EE2331 Data Structures and Algorithms

Recursion

Outline

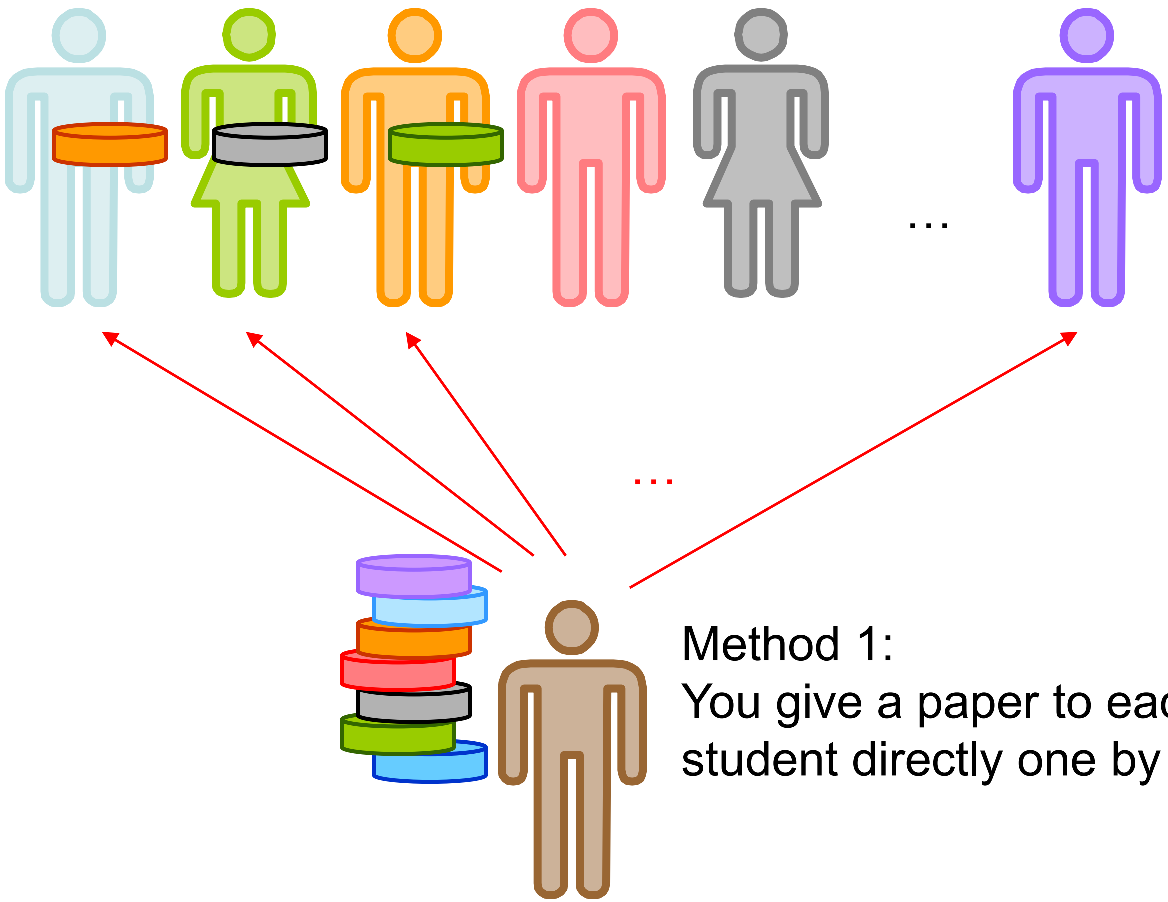
- Recursion
  - Factorial
  - Fibonacci Sequence
  - Binary Search

Introduction

- Recursion is a powerful and elegant algorithm in solving complex problems. It usually results in more "clean" code that is easier to understand

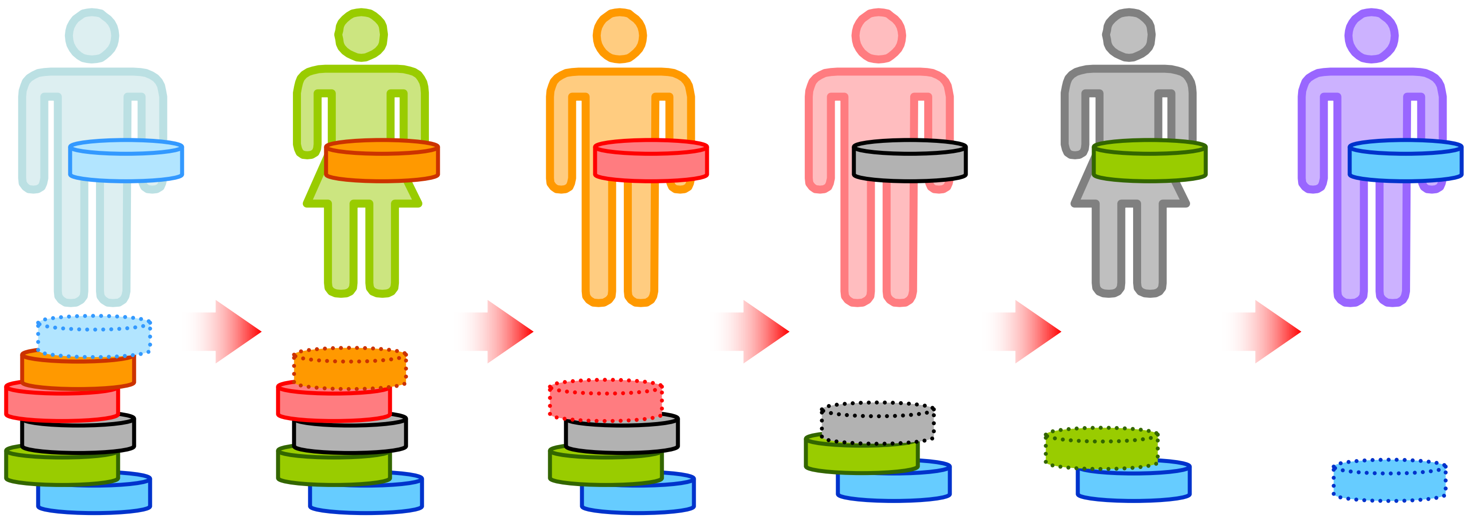
- Daily life problems solved by recursion
  - Distributing quiz papers
    - You want to distribute the quiz papers to each of the students in the classroom.
    - Method 1 – You give one paper to each student directly one by one
    - Method 2 – You ask each student to pick up one paper and pass the rest to the neighbor until everyone has a paper

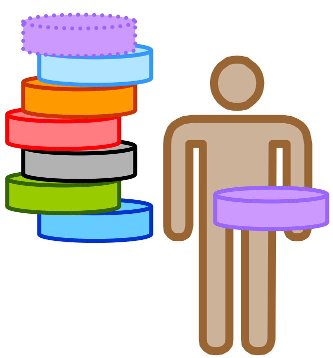
Introduction



Method 1:
You give a paper to each student directly one by one

Introduction





Method 2:
Each student picks up one paper and passes the rest (n-1) to the neighbor

Introduction

- Method 1 – Iteration (Pseudo Code):

```
distributeSomething(people[], items[]) {
    for each person in people[]
        take one item from items[]
        give item to person
}
```

