Course

04 – Recursion and Algorithm Complexity

Lecturer:

Contact:

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

Recursion and the basic components of recursive algorithms

Recursion

Definition
Recursion is a repetitive process in which an algorithm calls itself.

- Direct : A $\rightarrow$ A
- Indirect : A $\rightarrow$ B $\rightarrow$ A

Example

Factorial

$$Factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

Using recursion:

$$Factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times Factorial(n-1) & \text{if } n > 0 \end{cases}$$

Basic components of recursive algorithms

Two main components of a Recursive Algorithm

1. Base case (i.e. stopping case)
2. General case (i.e. recursive case)

Example

Factorial

$$Factorial(n) = \begin{cases} 1 & \text{if } n = 0 \quad \text{base case} \\ n \times Factorial(n-1) & \text{if } n > 0 \quad \text{general case} \end{cases}$$

Recursion

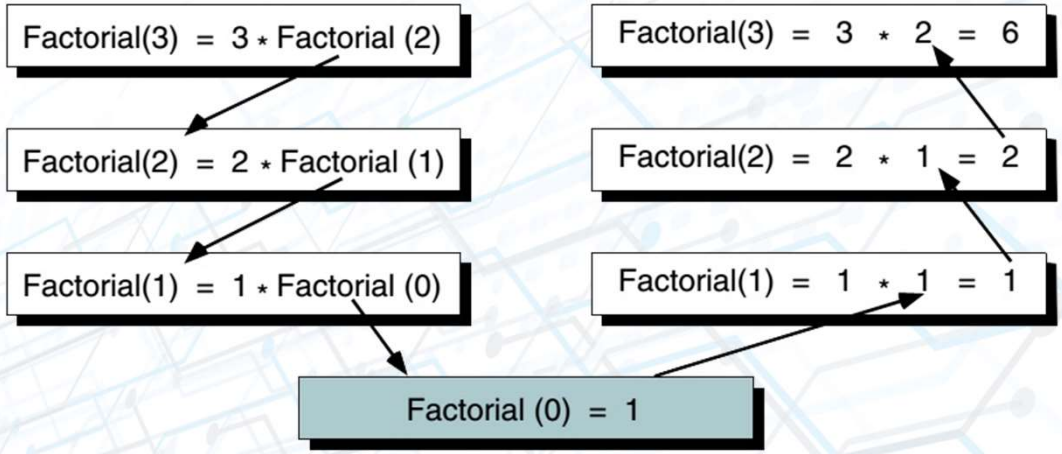


Figure 1: Factorial (3) Recursively (source: Data Structure - A pseudocode Approach with C++)

Recursion

Factorial: Iterative Solution

Algorithm iterativeFactorial(n)
Calculates the factorial of a number using a loop.
Pre: n is the number to be raised factorially
Post: n! is returned - result in factoN

```
i = 1
factoN = 1
while i <= n do
    factoN = factoN * i
    i = i + 1
end
return factoN
End iterativeFactorial
```

Recursion

Factorial: Recursive Solution

Algorithm recursiveFactorial(n)

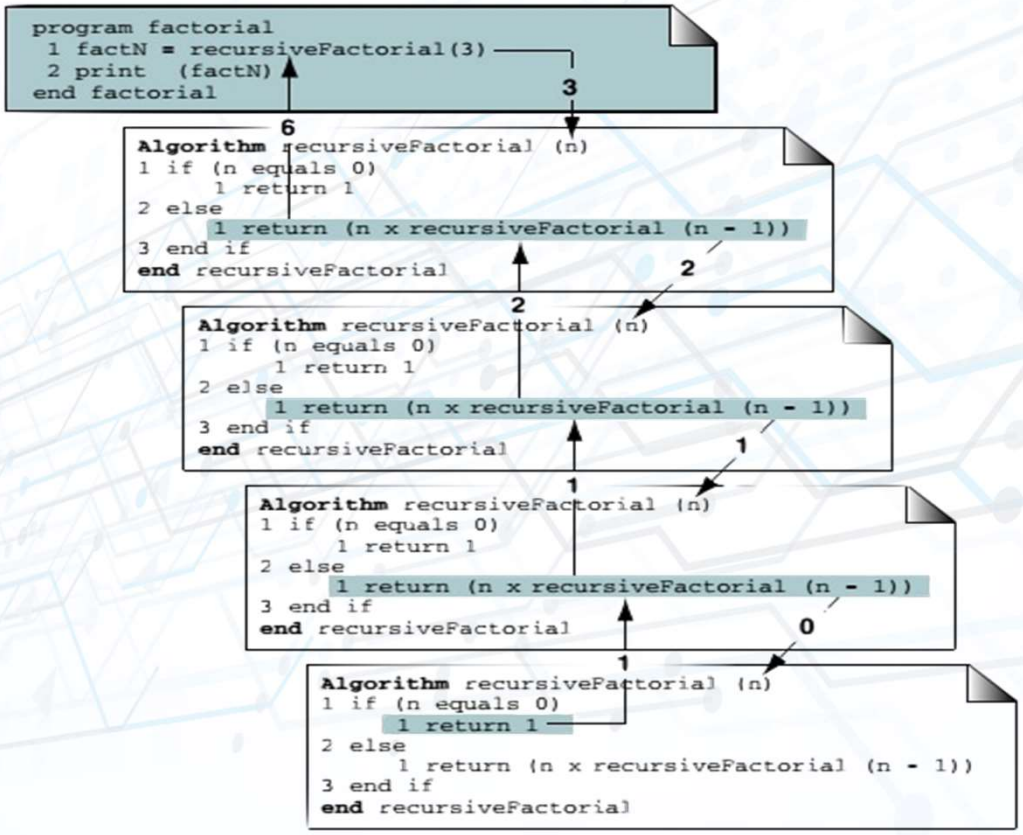
Calculates the factorial of a number using a recursion.

