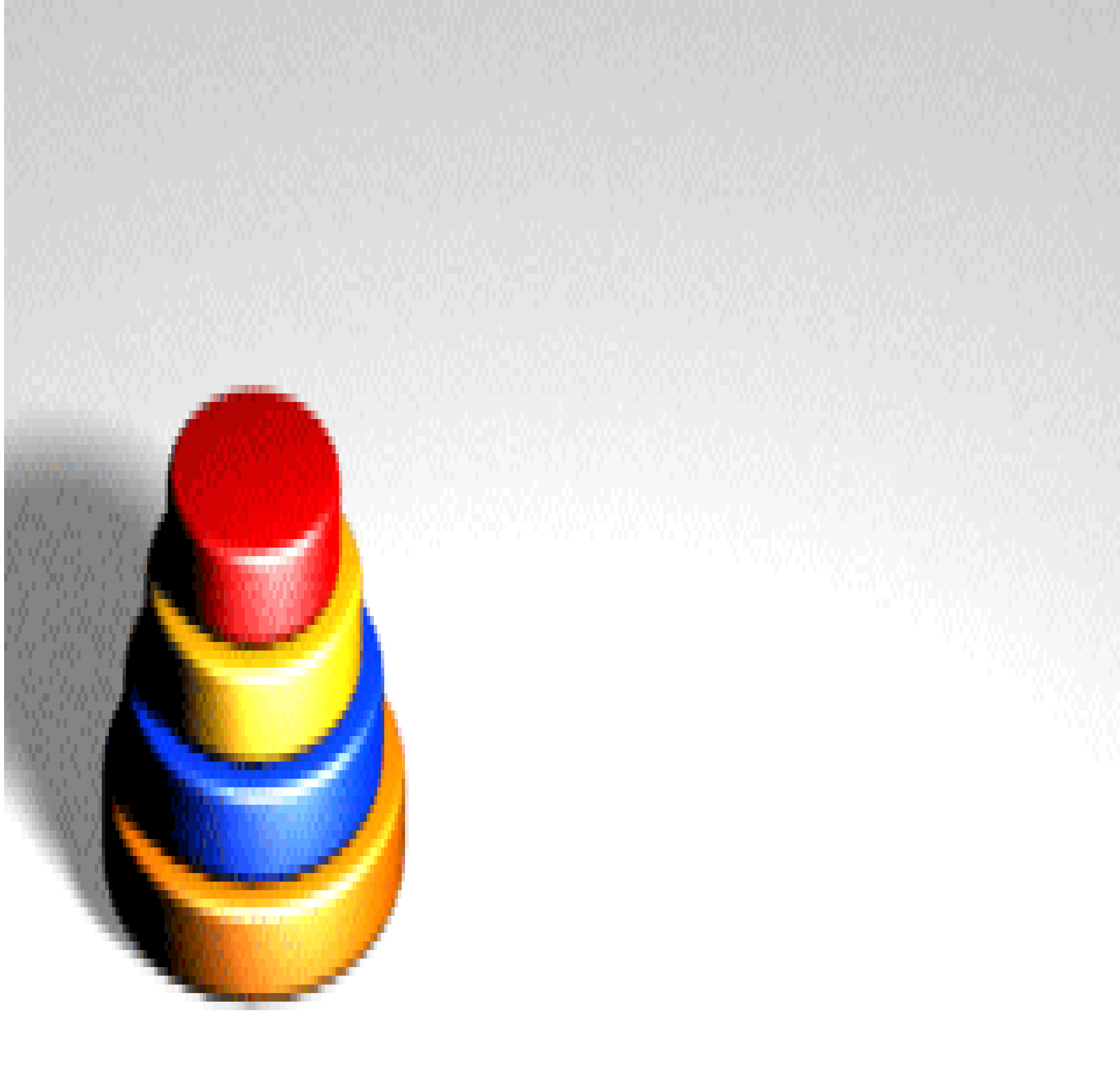
Group 11

Trương Quốc Trường
Nguyễn Quang Tuấn
Nguyễn Ngọc Tân

Introduce the method of designing a DIVIDE - AND - CONQUER algorithm

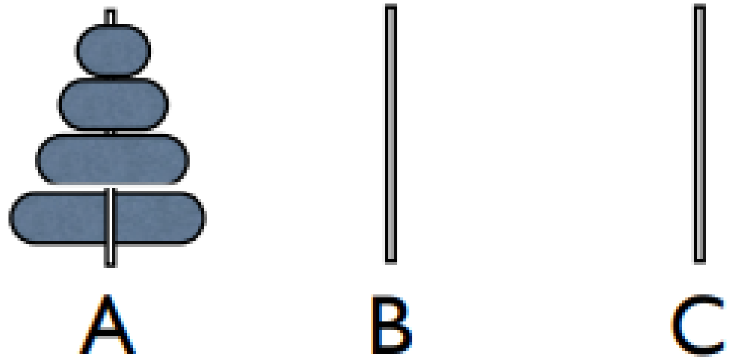
The classic problem in the practice part of tourism and transport engineering is the Hanoi Tower lesson.

=> One of the typical examples applying the divide and conquer method.



Solution Strategy

- Base case: if n is 1, the solution is trivial. Just move the disk
- Otherwise:
  - Move (n-1) disks from peg A to peg C *using Hanoi for n-1 disks*
- Move the left-over disk from page A to peg B
- Move (n-1) disks from peg C to peg B *using Hanoi for (n-1) disks*



