DATA STRUCTURES AND ALGORITHMS

ALGORITHM DESIGN TECHNIQUES

issntt@nus.edu.sg

Problem

A child is **going up** a **staircase** with ***n* steps**, and **can hop** either ***1*** step, ***2*** steps, or ***3*** **steps** at a time. Implement a program to count **how many possible ways** the child can go up the stairs.

Example

Input: 3

Output: 4 (111, 12, 21, 3)

Image by Pexels from Pixabay

What have we learnt so far?

ADT and Data Structures

- Linear: Array List, Linked List, Stack, Queue
- Non-Linear: Tree, BST, Trie, Heap

Algorithms

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort

Programming Techniques

- Recursions

Complexity Analysis

- Big-Oh

Are there any other **tools** that help us **solve** even **more problems**?

Algorithm Design Techniques

The following techniques can **solve many problems**

- Greedy
- Brute Force
- Dynamic Programming
- Divide and Conquer

Image by mohamed Hassan from Pixabay

Proven methods or **processes** for designing and constructing algorithms

Outline

- **Greedy Algorithms**
  - **Problem - Money Change**
  - Greedy Techniques
  - Problem - Classroom Scheduling
- Time Efficiency of Recursive Methods
- Dynamic Programming

Money Change Problem

- In Singapore, assume the available notes/coins are \$1, \$2, \$5, \$10, \$50
- **Given N (integer) dollars** in Singapore
- What is the **minimum number of notes/coins** to make N dollars of **change**?

Image credit to changiairport.com

Question

How do want to make change for **S$7**?

Which option has **less number** of notes / coins?

Question

How about **S$13**?

Question

How about **$27**?

From these **observations**, **what should we choose** to make minimum number of notes/coins?

Idea

For example, **S$63**

Next largest possible	Left
	S$13
	S$3
	S$1
	S$0

Implementation

Keep changing denominations **from high to low** value, **until** there is **no more** amount to change

```
static int MinChange(int amount) {
   int[] DENOS = { 1, 2, 5, 10, 20, 50, 100 };

   int res = 0;
   // Traverse through all denomination
   for (int i = DENOS.Length - 1; i >= 0; i--) {
      // Making change
      while (amount >= DENOS[i]) {
         amount -= DENOS[i];
         res++;
      }
   }

   return res;
}
```

Worst case ***O(n/100) ~ O(n)***

To make it faster, n/100 to get all 100-note first **~ O(1)**

Outline

- **Greedy Algorithms**
  - Problem - Money Change
  - **Greedy Techniques**
  - Problem - Classroom Scheduling
- Time Efficiency of Recursive Methods
- Dynamic Programming

Greedy Techniques

- Because the previous uses a **greedy technique**, it's called greedy algorithm
- A greedy algorithm **works in phrase**. At each phrase
  - **Take the best** we can **right now**, **without regard** for **future consequences**
  - So, hope that choosing **local optimum** at each phrase will **end up global optimum**

Image by kirillslov from Pixabay

Does it really work?

Is there any **other solution** that is **more optimum**?

For **Singapore money**, it's proven that the greedy technique **always** gives **optimum solutions**

But for **some other problems**, greedy techniques do **not always** give **optimum solutions**

A Failure Case

In India, assume available notes/coins are **1, 5, 10, 20, 25, 50** Rupees. What is the **optimum** change for **40** Rupees?

Greedy Algorithm

1 x 25
1 x 15
1 x 5

3 notes/coins

Optimum solution

2 x 20

2 notes/coins

So, is Greedy Technique useless?

Not At All!

Outline

- **Greedy Algorithms**
  - Problem - Money Change
  - Greedy Techniques
  - **Problem - Classroom Scheduling (Self Study)**
- Time Efficiency of Recursive Methods
- Dynamic Programming

Classroom Scheduling Problem

Self study

Suppose we have **one classroom** and want to hold **as many classes** there **as possible**

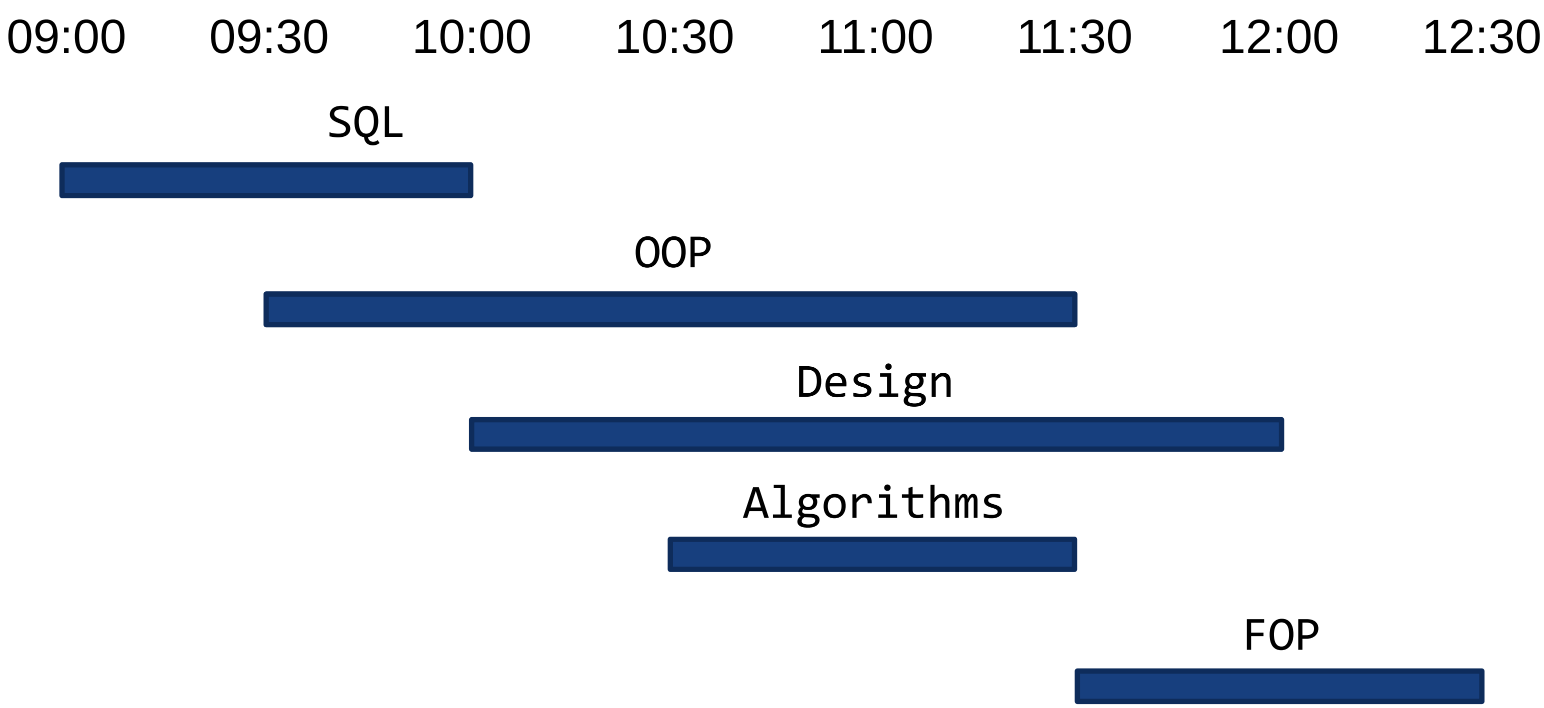
Class	Start	End
SQL	9:00	10:00
OOPCS	9:30	11:30
Design	10:00	12:00
Data Structures	10:30	11:30
FOPCS	11:30	12:30

Can we pick all of the classes?

The Scheduling Problem

Self study

We can’t, because some of them **overlap**



Which classes should we pick?

A greedy idea

Self study

Pick the **next class** that **starts** the **soonest**

Class	Start	End	Picked?
SQL	9:00	10:00	Yes
OOP	9:30	11:30	No
Design	10:00	12:00	Yes
Algorithm	10:30	11:30	No
FOP	11:30	12:30	No

Is it the optimal solution?

Why or why not?

A class may take **too long**

Sometimes, we need to try a different options for the correct greedy criterion

A greedy idea

Self study

Pick the **next class** that **ends** the **soonest**

Class	Start	End	Picked?
SQL	9:00	10:00	Yes
OOP	9:30	11:30	No
Design	10:00	12:00	No
Algorithm	10:30	11:30	Yes
FOP	11:30	12:30	Yes

Is there **any case** that this tactic **fail**?

No, it is proven that for this problem, this local optimum leads to global optimum

Outline

- Greedy Algorithms
- **Time efficiency of Recursive Methods**
  - **A review of Recursive Methods**
  - Problem – Calculating Sum
  - Time efficiency of Recursive Methods
  - Problem - Calculating Fibonacci Number
- Dynamic Programming

A review of Recursive Methods

A recursion is a technique **simplifying** a complicated **problem** by **breaking** it **down** into simpler sub-problems

```
static int Pow(int x, int exp)
{
    if (exp == 0)
    {
        return 1;
    }
    else
    {
        return x * Pow(x, exp - 1);
    }
}
```

A review of Recursive Methods

A recursive method performs a task by **calling itself** to **perform** some **subtasks** (recursive case)

```
static int Pow(int n, int exp)
{
    if (exp == 0)
    {
        return 1;
    }
    else
    {
        return n * Pow(n, exp - 1);
    }
}
```

A review of Recursive Methods

At some point, the method encounters a **subtask** that can **perform without calling itself** (base case)

```
static int Pow(int n, int exp)
{
   if (exp == 0)
   {
      return 1;
   }
   else
   {
      return n * Pow(n, exp - 1);
   }
}
```

Question

Can we call **more than one sub-tasks**?

Can we have **more than one base-cases**?

```
static int Pow(int x, int exp)
{
    if (exp == 0)
    {
        return 1;
    }
    else
    {
        return x * Pow(x, exp - 1);
    }
}
```

A Learning Tip

Computers can always solve sub-problems automatically

Given that, can we

1. **Solve** the **whole problem**? And
2. **Find base cases** and solve them?

