Algorithm Design and Analysis

Day 1
Algorithms and Their Analysis

2015, AUT - CJLU

Prequel

Course Organization

- **Instructor**: Jiamou Liu
  Email: `jiamou.liu@aut.ac.nz`
- **Textbook**: Algorithms (Dasgupta,Papadimitrious,Vazirani; McGraw Hill)
  Download: `http://beust.com/algorithms.pdf`
- **Other Resources**:
  Introduction to Algorithms (Cohen,Leiserson,Rivest,Stein; The MIT Press)
  Algorithms (4th Edition) (Sedgewick,Wayne; Addison-Wesley)

Course Organization

Assessments

- Tutorials/Labs: 20%
- Assignment: 20% (Pair Work)
- Written Mid Semester Test: 10% (Week 2)
- Written Examination: 50%

Course Content

Themes

Data structures:

- Trees
- Graphs

Graph algorithms:

- Traversals
- Distance
- Centrality
- Flow

Algorithm design techniques:

- Divide-and-Conquer
- Greedy algorithms
- Dynamic programming

Overview of Algorithms

What is an algorithm?

Overview of Algorithms

What is an algorithm?

- A way to compute
  e.g. Adding 534561 and 135362

Overview of Algorithms

What is an algorithm?

- A way to compute
  e.g. Adding 534561 and 135362
- A sequence of mechanical operations
  e.g. Store value 1 to variable a.
  Store value 2 to variable b.
  Add a and b, then store the result in c.

Overview of Algorithms

What is an algorithm?

- A way to compute

  e.g. Adding 534561 and 135362

- A sequence of mechanical operations

  e.g. Store value 1 to variable a.

  Store value 2 to variable b.

  Add a and b, then store the result in c.

- A computerised program

  e.g.

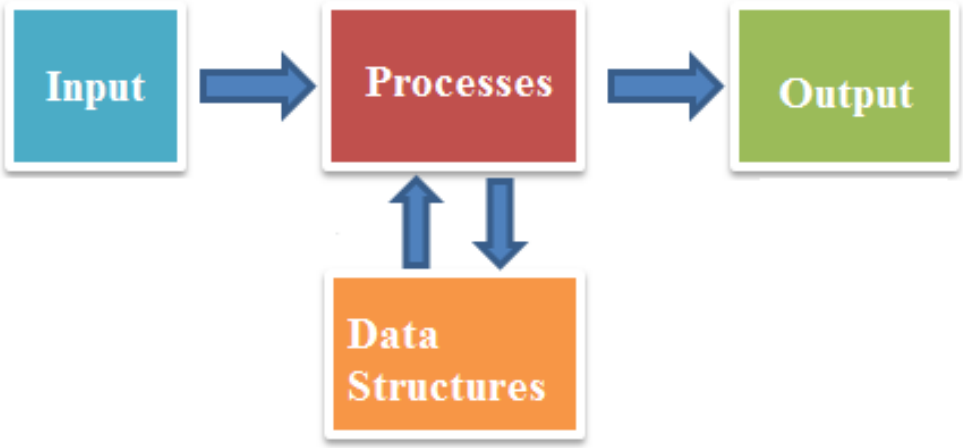
```
for(int i=0; i<5;i++){
    total = total + i;
}
```

Describing Algorithms

Question: How do we describe an algorithm?

We needs to specify:

- Input
- (May need) Auxiliary data structures
- Process
- Output



Languages of description: natural language, pseudocode, computer languages

Example 1: Maximal Number

Problem: Find the largest number in an integer array.

Description 1: Java

```
 public static int findMax(int[] arr){
    if(arr.length==0) return -1;
    int max=arr[0];
    for(int i=1;i<arr.length; i++){
        if (max<arr[i])
            max = arr[i];
    }
    return max;
}
```

Example 1: Maximal Number

Description 2: Pseudocode

Algorithm `findMax`:

INPUT: integer array *arr* of length n
OUTPUT: integer *max* that denotes the maximum number of *arr*

1. if $n = 0$, return -1
2. $max \leftarrow arr[0]$
3. for each $i = 1..n$ do
   if $max < arr[i]$ then
   $max \leftarrow arr[i]$
4. return *max*

Note: "$\leftarrow$" is a symbol for variable assignment

Example 1: Maximal Number

Description 3: Natural Language

Given, as input an array of integers, the algorithm performs the following operations:

1. If there are no numbers in the set then there is no highest number and return -1.
2. Assume the first number in the set is the largest number in the set.
3. For each remaining number in the set:

   if this number is larger than the current largest number, set this number to be the new largest number.
4. When there are no numbers left in the set to iterate over, consider the current largest number to be the largest number of the set; return it as the output of the algorithm.

Example 2: Euclid's Algorithm (300BC)

Problem: Find the greatest common divisor (gcd) of two positive integer a, b.

Description 1: Java

```
public static long euclidGCD(long a,long b){
    long c=0;
    if(a==b && b==0) return 1;
    if(a<b){
        c=b;b=a;a=c;
    }
    while(b!=0){
        c=a%b;
        a=b;b=c;
    }
    return a;
}
```

Example 2: Euclid's Algorithm (300BC)

Description 2: Pseudocode

Algorithm euclidGCD:

INPUT: Two positive integers a, b
OUTPUT: An integer denoting the gcd of a, b

while $b \neq 0$
    $t \leftarrow b$
    $b \leftarrow a \mod b$
    $a \leftarrow t$
return a

Example 2: Euclid's Algorithm (300BC)

Description 3: Natural Language

Given two positive integers a, b as input, the algorithm Repeats the following steps until b becomes 0:

- Each step begins with two nonnegative numbers a, b.
- We also assume as loop invariant that $a \geq b$.
- Let c be the remainder of $a \div b$.
  (Hence $c < b$)
- Assign the value of b to a, and then assign the value of c to b.
  (Note that this preserves the loop invariant of $a \geq b$.)

Return a as the output

Analysis of Algorithms

Main Questions

Given an algorithm

- Correctness: Is the algorithm correct?
  - Mathematics: Construct a mathematical proof for correctness.
  - Experimentation: Implement the algorithm using a program, and verify its correctness on different test cases.
- Complexity: How efficient is the algorithm?
  - Mathematics: Theoretically analyse the computational complexity based on a computation model
  - Experimentation: Implement the algorithm using a program, and measure the running time on different input size

Required Background

- Mathematical Background:
  - Algebra
  - Basic discrete mathematics
  - Proofs (esp. induction)
- Programming Background
- Data Structures: Arrays, Linked lists, Stacks, Queues, Trees, Priority Queues, Graphs

Part I: Recursion

Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

An Instructional Approach

```
public static int sum1(int n)
{
    int result = 0;
    for (int i = 1; i <= n; i++)
    {
        result = result + i;
    }
    return result;
}
```

Case Study: Summing Integers

Task. Compute the sum of the numbers

$$sum(n) = \sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

Strategy. Reduce the problem to a simpler problem.

Case Study: Summing Integers

Task. Compute the sum of the numbers

$$sum(n) = \sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

Strategy. Reduce the problem to a simpler problem.

To compute $sum(n)$:

- If $n = 0$, then the result is simply 0.
- If $n > 0$, then we first find $sum(n-1)$, then the result is:

$$sum(n-1) + n.$$

Case Study: Summing Integers

Task. Compute the sum of the numbers

$$sum(n) = \sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

Strategy. Reduce the problem to a simpler problem.

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

Case Study: Summing Integers

Task. Compute the sum of the numbers

$$sum(n) = \sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

e.g. To compute $sum(3)$:

$$\begin{aligned} sum(3) &= sum(2) + 3 \\ &= sum(1) + 2 + 3 \\ &= sum(0) + 1 + 2 + 3 \\ &= 0 + 1 + 2 + 3 + 4 \\ &= 1 + 2 + 3 + 4 \\ &= 3 + 3 + 4 \\ &= 6 + 4 \\ &= 10 \end{aligned}$$

Case Study: Summing Integers

Task. Compute the sum of the numbers

$$sum(n) = \sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

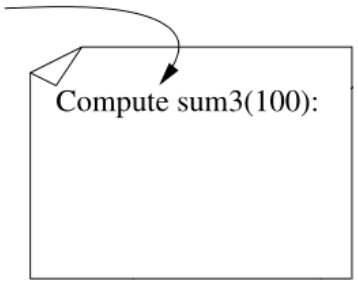
```
public static int sum2(int n)
// pre: n >= 0
// post: compute the sum of 0..n
{
    if (n < 1) return 0;
    else return sum2(n-1) + n;
}
```

Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

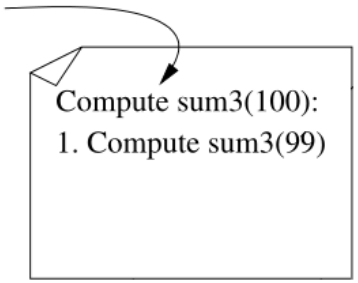


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

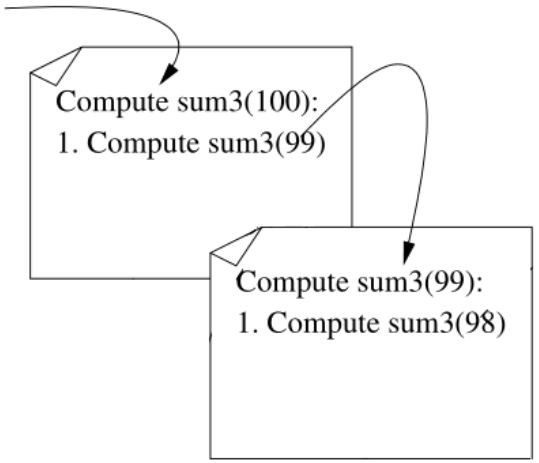


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

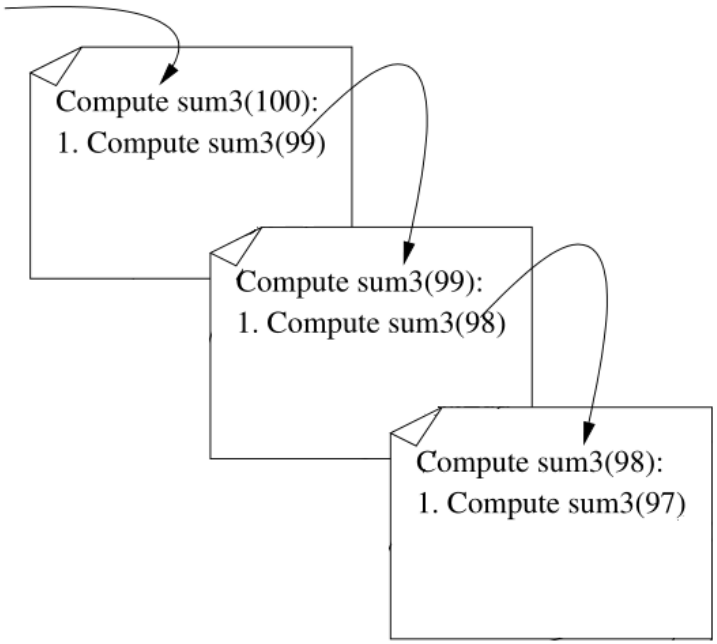


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

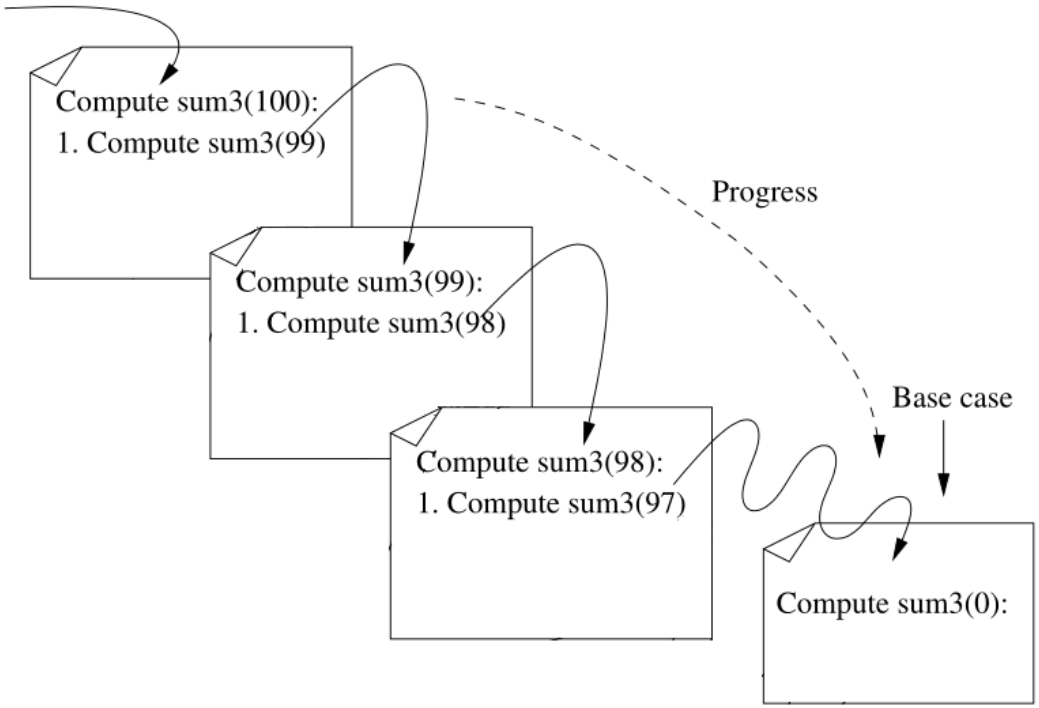


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

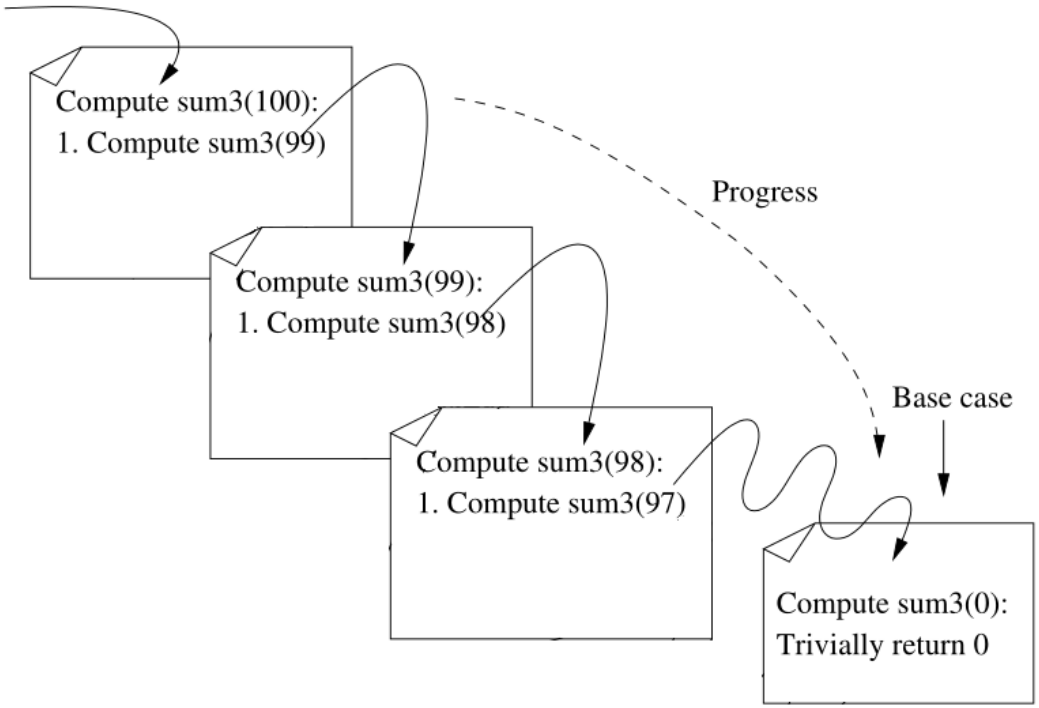


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

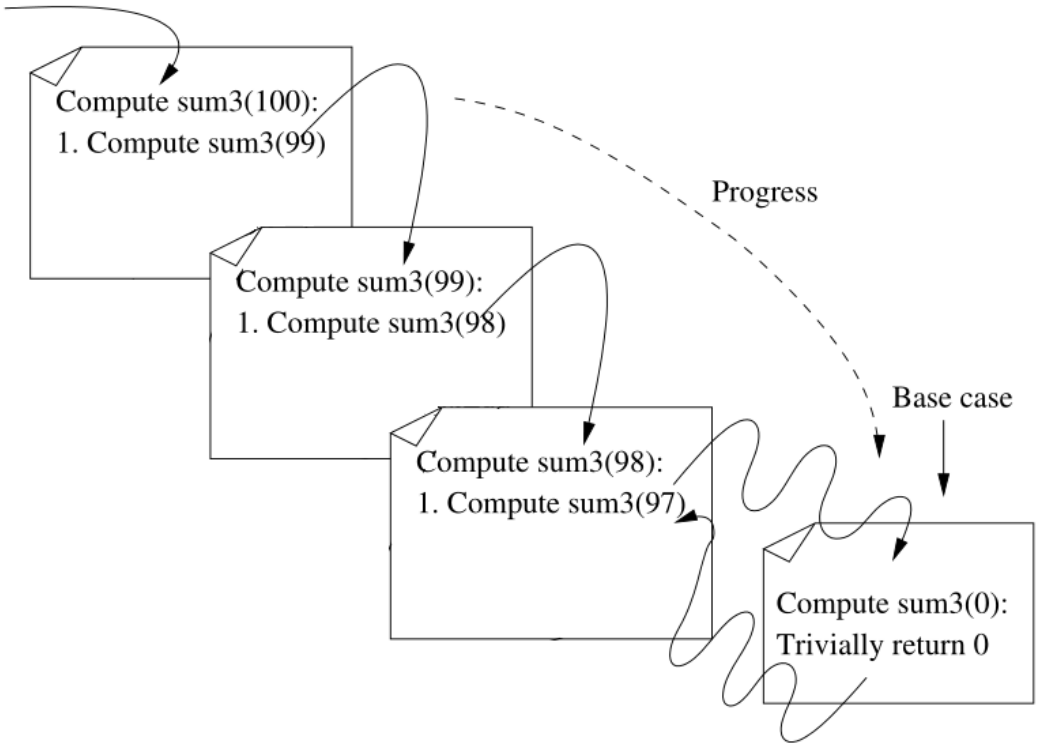


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

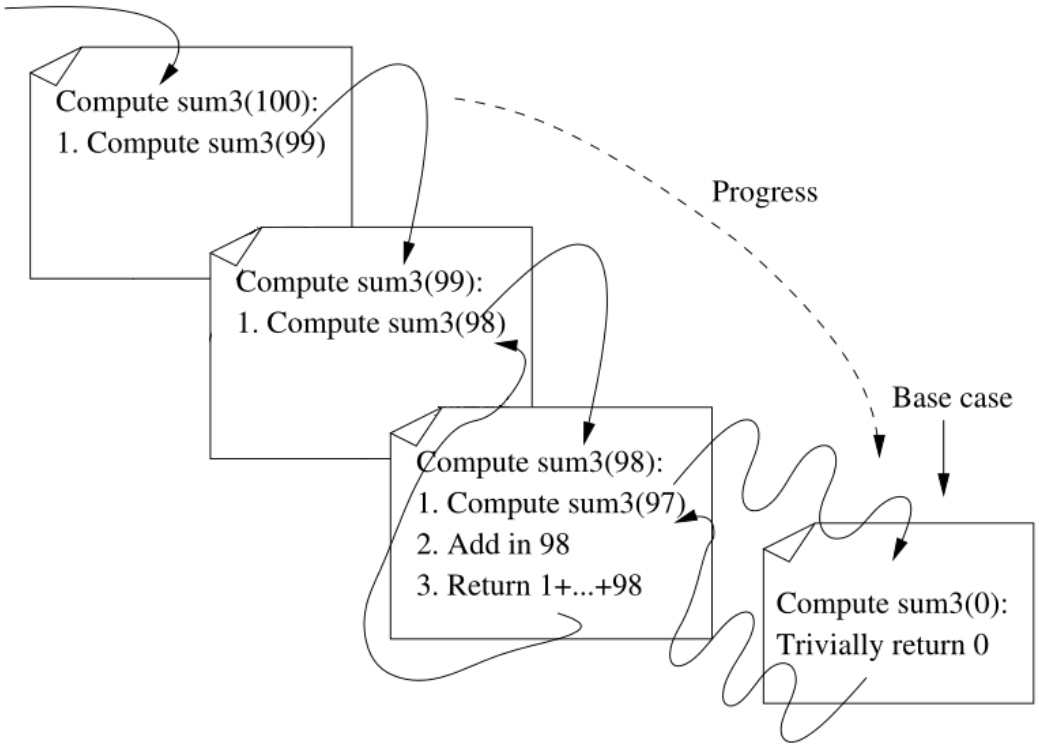


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

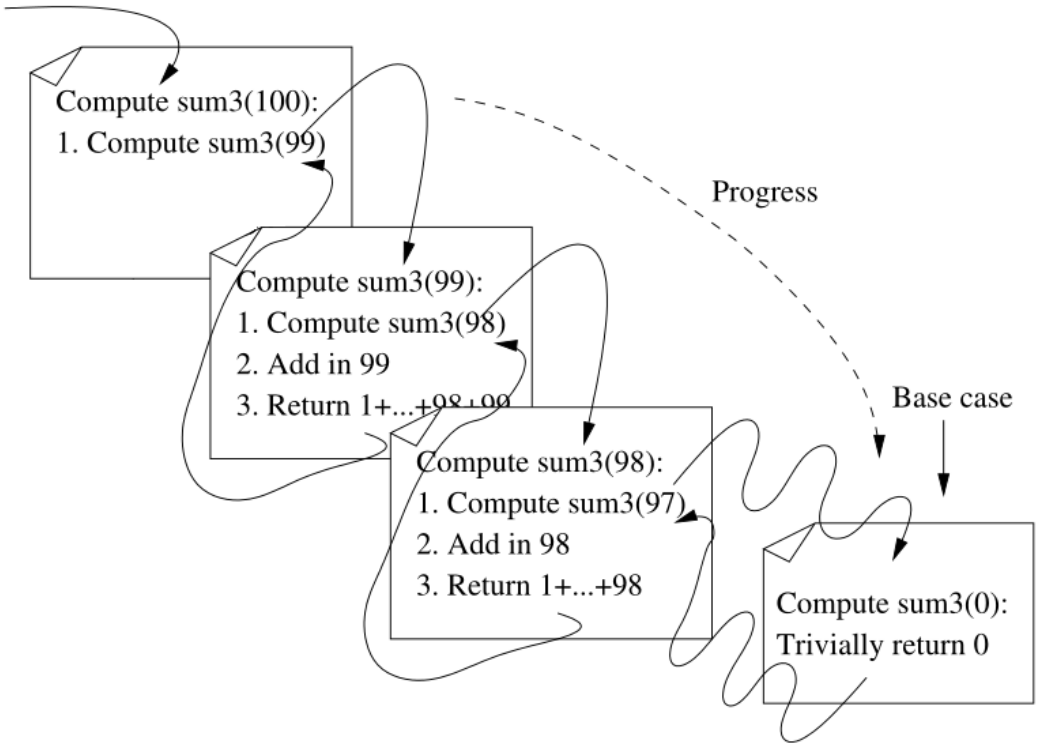


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach

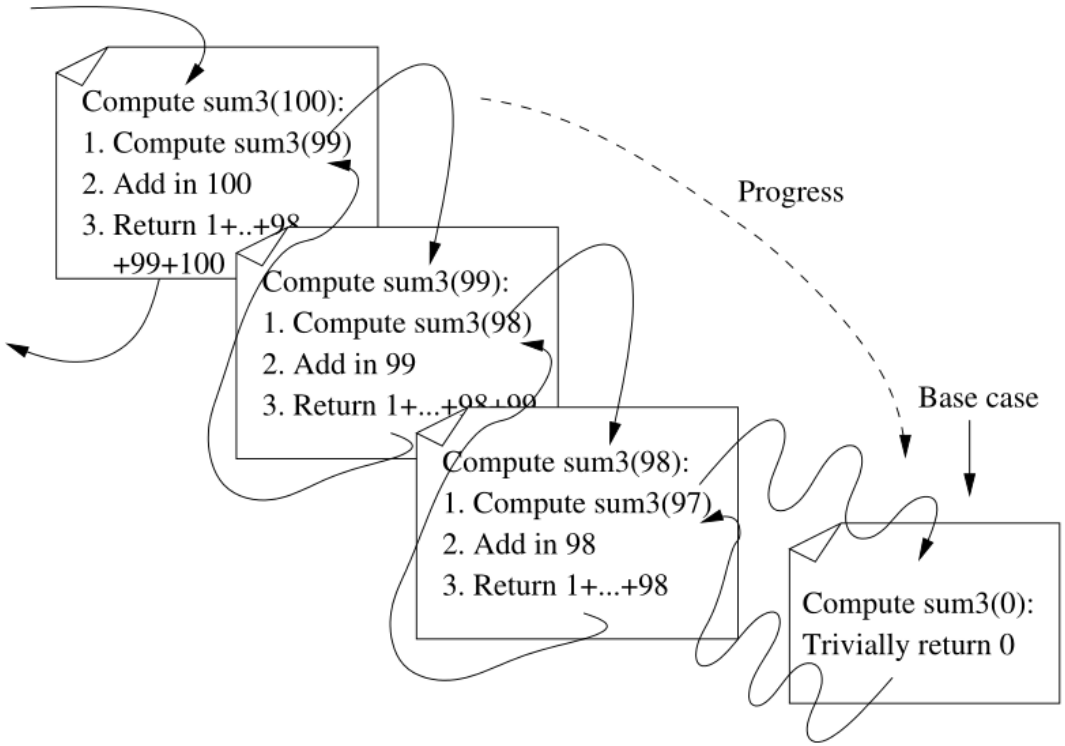


Case Study: Summing Integers

Task. Compute the sum of the numbers

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

A Declarative Approach



Recursion

What is Recursion?

- Recursion is a way to solve a problem using the solutions of smaller instances of the same problem

Recursion

What is Recursion?

- Recursion is a way to solve a problem using the solutions of smaller instances of the same problem
- Recursion can be viewed as declarative programming: Expressing the logic of a computation, without the control flow.

Recursion

What is Recursion?

- Recursion is a way to solve a problem using the solutions of smaller instances of the same problem
- Recursion can be viewed as declarative programming: Expressing the logic of a computation, without the control flow.
- Describe what should be done, not how to do it.

Recursion

What is Recursion?

- Recursion is a way to solve a problem using the solutions of smaller instances of the same problem
- Recursion can be viewed as declarative programming: Expressing the logic of a computation, without the control flow.
- Describe what should be done, not how to do it.

Why is Recursion Useful?

- Sometimes recursive description is more intuitive
- Recursion leads to more concise code
- Recursion minimizes side-effects
- Recursion allows rigorous analysis of the code

Recursion

Key Ingredients

- Find a parameter of the problem.

  We can reduce the problem to a smaller instance with a smaller parameter.

Recursion

Key Ingredients

- Find a parameter of the problem.

  We can reduce the problem to a smaller instance with a smaller parameter.

- Base Case: The parameter sizes are the smallest.

  Output the solution directly.

Recursion

Key Ingredients

- Find a parameter of the problem.

  We can reduce the problem to a smaller instance with a smaller parameter.

- Base Case: The parameter sizes are the smallest.