Pre: n is the number to be raised factorially

Post: n! is returned

```
if n = 0 then
    return 1
else
    return n * recursiveFactorial(n-1)
end
End recursiveFactorial
```

Recursion


program factorial
1 factN = recursiveFactorial(3)
2 print (factN)
end factorial
3
6
Algorithm recursiveFactorial (n)
1 if (n equals 0)
1 return 1
2 else
1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
2
2
Algorithm recursiveFactorial (n)
1 if (n equals 0)
1 return 1
2 else
1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
1
1
Algorithm recursiveFactorial (n)
1 if (n equals 0)
1 return 1
2 else
1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
0
1
Algorithm recursiveFactorial (n)
1 if (n equals 0)
1 return 1
2 else
1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial

Properties of recursion

Properties of all recursive algorithms

- A recursive algorithm solves the large problem by using its solution to a simpler sub-problem
- Eventually the sub-problem is simple enough that it can be solved without applying the algorithm to it recursively.
  → This is called the base case.

Designing recursive algorithms

The Design Methodology

Every recursive call must either solve a part of the problem or reduce the size of the problem.

Rules for designing a recursive algorithm

1. Determine the base case (stopping case).
2. Then determine the general case (recursive case).
3. Combine the base case and the general cases into an algorithm.

Limitations of Recursion

- A recursive algorithm generally runs more slowly than its nonrecursive implementation.
- BUT, the recursive solution shorter and more understandable.

Greatest Common Divisor

Definition

$$\gcd(a,b) = \begin{cases} a & if\ b = 0 \\ b & if\ a = 0 \\ \gcd(b, a \bmod b) & otherwise \end{cases}$$

Example

gcd(12, 18) = 6

gcd(5, 20) = 5

Greatest Common Divisor

Algorithm gcd(a, b)
Calculates greatest common divisor using the Euclidean algorithm.
Pre: a and b are integers
Post: greatest common divisor returned

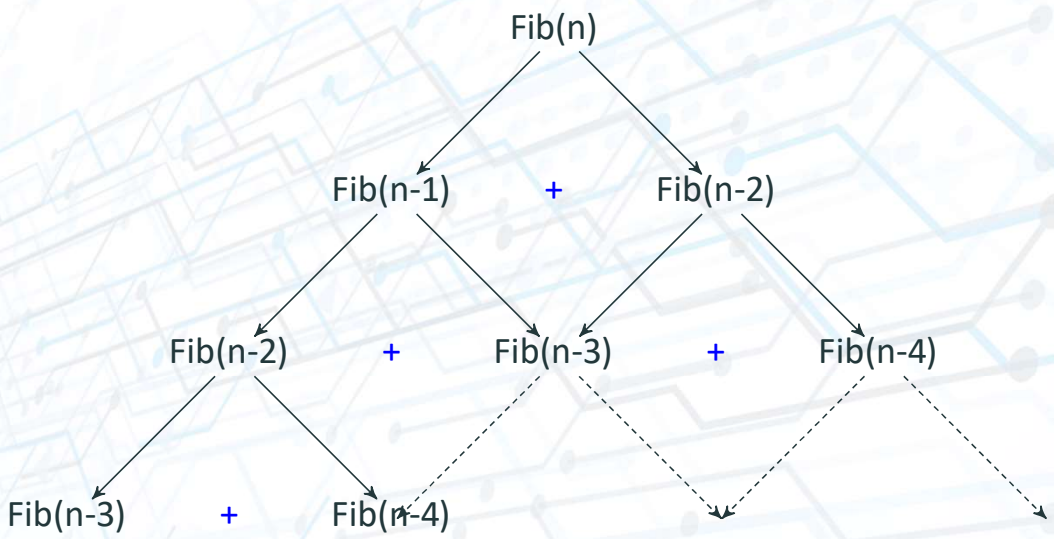
```
if b = 0 then
    return a
end
if a = 0 then
    return b
end
return gcd(b, a mod b)
End gcd
```