Image by Clker-Free-Vector-Images from Pixabay

Outline

- Greedy Algorithms
- **Time efficiency of Recursive Methods**
  - A review of Recursive Methods
  - **Problem – Calculating Sum**
  - Time efficiency of Recursive Methods
  - Problem - Calculating Fibonacci Number
- Dynamic Programming

Problem – Calculating Sum

Given an **array** of **integers**, write a method to **calculate** and return the **sum** of all elements

Sample Input: 3, 5, 1, 2, 2, 4, 1, 8

Sample Output: 26

Thinking 1

Providing that the sub-problem "calculate and return the sum of all elements **excluding the last" has been solved**

1. Can we calculate and return the sum of all elements?
2. What is/are the base case(s)?

3, 5, 1, 2, 2, 4, 1, 8

Implementing Linear Sum Left

Keep a variable ***toWhere***, indicating the **last index to include**

```
static int LinearSumLeft(int[] arr, int toWhere)
{
   if (toWhere == 0)
      return arr[0];

   return LinearSumLeft(arr, toWhere - 1)
                                      + arr[toWhere];
}

static int LinearSum_A(int[] arr)
{
   return LinearSumLeft(arr, arr.Length - 1);
}
```

Thinking 2

Self study

Providing that the sub-problem "calculate and return the sum of all elements **excluding the first**" has been solved

1. Can we calculate and return the sum of all elements?
2. What is/are the base case(s)?

3,	**5, 1, 2, 2, 4, 1, 8**

Implementing Linear Sum Right

Self study

Keep a variable ***fromWhere***, indicating the **first index to include**

```
static int LinearSumRight(int[] arr, int fromWhere)
{
   if (fromWhere == arr.Length - 1)
      return arr[arr.Length - 1];

   return arr[fromWhere] +
            LinearSumRight(arr, fromWhere + 1);
}

static int LinearSumRight(int[] arr)
{
   return LinearSumRight(arr, 0);
}
```

Thinking 3

Self study

Providing that the sum for the following **2 sub-problems** have been solved:

- “All elements in the **first half**”, and
- “All elements in the **second half**”

1. Can we calculate and return the sum of all elements?
2. What is/are the base case(s)?

3, 5, 1, 2	**2, 4, 1, 8**

Implementing Binary Sum

Self study

Keep a variable ***fromWhere*** and ***toWhere***, indicating the **first and last index to include**

```
static int BinarySum(int[] arr,
                    int fromWhere, int toWhere) {
   if (fromWhere > toWhere) return 0;

   if (fromWhere == toWhere) return arr[fromWhere];

   int middleIndex = (fromWhere + toWhere) / 2;
   return BinarySum(arr, fromWhere, middleIndex) +
            BinarySum(arr, middleIndex + 1, toWhere);
}

static int BinarySum(int[] arr)
{
   return BinarySum(arr, 0, arr.Length - 1);
}
```

Next

How can we **analyze** a **recursive method** to see if it runs **fast** enough?

Image by Thomas Wolter from Pixabay

Btw, how can we analyze a **non-recursive method**?

Outline

- Greedy Algorithms
- **Time efficiency of Recursive Methods**
  - A review of Recursive Methods
  - Problem – Calculating Sum
  - **Time efficiency of Recursive Methods**
  - Time efficiency of Recursive Fibonacci Numbers
- Dynamic Programming

Time efficiency of Linear Sum

When array length is ***1***, we need ***1*** basic operation

```
LinearSumLeft(arr, 5)
        LinearSumLeft(arr, 4)
              LinearSumLeft(arr, 3)
                     LinearSumLeft(arr, 2)
                            LinearSumLeft(arr, 1)
                                  LinearSumLeft(arr, 0)
```

```
static int LinearSumLeft(...) {
   if (toWhere == 0)
      return arr[0];
   ...
}
```

Time efficiency of Linear Sum

When array length is **2**, we need ***1 + 1*** basic operations

```
LinearSumLeft(arr, 5)
        LinearSumLeft(arr, 4)
              LinearSumLeft(arr, 3)
                     LinearSumLeft(arr, 2)
                          LinearSumLeft(arr, 1)
                                LinearSumLeft(arr, 0)
```

```
static int LinearSumLeft(...) {
  ...
  return
    LinearSumLeft(arr, toWhere - 1)
                     + arr[toWhere];
}
```

Time efficiency of Linear Sum

When array length is ***3***, we need ***`1 + 1 + 1`*** basic operations

```
LinearSumLeft(arr, 5)
       LinearSumLeft(arr, 4)
             LinearSumLeft(arr, 3)
                 LinearSumLeft(arr, 2)
                       LinearSumLeft(arr, 1)
                             LinearSumLeft(arr, 0)
```

Time efficiency of Linear Sum

When array length is ***n***, we need ***`1 + 1 + 1... + 1 (n times)`*** basic operations

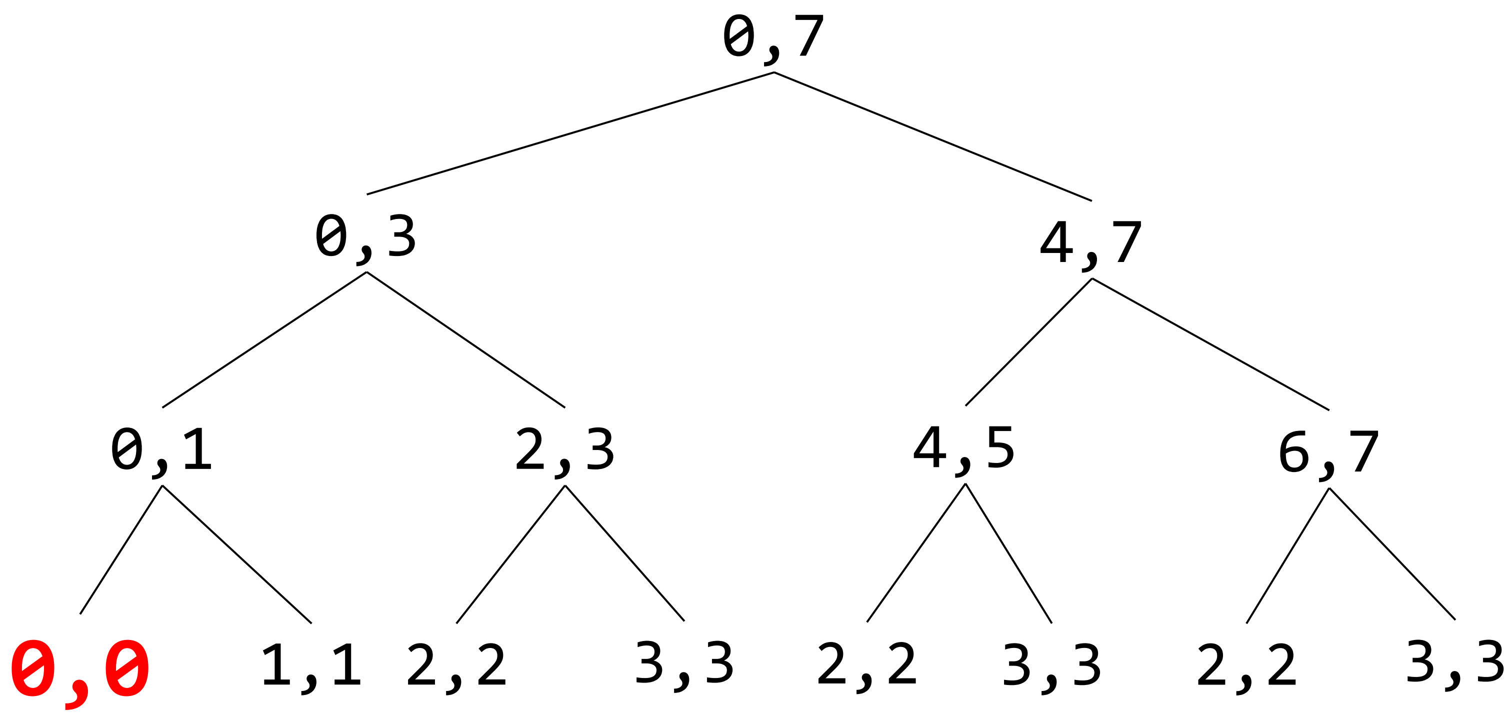
```
LinearSumLeft(arr, 5)
      LinearSumLeft(arr, 4)
           LinearSumLeft(arr, 3)
                 LinearSumLeft(arr, 2)
                      LinearSumLeft(arr, 1)
                           LinearSumLeft(arr, 0)
```

Time complexity: ***`O(n)`***

Time efficiency of Binary Sum

Self study

When array length is ***1***, we need ***1*** basic operation



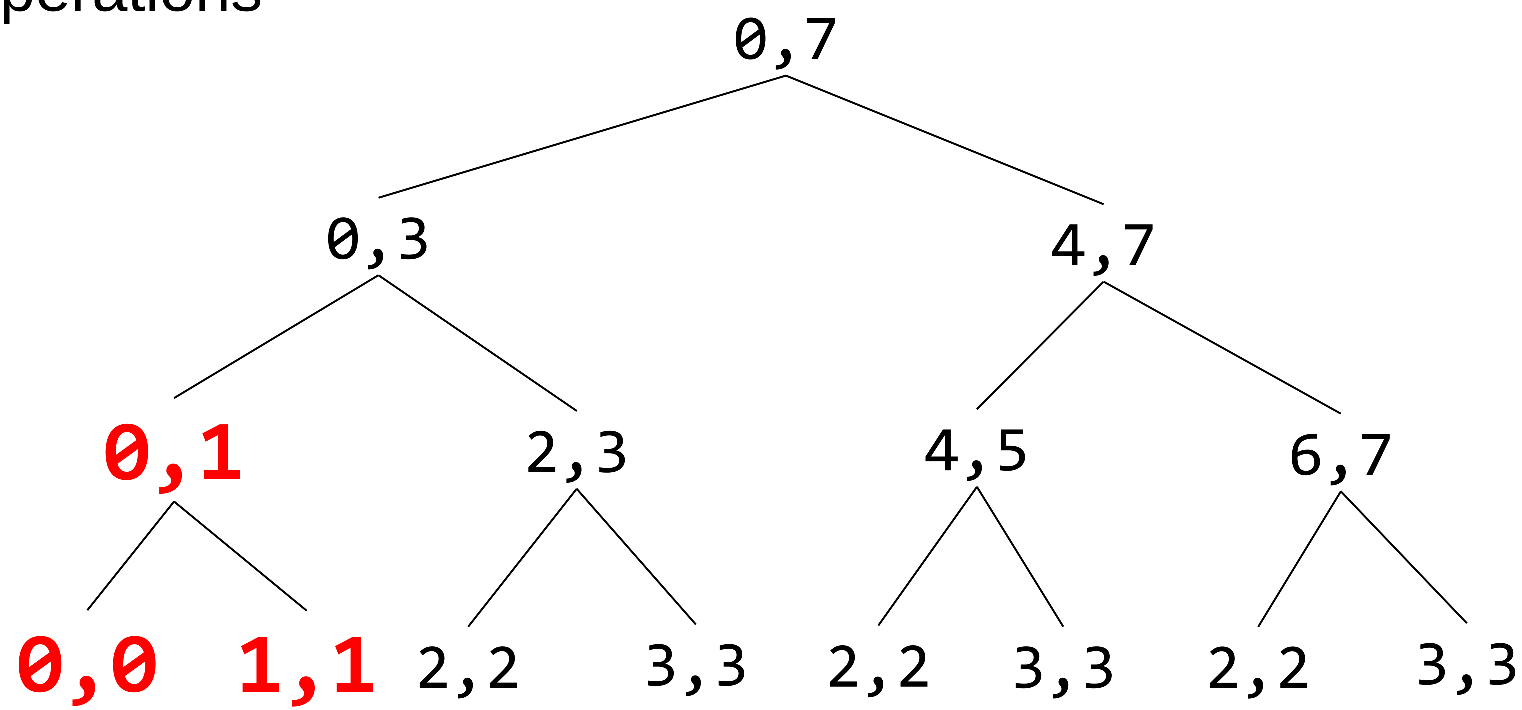

```
if (fromWhere == toWhere)
   return arr[fromWhere];
```

