Algorithm Design : Divide and Conquer

"Divide et impera "

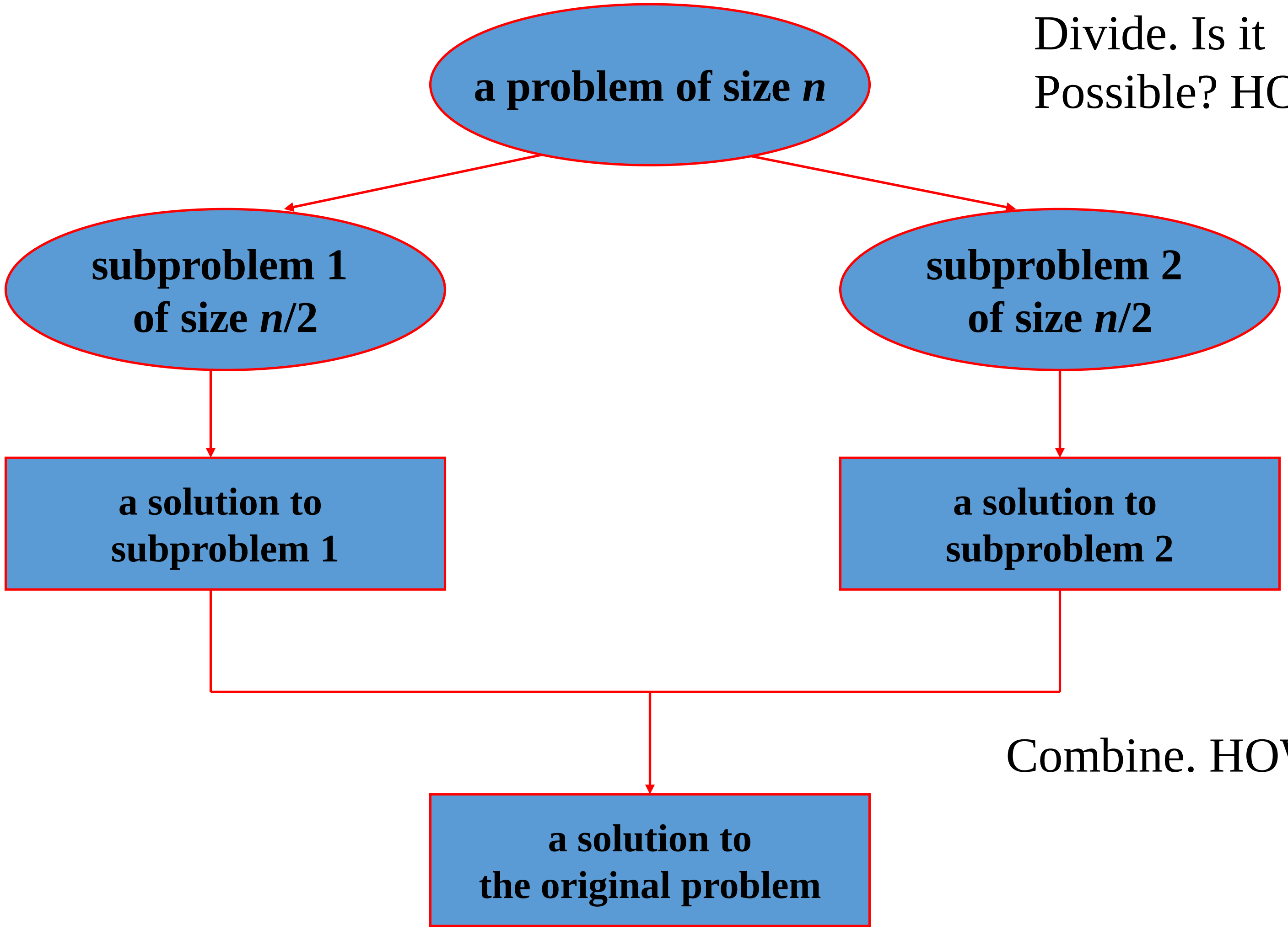
Divide and rule

Julius Caesar and Napoleon's favorite war tactic : Divide an opposing army in two halves and then assault one half with entire force.

Divide and Conquer

- *Divide-and-conquer* method for algorithm design:
  - If the problem size is small enough to solve it in a straightforward manner, solve it.
  - Else:
    - **Divide**: Divide the problem into two or more disjoint smaller *subproblems*
    - **Conquer:** Use divide-and-conquer recursively to solve the subproblems
    - **Combine**: Take the solutions to the subproblems and combine these solutions into a solution for the original problem

Divide-and-conquer Technique



Divide. Is it Possible? HOW?

Combine. HOW?

Solving Small and Large Instances

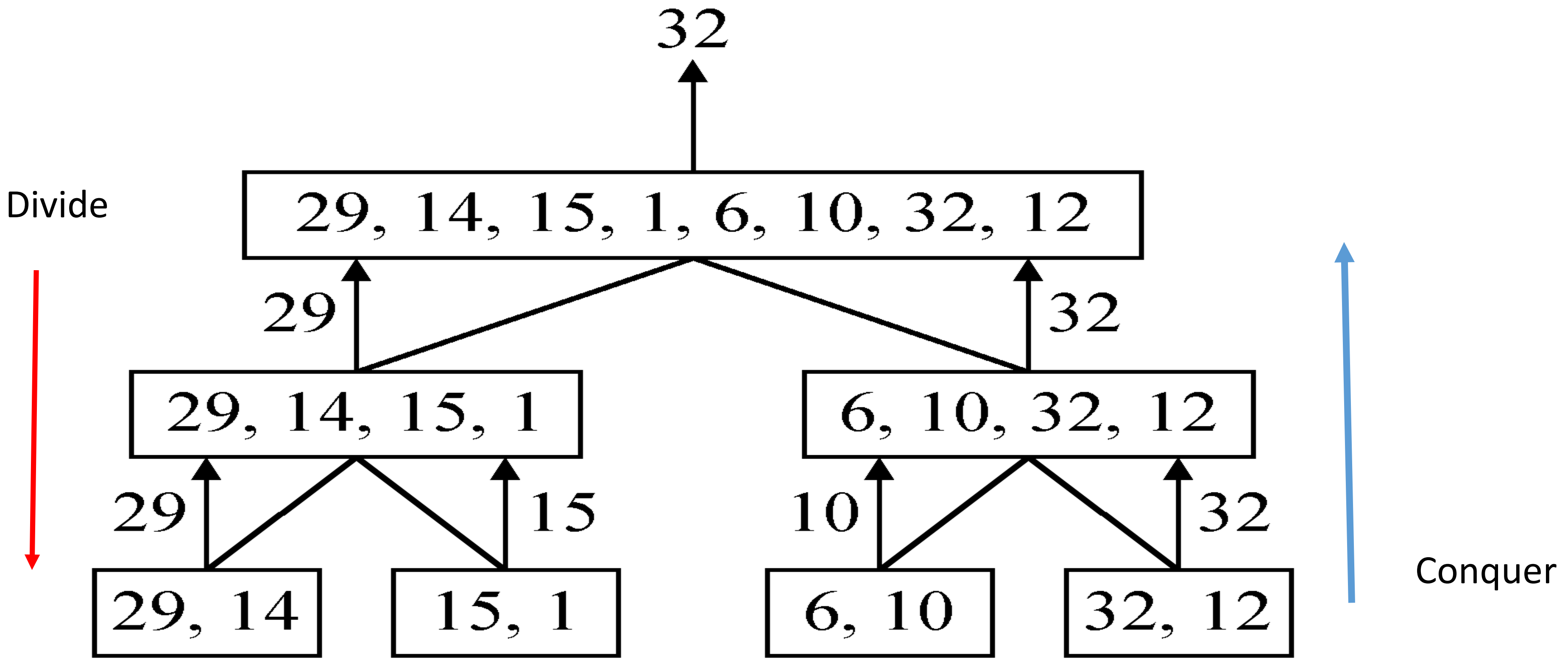
- Usually a small instance can be solved using some direct/simple strategy. Examples:
  - Sort a list that has say, n <= 10 elements.
    - Use insertion, bubble, or selection sort.
  - Find the minimum of n <= 2 elements.
    - When n = 0, there is no minimum element.
    - When n = 1, the single element is the minimum.
    - When n = 2, compare the two elements and determine the smaller.
- Find the minimum of n>2 elements.

→A large instance

We need a different solution method

A simple example : FindMax

- Finding the maximum of a set S of n numbers



Time complexity of FindMax : Recurrence

$$T(n)=\begin{cases} 2T(n/2)+1 & , n>2 \\ 1 & , n\le 2 \end{cases}$$

Assume $n = 2^k$. Use substitution

$$
\begin{aligned}
T(n) &= 2T(n/2)+1 \\
&= 2(2T(n/4)+1)+1 \\
&= 4T(n/4)+2+1 \\
&= 8T(n/8)+4+2+1 \\
&\;\vdots \\
&= 2^{k-1}T(2)+ 2^{k-2}+\ldots+4+2+1 \\
&= 2^{k-1}+2^{k-2}+\ldots+4+2+1 \\
&= 2^k-1 \qquad (\text{Using } 1+2+4+\ldots\ldots 2^{n-1} = 2^n - 1) \\
&= n-1 \qquad (\text{Since } n = 2^k) \\
&= O(n)
\end{aligned}
$$

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary Search
- Binary tree traversals
- Graph applications
- Big Integer Multiplication
- Closest Pair of Points Problem
- Strassen’s Algorithm for Matrix Multiplication
- …………………..

