# Dynamic Programming

Dudu Amzallag[*]

## 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

[*]E-mail: `amzallag@cs.technion.ac.il`, Office Hours: Sundays, 15:00-16:00, Phone: 04-8294959.

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.

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.

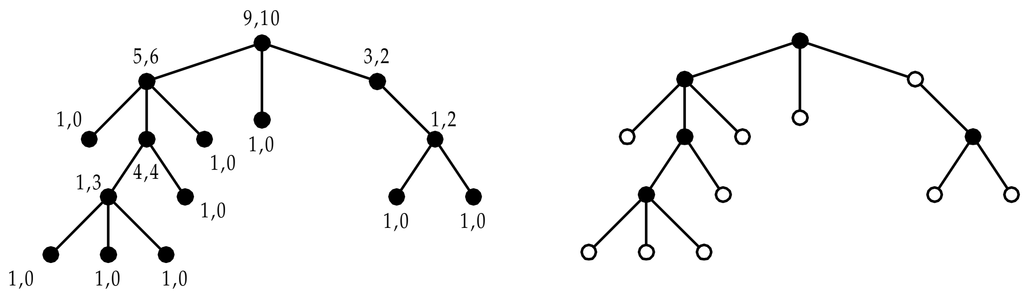

Figure 1: A rooted tree in which each vertex $v$ is labelled with the pair $S^+(T(v)), S^-(T(v))$ (left), and the maximum independent set of the tree denoted by the *white* vertices (right).

## 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 List Coloring Problem on Trees

Given is a graph $G = (V, E)$ and an assignment $\mathcal{S} = \{S(v) \mid v \in V\}$ of lists of admissible colors for its vertices. The *list coloring problem* is to decide whether the vertices of $G$ can be properly colored (i.e., adjacent vertices receive different colors) so that each vertex $v$ is colored with a color from $S(v)$. Clearly, this problem is a natural generalization of the "standard" graph coloring problem (why ?). It turns out that when the input graph is a tree, this problem can be solved in linear time using a dynamic programming algorithm. Before presenting the algorithm a short reminder is required.

A graph $G = (V, E)$ is *bipartite* if its vertex set can be partitioned into two sets $V_1$ and $V_2$ (i.e., $V_1 \cap V_2 = \emptyset$ and $V = V_1 \cup V_2$) in such a way that no two vertices from the same set are adjacent. In fact a graph being bipartite means that the vertices of $G$ can be colored with at most two colors, so that no two adjacent vertices have the same color. Since we already saw that a graph is bipartite if and only if it contains no odd cycle, any tree is 2-colorable.

We are now ready to handle the list coloring problem for trees. We assume that the tree $T = (V, E)$ is given with root $r \in V$ that has no predecessor, with a set of children $C(v) \subset V$ for each vertex $v \in V$ and an unique parent for each vertex $v \in V - \{r\}$. A *leaf* of $T$ is a vertex $v$ that has no children. The other vertices are called non-leaves or *inner vertices* of $T$.

We determine two values $\mathit{unique}(v) \in \{\text{TRUE}, \text{FALSE}\}$ and $\mathit{color}(v) \in \{0, 1, 2, \ldots, m\}$ for each vertex $v \in V$. If the color is unique at a vertex $v$ and cannot be changed in a feasible coloring of the subtree rooted at vertex $v$, then we set $\mathit{unique}(v) = \text{TRUE}$; otherwise we set $\mathit{unique}(v) = \text{FALSE}$. At the beginning, the unique colors for vertices with $S(v) = \{c\}$ are stored in $\mathit{color}(v)$; for other vertices we set $\mathit{color}(v) = 0$.

The vertices of the tree are traversed in postorder. If a child $y \in C(v)$ of an inner vertex $v$ has a unique color $c$, then this color cannot be used for $v$. We denote the unique colors of the children of $v$ by a set $forbidden(v)$. After that, we determine the values $\mathit{unique}(v)$ and $\mathit{color}(v)$. The input of the following algorithm is a tree $T = (V, E)$, and feasible color sets $S(v)$ with $|S(v)| \geq 1$ for all $v \in V$. The algorithm output YES, iff there exists a feasible list coloring of T.

**Algorithm.** LIST COLORING OF TREES

```
1: for all $v \in V$ in a postorder traversing of $T$ do
2:     $forbidden(v) = \emptyset$;
3:     for all $y \in C(v)$ do
4:         if $unique(y) = $ TRUE then
5:             $forbidden(v) = forbidden(v) \cup \{color(y)\}$;
6:     if $|S(v) - forbidden(v)| \geq 2$ then
7:         $unique(v) = $ FALSE;
8:         $color(v) = 0$;
9:     if $S(v) - forbidden(v) = \{c\}$ then
10:        $unique(v) = $ TRUE;
11:        $color(v) = c$;
12:    if $S(v) - forbidden(v) = \emptyset$ then
13:        return NO;
14: return YES.
```

To show the correctness of the algorithm, it is sufficient to prove the next lemma.

**Lemma 1** *For a fixed inner vertex $u \in V$ let $\lambda(u) = \{y \in C(u) \mid unique(y) = \text{FALSE}\}$ be the set of its children that are not uniquely colored, and let $forbidden(u)$ be the set of colors that are determined uniquely on the other children from $C(u) \setminus \lambda(u)$. Then, every coloring of vertex $u$ with color $f(u) \in S(u) \setminus forbidden(u)$ can be extended to the children of $u$.*

**Proof.** Since $unique(y) = \text{FALSE}$ for each vertex $y \in \lambda(u)$, there are at least two feasible colors $f_1(y)$ and $f_2(y)$ in colorings for the subtree rooted at $y$. Then, given a coloring $f(u)$ of $u$, at least one of those two colors is unequal to $f(u)$ and can be used. ■

The travel in postorder through the tree needs $O(|V|)$ time. The computation time for one set $forbidden(v)$ at a vertex $v \in V$ can be bounded by $O(|C(v)|)$. The decision whether one or two colors are in $S(v) \setminus forbidden(v)$ can be done in $O(|forbidden(v)| + 2)$ time. Since each vertex in the tree has at most one predecessor and since $|forbidden(v)| \leq |C(v)|$, the entire algorithm needs at most $O(|V|)$ time. It needs $O(\sum_{v \in V} |S(v)|)$ space in the worst case.

A feasible coloring, if it exists, can also be found in linear time. During the algorithm (Lines 6–8) we compute two possible colors $f_1(v)$ and $f_2(v)$ at the vertices with $unique(v) = \text{FALSE}$. We can do this in $O(|forbidden(v)| + 2)$ steps for any such vertex $v \in V$. At the end of the algorithm we have to walk once again through the tree, now in preorder and compute one possible coloring in $O(|V|)$ time.