Overview

I. Introduce algorithm:

- Divide and conquer is **an algorithm design paradigm.**
- The basis of efficient algorithms for many problems.

Question: What is the difference between algorithm design paradigm and programming technique?

=> It should be noted that, D-A-C isn't programming technique.
=> Algorithm design paradigm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems.
Programming technique is the technique of implementing a solutionsoftware (data structure + algorithm) based on one platformmethodology and one or more programming languages to suit the specific requirements of the application.

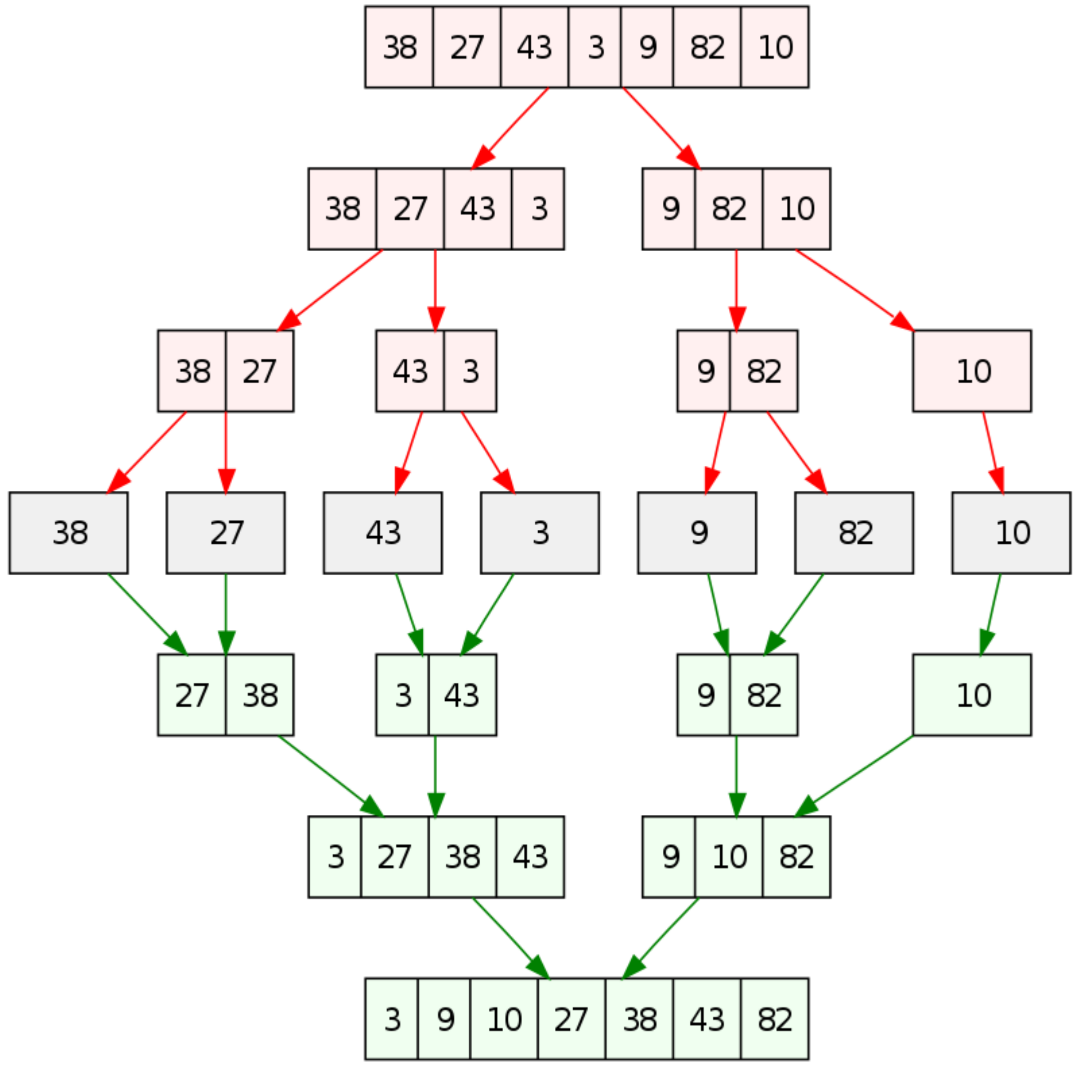
I. Introduce algorithm:

- Its basic idea is to decompose a given problem into two or more similar, to solve them in turn, and to compose their solutions to solve the given problem.

- Should be used only when each problem may generate two or more subproblems.

-It’s ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

Merge sort



II. Feature of the problem:

Problem 1:

“For an array of **n** elements, sort the elementsof the sequence in ascending order.”

=> Sort(array[n/2])

Problem 2:

“Give an integer **a** and an integer **n (n**≥1). Calculate $\mathbf{a}^n$.”

=> $A^n=a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}$

Problem 3:

“Given an unordered list of **n** elements, find **max** or **min**”

=> Find(array[n/2])

Question: *What are the common features of these problems?*

II. Feature of the problem:

- The original problem is replaced with a more general problem in order to create a recursive relationship.

<u>*Question*</u>*: Why do we have to use recursive call? If we don't use recursive calls, what other ways can we use them?*

=> Because divide-and-conquer creates at least two subproblems, so It makes multiple recursive calls. It’s easier and brief to use this way of programming.

=> In addition to the recursive call, we can also use other ways such as process logging in stack data structures, queue, etc.

II. Feature of the problem:

- Each difficult or large problem and it can be divided into two or more subproblems.

=> From the examples we see that most problems are broken down into two parts instead of multiple parts. It depends on the specific problem, but for ease of implementation we often divide the problem into 2 parts. And so on until the problem cannot be broken down anymore.