Reminder : Mergesort

Merge-sort on an input sequence $\boldsymbol{A}$ with $\boldsymbol{r}$ elements consists of three steps:

Divide: partition $\boldsymbol{A}$ into two sequences $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ of about $\boldsymbol{r}/2$ elements each

Conquer(Recur): recursively sort $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$

Combine: merge $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ into a unique sorted sequence

Reminder : Recursive Merge Sort

//p,r : first and last index values. Initially: p=1, r=n

MERGE_SORT(A,p,r)

if p<r

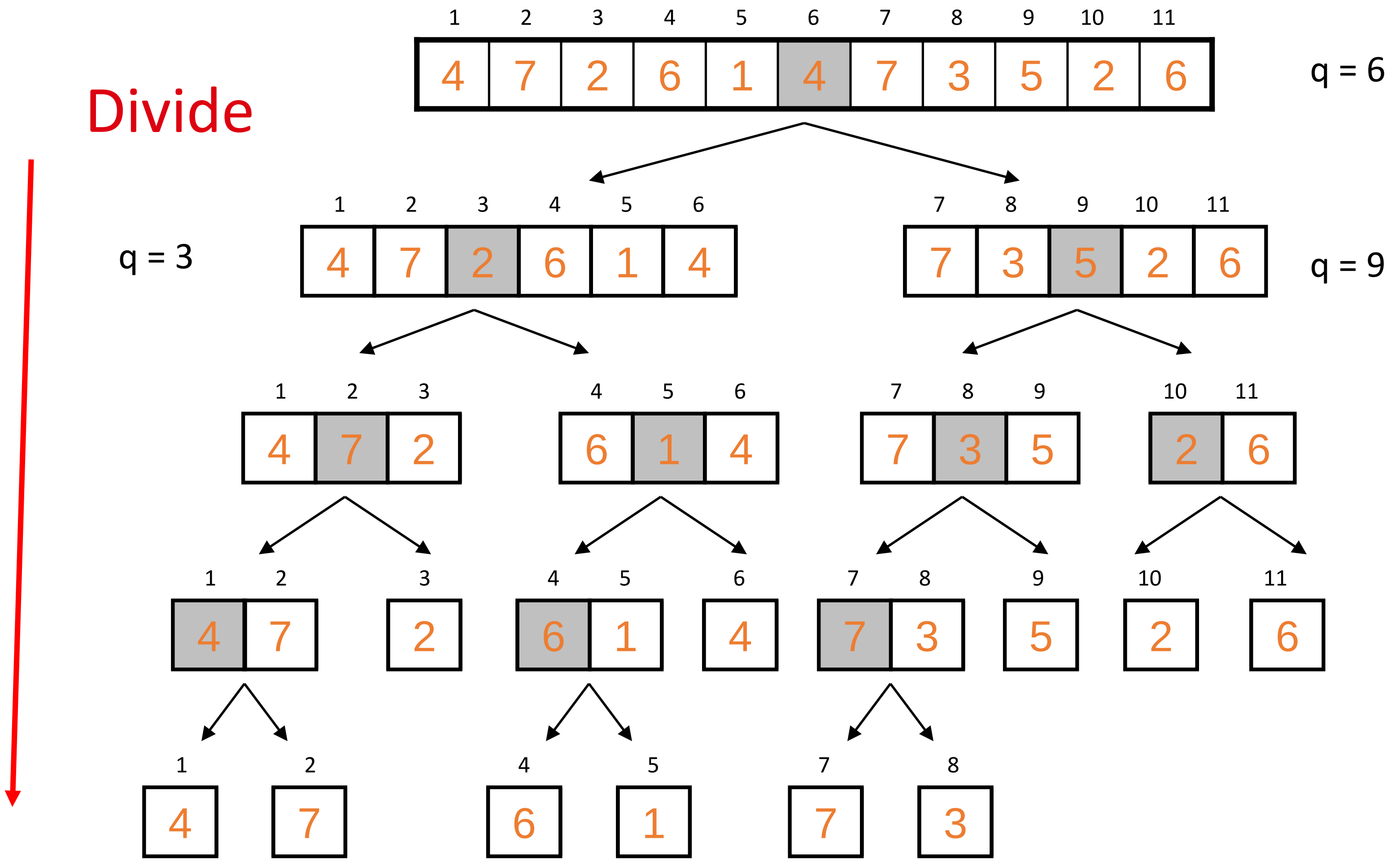
    then $q \leftarrow (p+r)/2$

        MERGE_SORT(A,p,q)

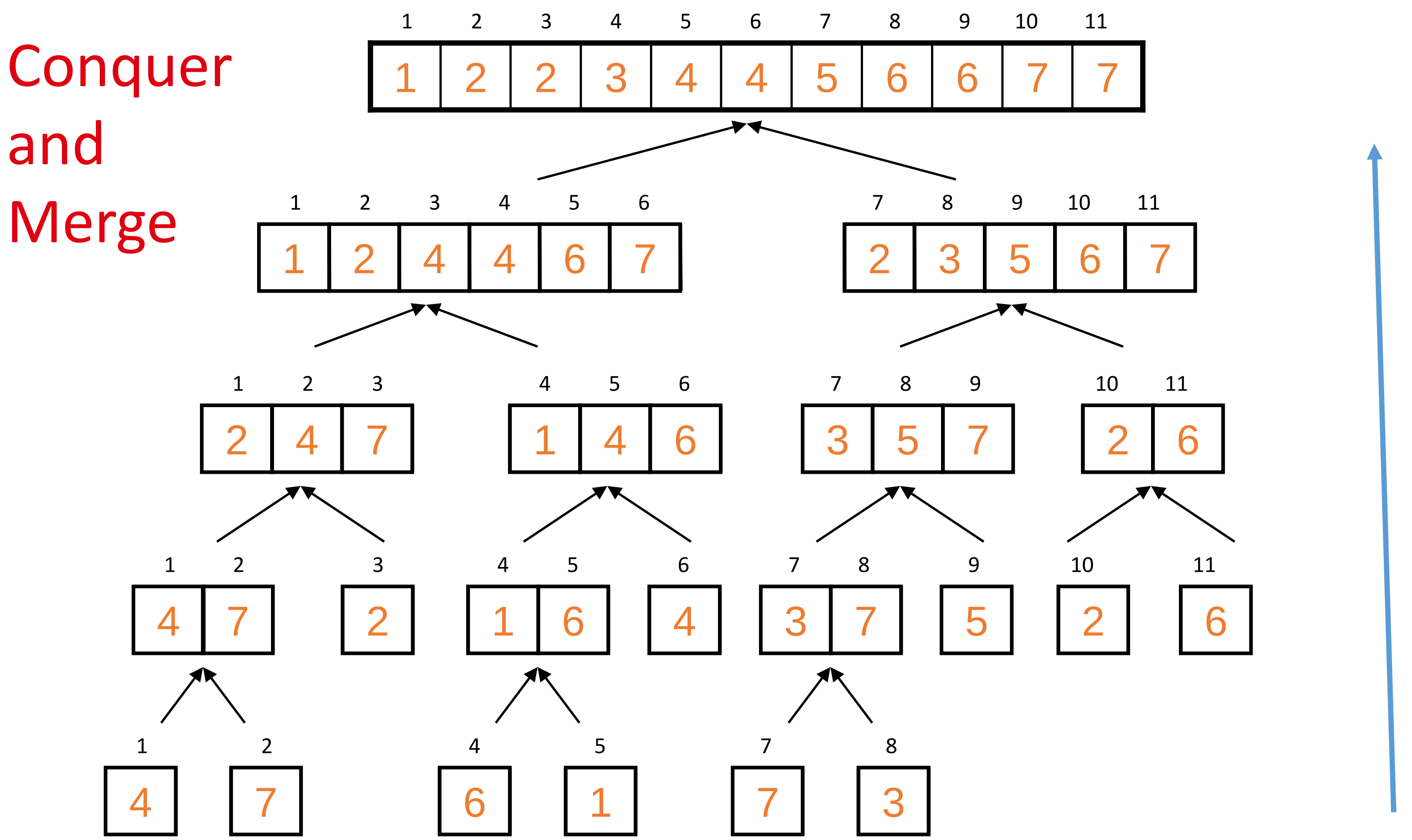
        MERGE_SORT(A,q+1, r)

        MERGE(A,p,q,r)

