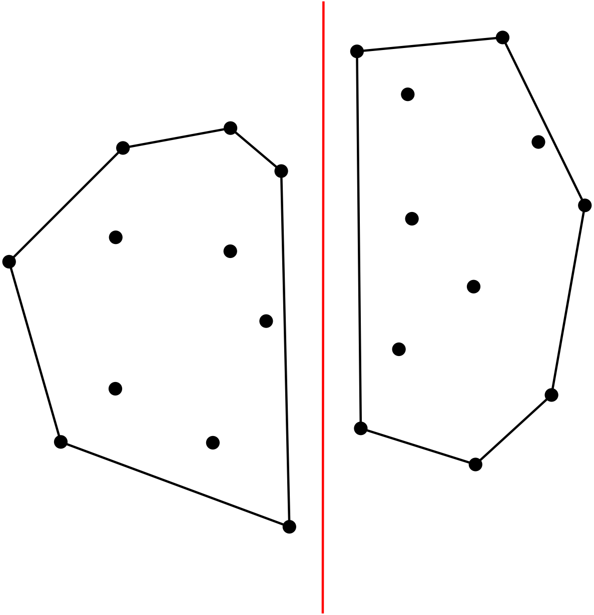
$O(n \lg n)$-Time Algorithm for Convex Hull



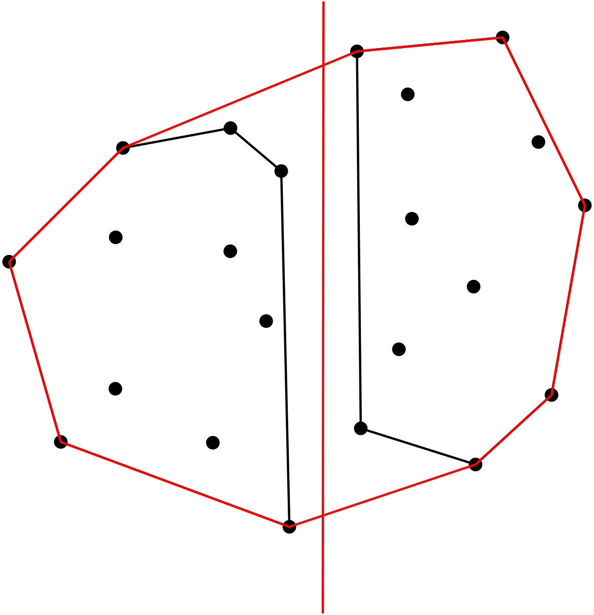
$O(n \lg n)$-Time Algorithm for Convex Hull



$O(n \lg n)$-Time Algorithm for Convex Hull



$O(n \lg n)$-Time Algorithm for Convex Hull



Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

Input: two $n \times n$ matrices A and B

Output: $C = AB$

Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

Input: two $n \times n$ matrices A and B

Output: $C = AB$

Naive Algorithm: matrix-multiplication(A, B, n)

for $i \leftarrow 1$ to n
    for $j \leftarrow 1$ to n
        $C[i, j] \leftarrow 0$
        for $k \leftarrow 1$ to n
            $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
return C

Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

Input: two $n \times n$ matrices A and B

Output: $C = AB$

Naive Algorithm: matrix-multiplication(A, B, n)

1. for $i \leftarrow 1$ to n
2.   for $j \leftarrow 1$ to n
3. $C[i,j] \leftarrow 0$
4.   for $k \leftarrow 1$ to n
5. $C[i,j] \leftarrow C[i,j] + A[i,k] \times B[k,j]$
6. return C

- running time $= O(n^3)$

Try to Use Divide-and-Conquer

$$A = \begin{array}{|c|c|}\hline A_{11} & \overbrace{A_{12}}^{n/2} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \quad \left.\vphantom{A_{12}}\right\} n/2 \qquad B = \begin{array}{|c|c|}\hline B_{11} & \overbrace{B_{12}}^{n/2} \\ \hline B_{21} & B_{22} \\ \hline \end{array} \quad \left.\vphantom{B_{12}}\right\} n/2$$

- $C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$
- matrix_multiplication(A, B) recursively calls
  matrix_multiplication(A_{11}, B_{11}),
  matrix_multiplication(A_{12}, B_{21}),
  $\cdots$

Try to Use Divide-and-Conquer

$$A = \begin{array}{|c|c|}\hline A_{11} & \overbrace{A_{12}}^{n/2} \\ \hline A_{21} & A_{22} \\ \hline \end{array}\; \} n/2 \qquad B = \begin{array}{|c|c|}\hline B_{11} & \overbrace{B_{12}}^{n/2} \\ \hline B_{21} & B_{22} \\ \hline \end{array}\; \} n/2$$

- $C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$
- matrix_multiplication(A, B) recursively calls
  matrix_multiplication(A_{11}, B_{11}),
  matrix_multiplication(A_{12}, B_{21}),
  $\cdots$
- Recurrence for running time: $T(n) = 8T(n/2) + O(n^2)$
- $T(n) = O(n^3)$

Strassen's Algorithm

- $T(n) = 8T(n/2) + O(n^2)$
- Strassen's Algorithm: improve the number of multiplications from 8 to 7!
- New recurrence: $T(n) = 7T(n/2) + O(n^2)$

Strassen's Algorithm

- $T(n) = 8T(n/2) + O(n^2)$
- Strassen's Algorithm: improve the number of multiplications from 8 to 7!
- New recurrence: $T(n) = 7T(n/2) + O(n^2)$
- Solving Recurrence $T(n) = O(n^{\log_2 7}) = O(n^{2.808})$

Outline

Methods for Solving Recurrences

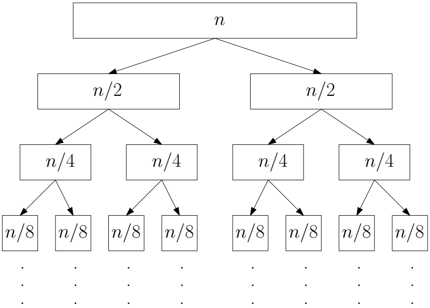
- The recursion-tree method
- The master theorem

Recursion-Tree Method

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

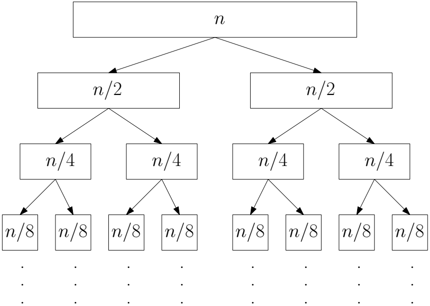
Recursion-Tree Method

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



Recursion-Tree Method

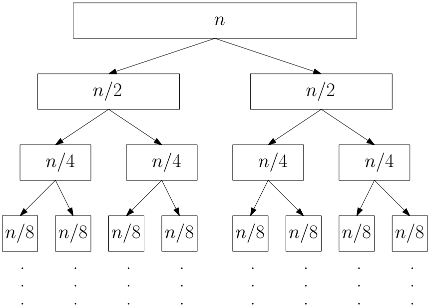
- $T(n) = 2T(n/2) + O(n)$



- Each level takes running time $O(n)$

Recursion-Tree Method

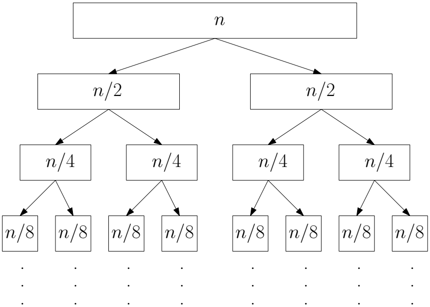
- $T(n) = 2T(n/2) + O(n)$



- Each level takes running time $O(n)$
- There are $O(\lg n)$ levels

Recursion-Tree Method

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



- Each level takes running time $O(n)$
- There are $O(\lg n)$ levels
- Running time $= O(n \lg n)$

Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n)$

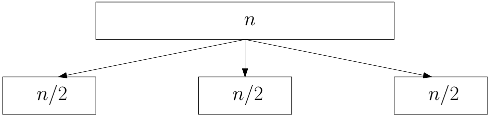
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n)$

$$\boxed{n}$$

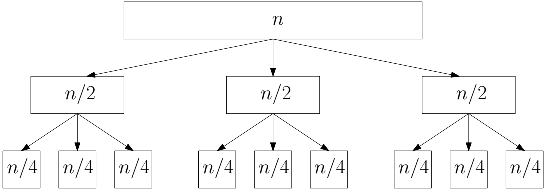
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n)$



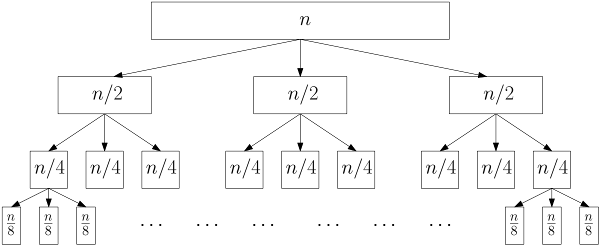
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n)$



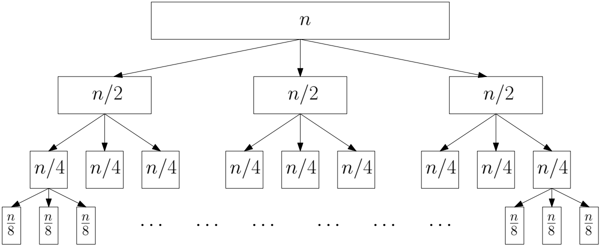
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n)$



Recursion-Tree Method

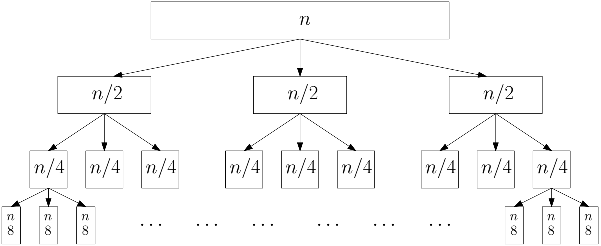
- $T(n) = 3T(n/2) + O(n)$



- Total running time at level i?

