COMP 550
Algorithm and Analysis

Divide & Conquer

Based on CLRS Sec 2.3, 4, and (33.4 of the 3rd edition)

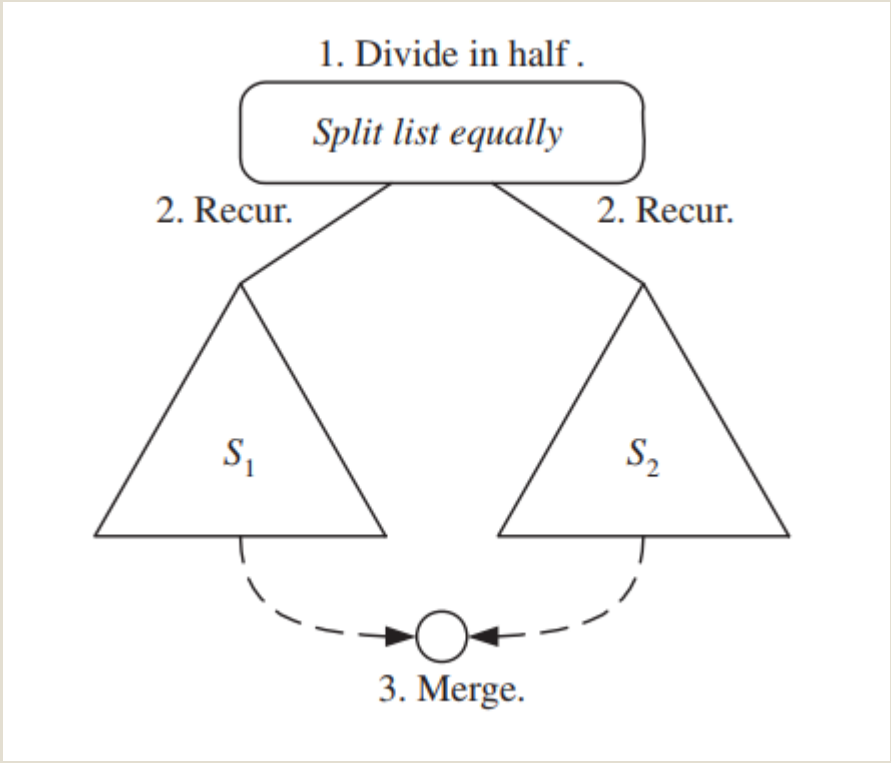
Some slides are adapted from ones by prior instructors Prof. Plaisted and Prof. Osborne

Divide & Conquer

- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

Divide & Conquer

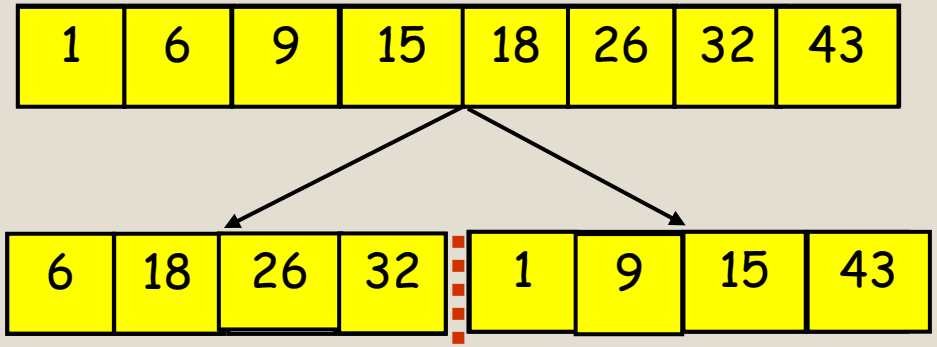
```
Divide-and-Conquer(A)
   if size of A is small
      calculate and return result
   Partition A into A1, A2, …, Ak
   R1 = Divide-and-Conquer(A1)
   R2 = Divide-and-Conquer(A2)
   …
   Rk = Divide-and-Conquer(Ak)
   R = Calculate result from R1, R2, …, Rk
   return R
```



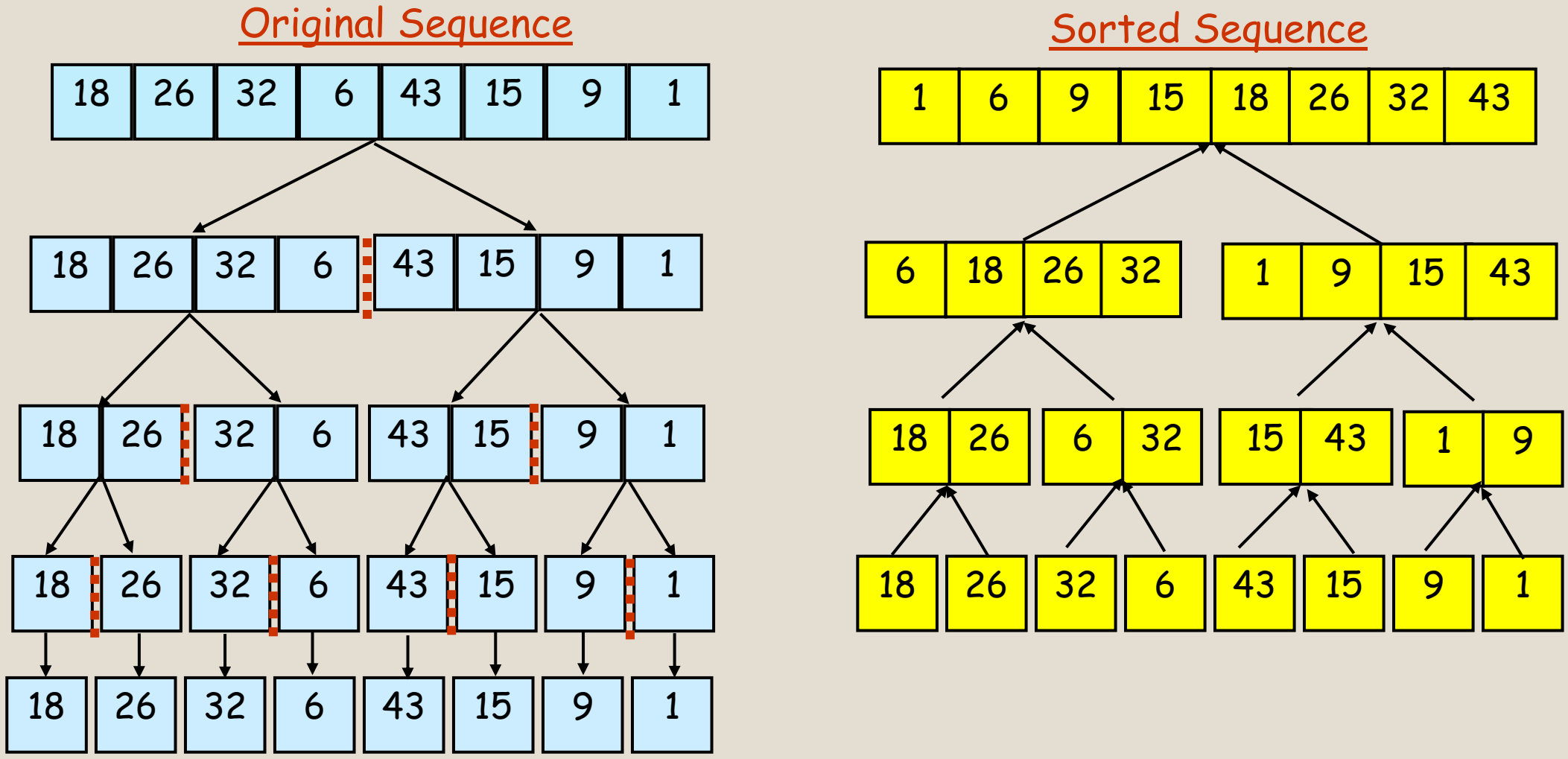
Merge Sort

Sorting Problem: Sort a sequence of *n* elements into non-decreasing order.

- **Divide:** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.



Merge Sort


Original Sequence
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
Sorted Sequence
1 6 9 15 18 26 32 43
6 18 26 32 1 9 15 43
18 26 6 32 15 43 1 9
18 26 32 6 43 15 9 1

Merge Sort

```
MERGE-SORT(A, p, r)
1  if p ≥ r                                  // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                           // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)                       // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)                   // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Merge Procedure

```
Merge(A, p, q, r)
1.  n_L = q − p + 1
2.  n_R = r − q
3.  Let L[1:n_L] and R[1:n_R] be new arrays
4.  for i = 1 to n_L
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_R
7.      R[j] = A[q + j]
8.  L[n_L+1] = R[n_R+1] = ∞
9.  i = j = 1
10. for k = p to r
11.     if L[i] ≤ R[j]
12.         A[k] = L[i]
13.         i = i + 1
14.     else
15.         A[k] = R[j]
16.         j = j + 1
```

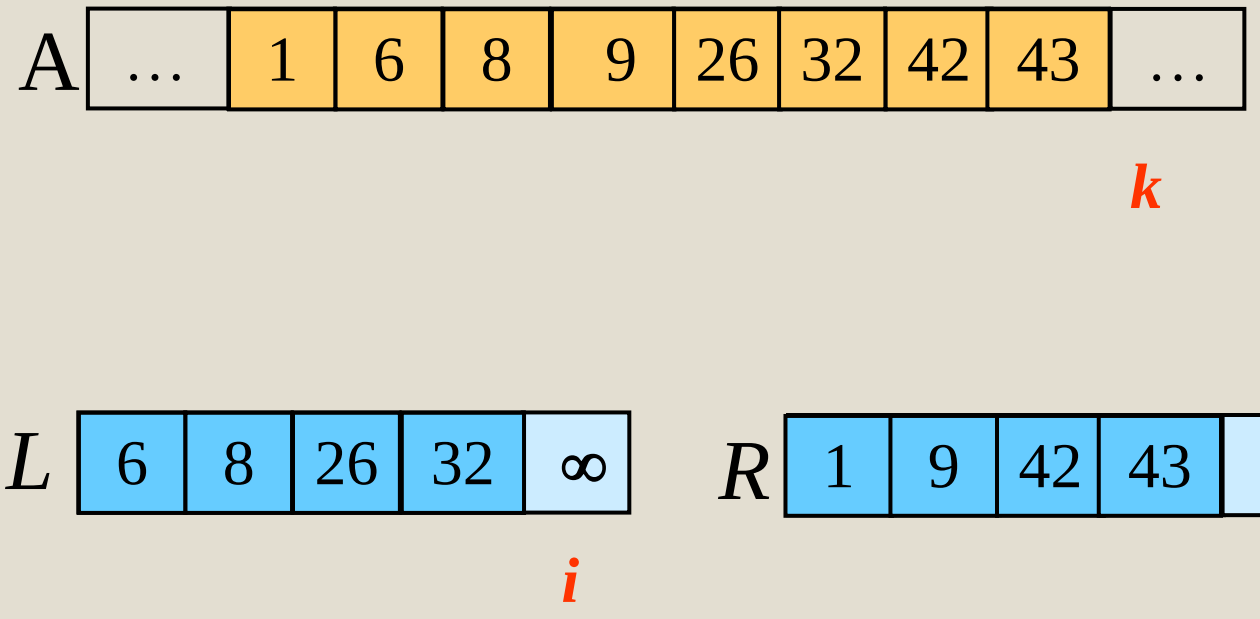
From CLSR 3rd edition

- Input: Array A with sorted subarrays $A[p:q]$ and $A[q+1,r]$
- Output: Array A with sorted subarray $A[p:r]$

Sentinels: Simplification by avoiding having to check if either subarray is fully copied at each step.

Merge Procedure

