Greedy Algorithms & Dynamic Programming

So far you have seen divide and conquer technique which recursively breaks down a problem into two or more subproblems of the same (or almost the same) type until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. Quick sort and merge sort are these types of algorithms. We prove these algorithms often by induction and obtain their running times by the Master theorem.

Backtracking or Branch-and-Bound technique is another approach for finding all (or some) solutions to a computational problem. They often incrementally build candidates to the solution recursively in each step and abandons each partial candidate (backtracks) as soon as it determines cannot possibly be completed to a valid (or optimum) solutions. The running times of these algorithms are often exponential (e.g., $O(2^n)$) and there are several techniques esp. in AI to improve their running times.

Greedy algorithms, unlike backtracking algorithms that try every possible solution, try to make locally optimal choice at each step with the hope of finding the global optimum. In other words, a greedy algorithm never reconsiders its choices. That is the reason that for many problems greedy algorithms fail to produce the solution or the optimal solution, e.g., say you have 25-cent, 10-cent and 4-cent coins and we want to change 41 cents, greedy produces 25, 10 and 4 and fails, though a backtracking algorithm can do by 25-cent and four 4-cents. However greedy algorithms are often very fast, unlike backtracking algorithms. Dijkstra's algorithm for shortest path (that we see later) is a greedy algorithm. Greedy coloring is another application.

Finally, Dynamic Programming is a method for solving problems by breaking them down into simpler subproblems. The main idea is as follows: In general to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. This is the place that dynamic programming saves time comparing to backtracking algorithms, since unlike backtracking, it seeks to solve each subproblem only once, thus reducing the number of computations. The proof of dynamic programming is often by induction.
A special knapsack problem: subset sum problem

Suppose we are given a knapsack and we want to pack it fully, if it is possible.

**The problem:** Given an integer \( k \) and \( n \) items of different sizes such that the \( i \)th item has an integer size \( s_i \); find a subset of the items whose sizes sum to exactly \( k \), or determine that no such subset exists.

**Greedy algorithm:** Always use the first (the largest) item that you can pack.

It fails: e.g., \( k = 13; \, n = 4; \, 6, 5, 4, 3 \).

Greedy picks 6 and 5 and fails since there is no 2, but we can pick 6, 4, 3.

**Backtracking algorithm:**

Brute-force or exhaustive search in this case.

\[ \text{BF}(n, k, \text{Sol}) \]

begin
  if \((n = 0 \text{ and } k = 0)\) return true;
  if \((n = 0 \text{ and } k > 0)\) return false; if \((k < 0)\) return false;
  return \( \text{BF}(n-1, k, \text{Sol}) \text{ or } \text{BF}(n-1, k-s_n, \text{Sol} \cup \{s_n\}) \)
end;

We call, at the beginning with \( \text{BF}(n, k, \emptyset) \) to get the answer. Since we try both cases at each stage, the running time in the worst case is \( \Omega(2^n) \).

**Dynamic Programming:**

Similar to backtracking, assume \( \text{DP}(n, k) \) is true if and only if we can construct \( k \) with numbers \( s_1, \ldots, s_n \). Then the recursion for \( \text{DP} \) is exactly the same as \( \text{BF} \).

However, we can improve the running time a lot by this observation: the total number of possible problems may not be too high: there are \( n \) possibilities for the first parameter and \( k \) possibilities for \( k \). Thus overall, we have only \( nk \) different sub-problems. Thus if we store all known results in an \( n \times k \) array, then we can compute each sub-problem only once. If we are interested in finding the actual subset, then we can add to each entry a flag that indicates whether the corresponding item was selected in that step. This flag then can be traced back from \( (n,k) \)th entry and the subset can be recovered.
DP(n,k) begins

if flag[n,k] < 0 then return flag[n,k]; // flag[n,k] is initially all -1;
if (n=0 and k=0) flag[n,k] = 1;
if (n=0 and k>0) flag[n,k] = 0; if (k<0) then flag[n,k] = 0;
if flag[n-1,k] then begin flag[n,k] = 1; sol[n,k] = 0 end
if flag[n-1,k-1] then begin flag[n,k] = 1; sol[n,k] = 1 end
return flag[n,k];

Another example: longest common subsequence (LCS):

A subsequence of a sequence is a sequence by deleting some elements without changing the order of the remaining elements. E.g. ADF is a subsequence of ABCDEF.

The problem: find the longest common subsequence of two sequences (strings) $a_1$,...,$a_n$ and $b_1$,...,$b_m$.

Again let LCS(i,j) be the length of LCS $a_1$,...,$a_i$ and $b_1$,...,$b_j$. Then

$$LCS(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ LCS(i-1,j-1)+1 & \text{if } a_i=b_j \\ \max(LCS(i,j-1), LCS(i-1,j)) & \text{if } a_i \neq b_j \\ \end{cases}$$

Again if we are not careful we have a brute-force backtracking algorithm with running time $O(2^{\min(n,m)})$. But if we use dynamic programming the running time is $O(nm)$.

Independent set (a set of vertices with no edges in a graph).

The problem: given a tree $T$, find the largest independent set in it. Without loss of generality assume the tree is rooted at $r$ and $T_i$ is the subtree rooted at node $i$. Again

$$IS(i) = \begin{cases} 1 & \text{if } i \text{ is a leaf} \\ \max(\sum_{j \text{ child of } i} IS(j), 1+ \sum_{k \text{ child of } i} IS(k)) \end{cases}$$

We can use brute force or dynamic programming. In the latter case the running time is $O(n)$, where $n$ is the number of vertices in tree $T$. 
Bin Packing Problem: packing different-sized objects into fixed-size bins using as few of the bins as possible.

The problem: let $x_1, x_2, ..., x_n$ be a set of real numbers each between 0 and 1; partition the numbers into as few subsets as possible such that the sum of numbers in each subset is at most 1.

Greedy algorithm: put $x_i$ in the first bin, and then for each $i$, to put $x_i$ in the first bin that has room for it, or to start a new bin if there is no room in any of the used bins. The algorithm is called First Fit.

Thm: The first fit algorithm uses at most twice the best possible number of bins.

Pf: First fit cannot leave two bins less than half full; otherwise the items in the second bin could be placed in the first bin. Thus the number of bins used is no more than twice the sum of the sizes of all items (rounded up). The theorem follows since the best solution cannot be less than the sum of all the sizes. Here if we match bins in pairs and put together $\frac{\#\text{bins}}{2} < \frac{\#\text{bins}}{2} \leq \frac{1}{2} x_i \leq \text{opt} \Rightarrow \#\text{bins} \leq 2\text{opt} + 1$.

There are algorithms for bin packing which give almost the best solutions and they are backtracking and dynamic programming but they are more involved.