Recursion-Tree Method

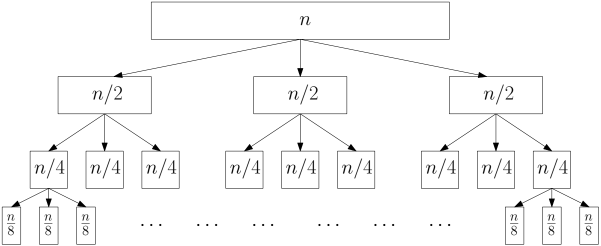
- $T(n) = 3T(n/2) + O(n)$



- Total running time at level i? $\frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n$

Recursion-Tree Method

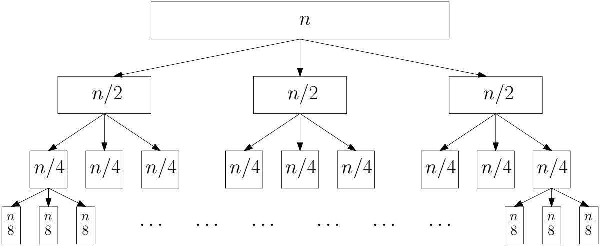
- $T(n) = 3T(n/2) + O(n)$



- Total running time at level i? $\frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n$
- Index of last level?

Recursion-Tree Method

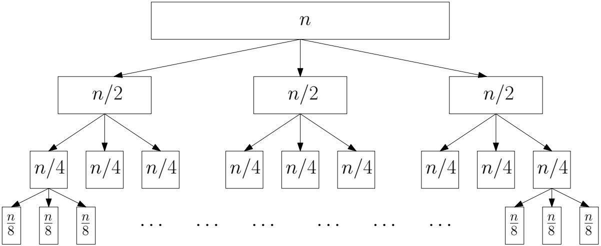
- $T(n) = 3T(n/2) + O(n)$



- Total running time at level i? $\frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n$
- Index of last level? $\lg_2 n$

Recursion-Tree Method

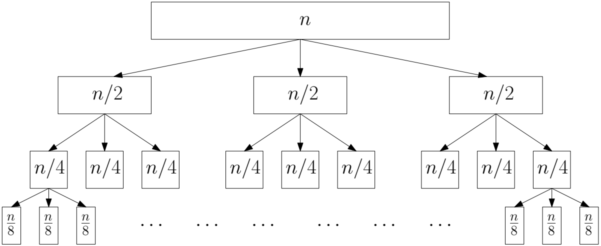
- $T(n) = 3T(n/2) + O(n)$



- Total running time at level i? $\frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n$
- Index of last level? $\lg_2 n$
- Total running time?

Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n)$



- Total running time at level i? $\frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n$
- Index of last level? $\lg_2 n$
- Total running time?

$$\sum_{i=0}^{\lg_2 n} \left(\frac{3}{2}\right)^i n = O\left(n\left(\frac{3}{2}\right)^{\lg_2 n}\right) = O(3^{\lg_2 n}) = O(n^{\lg_2 3}).$$

Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$

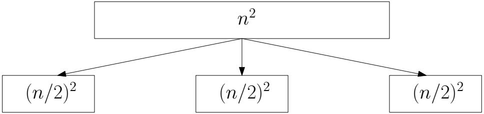
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$

$$\boxed{n^2}$$

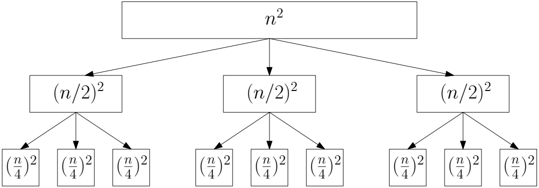
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$



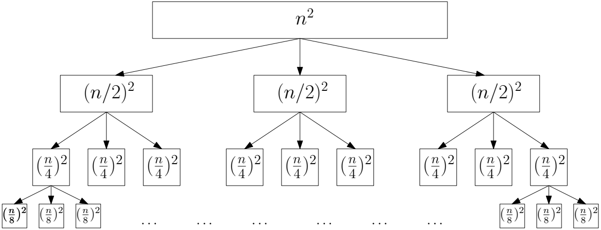
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$



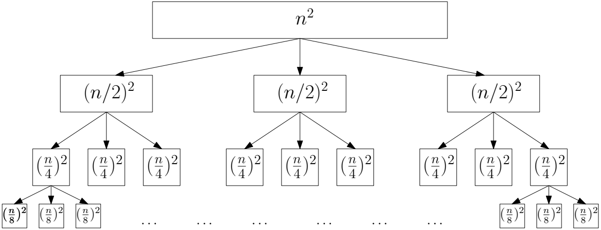
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$



Recursion-Tree Method

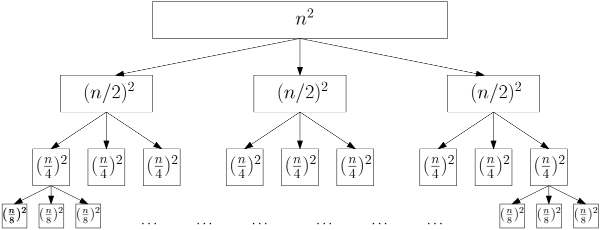
- $T(n) = 3T(n/2) + O(n^2)$



- Total running time at level i?

Recursion-Tree Method

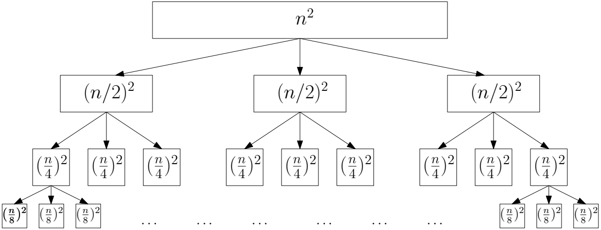
- $T(n) = 3T(n/2) + O(n^2)$



- Total running time at level i? $\left(\frac{n}{2^i}\right)^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2$

Recursion-Tree Method

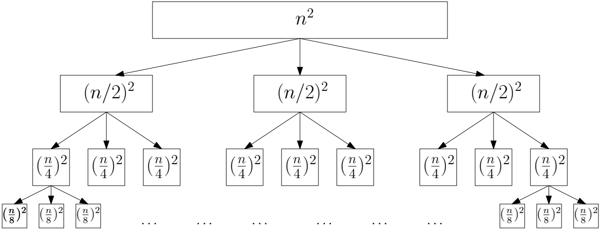
- $T(n) = 3T(n/2) + O(n^2)$



- Total running time at level i? $\left(\frac{n}{2^i}\right)^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2$
- Index of last level?

Recursion-Tree Method

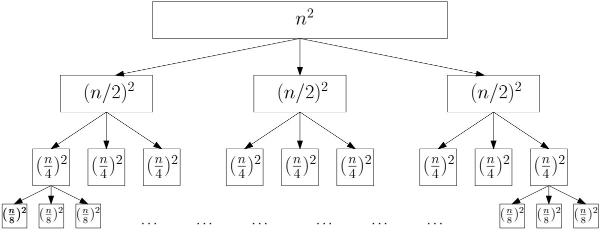
- $T(n) = 3T(n/2) + O(n^2)$



- Total running time at level i? $\left(\frac{n}{2^i}\right)^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2$
- Index of last level? $\lg_2 n$

Recursion-Tree Method

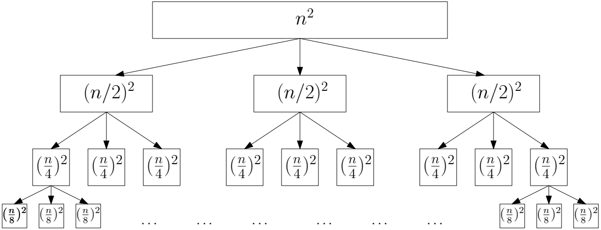
- $T(n) = 3T(n/2) + O(n^2)$



- Total running time at level i? $\left(\frac{n}{2^i}\right)^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2$
- Index of last level? $\lg_2 n$
- Total running time?

Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$

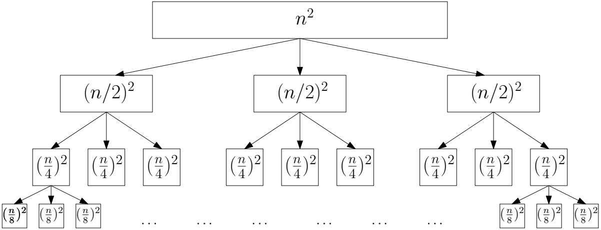


- Total running time at level i? $\left(\frac{n}{2^i}\right)^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2$
- Index of last level? $\lg_2 n$
- Total running time?

$$\sum_{i=0}^{\lg_2 n} \left(\frac{3}{4}\right)^i n^2 =$$

Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$



- Total running time at level i? $\left(\frac{n}{2^i}\right)^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2$
- Index of last level? $\lg_2 n$
- Total running time?

$$\sum_{i=0}^{\lg_2 n} \left(\frac{3}{4}\right)^i n^2 = O(n^2).$$

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$				$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$				$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$				$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$				$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$				$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$				$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} & \text{if } c < \lg_b a \\ & \text{if } c = \lg_b a \\ & \text{if } c > \lg_b a \end{cases}$$

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} ?? & \text{if } c < \lg_b a \\ & \text{if } c = \lg_b a \\ & \text{if } c > \lg_b a \end{cases}$$

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ & \text{if } c = \lg_b a \\ & \text{if } c > \lg_b a \end{cases}$$

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ & \text{if } c = \lg_b a \\ ?? & \text{if } c > \lg_b a \end{cases}$$

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ ?? & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

Master Theorem

Recurrences	a	b	c	time
$T(n) = 2T(n/2) + O(n)$	2	2	1	$O(n \lg n)$
$T(n) = 3T(n/2) + O(n)$	3	2	1	$O(n^{\lg_2 3})$
$T(n) = 3T(n/2) + O(n^2)$	3	2	2	$O(n^2)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Which Case?

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2.

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Which Case?

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1.

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1. $T(n) = O(n^{\lg_2 3})$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1. $T(n) = O(n^{\lg_2 3})$
- Ex: $T(n) = T(n/2) + O(1)$. Which Case?

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1. $T(n) = O(n^{\lg_2 3})$
- Ex: $T(n) = T(n/2) + O(1)$. Case 2.

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1. $T(n) = O(n^{\lg_2 3})$
- Ex: $T(n) = T(n/2) + O(1)$. Case 2. $T(n) = O(\lg n)$

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1. $T(n) = O(n^{\lg_2 3})$
- Ex: $T(n) = T(n/2) + O(1)$. Case 2. $T(n) = O(\lg n)$
- Ex: $T(n) = 2T(n/2) + O(n^2)$. Which Case?

Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1. $T(n) = O(n^{\lg_2 3})$
- Ex: $T(n) = T(n/2) + O(1)$. Case 2. $T(n) = O(\lg n)$
- Ex: $T(n) = 2T(n/2) + O(n^2)$. Case 3.

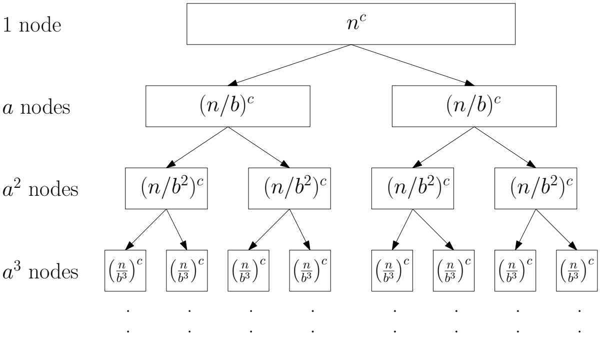
Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} O(n^{\lg_b a}) & \text{if } c < \lg_b a \\ O(n^c \lg n) & \text{if } c = \lg_b a \\ O(n^c) & \text{if } c > \lg_b a \end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \lg n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1. $T(n) = O(n^{\lg_2 3})$
- Ex: $T(n) = T(n/2) + O(1)$. Case 2. $T(n) = O(\lg n)$
- Ex: $T(n) = 2T(n/2) + O(n^2)$. Case 3. $T(n) = O(n^2)$

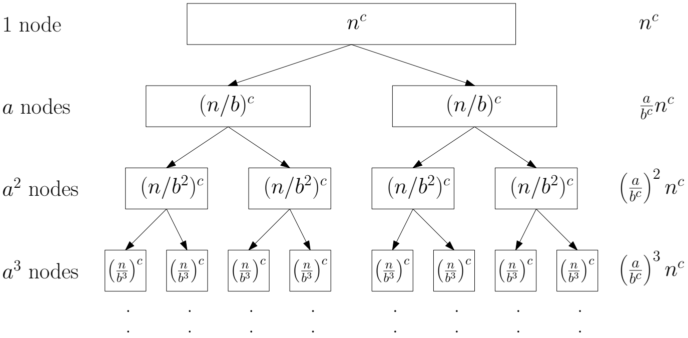
Proof of Master Theorem Using Recursion Tree

$$T(n) = aT(n/b) + O(n^c)$$



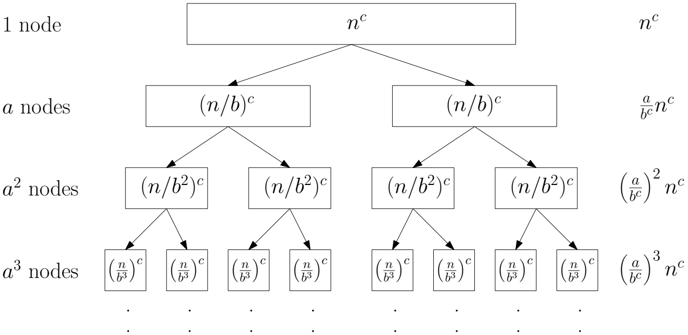
Proof of Master Theorem Using Recursion Tree

$$T(n) = aT(n/b) + O(n^c)$$



Proof of Master Theorem Using Recursion Tree

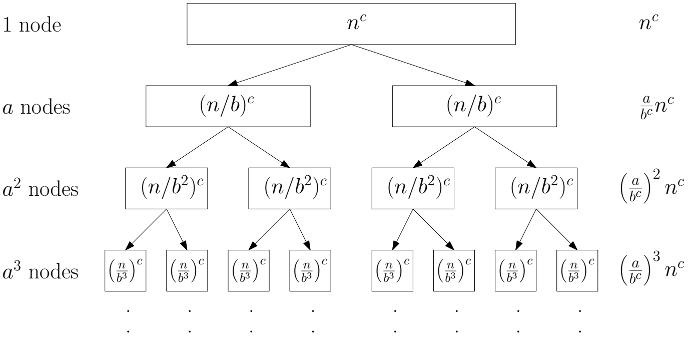
$$T(n) = aT(n/b) + O(n^c)$$



- $c < \lg_b a$: bottom-level dominates: $\left(\frac{a}{b^c}\right)^{\lg_b n} n^c = n^{\lg_b a}$

Proof of Master Theorem Using Recursion Tree

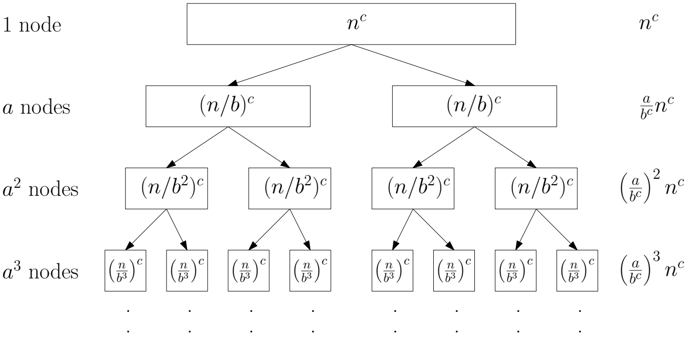
$$T(n) = aT(n/b) + O(n^c)$$



- $c < \lg_b a$: bottom-level dominates: $\left(\frac{a}{b^c}\right)^{\lg_b n} n^c = n^{\lg_b a}$
- $c = \lg_b a$: all levels have same time: $n^c \lg_b n = O(n^c \lg n)$

Proof of Master Theorem Using Recursion Tree

$$T(n) = aT(n/b) + O(n^c)$$



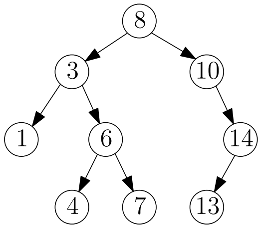
- $c < \lg_b a$: bottom-level dominates: $\left(\frac{a}{b^c}\right)^{\lg_b n} n^c = n^{\lg_b a}$
- $c = \lg_b a$: all levels have same time: $n^c \lg_b n = O(n^c \lg n)$
- $c > \lg_b a$: top-level dominates: $O(n^c)$

Outline

Binary Search Tree (BST)

- Elements are organized in a binary-tree structure
- Each element (node) is associated with a key value