```
Merge(A, p, q, r)
1.  n_L = q − p + 1
2.  n_R = r − q
3.  Let L[1:n_L] and R[1:n_R] be new arrays
4.  for i = 1 to n_L
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_R
7.      R[j] = A[q + j]
8.  L[n_L+1] = R[n_R+1] = ∞
9.  i = j = 1
10. for k = p to r
11.     if L[i] ≤ R[j]
12.         A[k] = L[i]
13.         i = i + 1
14.     else
15.         A[k] = R[j]
16.         j = j + 1
```



Merge Procedure

```
Merge(A, p, q, r)
1.  n_L = q − p + 1
2.  n_R = r − q
3.  Let L[1:n_L] and R[1:n_R] be new arrays
4.  for i = 1 to n_L
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_R
7.      R[j] = A[q + j]
8.  L[n_L+1] = R[n_R+1] = ∞
9.  i = j = 1
10. for k = p to r
11.     if L[i] ≤ R[j]
12.         A[k] = L[i]
13.         i = i + 1
14.     else
15.         A[k] = R[j]
16.         j = j + 1
```

- **Correctness:**

We need to show that-

- Given sorted subarrays $A[p:q]$ and $A[q+1:r]$, the Merge procedure constructs a sorted subarray $A[p:r]$ that consists of all elements in $A[p:q]$ and $A[q+1:r]$

Merge Procedure

```
Merge(A, p, q, r)
1.  n_L = q − p + 1
2.  n_R = r − q
3.  Let L[1:n_L] and R[1:n_R] be new arrays
4.  for i = 1 to n_L
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_R
7.      R[j] = A[q + j]
8.  L[n_L+1] = R[n_R+1] = ∞
9.  i = j = 1
10. for k = p to r
11.     if L[i] ≤ R[j]
12.         A[k] = L[i]
13.         i = i + 1
14.     else
15.         A[k] = R[j]
16.         j = j + 1
```

- **Loop Invariant for the *for* loop**

At the start of each iteration of the for loop of lines 10-16, subarray $A[p:k-1]$ consists of the $k-p$ smallest elements of L and R in sorted order. $L[i]$ and $R[j]$ are the smallest elements of L and R that have not been copied back into A.

Initialization: Trivial

Merge Procedure

```
Merge(A, p, q, r)
1.  n_L = q − p + 1
2.  n_R = r − q
3.  Let L[1:n_L] and R[1:n_R] be new arrays
4.  for i = 1 to n_L
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_R
7.      R[j] = A[q + j]
8.  L[n_L+1] = R[n_R+1] = ∞
9.  i = j = 1
10. for k = p to r
11.     if L[i] ≤ R[j]
12.         A[k] = L[i]
13.         i = i + 1
14.     else
15.         A[k] = R[j]
16.         j = j + 1
```

Maintenance:

Case 1: $L[i] \leq R[j]$

- LI holds at the start of k-th iteration
- By LI, *A* contains $k - p$ smallest elements of *L* and *R* in sorted order.
- By LI, *L*[*i*] and *R*[*j*] are the smallest elements of *L* and *R* not yet copied into *A*.
- Line 12 results in *A* containing $k - p + 1$ smallest elements (again in sorted order).
- Incrementing *i* and *k* establishes the LI for the next iteration.
- **Similarly, for $L[i] > R[j]$.**

Merge Procedure

```
Merge(A, p, q, r)
1.  n_L = q − p + 1
2.  n_R = r − q
3.  Let L[1:n_L] and R[1:n_R] be new arrays
4.  for i = 1 to n_L
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_R
7.      R[j] = A[q + j]
8.  L[n_L+1] = R[n_R+1] = ∞
9.  i = j = 1
10. for k = p to r
11.     if L[i] ≤ R[j]
12.         A[k] = L[i]
13.         i = i + 1
14.     else
15.         A[k] = R[j]
16.         j = j + 1
```

Termination:

- On termination, $k = r + 1$.
- By LI, A contains $r - p + 1$ smallest
- elements of L and R in sorted order.
- L and R together contain $r - p + 3$ elements.
- All but the two sentinels have been copied back into A.

Merge Sort

- Correctness of Merge Sort:

We need to prove that-

Given a (sub)array $A[p:r]$, merge sort correctly sort $A[p:r]$.

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Merge Sort

Use Strong Induction

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Base case: Array size $= 1$ (In other words, $p = r$)

Merge-Sort is correct due to lines 1-2

Inductive Hypothesis (IH): Merge sort correctly sorts any array of size $<$ n (In other words, $r - p < n$)

Inductive Step: By IH, after lines 4 and 5, $A[p{:}\,q]$ and $A[q+1{:}\,r]$ are sorted

By the Loop invariant we've proved, line 7 correctly constructs sorted $A[p{:}\,r]$

So, Merge Sort is correct.

Merge Procedure: Time Complexity

```
Merge(A, p, q, r)
1.  n_L = q − p + 1
2.  n_R = r − q
3.  Let L[1:n_L] and R[1:n_R] be new arrays
4.  for i = 1 to n_L
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_R
7.      R[j] = A[q + j]
8.  L[n_L+1] = R[n_R+1] = ∞
9.  i = j = 1
10. for k = p to r
11.     if L[i] ≤ R[j]
12.         A[k] = L[i]
13.         i = i + 1
14.     else
15.         A[k] = R[j]
16.         j = j + 1
```

Lines 1-3: $\Theta(1)$ time

Lines 4-7: $\Theta(n_L + n_R) = \Theta(n)$ time

$$n = n_L + n_R = r - p + 1$$

Lines 8-9: $\Theta(1)$ time

Lines 10-16: $\Theta(r - p + 1) = \Theta(n)$ time

Total running time: $\Theta(n)$

Merge Sort: Time Complexity

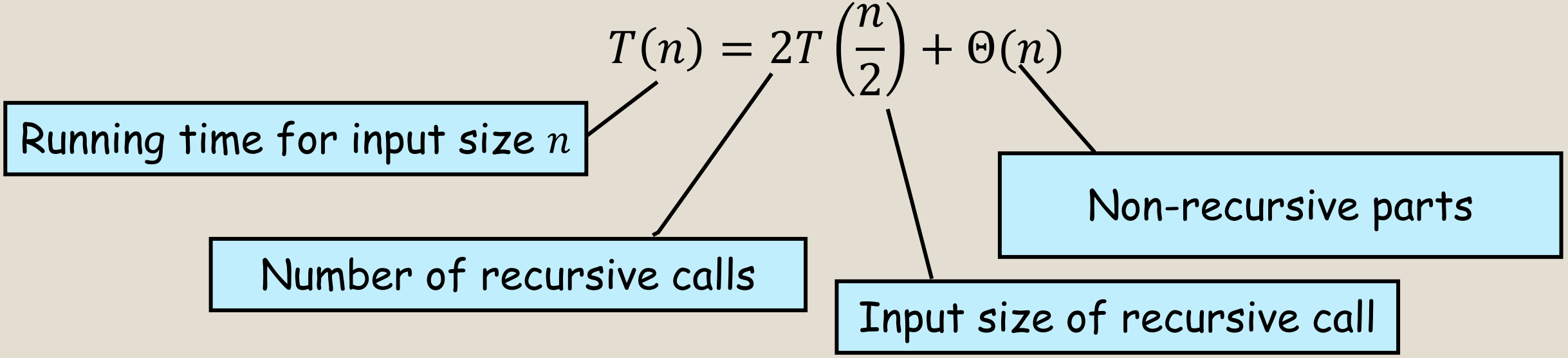
- Determine running time $T(n)$ to sort n elements