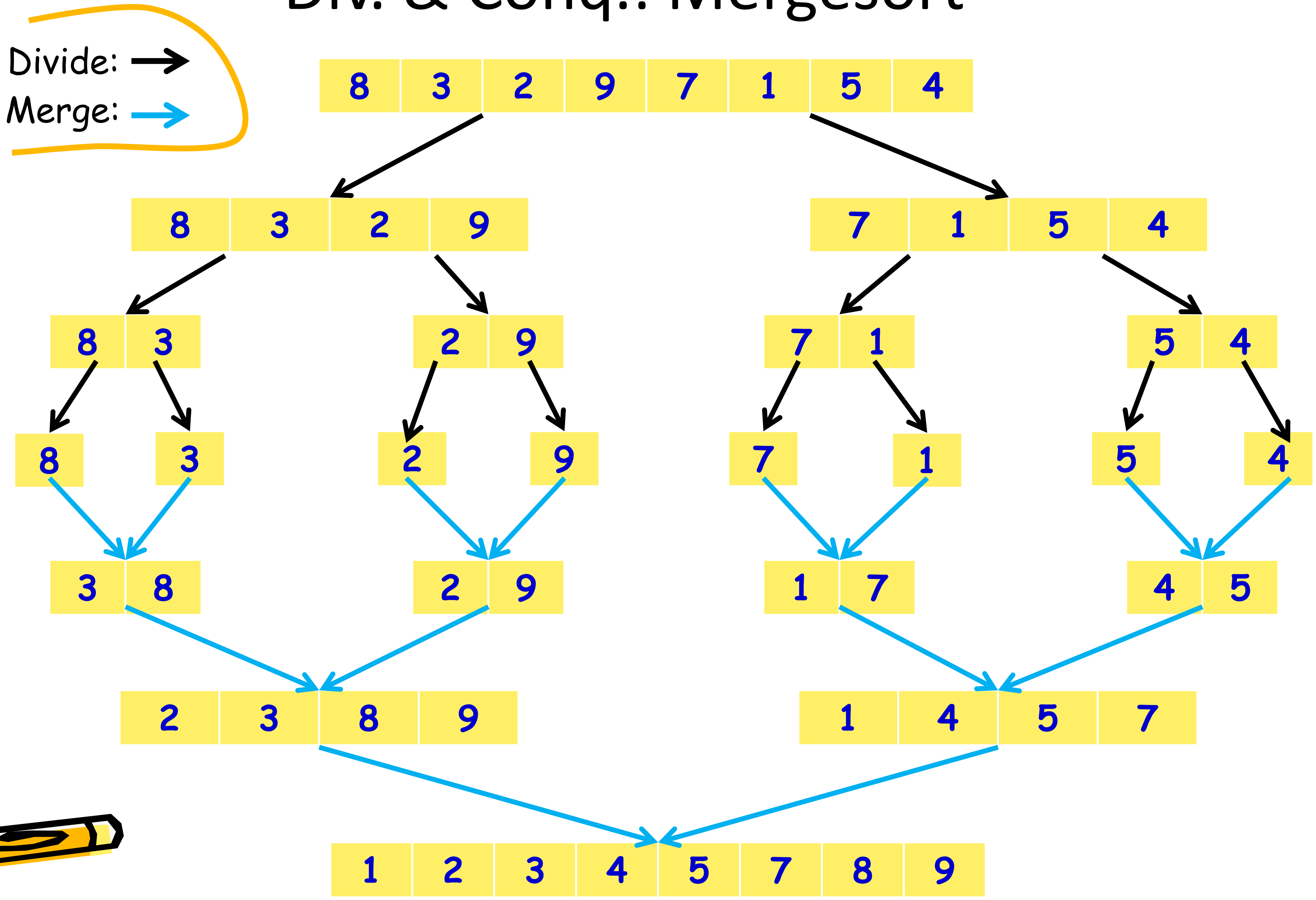
Example (n Not a Power of 2)



Example （ n Not a Power of 2 ）



Div. & Conq.: Mergesort



Running time:Recurrence Equation

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(n)$$

$$= 2T(\frac{n}{2}) + O(n)$$

(O(n) is for merge operations)

Assume that: $n = 2^k$

$$T(1) = 1$$

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

Running Time : Solving the Recurrence

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

$T(n) = 2[2T(n/4)+n/2] + n$

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

$\quad = 4[2T(n/8)+n/4] + 2n$

$\quad = 8T(n/8) + 3n$

....

$T(n) = 2^k T(n/2^k) + k\, n$ (After kth derivation)

But, $n = 2^k$ and $k = \log n$. By replacing the value for k we find:

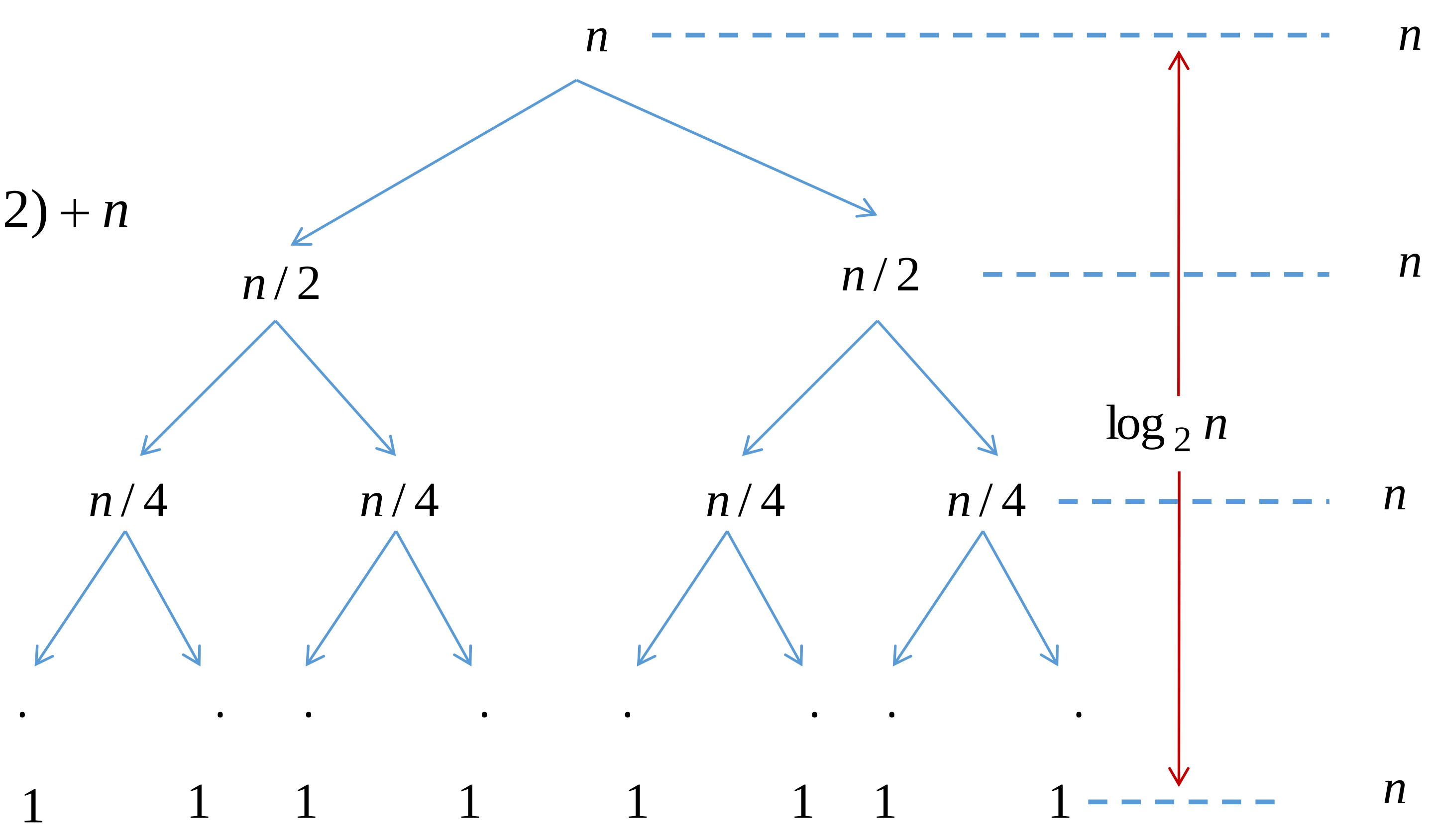
$T(n) = n\, T(1) + n\log n$

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

$\quad = \Theta(n \log n)$

Running Time : Recursion Tree Method

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



$$Total = n\log_2 n$$
$$T(n) = O(n\log_2 n) = O(n\lg n)$$

Merge sort : Discussion

- Merge sort is a Divide-and-conquer algorithm
  and its worst case complexity is : $O(n \log n)$
- The same operations are done regardless of input order

→All cases have the same complexity

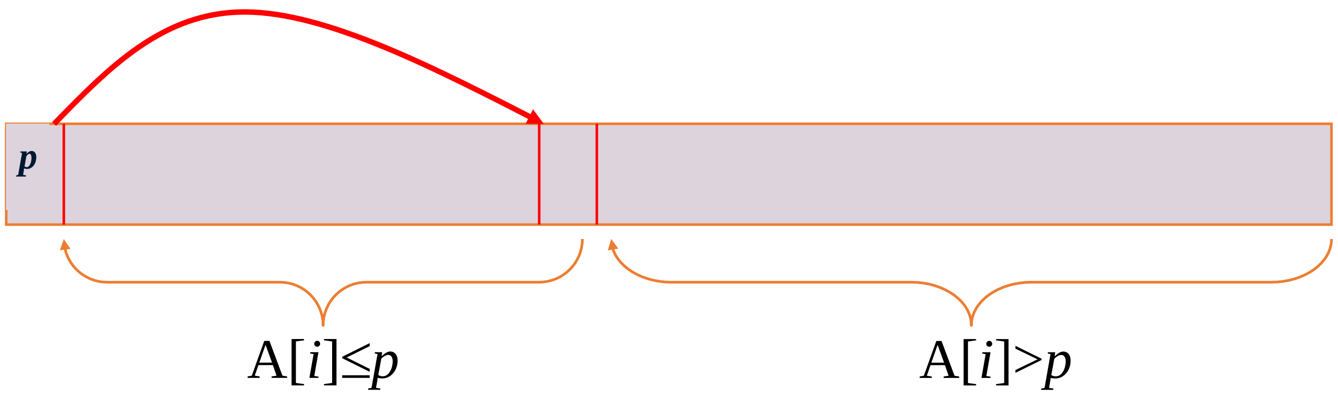
So the complexity is :

$T(n) = \Theta(n \log n)$

- Storage: Copying to and from temporary array
  - Extra memory requirement
  - Extra work

Reminder : Quicksort

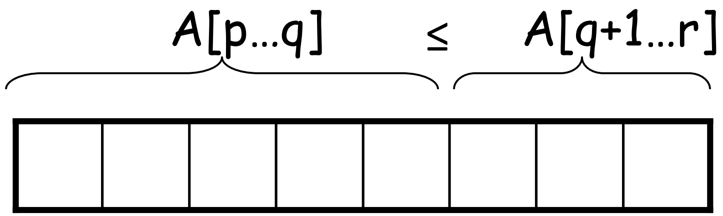
- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first ***s*** positions are smaller than or equal to the pivot and all the elements in the remaining ***n-s*** positions are larger than or equal to the pivot