- if node u is in the left sub-tree of node v, then $u.key \leq v.key$
- if node u is the right sub-tree of node v, then $u.key \geq v.key$

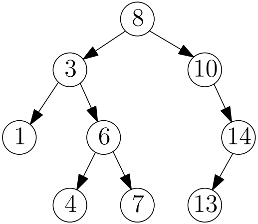


BST: numbers denote keys

Binary Search Tree (BST)

- Elements are organized in a binary-tree structure
- Each element (node) is associated with a key value

- if node u is in the left sub-tree of node v, then $u.key \leq v.key$
- if node u is the right sub-tree of node v, then $u.key \geq v.key$
- in-order traversal of tree gives a sorted list of keys



BST: numbers denote keys

Operations on Binary Search Tree T

- **insert**: insert an element to T

Operations on Binary Search Tree T

- **insert**: insert an element to T
- **delete**: delete an element from T

Operations on Binary Search Tree T

- **insert**: insert an element to T
- **delete**: delete an element from T
- **count-less-than**: return the number of elements in T with key values smaller than a given value

Operations on Binary Search Tree T

- **insert**: insert an element to T
- **delete**: delete an element from T
- **count-less-than**: return the number of elements in T with key values smaller than a given value
- check existence, return element with i-th smallest key value, $\ldots$

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. $\quad c \leftarrow c + T.$count-less-than$(A[i])$
5. $\quad T.$insert$(A[i])$
6. return c

running time $=$
$n \times$ (time for count $+$ time for insert)

tree elements

15	3	16	12	32	7

count-inversions(A, n)

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. $c \leftarrow c + T$.count-less-than$(A[i])$
5. T.insert$(A[i])$
6. return c

running time $=$
$n \times$ (time for count $+$ time for insert)

Counting Inversions Via Binary Search Tree (BST)

tree elements

15	3	16	12	32	7

i

count-less-than(7) $= 0$

count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

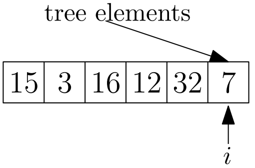
running time $=$
$n \times$ (time for count $+$ time for insert)

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

running time $=$
$n \times$ (time for count $+$ time for insert)



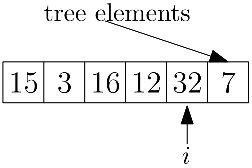
count-less-than(7) $= 0$
insert(7)

Counting Inversions Via Binary Search Tree (BST)

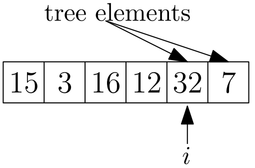
count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

running time $=$
$n \times$ (time for count $+$ time for insert)



count-less-than$(\ 7) = 0$
insert(7)
count-less-than$(32) = 1$



count-less-than(7) = 0
insert(7)
count-less-than(32) = 1
insert(32)

count-inversions(A, n)

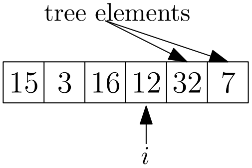
```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

running time $=$
$n \times$ (time for count $+$ time for insert)

count-inversions(A, n)

$T \leftarrow$ empty BST
$c \leftarrow 0$
for $i \leftarrow n$ downto 1
    $c \leftarrow c + T.\text{count-less-than}(A[i])$
    $T.\text{insert}(A[i])$
return c

running time $=$
$n \times$ (time for count $+$ time for insert)

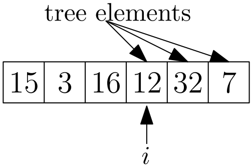


count-less-than(7) $= 0$
insert(7)
count-less-than(32) $= 1$
insert(32)
count-less-than(12) $= 1$

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```



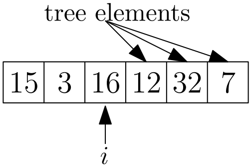
count-less-than(7) $= 0$
insert(7)
count-less-than(32) $= 1$
insert(32)
count-less-than(12) $= 1$
insert(12)

running time $=$
$n \times$ (time for count $+$ time for insert)

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```



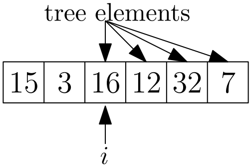
count-less-than$(7) = 0$
insert(7)
count-less-than$(32) = 1$
insert(32)
count-less-than$(12) = 1$
insert(12)
count-less-than$(16) = 2$

running time $=$
$n \times$ (time for count $+$ time for insert)

count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

running time $=$
$n \times$ (time for count $+$ time for insert)



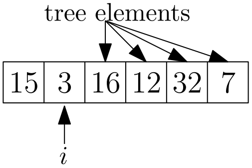
count-less-than$(7) = 0$
insert(7)
count-less-than$(32) = 1$
insert(32)
count-less-than$(12) = 1$
insert(12)
count-less-than$(16) = 2$
insert(16)

Counting Inversions Via Binary Search Tree (BST)

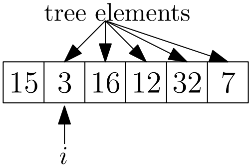
count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

running time $=$
$n \times$ (time for count $+$ time for insert)



count-less-than(7) $= 0$
insert(7)
count-less-than(32) $= 1$
insert(32)
count-less-than(12) $= 1$
insert(12)
count-less-than(16) $= 2$
insert(16)
count-less-than(3) $= 0$



count-inversions(A, n)

$T \leftarrow$ empty BST
$c \leftarrow 0$
for $i \leftarrow n$ downto 1
    $c \leftarrow c + T$.count-less-than$(A[i])$
    T.insert$(A[i])$
return c

running time $=$
$n \times$ (time for count $+$ time for insert)

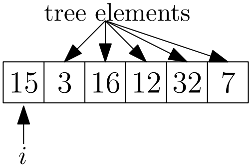
count-less-than$(\ 7) = 0$
insert(7)
count-less-than$(32) = 1$
insert(32)
count-less-than$(12) = 1$
insert(12)
count-less-than$(16) = 2$
insert(16)
count-less-than$(\ 3) = 0$
insert(3)

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

```
T ← empty BST
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

running time $=$
$n \times$ (time for count $+$ time for insert)



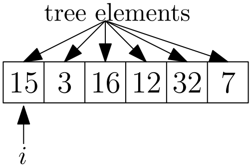
count-less-than(7) $= 0$
insert(7)
count-less-than(32) $= 1$
insert(32)
count-less-than(12) $= 1$
insert(12)
count-less-than(16) $= 2$
insert(16)
count-less-than(3) $= 0$
insert(3)
count-less-than(15) $= 3$

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

$T \leftarrow$ empty BST
$c \leftarrow 0$
for $i \leftarrow n$ downto 1
  $c \leftarrow c + T$.count-less-than$(A[i])$
  T.insert$(A[i])$
return c

running time $=$
$n \times$ (time for count $+$ time for insert)



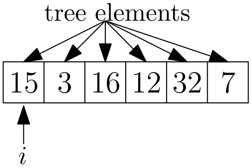
count-less-than$(7) = 0$
insert(7)
count-less-than$(32) = 1$
insert(32)
count-less-than$(12) = 1$
insert(12)
count-less-than$(16) = 2$
insert(16)
count-less-than$(3) = 0$
insert(3)
count-less-than$(15) = 3$
insert(15)

Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

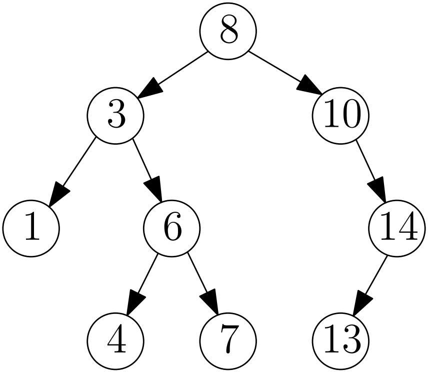
$T \leftarrow$ empty BST
$c \leftarrow 0$
for $i \leftarrow n$ downto 1
  $c \leftarrow c + T.$count-less-than$(A[i])$
  $T.$insert$(A[i])$
return c

running time $=$
$n \times$ (time for count $+$ time for insert)



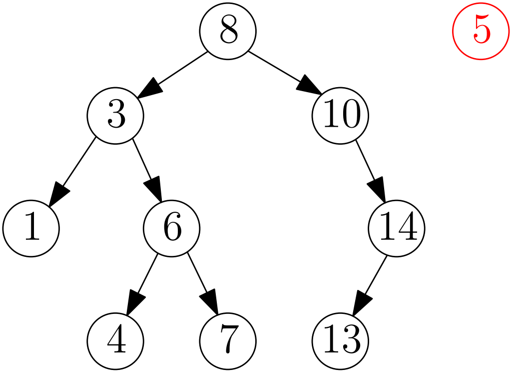
count-less-than(7) $= 0$
insert(7)
count-less-than(32) $= 1$
insert(32)
count-less-than(12) $= 1$
insert(12)
count-less-than(16) $= 2$
insert(16)
count-less-than(3) $= 0$
insert(3)
count-less-than(15) $= 3$
insert(15)
$c = 0 + 1 + 1 + 2 + 0 + 3 = 7$

