Divide and Conquer Algorithm

Group 8

Idea

- The basic idea is to decompose a given problem into independent and simpler sub-problems, to solve them in turn, and to compose their solutions to solve the given problem.
- Sub-problems must be independent. (Ex: Fibonacci)

Framework

Divide

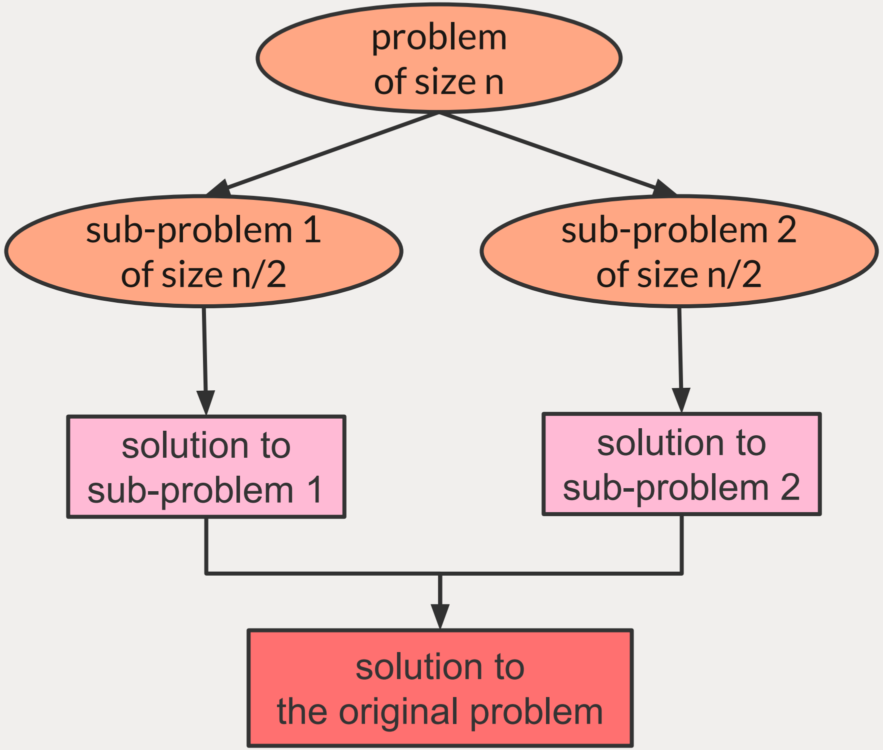
The problem is divided into smaller sub-problems that are similar to the original problem but of smaller size.

Conquer

The sub-problems are solved recursively using the same algorithm until they become simple enough to be solved directly.

Combine

The solutions to the sub-problems are combined to solve the original problem.



Specification

The problem must be able to divide into sub-problems similar to the big problem but with a smaller size.

Algorithm

```
DAC(a, i, j)
{
    if(small(a, i, j)
        return(Solution(a, i, j))
    else
        mid = divide(a, i, j)
        b = DAC(a, i, mid)
        c = DAC(a, mid+1, j)
        d = combine(b, c)
    return(d)
}
```

Time **Complexity**

size of input

n

size of each sub-problem; all sub-problems are assumed to have the same size

n/b

$$T(n) = aT(n/b) + f(n)$$

a

number of sub-problems in the recursion

f(n)

cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Master Theorem

If $f(n) \in \theta(n^d)$ where $d \geq 0$ in recurrence $T(n) = aT(n/b) + f(n)$, then

$$T(n) \epsilon \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Binary Search

Idea

A searching algorithm used in a sorted array by **repeatedly dividing the search interval in half**

Conditions

- The data structure must be sorted
- Access to any element of the data structure takes constant time

Binary Search

01

Set the low index to the first element of the array and the high index to the last element.

Search 23

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

Binary Search

02

Set the middle index to the average of the low and high indices.

- If the element at the middle index is the target element, return the middle index.
- Otherwise, based on the value of the key to be found and the value of the middle element, decide the next search space.
    - If the target is less than the element at the middle index, set the high index to **middle index – 1**.
    - If the target is greater than the element at the middle index, set the low index to **middle index + 1**.