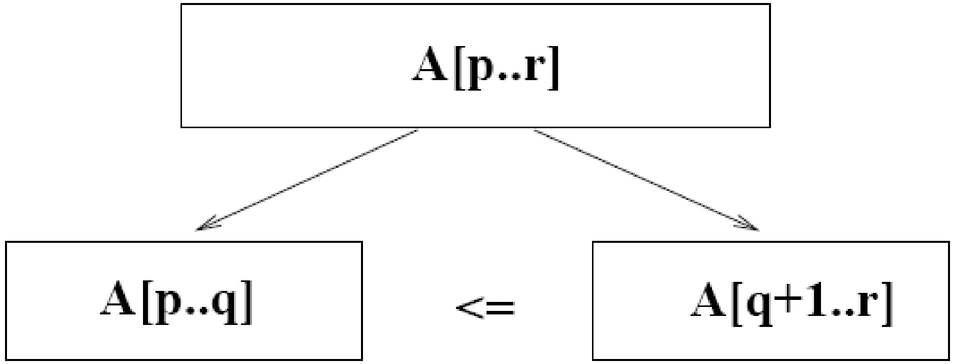
- Exchange the pivot with the last element in the first (i.e., ≤) sub-array — the pivot is now in its final position
- Sort the two sub-arrays recursively

→Divide and conquer solution

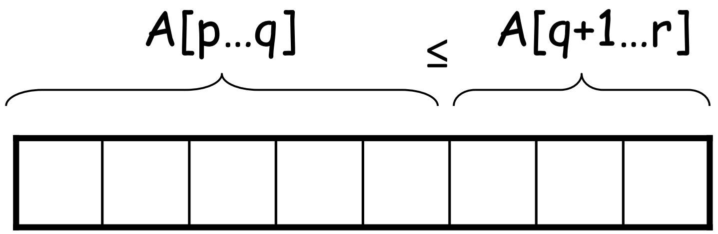
Quicksort : Divide



- Sort an array $A[p \ldots r]$
- Divide
  - Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
  - Need to find index q to partition the array



Quicksort :Conquer and Combine



- Conquer
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- Combine
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

Quicksort: Pseudocode

```
// lo:lower index,hi: higher index
Algorithm quicksort(A, lo, hi)
    if lo < hi then
        p ← partition(A, lo, hi)
        quicksort(A, lo, p – 1)
        quicksort(A, p + 1, hi)
```

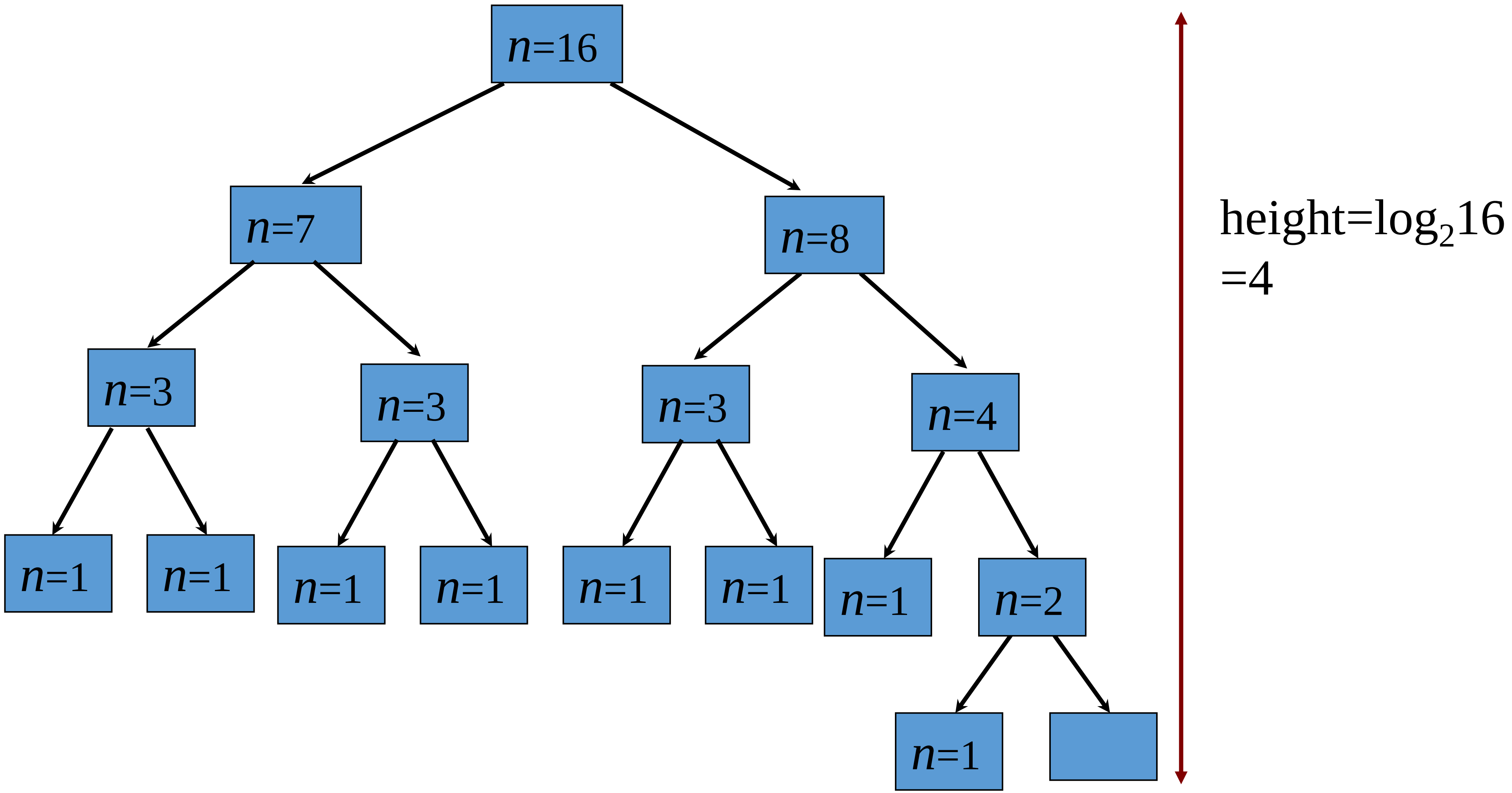
Quicksort : Partitioning

Algorithm partition(A, lo, hi)

```
    pivot ← A[hi]
    i ← lo - 1
    for j = lo to hi - 1 do
        if A[j] ≤ pivot then
            i ← i + 1
            swap ( A[i]  A[j] )
    swap ( A[i+1]  A[hi] )
    return i + 1
```

The complexity is linear : $T(n) = O(n)$

Best-Possible Partitioning : Illustration



Worst-Possible Partitioning

Pivot is always smallest element . There are Two cost factors:

- The partitioning cost for N items (+ a smaller number of exchanges, which may be ignored)
- The cost for recursively sorting the remaining n−1 items.

A simple argumentation : We are selecting the first element as pivot and the pivot divides the list of size N into two sublists of sizes 0 and N-1

The number of key comparisons :

$$T(N) = N-1 + N-2 + \ldots + 1$$

$$= N^2/2 - N/2$$

$$= O(N^2)$$

Quick Sort: Worst Case Analysis

Worst case occurs when pivot always splits the array into two subarrays of length N-1 and 0.

Partition is always unbalanced

Recurrence is :

$T(1)=1$

$T(N)=T(N-1) + N$

Worst-Case Analysis : Solving the Recurrence

$T(N) = T(N-1) + N$ // $T(N-1) = T(N-2) + (N-1)$

$= T(N-2) + (N-1) + N$

$= T(N-3) + (N-2) + (N-1) + N$

$= \ldots$

$= T(1) + 2 + 3 + \ldots + (N-1) + N$ // $T(1) = 1$

$= 1 + 2 + 3 + \ldots + (N-1) + N$ //Sum of integers

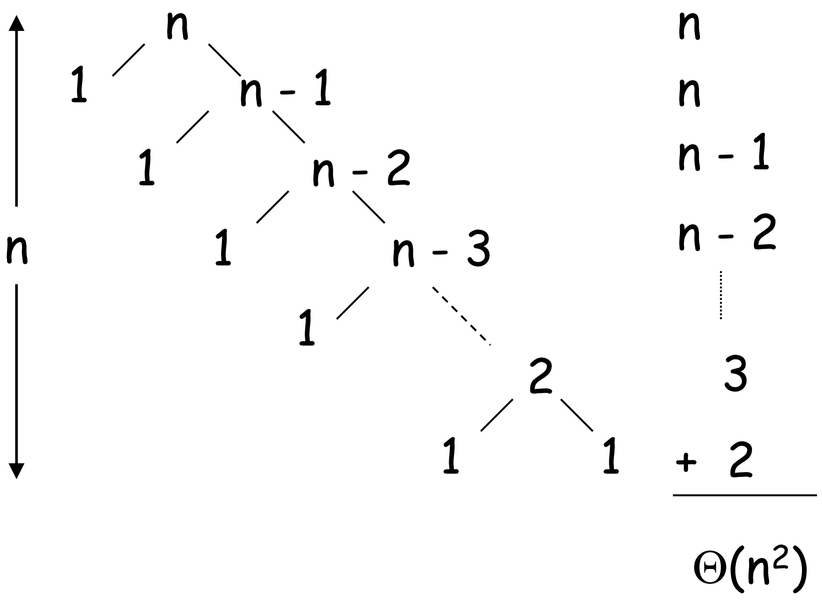
$= N(N+1)/2$

$= O(N^2)$

$= \Theta(N^2)$

Worst Case : Recursion Tree

- Worst-case partitioning
  - One region has one element(The pivot) and the other has n – 1 elements
  - Maximally unbalanced



