Reference textbook: R. Napolitan and K. Naimipour, Foundations of Algorithms (4^{th} ed.), Jones and Bartlett, 2011

ACM Tutorial: Dynamic Programming

Dr. Ukrit Watchareeruetai

Department of Engineering and Technology,
International College, KMITL

Contents

- Introduction to dynamic programming
  - The Fibonacci sequence
- The binomial coefficient

Introduction to dynamic programming

The Fibonacci sequence

- The Fibonacci sequence is defined recursively as follows:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{for } n \geq 2$$

- The first few terms of the sequence are:

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$
$$f_3 = f_2 + f_1 = 1 + 1 = 2$$
$$f_4 = f_3 + f_2 = 2 + 1 = 3$$
$$f_5 = f_4 + f_3 = 3 + 2 = 5, \text{ etc.}$$

Top-down approach

Algorithm 1.6: nth Fibonacci Term (Recursive)

Problem: Determine the *n*th term in the Fibonacci sequence.

Inputs: a nonnegative integer *n*.

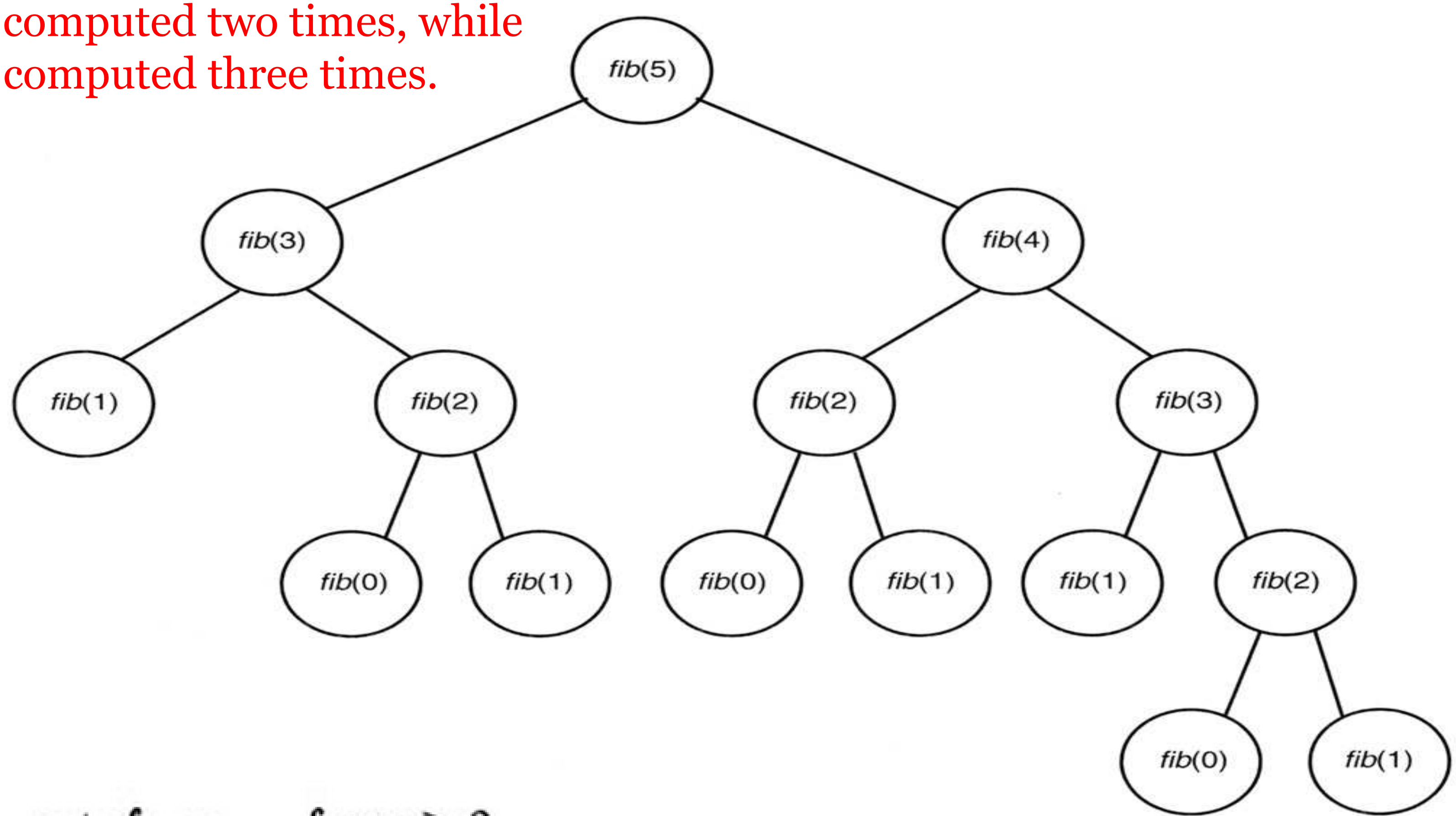
Outputs: *fib*, the *n*th term of the Fibonacci sequence.

```
int fib (int n)
{
        if (n <=  1)
           return n;
           else
           return fib (n - 1) + fib (n-2);
}
```

Divide into two smaller instances that are almost as large as the original instance. Not good!