Binary Search

02

23 > 16
take 2nd half

L=0	1	2	3	M=4	5	6	7	8	H=9
2	5	8	12	16	23	38	56	72	91

23 < 16
take 1st half

0	1	2	3	4	L=5	6	M=7	8	H=9
2	5	8	12	16	23	38	56	72	91

Binary Search

03

Perform step 2 repeatedly until the target element is found or the search space is exhausted.

0	1	2	3	4	L=5 M=5	H=6	7	8	9
2	5	8	12	16	23	38	56	72	91

Found 23,
Return 5

Binary Search

```
binarySearch(arr, x, low, high)
    if low > high
      return False
    else
      mid = (low + high) / 2
      if x = arr[mid]
        return mid
      else if x > arr[mid]
        return binarySearch(arr, x, mid + 1, high)
      else
        return binarySearch(arr, x, low, mid – 1)
```

Binary Search

2

Time Complexity

O(logn)

Auxiliary Space

O(1), if the recursive call stack is considered then the auxiliary space will be O(logn)

Merge Sort

2

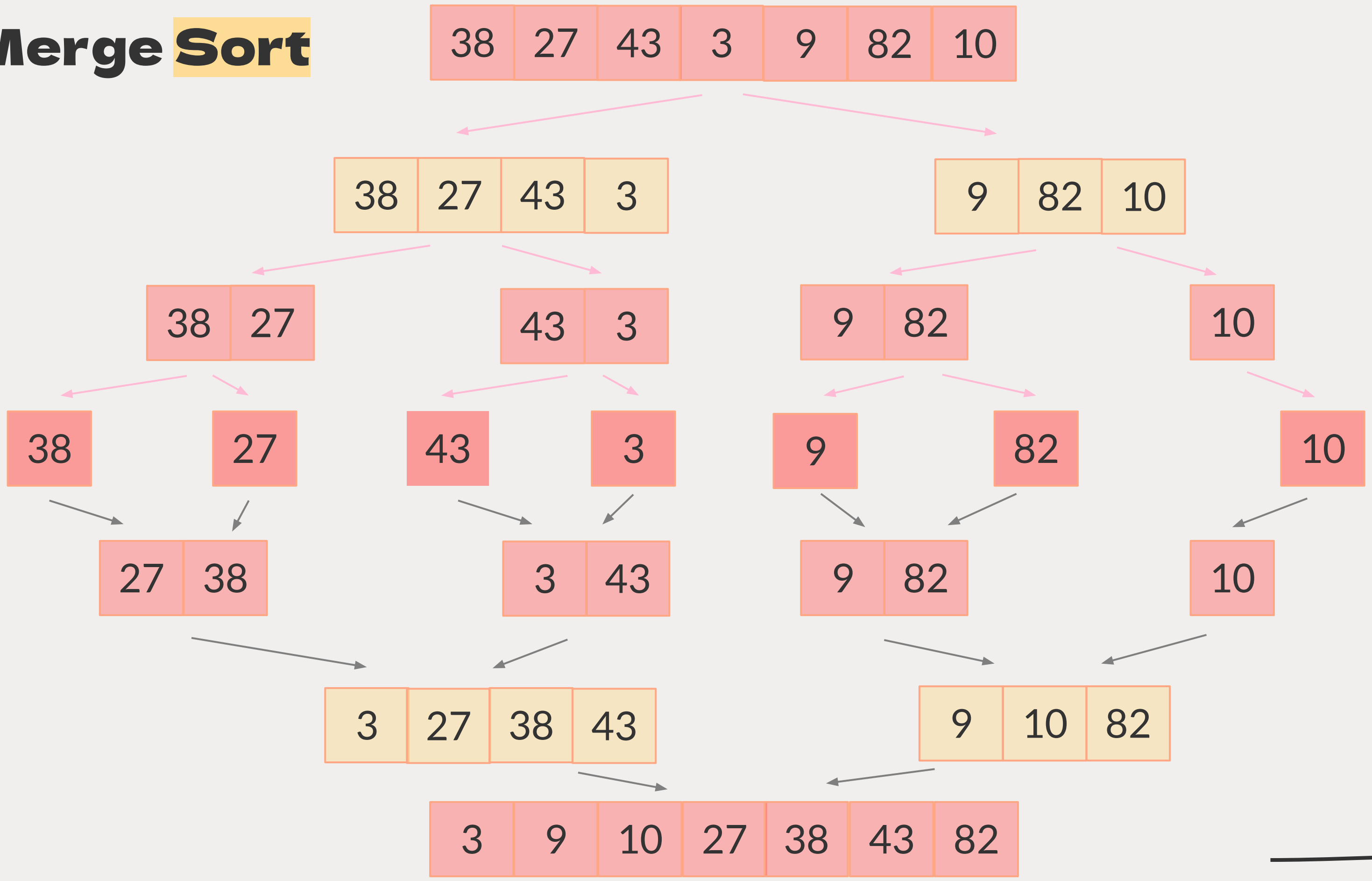
Sorting algorithm

Dividing the array into 2 halves, recursively sorting them, and finally merging the 2 sorted halves to get the original array sorted.

Time complexity

O(nlogn)

Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge **Sort**

```
mergeSort(arr, left, right)
   if left > right
      return
   mid = (left + right) / 2
   mergeSort(arr, left, mid)
   mergeSort(arr, mid + 1, right)
   merge(arr, left, mid, right)
```