```
MERGE-SORT(A, p, r)
1  if p ≥ r
2       return
3  q = ⌊(p + r)/2⌋
4  MERGE-SORT(A, p, q)
5  MERGE-SORT(A, q + 1, r)
6  // Merge A[p : q] and A[q + 1 : r]
7  MERGE(A, p, q, r)
```

Time
$\Theta(1)$
$\Theta(1)$
$\Theta(1)$
$T\left(\frac{n}{2}\right)$
$\Theta(n)$

$$T(n) = 2 \times \mathrm{T}\left(\frac{n}{2}\right) + \Theta(n)$$

Merge Sort: Time Complexity



Actual Merge Sort recurrence: $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$

- Safe to ignore floors and ceilings in asymptotic analysis

Merge Sort: Time Complexity

- Considering the base case

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- We can rewrite the above as

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + c_2 n & \text{if } n > 1 \end{cases}$$

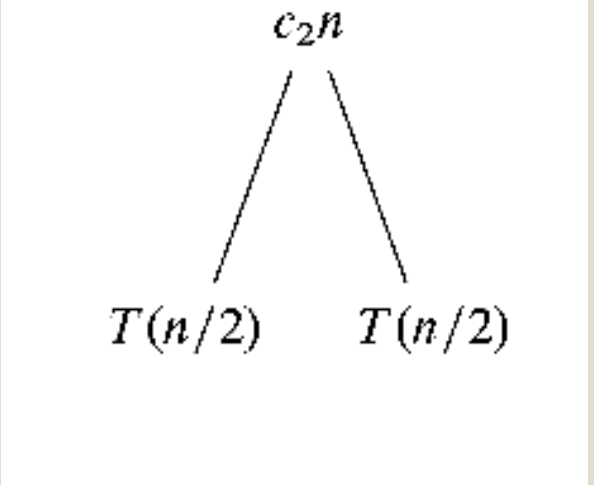
- Can we replace c_1 and c_2 by a single constant c?

Recurrence Relations

Solving Recurrence: Recursion Trees

$$T(n) = 2T\left(\frac{n}{2}\right) + c_2 n$$

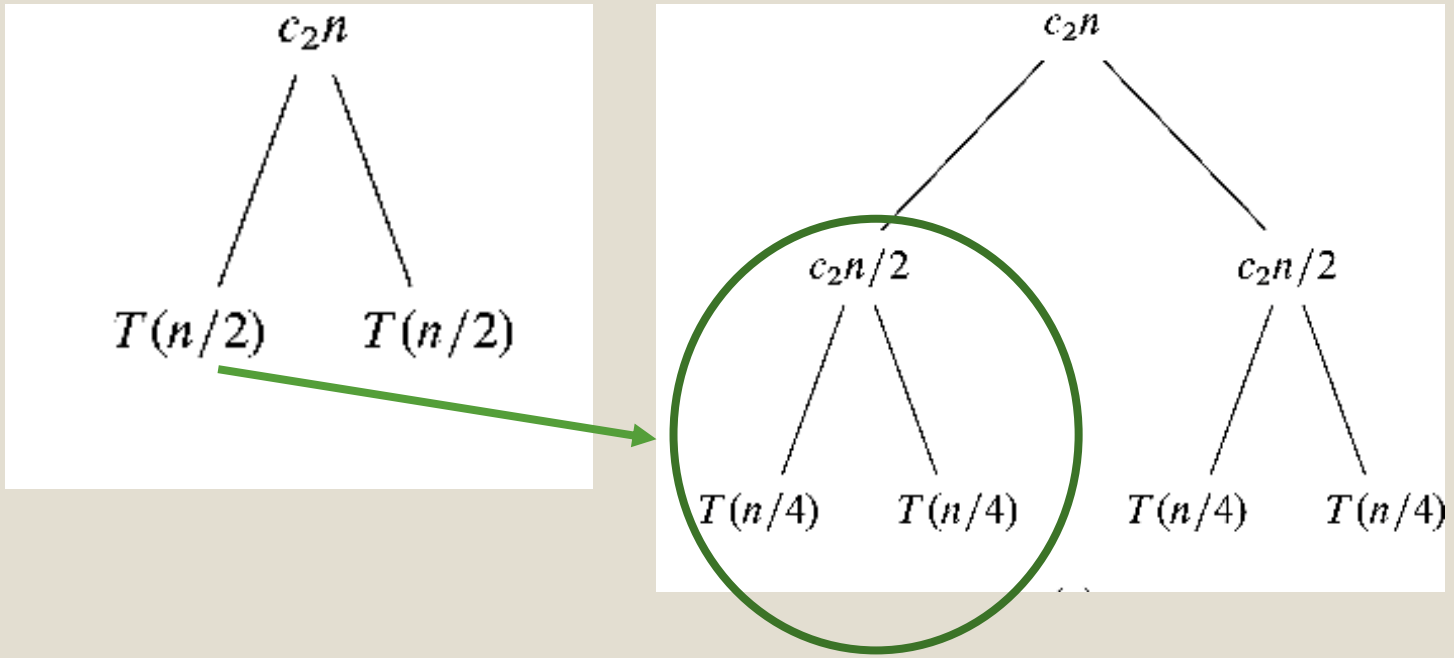
$$c_2 n$$
$$+$$
$$T\left(\frac{n}{2}\right) \quad + \quad T\left(\frac{n}{2}\right)$$



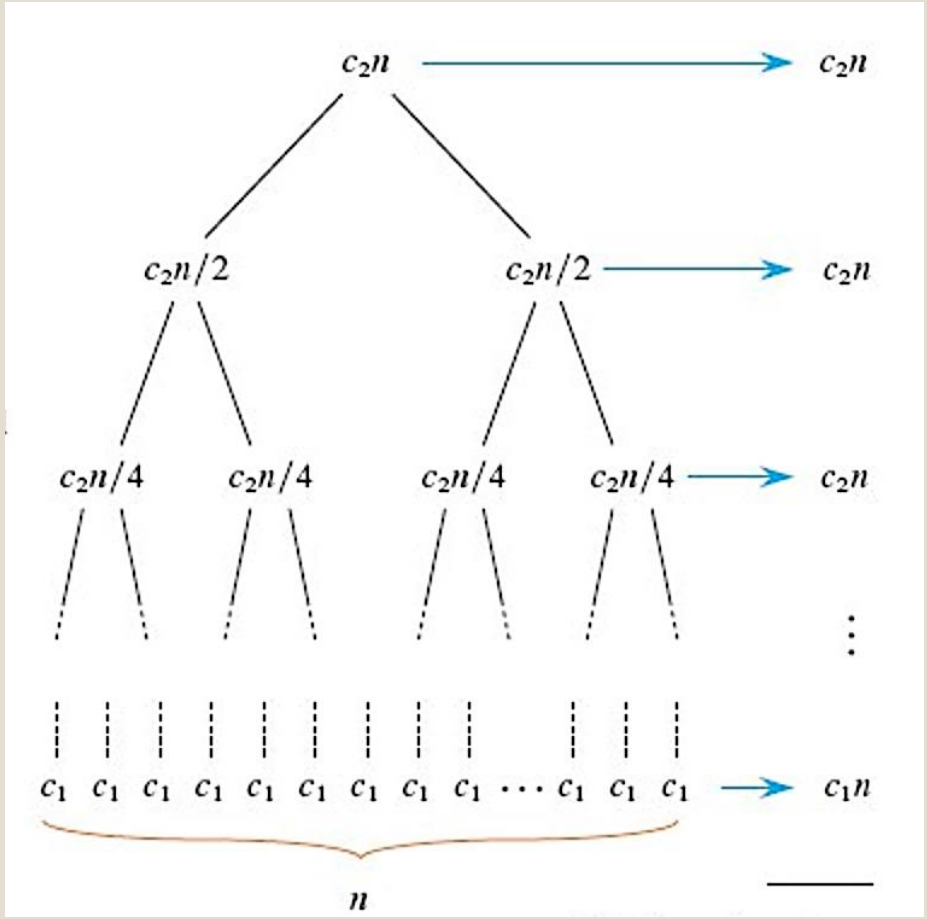
Recursion Tree Expansion

$$T(n) = c_2 n + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right)$$

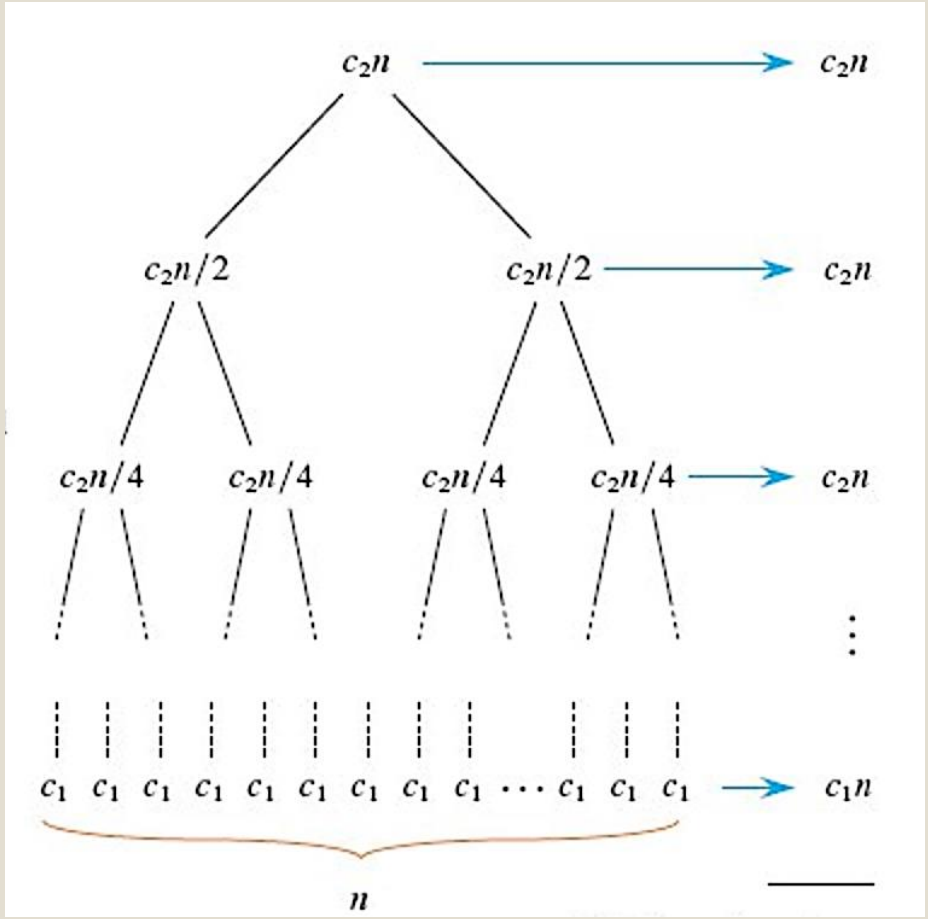
$$T\left(\frac{n}{2}\right) = c_2\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right)$$



Recursion Tree Expansion

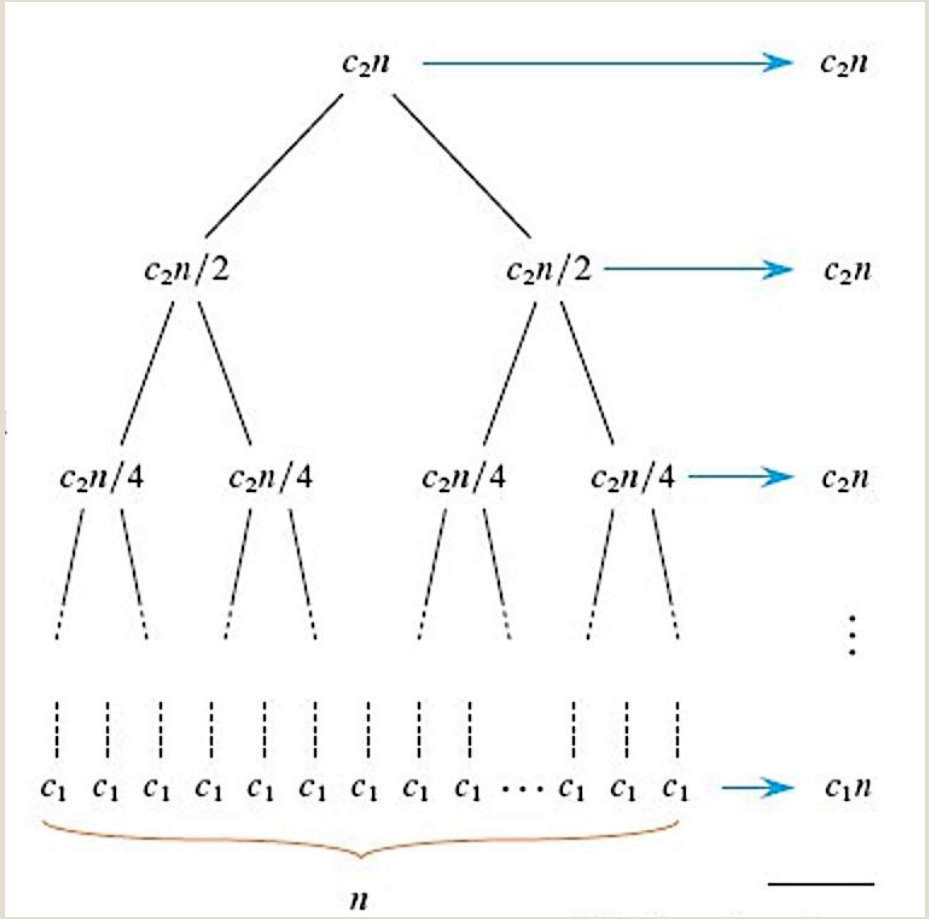


Recursion Tree Expansion



- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves
  - Cost per level remains the same (c_2n) until $n = 1$
- How many levels?
  - When will input size go from n to 1 by repetitive division by 2?
