**Divide and Conquer**

# Divide and Conquer

Divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

A typical Divide and Conquer algorithm solves a problem using following three steps:

1. **Divide**: Break the given problem into subproblems of same type.
2. **Conquer**: Recursively solve these subproblems
3. **Combine**: Appropriately combine the answers

# Divide and Conquer (D&C) vs Dynamic Programming (DP)

Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. How to choose one of them for a given problem?

Divide and Conquer should be used when same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used.

Example

* Binary Search is a Divide and Conquer algorithm, we never evaluate the same subproblems again.
* Calculating nth Fibonacci number, Dynamic Programming should be preferred.

# Advantages

## Solving difficult problems

It requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem.

## Algorithm efficiency

Divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and merge sort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms.

## Parallelism

Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

## Memory access

Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called cache-oblivious.

## Roundoff control

A divide-and-conquer algorithm may yield more accurate results than a superficially equivalent iterative method. For example, one can add N numbers either by a simple loop that adds each datum to a single variable, or by a D&C algorithm called pairwise summation that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate.

# Implementation Issues

## Recursion

Divide-and-conquer algorithms are naturally implemented as recursive procedures. In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack.

## Explicit 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.

## Stack size

In recursive implementations of D&C algorithms, one must make sure that there is sufficient memory allocated for the recursion stack, otherwise the execution may fail because of stack overflow. Fortunately, D&C algorithms that are time-efficient often have relatively small recursion depth.

For example, the quicksort algorithm can be implemented so that it never requires more than log2n nested recursive calls to sort n items.

## Choosing the base cases

In any recursive algorithm, there is considerable freedom in the choice of the base cases, the small subproblems that are solved directly in order to terminate the recursion.

## Sharing repeated subproblems

For some problems, the branched recursion may end up evaluating the same sub-problem many times over. In such cases it may be worth identifying and saving the solutions to these overlapping subproblems, a technique commonly known as memorization.

# END