II. Feature of the problem:

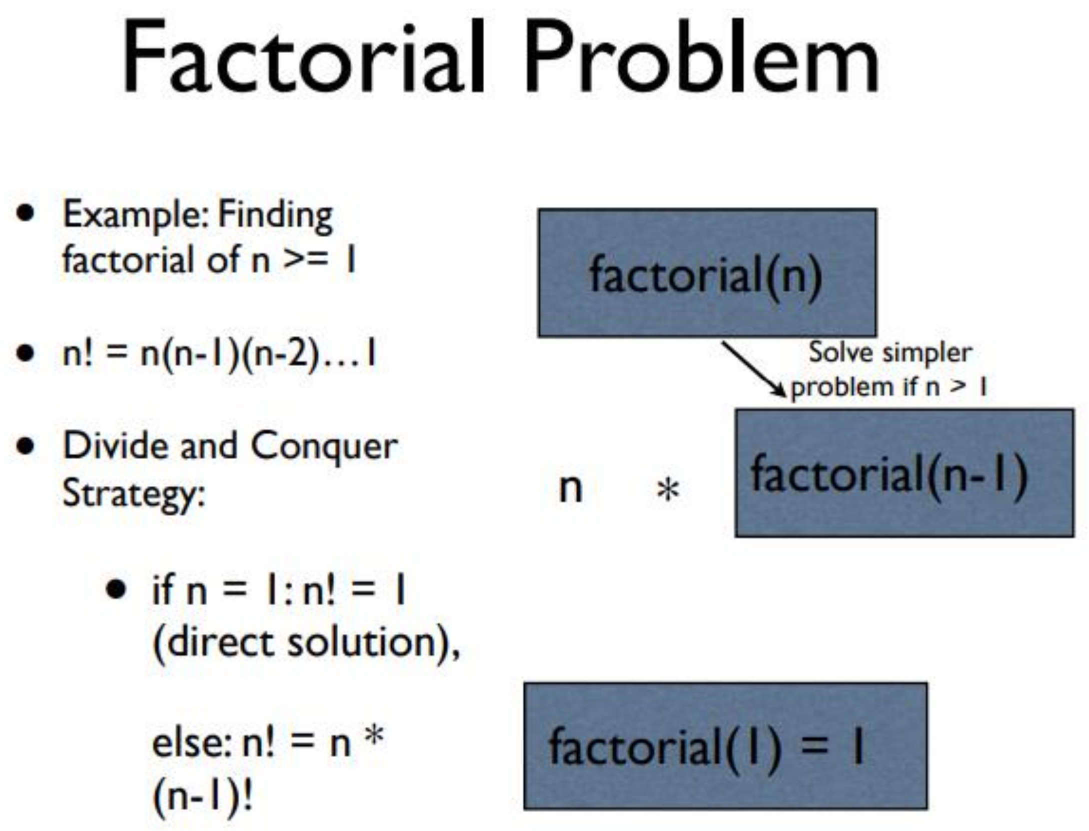
Example Problem 1: Mergesort is a perfect example of a successful application of the divide-andconquer technique

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Question: Predict what is the complexity of this algorithm??

II. Feature of the problem:

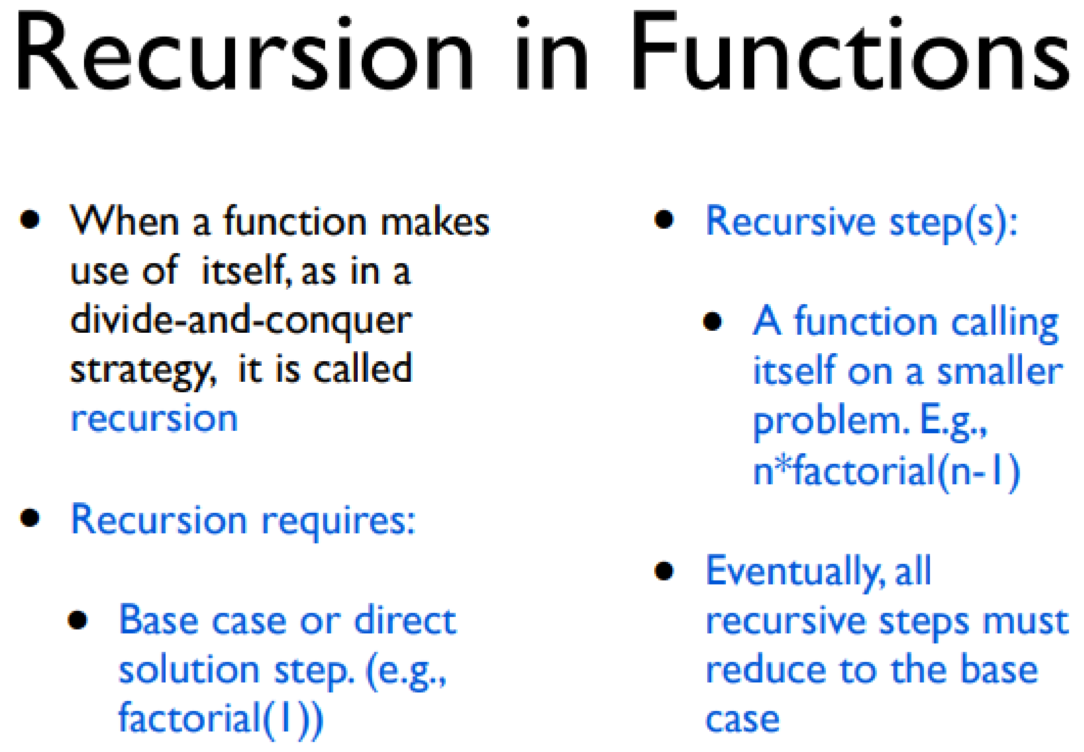
Example Problem 2:



Factorial Problem

- Example: Finding factorial of n >= 1
- n! = n(n-1)(n-2)…1
- Divide and Conquer Strategy:
  - if n = 1: n! = 1 (direct solution),

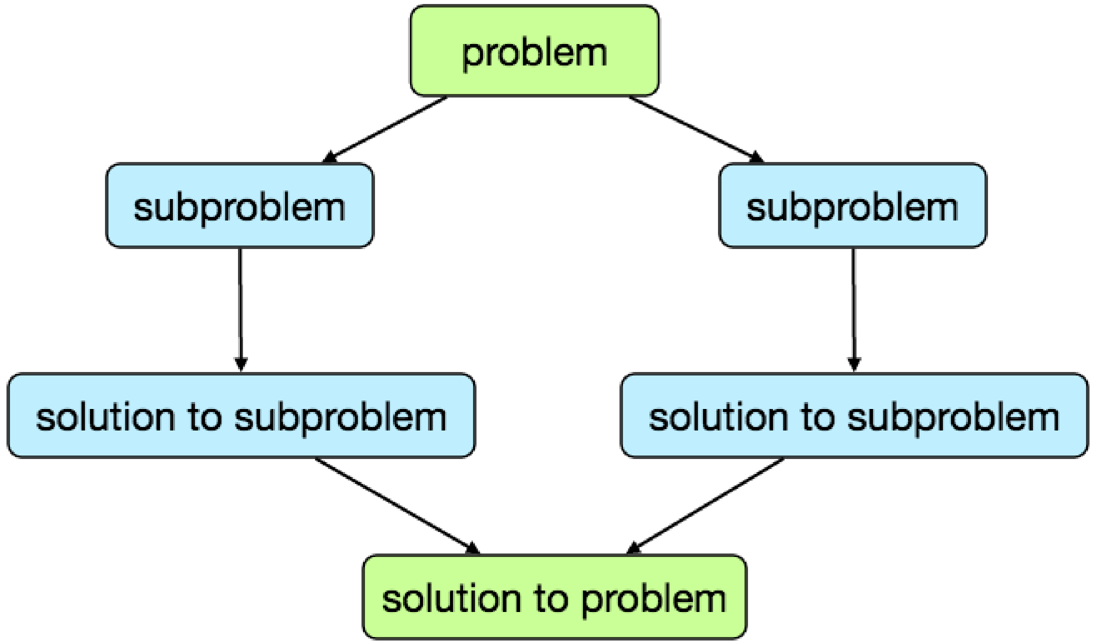
    else: n! = n * (n-1)!



Recursion in Functions

- When a function makes use of itself, as in a divide-and-conquer strategy, it is called recursion
- Recursion requires:
  - Base case or direct solution step. (e.g., factorial(1))
- Recursive step(s):
  - A function calling itself on a smaller problem. E.g., n*factorial(n-1)
- Eventually, all recursive steps must reduce to the base case

