Analysis and Design of Algorithms

Session 4.

Maestría en Sistemas Computacionales.

Luis Fernando Gutiérrez Preciado.

Divide and Conquer

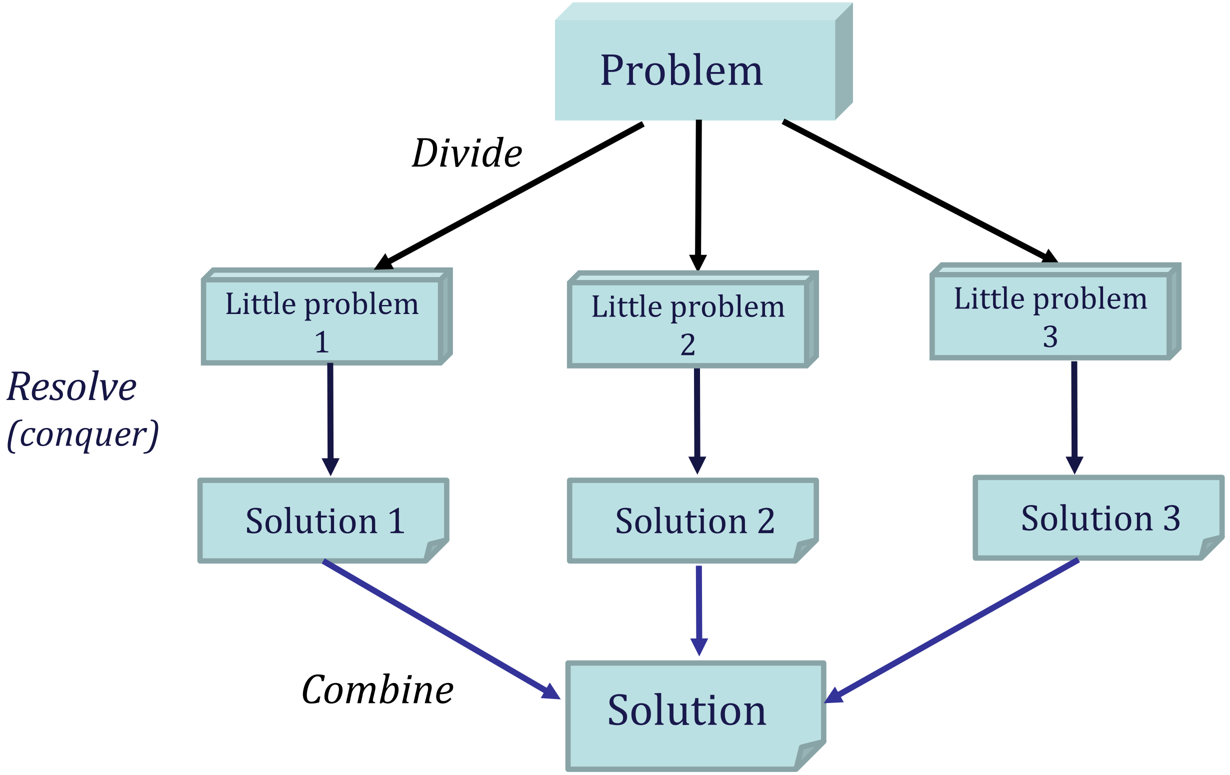
- There are two reasons to use this technique:

  1. The algorithm is more intuitive than the original (iterative) version and does not add computational cost.
  2. The algorithm is faster than the original version: it reduces temporal complexity.
- Not every recursive algorithm is Divide and Conquer:

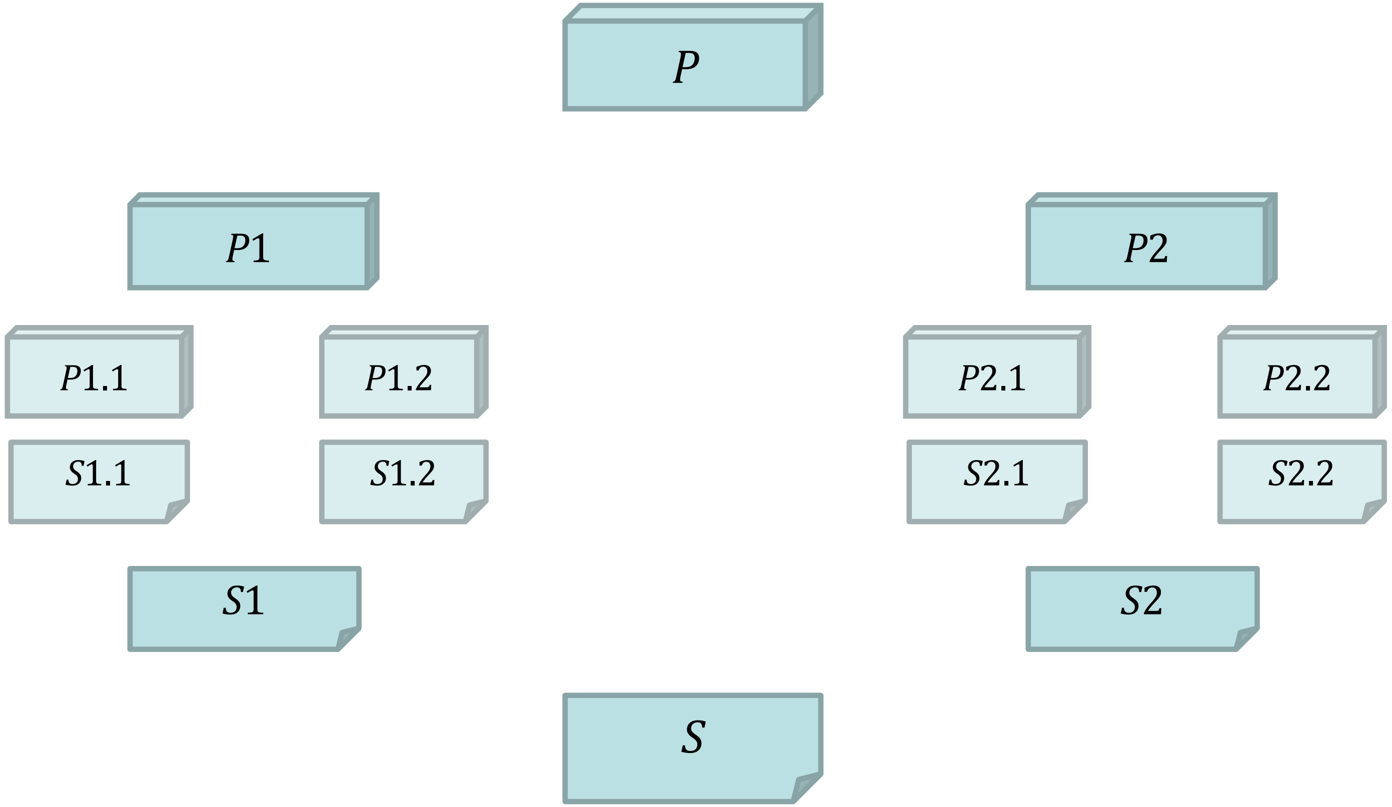
```
return Fibonacci(N – 1) + Fibonacci(N – 2)
```