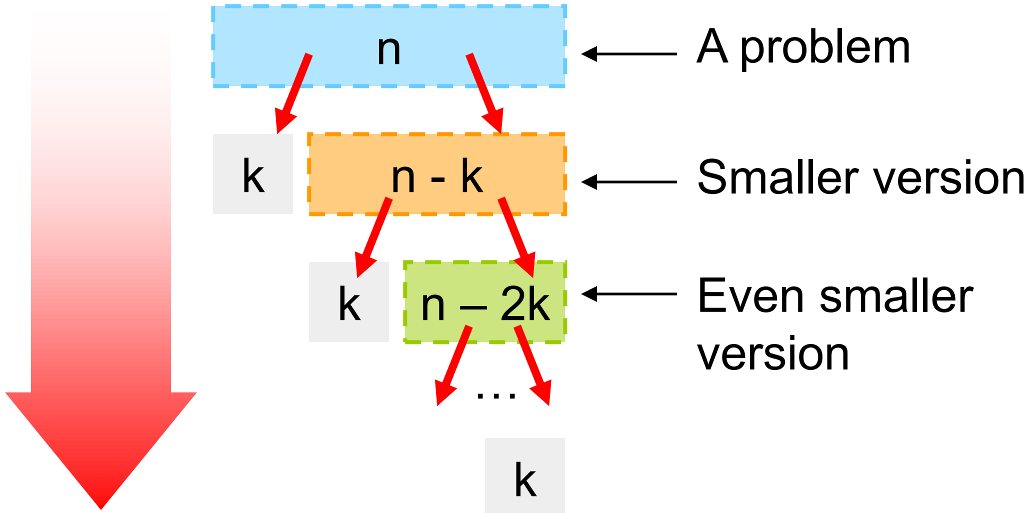
- Method 2 – Recursion (Pseudo Code):

```
distributeSomething(people[], items[], k) {
    if (each person in people[] got an item)
        return

    pick one item from items[]                    pass to (k+1)th person in the next cycle
    give item to kth person
    distributeSomething(people[], items[], k+1)
}
```

Recursion

- A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values
- Sometimes, a complicated problem can be simplified by breaking it into same problems of smaller scale
- Recursion is a technique that solves a problem by solving a smaller problem of the same kind
- Recursion is good when the problem is recursively defined, or when the data structure that the algorithm operates on is recursively defined.



Recursion

- In C++, a function may call itself directly

```
int functionA(…) {
    …
    functionA(…);
    …
}
```

- When a function call itself recursively, each invocation gets a fresh set of all automatic (local) variables, independent of the previous set. These automatic variables, parameters and return address (back to the caller) are stored collectively into a **call stack**, known as an activation record. The record is removed (pop from stack) when the function returns. Since each call creates a separate record, a subroutine can be reentrant, and recursion is automatically supported.

Two Essential Steps

- Express the problem in the form of recurrence essentially requires to define two things:

- **Base Case**
  - You must have some base cases, which can be solved without recursion
- **Recursive Case**
  - The cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case

Factorial Function

- *n factorial*, *n*!, is defined as the product of all integers between *n* and 1

- $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$
- $0! = 1$ (the base case)
- $1! = 1$
- $2! = 2 \times 1 = 2$
- $3! = 3 \times 2 \times 1 = 6$
- …

Factorial Function

- $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$

```
int factorial(int n) {
    int result = 1;
    while (n > 1)
        result *= n--;
    return result;
}
```

Loop n-1 times

Time Complexity:

O(n)

Space Complexity:

2 variables (n and *result*) throughout the whole function

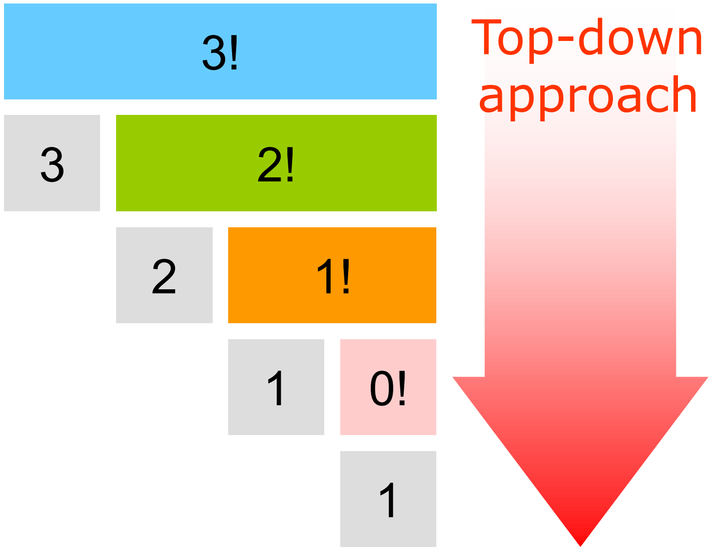
= O(1)

(i.e. independent of the size of n)

Factorial Function

- $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$ (closed-form)
- $n! = n \times (n-1)!$ (recursive form)

- 3! = 3 x 2!
- 2! = 2 x 1!
- 1! = 1 x 0!
- 0! = 1 (base case)



Factorial Function

- $n! = n \times (n - 1)!$

```
int factorial(int n) {
    //precondition: n >= 0
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

Terminate condition (base case, not solved by recursion)

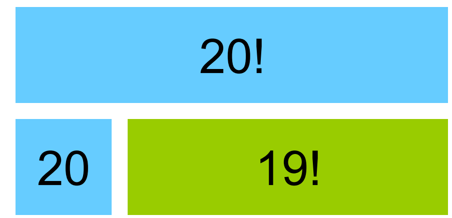
Invariant: as $n > 0$, so $n - 1 \geq 0$
Therefore, factorial(n-1) returns (n-1)! correctly

```
int factorial(int n) {
    return (n == 0? 1: n * factorial(n – 1));
}
```

Factorial Function: Example

- Calling factorial(20)

```
int factorial(int n) { n = 20
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```



Space requirement: allocated one integer (int n) through out the whole function

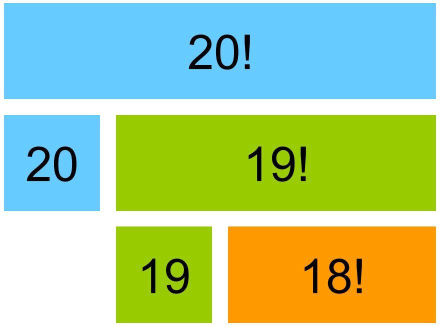
Factorial Function: Example

- Calling factorial(20)

```
int factorial(int n) {   n = 20
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

```
int factorial(int n) {   n = 19
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

Another integer (int n) being allocated in this function (i.e. totally 2 integers in memory)



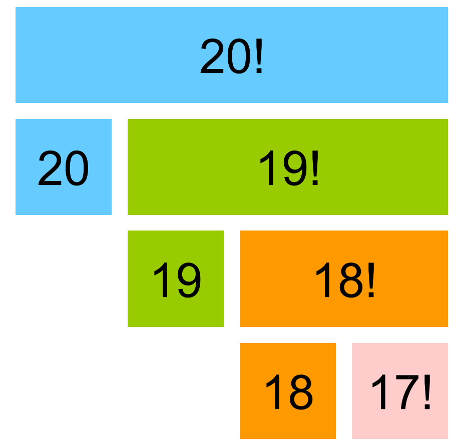
Factorial Function: Example

- Calling factorial(20)

```
int factorial(int n) {   n = 20
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

```
int factorial(int n) {   n = 19
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

