Course introduction, Recursion and Complexity of Algorithms

Data Structures and Algorithms
Monday 8^{th} March, 2021

Dept. Computer Science
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Overview

Basic concepts

What is Data?

(Source: datorama.com)

What is Data?

Data

Data is fact that has been translated into a form that is more convenient to calculate, analyze.

Example

- Numbers, words, measurements, observations or descriptions of things.

- Qualitative data: descriptive information,
- Quantitative data: numerical information (numbers).
  - Discrete data can only take certain values (like whole numbers)
  - Continuous data can take any value (within a range)

Data type

Class of data objects that have the same properties.

Data type

1. A set of values
2. A set of operations on values

Example

Type	Values	Operations
integer	$-\infty, ..., -2, -1, 0, 1, 2, ..., \infty$	$*, +, -, \%, /, ++, --, ...$
floating point	$-\infty, ..., 0.0, ..., \infty$	$*, +, -, /, ...$
character	$\backslash 0$, ..., 'A', 'B', ..., 'a', 'b', ..., $\sim$	$<, >, ...$

Data structure

What is a data structure?

1. A combination of elements in which each is either a data type or another data structure
2. A set of associations or relationships (structure) that holds the data together

Example

An array is a number of elements of the same type in a specific order.

1	2	3	5	8	13	21	34

Abstract data type

The concept of abstraction:

- Users know what a data type can do.
- How it is done is hidden.

Definition

An **abstract data type** is a data declaration packaged together with the operations that are meaningful for the data type.

1. Declaration of data
2. Declaration of operations
3. Encapsulation of data and operations

Abstract data type

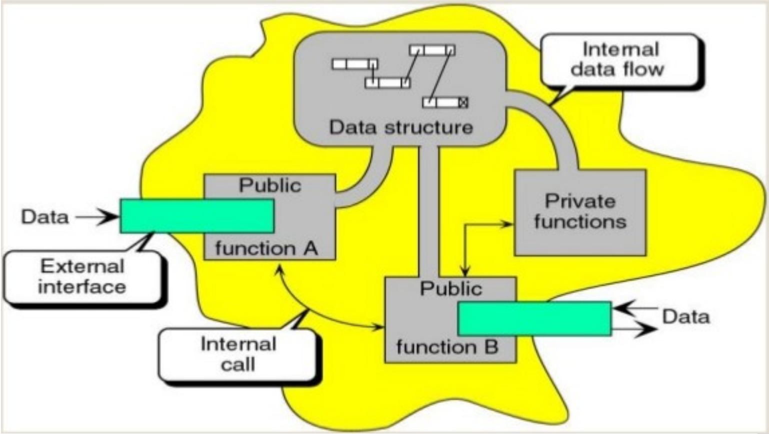


Figure: Abstract data type model (source: Slideshare)

Example: List

Interface

- **Data**: sequence of elements of a particular data type
- **Operations**: accessing, insertion, deletion

Implementation

- Array
- Linked list

Algorithm

What is an algorithm?

The logical steps to solve a problem.

What is a program?

Program = Data structures + Algorithms
(Niklaus Wirth)

Pseudocode

- The most common tool to define algorithms
- English-like representation of the algorithm logic
- Pseudocode = **English** + **code**

relaxed syntax being easy to read

instructions using basic control structures (sequential, conditional, iterative)

Pseudocode

Algorithm Header

- Name
- Parameters and their types
- Purpose: what the algorithm does
- Precondition: precursor requirements for the parameters
- Postcondition: taken action and status of the parameters
- Return condition: returned value

Algorithm Body

- Statements
- Statement numbers: decimal notation to express levels
- Variables: important data
- Algorithm analysis: comments to explain salient points
- Statement constructs: sequence, selection, iteration

Pseudocode: Example

Algorithm average

Pre nothing

Post the average of the input numbers is printed

```
i = 0
sum = 0
while all numbers not read do
    i = i + 1
    read number
    sum = sum + number
end
average = sum / i
print average
End average
```

Algorithm 1: How to calculate the average

Recursion and the basic components of recursive algorithms

Recursion

Definition

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

- Direct : A → A
- Indirect : A → B → A

Example

Factorial

$$Factorial(n) = \left[\begin{array}{ll} 1 & \text{if } n = 0 \\ n \times (n-1) \times ... \times 2 \times 1 & \text{if } n > 0 \end{array} \right.$$