  1. It adds computational cost to the iterative version.
  2. Do we Divide and Lose?

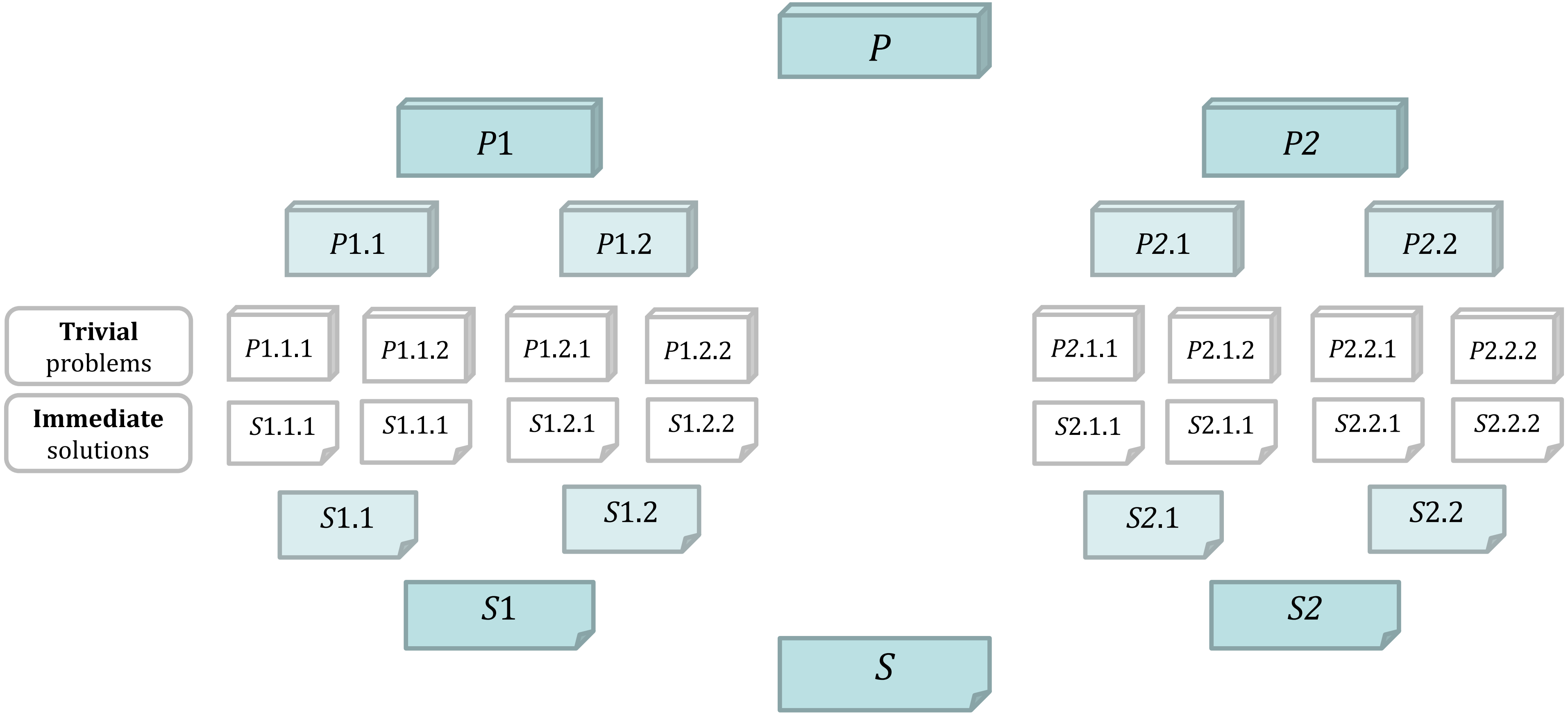
Divide and Conquer



Divide and Conquer



Divide and Conquer



Divide and Conquer

- General Form 1: (with return)

```
type Method(search space)
```

1. Perform operation(s) with the search space
2. Can we terminate with success?

   Return the value we were looking for.

1. Can we terminate with failure?

   Return an error value of type 'type'.

   Can't we finish yet?

   a. If necessary, return Method(subspace-1)...
   b. If necessary, return Method(subspace-n)...

Binary Search

- Constraint: the array must be sorted.
- Idea:
  1. Search for the value in the middle of the list.
  2. If it's not there, it must be:
     a) In the left half if the value to be found was smaller.
     b) In the right half if the value to be found was larger.
  3. Search for the value in the middle of the chosen half.
  4. If it's not there, it must be:
     a) In the left half if the value to be found was smaller
     b) And so on …

Binary Search

- Let's continue with an example:

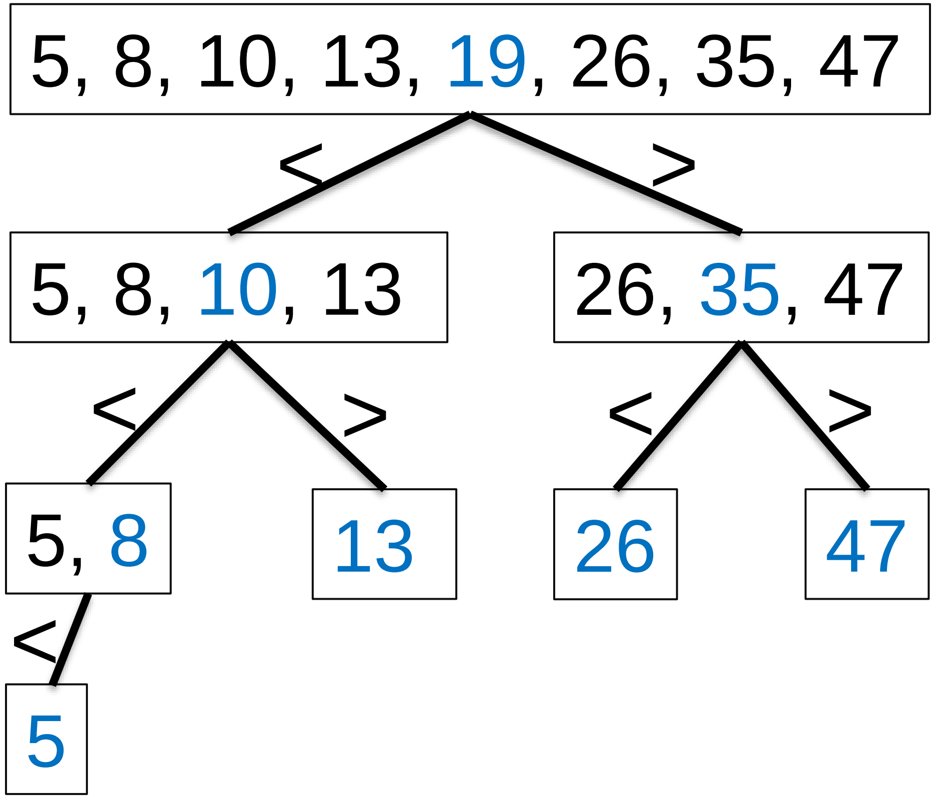
1. Value to find= **26**
2. Search space= [5, 8, 10, 13, 19, 26, 35, 47]
3. Middle point = 19
4. Search space = [5, 8, 10, 13, 19, 26, 35, 47]
5. Middle point = 35
6. Search space = [5, 8, 10, 13, 19, 26, 35, 47]
7. Middle point = 26 … ☺