Binary Search Tree: Insertion

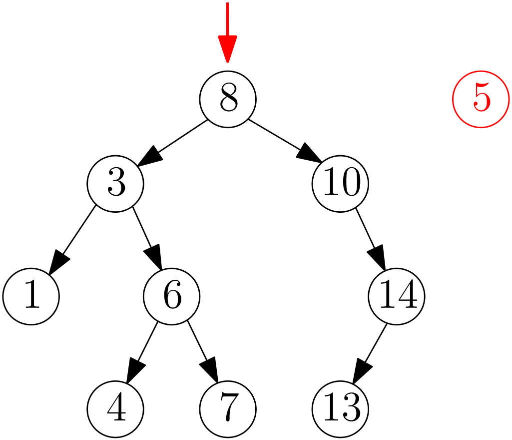


BST: numbers denote keys

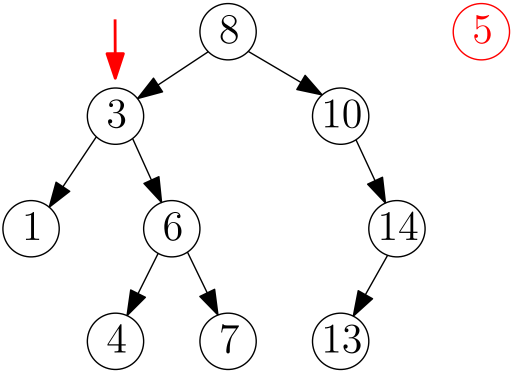
Binary Search Tree: Insertion



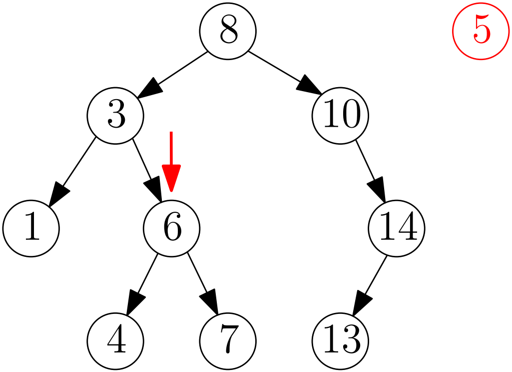
Binary Search Tree: Insertion



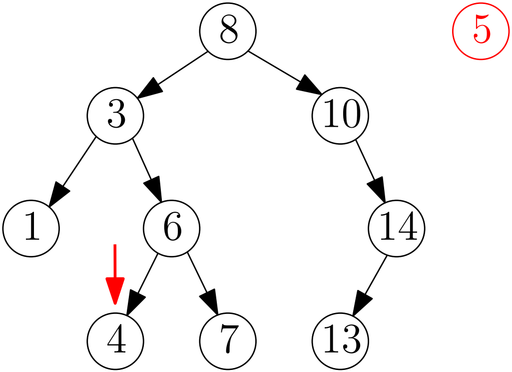
Binary Search Tree: Insertion



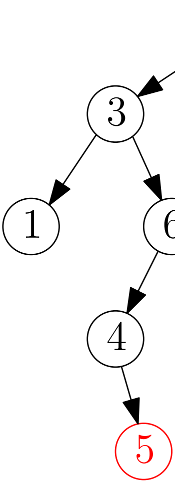
Binary Search Tree: Insertion



Binary Search Tree: Insertion



Binary Search Tree: Insertion



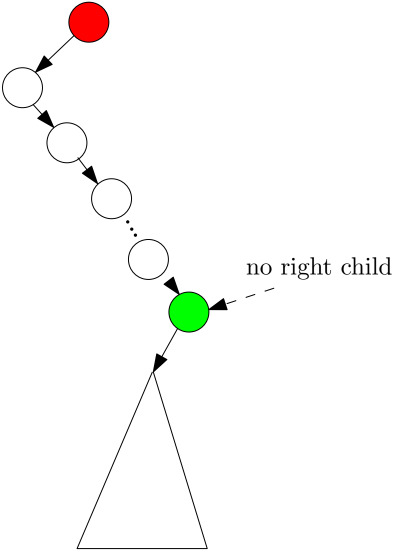
recursive-insert(v, key)

```
if v = nil then
    u ← new node with u.left = u.right = nil
    u.key ← key
    return u
if key < v.key then
    v.left ← recursive-insert(v.left, key)
else
    v.right ← recursive-insert(v.right, key)
return v
```

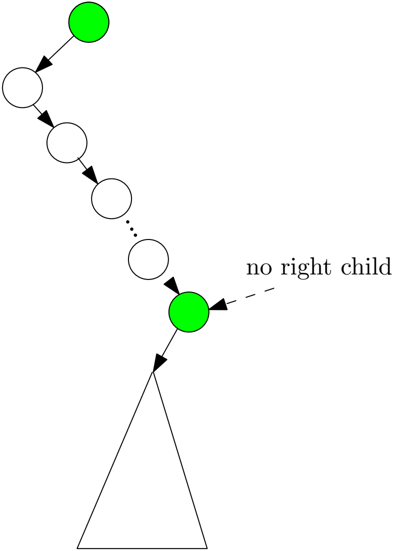
insert(key)

```
root ← recursive-insert(root, key)
```

