4/4

Greedy Algorithms & Dynamic programming:

So far you have seen <u>divide and conquer technique</u> 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 sub-problems are then combined to give a solution to the original problem. Quicksort and merge sort are these types of algorithms. We prove these algorithms often by induction and obtain their running times by the Master theorem

* <u>Backtracking or Branch-and-Bound technique</u> 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 c cannot possibly be completed to a valid (or optimum) solutions. The running times of these algorithms are often exponential (e.g. 2^n) and there are several techniques esp. in AI to improve their running times.

* <u>Greedy algorithms</u> unlike Backtracking Algorithms that try every possible choice (solutions) 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 <u>many problems</u> 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 make change 41 cents; greedy produces 25, 10 and 4 and fails, though a backtracking algorithm can do by one 25-cents 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. <u>Greedy coloring</u> is another application.

* Finally <u>Dynamic programming</u> 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 (sub problems), 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 back-tracking, 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. more precisely

The problem: given an integer K and n items of different sizes such that the ith 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.

```
BF(n, K, sol)
begin
    if (n=0 and K=0) return true;
    if (n=0 and K>0) return false; if (K<0) return false;
    return (BF(n-1, K, sol) OR BF(n-1, K-S_n, sol U {S_n}))
end;
```

we call at the beginning with $BF(n, K, \emptyset)$ 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 Back-tracking assume $DP(n,k)$ is true if and only if we can construct k with numbers $s_1, \dots, s_n$. Then the recursion for DP is exactly the same as BF. However we can improve the running time a lot by this observation that 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 compute each sub-problem only once. If we are interested in finding the actual subset, then we can add to each entry a flag (sol) that indicates whether the corresponding item was selected in that step. this flag (sol) then can be traced back from (n,k)th entry and the subset can be recovered.

```
DP(n,k)
begin
  if flag[n,k] <> -1 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-Sn] then begin flag[n,k]:=1; sol[n,k]:=1 end
  return flag[n,k];
end;
```

Another example: <u>longest common subsequence</u> (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) $a_1, \ldots a_n$ and $b_1, \ldots b_m$.

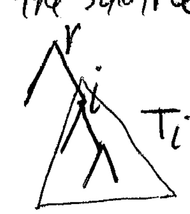
Again let $LCS(i,j)$ be the length of LCS $a_1, \ldots a_i$ and $b_1, \ldots b_j$. Then

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

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 $O(nm)$.

<u>Independent set</u> (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



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

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.

<u>Bin Packing Problem</u>: Packing different-sized objects into fixed-size bins using as few of the bins as possible.

The problem: let $x_1, x_2, \dots, x_n$ 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_1 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 <u>first fit</u>.

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

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 thm 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 ~~#bins~~ $\frac{\#bin-1}{2} \leq \left\lfloor \frac{\#bins}{2} \right\rfloor \leq \sum_{i=1}^{n} x_i \leq opt \Rightarrow \boxed{\#bin \leq 2opt+1}$)

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