The Fibonacci sequence

- fib(3) is computed two times, while fib(2) is computed three times.



$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{for } n \geq 2$$

Dynamic programming

- Dynamic programming (DP) technique is similar to divide-and-conquer in that an instance of a problem is divided into smaller instances.
- However, we solve small instances first, store the results, and later, whenever we need a result, look it up instead of re-computing it.
- The term "dynamic programming" comes from control theory, and in this sense "programming" means the use of an array (table) in which a solution is constructed.

Dynamic programming

- The steps in the development of a DP algorithm are as follows:
  - *Establish* a recursive property that gives the solution to an instance of the problem.
  - *Solve* an instance of the problem in a *bottom-up* fashion by solving smaller instances first.

Bottom-up approach

Algorithm 1.7: *n*th Fibonacci Term (Iterative)

Problem: Determine the *n*th term in the Fibonacci sequence.

Inputs: a nonnegative integer *n*.

Outputs : *fib2*, the *n*th term in the Fibonacci sequence.

```
int fib 2 (int n)
{
   index i;
   int f[0 .. n];
   f[ 0 ] = 0;
   if (n > 0)
         f[ 1 ] = 1;
         for (i = 2; i<= n; i++)
            f[ i ] = f[i - 1] + f [i -2 ];
      }
   return f[ n ];
}
```

Use an array to store the solution to smaller instances

Combine them later

The binomial coefficient

The binomial coefficient

- The binomial coefficient is given by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad for\ 0 \le k \le n$$

- For values of n and k that are not small, we cannot compute the binomial coefficient directly from this definition because $n!$ is very large even for moderate values of n.

The binomial coefficient

- We establish that

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \; or \; k = n \end{cases}$$

- So we can eliminate the need to compute *n!* or *k!* by using this recursive property.
- This suggests the following *divide-and-conquer* algorithm.

Algorithm 3.1: Binomial Coefficient Using Divide-and-Conquer

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k, where $k \leq n$.

Outputs: *bin*, the binomial coefficient $\binom{n}{k}$.

```
int bin (int n, int k)
{
  if ( k = = 0 || n = = k)
       return 1;
  else
     return bin (n-1, k - 1)+bin (n - 1, k);
}
```

Binomial coefficient using divide-and-conquer

- Like Algorithm 1.6 (Fibonacci, recursive), Algorithm 3.1 is *very inefficient*.
- It computes $2\binom{n}{k}-1$ terms to determine $\binom{n}{k}$.
- The problem is that the same instance are solved in each recursive.
  - E.g., *bin(n - 1, k - 1)* and *bin(n - 1, k)* both need the result of *bin(n - 2, k - 1)*.

Binomial coefficient using DP

- The steps for constructing a DP algorithm for this problem are as follows:
  1. Establish a recursive property. Written in terms of *B*, it is

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \; or \; j = i \end{cases}$$

  2. Solve an instance of the problem in a bottom-up fashion by computing the rows in *B* in sequence starting with the first row.

Example 1

- Compute $B[4][2] = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$
- Compute row 0:
  - $B[0][0] = 1$
- Compute row 1:
  - $B[1][0] = 1$
  - $B[1][1] = 1$
- Compute row 2:
  - $B[2][0] = 1$
  - $B[2][1] = B[1][0] + B[1][1] = 1+1 = 2$
  - $B[2][2] = 1$

Example 1

- Compute row 3:
  - $B[3][0] = 1$
  - $B[3][1] = B[2][0] + B[2][1] = 1+2 = 3$
  - $B[3][2] = B[2][1] + B[2][2] = 2+1 = 3$
- Compute row 4:
  - $B[4][0] = 1$
  - $B[4][1] = B[3][0] + B[3][1] = 1+3 = 4$
  - $B[4][2] = B[3][1] + B[3][2] = 3+3 = 6$

	0	1	2	3	4	j	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
i							
n							

$$\begin{tikzcd}
B[i-1][j-1] \arrow[dr] & B[i-1][j] \arrow[d] \\
 & B[i][j]
\end{tikzcd}$$

Algorithm 3.2: Binomial Coefficient Using Dynamic Programming

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k, where $k \leq n$.

Outputs: *bin* 2, the binomial coefficient $\binom{n}{k}$.

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n] [0..k];

  for (i = 0; i <= n; i++)
       for (j = 0; j <= minimum (i, k); j++)
          if (j = = 0 || j = = i)
               B[i][      j] = 1'
          else
               B[i[j] = B[i - 1][j - 1] + B[i - 1] [j];
 return B[n[k];
}
```

Binomial coefficient using DP

- By using DP instead of divide-and-conquer, we have developed a much more efficient algorithm.
- In both techniques, we find a *recursive property* that *divides an instance into smaller instances*.
- However, DP uses the recursive property to *iteratively solve the instance in sequence, starting from the smallest instance*, instead of blindly using recursion (as in divide-and-conquer).
- DP is a good technique to try when divide-and-conquer leads to an inefficient algorithm.