Note: **0,0** is short for **BinarySum(arr, 0, 0)**

Time efficiency of Binary Sum

Self study

When array length is **2**, we need ***`2x1 + 1x2`*** basic operations



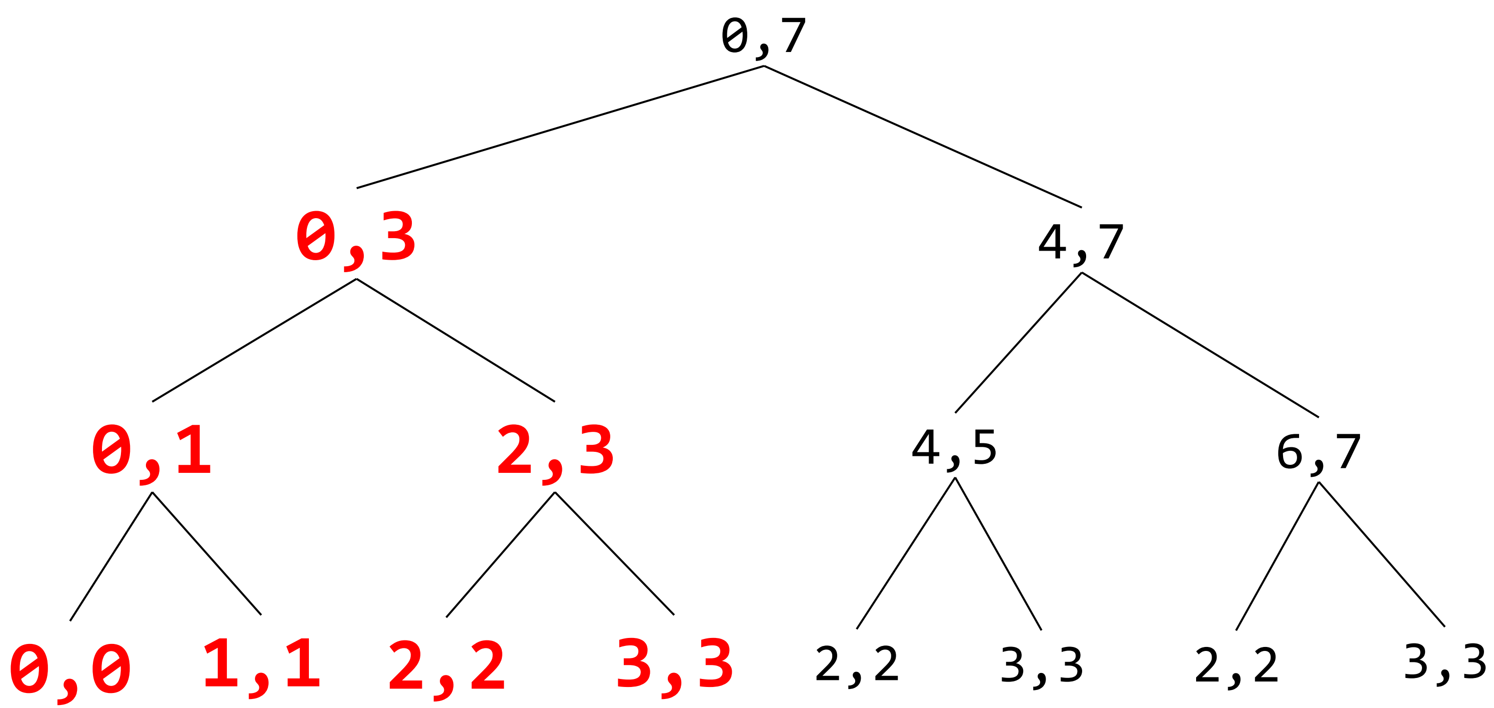

```
int middleIndex = (fromWhere + toWhere) / 2;
return BinarySum(arr, fromWhere, middleIndex) +
         BinarySum(arr, middleIndex + 1, toWhere);
```

