Dynamic Programming

Dudu Amzallag
amzallag@cs.technion.ac.il

June 2007

1 Computing binomial coefficients $\binom{n}{k}$

Consider the problem of computing the *binomial coefficient*

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \tag{1}$$

for a given non-negative integers n and k.

The problem of implementing directly Equation 1 is that the factorials grow quickly with increasing n and k. For example, $13\,! = 6\,227\,020\,800 > 2^{32}$. Therefore, it is not possible to represent, within 32 bits word, the binomial coefficients $\binom{n}{k}$ up to $n = 34$ without overflowing (notice that $\binom{34}{17} = 2\,333\,606\,220 < 2^{32}$).

Consider the following recursive definition of the binomial coefficients:

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \\ 1 & n = k \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise} \end{cases} \tag{2}$$

This formulation does not require the computation of factorials. In fact, the only computation needed is addition. If we implement Equation 2 directly as a recursive function, we get a routine whose running time is given by

$$T(n,k) = \begin{cases} O(1) & k = 0 \\ O(1) & n = k \\ T(n-1,k) + T(n-1,k-1) + O(1) & \text{otherwise} \end{cases}$$

which is very similar to Equation 2. In fact, we can show that $T(n,k) = \Omega\left(\binom{n}{k}\right)$ which (by Equation 1) is not a very good running time at all. Again the problem with the direct recursive implementation is that it does far more work that is needed because it solves the same subproblem many times.

An alternative to the top-down recursive implementation is to do the calculation from the bottom up. In order to do this we compute the following series of sequences

$$S_i = \left\{ \binom{i}{k} \;:\; k = 0, 1, \ldots, i \right\},$$

for every $i = 1, 2, \ldots, n$.

Notice that we can compute S_{i+1} from the information contained in S_i simply by using Equation 2. This method of representation the binomial coefficients also called *Pascal's triangle.*

The following dynamic programming algorithm calculates the binomial coefficient $\binom{n}{k}$ by computing Pascal's triangle. According to Equation 2, each subsequent row depends only on the preceding row – it is only necessary to keep track of one row of data. The implementation shown uses array of length $n+1$ to represent a row of Pascal's triangle. Consequently, instead of table of size $(n+1) \times (k+1) = O(n^2)$, the algorithm gets by with $O(n)$ space.

Algorithm. BINOM (n, k)

```
1: $A[0] \leftarrow 1$
2: for $i \leftarrow 1$ to $n$ do
3:     $A[i] \leftarrow 1$
4:     for $j \leftarrow i-1$ downto 1 do
5:         $A[j] \leftarrow A[j] + A[j-1]$
6: return $A[k]$
```

The running time of the above algorithm is clearly $O(nk)$. Is this time polynomial in the size of the input?

2 The independent set problem on trees

Consider the *independent set* problem on trees. The input is a tree $G = (V, E)$. We wish to find a maximum independent set (a set of vertices no two of which are adjacent) of G. This can be solved in linear-time using dynamic programming as follows.

Fix an arbitrary vertex r as the root of the tree, and orient all edges away from r. The subtree rooted at v, denoted by $T(v)$, includes v and all vertices reachable from v under this orientation of edges (hence, $T(r) = G$). The *children* of v, denoted by $C(v)$ are all those vertices u with directed edge (v, u). Leaves of the tree have no children.

Let $S(T(v))$ denote the size of a maximum independent set of $T(v)$. We want to compute $S(T(r))$. Let $S^+(T(v))$ denote the size of a maximum independent set of $T(v)$ that contains v, and let $S^-(T(v))$ denote the size of a maximum independent set of $T(v)$ that does not contain v. Then we have

$$\begin{aligned} S^+(T(v)) &= 1 + \sum_{u \in C(v)} S^-(T(u)) \\ S^-(T(v)) &= \sum_{u \in C(v)} \max\left\{S^-(T(u)),\, S^+(T(u))\right\} \end{aligned}$$

The dynamic program now works by repeating the following procedure:

1. Find a vertex v such that for all of its children $u \in C(v)$ we have already computed $S^-(T(u))$ and $S^+(T(u))$.

2. Compute $S^-(T(v))$ and $S^+(T(v))$ as described above.

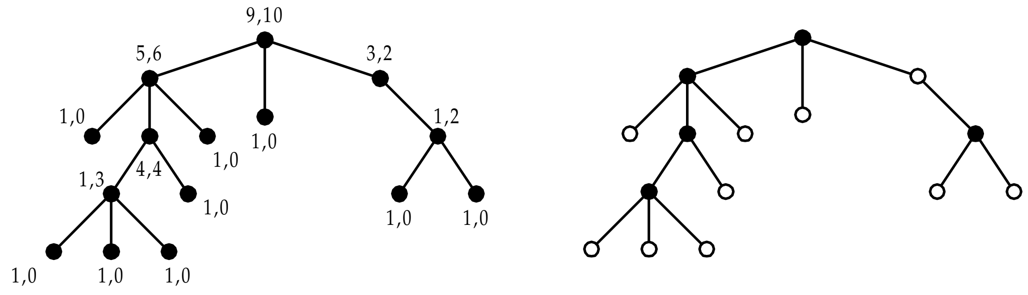

Figure 1: A rooted tree in which each vertex v is labelled with the pair $S^+(T(v)), S^-(T(v))$ (left), and the maximum independent set of the tree denoted by the *white* vertices (right).

Eventually, output $\max\{S^-(T(r)),\ S^+(T(r))\}$. Note that as long as there are unvisited vertices, these vertices form a tree and hence have a leaf, and this leaf can be chosen in step 1 of the above procedure. Hence the algorithm never gets stuck. Obviously, this algorithm can be made to run in linear-time. A particular example is shown in Figure 1; there, the size of the maximum independent set in T is 10.

3 The knapsack problem

One of the most classical problems in combinatorial optimization is the *knapsack* problem. Given is a knapsack of capacity W and n objects $a_1, a_2, \ldots, a_n$ having weights $w_1, w_2, \ldots, w_n$ and profit values $p_1, p_2, \ldots, p_n$, respectively. We want to select some subset of these objects to be placed in the knapsack, so that the total profit of the objects in the knapsack is maximized, while not violating its capacity constraint.

Dynamic programming algorithm. Define $F(i, w)$ to be the largest profit attainable by selecting some among the i first objects $\{a_1, a_2, \ldots, a_i\}$, so that their total weight is at most w, $w = 0, 1, \ldots, W$. It is easy to see that the $(n+1)(W+1)$ entries of the $F(i, w)$ table can be computed in order of increasing i, and with a constant number of operations per entry, as follows:

$$F(i+1, w) = \begin{cases} F(i, w) & \text{if } w_{i+1} > w \\ \max\{F(i, w),\ p_{i+1} + F(i, w - w_{i+1})\} & \text{otherwise} \end{cases}$$

This means that the best subset of objects $\{o_1, o_2, \ldots, o_{i+1}\}$ (in the sense of total sum of profits) $F(i+1, w)$ that has the total weight w, either contains object $i+1$ or not, starting with $F(0, w) = 0$ for all w. Two cases are possible:

1. $\boldsymbol{w_{i+1} > w}$. Object $i+1$ cannot be part of the solution, since if it was, the total weight would be larger than w, which is unacceptable.

2. $\boldsymbol{w_{i+1} \leq w}$. Object $i+1$ *can* be in the solution, and we choose the option with the greater profit.

Analysis. The $O(nW)$ running time of the above algorithm is *not* polynomial in the size of the input. Notice that the input size can only be bounded by $O(n \log P + n \log W)$, assuming $p_i \leq P$ and $w_i \leq W$, for all $i = 1, 2, \ldots, n$. Thus, this is not a polynomial time algorithm for knapsack (tough it can be quite effective if the numbers involved are not too large); this is a *pseudo-polynomial time* algorithm.

Example. Given are 4 pairs of (weight, profit) items: $(5, 10), (4, 40), (6, 30)$ and $(3, 50)$ while $W = 10$.

$i\,/\,w$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	**90**

Q. How can we describe which subset gives the optimal solution (items 2 and 4 in our example)?

Note. The *fractional* version for this problem, that is, the case we are allowed to use arbitrary fractions of an object. It is very interesting to know that this version of the problem can be *optimally* solved using a $O(n \log n)$ greedy algorithm. In this case, compute the value $\frac{p_i}{w_i}$ for every object and sort the objects by these values. Now fill the knapsack greedily as you can (take objects in order).