Using recursion:

$$Factorial(n) = \left[\begin{array}{ll} 1 & \text{if } n = 0 \\ n \times Factorial(n-1) & \text{if } n > 0 \end{array} \right.$$

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

$$Factorial(n) = \left[\begin{array}{lll} 1 & \text{if } n = 0 & \text{base} \\ n \times Factorial(n-1) & \text{if } n > 0 & \text{general} \end{array} \right.$$

Recursion

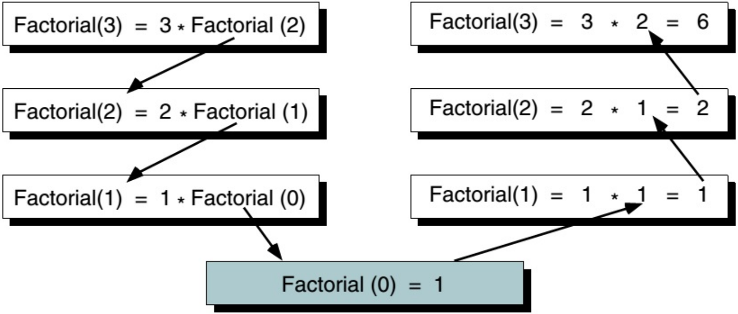


Figure: Factorial (3) Recursively
(Source: Data Structure - A pseudocode Approach with C++

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

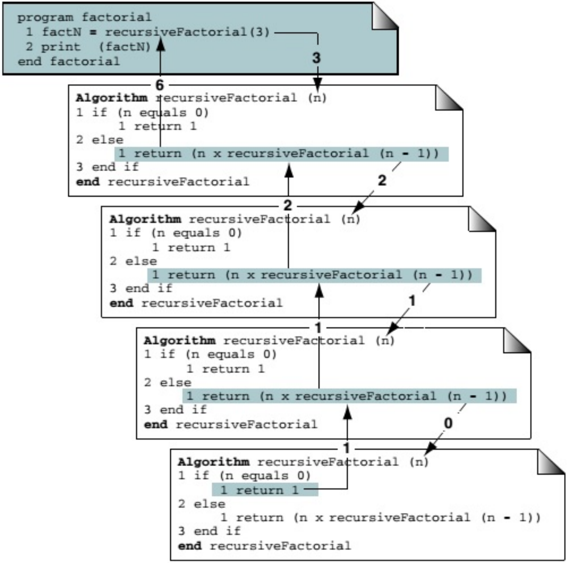


Figure: Calling a Recursive Algorithm (source: Data Structure - A pseudocode Approach with C++)

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.
  $\rightarrow$ 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

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

Example

$\gcd(12, 18) = 6$
$\gcd(5, 20) = 5$

Greatest Common Divisor

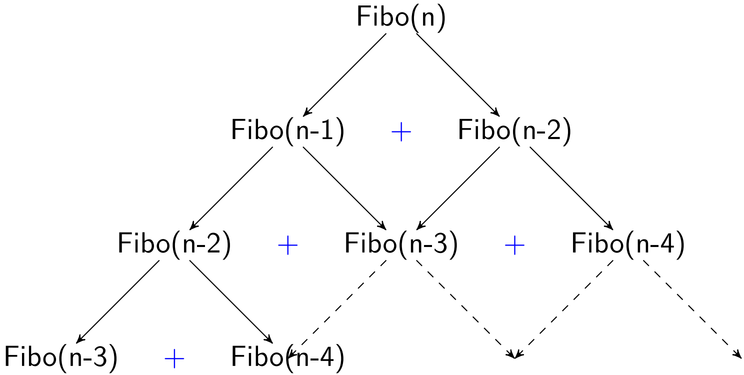
```
Algorithm gcd(a, b)
Calculates greatest common divisor using the Euclidean
  algorithm.
Pre: a and b are integers
Post: greatest common divisor returned

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

Fibonacci Numbers

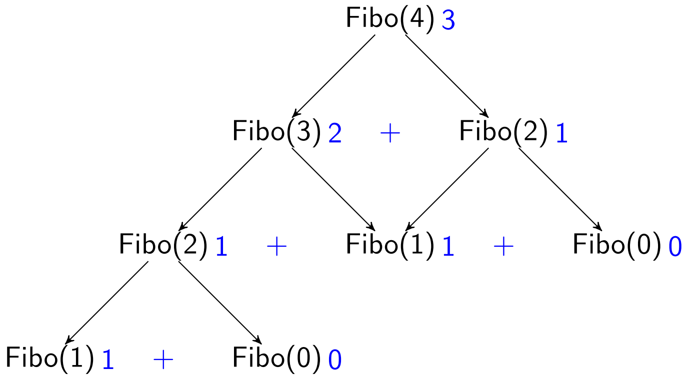
Definition

$$Fibo(n) = \left[\begin{array}{ll} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fibo(n-1) + Fibo(n-2) & \text{otherwise} \end{array} \right.$$

Fibonacci Numbers



Fibonacci Numbers



Result

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci Numbers

