CS 218 : Design and Analysis of Algorithms

Lecture 11: Introduction to Dynamic Programming

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering Indian Institute of Technology, Bombay

1 Computing $\binom{n}{k}$ by additions

Consider the following problem.

As input we are given two positive integers n & k. We wish to compute $C(n,k)$, i.e., the number of subsets of $1,2,\dots,n$ of size k. We wish to do this using only additions and we would like to do this from first principles.

Our first step, as in the design of many algorithms, is a recursive solution.

What does this mean? We need to write $C(n,k)$ in terms of $C(n',k')$ where either $n' < n$ and $k' \leq k$ or $n' \leq n$ and $k' < k$.

A well known and natural way to derive such a relation is to use the following idea: either a subset of size k contains the element 1 or it does not. We will count both separately (by induction or recursion) and add them up. The number of subsets of size k that do not contain 1 is $C(n-1,k)$ and the number of subsets which contain 1 is $C(n-1,k-1)$. Hence, $C(n,k) = C(n-1,k) + C(n-1,k-1)$.

When the output desired by the problem is a subset of a set, such an argument is typical. Either the first element is part of the answer or not. This splits the space in two and one can recurse to get the two solutions and if necessary put them together.

The base cases are $C(n,n) = 1$ and $C(n,0) = 1$.

On input n and k, how many additions does this perform? This is not our usual recurrence and our normal ways of solving it may not yield what we are looking for. However, the answer is

$$C(n,k) - 1$$

and can be checked easily by induction on n. *Exercise: Do this.*

This running time is not acceptable. For $n = 1000$ and $k = 500$, the program will not terminate in our lifetime (to put it mildly) even if all the computers in the world are harnessed to do this.

What is the problem? And is there a fix to this problem? An inkling to the problem shows up when we expand the recursion. It is generally a good idea to expand the recursion at least twice.

Expanding once yields

$$C(n,k) = C(n-1,k) + C(n-1,k-1).$$

Expanding each term once more yields

$$C(n,k) = C(n-2,k) + C(n-2,k-1) + C(n-2,k-1) + C(n-2,k-2).$$

Notice that $C(n-2,k-1)$ is called twice. This phenomenon gets worse inside the recursion. We are recalculating values. It is then natural to store values of $C(n',k')$ which have been computed and reuse them when needed. This brings us to the next crucial question. **How many distinct calls are made during the recursion?**. There are n possible values of first coordinate and k possible for the second. So there are at most nk possible distinct calls. We will requisition a table $T(i, j) 1 \leq i \leq n$ and $1 \leq j \leq k$ of size nk. $T(i, j)$ will be initialised to -1 and will contain the value of $C(i, j)$ at the end of execution.

Here is the modified procedure.

Initialise a table $T(i, j) = -1$, for $1 \leq i \leq n; 1 \leq j \leq k$. Initialise $T(i, 0) = 1$ and $T(i, i) = 1$, for each valid i.

$$C(n, k)$$

if $T(n, k) = -1$ then

$$T(n, k) = C(n-1, k) + C(n-1, k-1)$$

Return $T(n, k)$

How much time does this procedure take?

There are two operations that dictate the running times. The number of additions and the number of accesses to $T(i, j)$. The number of additions is at most nk since each time we perform an addition we update a table entry and there are at most nk table entries.

Once $T(i, j)$ is filled it will be accessed at most twice. Once for filling $T(i+1, j)$ and the other while filling $T(i+1, j+1)$. This yields the $O(nk)$ bound on the running times. By looking at the program, can you bound the number of accesses using the number of additions?

2 Key Steps

We flesh out the key points in the design of this algorithm.

1. Give your procedure a name and clearly describe the input and output.

2. Write a recursive procedure.

3. Determine the number of distinct calls.

4. Allocate a table, with one entry per distinct call.

After this the rest is routine. We initialise the table. And recurse only if the table entry has not been filled. Since each time we recurse we fill in a table entry, the runing times is usually an easy function of the size of the table.

There is just one complication we may have to deal during this process. That is the number of distinct calls may be exponential. In such a case there are some standard tricks one can try and these are explained in forthcoming lectures using other examples. In this case too, if you look closely we could have ended up with an exponential number of distinct calls. The different calls with the same value of n and k are actually over different subsets. The key (and in this case trivial) observation (which we had made implicitly) is that the value only depended on the size of the sets, not on the sets themselves.

This entire process goes under the fancy name of *Dynamic Programming.*