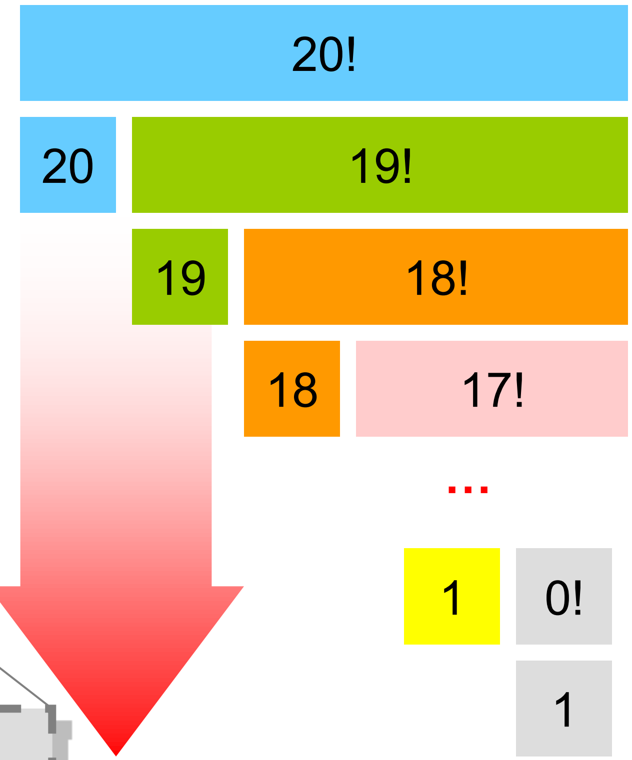
```
int factorial(int n) {   n = 18
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```



Another integer (int n) being allocated in this function (i.e. totally 3 integers in memory)

Factorial Function: Example

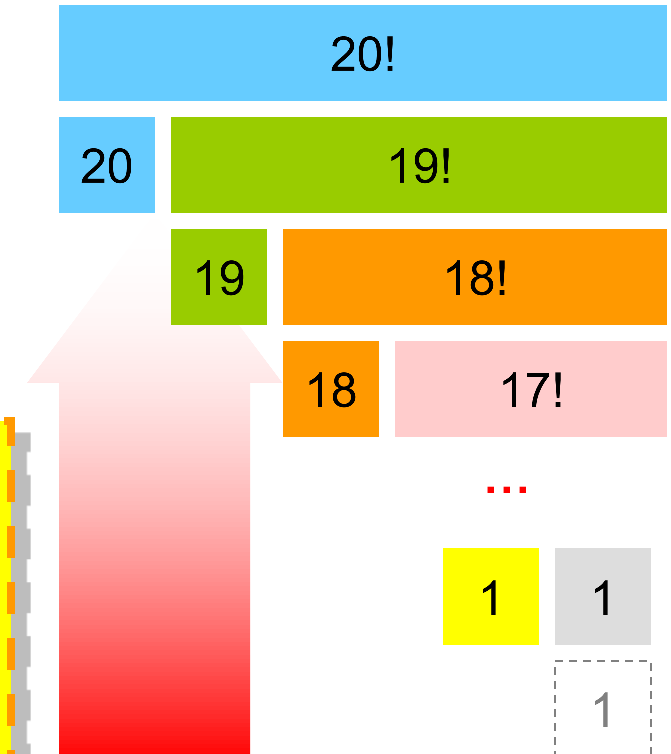
```
int factorial(int n) {  n = 20
}
    int factorial(int n) {  n = 19
    }
        int factorial(int n) {  n = 18
            …
        }
            int factorial(int n) {  n = 2
            }
                int factorial(int n) {  n = 1
                    if (n == 0) return 1;
                    return (n * factorial(n – 1));
                }
                    int factorial(int n) {  n = 0
                        if (n == 0) return 1;
                        return (n * factorial(n – 1));
                    }
```



Another integer (int n) being allocated in this function (i.e. totally 21 integers in memory)

Factorial Function: Example

```
int factorial(int n) {   n = 20
  int factorial(int n) {   n = 19
    int factorial(int n) {   n = 18
      …
    }
}
  }
      int factorial(int n) {   n = 2
      …
      }
        int factorial(int n) {   n = 1
            if (n == 0) return 1;
            return (n * [ 1 ]);
        }
```

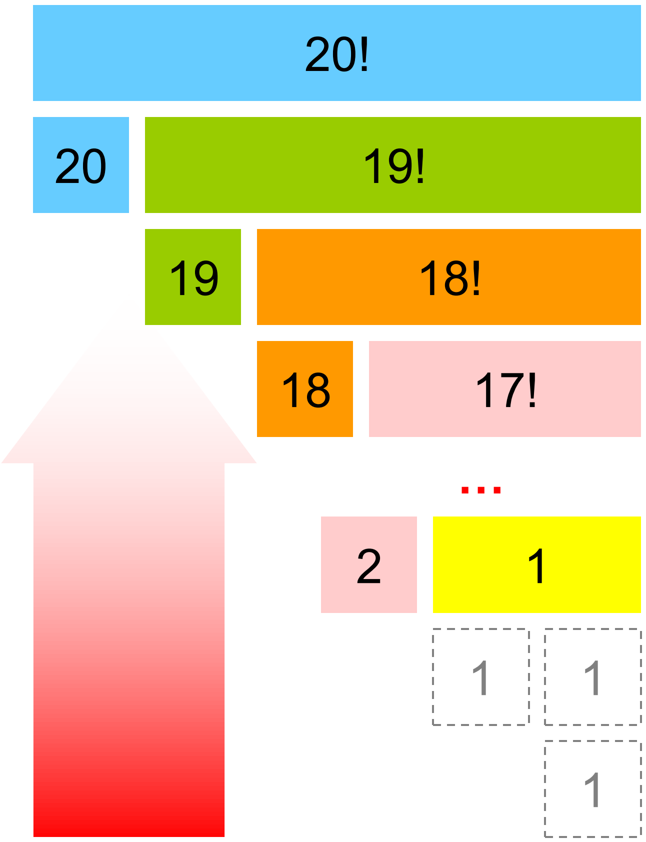


The function of n = 0 returns 1 and now totally only 20 integers in memory

Factorial Function: Example

```
int factorial(int n) {   n = 20
    int factorial(int n) {   n = 19
        int factorial(int n) {   n = 18
            ...
        }
            int factorial(int n) {   n = 2
                if (n == 0) return 1;
                return (n * [      1      ]);
            }
    }
}
```

The function of n = 1 returns 1
and now totally only 19 integers
in memory

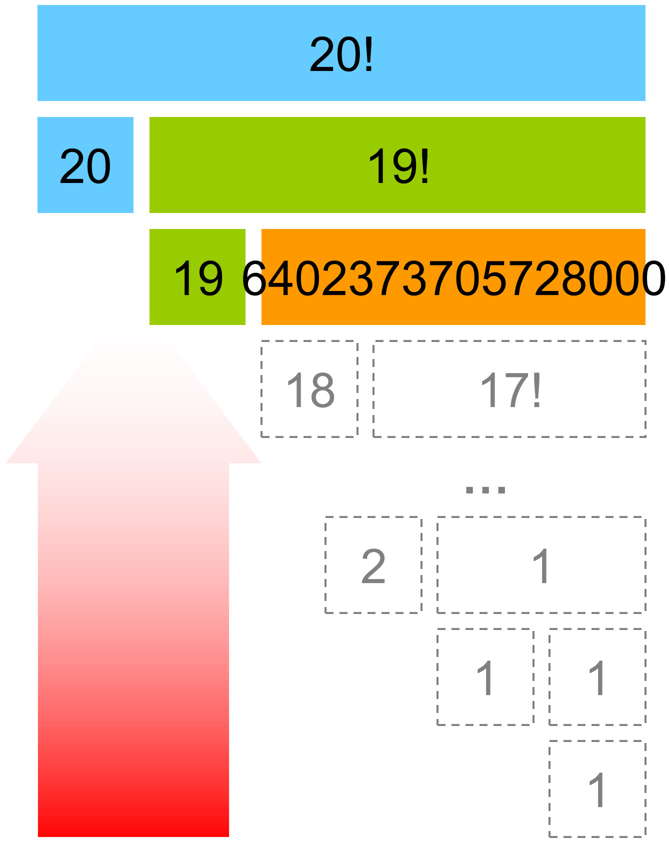


Factorial Function: Example