```
Algorithm Fibo(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 Fibo(n-1) + Fibo(n-2)
End fib
```

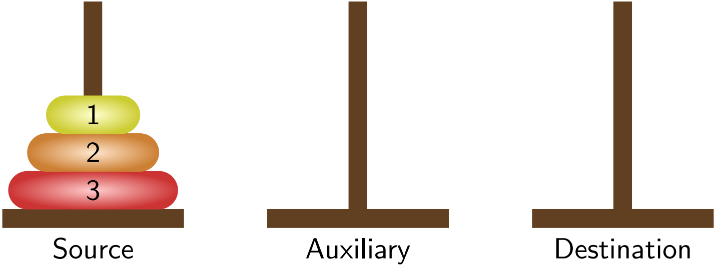
Fibonacci Numbers

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.

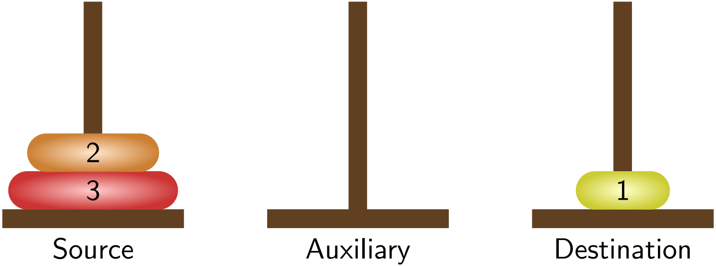
The Towers of Hanoi

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.

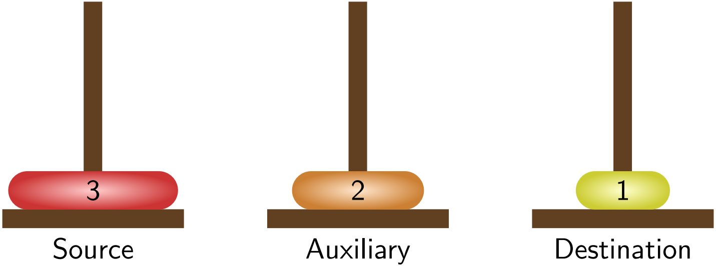


The Towers of Hanoi



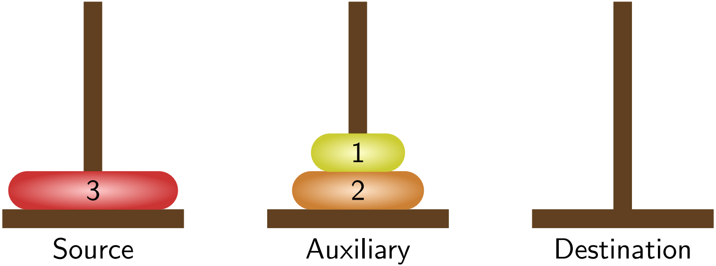
Moved disc from pole 1 to pole 3.

The Towers of Hanoi



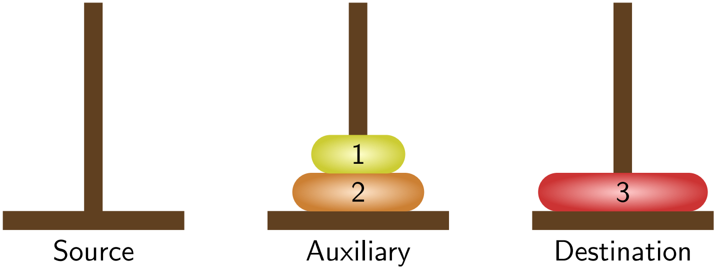
