CSE331 Introduction to Algorithms
Lecture 17: Introduction to Dynamic Programming

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 23, 2021

Introduction

- This is an introductory lecture on *dynamic programming*.
- This is an important algorithms design technique.
- The next 3 lectures are on the same topic.
- This lecture is not in the textbook.

Binomial Coefficients

Definition

The *binomial coefficient* $\binom{n}{k}$ is the coefficient of x^k in the expansion of $(1+x)^n$.

Example

$$(1+x)^4 = 1 + 4x + 6x^2 + 4x^3 + x^4$$

$$\binom{4}{0} = 1 \qquad \binom{4}{1} = 4 \qquad \binom{4}{2} = 6 \qquad \binom{4}{3} = 4 \qquad \binom{4}{4} = 1$$

Binomial Coefficients

- $\binom{n}{k}$ reads *n choose k*. Reason:

Proposition

Every set with n elements has $\binom{n}{k}$ subsets of size k.

Example

The set $S = \{a, b, c, d\}$ has $\binom{4}{2} = 6$ subsets of size 2:

$$\{a,b\} \quad \{a,c\} \quad \{a,d\} \quad \{b,c\} \quad \{b,d\} \quad \{c,d\}$$

Expression

For every integers n, k such that $0 \leqslant k \leqslant n$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n}{k} \cdot \frac{(n-1)}{(k-1)} \cdot \frac{(n-2)}{(k-2)} \cdots \frac{(n-k+1)}{1} \tag{1}$$

Binomial Coefficients

Recurrence relation

For all integers n, k such that $1 \leqslant k \leqslant n-1$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{2}$$

and for all $n \geqslant 0$

$$\binom{n}{0} = \binom{n}{n} = 1.$$

- Proof?
- This relation allows us to compute the binomial coefficients using *Pascal's triangle*. (See next slide.)

Pascal's Triangle

n \ k	0	1	2	3	4	5	6	$\cdots$
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
$\vdots$	$\vdots$							$\ddots$

The green triangle shows the relation

$$\binom{5}{2} = \binom{4}{1} + \binom{4}{2}$$
$$= 4 + 6$$
$$= 10.$$

Naive Algorithm for Computing $\binom{n}{k}$

- Suppose we want to compute $\binom{n}{k}$ using the recurrence relation (2).

Pseudocode

```
procedure Binomial($n, k$)
    if $k = 0$ or $k = n$ then
        return 1
    else
        return Binomial($n-1, k-1$)+Binomial($n-1, k$)
```

- C program on next slide.

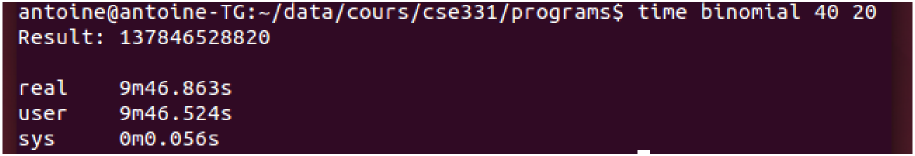
Naive Algorithm for Computing $\binom{n}{k}$

C Program

```
#include <stdio.h>

unsigned long long binomial(int n, int k){
  if ((k==0)||(k==n))
    return 1;
  return binomial(n-1,k-1)+binomial(n-1,k);
}

int main(int argc, char *argv[]){
    int n=atoi(argv[1]);
    int k=atoi(argv[2]);
    printf("Result: %llu\n",binomial(n,k));
}
```

Naive Algorithm for Computing $\binom{n}{k}$

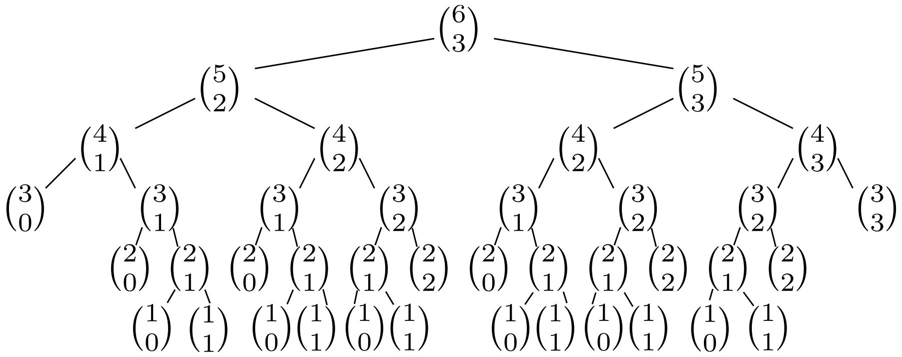
- Experiment: Computing $\binom{40}{20}$