  - Alternative: When will input size go from 1 to n by repetitive multiplication by 2?

Recursion Tree Expansion



- Assume that n is an exact power of 2

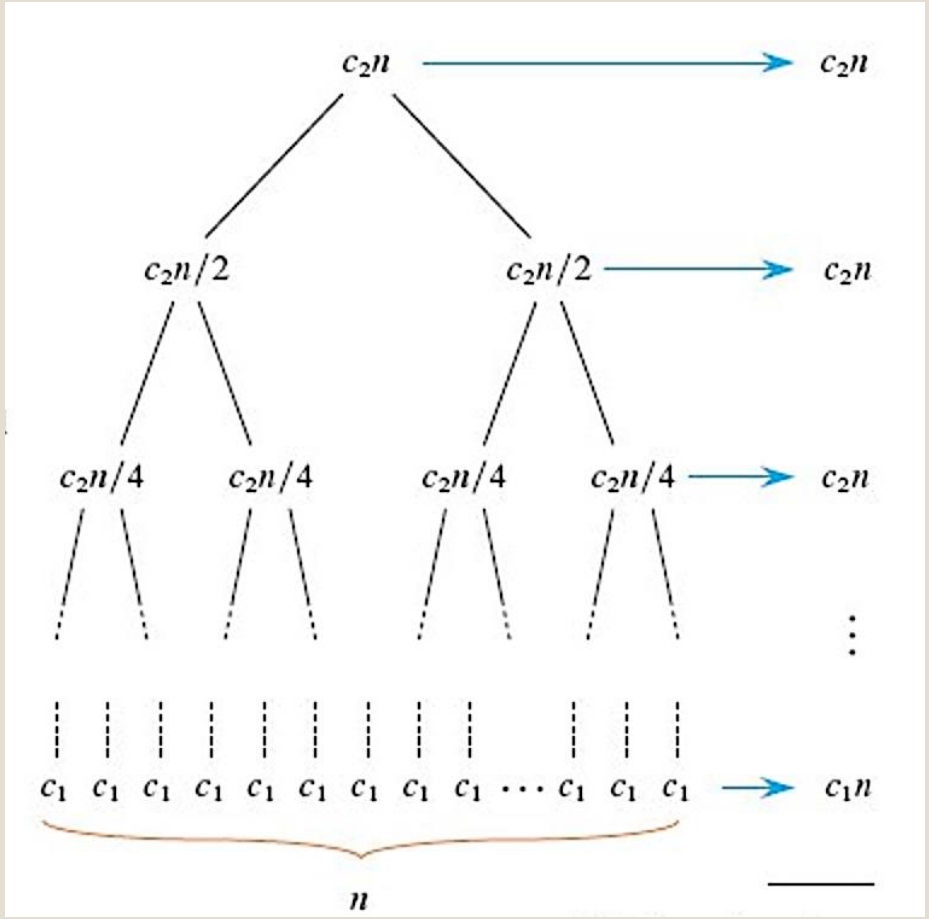
$$2 \times 2 \times 2 \times \cdots = n$$

$$\Rightarrow 2^y = n$$

$$\Rightarrow y = \lg n$$

- $y = \lg n$ levels from size 2 to size n
- $y + 1 = \lg n + 1$ steps from 1 to n

Recursion Tree Expansion



- Cost at each level except the base case (leaves) is $c_2 n$
- Number of such levels, $y = \lg n$
- Cost for base case $= c_1 n$
- Total cost $= \lg n \cdot c_2 n + c_1 n = \Theta(n \lg n)$

Ignoring Floors And Ceilings

- Actual merge sort recurrence,

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$$

$$\leq 2T\left(\frac{n}{2} + 1\right) + \Theta(n)$$

Define a new recurrence $S(n) = T(n + c)$, c is a constant

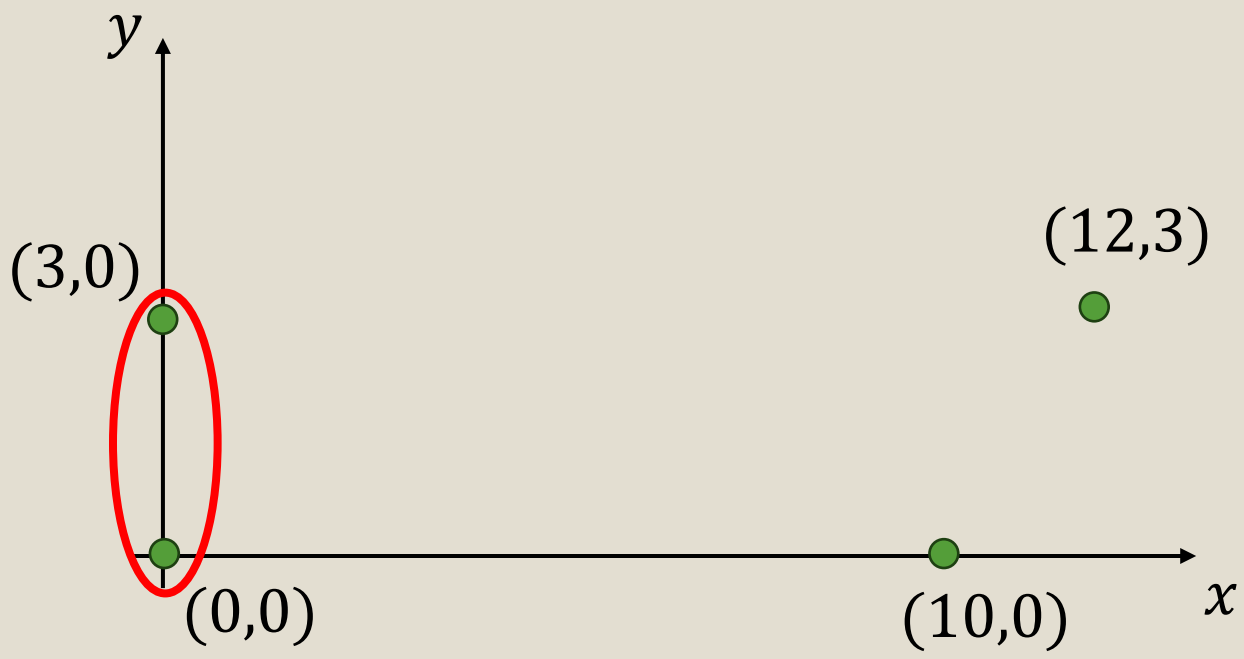
Applying def of T and S, $S(n) \leq 2S\left(\frac{n}{2} - \frac{c}{2} + 1\right) + n + c$

c=2 implies $S(n) \leq 2S\left(\frac{n}{2}\right) + n + 2$

Solution of $S(n) = O(n \lg n)$, asymptotically remains same for $T(n)$

Closest Pair of Points

- Input: A set $P = \{p_1, p_2, \ldots, p_n\}$ of n points on a plane. Each $p_i = (x_i, y_i)$
- Output: A pair of points that are closest together



Closest Pair of Points: Naïve Solution

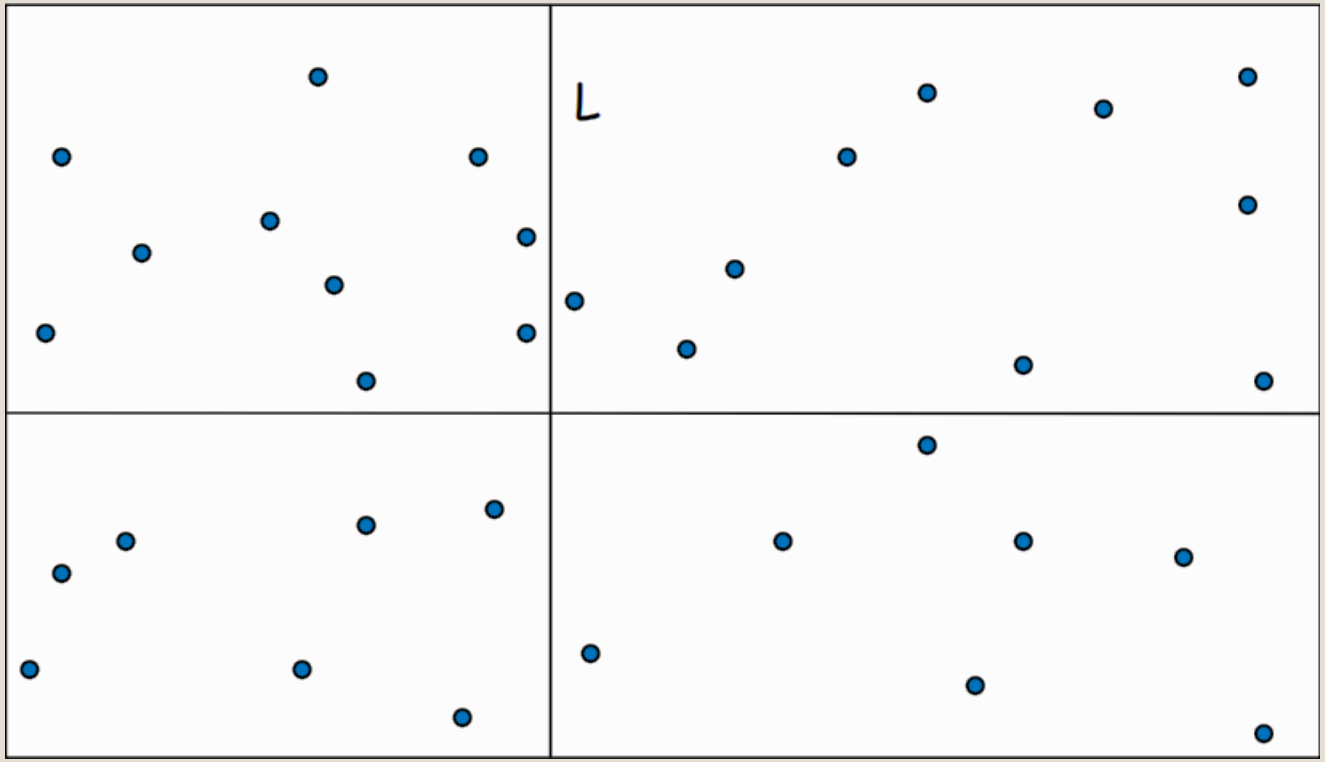
```
ClosestPointNaive(P,n)
1.  Minimum = ∞
2.  Closest = NIL
3.  for i = 1 to n
4.      for j = i + 1 to n
5.          if Dist(p_i , p_j) < Minimum
6.              Minimum = Dist(p_i , p_j)
7.              Closest = (p_i , p_j)
8.  return Closest