Moved disc from pole 1 to pole 2.

The Towers of Hanoi



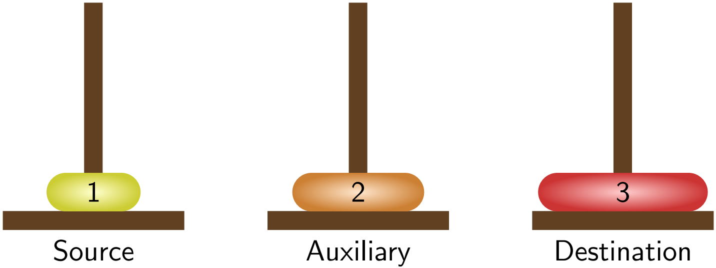
Moved disc from pole 3 to pole 2.

The Towers of Hanoi



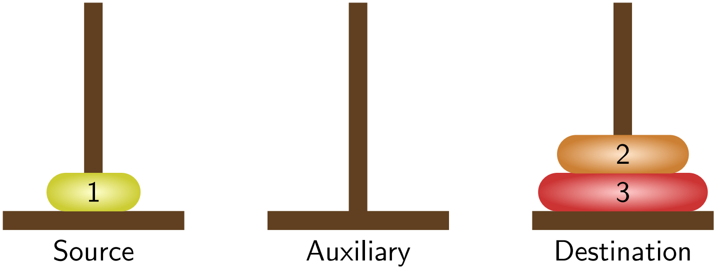
Moved disc from pole 1 to pole 3.

The Towers of Hanoi



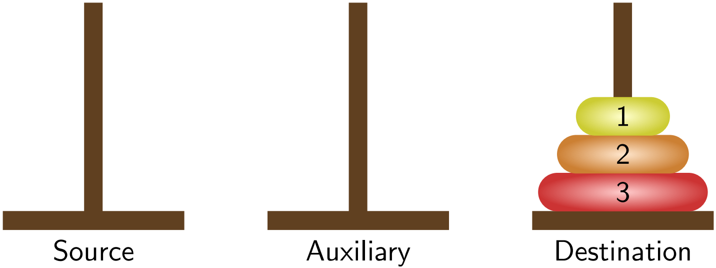
Moved disc from pole 2 to pole 1.

The Towers of Hanoi



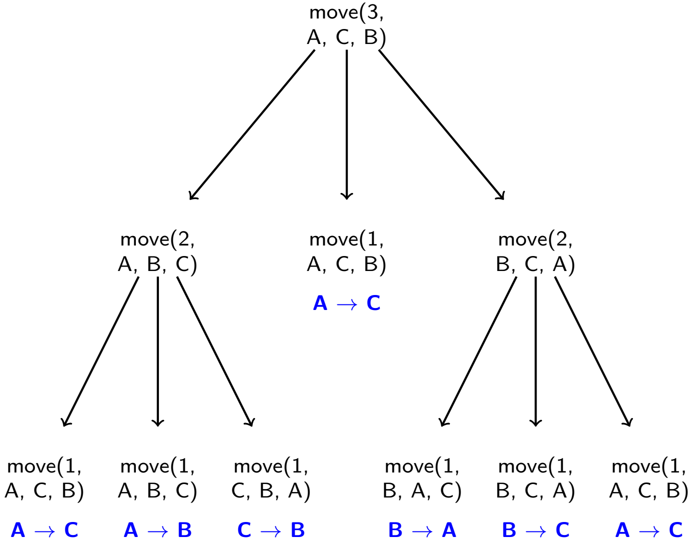
Moved disc from pole 2 to pole 3.

The Towers of Hanoi



Moved disc from pole 1 to pole 3.

The Towers of Hanoi



The Towers of Hanoi