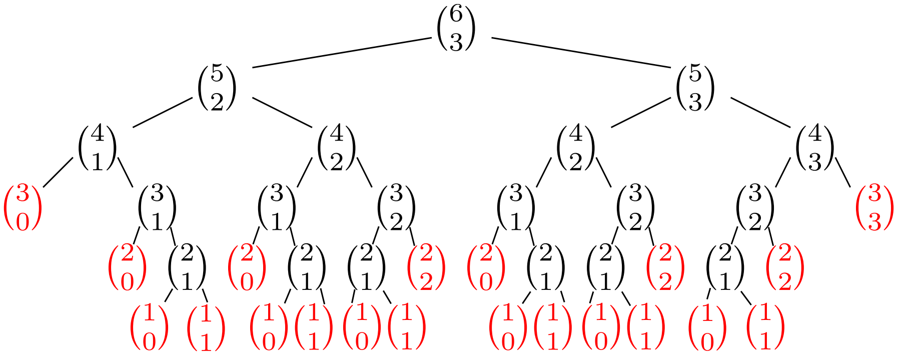
```
antoine@antoine-TG:~/data/cours/cse331/programs$ time binomial 40 20
Result: 137846528820

real	9m46.863s
user	9m46.524s
sys	0m0.056s
```

- It takes about 10 minutes. Why is it so slow?

Recursion Tree



Recursion Tree



- There are 20 leaves.
- $\binom{6}{3} = 20$.
- This is not surprising: The algorithm adds 1 to the result at each leaf.

Analysis

- The recursion tree has $\binom{n}{k}$ leaves.
- Its total number of nodes is $2\binom{n}{k} - 1$, because:

Theorem (CSE232)

A full binary tree is a tree such that each internal node has 2 children. A full binary tree with n leaves has $n-1$ internal nodes, and hence it has $2n-1$ nodes in total.

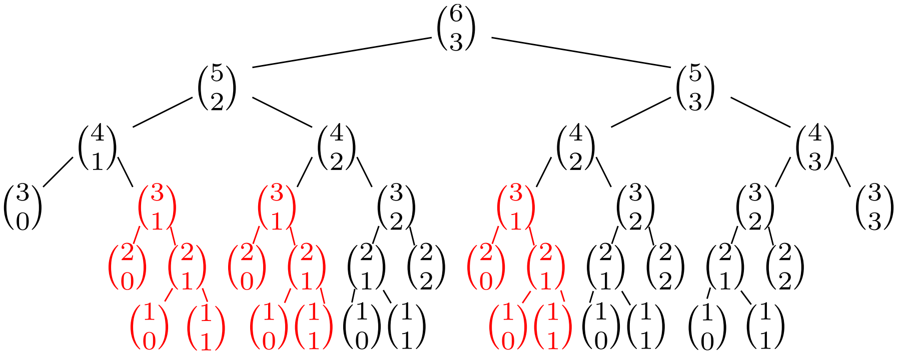
- It follows that:

Theorem

The naive algorithm (procedure BINOMIAL*) for computing $\binom{n}{k}$ runs in $\Theta\left(\binom{n}{k}\right)$ time.*

- This is very slow because $\binom{2n}{n} \geqslant 2^n$. (Proof?)

Recursion Tree



- Observation: BINOMIAL computes 3 times the subtree rooted at $\binom{3}{1}$.
- How can we avoid it?

Faster Algorithm

- We compute Pascal's triangle up to row n, and return $\binom{n}{k}$. (See Slide 7.)
- Intermediate results are stored in a table, so that we never recompute the same coefficient $\binom{i}{j}$.

Computing $\binom{n}{k}$ by *dynamic programming*

```
1: procedure DPBinomial($n, k$)
2:     $C[1 \dots n][0 \dots n] \leftarrow$ new bidimensional array
3:     for $i \leftarrow 1, n$ do
4:         $C[i][0] \leftarrow 1$
5:         $C[i][i] \leftarrow 1$
6:         for $j \leftarrow 1, i-1$ do
7:             $C[i][j] \leftarrow C[i-1][j-1] + C[i-1][j]$
8:     return $C[n][k]$
```

Dynamic programming

- This approach is called *dynamic programming*. It consists in storing intermediate results in a table, so that the solution to each subproblem is computed only once.
- It can make the algorithm exponentially faster.