2 leaf nodes x 1 operation + 1 non-leaf node x 2 operations

Time efficiency of Binary Sum

Self study

When array length is ***4***, we need ***4x1 + (2+1)x2*** basic operations

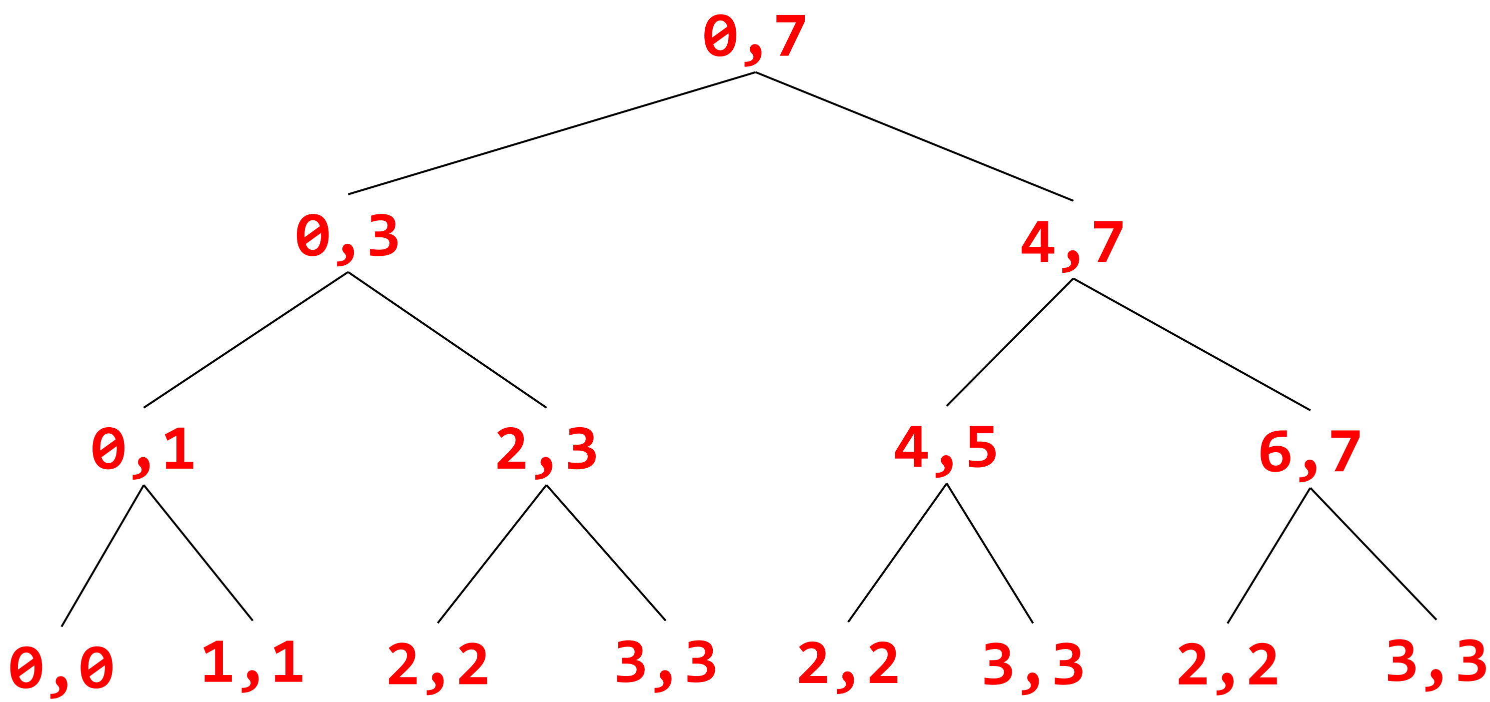


Counting is based on every level in the tree

Time efficiency of Binary Sum

Self study

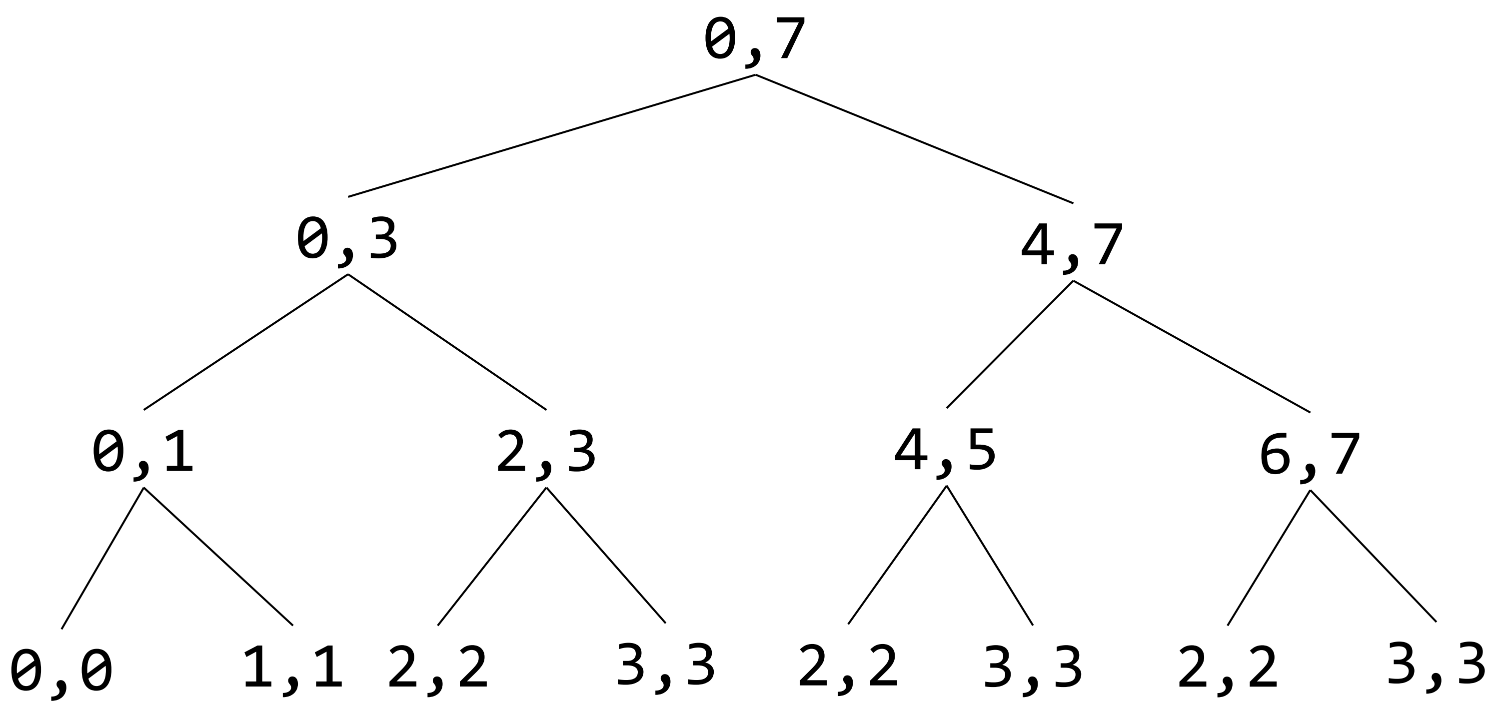
When array length is **8**, we need **`8x1 + (4+2+1)x2`** basic operations



Time efficiency of Binary Sum

Self study

When array length is ***n***, we need ***nx1 + (n-1)x2*** basic operations



Outline

- Greedy Algorithms
- **Time efficiency of Recursive Methods**
  - A review of Recursive Methods
  - Problem – Calculating Sum
  - Time efficiency of Recursive Methods
  - **Time efficiency of Recursive Fibonacci Numbers**
- Dynamic Programming

Recursive Fibonacci Numbers

Recursive methods can be used to calculate Fibonacci numbers

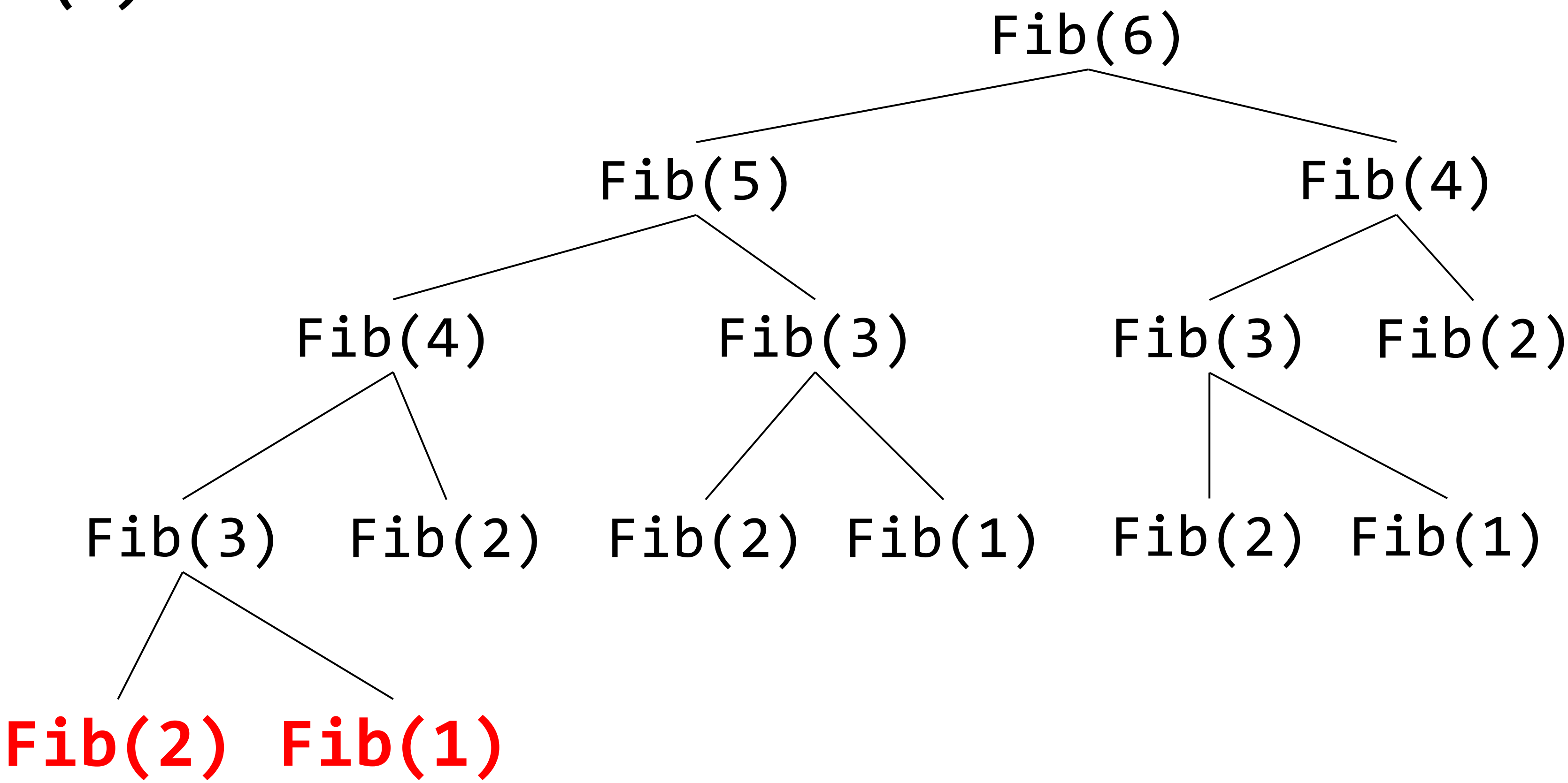
```
static int Fib(int n) // n > 0
{
   if (n == 1) return 1;
   if (n == 2) return 1;

   return Fib(n - 1) + Fib(n - 2);
}
```

What is the time efficiency of this implementation?

Time efficiency of Fibonacci

When n is ***1*** or ***2***, we need ***1*** basic operation, ***t(1) = t(2) = 1***

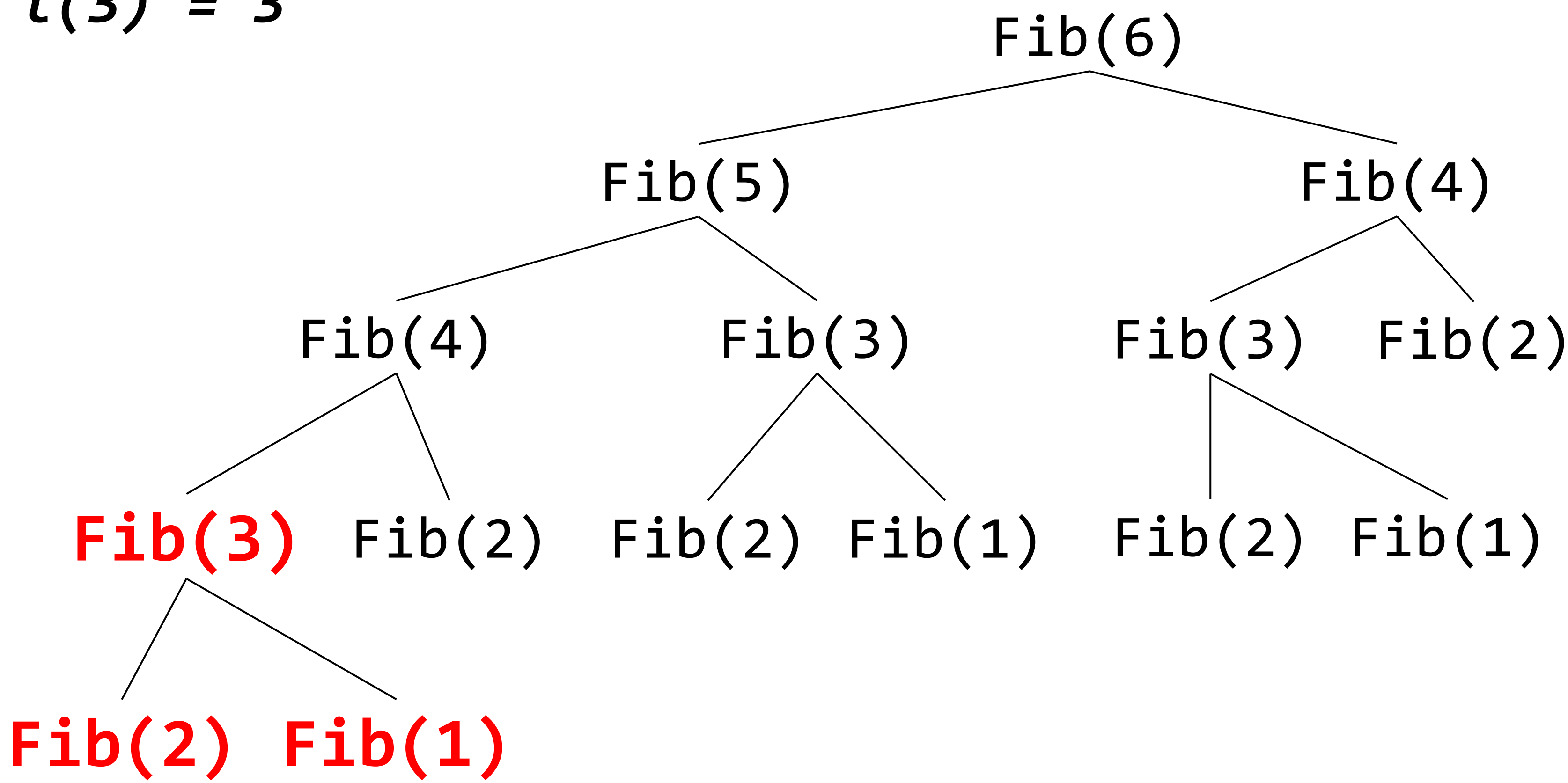



```
if (n == 1) return 1;
if (n == 2) return 1;
```