  Output the solution directly.

- Inductive Case: Bigger parameter sizes than the base case.

  1. Make Recursively calls on smaller instances.
  2. Use the result to obtain a solution of the current instance.

Recursion

Key Ingredients

- Find a parameter of the problem.

  We can reduce the problem to a smaller instance with a smaller parameter.

- Base Case: The parameter sizes are the smallest.

  Output the solution directly.

- Inductive Case: Bigger parameter sizes than the base case.
  1. Make Recursively calls on smaller instances.
  2. Use the result to obtain a solution of the current instance.

Before the time of computers (1930s), mathematicians/logicians use recursive functions to capture all computable functions.

Claim. All computation can be carried out recursively.

Example 1. Addition

Addition

$$add(m,n) = \begin{cases} m & \text{if } n = 0 \\ add(m, n-1) + 1 & \text{if } n > 0 \end{cases}$$

```
public static int add(int m, int n)
//pre:  m,n are natural numbers
//post:  return the sum of m,n
{
    if(n==0) return m;
    else return add(m,n-1)+1;
}
```

Example 2. Multiplication

Multiplication

Let m, n be natural numbers.

$$mult(m,n) = \begin{cases} 0 & \text{if } n = 0 \\ add(mult(m, n-1), m) & \text{if } n > 0 \end{cases}$$

Example 2. Multiplication

Multiplication

Let m, n be natural numbers.

$$mult(m,n) = \begin{cases} 0 & \text{if } n = 0 \\ add(mult(m, n-1), m) & \text{if } n > 0 \end{cases}$$

```
public static int multiply(int m, int n)
//pre:  m,n are natural numbers
//post:  return the product of m,n
{
    if(n==0) return 0;
    else return add(mult(m,n-1),m);
}
```

Example 3. Working with Strings

Counting Letters

Count the number of occurrences of space in a string s. Suppose s has length n

$$spaceCount(s) = \begin{cases} 0 & \text{if } s = \text{""} \\ spaceCount(s[0, n-2]) & \text{if } s[n-1] \neq \text{" "} \\ spaceCount(s[0, n-2]) + 1 & \text{if } s[n-1] = \text{" "} \end{cases}$$

Example 3. Working with Strings

Counting Letters

Count the number of occurrences of space in a string s. Suppose s has length n

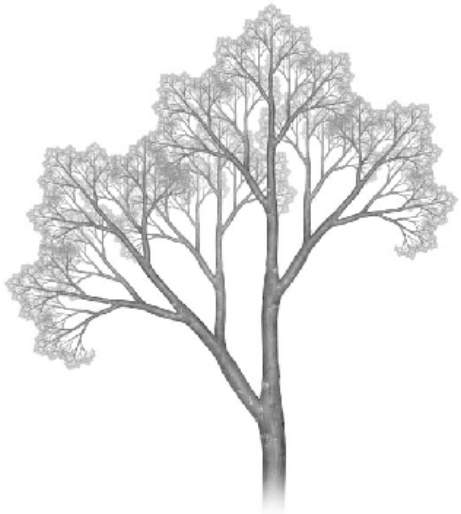
$$spaceCount(s) = \begin{cases} 0 & \text{if } s = \text{""} \\ spaceCount(s[0, n-2]) & \text{if } s[n-1] \neq \text{" "} \\ spaceCount(s[0, n-2]) + 1 & \text{if } s[n-1] = \text{" "} \end{cases}$$

```
public static int spaceCount(String s)
{
    int n=s.length();
    if(s.equals("")) return 0;
    else if(s.charAt(n-1)==' ')
        return spaceCount(s.substring(0,n-2));
    else
        return spaceCount(s.substring(0,n-2))+1;
}
```

Example 4. Drawing a Recursive Fractal

Fractals

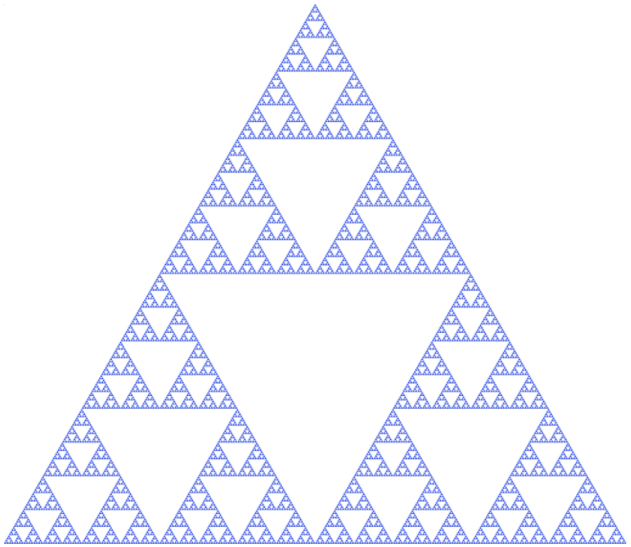
A fractal is a mathematical object that typically displays self-similar patterns.



Recursion can be used to draw fractal images on a screen.