Fibonacci Numbers

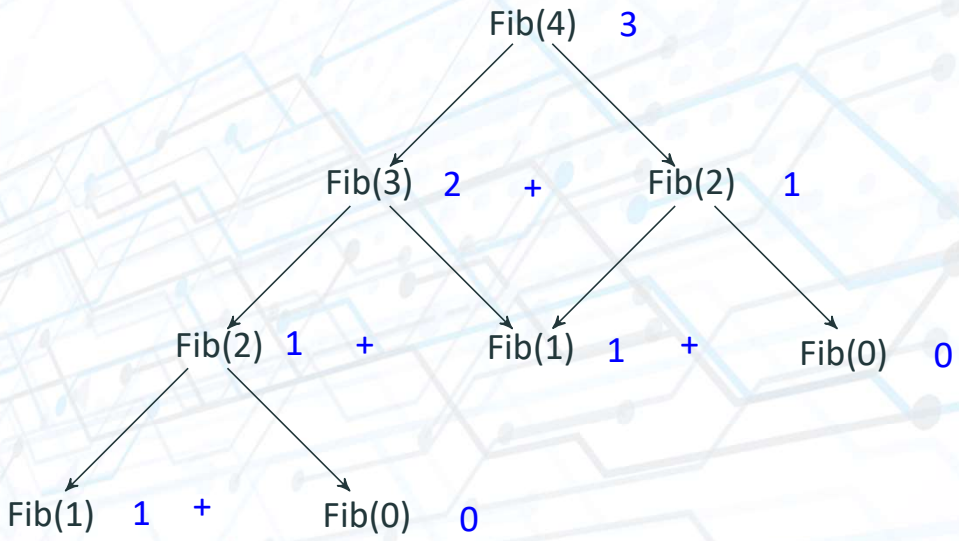
Definition

$$Fibonacci(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ Fibonacci(n-1) + Fibonacci(n-2) & otherwise \end{cases}$$

Fibonacci Numbers



Fibonacci Numbers



Result

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Fibonacci Numbers

Algorithm fib(n)

Calculates the n^{th} Fibonacci number.

Pre: n is postive integer

Post: the n^{th} Fibonnacci number returned

```
if n = 0 or n = 1 then
    return n
end
return fib(n-1) + fib(n-2)
End fib
```

Fibonacci Numbers

No	Calls	Time	No	Calls	Time
1	1	< 1 sec.	11	287	< 1 sec.
2	3	< 1 sec.	12	465	< 1 sec.
3	5	< 1 sec.	13	753	< 1 sec.
4	9	< 1 sec.	14	1,219	< 1 sec.
5	15	< 1 sec.	15	1,973	< 1 sec.
6	25	< 1 sec.	20	21,891	< 1 sec.
7	41	< 1 sec.	25	242,785	1 sec.
8	67	< 1 sec.	30	2,692,573	7 sec.
9	109	< 1 sec.	35	29,860,703	1 min.
10	177	< 1 sec.	40	331,160,281	13 min.

The Towers of Hanoi

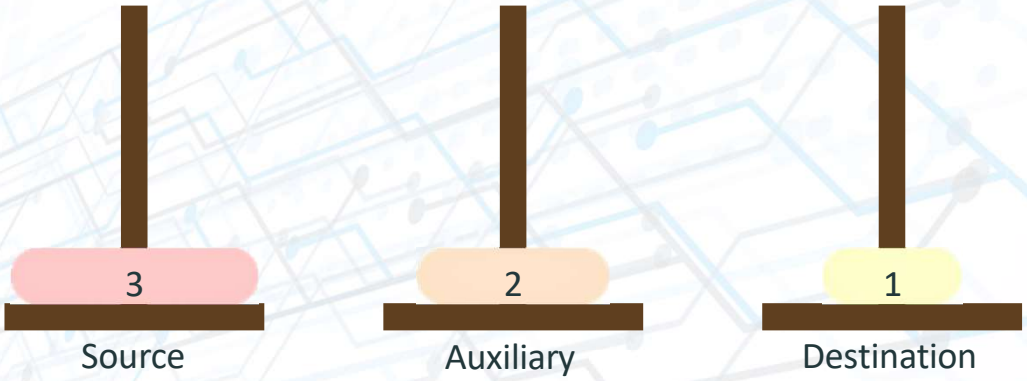
Move disks from Source to Destination using Auxiliary:

1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. Only one auxiliary needle could be used for the intermediate storage of disks.

The Towers of Hanoi

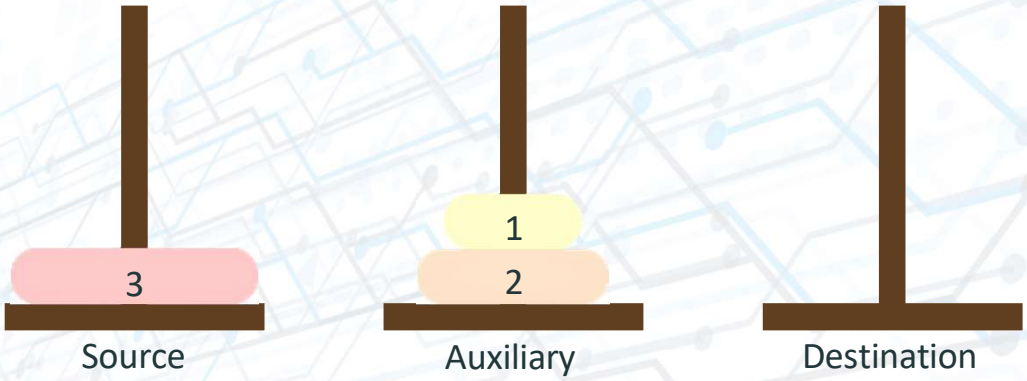
Moved disc from pole 1 to pole 3.