```
int factorial(int n) {  n = 20
}
```

```
int factorial(int n) {  n = 19
    if (n == 0) return 1;
    return (n * 6402373705728000);
}
```

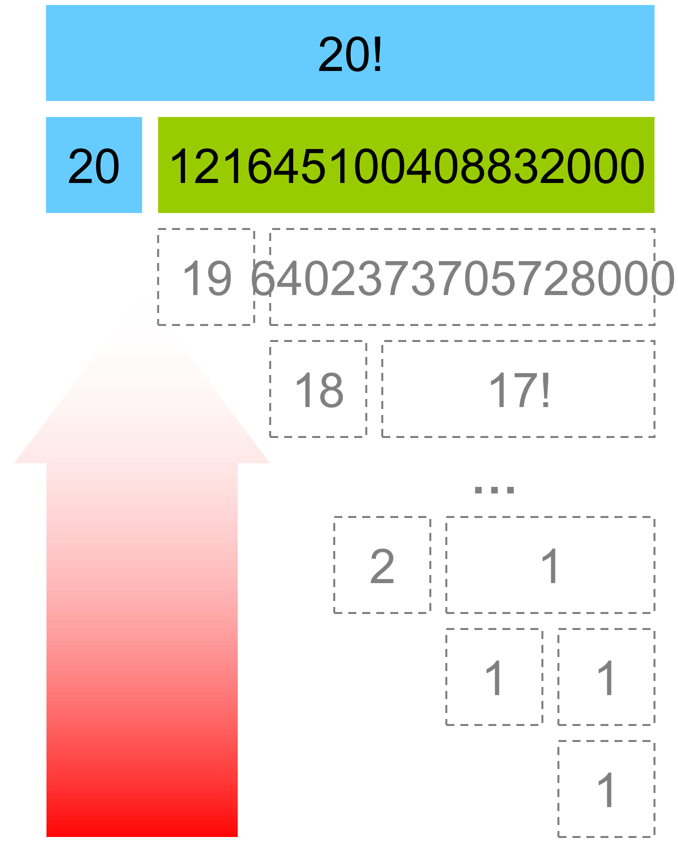
The function of n = 18 returns 6402373705728000 and now totally only 2 integers in memory



Factorial Function: Example

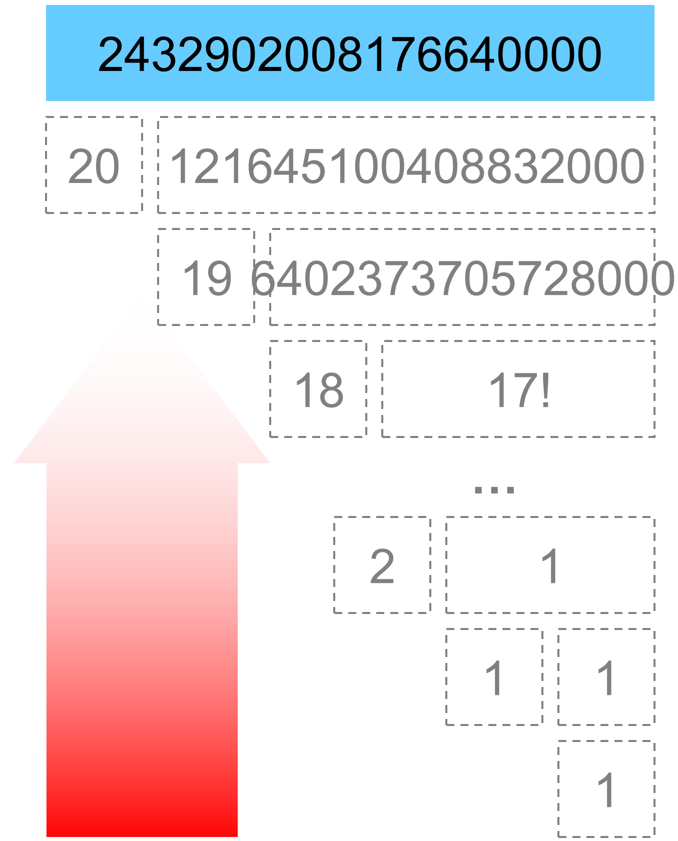
```
int factorial(int n) { n = 20
    if (n == 0) return 1;
    return (n * 121645100408832000);
}
```

The function of n = 19 returns 121645100408832000 and now totally only 1 integer in memory



Factorial Function: Example

The function of n = 20 returns 2,432,902,008,176,640,000 and now no integers in memory



Space Complexity =
Time Complexity =

Time Complexity

Running time

```
int factorial(int n) {
    if (n == 0) return 1;
    return (n * factorial(n – 1));
}
```

int factorial(int n) { → T(n)

1 if (n == 0) return 1; → O(1)

2 return (n * factorial(n – 1)); → T(n-1)

T(n) = $\begin{cases} O(1), & n = 0 \\ O(1) + T(n-1), & n \geq 1 \end{cases}$

- Let T(n) be the running time of input size equals to n
- The running time for line 1 is a constant O(1), let us use 1.
- The running time for line 2 is equal to a constant O(1) + T(n-1)

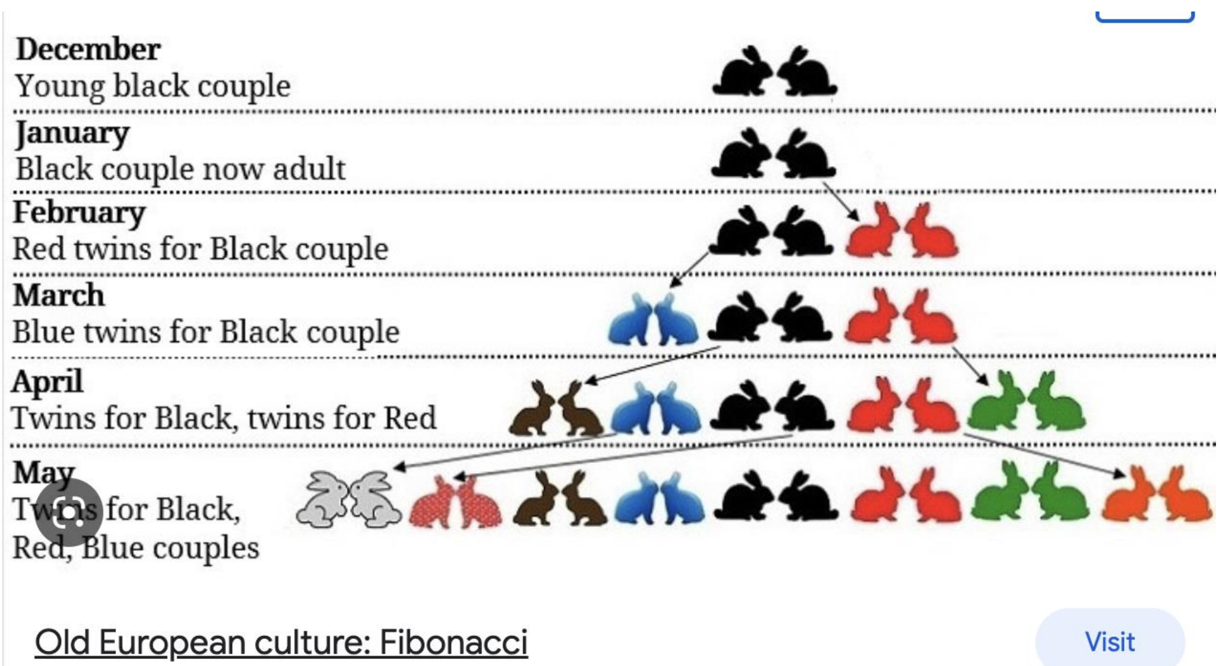
How to solve the above equation?

For n >= 1
T(n)=1+T(n-1)
T(n)=1+1+T(n-2)
T(n)=1+1+1+T(n-3)
….

T(n)=i+T(n-i)
Until n-i = 0, so i = n
T(n)=n+T(0)

Fibonacci sequence

- $fib(n) = \underbrace{fib(n-1) + fib(n-2)}_{\text{Recursively defined}}$



Leonardo Fibonacci

Fibonacci Sequence

- By definition, the first two Fibonacci numbers are 0 and 1, and each remaining number is the sum of the previous two.
- In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation:

  $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$

- So the Fibonacci number sequence is as follows:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

Fibonacci Sequence

- $fib(n) = fib(n - 1) + fib(n - 2)$
- e.g. Compute $fib(4)$

$= fib(3) + fib(2)$

$= [fib(2) + fib(1)] + [fib(1) + fib(0)]$

$= [\ \{fib(1) + fib(0)\} + fib(1)] + [fib(1) + fib(0)]$

$= [\ \{1 + 0\} + 1] + [1 + 0]$

$= \underline{3}$

Fibonacci Sequence

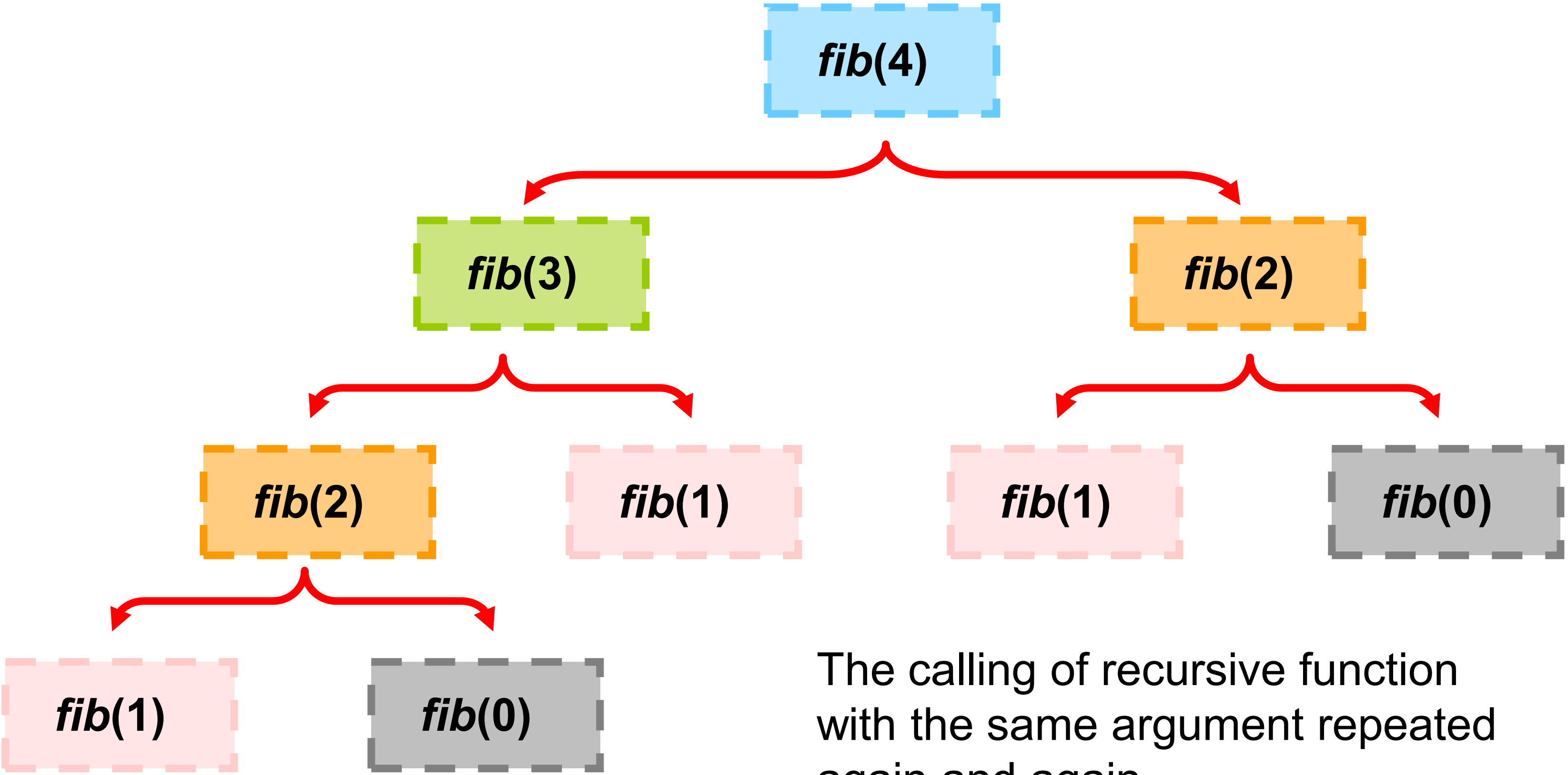
- *fib*(*n*) = *fib*(*n* - 1) + *fib*(*n* - 2)

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return (fib(n - 1) + fib(n – 2));
}
```