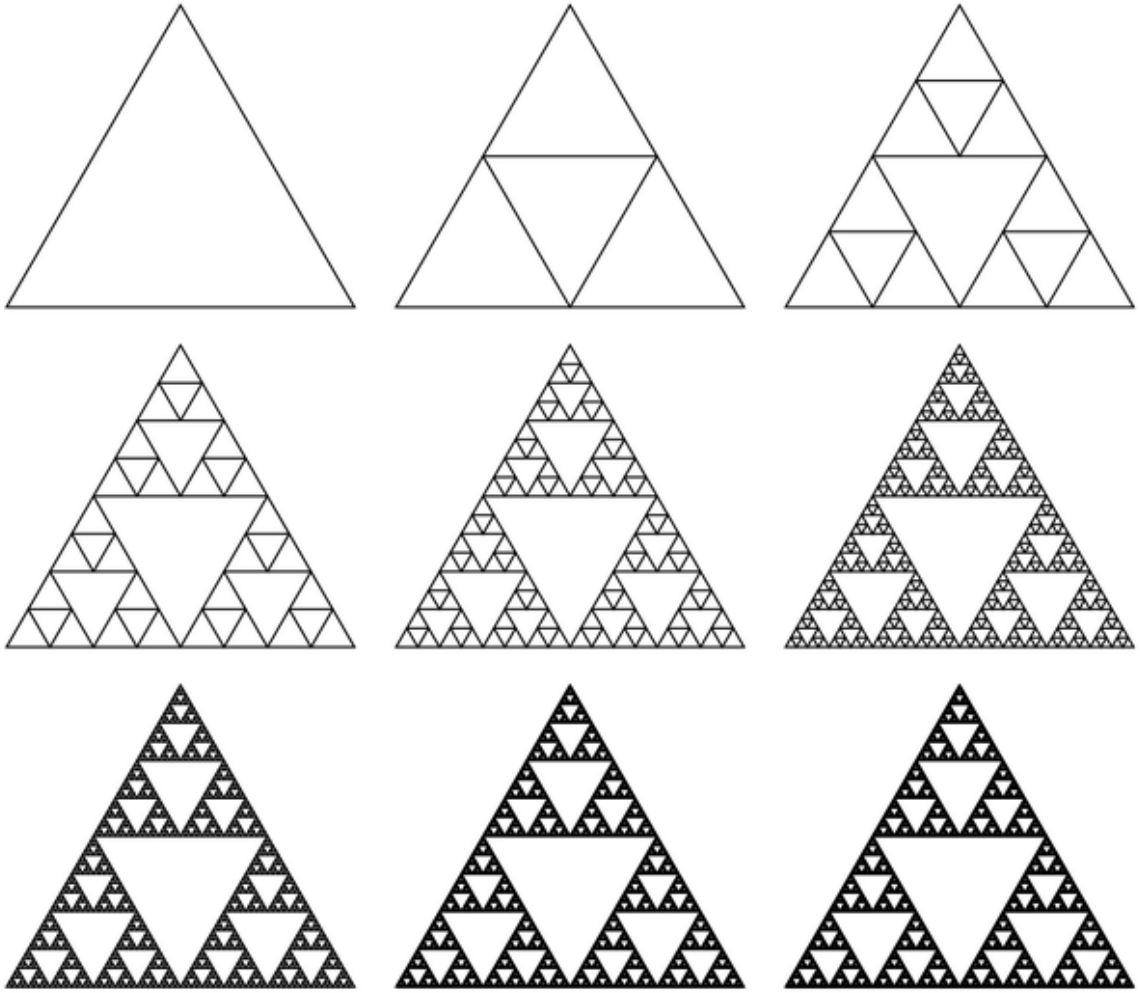
Example 4. Drawing a Recursive Fractal

Sierpinski's Triangle (1915)



Example 4. Drawing a Recursive Fractal

Sierpinski's Triangle (1915)



Example 4. Drawing a Recursive Fractal

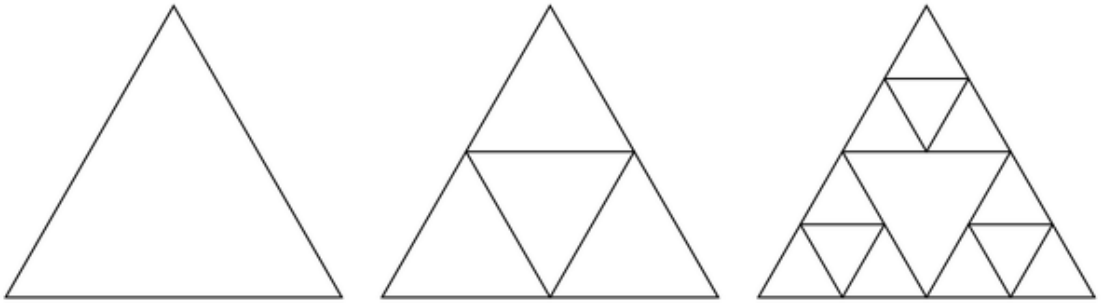
Sierpinski's Triangle (1915)

Draw a Sierpinski's triangle with length n:

- Draw a triangle with side length n
- Identify midpoints for all three sides
- Divide the triangle into four smaller triangles:

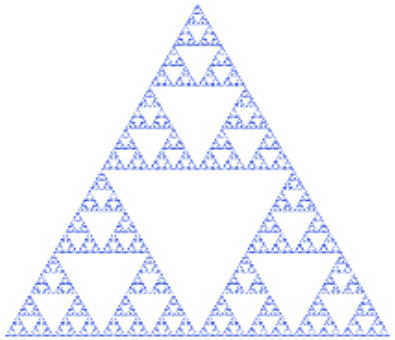
  top, left, right, center

- Recurse on the top, left and right triangles with length $n/2$.



Example 4. Drawing a Recursive Fractal

Sierpinski's Triangle (1915)



```
import javax.swing.*;
import java.awt.*;
public class Sierpinski extends JFrame{
        public static final int WINDOW_SIZE=450;
        public static final int THRESHOLD=10;
        public static int P1_x, P1_y, P2_x,P2_y,P3_x,P3_y;

        public Sierpinski(){
                super("Sierpinski");
                setSize(WINDOW_SIZE, WINDOW_SIZE);
                P1_x=(int)getSize().getWidth()/2;
                P1_y=20;
                P2_x=20;
                P2_y=(int)getSize().getHeight() - 20;
                P3_x=(int)getSize().getWidth() - 20;
                P3_y=(int)getSize().getHeight() - 20;
                setVisible(true);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
        public Point getMiddle(Point p1, Point p2){
                return new Point((int)(p1.getX()+p2.getX())/2,(int)(p1.getY()+p2.getY())/2);
        }
        public void paint(Graphics g){
                super.paint(g);
                sierpinski_draw(new Point(P1_x,P1_y), new Point(P2_x,P2_y), new Point(P3_x,P3_y));
        }
        public void sierpinski_draw(Point p1, Point p2, Point p3){
                if (p1.distance(p2)<THRESHOLD && p1.distance(p3)<THRESHOLD && p2.distance(p3)<THRESHOLD) return;

                Graphics g=getGraphics();
                g.drawLine((int)p1.getX(), (int)p1.getY(),(int)p2.getX(),(int)p2.getY());
                g.drawLine((int)p2.getX(), (int)p2.getY(),(int)p3.getX(),(int)p3.getY());
                g.drawLine((int)p3.getX(), (int)p3.getY(),(int)p1.getX(),(int)p1.getY());

                Point m12=getMiddle(p1,p2);
                Point m23=getMiddle(p2,p3);
                Point m31=getMiddle(p3,p1);

                sierpinski_draw(p1,m12,m31);
                sierpinski_draw(p2,m23,m12);
                sierpinski_draw(p3,m31,m23);
        }
        public static void main(String[] args){
                Sierpinski gasket  = new Sierpinski();
        }
}
```

Setting up initial points

draw a triangle (p1,p2,p3)

Recurse on smaller triangles

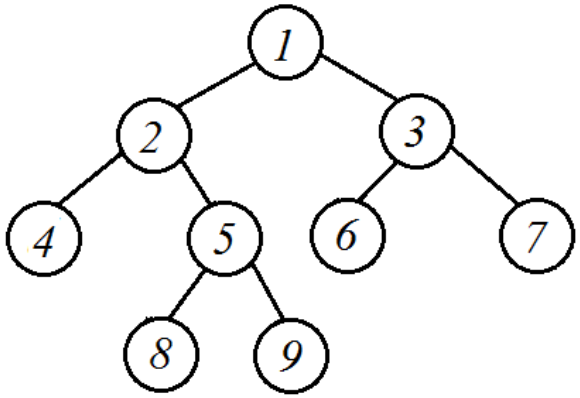
Example 5. Recursive Data Structures: Binary Trees

A recursive data structure is a data structure that is defined recursively.

Trees

A tree is a non-linear structure comprised of nodes, and edges which connect pairs of nodes in the following way:

- One (and only one) node is the root.
- A node u may have zero or more children below it; the node u is called the parent of its children.
- Each non-root node has exactly one parent; the root has no parent.
- From any node, iteratively following the parent pointers leads to the root.



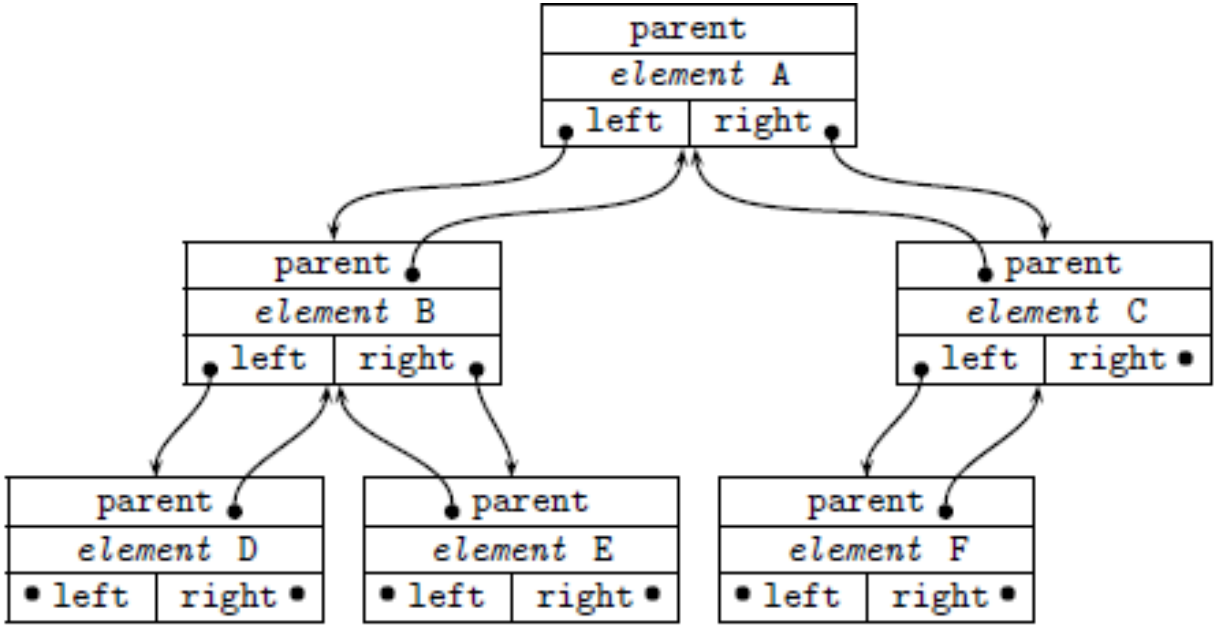
Example 5. Recursive Data Structures: Binary Trees

Binary Trees

A binary tree is a tree where every node can have at most 2 children, the left child and the right child.

The Binary Tree Interface

We use a recursive implementation for binary trees.
We represent an entire subtree as a reference to its root node.