The Towers of Hanoi



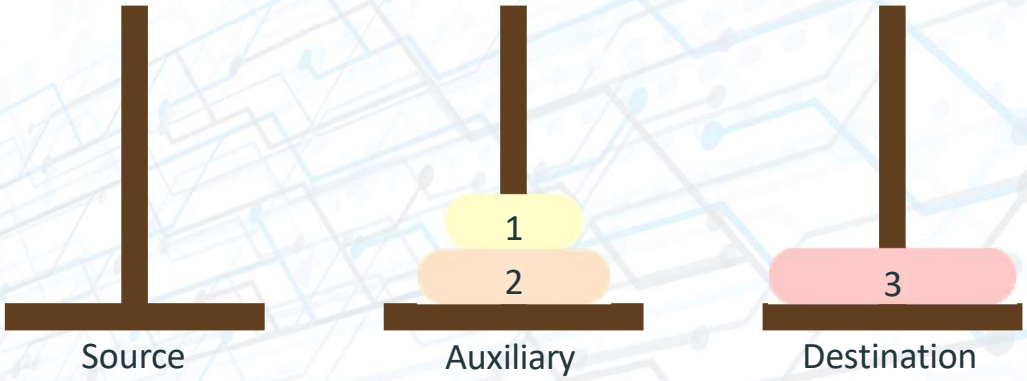
Moved disc from pole 1 to pole 2.

The Towers of Hanoi



Moved disc from pole 3 to pole 2.

The Towers of Hanoi



Moved disc from pole 1 to pole 3.

The Towers of Hanoi

Moved disc from pole 2 to pole 1.