Time efficiency of Fibonacci

When n is ***3***, we need ***1 + 1 + 1*** basic operations,

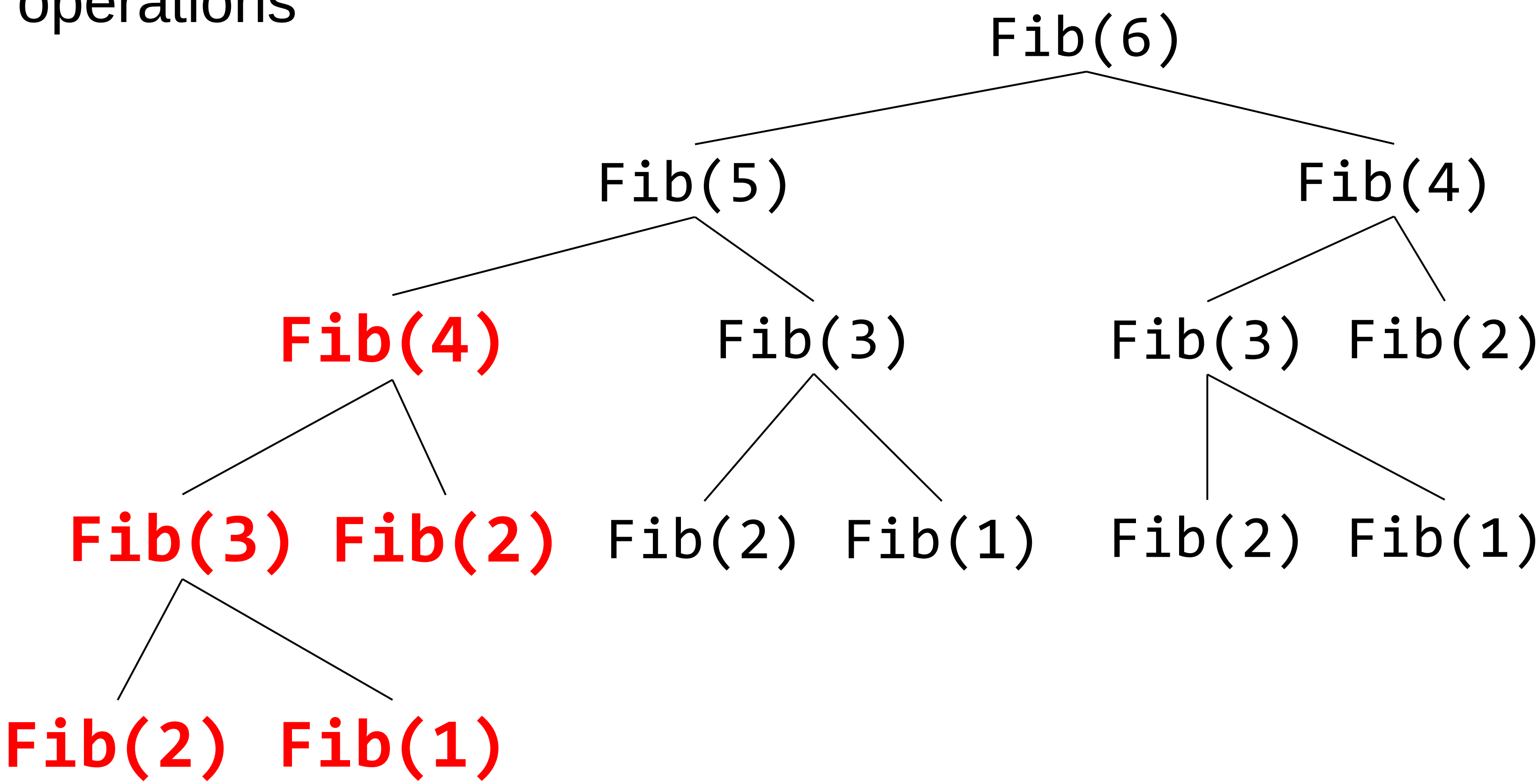
t(3) = 3




```
return Fib(n - 1) + Fib(n - 2);
```

Time efficiency of Fibonacci

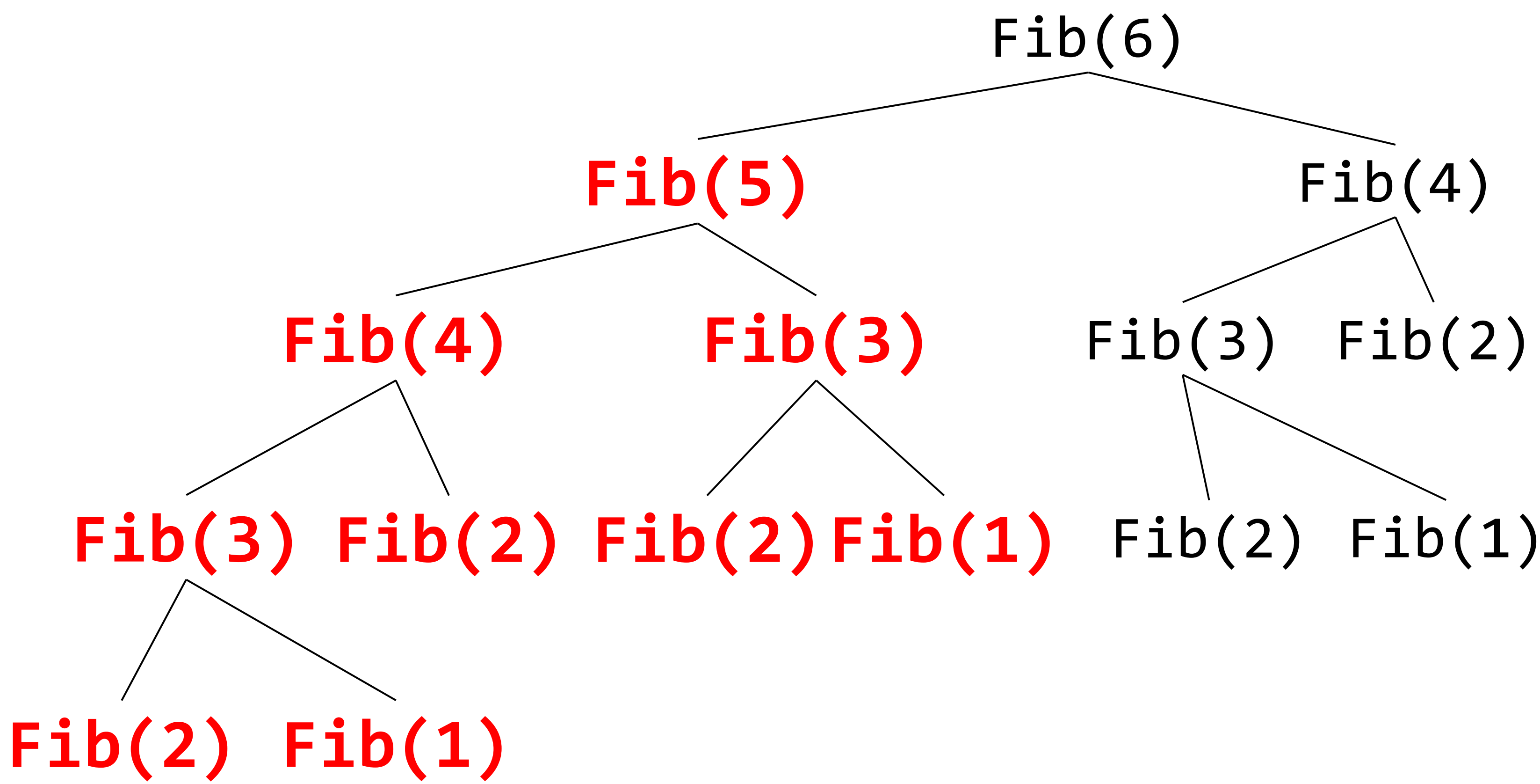
When n is ***4***, we need ***1 + (1 + 2) + 1*** basic operations



Observation: $\boldsymbol{t(4)} = 1 + (1 + 2) + 1 = 1 + F_3 + F_2 = \boldsymbol{1 + F_4 > F_4}$
where $\boldsymbol{F_4}$ is Fibonacci number 4

Time efficiency of Fibonacci

When n is ***5***, we need ***1 + (1 + (1 + 2) + 1) + (1 + 2)*** basic operations



Observation: $t(5) = 1 + (1 + (1 + 2) + 1) + (1 + 2) = 1 + F_5 > F_5$

Time efficiency of Fibonacci

- For ***n>=2***, ***t(n) = 1 +*** F_n ***>*** F_n basic operations
- It is proven that

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

- And conclude that time complexity for calculate F_n recursively **increases exponentially** as n increases
  $t(n) \sim O(1.618^n)$

You don't need to know all the details, which include several math. If interested, read Recurrence Relation

Time efficiency of Fibonacci

Self study

Roughly, **how long** does it take to **calculate F_{100}**?

Assume that our fast PC can do **100 millions** (100×10^6) basic **operations per second**

Image by Peter Fischer from Pixabay

No of basic operations:
$\sim 1.618^{100} = 790 \times 10^{18}$

No of seconds:
$\sim (790 \times 10^{18})/(100 \times 10^6) = 7.9 \times 10^{12}$

No of years:
$\sim (7.9 \times 10^{12})/(3.156 \times 10^7) = 2.5 \times 10^5$

Question

In **some cases**, when **recursions** are used, time complexity tends to be **very slow**