```
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

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

long fib(long num);

int main () {
    int num;
    cout << "n = ";
    cin >> num;
    cout << "fibonacci(" << num << ") = " <<
        << 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 src,
           char dest, char aux){
    if (n == 1)
        cout << "Move from "
             << src << " to "
             << dest << endl;
    else {
        move(n - 1, src, aux, dest);
        move(1, src, dest, aux);
        move(n - 1, aux, dest, src);
    }
    return;
}
```

COMPLEXITY OF ALGORITHMS

Fibonacci numbers: Naive solution

FibRecurs(n)

```
if n ≤ 1 then
    return n
else
    return FibRecurs(n - 1) + FibRecurs(n - 2)
end
```

Algorithm 2: Naive recursive fibonacci solution

Running times

FibRecurs(n)

```
if n ≤ 1 then
    return n
else
    return FibRecurs(n - 1) + FibRecurs(n - 2)
end
```

Let $T(n)$ denote the number of lines of code executed by **FibRecurs(n)**.

- If $n \leq 1$:

$$T(n) =$$

- If $n \geq 2$:

Running times

FibRecurs(n)

```
if n ≤ 1 then
    return n
else
    return FibRecurs(n - 1) + FibRecurs(n - 2)
end
```

Let $T(n)$ denote the number of lines of code executed by **FibRecurs(n)**.

- If $n \leq 1$:

$$T(n) = 2$$

- If $n \geq 2$:

Running times

FibRecurs(n)

```
if n ≤ 1 then
    return n
else
    return FibRecurs(n - 1) + FibRecurs(n - 2)
end
```

Let $T(n)$ denote the number of lines of code executed by **FibRecurs(n)**.

- If $n \leq 1$:

$$T(n) = 2$$

- If $n \geq 2$:

$$T(n) = 3$$

Running times

FibRecurs(n)

```
if n ≤ 1 then
    return n
else
    return FibRecurs(n - 1) + FibRecurs(n - 2)
end
```

Let $T(n)$ denote the number of lines of code executed by **FibRecurs(n)**.

- If $n \leq 1$:

$$T(n) = 2$$

- If $n \geq 2$:

$$T(n) = 3 + T(n - 1)$$

Running times

FibRecurs(n)

```
if n ≤ 1 then
    return n
else
    return FibRecurs(n - 1) + FibRecurs(n - 2)
end
```

Let $T(n)$ denote the number of lines of code executed by **FibRecurs(n)**.

- If $n \le 1$:

$$T(n) = 2$$

- If $n \ge 2$:

$$T(n) = 3 + T(n-1) + T(n-2)$$

Running times

$$T(n) = \begin{cases} 2 & , n \leq 1 \\ 3 + T(n-1) + T(n-2) & , n \geq 2 \end{cases}$$

Running times

$$T(n) = \begin{cases} 2 & , n \leq 1 \\ 3 + T(n-1) + T(n-2) & , n \geq 2 \end{cases}$$

Therefore: $T(n) \geq F_n$

Running times

$$T(n) = \begin{cases} 2 & , n \leq 1 \\ 3 + T(n-1) + T(n-2) & , n \geq 2 \end{cases}$$

Therefore: $T(n) \geq F_n$
For example:
$T(100) \approx 1.77 \times 10^{21}$ takes 56,000 years at 1GHz.

Fibonacci numbers: Efficient Algorithm

Imitate hand computation:
0, 1

Fibonacci numbers: Efficient Algorithm

Imitate hand computation:
0, 1, 1

$0 + 1 = 1$

Fibonacci numbers: Efficient Algorithm

Imitate hand computation:
0, 1, 1, 2

$0 + 1 = 1$
$1 + 1 = 2$

Fibonacci numbers: Efficient Algorithm

Imitate hand computation:
0, 1, 1, 2, 3

$0 + 1 = 1$
$1 + 1 = 2$
$1 + 2 = 3$

Fibonacci numbers: Efficient Algorithm

Imitate hand computation:
0, 1, 1, 2, 3, 5

$0 + 1 = 1$
$1 + 1 = 2$
$1 + 2 = 3$
$2 + 3 = 5$

Fibonacci numbers: Efficient Algorithm

Imitate hand computation:
0, 1, 1, 2, 3, 5, 8

$0 + 1 = 1$
$1 + 1 = 2$
$1 + 2 = 3$
$2 + 3 = 5$
$3 + 5 = 8$

Fibonacci numbers: Efficient Algorithm

FibList(n)

```
Create an array with length n, F[0...n]
F[0] ← 0
F[1] ← 1
for i ← 2 to n 1 do
    F[i] ← F[i − 1] + F[i − 2]
end
return F[n]
```

Algorithm 5: Efficient algorithm for Fibonacci numbers

Fibonacci numbers: Efficient Algorithm

FibList(n)

```
Create an array with length n + 1, F[0...n]
F[0] ← 0
F[1] ← 1
for i ← 2 to n 1 do
    F[i] ← F[i − 1] + F[i − 2]
end
return F[n]
```

- $T(n) = 2n + 2$. So $T(100) = 202$.
- Easy to compute.

Fibonacci numbers: Efficient Algorithm

FibList(n)