dist(p_i, p_j)
1.  // sqrt is not needed for this purpose
2.  return (p_i.x - p_j.x) * (p_i.x - p_j.x)
            + (p_i.y - p_j.y) * (p_i.y - p_j.y)
```

- What it this algorithm's running time?
  - $\Theta(n^2)$

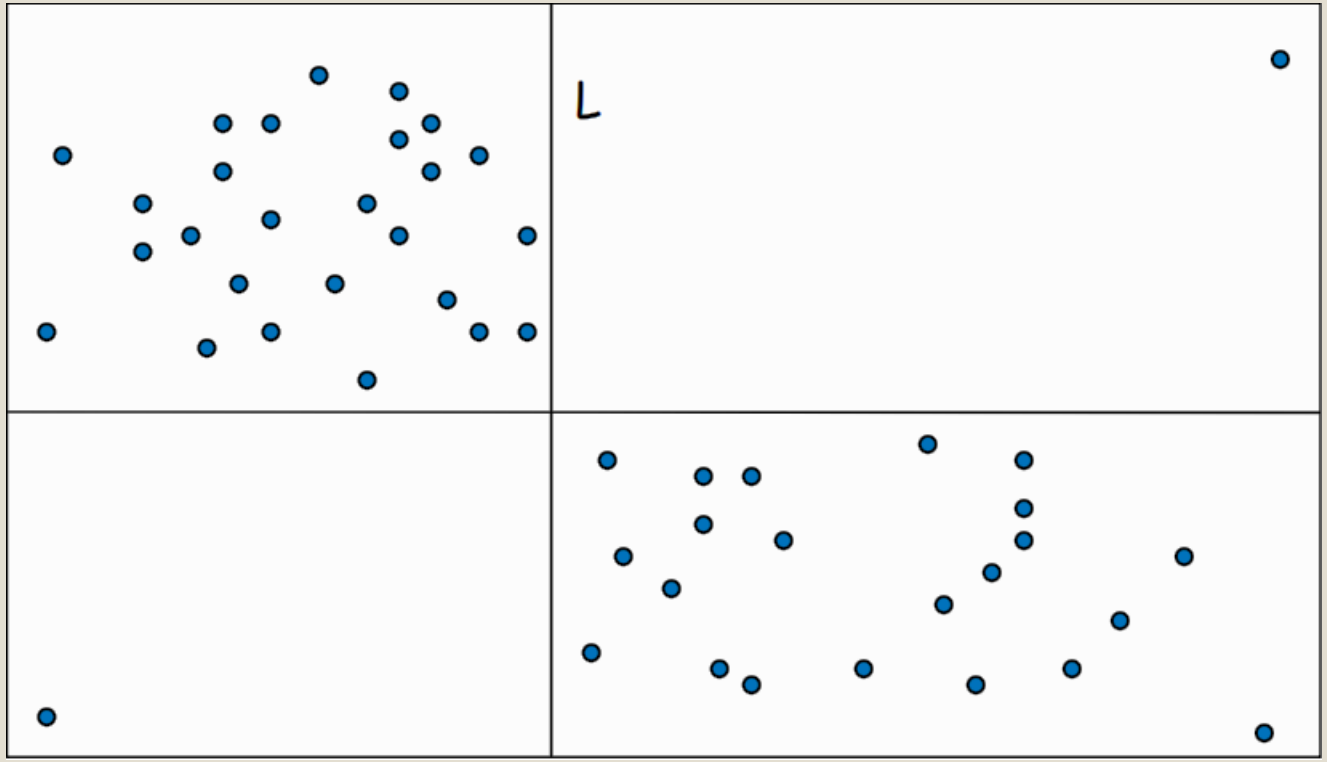
Divide and Conquer Solution

- Any better solution must run faster than $\Theta(n^2)$
  - Typical runtime to consider: $O(n), O(n \lg n), O(n \lg^2 n), \ldots$
- Divide and conquer approach similar to merge sort: first attempt



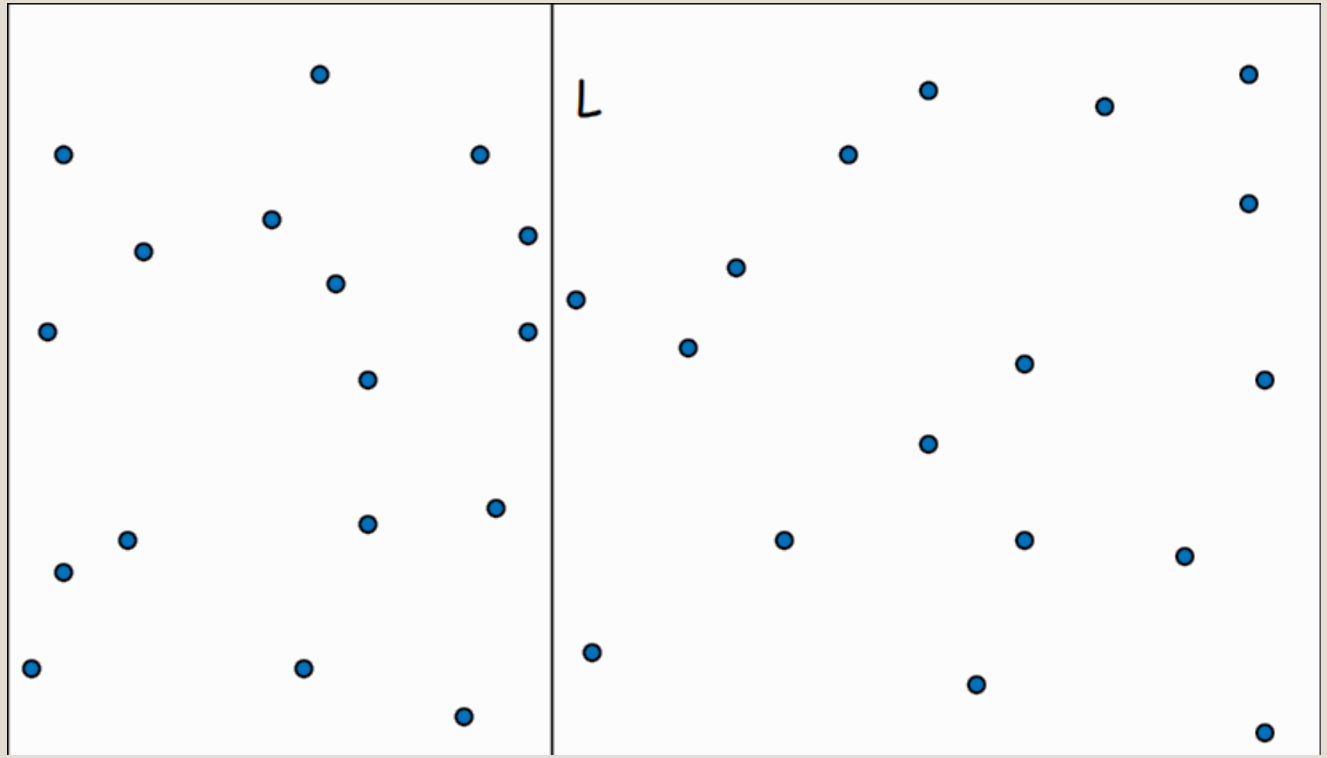
Divide and Conquer Solution

- Problem: Subdividing the entire region in 4 quadrants each with $\frac{n}{4}$ points is impossible.



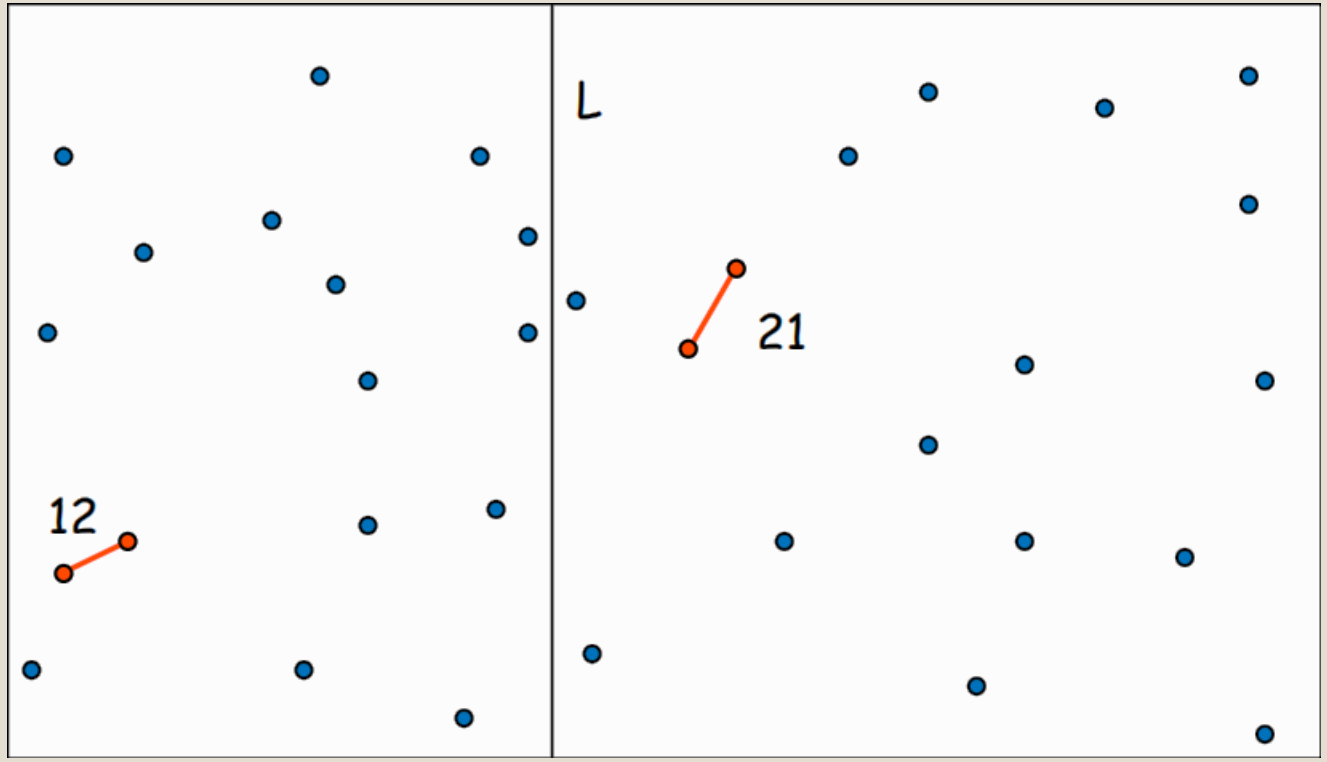
Divide and Conquer Solution

- Divide: n points into $\approx \frac{n}{2}$ points in left and right by a vertical line L
  - Need to sort all points according to x-coordinate



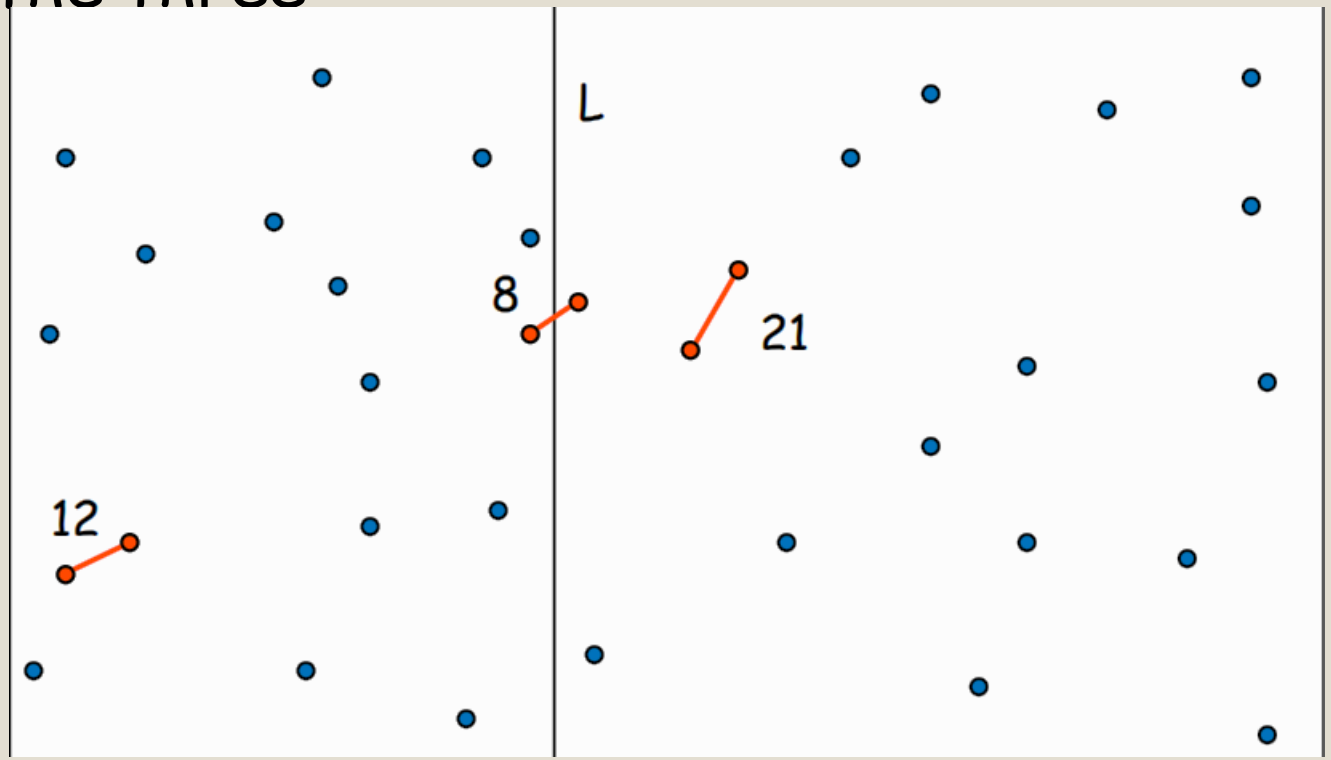
Divide and Conquer Solution