Example 5. Recursive Data Structures: Binary Trees

```
public class BinaryTree<E>
{
    public BinaryTree()
    // post: constructor that generates an empty node

    public BinaryTree(E value)
    // post: returns a tree referencing value and two empty subtrees

    public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right)
    // post: returns a tree referencing value and two subtrees

    public BinaryTree<E> left()
    // post: returns reference to (possibly empty) left subtree

    public BinaryTree<E> right()
    // post: returns reference to (possibly empty) right subtree

    public BinaryTree<E> parent()
    // post: returns reference to parent node, or null

    public E value()
    // post: returns value associated with this node
```

Example 5. Recursive Data Structures: Binary Trees

```
    public void setLeft(BinaryTree<E> newLeft)
    // post: sets left subtree to newLeft
    //       re-parents newLeft if not null
    public void setRight(BinaryTree<E> newRight)
    // post: sets right subtree to newRight
    //       re-parents newRight if not null
    protected void setParent(BinaryTree<E> newParent)
    // post: re-parents this node to parent reference, or null
    public void setValue(E value)
    // post: sets the value associated with this node

    public Iterator<E> iterator()
    // post: returns an iterator of the elements

}
```

Example 5. Recursive Data Structures: Binary Trees

Fields:

```
protected E val;
protected BinaryTree<E> parent;
protected BinaryTree<E> left, right;
```

Example 5. Recursive Data Structures: Binary Trees

Fields:

```
protected E val;
protected BinaryTree<E> parent;
protected BinaryTree<E> left, right;
```

Methods (Constructors):

```
public BinaryTree(){
    val=null;
    parent = null;
    left=right=this;
}

public BinaryTree(E value){
    val = value;
    right =left=new BinaryTree<E>();
    setLeft(left);
    setRight(right);
}
```

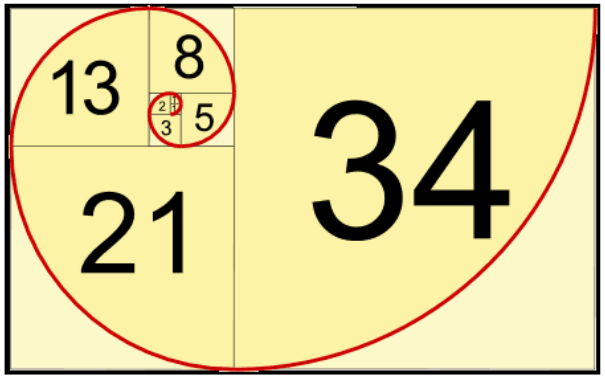
Example 5. Recursive Data Structures: Binary Trees

```
public BinaryTree(E value, BinaryTree<E> left,
BinaryTree<E> right){
    val = value;
    if (left == null) {
        left = new BinaryTree<E>();
    }
    setLeft(left);
    if (right==null) {
        right = new BinaryTree<E>();
    }
    setRight(right);
}
```

Part II: Introduction to Algorithms – A Case Study

Fibonacci Numbers: Definition

Leonardo of Pisa (Fibonacci)
1170 - 1250



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

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Fibonacci Problem

Note: F_n can be very large for small n

- $F_{30} = 832040$
- $F_{100} = 3.54 \times 10^{20}$

Fibonacci Problem

Note: F_n can be very large for small n

- $F_{30} = 832040$
- $F_{100} = 3.54 \times 10^{20}$

In fact, $F_n \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 1.618^n$

Fibonacci Problem

Note: F_n can be very large for small n

- $F_{30} = 832040$
- $F_{100} = 3.54 \times 10^{20}$

In fact, $F_n \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 1.618^n$

Fibonacci Problem

INPUT: Integer n
OUTPUT: The nth Fibonacci number F_n

We would like to design an algorithm to solve the Fibonacci problem.

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Correctness: The output of $fib1(n)$ is indeed F_n.

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Correctness: The output of $fib1(n)$ is indeed F_n.

Proof by induction:

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Correctness: The output of $fib1(n)$ is indeed F_n.

Proof by induction:

- Base Case: $fib1(0) = 0 = F_0$, and $fib(1) = 1 = F_1$

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Correctness: The output of $fib1(n)$ is indeed F_n.

Proof by induction:

- Base Case: $fib1(0) = 0 = F_0$, and $fib(1) = 1 = F_1$
- Inductive Step: Take a number $n \geq 1$.

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Correctness: The output of $fib1(n)$ is indeed F_n.

Proof by induction:

- Base Case: $fib1(0) = 0 = F_0$, and $fib(1) = 1 = F_1$
- Inductive Step: Take a number $n \geq 1$.
  Inductive hypothesis: Suppose $fib1(k) = F_k$ for all $0 \leq k \leq n$.

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Correctness: The output of $fib1(n)$ is indeed F_n.

Proof by induction:

- Base Case: $fib1(0) = 0 = F_0$, and $fib(1) = 1 = F_1$
- Inductive Step: Take a number $n \geq 1$.
  Inductive hypothesis: Suppose $fib1(k) = F_k$ for all $0 \leq k \leq n$.
  Then the $fib1(n+1)$ has output $fib1(n) + fib1(n-1)$.

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Correctness: The output of $fib1(n)$ is indeed F_n.

Proof by induction:

- Base Case: $fib1(0) = 0 = F_0$, and $fib(1) = 1 = F_1$
- Inductive Step: Take a number $n \geq 1$.
  Inductive hypothesis: Suppose $fib1(k) = F_k$ for all $0 \leq k \leq n$.
  Then the $fib1(n+1)$ has output $fib1(n) + fib1(n-1)$.
  By inductive hypothesis, this is $F_n + F_{n-1}$, which is F_{n+1}. $\square$

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Running time analysis

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Running time analysis

- Let $T(n)$ be the number of time units $fib1(n)$ takes

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Running time analysis

- Let $T(n)$ be the number of time units $fib1(n)$ takes
- If $n < 2$ then very fast: say $T(0) = T(1) = 1$

The First Try

Algorithm $fib1(n)$**:**

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Running time analysis

- Let $T(n)$ be the number of time units $fib1(n)$ takes
- If $n < 2$ then very fast: say $T(0) = T(1) = 1$
- If $n \geq 2$ then needs to go through line 1,2 and perform addition. So
$$T(n) \geq T(n-1) + T(n-2) + 3$$

The First Try

Algorithm $fib1(n)$**:**

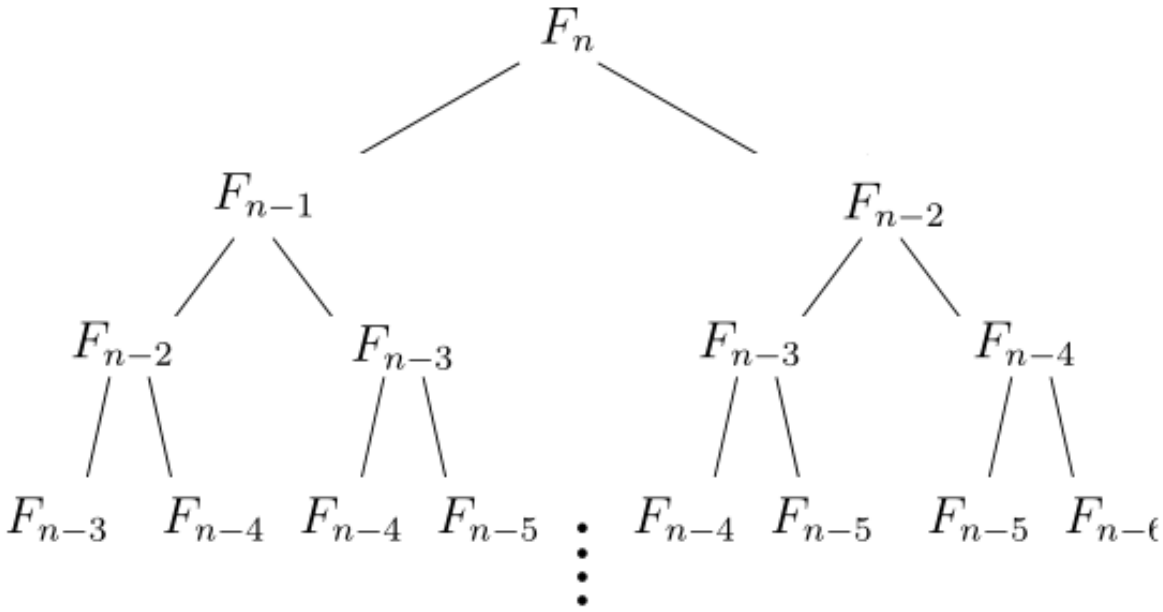
INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`return` $fib1(n-1) + fib1(n-2)$

Running time analysis

- Let $T(n)$ be the number of time units $fib1(n)$ takes
- If $n < 2$ then very fast: say $T(0) = T(1) = 1$
- If $n \geq 2$ then needs to go through line 1,2 and perform addition. So
$$T(n) \geq T(n-1) + T(n-2) + 3$$
- Thus $T(n) > f_n \approx 1.618^n$

The First Try

Main Problem with $fib1(n)$:
Too many redundant recursive calls.



Adding an auxiliary data structure may improve the efficiency of the algorithm.

Another Try

Algorithm $fib2(n)$**:**

INPUT: Integer n
OUTPUT: F_n

```
if n = 0 then return 0
create an array f[0..n]
f[0] ← 0,   f[1] ← 1
for i = 2..n do
    f[i] ← f[i − 1] + f[i − 2]
return f[n]
```

Another Try

Algorithm $fib2(n)$**:**

INPUT: Integer n
OUTPUT: F_n

```
if n = 0 then return 0
create an array f[0..n]
f[0] ← 0,   f[1] ← 1
for i = 2..n do
    f[i] ← f[i − 1] + f[i − 2]
return f[n]
```

Running time analysis

Another Try

Algorithm $fib2(n)$**:**

INPUT: Integer n
OUTPUT: F_n

```
if n = 0 then return 0
create an array f[0..n]
f[0] ← 0,   f[1] ← 1
for i = 2..n do
    f[i] ← f[i − 1] + f[i − 2]
return f[n]
```

Running time analysis

- Suppose creating an array of length $n + 1$ takes $n + 1$ time units
- Suppose each addition takes 1 time unit
- Suppose if statement,variable assignment,return takes 1 time unit

Another Try

Algorithm $fib2(n)$**:**

INPUT: Integer n
OUTPUT: F_n

```
if n = 0 then return 0
create an array f[0..n]
f[0] ← 0,   f[1] ← 1
for i = 2..n do
    f[i] ← f[i − 1] + f[i − 2]
return f[n]
```

Running time analysis

- Suppose creating an array of length $n + 1$ takes $n + 1$ time units
- Suppose each addition takes 1 time unit
- Suppose if statement,variable assignment,return takes 1 time unit
- Running time: $T(n) \leq n + 5 + 2(n - 1) = 3n + 3$ time units

Yet Another Try

A little bit of linear algebra

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Yet Another Try

A little bit of linear algebra

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Algorithm *multiply2by2*(A, B)

INPUT: Two 2×2 array A, B
OUTPUT: A 2×2 array representing $A \times B$
create a 2×2 array C
$C[0][0] \leftarrow A[0][0] \times B[0][0] + A[0][1] \times B[1][0]$
$C[0][1] \leftarrow A[0][0] \times B[0][1] + A[0][1] \times B[1][1]$
$C[1][0] \leftarrow A[1][0] \times B[0][0] + A[1][1] \times B[1][0]$
$C[1][1] \leftarrow A[1][0] \times B[0][1] + A[1][1] \times B[1][1]$

Yet Another Try

A little bit of linear algebra

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Algorithm *multiply2by2*(A, B)

INPUT: Two 2×2 array A, B
OUTPUT: A 2×2 array representing $A \times B$
create a 2×2 array C
$C[0][0] \leftarrow A[0][0] \times B[0][0] + A[0][1] \times B[1][0]$
$C[0][1] \leftarrow A[0][0] \times B[0][1] + A[0][1] \times B[1][1]$
$C[1][0] \leftarrow A[1][0] \times B[0][0] + A[1][1] \times B[1][0]$
$C[1][1] \leftarrow A[1][0] \times B[0][1] + A[1][1] \times B[1][1]$

Running time: No more than 36 units.

Yet Another Try

Let $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$.