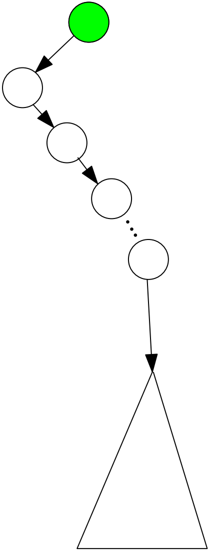
Binary Search Tree: Deletition



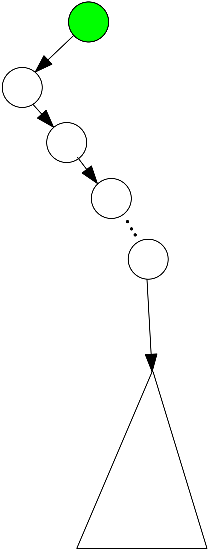
Binary Search Tree: Deletition

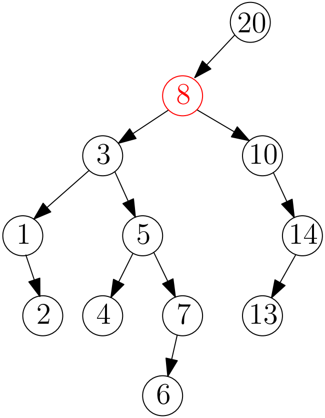


Binary Search Tree: Deletition

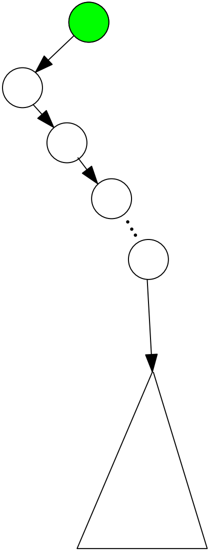


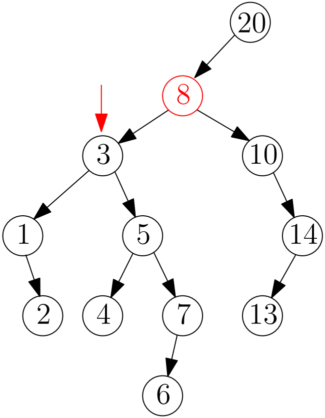
Binary Search Tree: Deletition



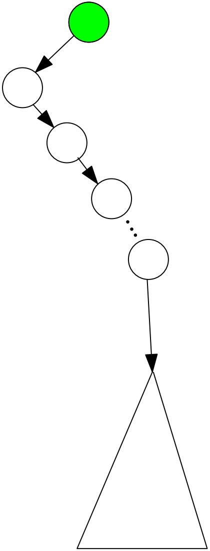


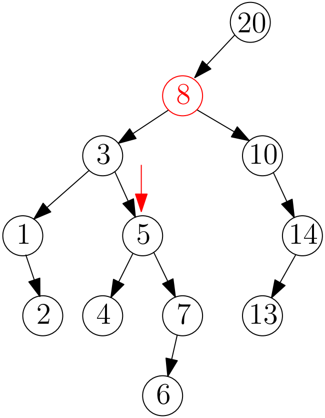
Binary Search Tree: Deletition



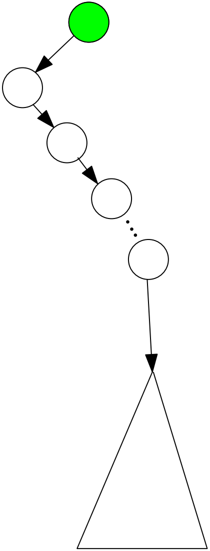


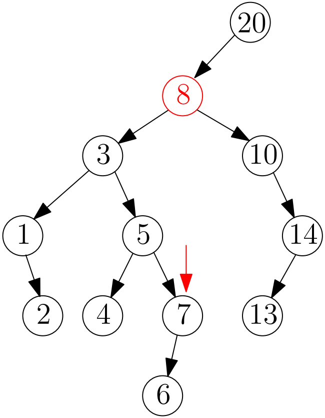
Binary Search Tree: Deletition



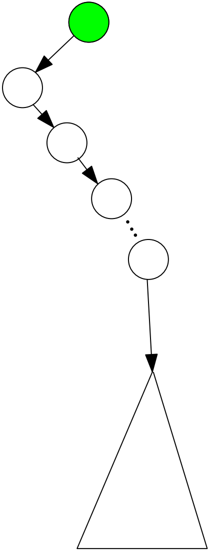


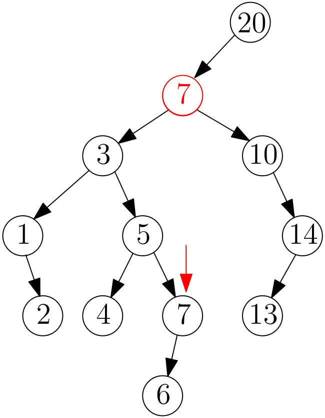
Binary Search Tree: Deletition

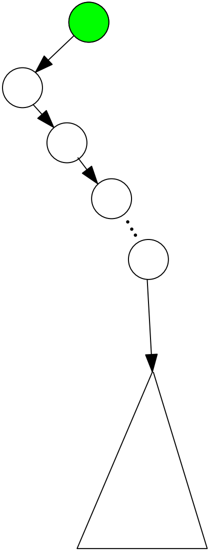


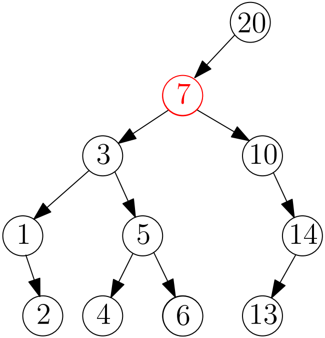


Binary Search Tree: Deletition








20
7
3
10
1
5
14
2
4
6
13

recursive-delete(v)

1. if $v.right =$ nil then return $(v.left, v)$
2. $(v.right, del) \leftarrow$ recursive-delete($v.right$)
3. return (v, del)

- recursive-delete(v) deletes the element in the sub-tree rooted at v with the largest key value

recursive-delete(v)

1. if $v.right =$ nil then return $(v.left, v)$
2. $(v.right, del) \leftarrow$ recursive-delete($v.right$)
3. return (v, del)

- recursive-delete(v) deletes the element in the sub-tree rooted at v with the largest key value
- returns: the new root and the deleted node

recursive-delete(v)

1. if $v.right =$ nil then return $(v.left, v)$
2. $(v.right, del) \leftarrow$ recursive-delete$(v.right)$
3. return (v, del)

- recursive-delete(v) deletes the element in the sub-tree rooted at v with the largest key value
- returns: the new root and the deleted node

delete(v) \\ returns the new root after deletion

1. if $v.left =$ nil then return $v.right$
2. $(r, del) \leftarrow$ recursive-delete$(v.left)$
3. $r.key \leftarrow del.key$
4. return r

recursive-delete(v)

1. if $v.right =$ nil then return $(v.left, v)$
2. $(v.right, del) \leftarrow$ recursive-delete$(v.right)$
3. return (v, del)

delete(v) $\backslash\backslash$ returns the new root after deletion

1. if $v.left =$ nil then return $v.right$
2. $(r, del) \leftarrow$ recursive-delete$(v.left)$
3. $r.key \leftarrow del.key$
4. return r

recursive-delete(v)

1. if $v.right =$ nil then return $(v.left, v)$
2. $(v.right, del) \leftarrow$ recursive-delete($v.right$)
3. return (v, del)

delete(v) \\ returns the new root after deletion

1. if $v.left =$ nil then return $v.right$
2. $(r, del) \leftarrow$ recursive-delete($v.left$)
3. $r.key \leftarrow del.key$
4. return r

- to remove left-child of v: call $v.left \leftarrow delete(v.left)$
- to remove right-child of v: call $v.right \leftarrow delete(v.right)$
- to remove root: call $root \leftarrow delete(root)$

Binary Search Tree: count-less-than

Binary Search Tree: count-less-than

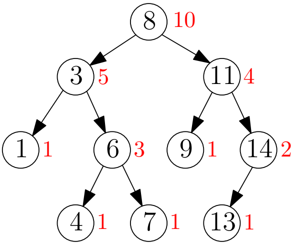
- Need to maintain a “size” property for each node

Binary Search Tree: count-less-than

- Need to maintain a "size" property for each node
- $v.size =$ number of nodes in the tree rooted at v

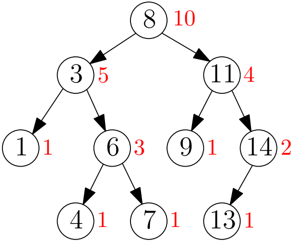
Binary Search Tree: count-less-than

- Need to maintain a "size" property for each node
- $v.size$ = number of nodes in the tree rooted at v



Binary Search Tree: count-less-than

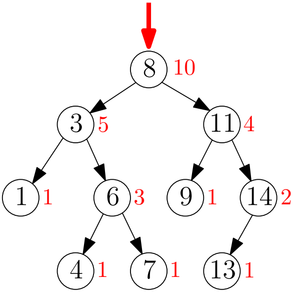
- Need to maintain a "size" property for each node
- $v.size$ = number of nodes in the tree rooted at v



$\#\,(\text{elements} < 10) =$

Binary Search Tree: count-less-than

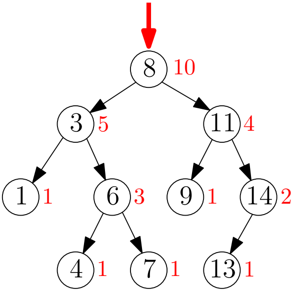
- Need to maintain a "size" property for each node
- $v.size$ = number of nodes in the tree rooted at v



$\#\ (\text{elements} < 10) =$

Binary Search Tree: count-less-than

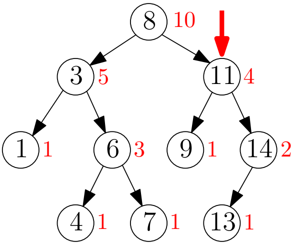
- Need to maintain a “size” property for each node
- $v.size$ = number of nodes in the tree rooted at v



$\#\ (\text{elements} < 10) = (5{+}1)$

Binary Search Tree: count-less-than

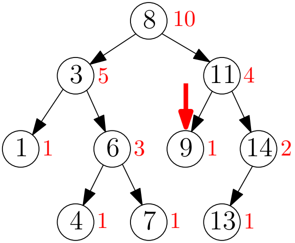
- Need to maintain a “size” property for each node
- $v.size$ = number of nodes in the tree rooted at v



$\#$ (elements < 10) $= (5+1)$

Binary Search Tree: count-less-than

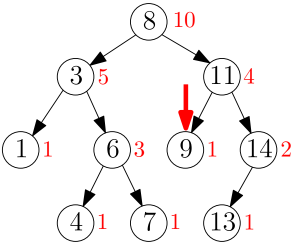
- Need to maintain a "size" property for each node
- $v.size$ = number of nodes in the tree rooted at v



$\#$ (elements < 10) $= (5+1)$

Binary Search Tree: count-less-than

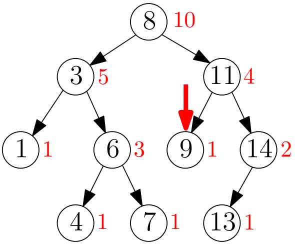
- Need to maintain a "size" property for each node
- $v.size$ = number of nodes in the tree rooted at v



$\#\ (\text{elements} < 10) = (5{+}1) + 1$

Binary Search Tree: count-less-than

- Need to maintain a "size" property for each node
- $v.size$ = number of nodes in the tree rooted at v



$\#$ (elements $<$ 10) $= (5{+}1) + 1 = 7$

- Trick: "nil" is a node with size 0.

recursive-count$(v, value)$

```
if v = nil then return 0
if value ≤ v.key
    return recursive-count(v.left, key)
else
    return v.left.size + 1 + recursive-count(v.right, key)
```

count-less-than$(value)$

```
return recursive-count(root, value)
```

Running Time for Each Operation

- Each operation takes time $O(h)$.
- $h =$ height of tree
- $n =$ number of nodes in tree

Running Time for Each Operation

- Each operation takes time $O(h)$.
- $h =$ height of tree
- $n =$ number of nodes in tree

Q: What is the height of the tree in the best scenario?

Running Time for Each Operation

- Each operation takes time $O(h)$.
- $h =$ height of tree
- $n =$ number of nodes in tree

Q: What is the height of the tree in the best scenario?

A: $O(\lg n)$

Running Time for Each Operation

- Each operation takes time $O(h)$.
- $h =$ height of tree
- $n =$ number of nodes in tree

Q: What is the height of the tree in the best scenario?

A: $O(\lg n)$

Q: What is the height of the tree in the worst scenario?