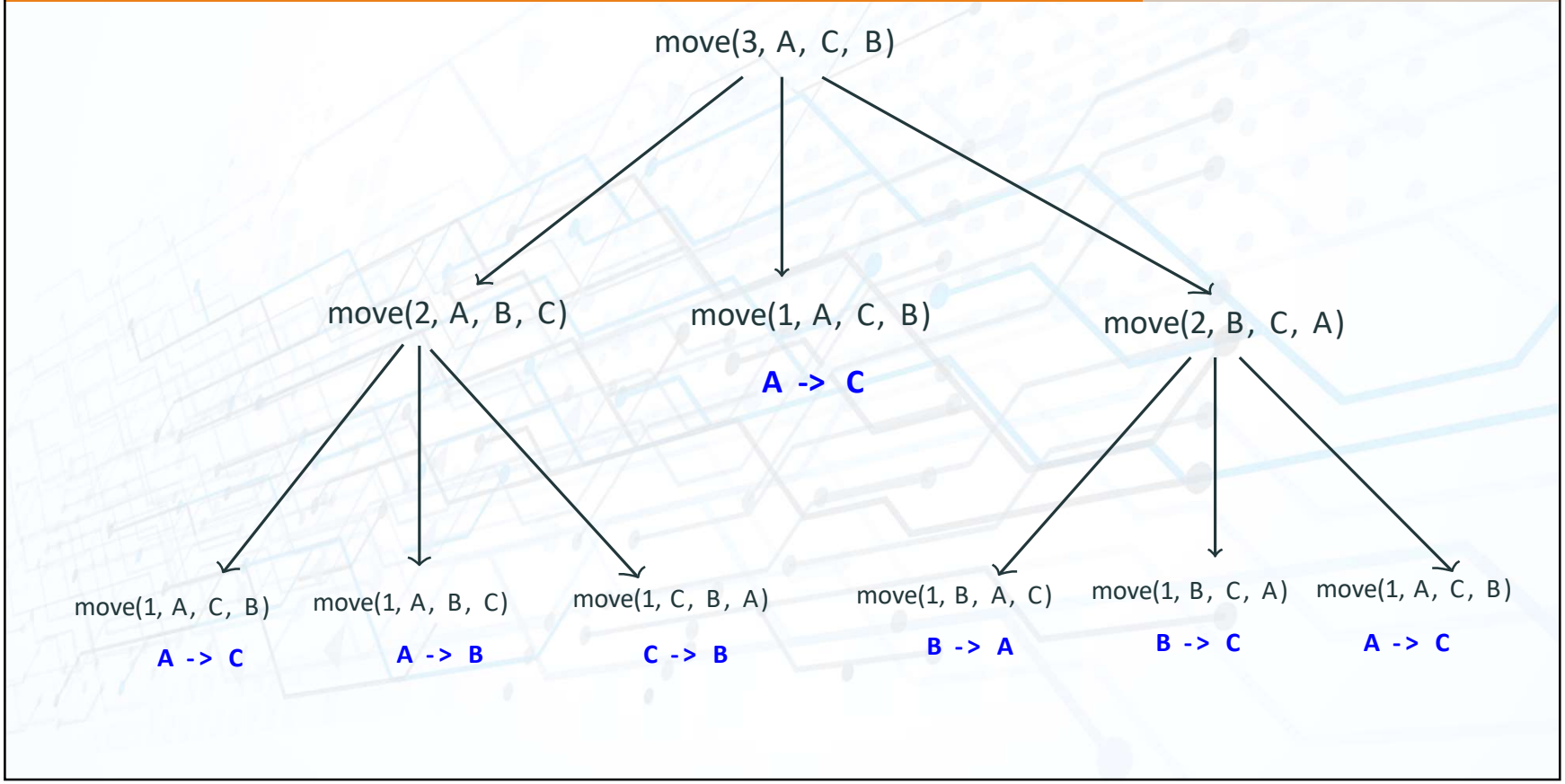
The Towers of Hanoi

Moved disc from pole 2 to pole 3.

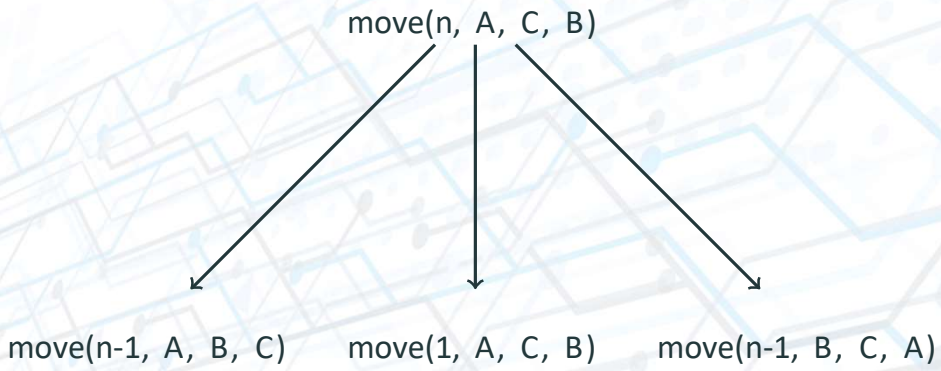
The Towers of Hanoi

Moved disc from pole 1 to pole 3.

The Towers of Hanoi



The Towers of Hanoi : General



The Towers of Hanoi

Algorithm move(val disks <integer>, val source <character>, val destination <character>, val auxiliary <character>)

Move disks from source to destination.

Pre: `disks` is the number of disks to be moved

Post: steps for moves printed

```
print("Towers: ", disks, source, destination, auxiliary)
if disks = 1 then
    print ("Move from", source, "to", destination)
else
    move(disks - 1, source, auxiliary, destination)
    move(1, source, destination, auxiliary)
    move(disks - 1, auxiliary, destination, source)
end
return
```

End move

Recursion implementation in C / C++

Fibonacci Numbers


```
#include <iostream>
using namespace std;

long fib(long num);

int main () {
  int num;
  cout << "What Fibonacci number do you want to calculate?";
  cin >> num;
  cout << "The " << num << " the Fibonacci number is: " << fib(num) << endl;
  return 0;
}

long fib(long num) {
  if (num == 0 || num == 1)
    return num;
  return fib(num-1) + fib(num-2);
}
```

The Towers of Hanoi


```
#include <iostream>
using namespace std;

void move(int n, char source,
          char destination, char auxiliary);

int main () {
  int numDisks;
  cout << "Please enter number of disks: ";
  cin >> numDisks;
  cout << "Start Towers of Hanoi" << endl;
  move(numDisks, 'A', 'C', 'B');
  return 0;
}
```

The Towers of Hanoi

```
void move(int n, char source,
          char destination, charauxiliary){
     static int step = 0;
     if (n == 1)
        cout << "Step u" << ++step << ":uMoveu fromu"
             << source << "utou" << destination << endl;
     else {
        move(n-1, source, auxiliary, destination);
        move(1, source, destination, auxiliary);
        move(n - 1, auxiliary, destination, source);
     }
     return;
}
```

Algorithm Efficiency

Algorithm Efficiency

- A problem often has many algorithms.
- Comparing two different algorithms

$\Rightarrow$ Computational complexity: measure of the difficulty degree (time and/or space) of an algorithm.

- How fast an algorithm is?
- How much memory does it cost?

Algorithm Efficiency

General format

$$\text{efficiency} = f(n)$$

n is the size of a problem (the key number that determines the size of input data)

Linear Loops

```
for (i = 0; i < 1000; i++)
  // application code
```

The number of times the body of the loop is replicated is 1000.