Then F_n is the $(0,1)$-entry in the matrix A^n.

Yet Another Try

Let $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$.
Then F_n is the $(0,1)$-entry in the matrix A^n.

Algorithm $fib3(n)$

INPUT: Integer n
OUTPUT: F_n

```
if n = 0 then return 0
if n = 1 then return 1
create a 2×2 array A ← [[0,1],[1,1]]
create a 2×2 array B ← [[1,0],[0,1]] % (The Identity matrix)
for i = 1..n do
    B ← multiply2by2(B, A)
return B[0][1]
```

Yet Another Try

Let $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$.
Then F_n is the $(0,1)$-entry in the matrix A^n.

Algorithm $fib3(n)$

INPUT: Integer n
OUTPUT: F_n
`if` $n = 0$ `then return 0`
`if` $n = 1$ `then return 1`
`create a` 2×2 `array` $A \leftarrow [[0,1],[1,1]]$
`create a` 2×2 `array` $B \leftarrow [[1,0],[0,1]]$ `% (The Identity matrix)`
`for` $i = 1..n$ `do`
    $B \leftarrow multiply2by2(B, A)$
`return` $B[0][1]$

Running time: $T(n) \leq 36n + 12$ time units

Yet Another Try

Observation: We only need roughly $\log_2 n$ number of matrix multiplications to compute the nth power of a matrix.

Yet Another Try

Observation: We only need roughly $\log_2 n$ number of matrix multiplications to compute the nth power of a matrix.

Say $n = 25$.

Yet Another Try

Observation: We only need roughly $\log_2 n$ number of matrix multiplications to compute the nth power of a matrix.

Say $n = 25$.

- Express n in binary form: $25 = (11001)_2$

Yet Another Try

Observation: We only need roughly $\log_2 n$ number of matrix multiplications to compute the nth power of a matrix.

Say $n = 25$.

- Express n in binary form: $25 = (11001)_2$
- Therefore $25 = 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1$

Yet Another Try

Observation: We only need roughly $\log_2 n$ number of matrix multiplications to compute the nth power of a matrix.

Say $n = 25$.

- Express n in binary form: $25 = (11001)_2$
- Therefore $25 = 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1$
- Thus $A^25 = A^{16} \times A^8 \times A^1$

Yet Another Try

Observation: We only need roughly $\log_2 n$ number of matrix multiplications to compute the nth power of a matrix.

Say $n = 25$.

- Express n in binary form: $25 = (11001)_2$
- Therefore $25 = 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1$
- Thus $A^25 = A^{16} \times A^8 \times A^1$
- Thus we only need to compute:

$$A^1 \rightarrow A^2 \rightarrow A^4 \rightarrow A^8 \rightarrow A^16$$

then multiply $A^1 \times A^8 \times A^{16}$.
$\Rightarrow$ 6 matrix multiplications instead of 25

Yet Another Try

Observation: We only need roughly $\log_2 n$ number of matrix multiplications to compute the nth power of a matrix.

Say $n = 25$.

- Express n in binary form: $25 = (11001)_2$
- Therefore $25 = 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1$
- Thus $A^25 = A^{16} \times A^8 \times A^1$
- Thus we only need to compute:

$$A^1 \to A^2 \to A^4 \to A^8 \to A^16$$

  then multiply $A^1 \times A^8 \times A^{16}$.
  $\Rightarrow$ 6 matrix multiplications instead of 25

In general, the maximum number of matrix multiplications is proportional to the length of the binary representation of n, which is $\approx \log_2 n$

Yet Another Try

Algorithm $fib4(n)$

INPUT: Integer n
OUTPUT: F_n

```
if n = 0 then return 0
if n = 1 then return 1
create a 2×2 array A ← [[0,1],[1,1]]
create a 2×2 array B ← [[1,0],[0,1]]
while n > 0 do
    if n%2 = 1 then
        B ← multiply2by2(B, A)
    A ← multiply2by2(A, A)
    n ← n ÷ 2
return B[0][1]
```

Yet Another Try

Algorithm $fib4(n)$

INPUT: Integer n
OUTPUT: F_n

```
if n = 0 then return 0
if n = 1 then return 1
create a 2×2 array A ← [[0,1],[1,1]]
create a 2×2 array B ← [[1,0],[0,1]]
while n > 0 do
    if n%2 = 1 then
        B ← multiply2by2(B, A)
    A ← multiply2by2(A, A)
    n ← n ÷ 2
return B[0][1]
```

Running time: $T(n) \approx 72 \log_2 n + 9$ time units.

Summary

Algorithm	Running Time (time units)
$fib1$	$\geq 1.618^n$
$fib2$	$\leq 3n + 3$
$fib3$	$\leq 36n + 12$
$fib4$	$\leq 72 \log_2 n + 9$

Next question: Which algorithm is the "best"?

Part III: Measuring Performance

Measuring Performance

Question: How do we measure the efficiency of an algorithm?

Resource-bounded Complexity

- Time: Execution time of an algorithm
- Space: Disk space taken by an algorithm

Time and Space depend on

- machine
- operating system
- programming language
- system environment
- etc.

Machine Models

Algorithmic Analysis and Machine Models

- The theoretical analysis of algorithms aims to rigorously study on the performance of algorithms based on a certain model of computation.
- A model of computation is a collection of assumptions/idealisation about the type of machines and system environment that the algorithm is running on.
- The model of computation specifies the elementary operations.

Random Access Model

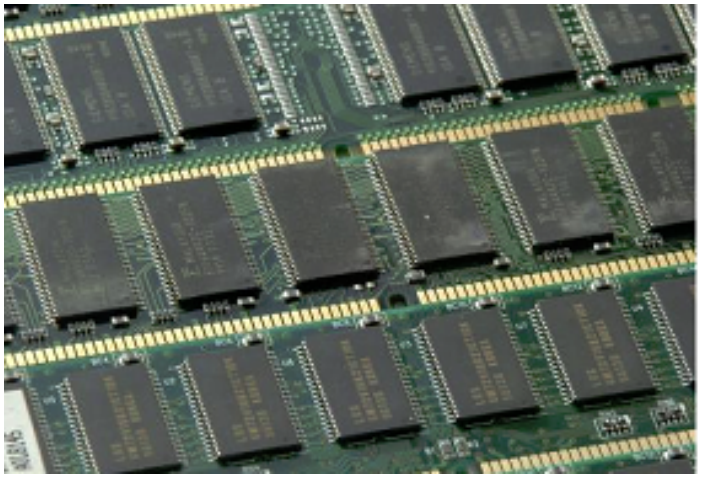
In the classical study of algorithms (and in ADA), we assume the random access model (RAM).

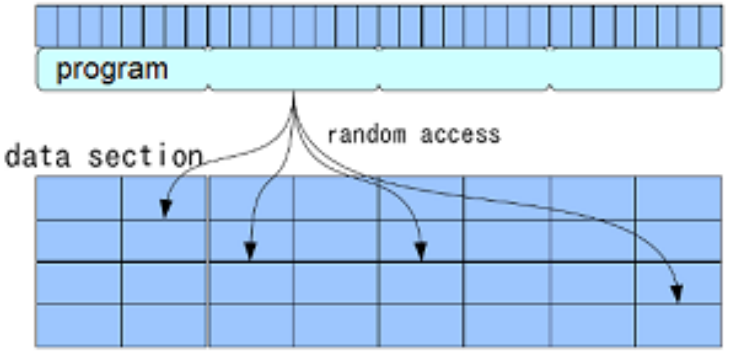
Random Access Model

In the classical study of algorithms (and in ADA), we assume the random access model (RAM).

RAM (Random Access Model)

A single processor connected to memory which it can access each part in constant time.
Consequences: Creating/accessing/changing a single primitive variable takes constant time.
Concurrency is only achieved by time sharing.





Machine Model

There are other types of models of computation:

- Sequential Access Model: Memory is in the form of an ordered sequence (a tape) which is accessed only in sequence by a single processor.
- λ-Calculus: Model of functional programming.
- Parallel Random Access Model (PRAM): Multiple processors; concurrency; random accessed memory.
- Other models: register machines, actor model, Neural Networks, quantum model, DNA model, cellular automata etc.

Measuring Performance

Measure 1. Time

Measuring Performance

Measure 1. Time

- In RAM, we assume
  $+, -, \times, \div, \%, ==, !=, <, >$,creation/assignment of primitive variables,pointer manipulations are elementary operations.

Measuring Performance

Measure 1. Time

- In RAM, we assume $+, -, \times, \div, \%, ==, !=, <, >$,creation/assignment of primitive variables,pointer manipulations are elementary operations.
- The running time of an algorithm on an input is the number of elementary operations performed by the algorithm.

Measuring Performance

Measure 1. Time

- In RAM, we assume $+, -, \times, \div, \%, ==, !=, <, >$,creation/assignment of primitive variables,pointer manipulations are elementary operations.
- The running time of an algorithm on an input is the number of elementary operations performed by the algorithm.
- Note that the same algorithm may have different running time even on inputs of the same size. (e.g. sorting)

Measuring Performance

Measure 1. Time

- In RAM, we assume $+, -, \times, \div, \%, ==, !=, <, >$,creation/assignment of primitive variables,pointer manipulations are elementary operations.
- The running time of an algorithm on an input is the number of elementary operations performed by the algorithm.
- Note that the same algorithm may have different running time even on inputs of the same size. (e.g. sorting)
- The worst-case time complexity of an algorithm is a function that maps n to the longest running time of the algorithm on any input with size n.

Measuring Performance

Measure 1. Time

- In RAM, we assume $+, -, \times, \div, \%, ==, !=, <, >$,creation/assignment of primitive variables,pointer manipulations are elementary operations.
- The running time of an algorithm on an input is the number of elementary operations performed by the algorithm.
- Note that the same algorithm may have different running time even on inputs of the same size. (e.g. sorting)
- The worst-case time complexity of an algorithm is a function that maps n to the longest running time of the algorithm on any input with size n.
- The average-case time complexity of an algorithm is a function that maps n to the average running time of the algorithm over all input with size n (assuming some distribution).

Measuring Performance

Measure 2. Space

Measuring Performance

Measure 2. Space

- The space required by an algorithm for an input is the maximal number of bits the algorithm needs in the memory at any instance.

Measuring Performance

Measure 2. Space

- The space required by an algorithm for an input is the maximal number of bits the algorithm needs in the memory at any instance.
- The worst-case space complexity of an algorithm is a function that maps n to the biggest space required by an algorithm for any input with size n.

Measuring Performance

Measure 2. Space

- The space required by an algorithm for an input is the maximal number of bits the algorithm needs in the memory at any instance.
- The worst-case space complexity of an algorithm is a function that maps n to the biggest space required by an algorithm for any input with size n.
- Similarly we can define average-case space complexity.

Measuring Performance

Note

- The mentioned worst-case / average case time/space complexity are only theoretical measures which makes sense only under some idealised assumptions.
- They should not be treated as the ONLY rule for measuring performances of algorithms.
- Another very important measure is how an algorithm actually performs in practice, which can be obtained through experimentations.
- *"Algorithms are meant to be implemented and used"*

Measuring Performance

Recall: These running times were obtained assuming RAM:

Algorithm	Running Time (time units)
$fib1$	$\geq 1.618^n$
$fib2$	$\leq 3n + 3$
$fib3$	$\leq 36n + 12$
$fib4$	$\leq 72 \log_2 n + 9$

A Note on n

The n above is the input value, not the input size.
The input size is the number of bits we need to store the input in the computer.
If the input size of an integer is n, then the input value is at most 2^n.
Therefore if the running time is $f(n)$ in terms of input value, the complexity is $f(2^n)$ in terms of input size.