How many comparisons were made?

How did the search space change?

Binary Search

- The blue value is the one compared at that moment with the searched value.



Individual Activity

- Pseudocode for binary search
- Return the index where the value to be found is located.

Team Activity

- Understanding the following algorithm

- Constraint: No repeated elements exist.
- Test with: [19, 10, 47, 5, 13, 26, 8, 35]
- $L_{N/2}$ (**value** in the middle of the list)
- Initial K is N/2

```
Algorithm(L: list of N elements, K: expected position): int
```

P ← the number of elements smaller than $L_{N/2}$

1. If P = K, return $L_{N/2}$
2. If P > K
   a) Create a list L1 with elements smaller than LN/2
   b) Return `Algorithm`(L1, K)
3. Si P < K
   a) Create a list L2 with elements greater than $L_{N/2}$
   b) Return `Algorithm`(L2, K – P – 1)

Median

- Constraint: No repeated elements exist.
- Test with: [19, 10, 47, 5, 13, 26, 8, 35]
- $L_{N/2}$ (**value** in the middle of the list)
- Initial K is N/2

```
Algorithm(L: list of N elements, K: expected position): int
```

P ← the number of elements smaller than $L_{N/2}$

1. If P = K, return $L_{N/2}$
2. If P > K
   a) Create a list L1 with elements smaller than LN/2
   b) Return `Algorithm`(L1, K)
3. Si P < K
   a) Create a list L2 with elements greater than $L_{N/2}$
   b) Return `Algorithm`(L2, K – P – 1)

Median

- Example: The initial value of K will be N/2

1. List = [19, 10, 47, 5, 13, 26, 8, 35], K = 4

   $L_{N/2}$ = 13, P = 3 < K ∴ K = K – P – 1 = 0

2. List = [19, 47, 26, 35], K = 0

   $L_{N/2}$ = 26, P = 1 > K

3. List = [19], K = 0

   $L_{N/2}$ = 19, P = 0 = K ∴ Median = 19

How many comparisons were made?

Quicksort

- Invented in 1960 by C.A.R. Hoare, British.
- Average runtime is N log N operations for sorting N elements.
- One of the most popular efficient sorting algorithms: not difficult to implement.
- Disadvantages:
  - It's recursive in its original form (can be fixed).
  - Executes N^2 operations in the worst case.
  - It's fragile: a small error in implementation can cause it to fail in several cases.

Quicksort

- Partitions the search space.

1. Chooses an element from the list: pivot (typically the last, but could also be the first).
2. Determines the final position for the pivot.
3. Places elements smaller than the pivot on its left and larger ones on its right.
   – Note that the two formed sub-arrays can be sorted independently.
4. Repeats all steps with the left and right halves of the pivot until the size of each sub-array allows manual sorting ($N \leq 2$).

Quicksort

- Sub-arrays are managed using left and right indices (not new arrays)
- Quicksort(array, left, right)
  1. If the array delimited by left and right is small enough, sort it (if necessary) and terminate.
  2. Let p = partition(array, left, right)
     - p is the final position of the pivot element.
     - The implementation of partition() varies but is crucial.
  3. Quicksort(array, left, p – 1)
  4. Quicksort(array, p + 1, right)