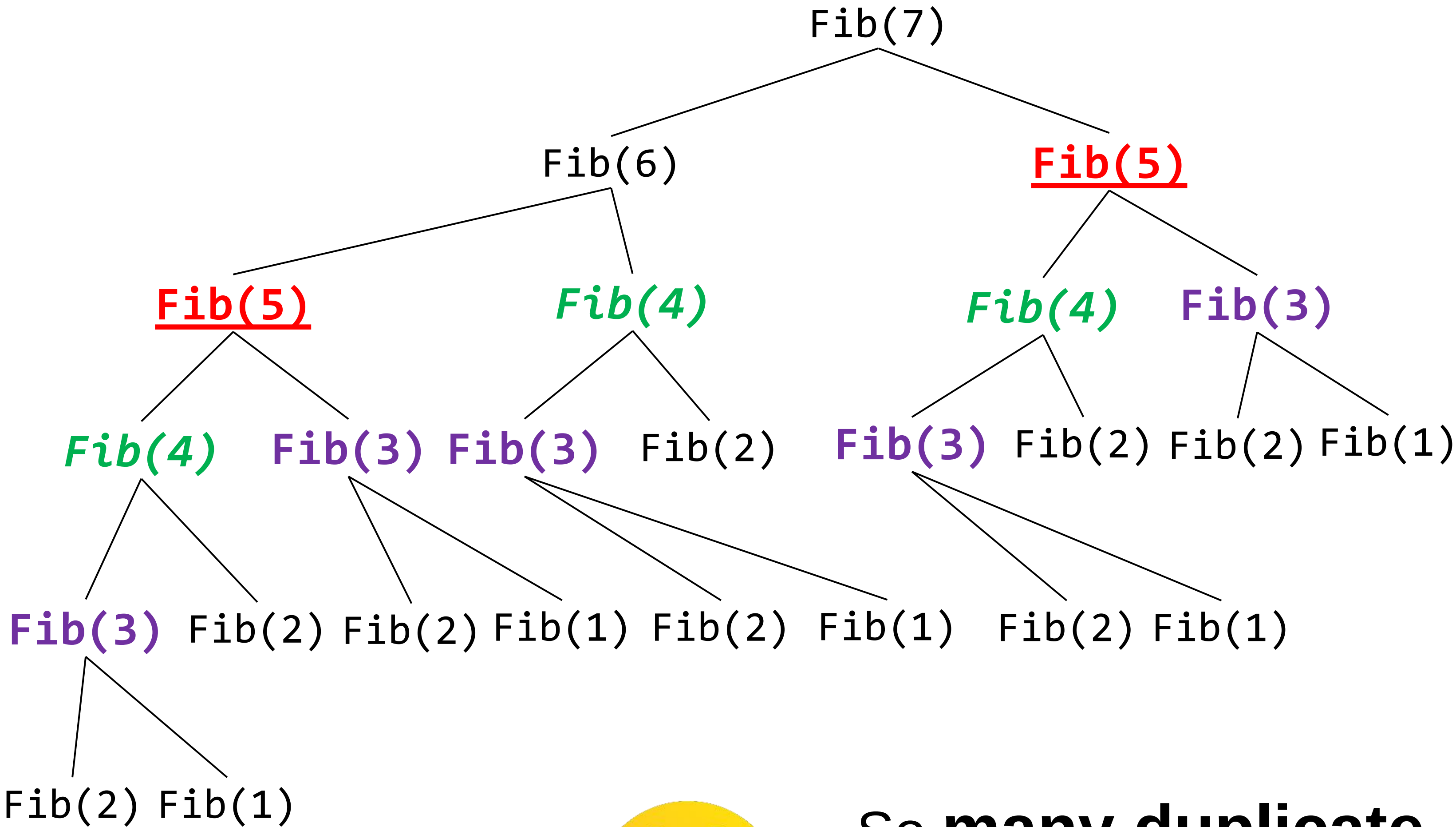
Can our solution still **keep** the **recursions** but **perform better**?


Image by jacqueline macou from Pixabay

Outline

- Greedy Algorithms
- Time efficiency of Recursive Methods
- **Dynamic Programming**
  - Problem - Recursive Fibonacci Numbers revisit
  - Problem - Different Ways of Sum
  - Problem - Money Change revisit

Why is it so BAD?



So **many duplicate** computations! Can we remove them?

Dynamic Programming

Dynamic Programming is mainly an **optimization** over a **plain recursion** by **caching** the **results** of subproblems

Dynamic Programming = Recursion + Memorization (caching)

When it takes us **so long** to **work** on a **solution**, do **remember** its **answer** for the subsequent efforts

Outline

- Greedy Algorithms
- Time efficiency of Recursive Methods
- **Dynamic Programming**
  - **Problem - Recursive Fibonacci Numbers revisit**
  - Problem - Different Ways of Sum
  - Problem - Money Change revisit

DP for Fibonacci

Store the results and try **retrieving before computing**

```
static int Fib(int n) // n > 0
{
   // 1. Try retrieving res from memo
   if (value Fib_n is available in memo)
      return Fib_n

   // 2. Compute res
   if (n == 1) res = 1;
   else if (n == 2) res = 1;
   else res = Fib (n - 1) + Fib(n - 2);

   // 3. Store the res to memo
   // 4. Return the res
   return res
}
```

What **data structure** is **appropriate** for ***memo***?

DP for Fibonacci

Following is an implementation using **Dictionaries** (**Hash Tables** inside) for **memorization**

```
static long Fib_DP1(long n, Dictionary<long, long> memo) {
   if (memo.ContainsKey(n))
      return memo[n];

   long res;
   if (n == 1)
      res = 1;
   else if (n == 2)
      res = 1;
   else
      res = Fib_DP1(n - 1, memo)
            + Fib_DP1(n - 2, memo);

   memo.Add(n, res);
   return res;
}
```

Why is ***memo*** **declared** in the **parameter list** and not in method body?

DP for Fibonacci

To use the method, **create** an **empty Dictionary** and **use it** as a **parameter**

```
static void Main()
{
  Console.WriteLine(Fib_DP1(40));
  Console.WriteLine(Fib_DP1(50));
}

public static long Fib_DP1(long n)
{
  Dictionary<long, long> memo =
               new Dictionary<long, long>();
  return Fib_DP1(n, memo);
}
```

```
102334155
12586269025
```

DP for Fibonacci

Alternatively, using **direct addressing** to implement **memorization**. Faster? *(in fact, not much!)*

```
static long Fib_DP2(long n, long[] memo)  {
   if (memo[n-1] > 0)
      return memo[n-1];

   long res;
   if (n == 1)
      res = 1;
   else if (n == 2)
      res = 1;
   else
      res = Fib_DP2(n - 1, memo)
            + Fib_DP2(n - 2, memo);

   memo[n-1] = res;
   return res;
}
```

Compared to Dictionary, which implementation has **better readability**?

DP for Fibonacci Numbers

To use the method, create an **empty array** whose **length equals** to the Fibonacci number to calculate

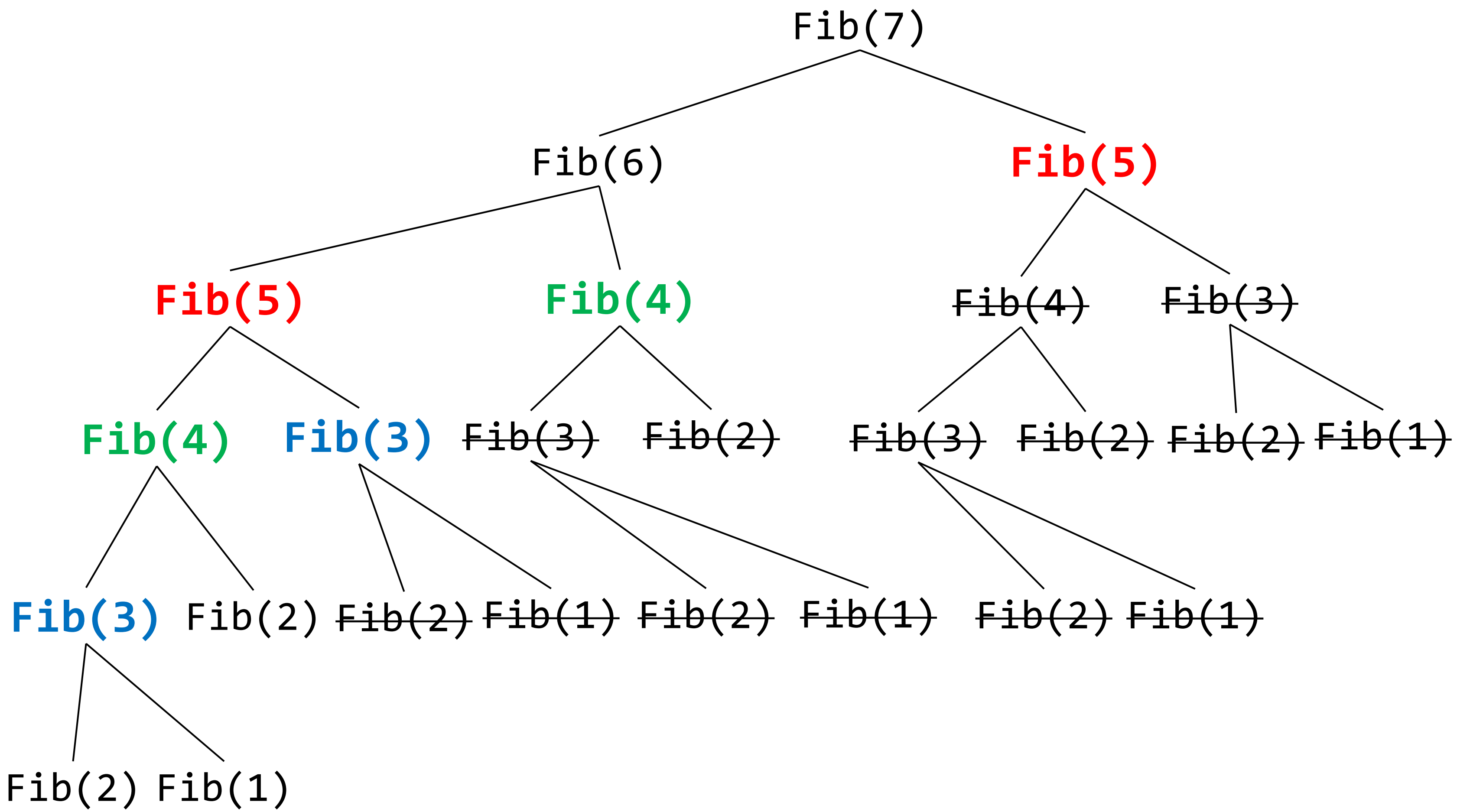
```
static void Main()
{
  Console.WriteLine(Fib_DP2(40));
  Console.WriteLine(Fib_DP2(50));
}

public static long Fib_DP2(long n)
{
  long[] memo = new long[n];
  return Fib_DP2(n, memo);
}
```

```
102334155
12586269025
```