Measuring Performance

Therefore the following running time should be shown, which is stated treating n as the input size:

Algorithm	Running Time (time units)
$fib1$	$\geq 1.618^{2^n}$
$fib2$	$\leq 3 \times 2^n + 3$
$fib3$	$\leq 36 \times 2^n + 12$
$fib4$	$\leq 72n + 9$

Comparing Functions

Question

Say we have three algorithms A, B, C.
- A runs in time $10 \log n$
- B runs in time $2n + 3$
- C runs in time $10n + 5$

What can we say about their running time?

Comparing Functions

Question

Say we have three algorithms A, B, C.
- A runs in time $10 \log n$
- B runs in time $2n + 3$
- C runs in time $10n + 5$

What can we say about their running time?

- We only care about the situation when the input is very big.
- We can "speed-up" an algorithm by running it on a faster computer by a constant factor.

Comparing Functions

Question

Say we have three algorithms A, B, C.
- A runs in time $10 \log n$
- B runs in time $2n + 3$
- C runs in time $10n + 5$

What can we say about their running time?

- We only care about the situation when the input is very big.
- We can "speed-up" an algorithm by running it on a faster computer by a constant factor.

Therefore A has much better running time than B, C
but B is not very different from C.

Big-O

Definition [O]

We say that a function $g(n)$ is not bigger than another function $f(n)$ if there are c and n_0 such that for all $n > n_0$ we have $g(n) \leq cf(n)$.
In this case, we also say that $g(n)$ is $O(f(n))$, or $f(n)$ is an asymptotic upperbound for $g(n)$. i.e., $g(n)$ is $O(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

Big-O

Definition [O]

We say that a function $g(n)$ is not bigger than another function $f(n)$ if there are c and n_0 such that for all $n > n_0$ we have $g(n) \leq cf(n)$.
In this case, we also say that $g(n)$ is $O(f(n))$, or $f(n)$ is an asymptotic upperbound for $g(n)$. i.e., $g(n)$ is $O(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

Examples:

Big-O

Definition [O]

We say that a function $g(n)$ is not bigger than another function $f(n)$ if there are c and n_0 such that for all $n > n_0$ we have $g(n) \leq cf(n)$.
In this case, we also say that $g(n)$ is $O(f(n))$, or $f(n)$ is an asymptotic upperbound for $g(n)$. i.e., $g(n)$ is $O(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

Examples:

- $5n - 2$ is $O(n)$.
- $5n - 2$ is also $O(15n + 33)$, $O(n^2)$, $O(n \log n)$ and $O(2^n)$.

Big-O

Definition [O]

We say that a function $g(n)$ is not bigger than another function $f(n)$ if there are c and n_0 such that for all $n > n_0$ we have $g(n) \leq cf(n)$.
In this case, we also say that $g(n)$ is $O(f(n))$, or $f(n)$ is an asymptotic upperbound for $g(n)$. i.e., $g(n)$ is $O(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

Examples:

- $5n - 2$ is $O(n)$.
- $5n - 2$ is also $O(15n + 33)$, $O(n^2)$, $O(n \log n)$ and $O(2^n)$.
- $3 \log_{10} n + 6$ is $O(\log n)$
- $3 \log_{10} n + 6$ is also $O(100 \log_8(n^2 + 2n))$.

Big-O

Definition [O]

We say that a function $g(n)$ is not bigger than another function $f(n)$ if there are c and n_0 such that for all $n > n_0$ we have $g(n) \leq cf(n)$.
In this case, we also say that $g(n)$ is $O(f(n))$, or $f(n)$ is an asymptotic upperbound for $g(n)$. i.e., $g(n)$ is $O(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

Examples:

- $5n - 2$ is $O(n)$.
- $5n - 2$ is also $O(15n + 33)$, $O(n^2)$, $O(n \log n)$ and $O(2^n)$.
- $3 \log_{10} n + 6$ is $O(\log n)$
- $3 \log_{10} n + 6$ is also $O(100 \log_8(n^2 + 2n))$.
- $4n^2$ is $O(n^2)$, $O(n^3)$ and $O(n^3 + n^2)$.
- $4n^2$ is NOT $O(n)$.

Big-Omega

Definition [Ω]

We say that a function $g(n)$ is not smaller than another function $f(n)$ if there are c and n_0 such that for all $n > n_0$ we have $g(n) \geq cf(n)$.
In this case, we also say that $g(n)$ is $\Omega(f(n))$, or $f(n)$ is an asymptotic lowerbound for $g(n)$, i.e., $g(n)$ is $\Omega(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} > 0$$

Big-Omega

Definition [Ω]

We say that a function $g(n)$ is not smaller than another function $f(n)$ if there are c and n_0 such that for all $n > n_0$ we have $g(n) \geq cf(n)$.
In this case, we also say that $g(n)$ is $\Omega(f(n))$, or $f(n)$ is an asymptotic lowerbound for $g(n)$, i.e., $g(n)$ is $\Omega(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} > 0$$

Examples:

- $5n - 2$ is $\Omega(16n + 33)$.
- $3n^2$ is $\Omega(n)$.
- $2n \log n$ is $\Omega(n + 2)$.

Big-Theta

Definition [Θ]

We say that a function $g(n)$ has the same rate as another function $f(n)$ if there are c_1, c_2 and n_0 such that for all $n > n_0$ we have $c_1 f(n) \leq g(n) \leq c_2 f(n)$.
In this case, we also say that $g(n)$ is $\Theta(f(n))$, or $g(n)$ has the same asymptotic complexity as $f(n)$, i.e., i.e., $g(n)$ is $\Omega(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \text{ is a positive real number}$$

Big-Theta

Definition [Θ]

We say that a function $g(n)$ has the same rate as another function $f(n)$ if there are c_1, c_2 and n_0 such that for all $n > n_0$ we have $c_1 f(n) \leq g(n) \leq c_2 f(n)$.
In this case, we also say that $g(n)$ is $\Theta(f(n))$, or $g(n)$ has the same asymptotic complexity as $f(n)$, i.e., i.e., $g(n)$ is $\Omega(f(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \text{ is a positive real number}$$

Examples:

- $5n - 2$ is $\Theta(16n + 33)$.
- $\log_{10} n$ is $\Theta(\log n)$.
- $2n^2 + 3n$ is $\Theta(n^2)$.

Properties of Big-O, Big-Omega, Big-Theta

- Reflexivity:
  $f(n)$ is $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$

Properties of Big-O, Big-Omega, Big-Theta

- Reflexivity:
  $f(n)$ is $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$
- Transitivity:
  $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n)) \Rightarrow f(n)$ is $O(h(n))$.
  $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n)) \Rightarrow f(n)$ is $\Omega(h(n))$.
  $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n)) \Rightarrow f(n)$ is $\Theta(h(n))$.

Properties of Big-O, Big-Omega, Big-Theta

- Reflexivity:
  $f(n)$ is $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$
- Transitivity:
  $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n)) \Rightarrow f(n)$ is $O(h(n))$.
  $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n)) \Rightarrow f(n)$ is $\Omega(h(n))$.
  $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n)) \Rightarrow f(n)$ is $\Theta(h(n))$.
- Symmetry:
  $f(n)$ is $\Theta(g(n)) \Leftrightarrow g(n)$ is $\Theta(f(n))$

Properties of Big-O, Big-Omega, Big-Theta

- Reflexivity:
  $f(n)$ is $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$
- Transitivity:
  $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n)) \Rightarrow f(n)$ is $O(h(n))$.
  $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n)) \Rightarrow f(n)$ is $\Omega(h(n))$.
  $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n)) \Rightarrow f(n)$ is $\Theta(h(n))$.
- Symmetry:
  $f(n)$ is $\Theta(g(n)) \Leftrightarrow g(n)$ is $\Theta(f(n))$
- Other:
  $f(n)$ is $O(g(n)) \Leftrightarrow g(n)$ is $\Omega(f(n))$

Properties of Big-O, Big-Omega, Big-Theta

- Reflexivity:
  $f(n)$ is $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$
- Transitivity:
  $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n)) \Rightarrow f(n)$ is $O(h(n))$.
  $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n)) \Rightarrow f(n)$ is $\Omega(h(n))$.
  $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n)) \Rightarrow f(n)$ is $\Theta(h(n))$.
- Symmetry:
  $f(n)$ is $\Theta(g(n)) \Leftrightarrow g(n)$ is $\Theta(f(n))$
- Other:
  $f(n)$ is $O(g(n)) \Leftrightarrow g(n)$ is $\Omega(f(n))$
  $f(n)$ is $\Theta(g(n)) \Leftrightarrow f(n)$ is both $O(g(n))$ and $\Omega(g(n))$

Properties of Big-O, Big-Omega, Big-Theta

- Reflexivity:
  $f(n)$ is $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$
- Transitivity:
  $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n)) \Rightarrow f(n)$ is $O(h(n))$.
  $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n)) \Rightarrow f(n)$ is $\Omega(h(n))$.
  $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n)) \Rightarrow f(n)$ is $\Theta(h(n))$.
- Symmetry:
  $f(n)$ is $\Theta(g(n)) \Leftrightarrow g(n)$ is $\Theta(f(n))$
- Other:
  $f(n)$ is $O(g(n)) \Leftrightarrow g(n)$ is $\Omega(f(n))$
  $f(n)$ is $\Theta(g(n)) \Leftrightarrow f(n)$ is both $O(g(n))$ and $\Omega(g(n))$
  $f(n)$ is $O(g(n)) \Rightarrow f(n) + g(n)$ is $O(g(n))$

Summary

Algorithm Analysis

- Model of Computation: RAM model
- Measuring criteria: Time and space

  In this course we mainly look at the worst case running time.
- Input size: We measure the complexity of algorithms in terms of the number of bits of the input.
- Comparing functions: We express asymptotic complexity using O, Ω, Θ.