Total number of comparisons=

Best-case Analysis

- Partition is perfectly balanced.
- Pivot is always in the middle (median of the array)
- Thus, the recurrence relation in this case is the same as the recurrence of merge sort (See slides 13-14) :

  $T(1) = 1$

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

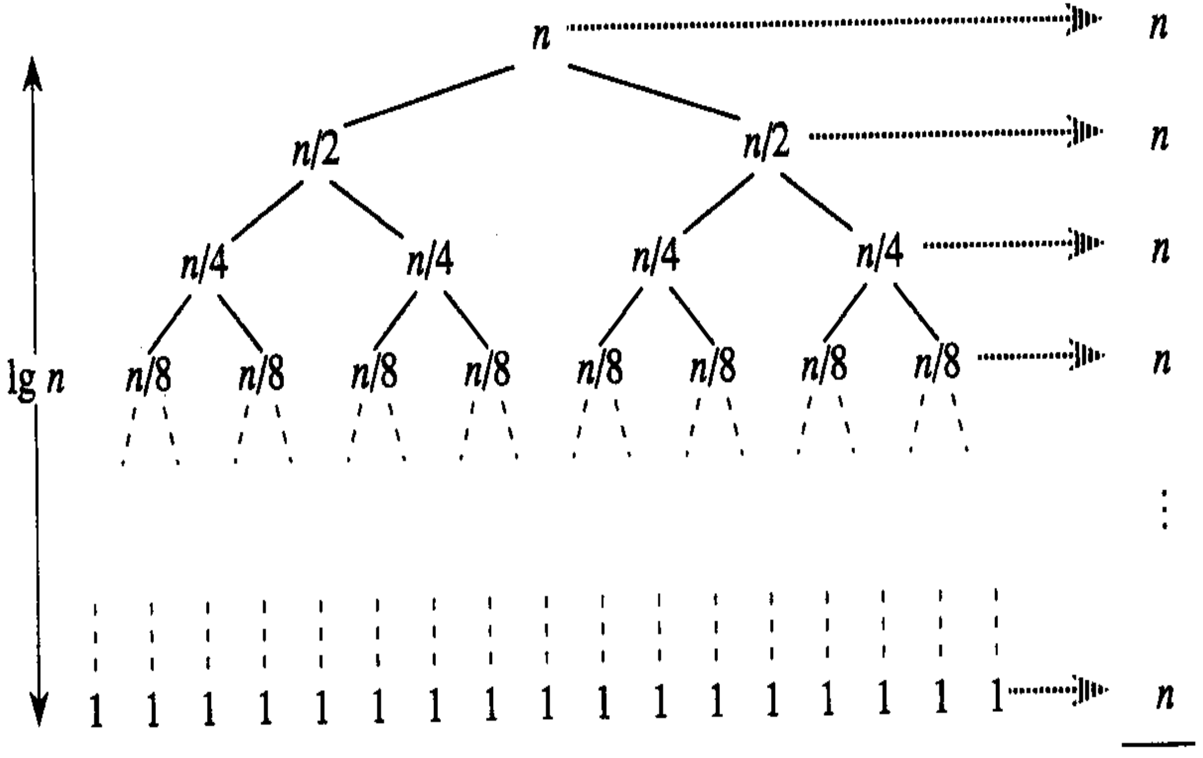
The solution is :

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

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

Best Case : Recursion Tree

- Best-case partitioning
  - Partitioning produces two regions of size ~ n/2



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

Average case analysis

- The average case running time analysis is close to the best case.

Example : Suppose that the partitioning always produces a 9 to 1 split. For such a case the running time is

$$T(n) = T(9n/10) + T(n/10) + O(n)$$

- It can be shown that the solution for this recurrence is

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

which is the same as the best case.

Note:A detailed analysis shows that the average case has a coefficient 1.4 :

T(n) ~ 1.4 O(nlogn)

QuickSort Performance : Discussion

- QuickSort is not guaranteed to be more efficient than Insertion Sort or others because of its worst case performance:
  - if it makes an unlucky choice for the pivot the array will not be divided equally → $O(n^2)$ complexity
  - However, many tests on real-world data show that QuickSort is very effective in practice and it is a popular choice in many applications.

Is QuickSort Faster than Merge Sort?

- Quicksort typically performs more *comparisons* than Mergesort, because partitions are not always perfectly balanced
  - Mergesort : n log n comparisons
  - Quicksort : c* n log n comparisons on average (c >1)
- However, Quicksort performs many fewer *copyings*, because on average half of the elements are on the correct side of the partition – while Mergesort copies *every* element when merging.

Divide and Conquer :Binary Search

- Binary Search is also a divide and conquer algorithm :
  - Divide : Split the list around the mid point
  - Conquer :Recursively Search in each half
  - Combine :Each time remove half of the list from search

(Remember that the complexity of Binary Search is log n)

Binary Tree Algorithms: Traversals

Binary tree is a divide-and-conquer ready data structure by definition.

"Any node n of a binary tree B defines a binary tree "

Classic traversals : preorder, inorder, postorder

These are Divide and Conquer methods.Example:

Algorithm *Inorder*(*T*)

    if $T \neq \varnothing$

        Inorder(*Tleft*)

        process(root of *T*)

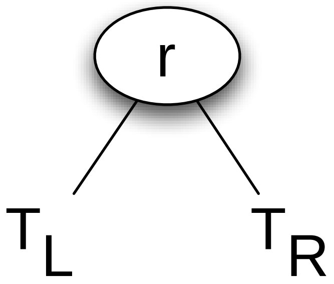
        Inorder(*Tright*)

→Traversal moves down by splitting the subtrees.

Complexity : $\Theta(n)$

Binary Tree Algorithms : Height of Binary Tree

- The height of a binary tree T can be defined recursively as:
  - If T is empty, its height is 0.
  - If T is non-empty tree, then since T is of the form :



→height of T is 1 greater than height of its root's taller subtree:

height(T) = 1 + **max**{height(T_L),height(T_R)}

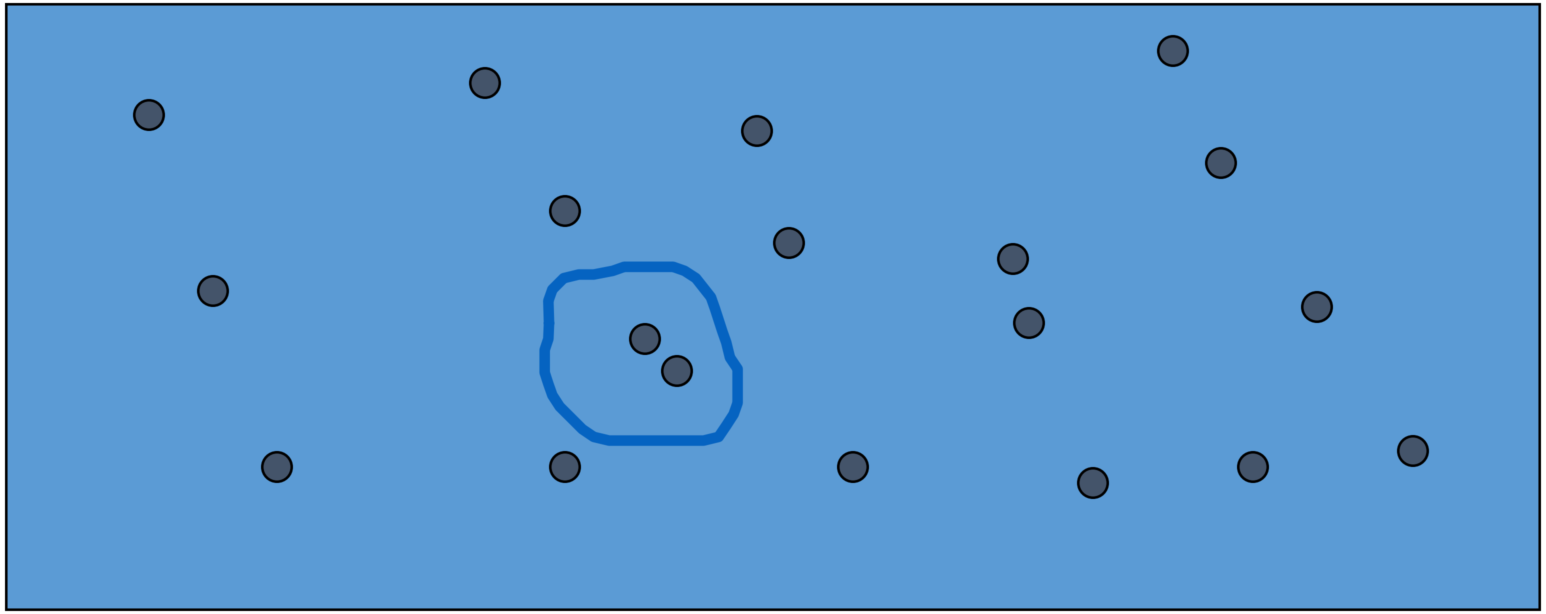
Complexity: $\Theta(n)$

Height of Binary Tree : Divide and Conquer

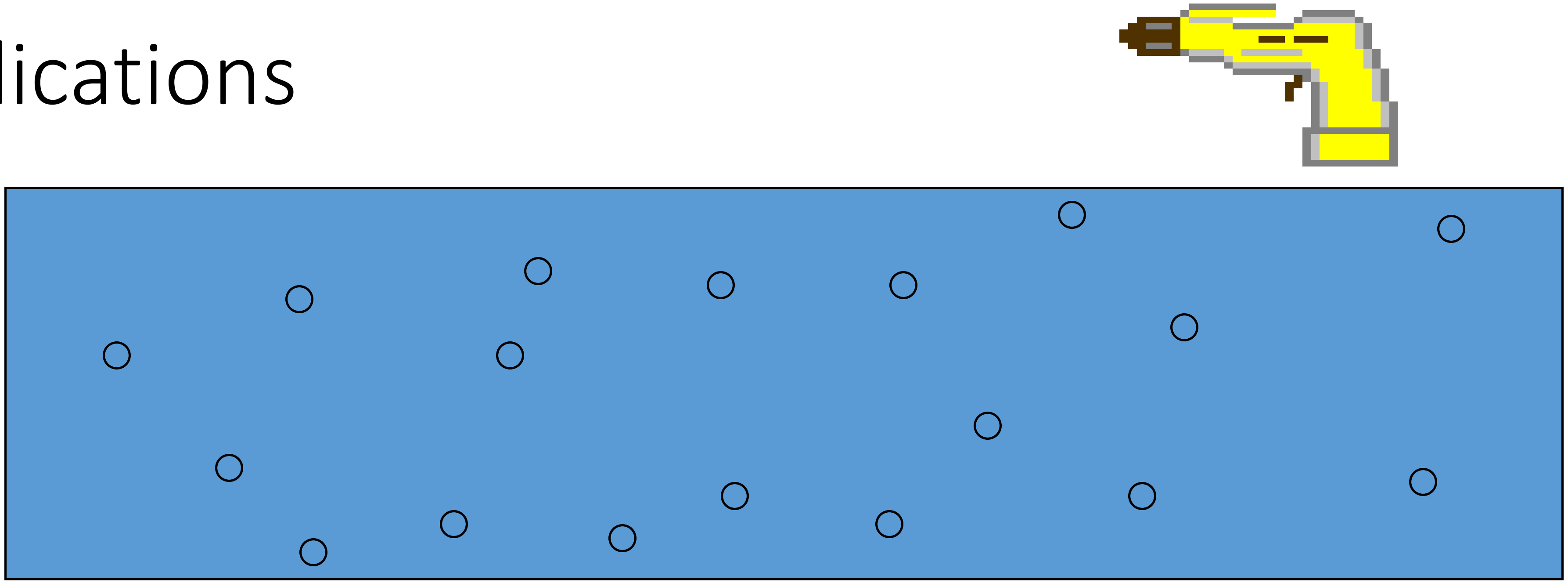
```
//Returns the height h of binary tree T
 Algorithm Height( T )
    If T ≠ ∅
     hleft ← height( leftSubtree T)
     hright ← height( rightSubtree  T)
      h ← 1 + max( hleft , hright)
 return h
```

Closest Pair Of Points

- Given n points in 2D, find the pair of points that are closest.

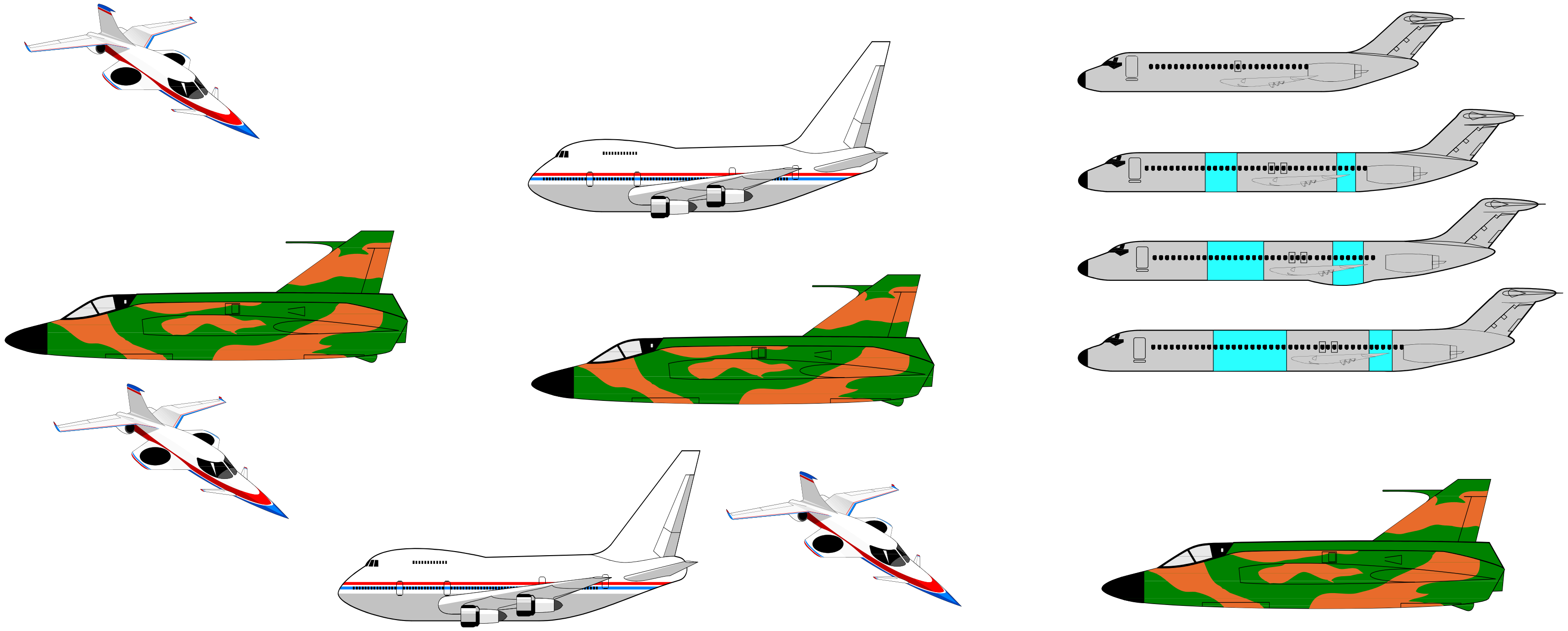


Applications



- We plan to drill holes in a metal sheet.
- If the holes are too close, the sheet will tear during drilling.
- Verify that no two holes are closer than a threshold distance (e.g., holes are at least 1 inch apart).

Air Traffic Control



- 3D -- Locations of airplanes flying in the neighborhood of a busy airport are known.
- To avoid crash,make sure that no two planes get closer than a given threshold distance.

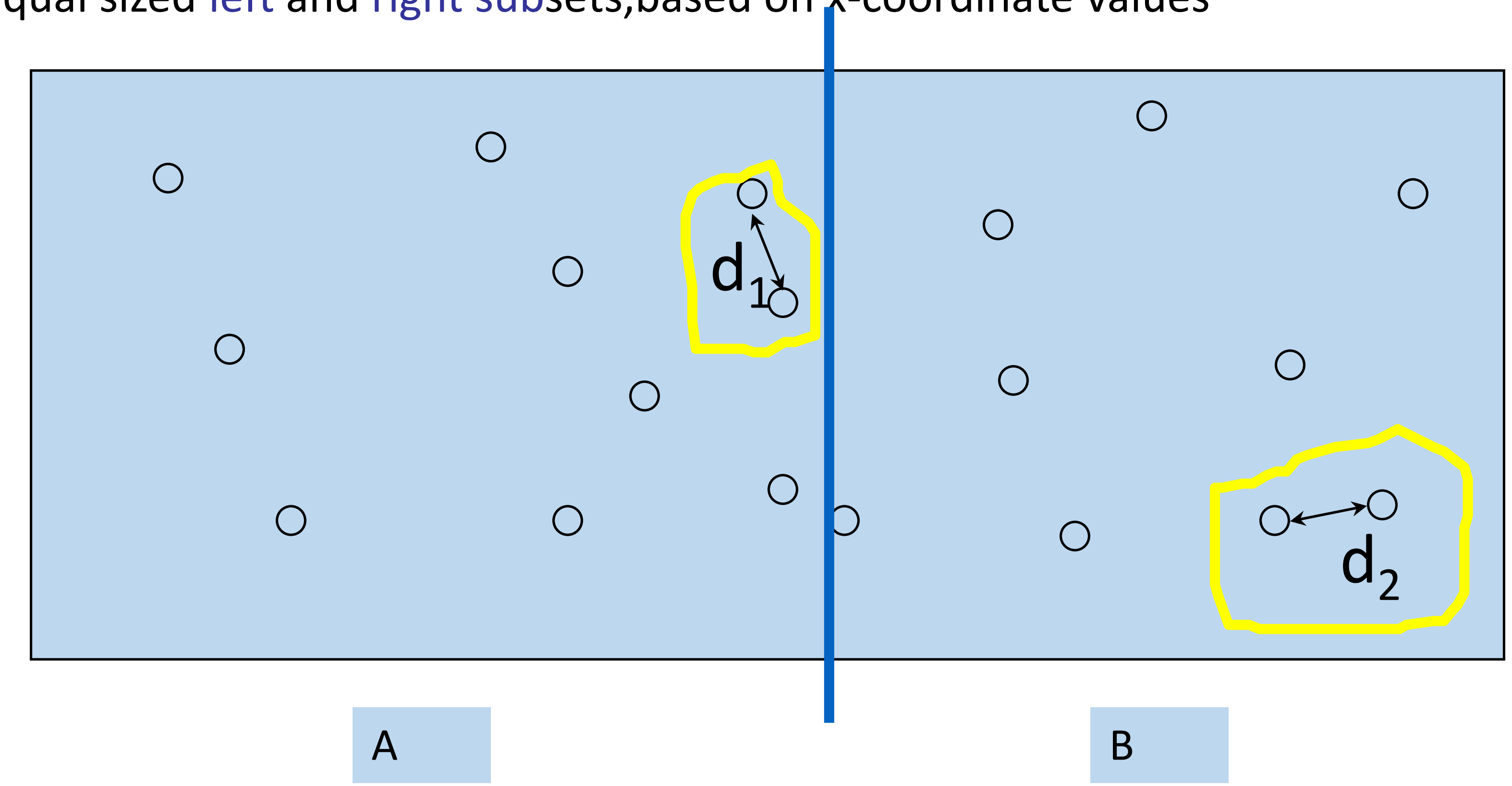
Divide-And-Conquer Solution

- When n is small, use brute force solution : $O(n^2)$
- When n is large
  - Divide the point set into two roughly equal parts A and B.
  - Conquer
    - Determine the closest pair of points in A.
    - Determine the closest pair of points in B.
  - Combine : Determine the closest pair of points such that one point is in A and the other in B.
  - From the closest pairs computed, select the one with least distance.