Example

DPBinomial runs in $\Theta(n^2)$ time, while Binomial runs in $\Omega(2^n)$ time.

- Unfortunately, it does not always work. For a large class of problems (**NP**-hard problems, see later in this course), the best known algorithms are exponential-time.

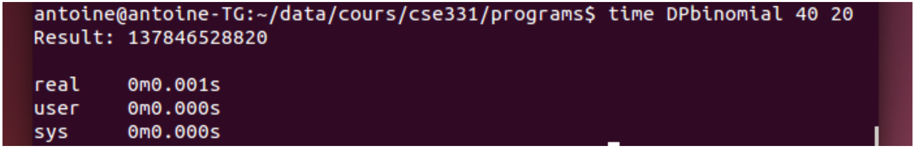
Computing $\binom{n}{k}$ by Dynamic Programming

C Function

```
unsigned long long DPbinomial(int n, int k){
  int i,j;
  unsigned long long C[n+1][n+1];
  for(i=1;i<=n;i++){
    C[i][0]=1;
    C[i][i]=1;
    for(j=1;j<i;j++)
      C[i][j]=C[i-1][j-1]+C[i-1][j];
  }
  return C[n][k];
}
```

Naive Algorithm for Computing $\binom{n}{k}$

- Computing $\binom{40}{20}$:



```
antoine@antoine-TG:~/data/cours/cse331/programs$ time DPbinomial 40 20
Result: 137846528820

real	0m0.001s
user	0m0.000s
sys	0m0.000s
```

- It takes less than 0.001 seconds.
- It is much better than the naive approach which took 10 minutes. (Slide 10.)

Space Complexity

Definition (Space complexity)

We say that an algorithm *uses space* $S(n)$ if $S(n)$ is the maximum amount of space that it requires on an input of size n. In this case, we may say that the *space complexity* of this algorithm is $S(n)$.

- Space complexity is important, although perhaps less important than time complexity.
- Reason: If the memory requirement exceeds the amount of RAM in your computer, the program will use the hard disk (swapping), which is much slower. Even an SSD is much slower. (100–1000 times slower.)
- Similarly, CPUs use cache memory. If the whole execution of the program fits in cache memory, it is much faster.

Improved Dynamic Programming Approach

- The naive algorithm (BINOMIAL) uses $\Theta(n)$ space. Reason: The depth of recursion is $\Theta(n)$, so $\Theta(n)$ recursive calls must be recorded in the stack.
- The dynamic programming approach (DPBINOMIAL) uses $\Theta(n^2)$ space. Reason: The array $C[n][n]$ takes $\Theta(n^2)$ space.
- It is often the case with dynamic programming: We trade space for running time.
- Can we improve the space requirement of the dynamic programming approach?
- Yes. Idea: Only remember one row of Pascal's triangle. (See next slide.)

Improved Dynamic Programming Approach

Pseudocode

```
1: procedure ImprovedDPBinomial($n, k$)
2:     $B[0 \dots n] \leftarrow$ new array            ▷ Records the current row
3:     $B[0] \leftarrow 1$                              ▷ $\binom{i}{0} = 1$ for all $i$
4:     for $i \leftarrow 1, n$ do
5:         $B[i] \leftarrow 1$                          ▷ $\binom{i}{i} = 1$
6:         for $j \leftarrow i - 1$ downto 1 do
7:             $B[j] \leftarrow B[j-1] + B[j]$          ▷ $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$
8:     return $B[k]$
```

- This algorithm uses $\Theta(n)$ space and runs in $\Theta(n^2)$ time.
- So it runs as fast as DPBinomial and has the same space requirement as the naive algorithm. (At least up to a constant factor.)

Concluding Remarks

- Dynamic programming is a very important algorithm design technique. It often allows us to bring down the running time from exponential to polynomial.
- It often works for discrete optimization problems. We will see several examples later this semester.
- The memory requirement increases, but sometimes it can be kept under control by not remembering all subproblems.
- In particular, it often happens when the table is bidimensional that it suffices to remember one row or column, as we did in the last algorithm.
- There are better ways to compute binomial coefficients. The purpose of this lecture was to give a simple introduction to dynamic programming.