CS 125 Section #3 Dynamic Programming and Divide and Conquer September 20, 2016

1 Divide and Conquer

We already saw in the *divide and conquer* paradigm how we can divide the problem into subproblems, recursively solve those, and combine those solutions to get the answer of the original problem.

Some examples of the divide and conquer paradigm are mergesort and binary search. When the subproblems are independent apart from needing to merge them together at the end, the problem may often lend itself to parallelization in the implementation.

Here's a somewhat different example:

Exercise. *Suppose you're given n points in the coordinate plane, and for convenience assume that no two of their x or y coordinates are the same. Design an algorithm that finds the shortest possible distance between a pair of these points. What is the time complexity of your algorithm? (You should assume that calculating the distance can be done in unit time.)*

Solution.
There is an obvious solution that takes n^2 distance calculations, but we can do much better.

We're going to use (not surprisingly) divide and conquer. First sort the points by x coordinate, and using that draw a vertical line l that splits them into two parts of roughly equal size.

Now find the shortest distances d_L and d_R on each side recursively. We have to see if any of the segments crossing l give us a shorter distance than $d := \min\{d_L, d_R\}$.

The key observation is that for each point on, say, the left side, there cannot be too many points that are closer than d to it on the right side - and in fact, at most a constant number.

To see this, it's best to think in terms of pictures. First, observe that only the points contained in the strips of width d on both sides of l are of interest; call these points S_L and S_R respectively. For a point $p_1 = (x_1, y_1) \in S_L$ of distance $\leq d$ to l, the points in S_R that have any chance to be within d of x are contained in a $2d \times d$ rectangle symmetric about the line $y = x_1$ and having one side on l.

Why can't there be too many points in this rectangle? It's because they are 'pushing away' from each other on the scale of d - the circles of radii $d/2$ centered at them are disjoint. In particular, there can be at most a constant number of them inside a $2d \times d$ rectangle!

Thus, we only need to check, for each such p_1, the distances to the several points on the right side that have y coordinate close to y_1. How do we do this efficiently? We sort $S_L \cup S_R$ by y coordinate, and check p_1's several neighbors in the ordering.

This gives a total running time of

$$\begin{aligned} T(n) &= \underbrace{O(n \log n)}_{\text{sort by } x_i} + \underbrace{2T(n/2)}_{\text{recursive call}} + \underbrace{O(n)}_{\text{find } S_L, S_R} + \underbrace{O(n \log n)}_{\text{sort } S_L \cup S_R \text{ by } y_i} + \underbrace{O(n)}_{\text{check close neighbors}} \\ &= 2T(n/2) + O(n \log n) \end{aligned}$$

which gives a total of $T(n) = O(n \log^2 n)$.

Follow-up: Can you improve the running time? Are we doing any unnecessary work?

2 Dynamic Programming

But for some problems, divide and conquer ends up having exponential time complexity because we will be recomputing the answer of same subproblems over and over again. To avoid recalculations, we use a lookup table. And to make it more effectively, we build the lookup table in a bottom-up fashion, until we reach the original problem.

Usually, any problem where you can write a recursion for the solution (not just for the runtime) is a good dynamic programming candidate.

Here's a good template (followed up by a runtime and/or space analysis) for writing solutions to dynamic programming problems:

1. Define a set of subproblems that can lead to the answer of the original problem.
2. Write the recursion solution of the original problem from the solution of subproblems.
3. Build the solution lookup table in a bottom-up fashion, until reaching the original problem.
   - Initialize the lookup table.
   - Fill the other terms by recursion.

Some things to keep in mind:

- To figure out how to write down the recursion, it's often helpful to think about how to divide into subproblems–along what axis are your subproblems "smaller" than the original?
- In many cases, the size of your lookup table is greater than the actual space complexity, since it may be enough to keep a row or two of the table in memory rather than the entire thing.

2.1 Making change with coins

Exercise. *Given a monetary system with M coins of different values $c_1, c_2, \ldots, c_M$, devise an algorithm that can make change of amount N using the minimum number of coins. What is the complexity of your algorithm?*

Solution.
Note that the greedy algorithm does not work here. (Exercise: can you come up with a counterexample?)

Instead, we give a dynamic programming solution:

1. Define $X[n]$ is the minimum number of coins needed to make change of amount n.
2. The recurion will be

$$X[n] = \min\{X[n-c_1], X[n-c_2], \cdots, X[n-c_M]\} + 1$$

Note that the minimum only consider coin c_m if $n \geq c_m$. If $n < c_m$, $X[n - c_m]$ is considered as infinity.

3. Initialize $X[0] = 0$, and fill up the other $X[n]$ until $n = N$.

The complexity of the algorithm is $O(MN)$.

Exercise (Extension)**.** *Change your algorithm to compute the number of different ways of making changes for amount N with the given coins (where the order of the coins doesn't matter).*

Solution.
Define subproblem $X(m, n)$ be the number of different possible ways to make changes of amount n using coins $\{c_1, c_2, \dots, c_m\}$. Then we can write the recurrsion

$$\begin{aligned} X(m, n) =& X(m-1, n) + X(m-1, n-c_m) + X(m-1, n-2c_m) + \cdots \\ =& \sum_{0 \leq k \leq \lfloor n/c_m \rfloor} X(m-1, n-kc_m) \end{aligned}$$

The equation means to make changes of amount n with $\{c_1, c_2, \dots, c_m\}$, we can either use $0c_m, 1c_m, 2c_m, \cdots, \lfloor n/c_m \rfloor c_m$, and use the $\{c_1, c_2, \dots, c_{m-1}\}$ to make changes for the rest of them.