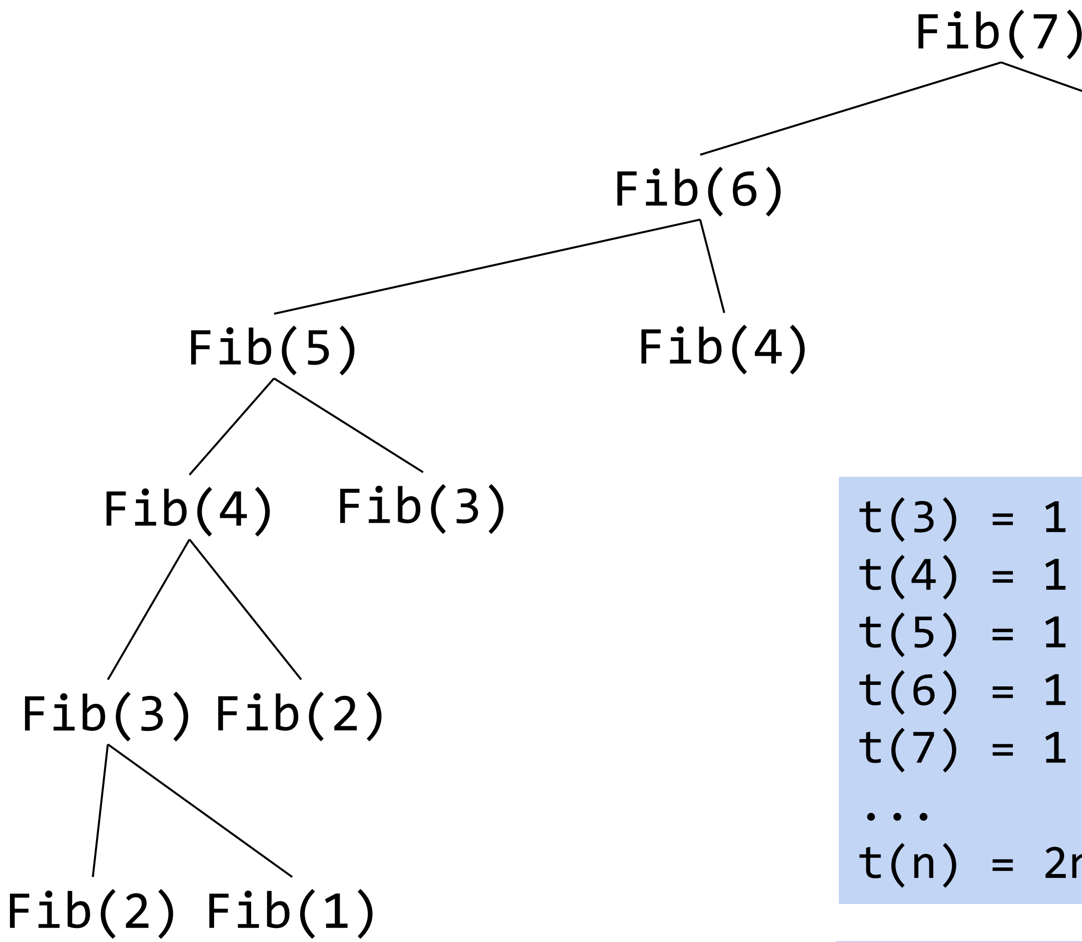
Time Complexity

Now **every** Fibonacci **number** $\boldsymbol{F_n}$ is **calculated** exactly **only one time**



Time Complexity

Time complexity is $t(n)=2n-3$, which is $\boldsymbol{O(n)}$




```
t(3) = 1 + 1 + 1 = 3
t(4) = 1 + 3 + 1 = 5
t(5) = 1 + 5 + 1 = 7
t(6) = 1 + 7 + 1 = 9
t(7) = 1 + 9 + 1 = 11
...
t(n) = 2n - 3
```

E.g., t(7) = 1(root) + 9(left sub-tree) + 1(right left-node) = 11

Outline

- Greedy Algorithms
- Time efficiency of Recursive Methods
- **Dynamic Programming**
  - Problem - Recursive Fibonacci Numbers revisit
  - **Problem - Different Ways of Sum (Self Study)**
  - Problem - Money Change revisit

Problem: Different Ways of Sum

Self study

Given n, find the **number** of **different ways** to write n as the **sum of 1, 3 and 4**

Input: n = 4

Output: 4

Explanation:

4 = 1 + 1 + 1 + 1
= 1 + 3
= 3 + 1
= 4

Problem: Different Ways of Sum

Self study

Given n, find the **number** of **different ways** to write n as the **sum of 1, 3 and 4**

Input: n = 6

Output: 9

Explanation:

$$
\begin{aligned}
6 &= 1 + 1 + 1 + 1 + 1 + 1 \\
&= 1 + 1 + 1 + 3 \\
&= 1 + 1 + 3 + 1 \\
&= 1 + 1 + 4 \\
&= 1 + 3 + 1 + 1 \\
&= 1 + 4 + 1 \\
&= 3 + 1 + 1 + 1 \\
&= 3 + 3 \\
&= 4 + 1 + 1
\end{aligned}
$$

Thinking

Self study

Let’s consider n = 6

The sub-problem “find number of different ways to write **(6 - 1)** as the sum of 1, 3, 4” has been solved

Can we calculate the result for 6?

6 = 1 + 1 + 1 + 1 + 1 + 1
= 1 + 1 + 1 + 3
= 1 + 1 + 3 + 1
= 1 + 1 + 4
= 1 + 3 + 1 + 1
= 1 + 4 + 1
= 3 + 1 + 1 + 1
= 3 + 3
= 4 + 1 + 1

Where is the result of **(6-1)** in this picture?

Thinking

Self study

Let's consider n = 6

The sub-problem "find number of different ways to write **(6 - 1)** as the sum of 1, 3, 4" has been solved, and

The sub-problem "find number of different ways to write **(6 - 3)** as the sum of 1, 3, 4" has been solved

Can we calculate the result for 6?

6 = 1 + 1 + 1 + 1 + 1 + 1
= 1 + 1 + 1 + 3
= 1 + 1 + 3 + 1
= 1 + 1 + 4
= 1 + 3 + 1 + 1
= 1 + 4 + 1
= 3 + 1 + 1 + 1
= 3 + 3
= 4 + 1 + 1

Where is the result of ***(6-3)*** in this picture?

Different Ways of Sum

Self study

- Let D_n be the number of ways to write n as the sum of 1, 3, 4
- Consider one possible solution

$$n = x_1 + x_2 + x_3 \dots + x_m$$

$x_1 = 1$

- The rest must sum to $n - 1$
- Thus, the number of sums that start with $x_1 = 1$ is equal to D_{n-1}

Similar for $x_1 = 3$

- The rest must sum to $n - 3$
- The number of sums that start with $x_1 = 3$ is equal to D_{n-3}

Similar for $x_1 = 4$

- The rest must sum to $n - 4$
- The number of sums that start with $x_1 = 4$ is equal to D_{n-4}

If we can define a problem as some sub-problems, we can use recursions

Different Ways of Sum

Self study

- **Recurrence case**:

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- **Base cases**: we need D_1, D_2, D_3, D_4, D_5. Why?
  - $D_1 = 1$
  - $D_2 = 1$
  - $D_3 = 2$
  - $D_4 = 4$
  - $D_5 = 6$
  - Any alternatives?

If we can define a problem as some sub-problems, we can use recursions

Different Ways of Sum

Self study

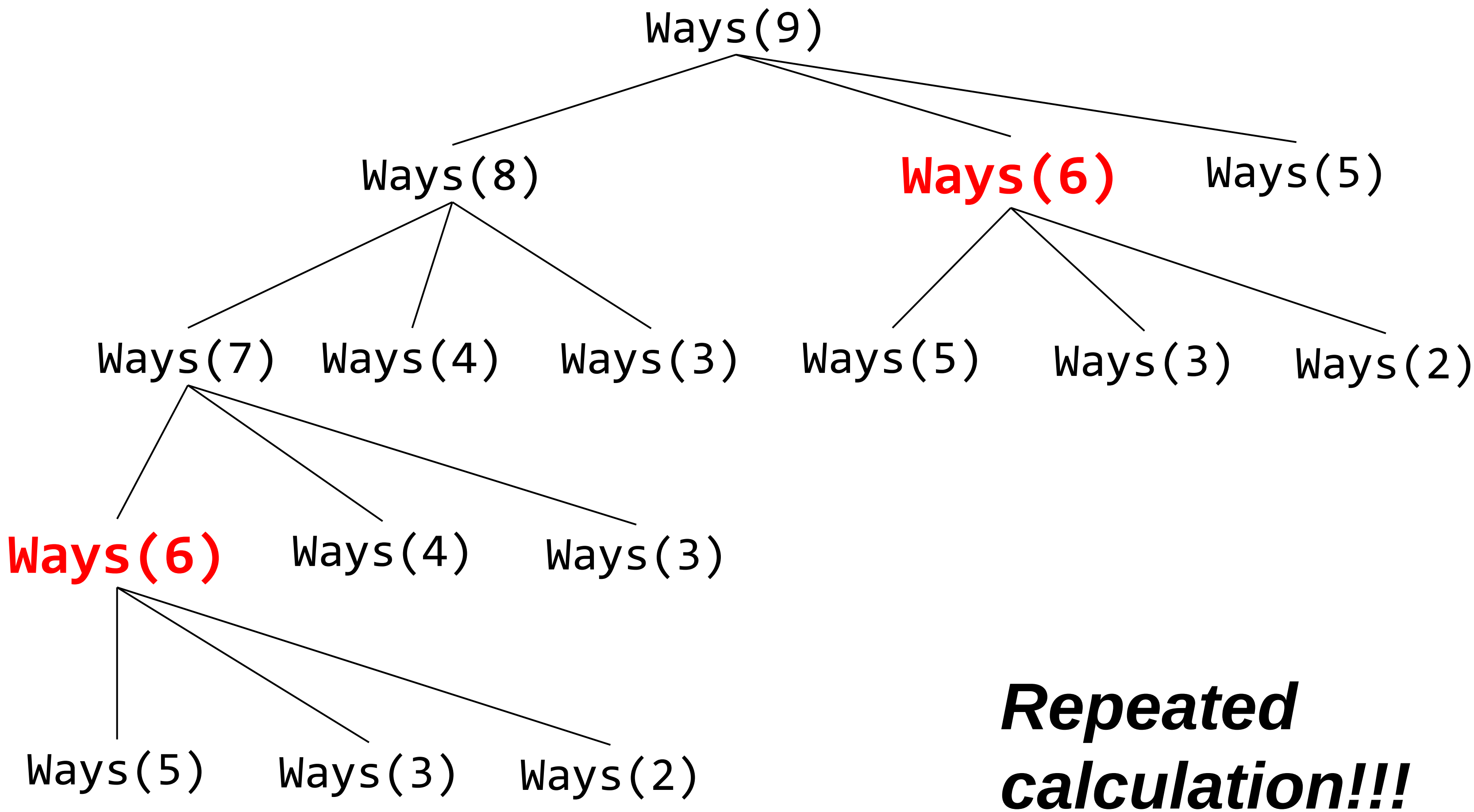
The implementation, once again, includes base cases and recursive cases

```
public static int WaysOfSum(int n)
{
   if (n == 1 || n == 2)
      return 1;
   if (n == 3)
      return 2;
   if (n == 4)
      return 4;
   if (n == 5)
      return 6;

   return WaysOfSum(n - 1) +
      WaysOfSum(n - 3) + WaysOfSum(n - 4);
}
```

Can we do better?

