2. Prologue

2.1 Introduction

2.2 Computational complexity

2.3 Time complexity of common functions

2.4 Recurrence relation

2.5 Fibonacci

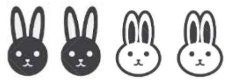
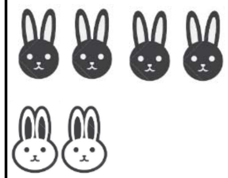
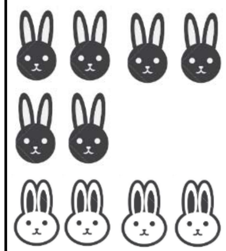
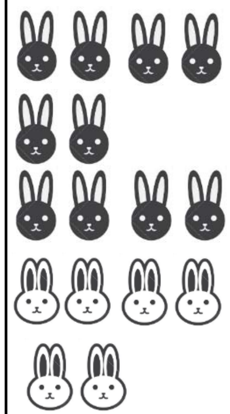
2.5 Fibonacci

- What is Fibonacci sequence?
  - Count the number of rabbits at n-th month
  - Rule

1. At the first month, a new pair of rabbits arrive.
2. A rabbit of more than two months can mate.
3. A pair of rabbits bear a new pair of rabbit every month.
4. A rabbit never dies.

2.5 Fibonacci

- What is Fibonacci sequence?

Month	0	1	2	3	4	5	6	7
Rabbit (pair)								
No. of pairs	0	1	1	2	3	5	8	13

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
```

2.5 Fibonacci

- What is Fibonacci sequence?

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
```

  - Fibonacci?
    - An Italian mathematician in 13th century

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & if\ n > 1 \\ 1 & if\ n = 1 \\ 0 & if\ n = 0. \end{cases}$$

2.5 Fibonacci

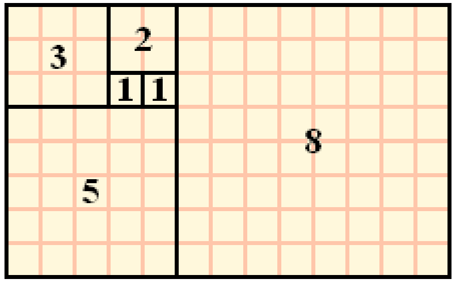
- Why is Fibonacci sequence important?

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

0,	1,	1,	2,	3,	5,	8,	13,	21,	34,	…
1,	1,	2,	3,	5,	8,	13,	21,	34,	55,	…

  – Converges to 0.6180339… = 1/1.6180339…

  – Golden section = 1.618



2.5 Fibonacci

- Problem:
  - How to compute Fibonacci sequence?
  - What is the n-th Fibonacci number?

- Solution:
  - Simple and intuitive, but not efficient → **bruteforce algorithm**
  - Simple and efficient, but improvable
  - More efficient → **Optimal algorithm**

2.5 Fibonacci

(1) Solution 1: A recursive call

```
int Fib ( int n )
{
    if ( n == 0 || n == 1 )
        return n;

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

– Does it satisfy the five requirements?

2.5 Fibonacci

(1) Solution 1: A recursive call

```
int Fib ( int n )
{
 if ( n == 0 || n == 1 )
        return n;

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

- Efficiency? → Time complexity?
- $O(2^n)$ time complexity → Bruteforce algorithm
- Is there any faster ways?

2.5 Fibonacci

(2) Solution 2: An iterative approach

```
int Fib ( int n )
{
    int FS[n+1];
    FS[0] = 0;
    FS[1] = 1;

    for ( int i = 2; i <= n; i++ )
        FS[i] = FS[i-1] + FS[i-2];

    return FS[n];
}
```

– Efficiency? → O(n) time complexity
  - **Are you satisfied?**

2.5 Fibonacci

(3) Solution 3: A divide-and-conquer approach

$$\begin{cases} F_{n-1} = F_{n-1} \\ F_n \quad\;\, = F_{n-1} + F_{n-2} \end{cases}$$

$$\begin{aligned}
\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} \\
&= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-3} \\ F_{n-2} \end{pmatrix} \\
&= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-4} \\ F_{n-3} \end{pmatrix} \\
&\cdots\cdots\cdots \\
&= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}
\end{aligned}$$

– Solve ( )^n → solve similar problem such as k^n

2.5 Fibonacci

(3) Solution 3: A divide-and-conquer approach

– Iterative approach

```
int get_k_n ( int k, int n )  //    compute k^n
{
      int i;
      int kn;
      for ( i = 1, kn = 1; i <= n; i = i+1 )
            kn = k * kn;

      return kn;
}
```

– Efficiency? → O(n) time complexity

2.5 Fibonacci

(3) Solution 3: A divide-and-conquer approach

– Another iterative approach (divide and conquer)

```
int get_k_n ( int k, int n )  //    compute k^n
{
      int i;
      int kn;
      for ( i = 2, kn = k; i <= n; i = i*i )
            kn = kn * kn;

      return kn;
}
```

– Efficiency? → O(log n) time complexity

2.5 Fibonacci

(3) Solution 3: A divide-and-conquer approach

– Divide and conquer approach

```
int get_k_n ( int k, int n )  //    compute k^n
{
      if ( n == 0 )
            return 1;
      if ( n == 1 )
            return k;

      int kn = get_k_n ( k, n/2 );
      return kn * kn;
}
```

– Efficiency? → O(log n) time complexity

2.5 Fibonacci

- In summary
  - Bruteforce algorithm → $O(2^n)$
    - Duplicate recursive call
  - Efficient algorithm → $O(n)$
    - Loop with auxiliary memory
  - More efficient algorithm → $O(\log n)$
    - Divide and conquer
  - Optimal algorithm → ??

2. Prologue

2.1 Introduction

2.2 Computational complexity

2.3 Time complexity of common functions

2.4 Recurrence relation

2.5 Fibonacci

Contents