```
Create an array with length n + 1, F[0...n]
F[0] ← 0
F[1] ← 1
for i ← 2 to n 1 do
    F[i] ← F[i − 1] + F[i − 2]
end
return F[n]
```

- $T(n) = 2n + 2$. So $T(100) = 202$.
- Easy to compute.

Moral: **The right algorithm makes all the difference.**

Runtime Analysis

FibList(n)

```
Create an array with length n + 1, F[0...n]
F[0] ← 0
F[1] ← 1
for i ← 2 to n by 1 do
    F[i] ← F[i − 1] + F[i − 2]
end
return F[n]
```

Runtime Analysis

FibList(n)

Create an array with length $n+1$, F[0. . .n]
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **by** 1 **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
end
return $F[n]$

$2n+2$ lines of code. Does this really describe the runtime of the algorithm?

Runtime Analysis

FibList(n)

```
1 Create an array with length n + 1, F[0...n]
2 F[0] ← 0
3 F[1] ← 1
4 for i ← 2 to n by 1 do
5 |   F[i] ← F[i − 1] + F[i − 2]
6 end
7 return F[n]
```

Line 1: Depends on memory management system.

Runtime Analysis

FibList(n)

```
1 Create an array with length n + 1, F[0...n]
2 F[0] ← 0
3 F[1] ← 1
4 for i ← 2 to n by 1 do
5     F[i] ← F[i − 1] + F[i − 2]
6 end
7 return F[n]
```

Line 2, 3: Assignment.

Runtime Analysis

FibList(n)

```
1 Create an array with length n + 1, F[0...n]
2 F[0] ← 0
3 F[1] ← 1
4 for i ← 2 to n by 1 do
5 |   F[i] ← F[i − 1] + F[i − 2]
6 end
7 return F[n]
```

Line 4: Increment, comparison, branch.

Runtime Analysis

FibList(n)

```
1 Create an array with length n + 1, F[0...n]
2 F[0] ← 0
3 F[1] ← 1
4 for i ← 2 to n by 1 do
5 |   F[i] ← F[i − 1] + F[i − 2]
6 end
7 return F[n]
```

Line 5: Lookup, assignment, addition of big integers.

Runtime Analysis

FibList(n)