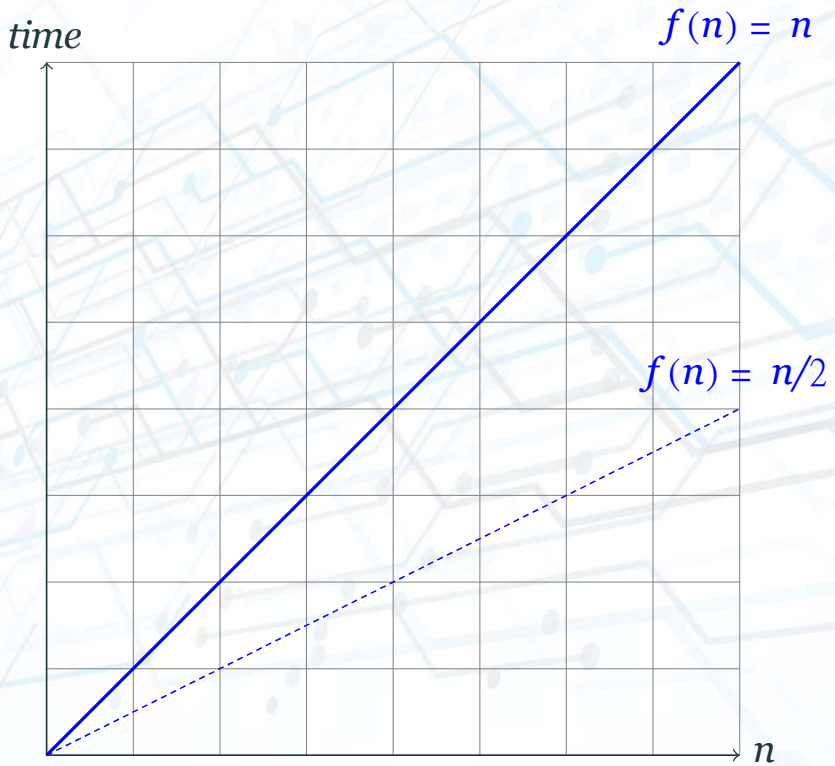
$$f(n) = n$$

```
for (i = 0; i < 1000; i += 2)
  // application code
```

The number of times the body of the loop is replicated is 500.

$$f(n) = n/2$$

Linear Loops



Logarithmic Loops

Multiply loops

```
i = 1
while (i <= n)
    // application code
    i = i x 2
end while
```

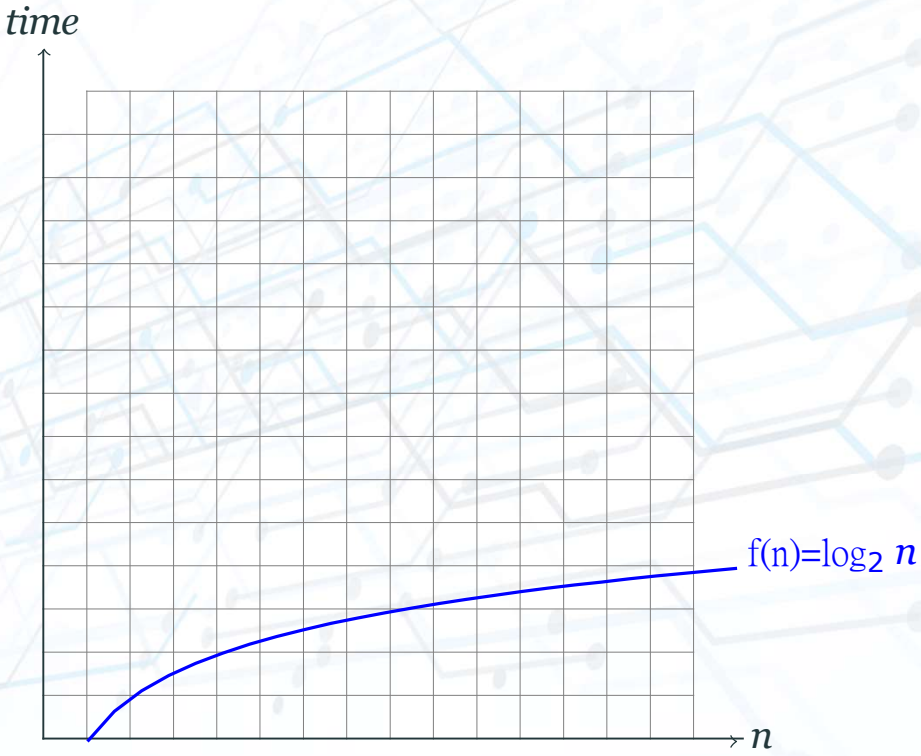
Divide loops

```
i = n
while (i >= 1)
    // application code
    i = i / 2
end while
```