Running Time for Each Operation

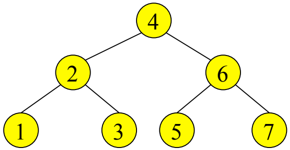
- Each operation takes time $O(h)$.
- $h =$ height of tree
- $n =$ number of nodes in tree

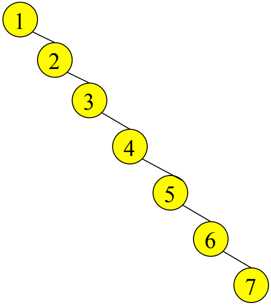
Q: What is the height of the tree in the best scenario?

A: $O(\lg n)$

Q: What is the height of the tree in the worst scenario?

A: $O(n)$


4
2
6
1
3
5
7


1
2
3
4
5
6
7

Def. A self-balancing BST is a BST that automatically keeps its height small

Def. A self-balancing BST is a BST that automatically keeps its height small

- AVL tree
- red-black tree
- Splay tree
- Treap
- ...

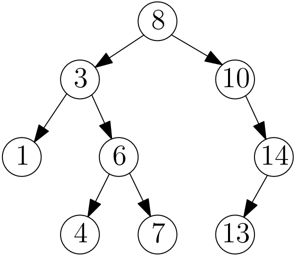
Def. A self-balancing BST is a BST that automatically keeps its height small

- AVL tree
- red-black tree
- Splay tree
- Treap
- ...

AVL Tree

An AVL Tree Is Balanced

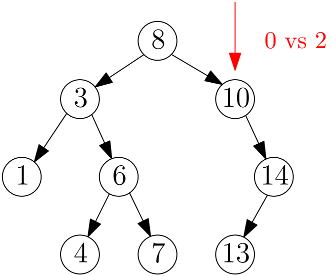
Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.



AVL Tree

An AVL Tree Is Balanced

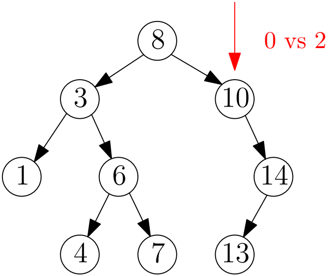
Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.



AVL Tree

An AVL Tree Is Balanced

Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.

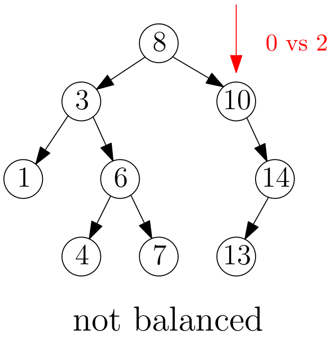


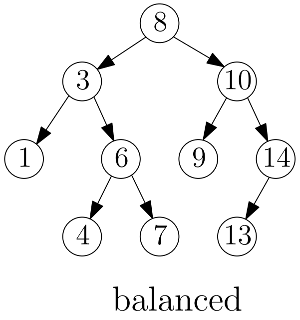
not balanced

AVL Tree

An AVL Tree Is Balanced

Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.





An AVL Tree Is Balanced

Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.

Lemma Property guarantees height $= O(\log n)$.

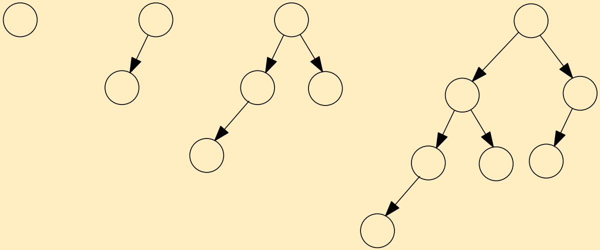
- $f(h)$: minimum size of a balanced tree of height h

An AVL Tree Is Balanced

Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.

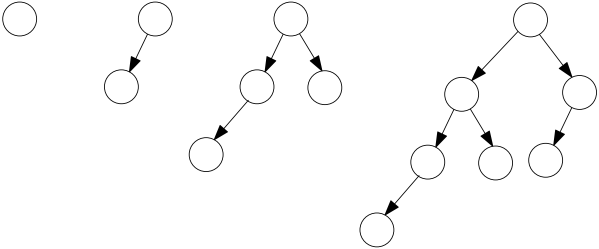
Lemma Property guarantees height $= O(\log n)$.

- $f(h)$: minimum size of a balanced tree of height h



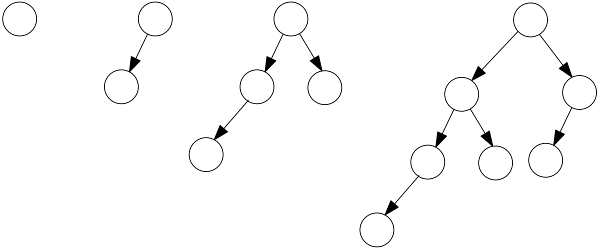
- $f(0) = 0, f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 7 \cdots$

- $f(h)$: minimum size of a balanced tree of height h



$$f(0) = 0$$
$$f(1) = 1$$
$$f(h) = f(h-1) + f(h-2) + 1 \qquad h \geq 2$$

- $f(h)$: minimum size of a balanced tree of height h



$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(h) &= f(h-1) + f(h-2) + 1 \qquad\qquad h \geq 2
\end{aligned}
$$

- $f(h) = 2^{\Theta(h)}$ (i.e, $\lg f(h) = \Theta(h)$)

Depth of AVL tree

- $f(h)$: minimum size of a balanced tree of height h
- $f(h) = 2^{\Theta(h)}$

Depth of AVL tree

- $f(h)$: minimum size of a balanced tree of height h
- $f(h) = 2^{\Theta(h)}$
- If a AVL tree has size n and height h, then

$$n \geq f(h) = 2^{\Theta(h)}$$

Depth of AVL tree

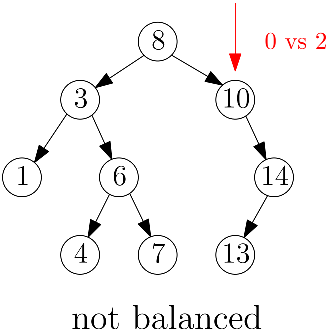
- $f(h)$: minimum size of a balanced tree of height h
- $f(h) = 2^{\Theta(h)}$
- If a AVL tree has size n and height h, then

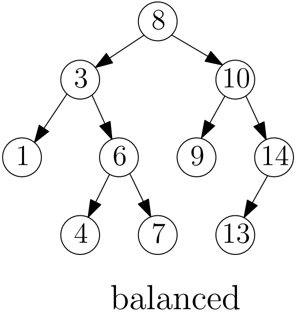
$$n \geq f(h) = 2^{\Theta(h)}$$

- Thus, $h \leq \Theta(\log n)$

An AVL Tree Is Balanced

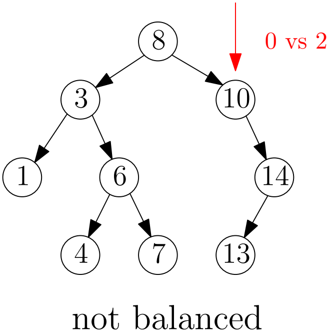
Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.





An AVL Tree Is Balanced

Balanced: for every node v in the tree, the heights of the left and right sub-trees of v differ by at most 1.



- How can we maintain the balanced property?

Maintain Balance Property After Insertion

Maintain Balance Property After Insertion

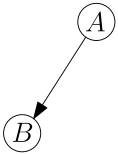
- A: the deepest node such that the balance property is not satisfied after insertion

Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A

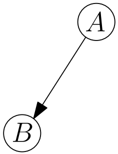
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A



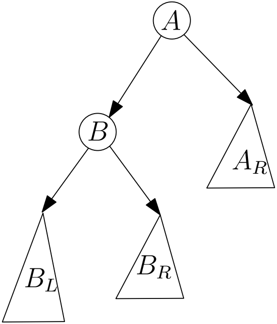
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



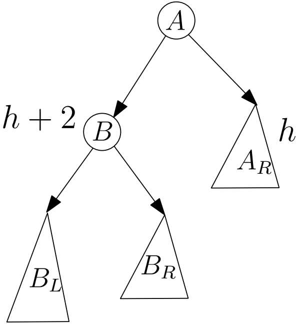
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



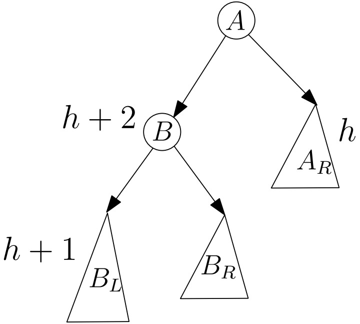
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



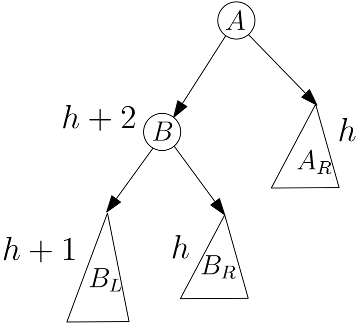
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