- Divide: n points into $\approx \frac{n}{2}$ points in left and right by a vertical line L
- Conquer: Find closest pair of points in each side recursively



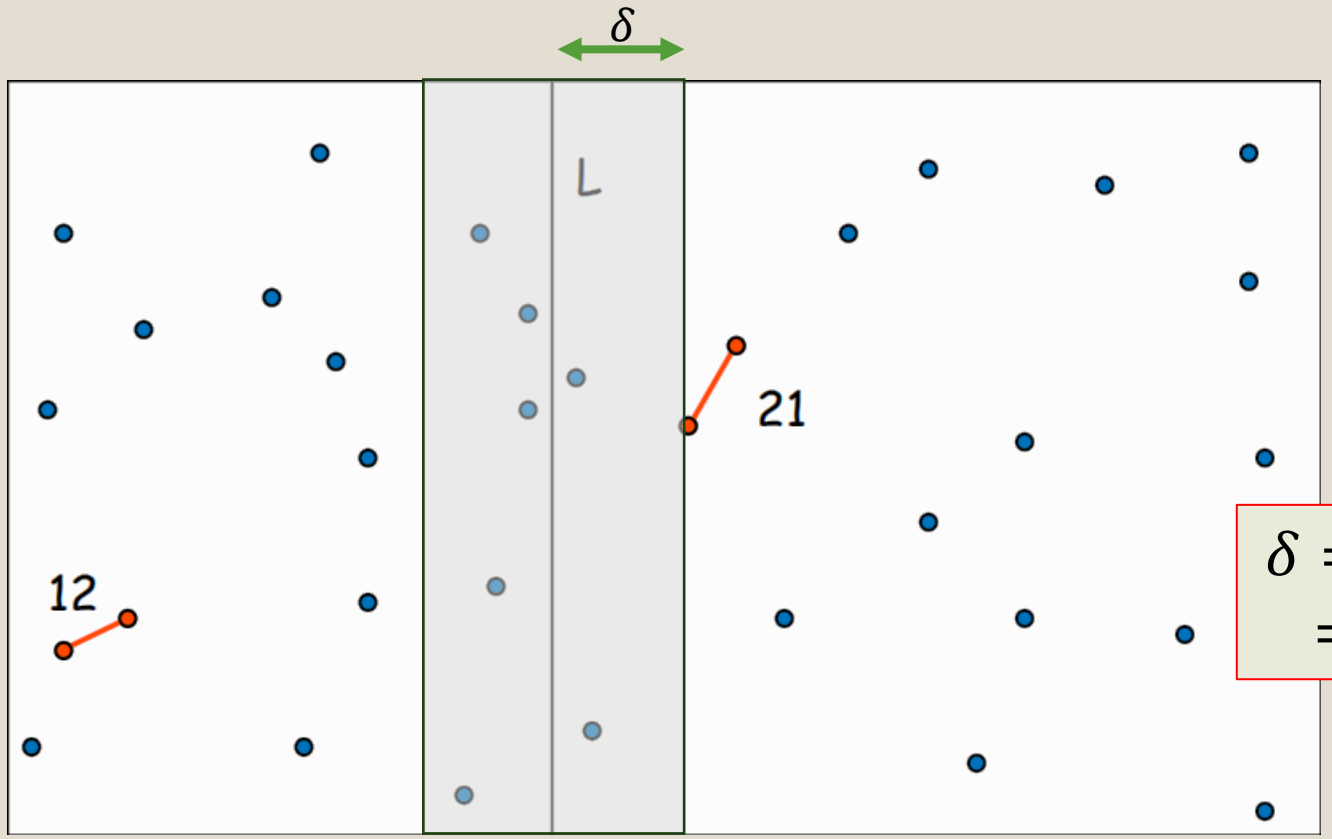
Divide and Conquer Solution

- Divide: n points into $\approx \frac{n}{2}$ points in left and right by a vertical line L
- Conquer: Find closest pair of points in each side recursively
- Combine: Find closest pair with one point in each side and return the closest among the three



Combine

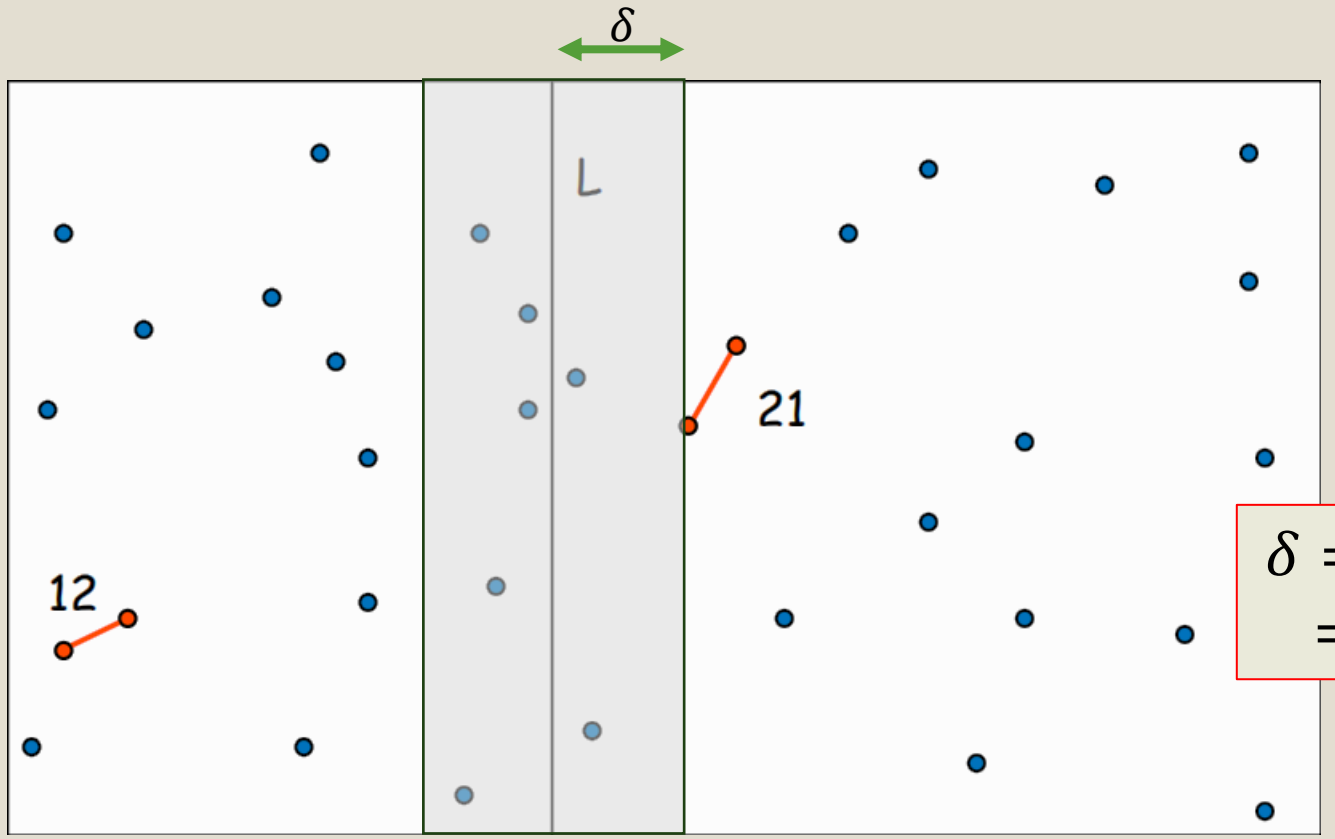
- Let δ_1, δ_2 be the distance between closest pair in each side.
- Let $\delta = \min\{\delta_1, \delta_2\}$
- Observation: Only points within δ within L can be closer than the current closest pair



$\delta = \min\{12,21\}$
$= 12$

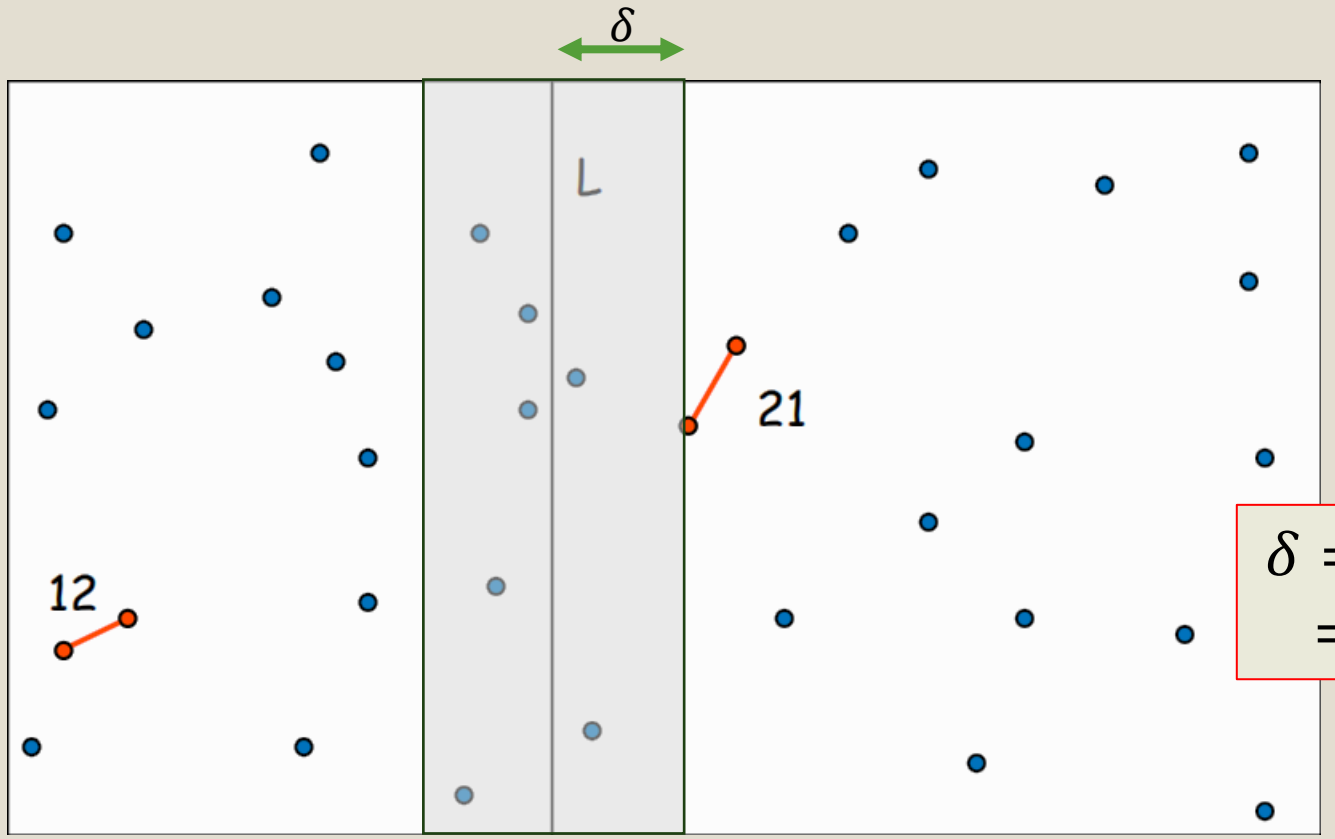
Combine

- Observation: Only points within δ within L are needed to be checked
- Does calculating all pair distance between two shaded region help?



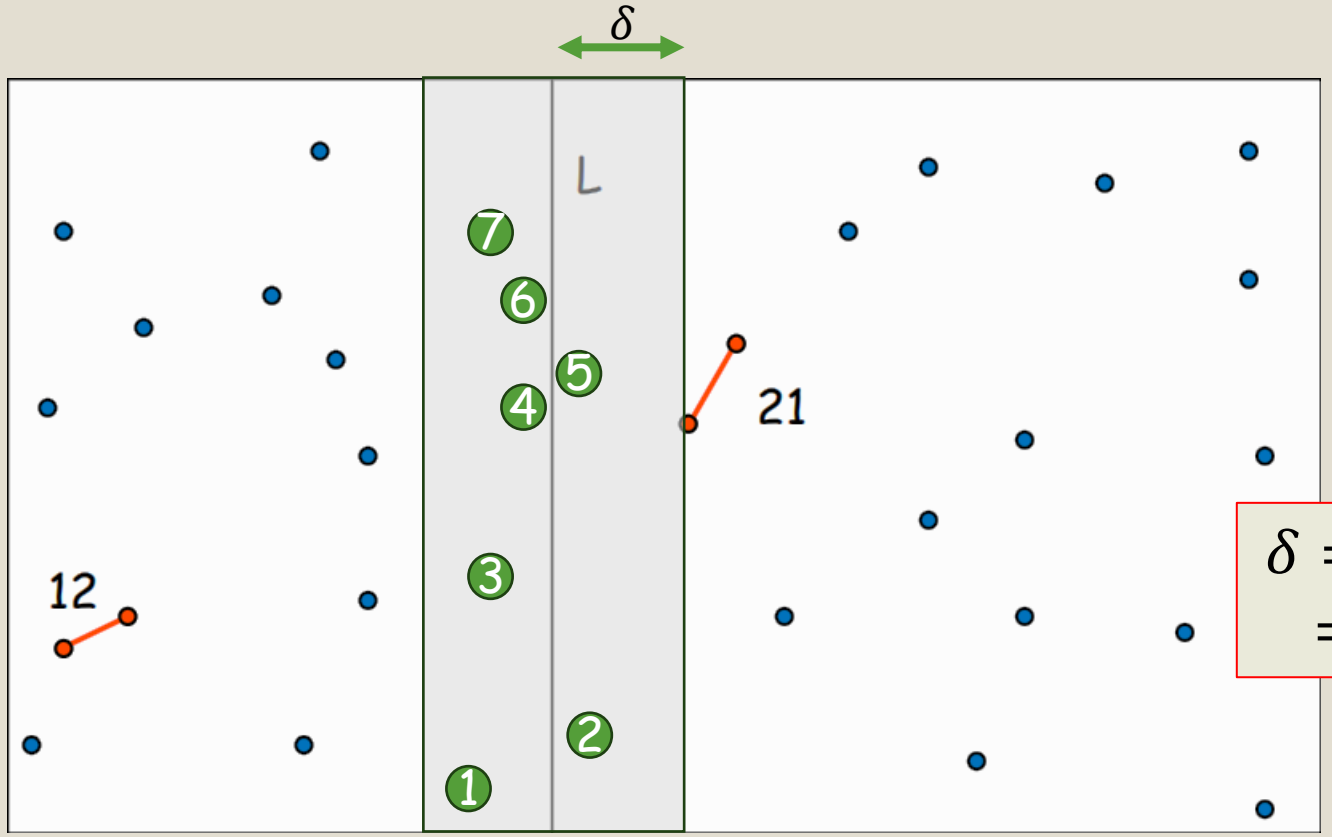
Combine