The number of times the body of the loop is replicated is

$$f(n) = \log_2 n$$

Logarithmic Loops



Nested Loops

Iterations = Outer loop iterations × Inner loop iterations

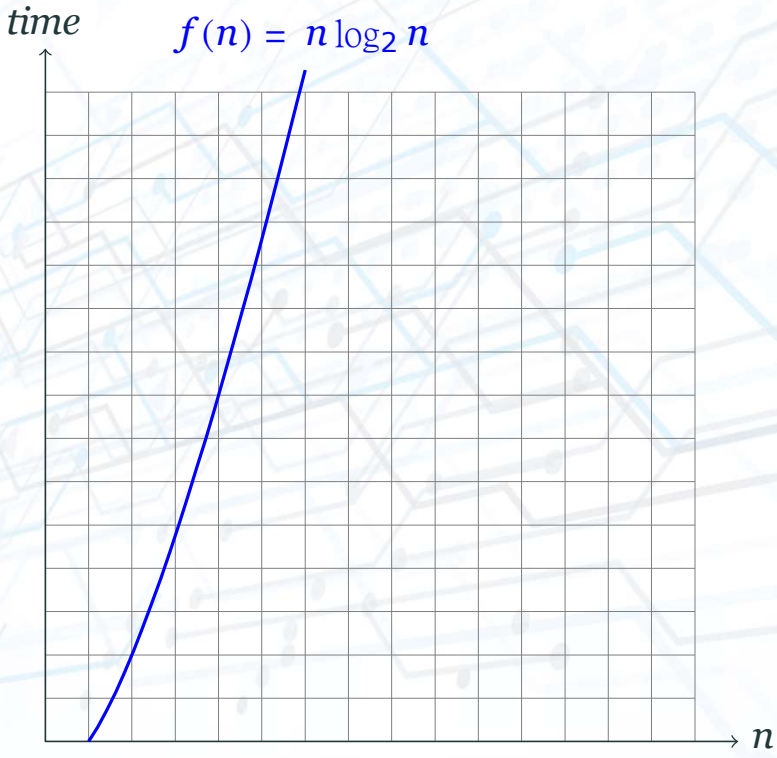
Example

```
i = 1
while (i <= n)
    j = 1
    while (j <= n)
        // application code
        j = j * 2
    end while
    i = i + 1
end while
```

The number of times the body of the loop is replicated is

$$f(n) = n \log_2 n$$

Nested Loops



Quadratic Loops

Example

```
i = 1
while (i <= n)
    j = 1
    while (j <= n)
        // application code
        j = j + 1
    end while
    i = i + 1
end while
```

The number of times the body of the loop is replicated is

$$f(n) = n^2$$

Dependent Quadratic Loops

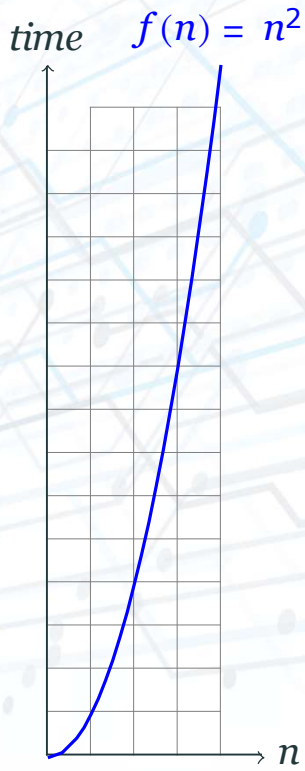
Example

```
i = 1
while (i <= n)
    j = 1
    while (j <= i)
        // application code
        j = j + 1
    end while
    i = i + 1
end while
```

The number of times the body of the loop is replicated is

$$1 + 2 + \ldots + n = n(n + 1)/2$$

Quadratic Loops



Asymptotic Complexity

- Algorithm efficiency is considered with only big problem sizes.
- We are not concerned with an exact measurement of an algorithm's efficiency.
- Terms that do not substantially change the function's magnitude are eliminated.

Big-O notation

Big-O notation

The formal Definition of Big-O

$f \in O(g)$ if and only if $\exists c \in R^+, n_0 \in R$ such that $n \geq n_0, f(n) \leq cg(n)$

We are only interested in the most reduced upper bound. For example, instead of saying that f is in $O(g)$ where $g(x) = 5x^2 + x$, we would say that f is in $O(g)$ where $g(x) = x^2$. Why?

Big-O notation

Example

$$f(n) = c.n \rightarrow f(n) \in O(n)$$

$$f(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \rightarrow f(n) \in O(n^2)$$

- Set the coefficient of the term to one.
- Keep the largest term and discard the others.

Some example of Big-O:

$\log_2 n,$ $n,$ $n \log_2 n,$ $n^2, \ldots$ n^k $\ldots$ $2^n,$ $n!$

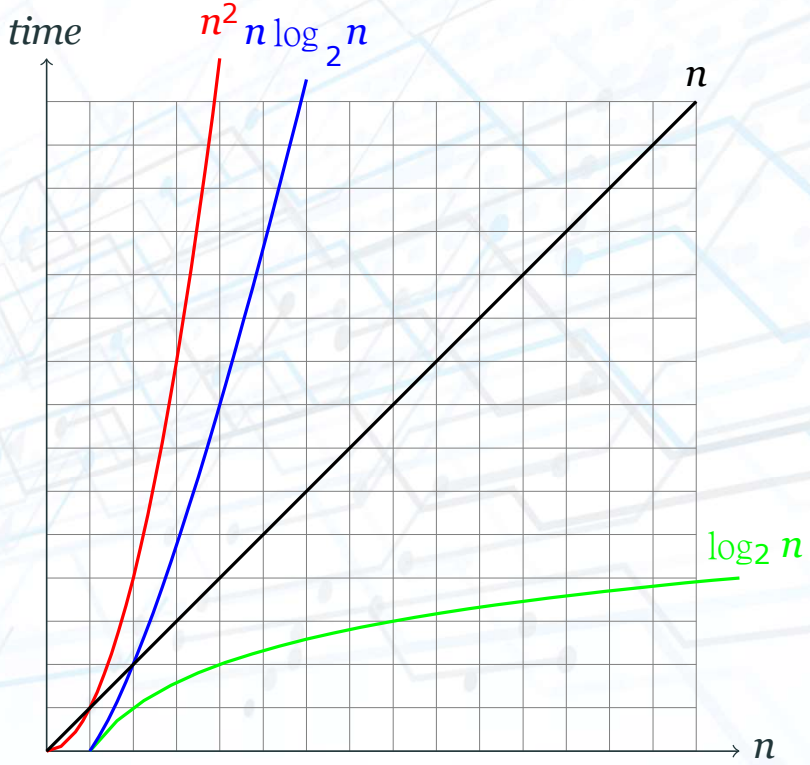
Standard Measures of Efficiency

Efficiency	Big-O	Iterations	Est. Time
logarithmic	$O(\log_2 n)$	14	microseconds
linear	$O(n)$	10 000	0.1 seconds
linear log	$O(n \log_2 n)$	140 000	2 seconds
quadratic	$O(n^2)$	10000^2	15-20 min.
polynomial	$O(n^k)$	10000^k	hours
exponential	$O(2^n)$	2^{10000}	intractable
factorial	$O(n!)$	$10000!$	intractable

Assume instruction speed of 1 microsecond and 10 instructions in loop.

$n = 10000$

Standard Measures of Efficiency



Big-O Analysis Examples

Algorithm addMatrix(val matrix1<matrix>, val matrix2<matrix>, val size<integer>, ref matrix3<matrix>)

Add matrix1 to matrix2 and place results in matrix3

Pre: matrix1 and matrix2 have data size is number of columns and rows in matrix

Post: matrices added – result in matrix3 r=1

```
while r <= size do
    c = 1
    while c <= size do
        matrix3[r, c] = matrix1[r, c] + matrix2[r, c]
        c = c + 1
    end
    r = r + 1
end
return matrix3
End addMatrix
```

Big-O Analysis Examples

Nested linear loop:

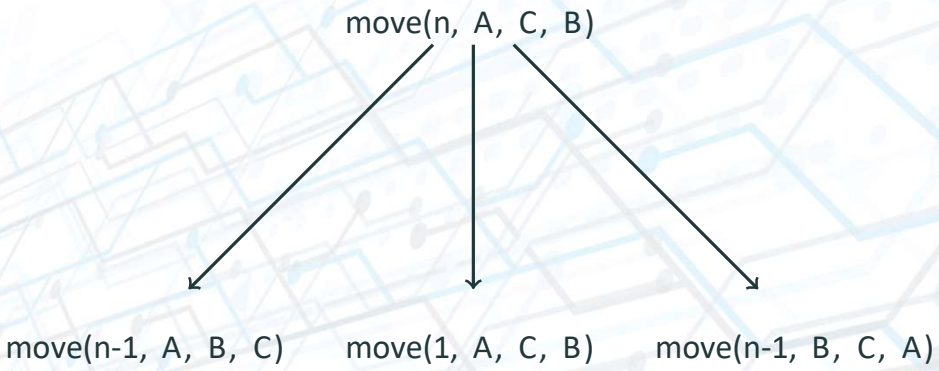
$$f(size) = O(size^2)$$

Time Costing Operations

- The most time consuming: data movement to/from memory/storage.
- Operations under consideration:
  - Comparisons
  - Arithmetic operations
  - Assignments

Problems and common complexities

The Towers of Hanoi : General



Complexity

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

The Towers of Hanoi

Complexity

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

$=> T(n) = 1 + 2 + 2^2 + ... + 2^{n-1}$

$=> T(n) = 2^n - 1$

$=> T(n) = O(2^n)$

- With 64 disks, total number of moves:
  $2^{64} - 1 \approx 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$
- If one move takes 1s, 2^{64} moves take about 5×10^{11} years (500 billions years).

Binary search

Recurrence Equation
An equation or inequality that describes a function in terms of its value on smaller input.

1	2	3	5	8	13	21	34	55	89

$$T(n) = 1 + T(n/2) \Rightarrow T(n) = O(\log_2 n)$$

Binary search

- Best case: when the number of steps is smallest. $T(n) = O(1)$
- Worst case: when the number of steps is largest. $T(n) = O(\log_2 n)$
- Average case: in between. $T(n) = O(\log_2 n)$

Sequential search

8	5	21	2	1	13	4	34	7	18

- Best case: $T(n) = O(1)$
- Worst case: $T(n) = O(n)$
- Average case: $T(n) = \sum_{i=1}^{n} i\, p_i$

  p_i : probability for the target being at a[i]

$$p_i = 1/n \rightarrow T(n) = (\sum_{i=1}^{n} i)/n = O(n(n+1)/2n) = O(n)$$

Quick sort

19	8	3	15	28	10	22	4	12	83

Recurrence Equation

$T(n) = O(n) + 2T(n/2)$

- Best case: $T(n) = O(n \log_2 n)$
- Worst case: $T(n) = O(n^2)$
- Average case: $T(n) = O(n \log_2 n)$