```
1 Create an array with length n + 1, F[0...n]
2 F[0] ← 0
3 F[1] ← 1
4 for i ← 2 to n by 1 do
5 |   F[i] ← F[i − 1] + F[i − 2]
6 end
7 return F[n]
```

Line 7: Loopup, return

Computing Runtime

To figure out how long this simple program would actually take to run on a real computer, we would also need to know things like:

1. Speed of the Computer

Computing Runtime

To figure out how long this simple program would actually take to run on a real computer, we would also need to know things like:

1. Speed of the Computer
2. The System Architecture

Computing Runtime

To figure out how long this simple program would actually take to run on a real computer, we would also need to know things like:

1. Speed of the Computer
2. The System Architecture
3. The Compiler Being Used

Computing Runtime

To figure out how long this simple program would actually take to run on a real computer, we would also need to know things like:

1. Speed of the Computer
2. The System Architecture
3. The Compiler Being Used
4. Details of the Memory Hierarchy

Computing Runtime: Problem

- Figuring out accurate runtime is a huge mess.
- In practice, you might not even know some of these details.

Computing Runtime: Goals

Want to:

- Measure runtime without knowing these details.
- Get results that work for large inputs.

ASYMPTOTIC NOTATION

Idea, Problem and Solution

Idea

All of these issues can multiply runtimes by (large) constant.

Idea, Problem and Solution

Idea

All of these issues can multiply runtimes by (large) constant.
So measure runtime in away that ignores constant multiples.

Idea, Problem and Solution

Idea

All of these issues can multiply runtimes by (large) constant.
So measure runtime in away that ignores constant multiples.

Problem and Solution

Unfortunately, 1 second, 1 hour, 1 year only differ by constant multiples.

Idea, Problem and Solution

Idea

All of these issues can multiply runtimes by (large) constant.
So measure runtime in away that ignores constant multiples.

Problem and Solution

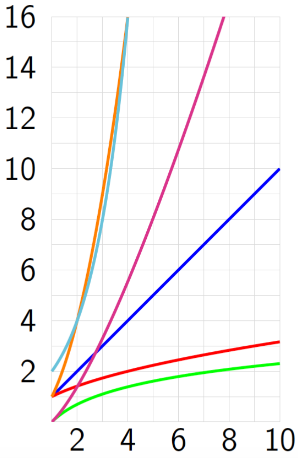
Unfortunately, 1 second, 1 hour, 1 year only differ by constant multiples.
$\Rightarrow$ Consider ASYMPTOTIC RUNTIMES. How does runtime scale with input size.

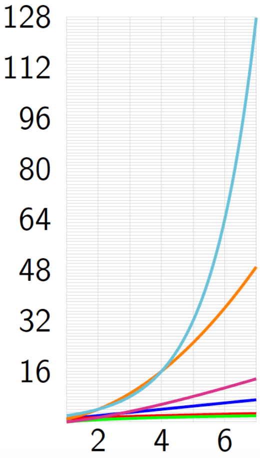
Approximate Runtimes

	n	$n \log n$	n^2	2^n
$n = 20$	1 sec	1 sec	1 sec	1 sec
$n = 50$	1 sec	1 sec	1 sec	13 day
$n = 10^2$	1 sec	1 sec	1 sec	$4 \cdot 10^{13}$ year
$n = 10^6$	1 sec	1 sec	17 min	
$n = 10^9$	1 sec	30 sec	30 year	
max n	10^9	$10^{7.5}$	$10^{4.5}$	30

Approximate Runtimes

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$





BIG-O NOTATION

Big-Oh notation

Definition

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ or $f \preceq g$ if there are positive constants c and n_0 such that:

$$f(n) \leq c.g(n), \forall n \geq n_0$$

.

Big-Oh notation

Definition

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ or $f \preceq g$ if there are positive constants c and n_0 such that:

$$f(n) \leq c.g(n), \forall n \geq n_0$$

.

f is bounded above by some constant multiple of g.

Big-Oh notation

Definition

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ or $f \preceq g$ if there are positive constants c and n_0 such that:

$$f(n) \leq c.g(n), \forall n \geq n_0$$

.

Example

$3n^2 + 5n + 2 = O(n^2)$ since if $n \geq 1$,

Big-Oh notation

Definition

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ or $f \preceq g$ if there are positive constants c and n_0 such that:

$$f(n) \leq c.g(n), \forall n \geq n_0$$

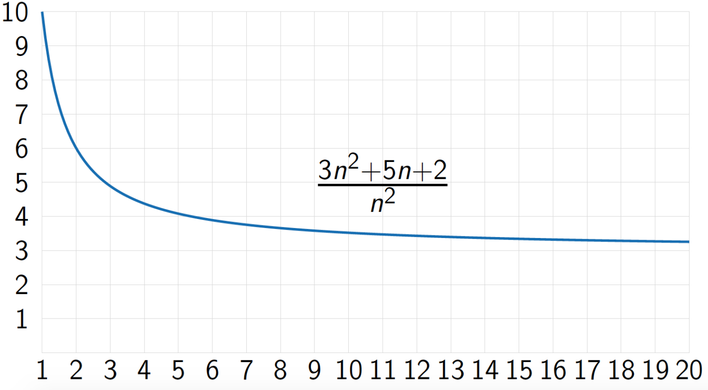
.

Example

$3n^2 + 5n + 2 = O(n^2)$ since if $n \geq 1$,
$3n^2 + 5n + 2 \leq 3n^2 + 5n^2 + 2n^2 = 10n^2$

Growth Rate

$3n2 + 5n + 2$ has the same growth rate as n^2.



USING BIG-O

Using Big-O

We will use Big-O notation to report algorithm runtimes. This has several advantages:

- Clarifies Growth Rate
- Cleans up Notation $\Rightarrow$ Makes algebra easier.
- Can ignore complicated details.

Using Big-O: Warning

- Using Big-O loses important information about constant multiples.
- Big-O is only asymptotic.

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3), \frac{n^2}{3} = O(n^2)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3), \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2), \sqrt{n} = O(n)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3), \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2), \sqrt{n} = O(n)$$

$n^a \prec b^n$ for $a > 0, b > 1$:

$$n^5 = O(\sqrt{2}^n), \sqrt{n^{100}} = O(1.1^n)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3), \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2), \sqrt{n} = O(n)$$

$n^a \prec b^n$ for $a > 0, b > 1$:

$$n^5 = O(\sqrt{2}^n), \sqrt{n^{100}} = O(1.1^n)$$

$(\log n)^a \prec n^b$ for $a, b > 0$:

$$(\log n)^3 = O(\sqrt{n}), n \log n = O(n^2)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3), \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2), \sqrt{n} = O(n)$$

$n^a \prec b^n$ for $a > 0, b > 1$:

$$n^5 = O(\sqrt{2}^n), \sqrt{n^{100}} = O(1.1^n)$$

$(\log n)^a \prec n^b$ for $a, b > 0$:

$$(\log n)^3 = O(\sqrt{n}), n \log n = O(n^2)$$

Smaller terms can be omitted:

$$n^2 + n = O(n^2), 2^n + n^9 = O(n2^)$$

Big-O in Practice: Recall FibList

Operation	Runtime

Big-O in Practice: Recall FibList

Operation	Runtime
Create an array $F[0 \ldots n]$	$O(n)$

Big-O in Practice: Recall FibList

Operation	**Runtime**
Create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$

Big-O in Practice: Recall FibList

Operation	Runtime
Create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$

Big-O in Practice: Recall FibList

Operation	Runtime
Create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times

Big-O in Practice: Recall FibList

Operation	Runtime
Create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times
$F[i] \leftarrow F[i-1] + F[i-2]$	$O(n)$

Big-O in Practice: Recall FibList

Operation	Runtime
Create an array $F[0 \dots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times
$F[i] \leftarrow F[i-1] + F[i-2]$	$O(n)$
return $F[n]$	$O(1)$

Big-O in Practice: Recall FibList

Operation	**Runtime**
Create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times
$F[i] \leftarrow F[i-1] + F[i-2]$	$O(n)$
return $F[n]$	$O(1)$

Total:

$$O(n) + O(1) + O(1) + O(n) \times O(n) + O(1) = O(n^2)$$

Other notations

Definition

For function $f, g : \mathbb{N} \to \mathbb{R}^+$, we say that:

- $f(n) = \Omega(g(n))$ or $f \succeq g$ if for some c, $f(n) \geq c \times g(n)$ (f grows no slower than g).
- $f(n) = \Theta(g(n))$ or $f \asymp g$ if $f = O(g)$ and $f = \Omega(g)$ (f grows at the same rate as g).
- $f(n) = o(g(n))$ or $f \prec g$ if $\dfrac{f(n)}{g(n)} \to 0$ as $n \to \infty$ (f grows slower than g).

Asymptotic Notation

- Lets us ignore messy details in analysis.
- Produces clean answers.
- Throws away a lot of practically useful information.

P and NP Problems

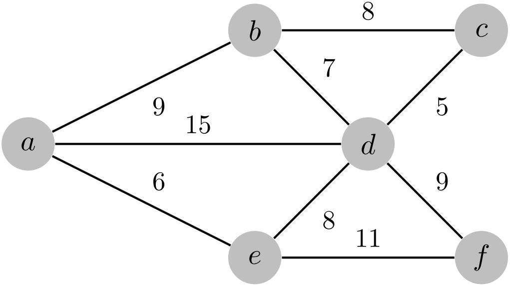
P and NP Problems

- P: Polynomial (can be solved in polynomial time on a deterministic machine).
- NP: Nondeterministic Polynomial (can be solved in polynomial time on a nondeterministic machine).

P and NP Problems

Travelling Salesman Problem:

A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list.
Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.



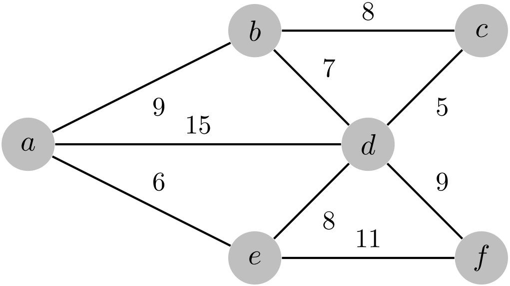
P and NP Problems

Travelling Salesman Problem:

Deterministic machine:

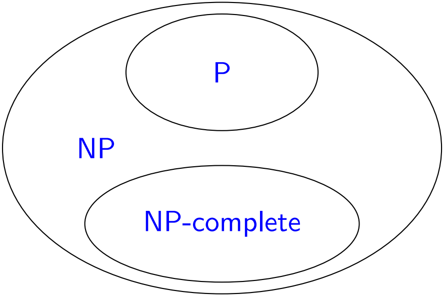
$f(n) = n(n-1)(n-2)\dots 1 = O(n!)$

NP problem



P and NP Problems

NP-complete: NP and every other problem in NP is polynomially reducible to it.



$$P = NP?$$

THANK YOU.