Closest pair in the plane

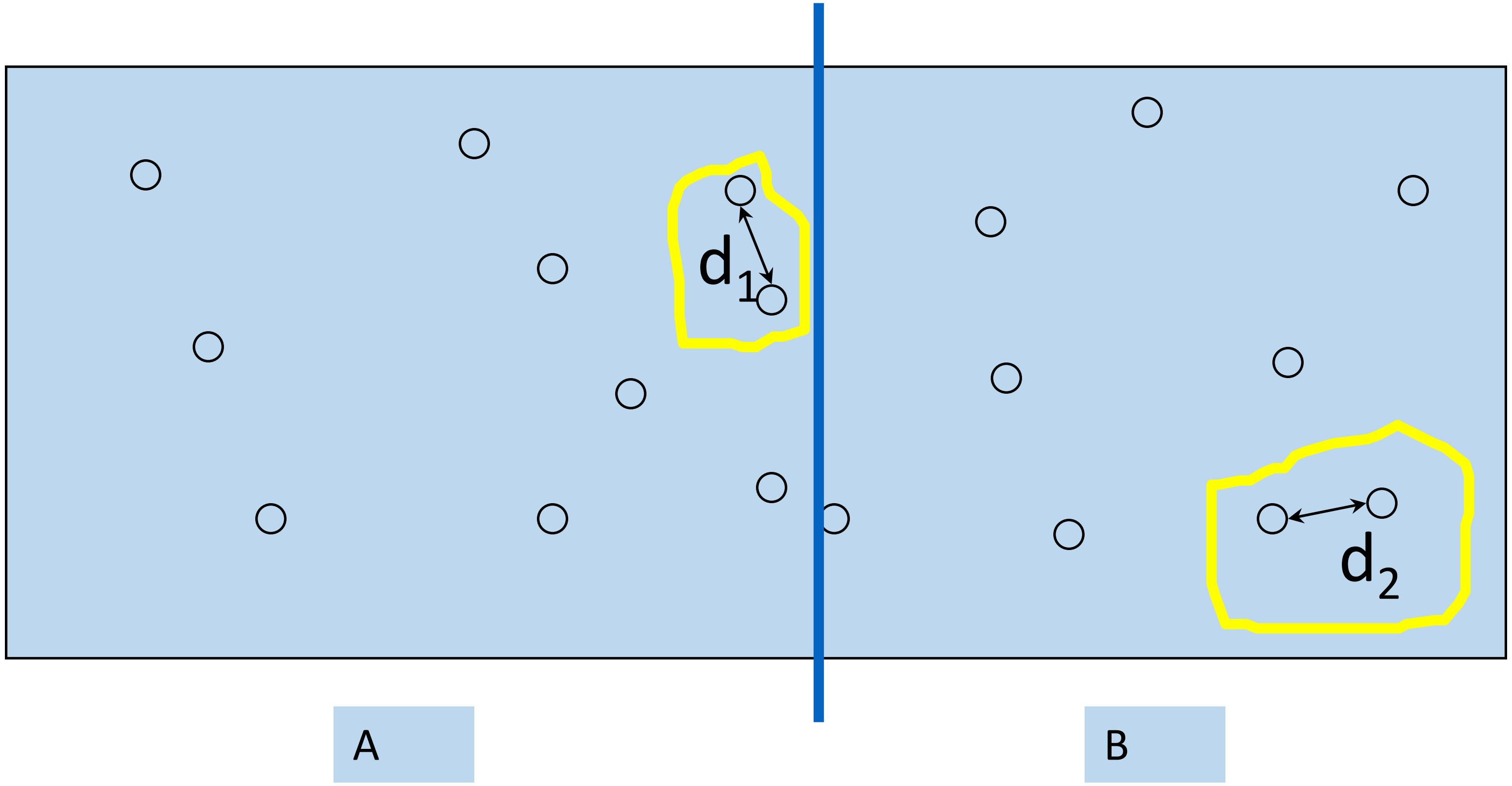
Divide and conquer approach:

split point set in equal sized left and right subsets,based on x-coordinate values



- Find closest pair distances in A and B as : d_1 and d_2

Closest pair in the plane



Let d = min {d1, d2}. Is d the minimum distance ? Not necessarily.

The other possibility?

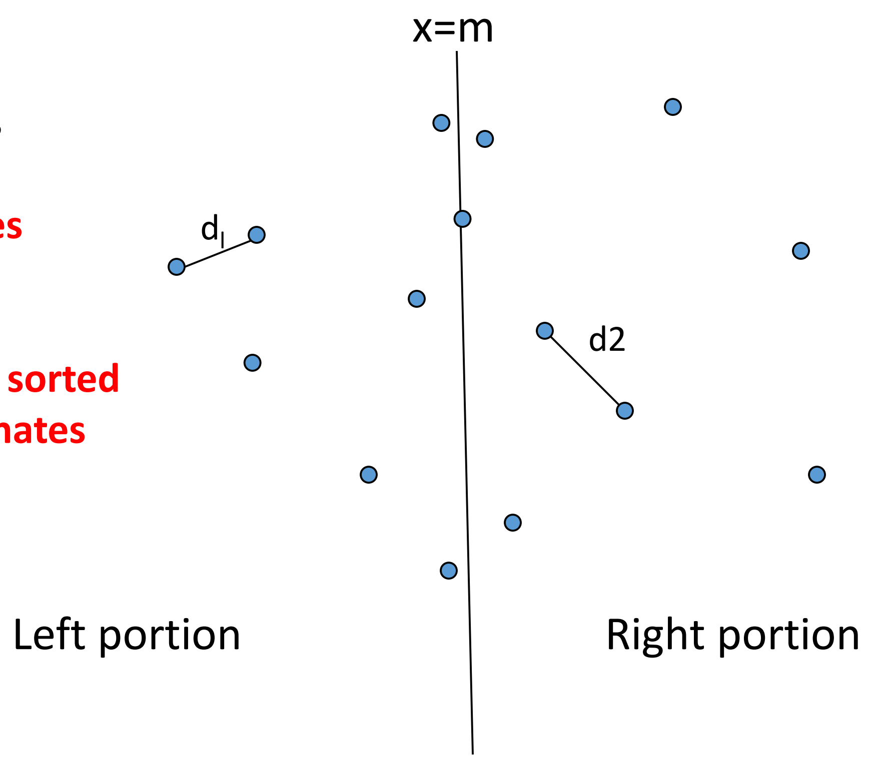
There may be a pair with one point in A, the other in B and their distance $< d$?

Closest pair in the plane : Illustration

How to divide and then conquer?

Let P be the set of points sorted by x-coordinates

let Q be the same points sorted by y-coordinates



Solve right and left portions recursively and then combine the partial solutions

How should we combine?

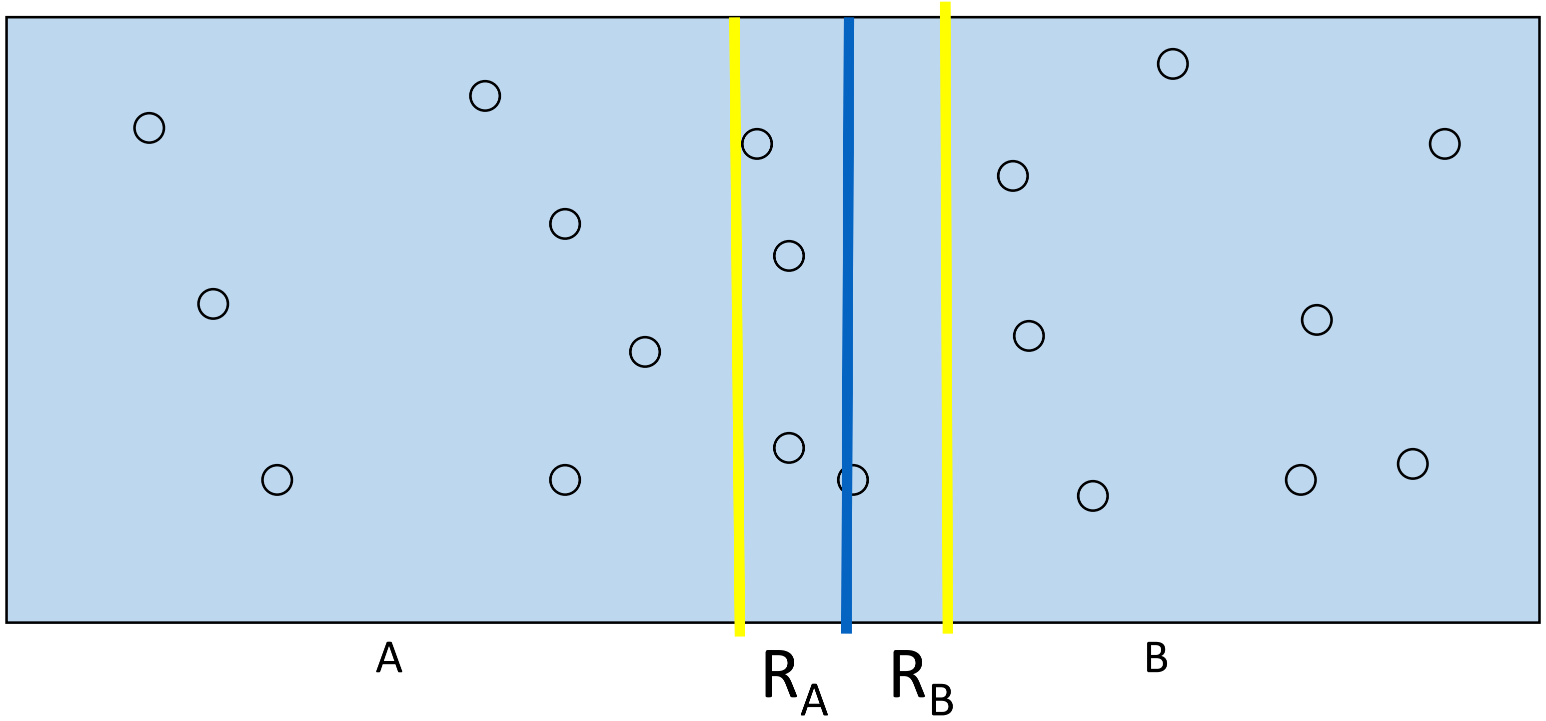
$d = \min \{d_l, d_2\}$?