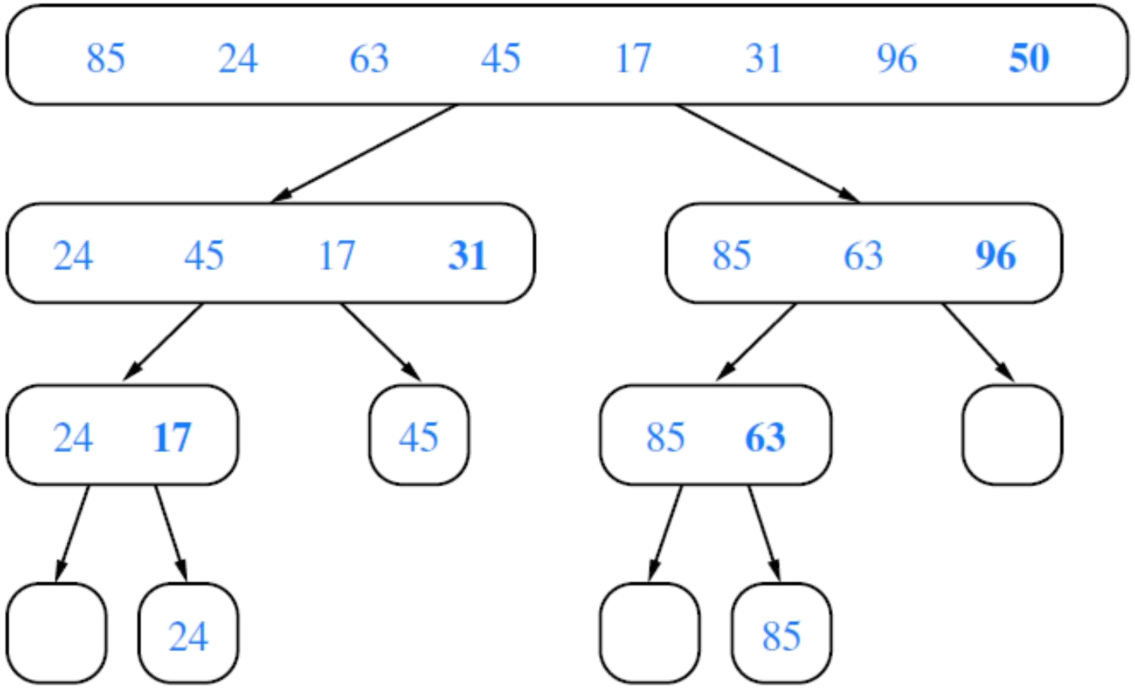
How to do the partition?

1. Choose the last element as the pivot (right).
2. Determine the position p1 of the first element greater than the pivot (that shouldn't be there).
   - The search goes from left to right of the sub-array.
3. Determine the position p2 of the first element smaller than the pivot (that shouldn't be there).
   - The search goes from right – 1 to left of the sub-array.
4. Did p1 and p2 cross (p1>=p2)?
   1. Swap the elements at p1 and the pivot (right).
   2. The partition position is p1.
5. Didn't they cross?
   1. Swap the elements at p1 and p2.
   2. Return to step 2 with the next p1, p2.

Example

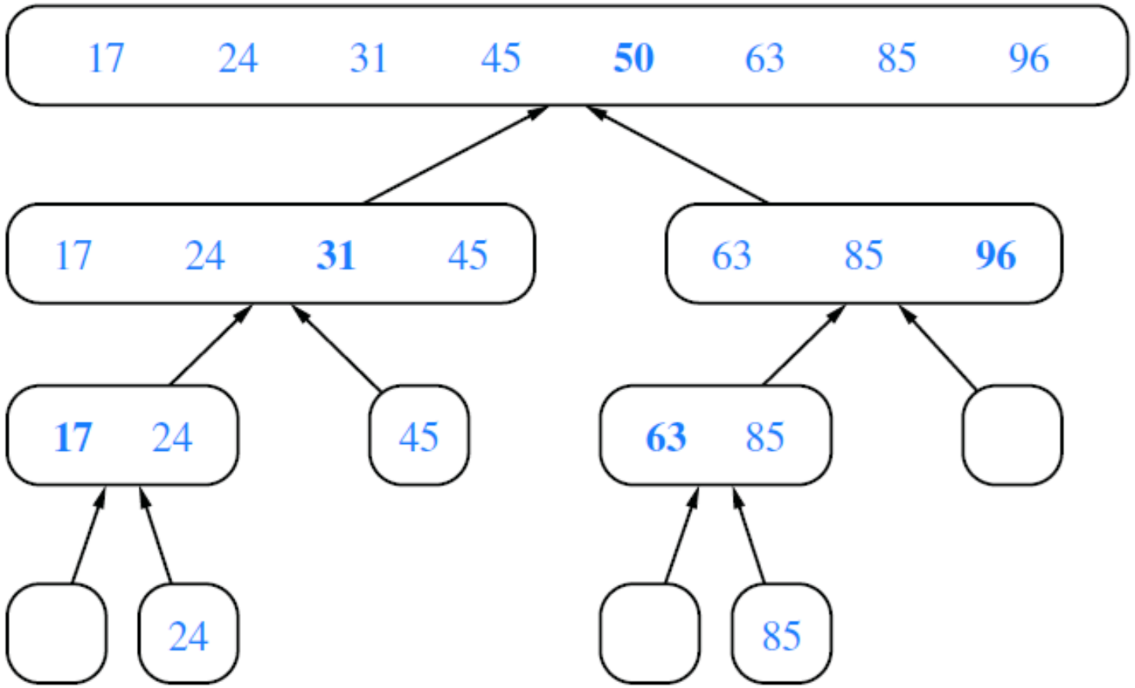
1. List to sort= {5, 4, -8, 2, -1, 9, 0, -3, 7, 6}
2. Left = 0, Right = 9, Pivot = 6
3. First partition:
    1. p_1 = 5 {$list_5$ = 9} (first larger than the pivot)
    2. p_2 = 7 {$list_7$ = -3} (first smaller than the pivot)
    3. They don't cross
    4. Swap(5, 7) (swap (p1, p2)
    5. List = {5, 4, -8, 2, -1, **-3**, 0, 9, 7, 6}
    6. List = {5, 4, -8, 2, -1, -3, 0, 9, 7, 6}
    7. p_1 = 7 {$lista_7$ = 9} (first larger than the pivot)
    8. p_2 = 6 {$lista_6$ = 0} (first smaller than the pivot)
    9. They cross
    10. Swap(7, 9) (swap (p1, right))
    11. List = {**5, 4, -8, 2, -1, -3, 0**, **6**, **7, 9**}
    12. Partition = 7