Self study



Repeated calculation!!!

DP for Different Ways of Sum

Self study

```
public static long WaysOfSum2(
                    long n, Dictionary<long, long> memo) {
   if (memo.ContainsKey(n))
      return memo[n];

   long res;
   if (n == 1 || n == 2)
      res = 1;
   else if (n == 3)
      res = 2;
   else if (n == 4)
      res = 4;
   else if (n == 5)
      res = 6;
   else res = WaysOfSum2(n - 1, memo) +
         WaysOfSum2(n - 3, memo) + WaysOfSum2(n - 4, memo);

   memo.Add(n, res);

   return res;
}
```

Outline

- Greedy Algorithms
- Time efficiency of Recursive Methods
- **Dynamic Programming**
  - Problem - Recursive Fibonacci Numbers revisit
  - Problem - Different Ways of Sum
  - **Problem - Money Change revisit (Self-Study)**

Problem: Money Change revisit

Self study

In India, assume available notes/coins are **1, 5, 10, 20, 25, 50** Rupees. What is the **optimum** change for **40** Rupees?

Greedy Algorithm	**Optimum solution**
1 x 25 *1 x 15* *1 x 5*	*2 x 20*
3 notes/coins	**2** notes/coins

How can we reach the real optimum solution?

A “Stupid” Idea

Self study

Examine all possible options of changing 40 Rupees and **pick the best**

40 = 1 x 25, 1 x 10, 1 x 5 (3 notes/coins)
= 1 x 25, 1 x 10, 5 x 1 (7)
= 2 x 20 (2)
= 1 x 20, 2 x 10 (3)
= 1 x 20, 1 x 10, 2 x 5 (4)
= 1 x 20, 4 x 5 (5)
= ...

In programming, many excellent solutions start from such “stupid” ideas that **consider all possibilities**, called **Brute-Force**

Question

Self study

Let ***MinChange(amount)*** be the **optimal change** of a given amount. Can we define it by any sub-problems?

Hint: what is the MinChange(40)? How about MinChange(40-25)?

40 = 1 x 25, 1 x 10, 1 x 5 (3)
= 1 x 25, 1 x 10, 5 x 1 (7)
= 1 x 20, 1 x 20 (2)
= 1 x 20, 1 x 10, 1 x 10 (3)
= 1 x 20, 1 x 10, 2 x 5 (4)
= 1 x 20, 4 x 5 (5)
= ...

If we can define a problem as some sub-problems, we can use recursions

Define sub-problems

Self study

If the optimal includes first change of	Then MinChange
25	*MinChange(40) = 1 + MinChange(40 -* ***25****)*
20	*MinChange(40) = 1 + MinChange(40 -* ***20****)*
10	*MinChange(40) = 1 + MinChange(40 -* ***10****)*
5	*MinChange(40) = 1 + MinChange(40 -* ***5****)*
1	*MinChange(40) = 1 + MinChange(40 -* ***1****)*

Finally, the **real optimal** *MinChange(40)* **must be** the **minimum** of **above**

Define sub-problems

Self study

$$\texttt{MinChange(n)} = 1 + \min_{\text{(valid cases only)}} \begin{cases} \texttt{MinChange(n-50)} \\ \texttt{MinChange(n-25)} \\ \texttt{MinChange(n-20)} \\ \texttt{MinChange(n-10)} \\ \texttt{MinChange(n-5)} \\ \texttt{MinChange(n-1)} \end{cases}$$

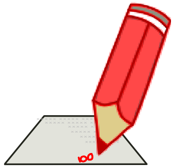
What should be our base cases?

An implementation

Self study

```
public static long MinChange(long amount) {
   long min;
   if (amount >= 50) {
      List<long> alls = new List<long> {
         MinChange(amount - 50), MinChange(amount - 25),
         MinChange(amount - 20), MinChange(amount - 10),
         MinChange(amount - 5),  MinChange(amount - 1)
      };
      min = alls.Min();
   }
   else if (amount >= 25) {
      List<long> alls = new List<long> {
         MinChange(amount - 25), MinChange(amount - 20),
         MinChange(amount - 10), MinChange(amount - 5),
         MinChange(amount - 1)
      };
      min = alls.Min();
   }
   ...
   else if (amount >= 1) {
      min = MinChange(amount - 1);
   }
   else return 0;

   return 1 + min;
}
```



Can you make the code more concise?

Can we do better?

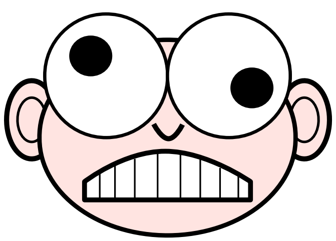
Self study

For large amount, we solve **6 recursive problems**

```
                MC(110)

MC(109)   MC(105)   MC(100)   MC(90)   MC(90)
```

1 problem **becomes 6** subproblems, **each** of which will **subsequently become 6** subproblems…



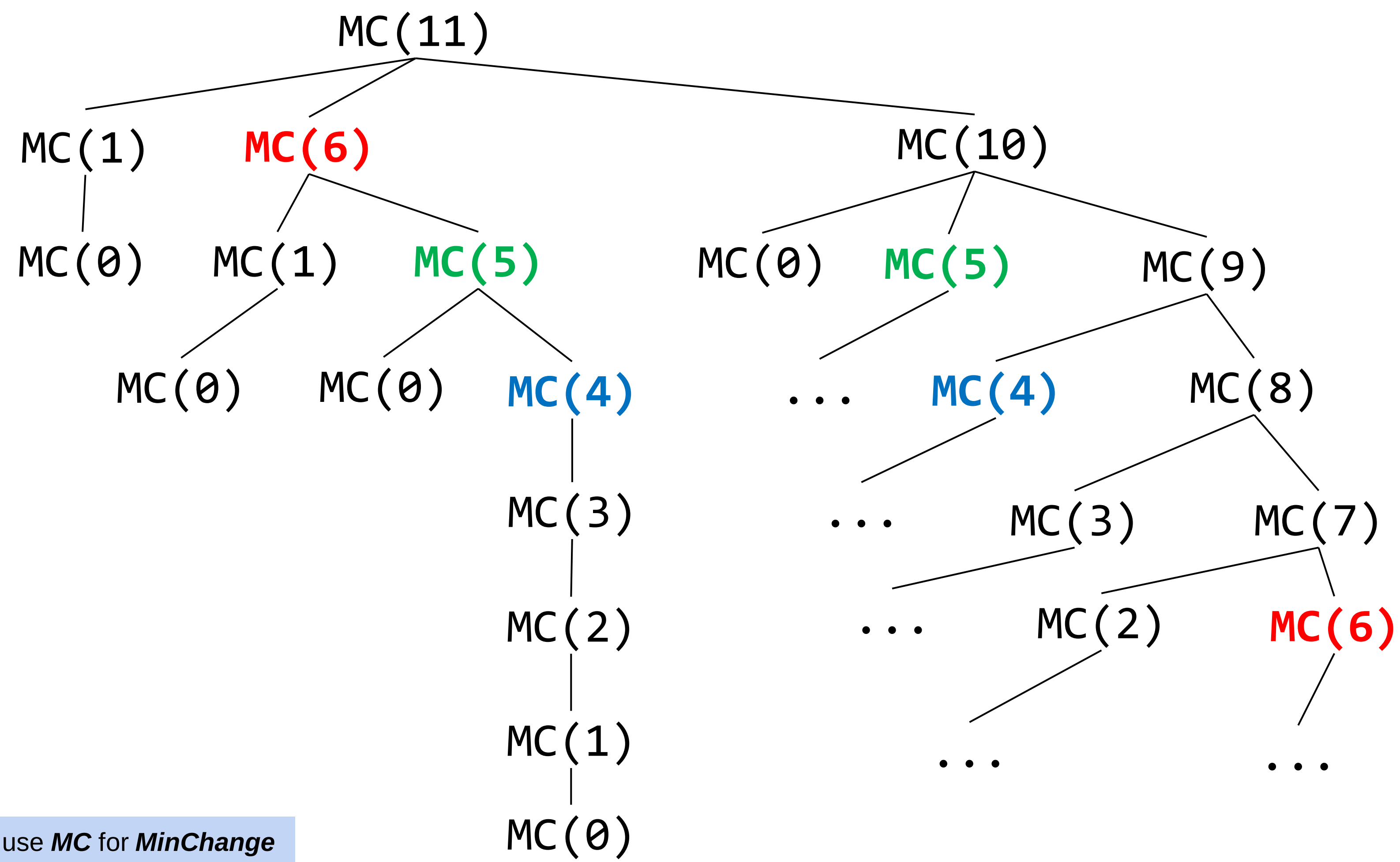
Very expensive!!!

I use ***MC*** for ***MinChange***

Can we do better?

Self study

But many calculations are **duplicate** ☺



I use ***MC*** for ***MinChange***

DP for Money Change Problem

Self study

```
public static long MinChange(long amount, Dictionary<long, long> memo) {
   if (memo.ContainsKey(amount)) return memo[amount];

   long min;
   if (amount >= 50) {
      List<long> alls = new List<long> {
         MinChange(amount - 50, memo), MinChange(amount - 25, memo),
         MinChange(amount - 20, memo), MinChange(amount - 10, memo),
         MinChange(amount - 5, memo),  MinChange(amount - 1, memo)
      };
      min = alls.Min();
   }
   else if (amount >= 25) {
      ...
   }
   ...
   else if (amount >= 1) {
      min = MinChange(amount - 1, memo);
   }
   else return 0;

   memo.Add(amount, 1 + min);
   return 1 + min;
}
```

Time complexity: $t(n)=O(6N)=O(N)$, where 6 is the number of different kinds of coins

Question

Do you remember recursive Factorial?

Can we use DP for it?

```
Factorial(4)
4*Factorial(3)
4*3*Factorial(2)
4*3*2*Factorial(1)
4*3*2*1*Factorial(0)
4*3*2*1*1=24
```

If there's **no duplicate computations**, **memorization** is **useless**!

Readings

- Data structures and abstractions with Java, 4ed – Chapter 7, Recursion, section 7.22 - 7.27, 7.37 - 7.41, *Frank M.Carrano and Timothy M. Henry*
- Data structures and algorithms using C# - Chapter 17, Advanced Algorithms, *by Michael McMillan (2007)*