Algorithms and Data Structures

Module 3. Dynamic programming

Lecture 11

Dynamic programming intro.

Fibonacci numbers

Fibonacci numbers sequence:

$$F_0 = 0, F_1 = 1.$$
$$F_n = F_{n-1} + F_{n-2}.$$

Problem: given integer *n*, calculate F_n.

Fibonacci numbers: recursive algorithm

Naïve recursive algorithm:

FR(n):

```
if n=0 then return 0;
if n=1 then return 1;
  else return FR(n-1)+FR(n-2).
```

Fibonacci numbers: recursive algorithm

Let us evaluate the time complexity of the recursive function.

Let $T(n)$ be the time complexity function.

The following conditions hold:

$$T(0) = T(1) = const.$$

$$T(n) = T(n-1) + T(n-2) + O(1).$$

Thus, $T(n) \sim O(F_n)$.

Fibonacci numbers: recursive algorithm

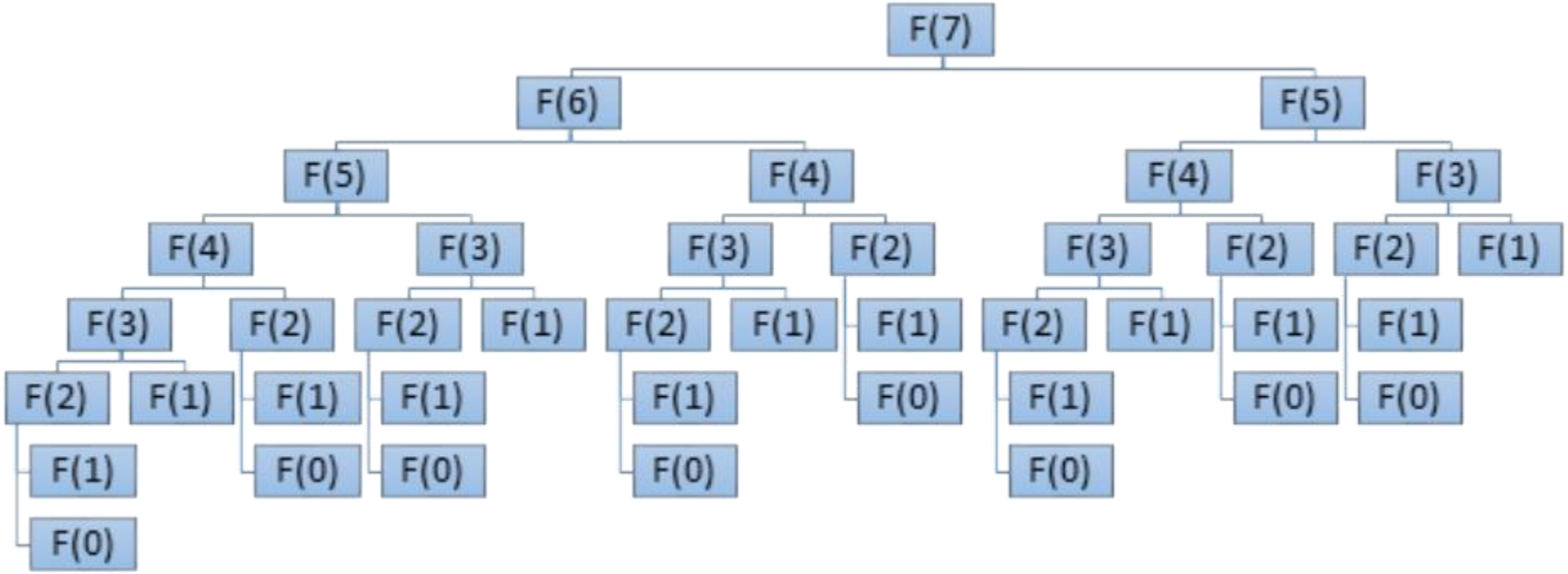
Time complexity of the recursive algorithm: $T(n) \sim O(F_n)$.