- Observation: Only points within δ within L are needed to be checked
- Does calculating all pair distance between two shaded region help?



Combine

- Observation: Only points within δ within L are needed to be checked
- Sort points in 2δ-strip by their y-coordinates
- For each point, only check distances of those within 11 positions in sorted list!



$$\delta = \min\{12, 21\} = 12$$

Combine

- Let S_y be the sorted points within the 2δ-strip

Lemma. If p_i, p_j lie in different side of L and $dist(p_i, p_j) < \delta$ holds, then p_i and p_j are within 11 points of each other in S_y

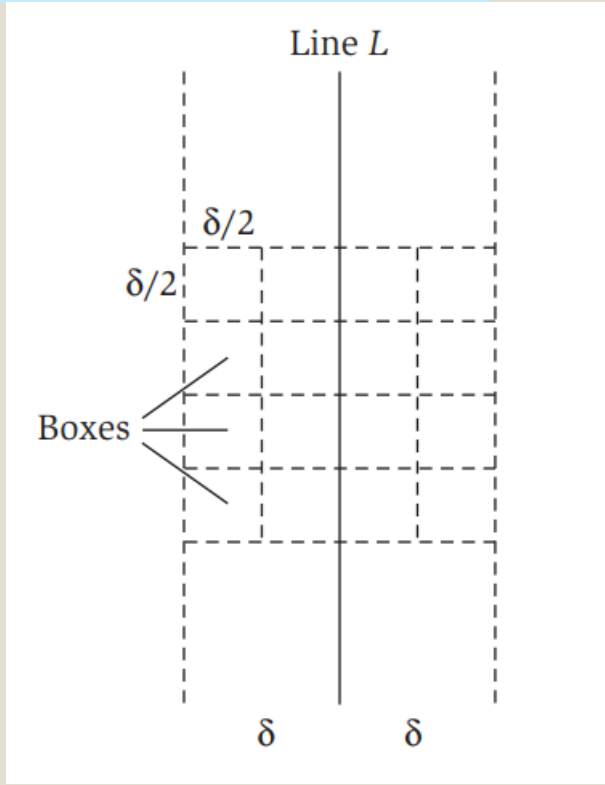
Proof: Assume that p_i, p_j are at least 12 points of each other.

Subdivide the 2δ-strip into $\frac{\delta}{2} \times \frac{\delta}{2}$ boxes.

Each box contains at most one point. **Why?**

So, there must be two rows of boxes between p_i, p_j. **Why?**

Then, $dist(p_i, p_j) \geq 2 \cdot \frac{\delta}{2} = \delta$, contradiction.



The Algorithm

