# CMSC 351 - Introduction to Algorithms
# Spring 2012
# Lecture 16

**Instructor:** MohammadTaghi Hajiaghayi
**Scribe:** Rajesh Chitnis

## 1 Introduction

In this lecture we will look at Greedy Algorithms and Dynamic Programming.

## 2 Preliminaries

So far we have seen divide-and-conquer algorithms which recursively break down the 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. Quicksort and Mergesort are these types of algorithms. We prove the correctness of such algorithms often by induction and obtain their running times by the Master Theorem.

**Backtracking or Branch-and-Bound technique:** It 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 (backwards) as soon as it determines that it cannot be completed to a valid or optimum solution. The running times of these algorithms is often exponential (e.g. $2^n$) and there are several techniques especially in AI to improve their running times.

**Greedy Algorithms:** Unlike backtracking algorithms that try every possible choice (solutions), a greedy algorithm tries to make a locally optimal choice at each step with the hope of finding a global optimum. In other words, a greedy algorithm never reconsiders its choices. That is the reason for many problems greedy algorithms fail to produce the solution or the optimal solution. Say you have 25-cent,10-cent and 4-cent coins and we want to make change of 41 cents: Greedy produces25,10 and 4 and fails while a backtracking algorithm gives 25 and four 4 cent coins. However greedy algorithms are often very fast unlike backtracking algorithms. Djikstra's algorithm for shortest paths (that we see later) is a greedy algorithm. Greedy coloring is another application.

**Dynamic Programming:** It is a method for solving problems by breaking them down into simpler subproblems. The main idea is as follows: In general to solve a 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 compared to backtracking algorithms, since unlike backtracking algorithms, it seeks to solve each subproblem only once, thus reducing the number of computations. The proof of correctness for dynamic programming is often by induction.

# 3 A Special Knapsack Problem: Subset Sum

Suppose we are given a knapsack and we want to pack it fully, if it is possible. More precisely, we have the following 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.

## 3.1 Greedy Algorithm:

Always use the first (the largest) item that you can pack. This algorithm fails. Example is $k = 13, n = 4$ and the sizes as $6, 5, 4, 3$. Then greedy packs only 6 and 5, but we can pack 6, 4, and 3.

## 3.2 Backtracking Algorithm:

We do brute-force or exhaustive search in this case.

**Algorithm 1** BF($n$, $k$,sol)

1: **if** $n = 0$ and $k = 0$ **then**
2: **return** true;
3: **end if**
4: **if** $n = 0$ and $k > 0$ **then**
5: **return** false;
6: **end if**
7: **if** $k < 0$ **then**
8: **return** false;
9: **end if**
10: **return** $\Big($BF($n-1$, $k$,sol) OR BF($n-1$, $k-s_n$,sol$\cup\{s_n\}$)$\Big)$

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 in $\Omega(2^n)$.

### 3.3 Dynamic Programming

Similar to backtracking assume DP$(n, k)$ is true if and only if we can construct $k$ with numbers from $s_1, s_2, \ldots, s_n$. 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 problems may not be too high. There are $n$ possibilities for the first parameter and $k$ possibilities for the second parameter. Thus overall we only have $nk$ different subproblems. Thus if we store all known results in a $n \times k$ array then we compute each subproblem 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 or not. This flag (sol) can be traced back from the $(n, k)$-th entry and the subset can be recovered.

**Algorithm 2** DP$(n, k)$

```
1: Set all flag[n, k] to -1.
2: if flag[n, k] <> −1 then
3:    return flag[n, k];
4: end if
5: if n = 0 and k = 0 then
6:    flag[n, k] = 1;
7: end if
8: if n = 0 and k > 0 then
9:    flag[n, k] = 0;
10: end if
11: if k < 0 then
12:    flag[n, k] = 0;
13: end if
14: if flag[n − 1, k] = 1 then
15:    flag[n, k] = 1;
16:    sol[n, k] = 0;
17: end if
18: if flag[n − 1, k − s_n] = 1 then
19:    flag[n, k] = 1;
20:    sol[n, k] = 1;
21: end if
22: return flag[n, k];
```

## 4 Longest Common Subsequence (LCS)

A subsequence of a sequence is a sequence obtained by deleting some elements without changing the order of the remaining elements. For example, ADF is a subsequence of ABCDEF.

The problem is to find the LCS of two sequences (strings) given by $a_1, a_2, \ldots, a_n$

and $b_1, b_2, \ldots, b_m$. Again let $\text{LCS}(i,j)$ be the length of LCS of $a_1, a_2, \ldots, a_i$ and $b_1, b_2, \ldots, b_j$. Then

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

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

# 5 Independent Set in Trees

An independent set in a graph is a set of vertices such that no two of the vertices are adjacent. Given a tree we want to find a largest independent set in it. Without loss of generality we assume the tree is rooted at say $r$ and $T_i$ is the subtree rooted at node $i$. For every vertex $v$ let $C(v)$ denote the set of children of $v$. Then we have

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

Using dynamic programming, we have a running time of $O(n)$.

# 6 Bin Packing

Packing different-sized objects into fixed sized bins using as few of the bins as possible. Formally the problem is as follows: Let $x_1, x_2, \ldots, x_n$ be a set of real number 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.

## 6.1 Greedy Algorithm

Put $x_1$ in the first bin, and then for each $i$, put $x_i$ in the first bin that has room for it, or start a new bin if there is no room in any of the used bins. This algorithm is called as **First-Fit**.

**Theorem 1** *The First-Fit algorithm uses at most twice plus one bins than the best possible number.*

**Proof:** First-Fit cannot leave two bins less than half-full; otherwise the items in the second bin could have been placed in the first bin. Thus the number of bins used is no more than twice the sum of sizes of all items (rounded up). The theorem follows since the best solution cannot use than less than the sum of all the sizes. Suppose we use $b$ bins. If we match the bins in pairs and put together then we have $\frac{b-1}{2} \leq \lfloor \frac{b}{2} \rfloor \leq \sum_{i=1}^{n} x_i \leq \text{OPT} \Rightarrow b \leq 2 \cdot \text{OPT} + 1$. ■

There are algorithms for bin packing which give almost optimal solutions and there are also some backtracking and dynamic programming algorithms but they are more involved.

# References

[1] Udi Manber, *Introduction to Algorithms - A Creative Approach*