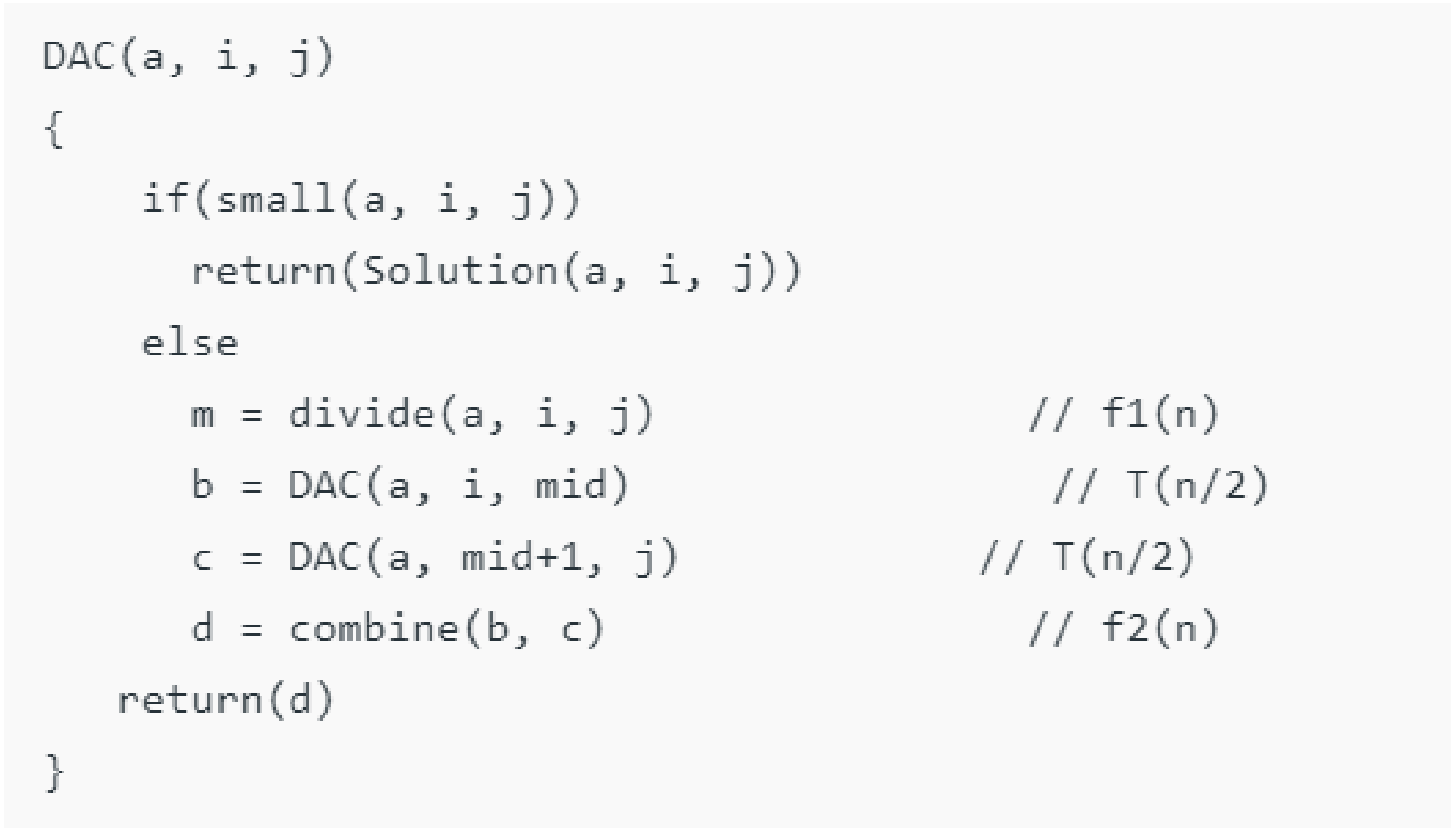
III. Generality algorithm:



1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

III. Generality algorithm:

Pseudocode:



```
DAC(a, i, j)
{
    if(small(a, i, j))
      return(Solution(a, i, j))
    else
      m = divide(a, i, j)               // f1(n)
      b = DAC(a, i, mid)                 // T(n/2)
      c = DAC(a, mid+1, j)            // T(n/2)
      d = combine(b, c)                 // f2(n)
   return(d)
}
```

III. Generality algorithm:

Time Complexity:

T(n) = aT(n/b) + f(n),
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed to have the same size.
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.

