**Divide and Conquer**

Divide the input into smaller parts, and then solve the parts individually.

Combine the results to give the final result.

Often done recursively

More efficient in a max/min search because we never compare the maximum and minimum elements

Some algorithms can be improved in efficiency, but without affecting their asymptotic time complexity.

Some things for example, can be improved, but still always need to check each element once.

D&C often used in

* Efficient Sorting
* Fast Integer multiplication
* Matrix multiplication
* Nearest Neighbour problems

Remark:

If the subproblems are not independent, i.e. subproblems share subsubproblems, then a divide-and-conquer algorithm repeatedly solves the common subsubproblems. Thus, it does more work than necessary! This is why it isn’t great for knapsack problem (branch and bound is divide and conquer)

**Greedy Methods**

Imagine a shop scenario, and we want to make change using the minimum number of coins.

Start with largest denomination, use as many as we can, then do the same with the next largest denomination and continue until we have the required change.

This is called a greedy strategy.

Always gives optimal solution (with earth currency)

Below is an example of where the greedy algorithm might not produce the optimal solution (with a strange currency)

we have the coins 1, 10, 6 and need to make 12

Greedy algorithm takes 10, 1, 1

not most efficient, should be 6, 6

An **Optimisation Problem** requires not only to solve the problem, but also to produce the “best” solution (based on some quality criteria).

For different optimization problems, greedy methods can always work, sometimes work, give poor optimization but correct, or not work at all.

Greedy algorithms are very fast algorithms, as no operations are undone

Greedy used in

* Pathfinding - Dijkstra
* Job Scheduling
* Spanning trees of a graph
* Knapsack problems

The Greedy algorithm worked well when applied to a sensible, familiar currency in the change problem, but how do we solve the general case?

**Dynamic Programming**

***My understanding of DP is fairly hazy, so I’d appreciate if someone could produce a better explanation of this, thanks***

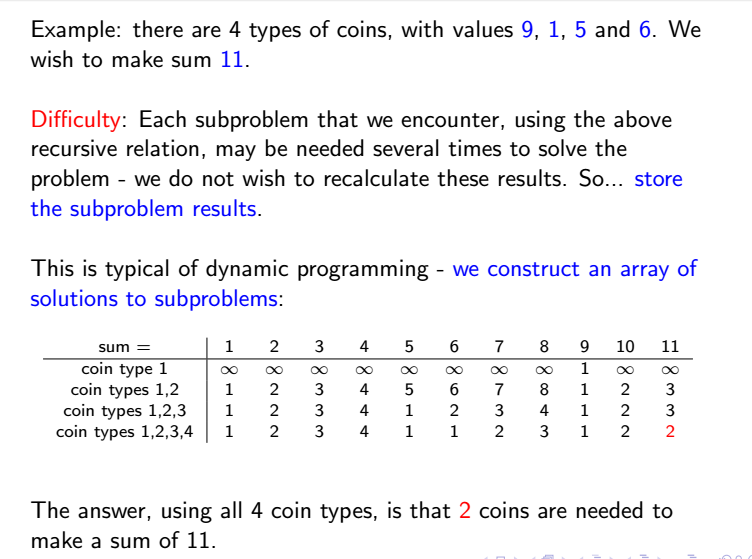
Dynamic Programming always produces an optimal solution to the coin problem.

If we have a 3p coin, we check whether using 1 3p is optimal, then 2p etc

This method requires already having all the solutions on the table, otherwise we repeat.

DP is a method of generating solutions from other solutions. Requires remembering the solutions of smaller subproblems in an array

Is a bottom-up method. Solve all smaller problems first.



Complexity of O(nK), where K is the number of columns, and n is the number of rows (items to try). In Knapsack, K is W or C, the Weight Capacity of the knapsack.