Quick Sort

2

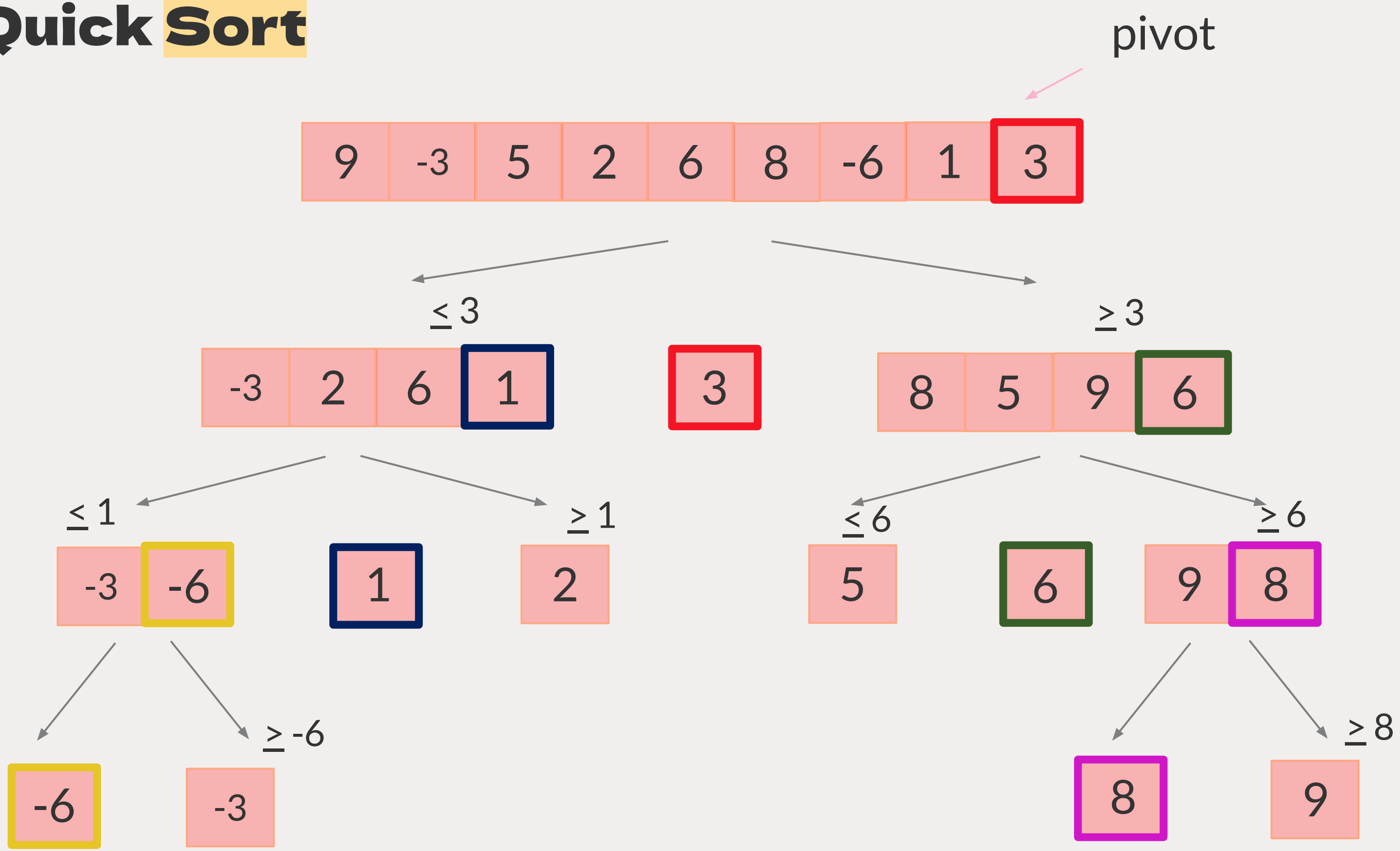
Sorting algorithm

Picking an element as a pivot and partitioning the given array around the picked pivot

Time complexity

O(nlogn)

Quick Sort



Quick Sort

```
quickSort(arr, low, high)
   if low < high
      pi = partition(arr, low, high)
      quickSort(arr, low, pi – 1)
      quickSort(arr, pi + 1, high)
```

Applications

Binary Search

Merge Sort
Quick Sort

Median Finding

Fast Power

Matrix Multiplication

Closet Pair Problem

Basic Matrix Multiplication

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

Basic Matrix Multiplication

$$
\begin{aligned}
&for\ i \leftarrow 1\ to\ n\ do \\
&\quad for\ j \leftarrow 1\ to\ n\ do \\
&\qquad c_{ij} \leftarrow 0 \\
&\qquad for\ k \leftarrow 1\ to\ n\ do \\
&\qquad\quad c_{ij} = c_{ij} + a_{ik} \times b_{kj}
\end{aligned}
$$

Time complexity

$$O(n^3)$$

DAC Matrix Multiplication

$n \times n$ matrix = 2 × 2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]$$

$$C \quad = \quad A \quad \cdot \quad B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dh \end{aligned}\right\}$$

8 mults of $(n/2) \times (n/2)$ submatrices
4 adds of $(n/2) \times (n/2)$ submatrices

DAC Matrix Multiplication

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

$n = A.rows$
let C be a new $n \times n$ matrix
if $n == 1$
    $c_{11} = a_{11} \cdot b_{11}$
else partition A, B, and C as in equations (4.9)
    $C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
    $C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
    $C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
    $C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
return C

Time complexity

$$T(n) = \Theta\left(n^{\log_2 8}\right) = \Theta\left(n^3\right)$$

Strassen's Matrix

Multiply 2x2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f - h)\\
P_2 &= (a + b) \cdot h\\
P_3 &= (c + d) \cdot e\\
P_4 &= d \cdot (g - e)\\
P_5 &= (a + d) \cdot (e + h)\\
P_6 &= (b - d) \cdot (g + h)\\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6\\
s &= P_1 + P_2\\
t &= P_3 + P_4\\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

Time complexity

$$T(n) = \Theta\left(n^{\log_2 7}\right)$$

Strassen's Matrix

Note: If n is small enough, 7 mults and 18 adds or subs the matrix may run slower than basic multiplication, so the algorithm is usually only run until it reaches the predetermined threshold and then fallbacks to multiply by definition.

Advantages

solving difficult problems

algorithm efficiency

parallelism

memory access

Disadvantages

overhead

difficulty of implementation

memory limitations