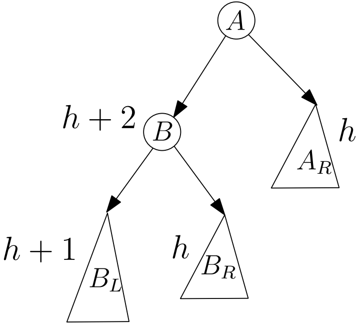
Maintain Balance Property After Insertion

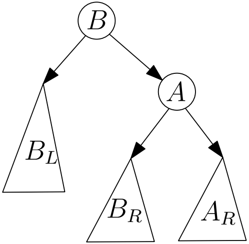
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



Maintain Balance Property After Insertion

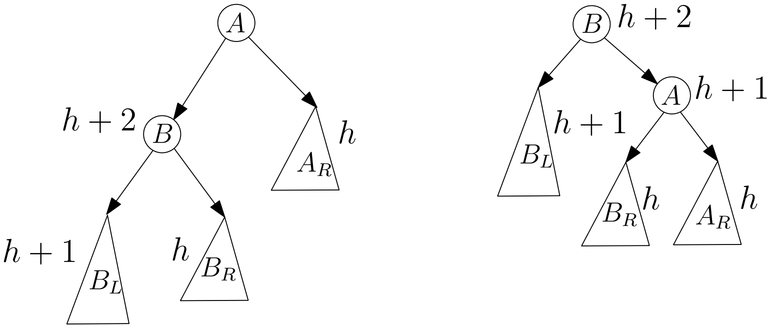
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B





Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A

Maintain Balance Property After Insertion

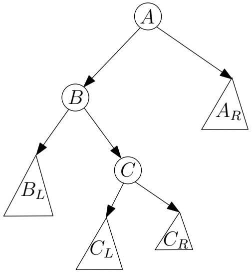
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B

Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B

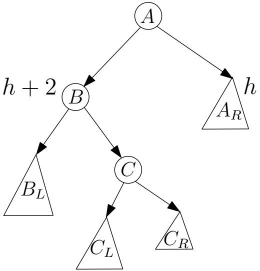
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



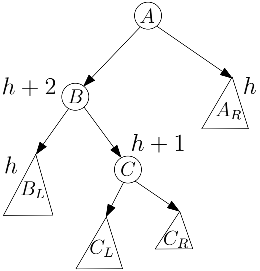
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



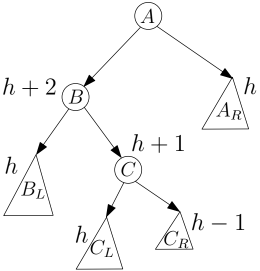
Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



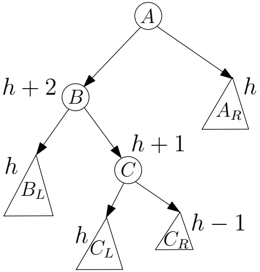
Maintain Balance Property After Insertion

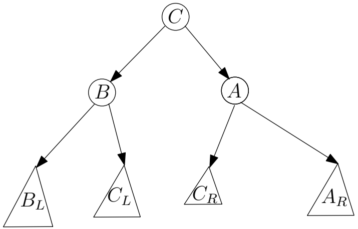
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



Maintain Balance Property After Insertion

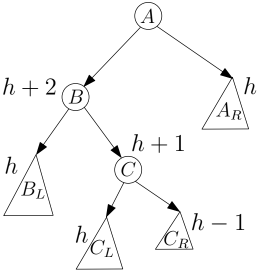
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B

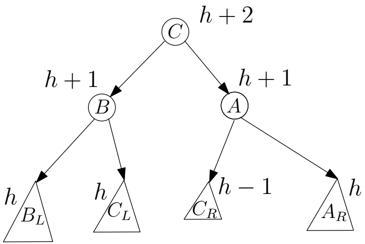




Maintain Balance Property After Insertion

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B





count-inversions(A, n)

$T \leftarrow$ empty AVL tree
$c \leftarrow 0$
for $i \leftarrow n$ downto 1
    $c \leftarrow c + T$.count-less-than$(A[i])$
    T.insert$(A[i])$
return c

count-inversions(A, n)

```
T ← empty AVL tree
c ← 0
for i ← n downto 1
    c ← c + T.count-less-than(A[i])
    T.insert(A[i])
return c
```

- Each operation (insert, delete, count-less-than, etc.) takes time $O(h) = O(\lg n)$.

count-inversions(A, n)

$T \leftarrow$ empty AVL tree
$c \leftarrow 0$
for $i \leftarrow n$ downto 1
    $c \leftarrow c + T$.count-less-than$(A[i])$
    T.insert$(A[i])$
return c

- Each operation (insert, delete, count-less-than, etc.) takes time $O(h) = O(\lg n)$.
- Running time $= O(n \lg n)$

Outline

Fibonacci Numbers

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots$

n-th Fibonacci Number

Input: integer $n > 0$

Output: F_n

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

1. if $n = 0$ return 0
2. if $n = 1$ return 1
3. return Fib$(n-1)$ + Fib$(n-2)$

Q: Is the running time of the algorithm polynomial or exponential in n?

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

1. if $n = 0$ return 0
2. if $n = 1$ return 1
3. return Fib($n-1$) + Fib($n-2$)

Q: Is the running time of the algorithm polynomial or exponential in n?

A: Exponential

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

1. if $n = 0$ return 0
2. if $n = 1$ return 1
3. return Fib$(n-1)$ + Fib$(n-2)$

Q: Is the running time of the algorithm polynomial or exponential in n?

A: Exponential

- Running time is at least $\Omega(F_n)$

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

1. if $n = 0$ return 0
2. if $n = 1$ return 1
3. return Fib$(n-1)$ + Fib$(n-2)$

Q: Is the running time of the algorithm polynomial or exponential in n?

A: Exponential

- Running time is at least $\Omega(F_n)$
- F_n is exponential in n

Computing F_n: Reasonable Algorithm

Fib(n)

1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i \leftarrow 2$ to n do
4. $\quad F[i] \leftarrow F[i-1] + F[i-2]$
5. return $F[n]$

- Dynamic Programming

Computing F_n: Reasonable Algorithm

Fib(n)

1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i \leftarrow 2$ to n do
4. $\quad F[i] \leftarrow F[i-1] + F[i-2]$
5. return $F[n]$

- Dynamic Programming
- Running time $=$?

Computing F_n: Reasonable Algorithm

Fib(n)

1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i \leftarrow 2$ to n do
4. $\quad F[i] \leftarrow F[i-1] + F[i-2]$
5. return $F[n]$

- Dynamic Programming
- Running time $= O(n)$

Computing F_n: Even Better Algorithm

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} F_{n-2} \\ F_{n-3} \end{pmatrix}$$

$$\cdots$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

power(n)

1. if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. $R \leftarrow$ power($\lfloor n/2 \rfloor$)
3. $R \leftarrow R \times R$
4. if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5. return R

Fib(n)

1. if $n = 0$ then return 0
2. $M \leftarrow$ power($n - 1$)
3. return $M[1][1]$

power(n)

1. if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. $R \leftarrow$ power($\lfloor n/2 \rfloor$)
3. $R \leftarrow R \times R$
4. if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5. return R

Fib(n)

1. if $n = 0$ then return 0
2. $M \leftarrow$ power($n - 1$)
3. return $M[1][1]$

- Recurrence for running time?

power(n)

1. if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. $R \leftarrow$ power($\lfloor n/2 \rfloor$)
3. $R \leftarrow R \times R$
4. if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5. return R

Fib(n)

1. if $n = 0$ then return 0
2. $M \leftarrow$ power($n - 1$)
3. return $M[1][1]$

- Recurrence for running time? $T(n) = T(n/2) + O(1)$

power(n)

1. if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. $R \leftarrow$ power($\lfloor n/2 \rfloor$)
3. $R \leftarrow R \times R$
4. if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5. return R

Fib(n)

1. if $n = 0$ then return 0
2. $M \leftarrow$ power($n - 1$)
3. return $M[1][1]$

- Recurrence for running time? $T(n) = T(n/2) + O(1)$
- $T(n) = O(\lg n)$

Running time $= O(\lg n)$: We Cheated!

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time

Running time = $O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time
- Even printing $F(n)$ requires time much larger than $O(\lg n)$

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time
- Even printing $F(n)$ requires time much larger than $O(\lg n)$

Fixing the Problem

To compute F_n, we need $O(\lg n)$ basic arithmetic operations on integers

Summary: Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

Summary: Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance
- Write down recurrence for running time
- Solve recurrence using master theorem

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, $\cdots$:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, $\cdots$:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$
- Integer Multiplication:
  $T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\lg_2 3})$

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, $\cdots$:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$
- Integer Multiplication:
  $T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\lg_2 3})$
- Matrix Multiplication:
  $T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) = O(n^{\lg_2 7})$

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, $\cdots$:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$
- Integer Multiplication:
  $T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\lg_2 3})$
- Matrix Multiplication:
  $T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) = O(n^{\lg_2 7})$

- Usually, designing better algorithm for "combine" step is key to improve running time