But recall that there is a closed form for the *n*-th Fibonacci number: $F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$, where $\varphi = \frac{1+\sqrt{5}}{2}, \psi = \frac{1-\sqrt{5}}{2}$. This implies that $F_n = O(\varphi^n)$ and thus, the recursive algorithm has exponential time complexity…

Is there a faster algorithm?

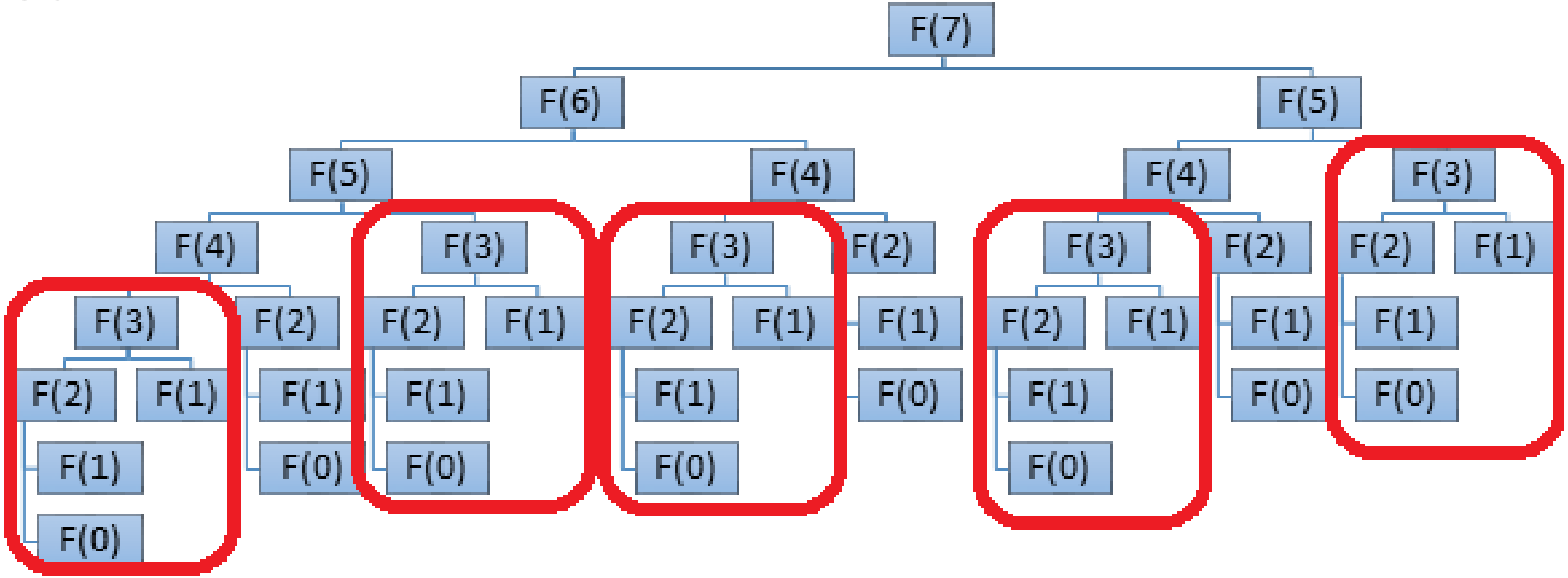
Fibonacci numbers: recursive algorithm

Let's analyze the run of the recursive algorithm for F_7.



Fibonacci numbers: recursive algorithm

We see that many values have been calculated many times...



Fibonacci numbers: memoization

Since the recursive algorithm spends too much time for recalculating values several times, the idea is to store calculated values and return stored value instead of recalculating it.

This technique is known as *memoization* (or caching).

Fibonacci numbers: memoization

Memoization-based version of the recursive algorithm:

```
Fibonacci_Memo(n):
        Create M[2..n] and fill it with NULL values.
        Return FM(M,n).
Function FM(M,n):
    if n=0 then return 0;
    if n=1 then return 1;
        else
                if M[n] == NULL then M[n] = FM(n-1)+FM(n-2).
                return M[n].
```

Time complexity: $O(n)$. So, we get an exponential speed-up!!!