```
ClosestPoint (P, n)
1.   if n ≤ 3
2.       Calculate all distances and return min
3.   Sort P by x-coordinates
4.   mid = ⌊n/2⌋
5.   L = vertical line according to p_mid
6.   (sL, sR, δ1) = ClosestPoint({p1, p2,…, p_mid})
7.   (tL, tR, δ2) = ClosestPoint({p_mid+1, p_mid+2,…, pn})
8.   δ = min(δ1, δ2)
9.   m1, m2 = Points corresponding to δ
10.  Sy = All points within 2δ-strip from L
11.  Sort Sy by y-coordinates
12.  ny = |Sy|
13.  for i = 1 to ny-1
14.      for j = i + 1 to min(i + 11, ny)
15.          if dist(pi, pj) < δ
16.              (m1,m2,δ) = (pi,pj,dist(pi, pj))
17.  return (m1,m2,δ)
```

- Lines 1-2, $\Theta(1)$ time
- Line 3, $\Theta(n \lg n)$ time
- Lines 4-5, $\Theta(1)$ time
- Lines 6-7, $2T\left(\frac{n}{2}\right)$ time
- Lines 8-9, $\Theta(1)$ time
- Line 10, $\Theta(n)$ time
- Line 11, $\Theta(n \lg n)$ time
- Line 12, $\Theta(1)$ time
- Lines 13-16, $\Theta(n)$ time
- Line 17, $\Theta(1)$ time

Recurrence

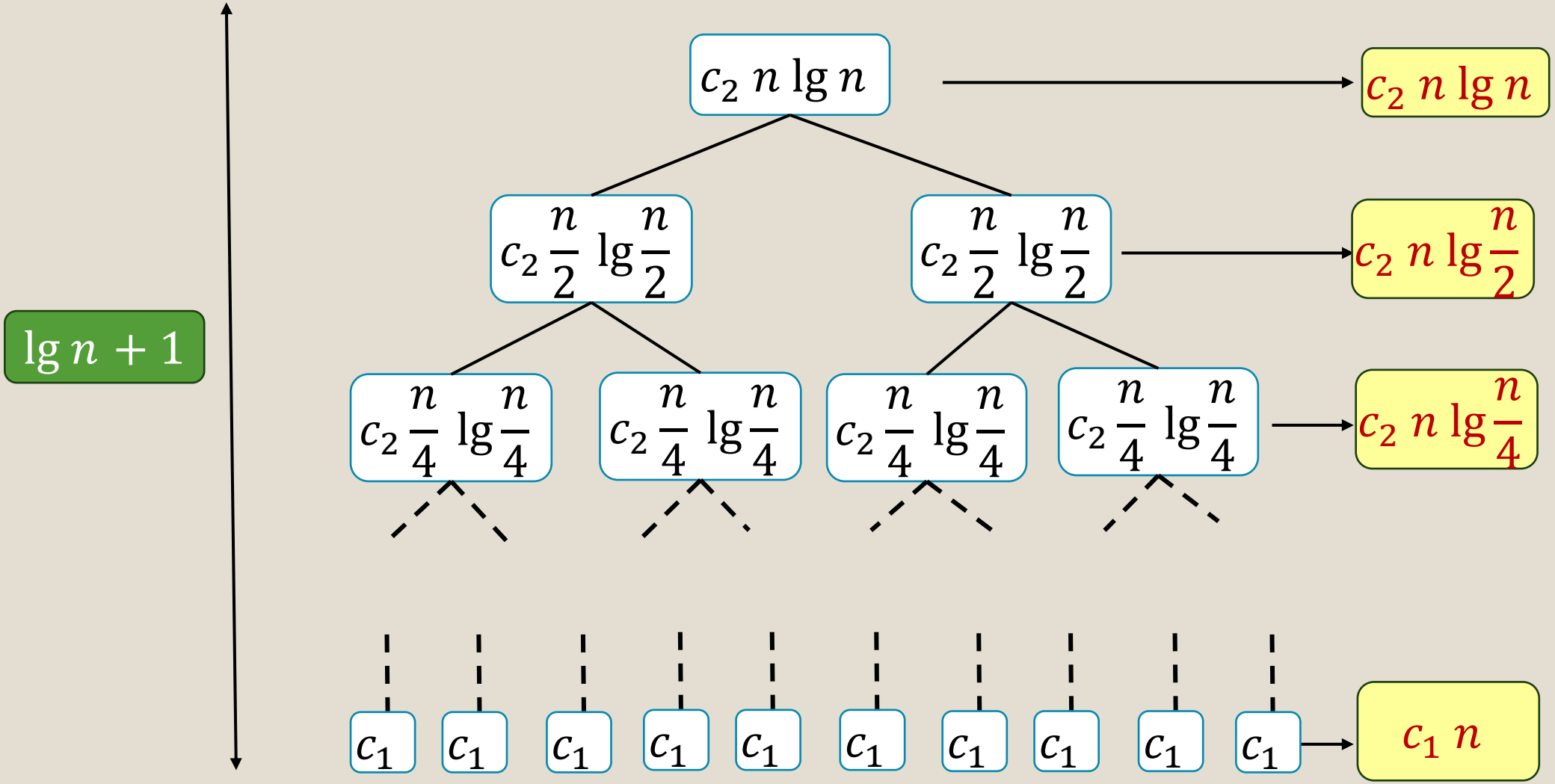
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 3 \\ 2T\left(\frac{n}{2}\right) + \Theta(n \lg n) & \text{if } n > 3 \end{cases}$$

Solving the recurrence,

$$T(n) = \Theta(n \lg^2 n)$$

How?

Running Time



Running Time

Assume $n = 2^y$

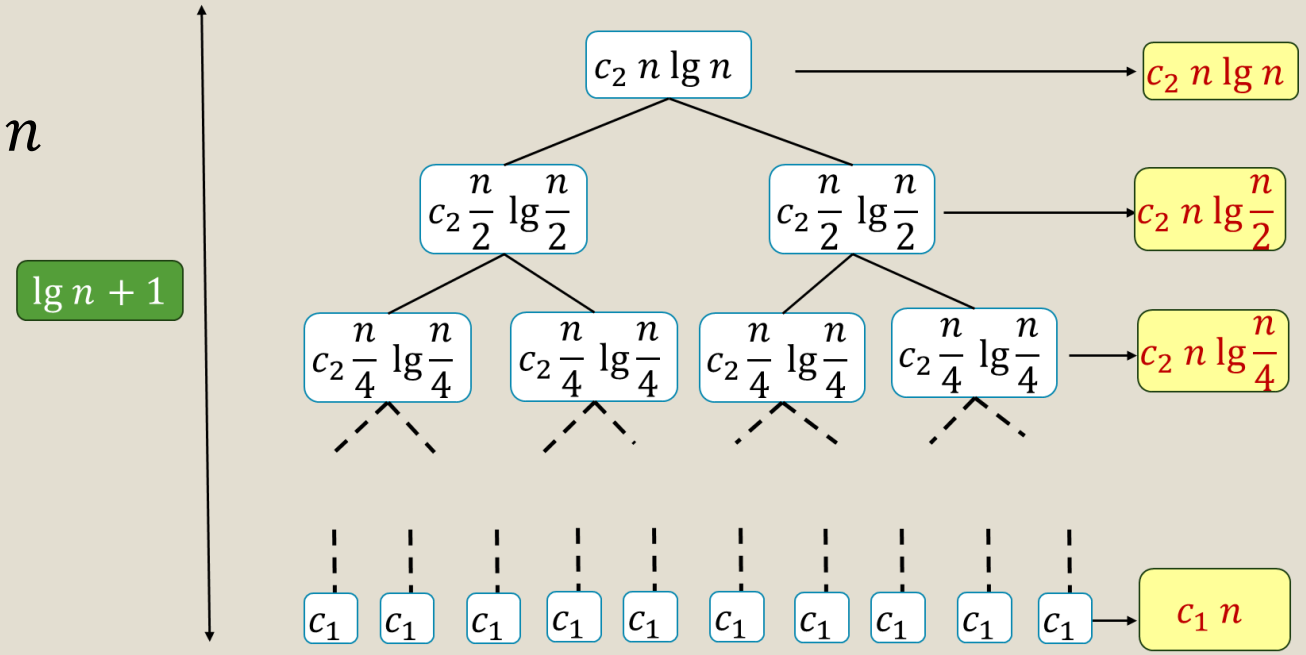
$$T(n) = c_2 n \cdot \left(\lg n + \lg \frac{n}{2} + \cdots + \lg 2\right) + c_1 n$$

$$= c_2 n \cdot ((\lg n + \lg n + \cdots + \lg n) - (1 + 2 + \cdots + 2^{y-1})) + c_1 n$$

$$= c_2 n \lg^2 n - c_2 n \frac{2^y - 1}{2-1} + c_1 n$$

$$= c_2 n \lg^2 n - c_2 n(\lg n - 1) + c_1 n$$

$$= \Theta(n \lg^2 n)$$



Running Time

Conclusion: Closest pair of points can be computed in $\Theta(n \lg^2 n)$ time

It is possible to improve the running time to $\Theta(n \lg n)$

Thank You!