Does not work! because one point can be in left portion and the other could be in right portion having distance < d between them…

Closest pair in the plane

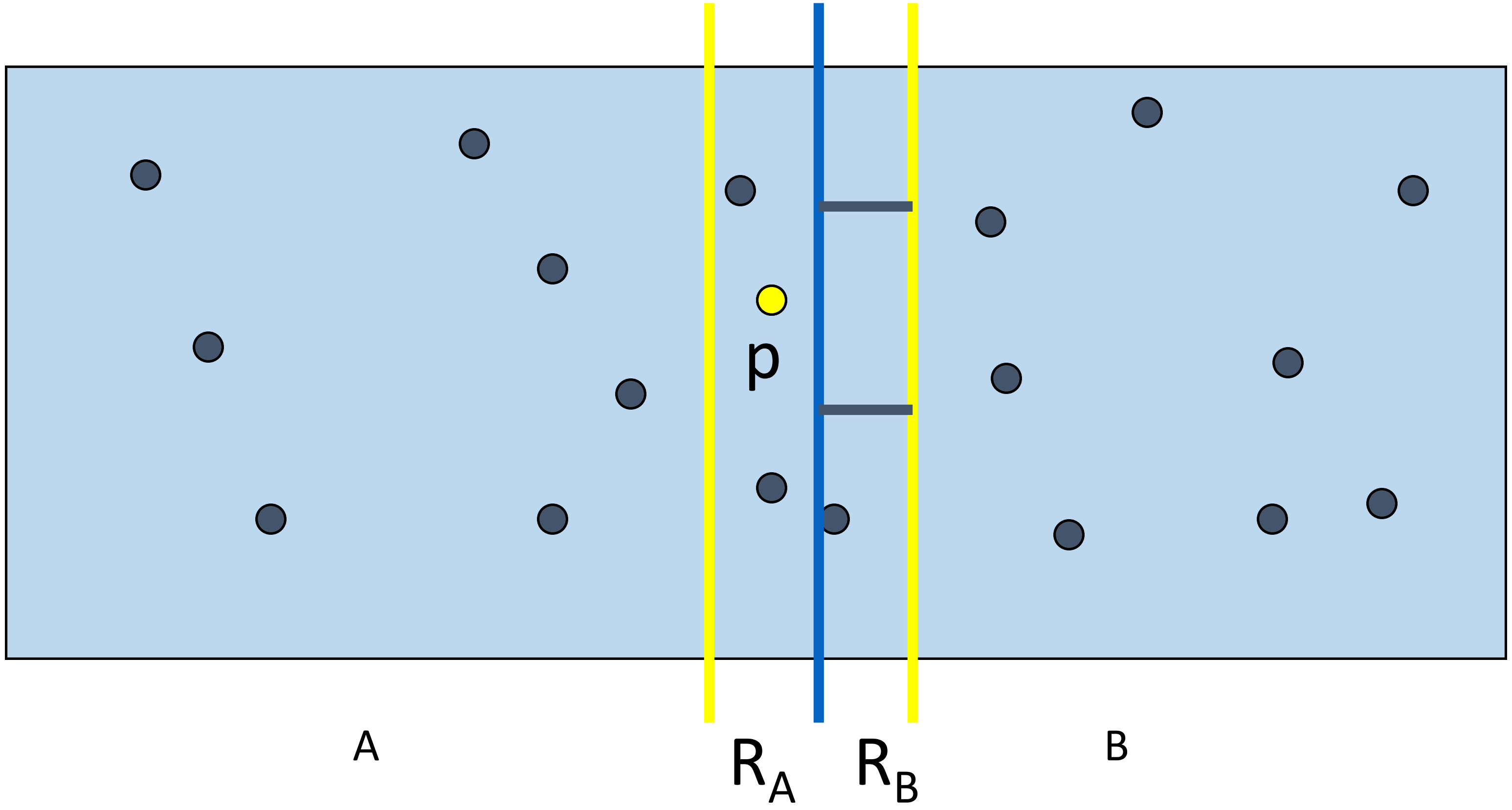
Candidates must lie within d of the dividing line. Call this region M.

M includes left and right parts: R_A and R_B .



We have to limit the search to the points in the strip M since the distance will be > d for all pairs outside this region.

Example

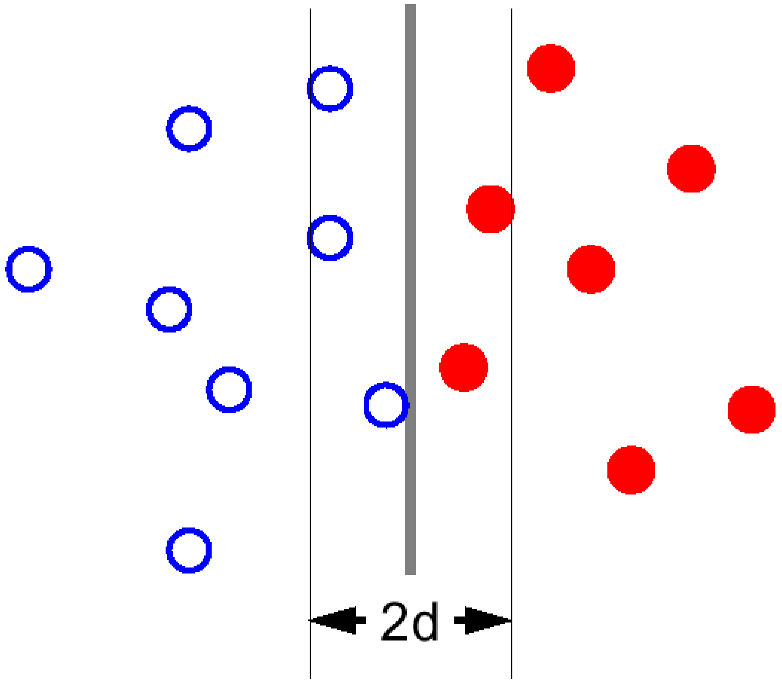


- Let p be a point in this region.
- p need be paired only with those points in RB that are at most d apart. Distance to all other points will be >d.

Continue recursively splitting both parts until one pair remains.

Combining two Solutions

- *Combining solutions*: Finding the closest pair in a strip of width 2*d*, knowing that no left or right pair is closer than *d.*
- *Sort the points according to y values and compute deiğstances.*
- *The key point here is that we don't have to check all pairs ,but a max. of 7 pairs.(Proof is omitted!).This reduces complexity down from O(n^2) to 7n.*



Closest pair in the Plane : Informal Algorithm

Closest-Pair (P: set of n points in the plane)

sort by x coordinate and split equally into L and R subsets

Find closest pairs in both regions

(p , q) = Closest-Pair(Left)

(r , s) = Closest-Pair(Right)

d = min (distance(p , q), distance(r , s))

Find closest pairs across regions around midline

scan p by x coord. to find strip M : points within d of midline

sort points in M by y coordinate.

compute closest pair among all pairs in M

//For each point at most 7 comparisons!

Return best among (closest pair, (p,q), (r,s))

Time Complexity

A simplified argumentation:

- Let $T(n)$ be the time to find the closest pair (excluding the time to create the two sorted lists).
- Simple case : $T(n) = c$, $n < 4$, where c is a constant,use Brute force.
- When $n >= 4$:

Divide : into two subsets (according to x-coordinate) :

$P_L <= l <= P_R$ (O(n))

Conquer: Find closest recursively on each half : $2T(n/2)$

Combine: select closest pair of the above two and the region M.

Time Complexity

So that ,the recurrence is

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

(Assumes that sorting x and y axis values are done only once)

- To solve the recurrence, assume n is a power of 2 and use repeated substitution as we did in previous examples :

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

$= O(n \log n)$

Divide and Conquer :Performance Summary

- The divide and conquer strategy often reduces the number of iterations of the main loop from *n* to $\log_2 n$
  - binary search: $O(\log_2 n)$
  - merge sort: $O(n\log_2 n)$
  - QuickSort: $O(n\log_2 n)$
  - Closest Pair: $O(n\log_2 n)$

- It may not look like much, but the reduction in the number of iterations is significant for larger problems.