Check base cases before recursion

Calling itself (recursion)

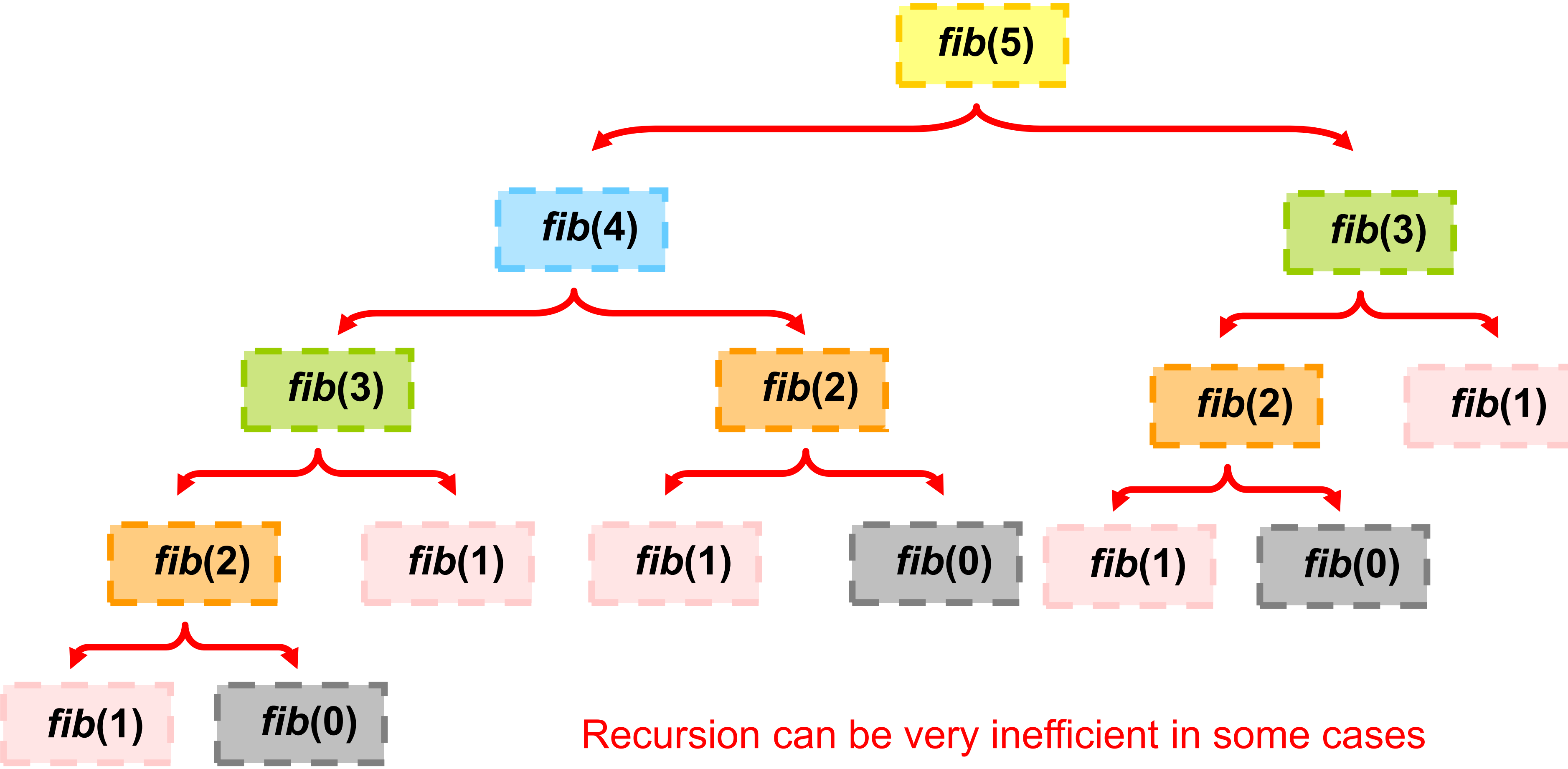
Fibonacci Sequence



The calling of recursive function with the same argument repeated again and again

e.g. *fib*(2) being called twice, *fib*(1) being called 3 times

Fibonacci Sequence



Recursion can be very inefficient in some cases

In-class exercise

- $fib(n) = fib(n - 1) + fib(n - 2)$

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return (fib(n - 1) + fib(n – 2));
}
```

Exercise:
Compute fib(n) using a non-recursive method; use loop/iteration. Write your pseudocode.

In-class exercise

- *fib*(*n*) = *fib*(*n* - 1) + *fib*(*n* - 2)

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return (fib(n - 1) + fib(n – 2));
}
```

