CMSC 351 - Introduction to Algorithms
Spring 2013

Instructor: Hamid Mahini

1 Introduction

In this lecture we will look at Dynamic Programming [1].

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.

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.

Greedy Algorithms: Unlike backtracking algorithms that try every possible choice (solutions), a greedy algorithm tries to make a locally optimal choice

[1] Thanks Prof. Hajiaghayi for sharing his notes for CMSC 351.

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 produces 25,10 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.

3 Computing Fibonacci Series

If we compute the Fibonacci series by a recursive algorithm, the running time will be an exponential function. The recursive algorithm solves a subproblem several times and results in an exponential running time. If we compute the Fibonacci series by a dynamic programming algorithm, the running time will be linear. The dynamic programming algorithm solves each subproblem once and saves the solution for the further usage.

4 Subset Sum Problem

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.

4.1 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.

4.2 Backtracking Algorithm:

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

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 $\Omega(2^n)$.

4.3 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

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 ( BF(n − 1, k,sol) OR BF(n − 1, k − s_n,sol∪{s_n}) )
```

$n \times k$ 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)

```
flag[0, 0] = 1
for j = 1 to k do
   flag[0, j] = 0
end for
for i = 1 to n do
   for j = 0 to k do
      flag[i, j] = flag[i − 1, j];
      sol[i, j] = 0;
      if s_i ≤ j flag[i − 1, j − s_i] = 1 then
         flag[i, j] = 1;
         sol[i, j] = 1;
      end if
   end for
end for
```

5 Knapsack Problem

We have n items where item i has a value v_i and an integer weight w_i. We also have a knapsack. The maximum weight that we can carry in the knapsack is k. The goal is to pick a subset of items to maximize the sum of the values of the items in the knapsack.

5.1 Dynamic Programming

Similar to the subset sum problem let $\mathtt{flag}[n, k]$ be the optimum solution, if we have a knapsack with capacity k and we can choose between items 1 to n. Then the recursion for DP is the same as the subset sum problem. 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.

Algorithm 3 KNAPSAK(n, k)

```
for j = 0 to k do
  flag[0, j] = 0
end for
for i = 1 to n do
  for j = 0 to k do
    flag[i, j] = flag[i − 1, j];
    sol[i, j] = 0;
    if w_i ≤ j flag[i − 1, j − w_i] + v_i > flag[i, j] then
      flag[i, j] = flag[i − 1, j − w_i] + v_i;
      sol[i, j] = 1;
    end if
  end for
end for
```

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

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

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 $O(nm)$.

References

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