Example



Data Structures and Algorithms in Java, 6th Edition

Example



Data Structures and Algorithms in Java, 6th Edition

Mergesort

- Invented in 1945 by John von Neumann, Hungarian.
- Its temporal complexity is N log N in the best, worst, and average cases.
- Merges the content of two sorted arrays into a larger array, linear complexity.
- Disadvantages:
  - It's recursive in its original form (can be fixed).
  - The need to create new arrays in each recursive call.

Mergesort

- Partitions the search space.

1. If the array has a minimum length, sort it manually and return it.
2. Create two sub-arrays from the original array, one with the left half and one with the right half.
   – One array may be larger than the other.
3. Sort each sub-array through two calls to this same method.
4. Return the result of merging the two previously sorted sub-arrays.Realiza particiones del espacio de búsqueda.

¿Cómo hacer la mezcla?

- Array1 = {3, 5, 7, 8, 10}
- Array2 = {2, 4, 5, 6, 10, 12}
- Array3 will be of 11 elements
- A counter is required for each array
  - Array3[0] = Array2[0]
  - Array3[1] = Array1[0]
  - Array3[2] = Array2[1]
  - Array3[3] = Array1[1]
  - Array3[4] = Array2[2]
  - Array3[5] = Array2[3] ...
  - Array3[10] = Array2[5]

Example

1. List to sort = {5, 4, -8, 2, -1, 9, 0, -3, 7, 6}
2. **Left** = {5, 4, -8, 2, -1}
   a) Left' = {5, 4}
      i. Left'' = {5}
      ii. Right'' = {4}
      iii. Merge'' = {4, 5}
   b) Right' = {-8, 2, -1}
      i. Left'' = {-8}
      ii. Right'' = {2, -1}
      iii. Merge'' = {-8, -1, 2}
   c) Merge' = {-8, -1, 2, 4, 5}
3. **Right** = {9, 0, -3, 7, 6}
   a) Left' = {9, 0}
   b) Right' = {-3, 7, 6}
   c) Merge' = {-3, 0, 6, 7, 9}

Sorting left side'