Fibonacci numbers

Recursion makes overhead memory and time expenses…

Do we really need recursion?

Fibonacci numbers: dynamic programming

We can see that F_n can be calculated only when $F_2, \ldots, F_{n-1}$ are calculated. What if we fill the memoization table directly, in this order?

Fibonacci DP(n):

```
// Creating table
Create M[0..n]
// Initialization
M[0] = 0;
M[1] = 1;
// Filling the table
for i=2 to n: M[i] = M[i-1]+M[i-2];
// Return the resulting value
return M[n].
```

Fibonacci numbers: dynamic programming

This way, we get a non-recursive algorithm with $O(n)$ time complexity and $O(n)$ space complexity.

Let's compare these two algorithms:

```
Fibonacci_Memo(n):
    Create M[2..n]
    Fill it with NULL values.
    Return FM(M,n).

Function FM(M,n):
  if n=0 then return 0;
  if n=1 then return 1;
  else
    if M[n] == NULL then
       M[n] = FM(M,n-1)+FM(M,n-2).

    return M[n].
```

```
Fibonacci_DP(n):

    // Creating table
    Create M[0..n]

    // Initialization
    M[0] = 0;
    M[1] = 1;

    // Filling the table
    for i=2 to n: M[i] = M[i-1]+M[i-2];

    // Return the resulting value
    return M[n].
```

Fibonacci numbers: dynamic programming

We can even decrease the space complexity to $O(1)$:

Fibonacci_DP1(n):

```
A = 0;
B = 1;
for i=2 to n:
      C = A + B;
      A = B;
      B = C;
return C.
```

Fibonacci numbers: dynamic programming

A general scheme of a simple version of a dynamic programming algorithm.

1. Decompose the problem into a number of smaller subproblems.
2. Create a table for keeping solutions of all subproblems.
3. For the smallest subproblems calculate the solutions directly or using a special algorithm.
4. Fill the table in the bottom-up manner, from the small subproblems to the initial problem.
5. Return one of the entries of the table as the result.

Fibonacci numbers: dynamic programming

Do you need a dynamic algorithm for your problem?

1. Construct a recursive algorithm for the problem.
2. Evaluate the complexity of the recursive algorithm.
3. If the recursive algorithm is fast enough, that is OK ☺
4. If the recursive algorithm has exponential time complexity, analyze its run.
5. If the recursive algorithm recalculates solutions to the same subproblems for many times, apply memorization or transform it into a DP algorithm!

Fibonacci numbers: dynamic programming

A meta-algorithm for transforming a recursive algorithm into a dynamic programming (DP) algorithm.

1. Create a table for storing solutions of subproblems.

The dimensionality of the table is equal to the number of parameters of the recursive function.

The size of the table in the *i*-th dimension is equal to the number of possible values of the *i*-th parameter.

```
Create a table M[0..n]
```

Fibonacci numbers: dynamic programming

2. Turn the base cases of the recursive algorithm into initial filling of the corresponding entries of the table.

```
M[0] = 0;
M[1] = 1;
```

Fibonacci numbers: dynamic programming

3. Replace recursive calls with getting values from the table.
4. Replace returning solution of a subproblem with storing it to the table.

Put (3) and (4) within a loop.

```
for i=2 to n: M[i] = M[i-1]+M[i-2];
```

Fibonacci numbers: dynamic programming

5. Return the value of a proper entry of the table.

In many cases the proper entry is the entry with the biggest indices.

But there are different cases.

```
return M[n].
```

Fibonacci numbers: dynamic programming

6. Analyze the space complexity. In many cases we do not need all entries till the finish, so we can reduce the amount of memory needed for calculations.

But there are cases for which we need the whole table to build the resulting solution.

```
A = 0;
B = 1;
…
```