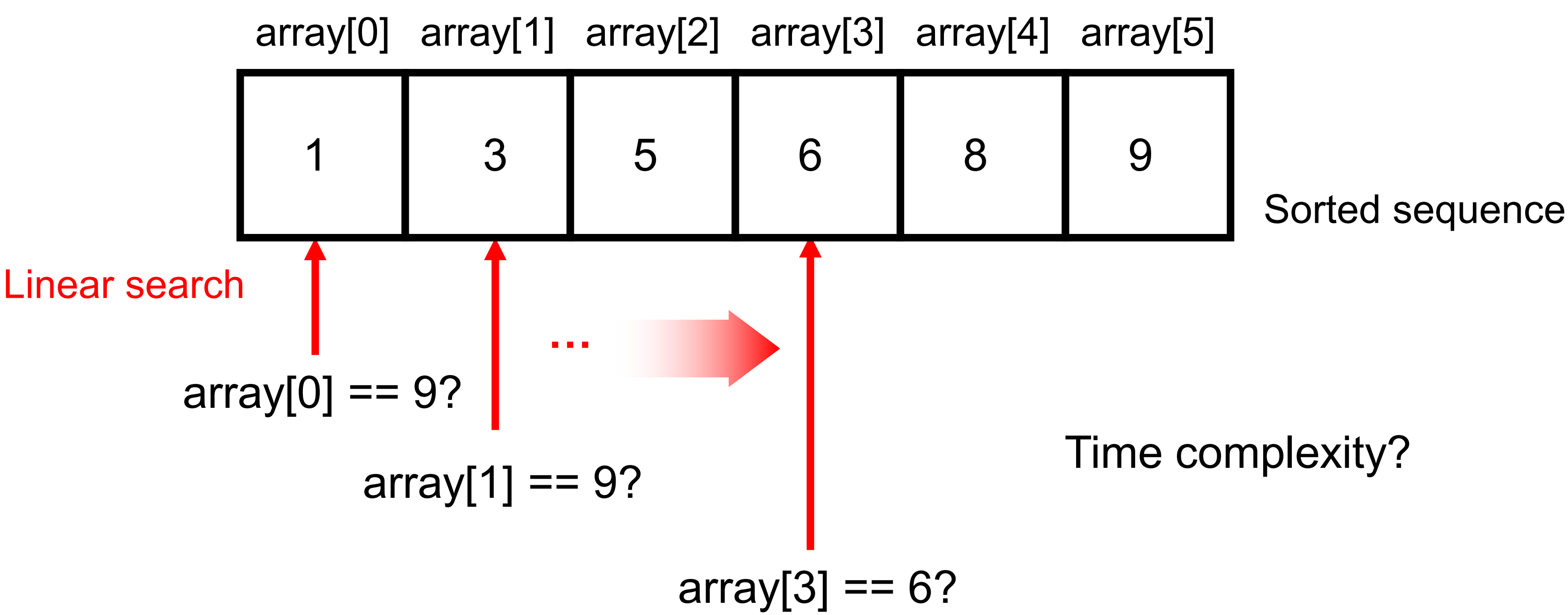
Exercise:
Compute fib(n) using a non-recursive method; use loop/iteration. Write your pseudocode.

There are two methods. In method 1, you can use an array to save all the fib(i) for i=0 to n. You can use vector or int* to declare this array. In method 2, we just use 2 int variables to save fib(n-1) and fib(n-2).

```
int fib(int n) {
        int grandp=0; int p=1; int current;
        for(int i=2; i<=n; i++)
                {
                        current=grandp+p;
                        grandp=p;
                        p=current;
                }
}
```

Recursive function continued. Oct. 13, 23

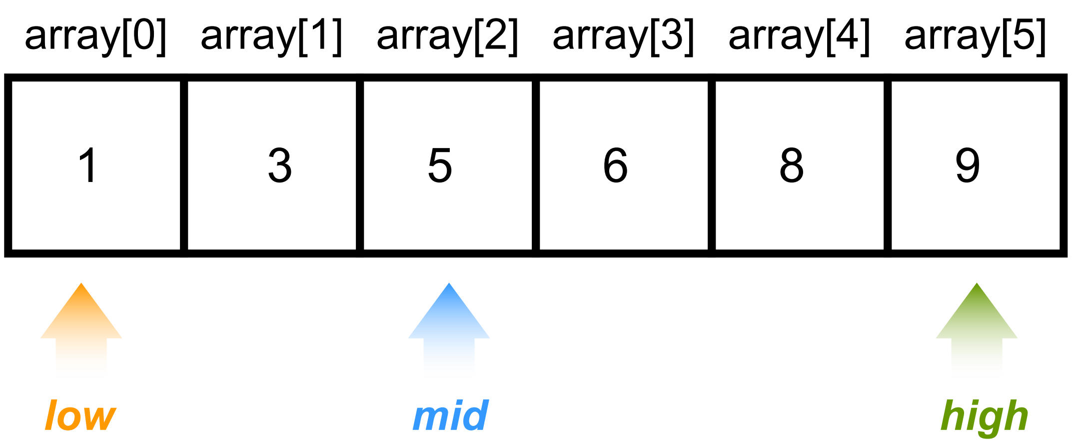
Searching in Sorted Array



Time complexity?

To look for a certain element in the array, e.g. 6

Binary Search



Sorted sequence

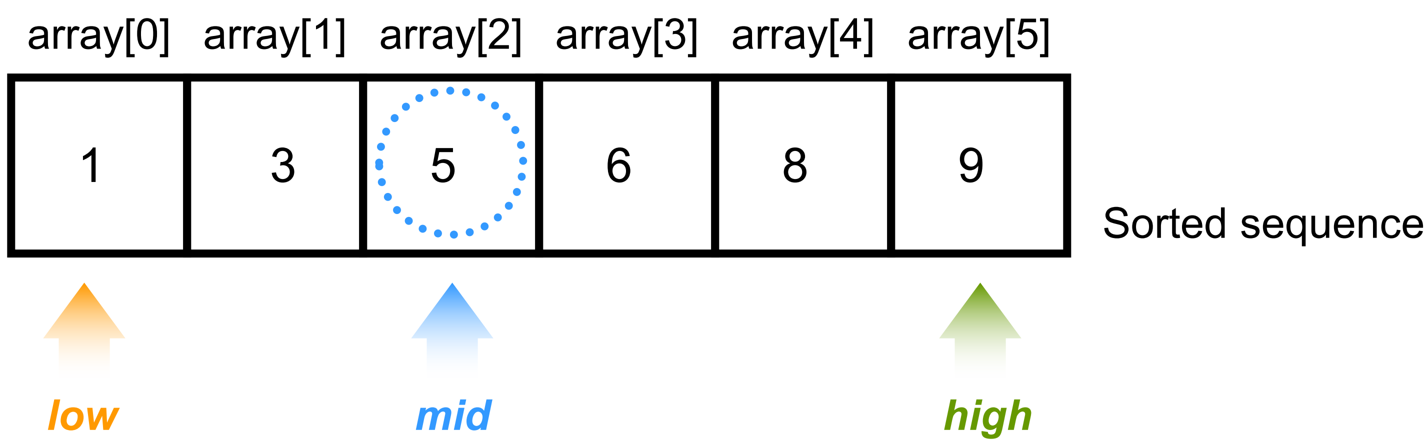
low = ***index*** ***of the left most element***