Question: Applying the formula, calculate the complexity of the Birnary Search problem?

IV. Positive & Negative:

$O(n.\log(n))$

Pos:

- Solving difficult problems: A powerful tool for solving conceptually dificult problems.
- Algorithm efficiency: Offen helps in the discovery of efficient algorithms.
- Memory access: Naturally tend to make efficient use of memory caches.

IV. Positive & Negative:

Neg:

- Divide and conquer cannot save the resultsthrough of problems resolved for the next request.

Example: Dãy Fibonacci

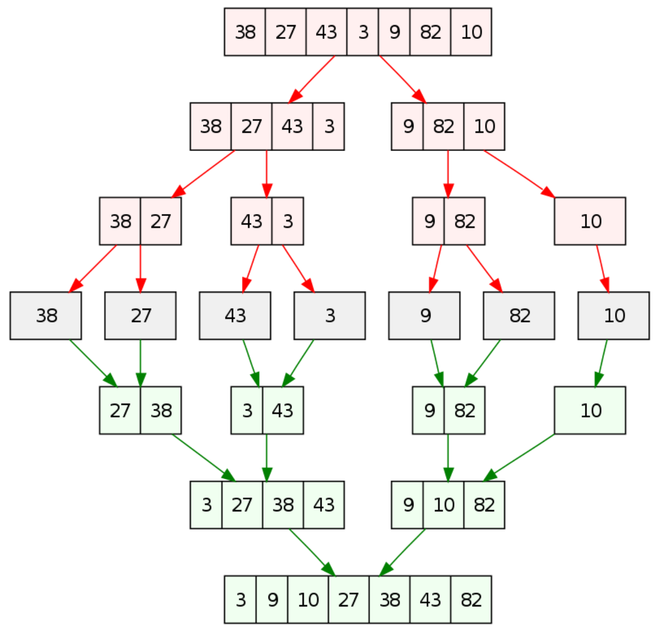
Divide and Conquer approach:

```
fib(n)
    If n < 2, return 1
    Else , return f(n - 1) + f(n -2)
```

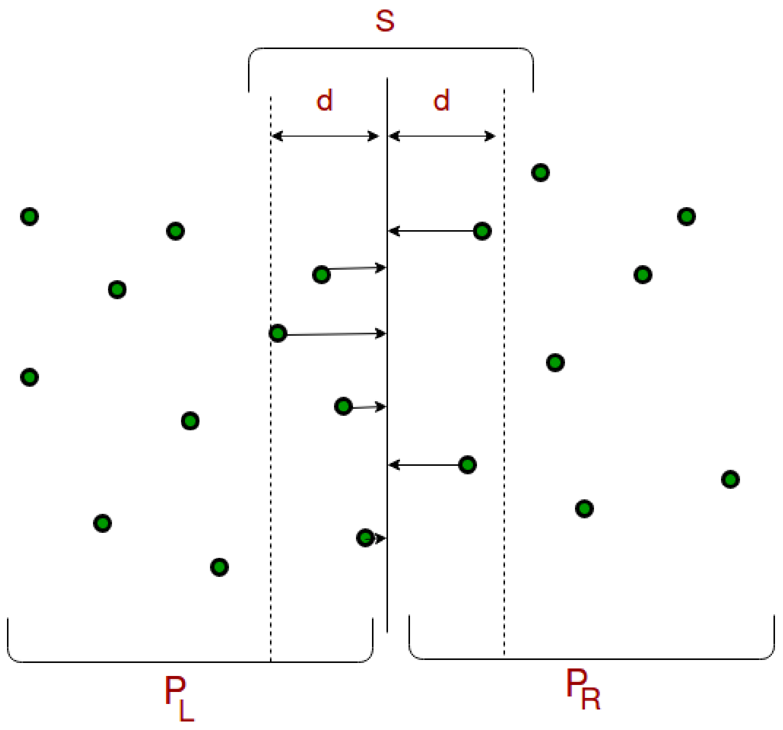
Dynamic approach:

```
mem = []
fib(n)
    If n in mem: return mem[n]
    else,
        If n < 2, f = 1
        else , f = f(n - 1) + f(n -2)
        mem[n] = f
        return f
```