For example, consider 3 pairs of (weight, profit) items: $(10, 60), (20, 100)$ and $(30, 120)$ and $W = 50$. The greedy algorithm for the fractional version yields a total profit of 240 while the one for the 0/1 version outputs total profit of 220.

4 The subset sum problem

Given n items of "size" $a_1, a_2, \ldots, a_n$ (positive integers) and a non-negative integer bound B, we wish to decide whether there is a subset $S \subseteq \{1, 2, \ldots, n\}$ of the items such that $\sum_{i \in S} a_i = B$. The problem has the following recursive solution:

Base case: $n = 0$. The answer is *no*, unless $B = 0$.

General case: The answer is *yes* if and only if

- either there is a subset $S \subseteq \{1, 2, \ldots, n-1\}$ such that $\sum_{i \in S} a_i = B$ (in which case S is the solution), or
- there is a subset $S \subseteq \{1, 2, \ldots, n-1\}$ such that $\sum_{i \in S} a_i = B - a_n$ (in which case $S \cup \{n\}$ is the solution).

A divide-and-conquer algorithm based on this recursive solution as a running time given (why ?) by the recurrence

$$T(n) = 2T(n-1) + O(1), \quad T(1) = O(1)$$

which gives $T(n) = \Omega(2^n)$ (again, why?).

However, there are only Bn problems that one needs to solve. Namely, it is enough to compute for every $1 \leq k \leq n$ and every $1 \leq B' \leq B$ whether there is a subset $S \subseteq \{1, 2, \ldots, k\}$ such that $\sum_{i \in S} a_i = B'$.

A dynamic programming algorithm computes a boolean matrix M having the property that $M[k, B'] = T$ if and only if there is a subset $S \subseteq \{1, 2, \ldots, k\}$ such that $\sum_{i \in S} a_i = B'$. The algorithm computes the matrix iteratively. Initially, the first row of the matrix is initialized by setting $M[0, B'] = F$ for $B' \neq 0$ and $M[0, B'] = T$ otherwise. Then the matrix is filled for k going from 2 to n and for B' going from 1 to B. The entry $M[k, B']$ is computed as the OR operation of the entries $M[k-1, B']$ and $M[k-1, B'-a_k]$.

Each entry is computed in constant time, therefore the matrix is filled in $O(nB)$. Once the matrix is filled, the answer to the problem is contained in $M[n, B]$.

5 The reboot scheduling problem

You are working for a company that runs a server that is accessed by millions of customers per day. Let us suppose that you have an estimate, $x_1, \ldots, x_n$, of the numbers of customers (in millions) that you expect to access your server over the next n days. Now the server software is not very well written, and the number of customers that it can handle per day decreases with each day since the most recent reboot. Let s_i denote the number of customers that it can handle on day i after the last reboot, where we assume $s_1 > s_2 > \ldots, > s_n$. If you can choose to reboot on a given day, you cannot serve any customers on that day.

Given the expected loads $x_1, \ldots, x_n$, and the limits $s_1, \ldots, s_n$, the REBOOT SCHEDULING PROBLEM is to find a plan that specifies the days on which you will reboot the server so as to maximize the total number of customers that you can serve.

We solve this problem backwards via a dynamic programming algorithm. Let $C(i, j)$ denote the optimal (maximum) number of customer that can be served between days i and n, inclusive, assuming that the server was last rebooted on day $i - j$ (i.e., last rebooted j days ago), and also has rebooted on day 0. The final number we want is $C(1, 1)$.

The base cases are $C(n+1, j) = 0$ for all j. Now we consider how to compute $C(i, j)$. We have two scenarios at day i: we can either reboot the server (with 0 productivity on day i), or not (with productivity $\min\{x_i, s_i\}$ on day i). Thus we have the following relation:

$$C(i, j) = \max \begin{cases} C(i+1, 1), & \text{if we reboot the server on day } i. \\ C(i+1, j+1) + \min\{x_i, s_j\}, & \text{otherwise.} \end{cases}$$

We can compute $C(1, 1)$ by filling the table (of size n^2) backwards (starting with $C(n, n)$). We fill in $O(n^2)$ table elements, each taking constant time to compute. Hence the total running time is $O(n^2)$.