high = ***index*** ***of the right most element***

mid = **(*high* + *low*) / 2**

To look for a certain element in the array, e.g. 6

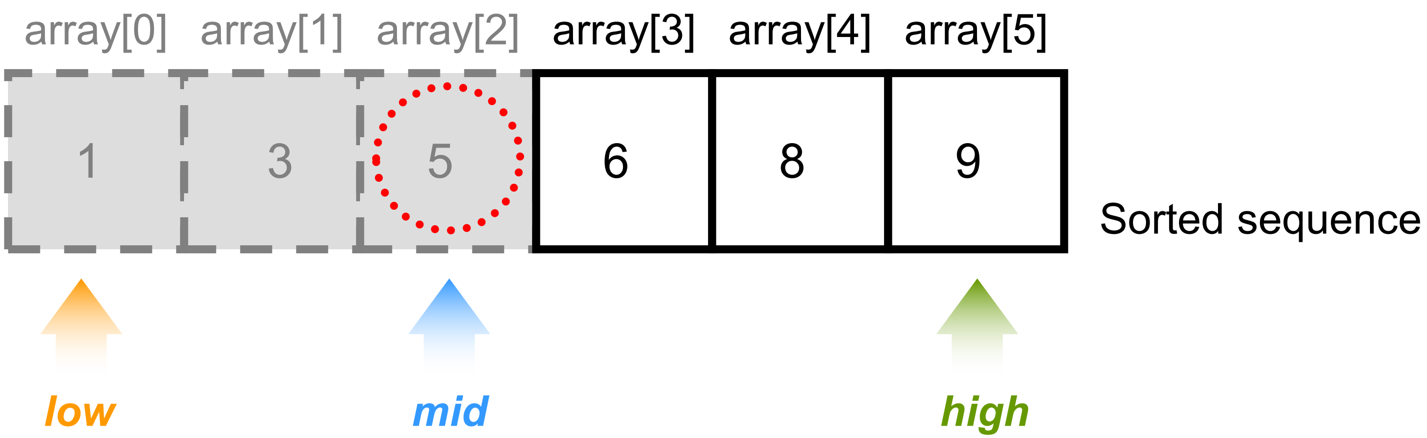
Binary Search



Compare array[*mid*] with 6

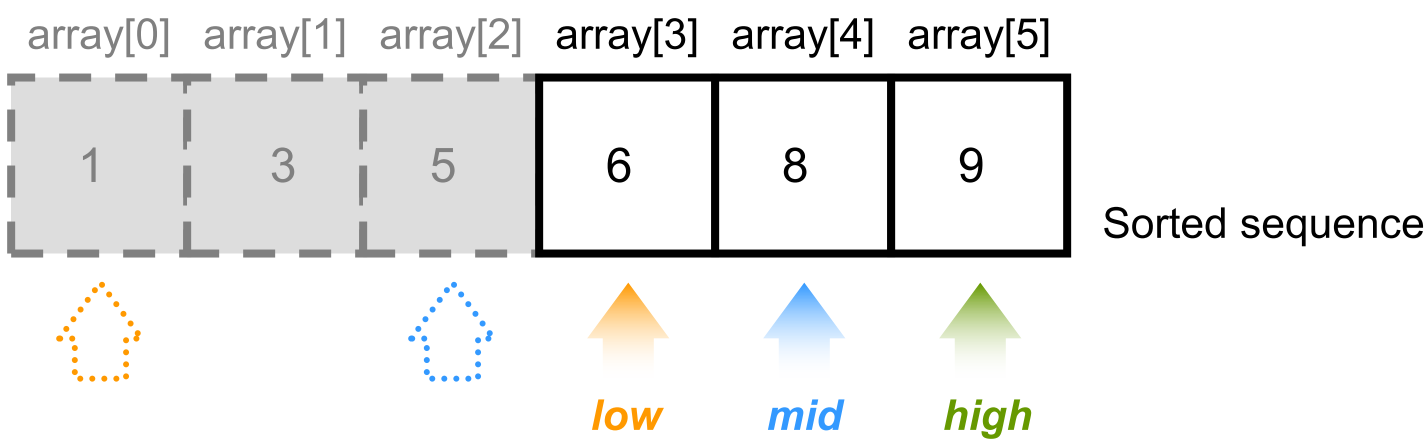
- **array[*mid*] > 6 : search left sub-sequence**
- **array[*mid*] == 6 : the answer!**
- **array[*mid*] < 6 : search right sub-sequence**

Binary Search



Since 5 < 6, the answer must be in the right sub-sequence

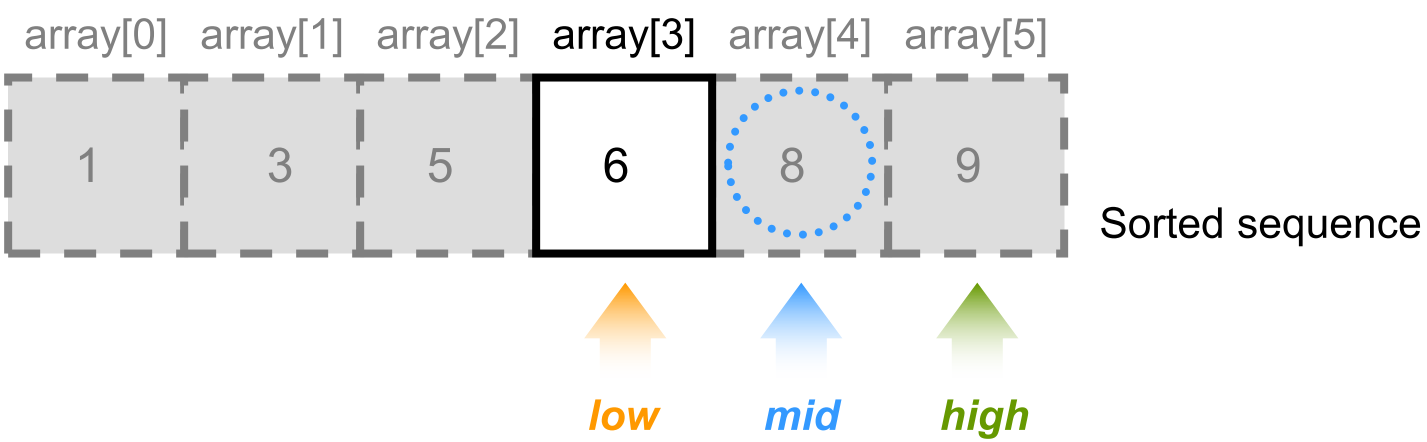
Binary Search



The new search windows is [*mid* + 1, *high*]

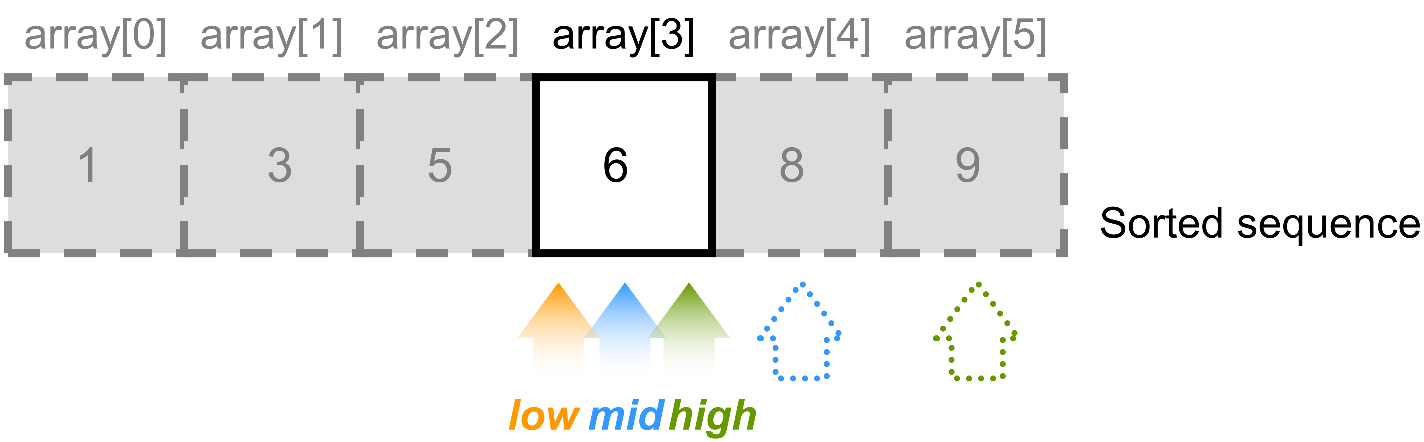
update *low* and recalculate *mid* pointers