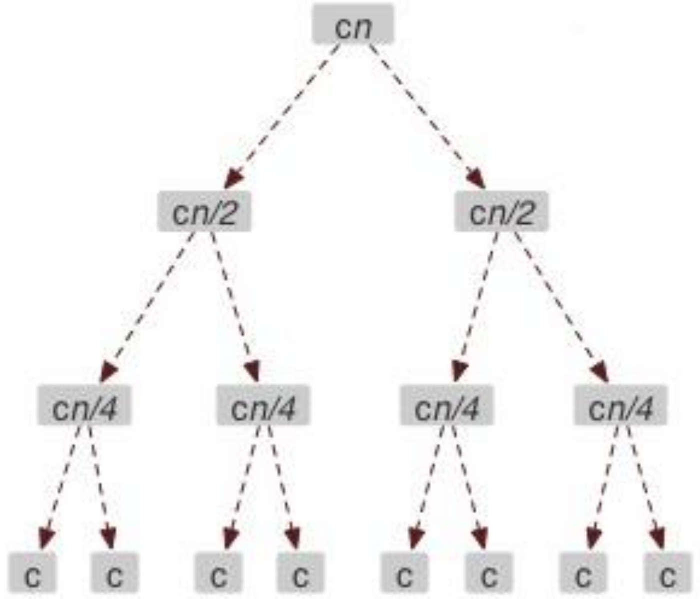
Example:



Merge sort



Closest pair of points



Multiplying large number (Karatsuba)

V. Problems:

Suitable case:

Given a sorted array Arr[] of n elements, write a function to search a given element x in Arr[].

=> within a Sorted array.
=> Binary search runs in logatithmic time in the worst case.

Pseudocode:

```
function binary_search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m − 1
        else:
            return m
    return unsuccessful
```

V. Problems:

Unsuitable case:

Given a sorted array Arr[] of n elements, write a function to calculate sum n elements.

=> Divide array into two section.
=> Calculate sum in each section.

Time complexity:

D-A-C:

```
total run-time: 601.881504 ms
Sum = 4998100215846
```

Others:

```
total run-time: 185.421944 ms
Sum = 4998100215846
```

Pseudocode:

```
def sum(a, L,  R):
        if (L == R): return a[L]
        m = (L + R) // 2
        sumLeft = sum(a, L, m)
        sumRight = sum(a, m + 1, R)

        return sumLeft + sumRight
```

VI. Conclusion:

The divide and conquer method is also a way to solve when we encounter difficult tasks, dividing them into smaller tasks to do instead of doing a whole big task, dividing the smaller task will make it easier to perform.

Homework:

Given an array of n points in the plane, and the problem is to find out the closest pair of points in the array.

=> " The Brute force solution is O(n^2), compute the distance between each pair and return the smallest. We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy."

Input: *[(1,1) (1,4), (2,5), (3,1), (3,3), (5,3), (5,5), (5,6), (6,2), (7,4)]*

Output: *(5,6), (5,5)*

Extensions:

- The partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack
- Divide-and-conquer algorithms can also be implemented by a non-recursive program that stores the partial sub-problems in some explicit data structure, such as a stack, queue, or priority queue
- Must make sure that there is sufficient memory allocated for the recursion stack, otherwise, the execution may fail because of stack overflow.
- Choosing the smallest or simplest possible base cases usually leads to simpler programs.

Extensions:

Reference source:

- https://www.javatpoint.com/divide-and-conquer-introduction
- https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/
- http://www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/lectures/dandc-notes.pdf
- Anany Levitin, Introduction to the Design and Analysis of Algorithms, 3rd Edition, 2014
- https://www.geeksforgeeks.org/fundamentals-of-algorithms/#AnalysisofAlgorithms
- https://www.codechef.com/wiki/test-generation-plan

Thank you !