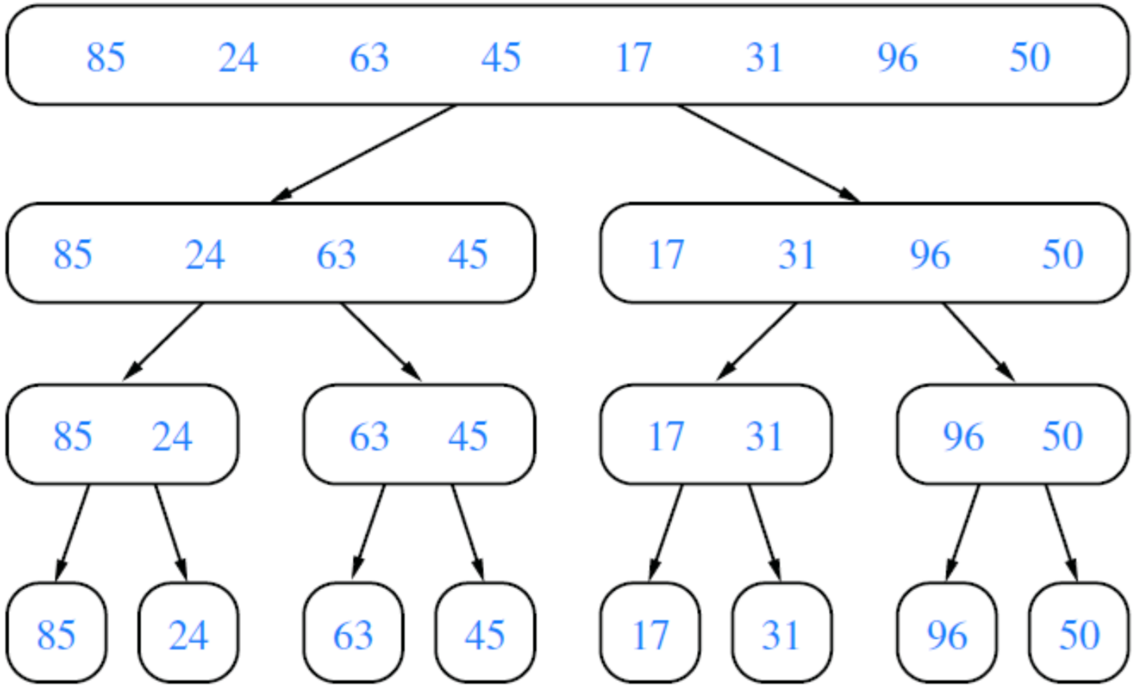
Sorting Right side'

Sorting left side

Sorting right side

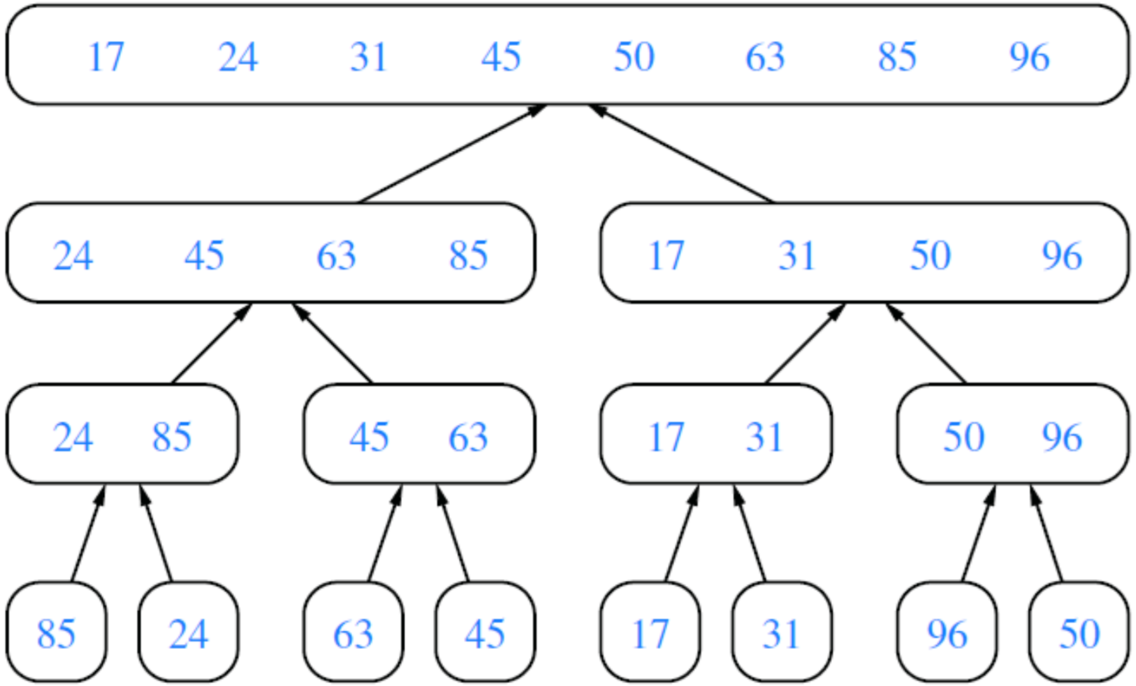
[Merge Two Sorted Lists Animation Animation by Y. Daniel Liang (pearsoncmg.com)]

Ejemplo



Data Structures and Algorithms in Java, 6th Edition

Ejemplo



Data Structures and Algorithms in Java, 6th Edition

Recursive Algorithm Analysis

- Case 1:
  - In each recursive call, the search space is reduced by one.
  - One operation is performed with each element.