Binary Search



Since 8 > 6, the answer must be in the left sub-sequence

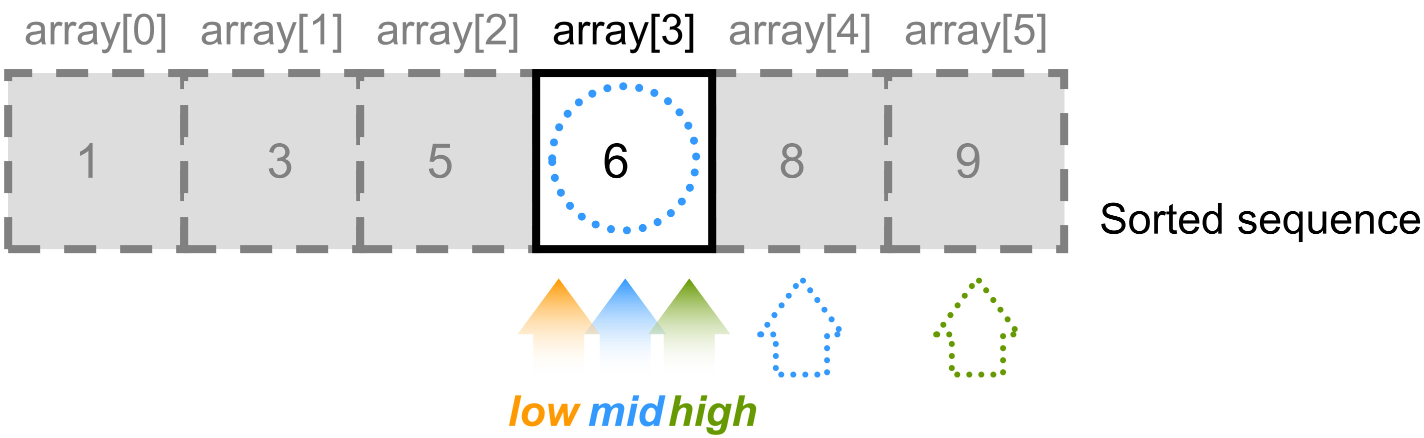
Binary Search



The new search window is [*low*, *mid* – 1]

Update *high* and recalculate *mid* pointers

Binary Search



array[3] == 6

The answer is 3

Binary Search

- The no. of elements to be searched is halved in each search cycle
- The expected number of elements to be searched is $\log_2 n + 1$, where n is total number of elements
- The procedures in each search cycle are the same and could be recursively defined
- Binary Search can be implemented with Iterative (looping) approach or Recursive approach

Iterative Implementation of Binary Search

- An iterative approach (using loops)
  - Update either the *mid*, *low* or *high* indexes in each iteration
  - Loop until *low > high* (the failure condition)
  - Time: O(log n) / Space: O(1)

```
int binsch(int array[], int low, int high, int x) {

    int mid;
    while (low <= high) {
        mid = (high + low) / 2;
        if (array[mid] == x) return mid;  //x has been found
        if (array[mid] > x) high = mid - 1;
        if (array[mid] < x) low = mid + 1;
    }
    return -1;  //cannot find x in the array

}
```

Recursive Implementation of Binary Search

```
int binsch(int array[], int low, int high, int x) {
    int mid = (high + low) / 2;
    if (low > high)
        return -1;  //cannot find x in the array
    if (array[mid] == x)
        return mid; //x is found
    if (array[mid] > x)
        return binsch(array, low, mid - 1, x);
    if (array[mid] < x)
        return binsch(array, mid + 1, high, x);

}
```

Time complexity? Space complexity?

Recursion vs. Iteration

- Iteration sometimes can be used in place of recursion
  - An iterative algorithm uses a looping structure
  - A recursive algorithm uses a branching structure
- Recursive solutions are often less efficient, in terms of both time and space, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

Pros of Using Recursion

- Natural and elegant way of solving problems
- Logical simplicity
  - e.g. Fibonacci sequence
- Self-documentation, increase readability
  - e.g. factorial, recursive binary search
- Handle complicated problems
- Programming efficiency

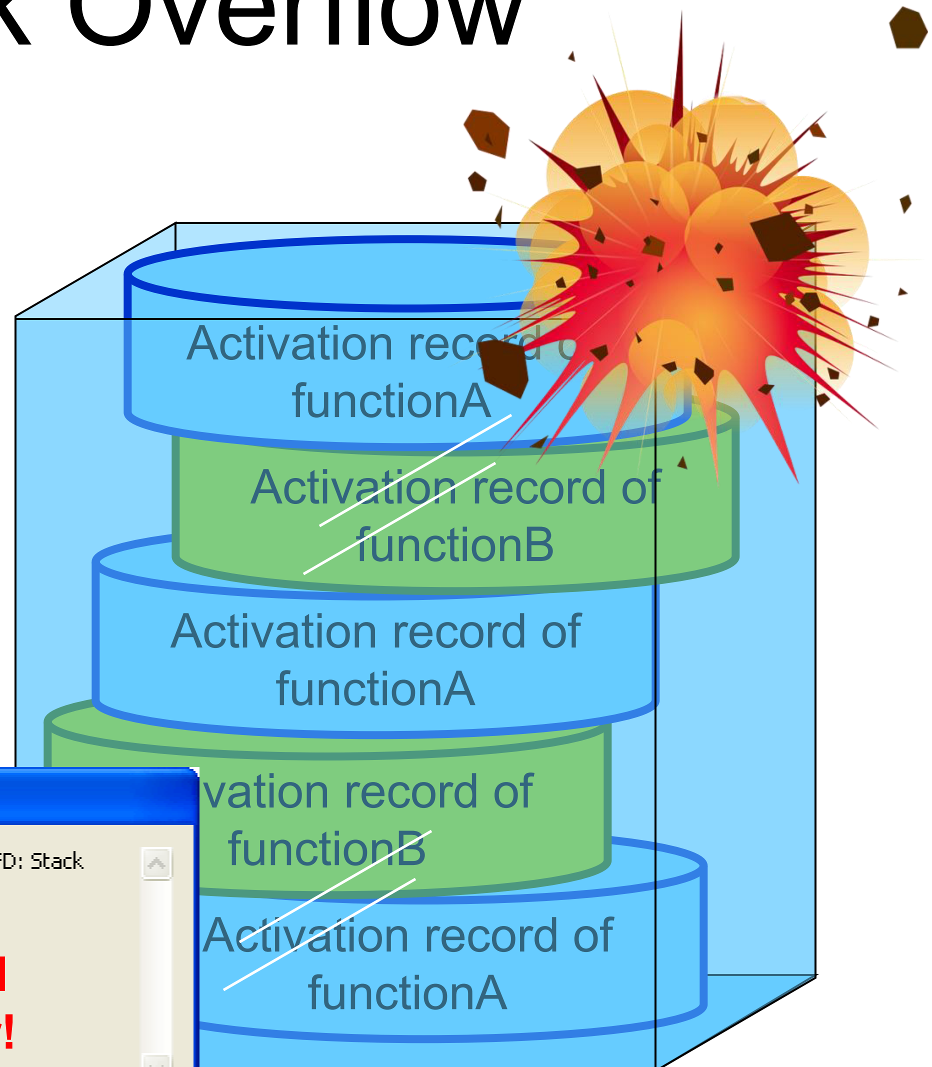
Cons of Using Recursion

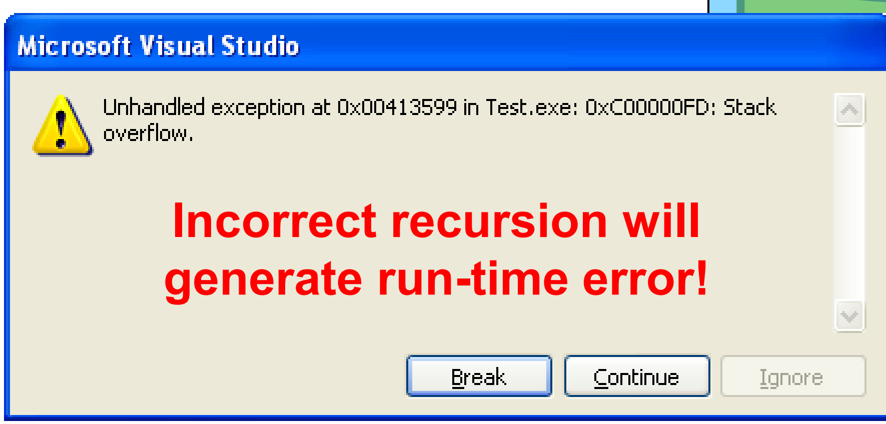
- Often more expensive than non-recursive solution, in terms of time and space
- Space:
  - Activation record and stack
  - Recursive algorithm may need space proportional to the number of nested calls to the same function.
- Time:
  - Introduced overhead
  - The operations involved in calling a function - allocating, and later releasing, local memory, copying values into the local memory for the parameters, branching to / returning from the function

Call Stack Overflow

```
void functionA() {
    functionB();
}
```

```
void functionB() {
    functionA();
}
```





Incorrect recursion will generate run-time error!