We also need to be careful about the initial condition, which can be cleverly set as

$$X(0,0) = 1, X(0, n) = 0, \forall n \geq 1$$

2.2 Robot on a grid

Exercise (Basic problem)**.** *Imagine a robot sitting on the upper-left corner of an $M \times N$ grid. The robot can only move in two directions at each step: right or down. Write a program to compute the number of possible paths of the robot to the lower-right corner. What is the complexity of your program? (You can assume that integer multiplication can be done in constant time!)*

Solution. 1. Define $X[m, n]$ is the number of possible paths of getting square (m, n).

2. The recurion will be
$$X[m, n] = X[m-1, n] + X[m, n-1]$$

3. Initialize $X[1, n] = X[m, 1] = 1$, and fill up the other $X[m, n]$ until $m = M, n = N$.

The complexity of the algorithm is $O(MN)$.

Exercise (Mathematician's solution)**.** *Can you derive a mathematical formula to directly find the number of possible paths? What is the complexity for a computer program that computes this formula?*

Solution.
The robot needs to move $(M-1)+(N-1)$ steps to get the lower-right corner, among which $M-1$ should be downwards and $N-1$ should be rightwards. So the number of different paths is exactly the number of possible ways to pick $M-1$ downward steps among the $(M-1)+(N-1)$ steps in total. But this is

$$\binom{(M-1)+(N-1)}{M-1} = \frac{((M-1)+(N-1))!}{(M-1)!(N-1)!}$$

The complexity of computing this mathematical formula is $O(M+N)$.

Exercise (Extension)**.** *Imagine that certain squares on the grid are occupied by some obstacles (probably your fellow robots, but they don't move), change your program to find the number of possible paths to get the lower-right corner without going through any of those occupied squares.*

Solution.
The first and second step is the same as the basic problem. In the third step, we need to initialize the occupied squares to be zero, and remember not to update them by recursion.

2.3 Balanced partition

Exercise (A wedge: positive integer subset sum problem)**.** *Given a set of M positive integers $A = \{a_1, a_2, \ldots, a_M\}$, and a number N, design a program to find out whether it is possible to find a subset of A such that the sum of elements in this subset is N.*
Example: $A = \{1, 4, 12, 20, 9\}$ and $N = 14$, we can find a subset $\{1, 4, 9\}$ such that $1 + 4 + 9 = 14$.

Solution. 1. Define

$$X(m,n) = \begin{cases} \text{true} & \text{if it is possible to find a subset of } \{a_1, a_2, \ldots, a_m\} \text{ such that the element sum is } m \\ \text{false} & \text{otherwise} \end{cases}$$

2. The recursion will be

$$X(m,n) = \{X(m-1,n) \text{ or } X(m-1, n-a_m)\}$$

Note that we only consider the second term if $n \leq a_m$.

3. Initialize $X(m,0) = \text{true}, \forall m \geq 0; X(0,n) = \text{false}, \forall n \geq 1$, and fill up the other $X(m,n)$ until $m = M, n = N$.

The complexity of this algorithm is $O(MN)$.

Exercise (Balanced partition problem)**.** *Given a set of M positive integers $A = \{a_1, a_2, \ldots, a_M\}$, find a partition A_1, A_2, such that $|S_1 - S_2|$ is minimized, where S_i is the sum of elements in subset A_i. (A partition of A means two subset A_1, A_2 such that $A = A_1 \bigcup A_2$ and $A_1 \bigcap A_2 = \emptyset$.)*
Example: $A = \{1, 4, 12, 20, 9\}$.The best balanced partition we can find is $A_1 = \{1, 12, 9\}, S_1 = 1+12+9 = 22$ and $A_2 = \{4, 20\}, S_2 = 4 + 20 = 24$.

Solution.
Let $S = \sum_{m=0}^{M} a_m$. Fill in the lookup table $X(m,n)$ in previous problem until $m = M, n = \lfloor S/2 \rfloor$, and find the maximum n such that $X(M,n)$ is true. Then let A_1 be the corresponding subset of the a_i (how can we modify the algorithm in order to be able to efficiently find that?) and A_2 be the complement.

It can be easily shown that $A = A_1 \cup A_2$ is a minimizer for $|S_1 - S_2|$.

2.4 Palindrome

A palindrome is a word (or a sequence of numbers) that can be read the same say in either direction, for example "abaccaba" is a palindrome.

Exercise (Palindrome)**.** *Design an algorithm to compute what is the minimum number of characters you need to remove from a given string to get a palindrome.*
Example: you need to remove at least 2 characters of string "abbaccdaba" to get the palindrome "abaccaba".

Solution.
1. Let N be the length of S. Define $X(i,j)$ the minimum number of characters we need to remove in order to make substring S[i,i+1,...,j] a palindrome.
2. The recurion will be
$$X(i,j) = \begin{cases} X(i+1, j-1) & \text{if S[i]=S[j]} \\ \min\{X(i+1,j), X(i,j-1)\} + 1 & \text{otherwise} \end{cases}$$
3. Initialize $X(i,i) = 0 = X(i+1,i), \forall i \geq 0$, and fill up the other $X(i,j)$ until $i = 0, j = N-1$.

The complexity of this algorithm is $O(N^2)$.