T(n) = T(n-1) + n

T(n) + n	n
T(n-1)+ n-1	n-1
T(n-2) + n-2	n-2
…	
T(1) + 1	1

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}(n^2 + n) \in O\,(n^2)$$

Recursive Algorithm Analysis

- Case 2:
  - In each recursive call, the search space is reduced by half:
  - Only one operation is performed.

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

$$\log_2 n \left\{ \begin{array}{ll|l} T(n) & + 1 & 1 \\ T(n/2) & + 1 & 1 \\ T(n/4) & + 1 & 1 \\ \cdots & & \\ 1 & + 1 & 1 \end{array} \right.$$

$$\sum_{i=1}^{\log_2 n} 1 = \log_2(n)$$

Recursive Algorithm Analysis

- Case 3 (tree mode)
  - In each recursive call, the search space is reduced by half.
  - One operation is performed for each element.

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

$$
\log_2 n \left\{
\begin{array}{l|l}
T(n) \quad + n & n \\
T(n/2) + n/2 & n/2 \\
T(n/4) + n/4 & n/4 \\
\ldots & \\
1 + 1 & 1
\end{array}
\right.
$$

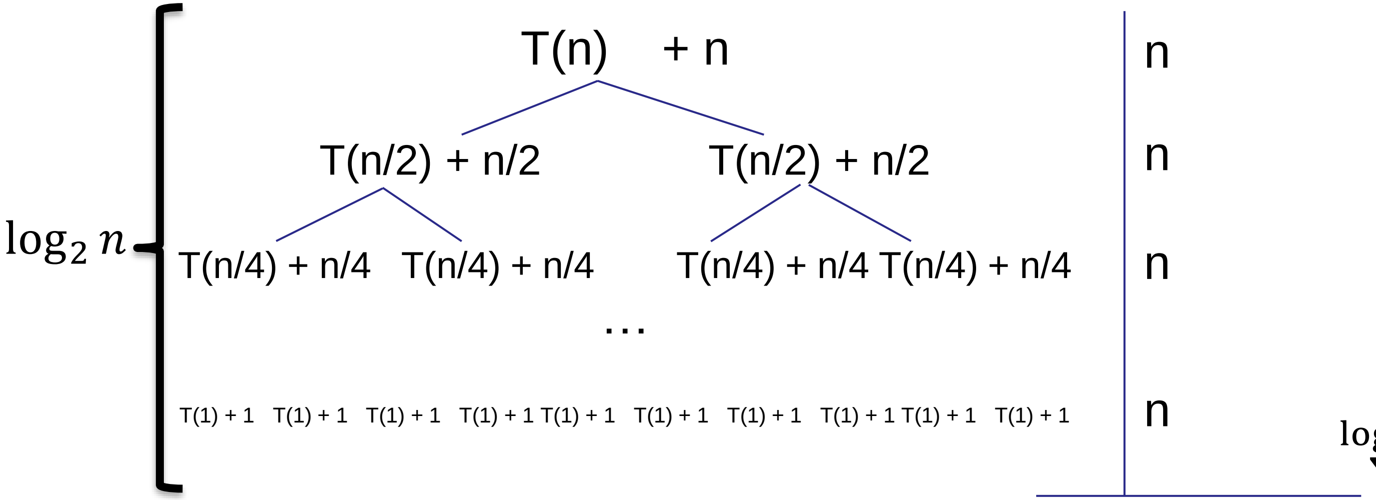
$$\boxed{\sum_{i=0}^{n-1} \frac{1}{2^i} = 2 - \frac{1}{2^{n-1}}}$$

$$\sum_{i=0}^{\log_2(n)} \frac{n}{2^i} = n \sum_{i=0}^{\log_2(n)} \frac{1}{2^i} = n\left(2 - \frac{1}{2^{\log_2(n)}}\right) = n\left(2 - \frac{1}{n}\right) = 2n - 1$$

Recursive Algorithm Analysis

- Case 4
  - Two recursive calls are made, each processing a half of the current search space.
  - One operation is performed with each element.

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



$$\sum_{i=1}^{\log_2(n)} n = nlog_2(n)$$

Conclusions

- Estrategia de divide y vencerás
- Búsqueda binaria
- Cálculo de la mediana
- Algoritmos de ordenamiento:
  - Merge Sort
  - Quick Sort (partition)
- Análisis de algoritmos recursivos