Part IV: An Excursion to Function Wonderland

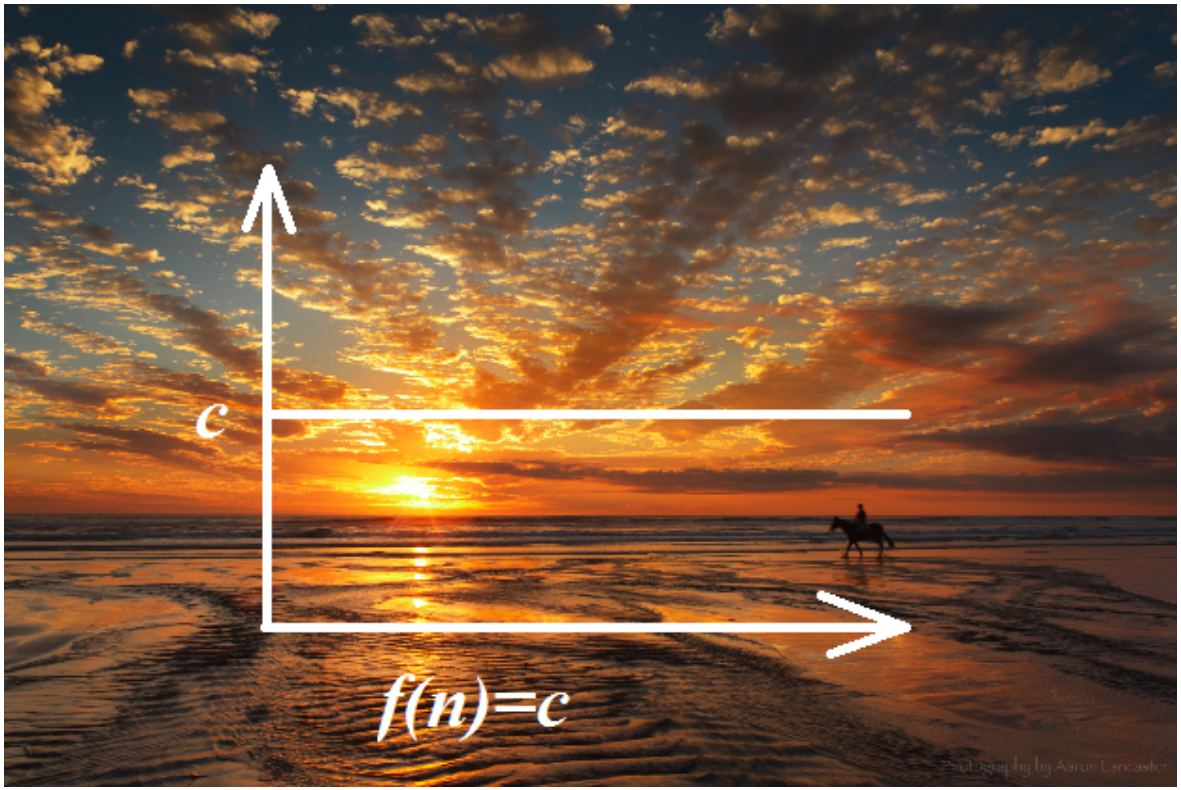
Before We Go

- $f_1(n) = n + 100$
- $f_2(n) = 4n^{1/2} + 3\log n$
- $f_3(n) = n^2 + 3n\log n$
- $f_4(n) = n^4 + \log^5 n$
- $f_5(n) = n + \log^3(2n)$
- $f_6(n) = 2^n + n^3$
- $f_7(n) = n^{1000} + 1000n^3$
- $f_8(n) = n^2\log n + n\log^2 n$

How do these functions compare asymptotically?

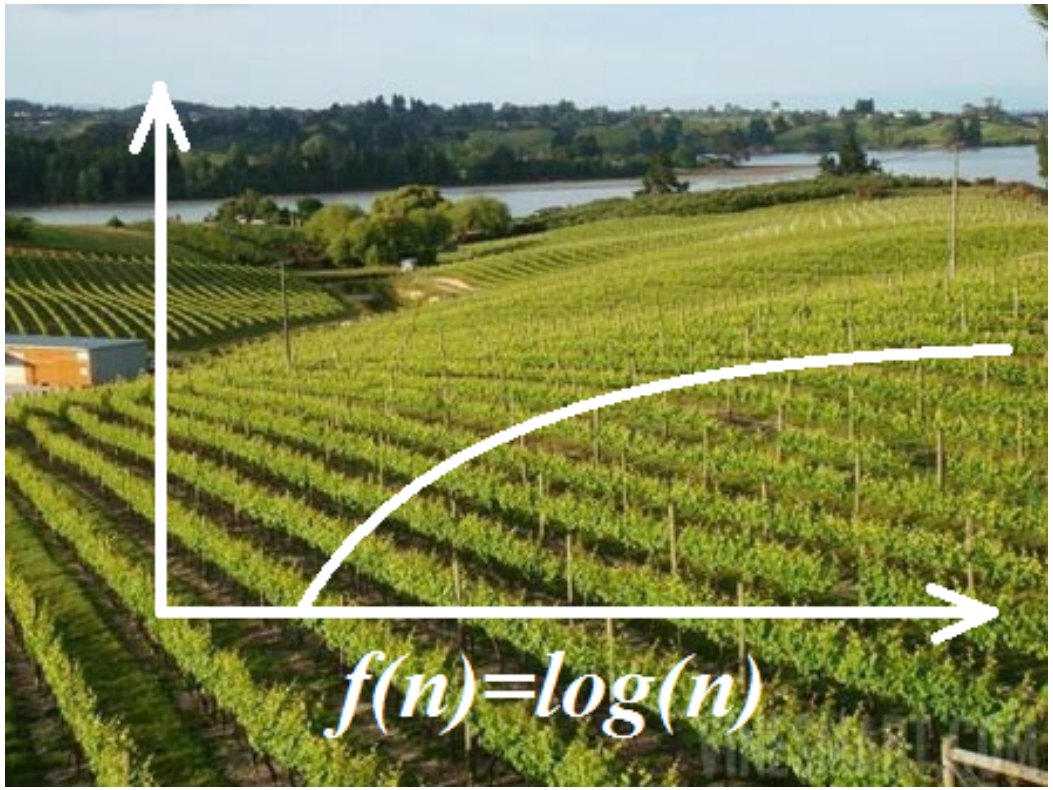
Constants

Constants



Logarithms

Logarithms



Facts about Logarithms

- We use base 2, i.e, $2^{\log n} = n$.

Facts about Logarithms

- We use base 2, i.e, $2^{\log n} = n$.
- $\log mn = \log m + \log n$

Facts about Logarithms

- We use base 2, i.e, $2^{\log n} = n$.
- $\log mn = \log m + \log n$
- $\log n^c = c \log n$

Facts about Logarithms

- We use base 2, i.e, $2^{\log n} = n$.
- $\log mn = \log m + \log n$
- $\log n^c = c \log n$

Consequences

- $\log 100n$, $\log n^2$, $\log n^3$, $\log n^{100}$ are all $\Theta(\log n)$

Therefore we normally omit the base in expressing logarithms.

Facts about Logarithms

- We use base 2, i.e, $2^{\log n} = n$.
- $\log mn = \log m + \log n$
- $\log n^c = c \log n$
- $\log_b n = \log n / \log b$

Consequences

- $\log 100n$, $\log n^2$, $\log n^3$, $\log n^{100}$ are all $\Theta(\log n)$

Therefore we normally omit the base in expressing logarithms.

Facts about Logarithms

- We use base 2, i.e, $2^{\log n} = n$.
- $\log mn = \log m + \log n$
- $\log n^c = c \log n$
- $\log_b n = \log n / \log b$

Consequences

- $\log 100n$, $\log n^2$, $\log n^3$, $\log n^{100}$ are all $\Theta(\log n)$
- $\log_3 n$, $\log_4 n$, $\log_{100} n$ are all $\Theta(\log n)$

Therefore we normally omit the base in expressing logarithms.

Powers of Logarithms

Note: $(\log n)^2 = (\log n) \times (\log n) \neq \log n^2 = \log(n \times n)$

Powers of Logarithms

Note: $(\log n)^2 = (\log n) \times (\log n) \neq \log n^2 = \log(n \times n)$

Powers of Logarithms

- We use $\log^c n$ to denote $(\log n)^c$.

Powers of Logarithms

Note: $(\log n)^2 = (\log n) \times (\log n) \neq \log n^2 = \log(n \times n)$

Powers of Logarithms

- We use $\log^c n$ to denote $(\log n)^c$.
- $\log^c n$ is $O(\log^d n) \Leftrightarrow c \leq d$.
- $\log^c n$ is $\Omega(\log^d n) \Leftrightarrow c \geq d$.
- $\log^c n$ is $\Theta(\log^d n) \Leftrightarrow c = d$.

Powers of Logarithms

Note: $(\log n)^2 = (\log n) \times (\log n) \neq \log n^2 = \log(n \times n)$

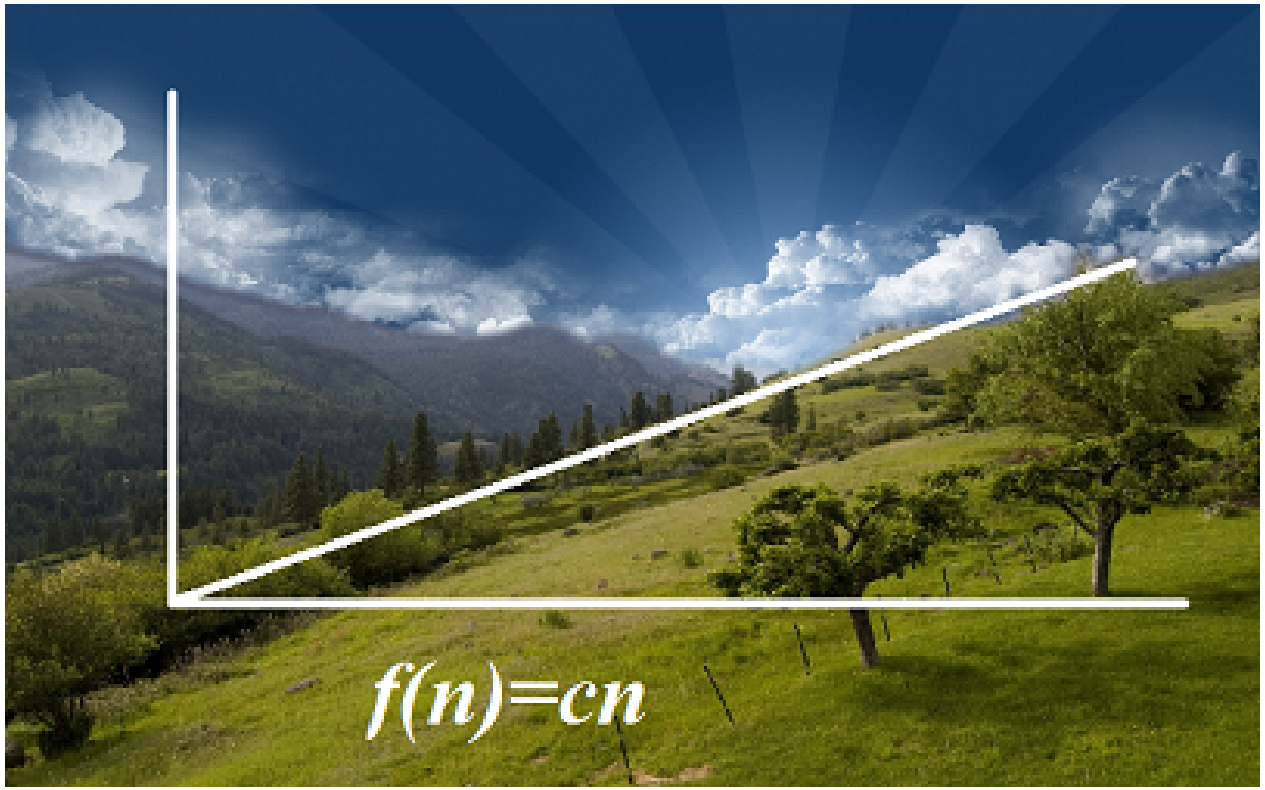
Powers of Logarithms

- We use $\log^c n$ to denote $(\log n)^c$.
- $\log^c n$ is $O(\log^d n) \Leftrightarrow c \leq d$.
- $\log^c n$ is $\Omega(\log^d n) \Leftrightarrow c \geq d$.
- $\log^c n$ is $\Theta(\log^d n) \Leftrightarrow c = d$.

Therefore $f(n) = \log n$ is $O(\log^2 n)$ but not $\Omega(\log^2 n)$.

Linear Functions

Linear Functions



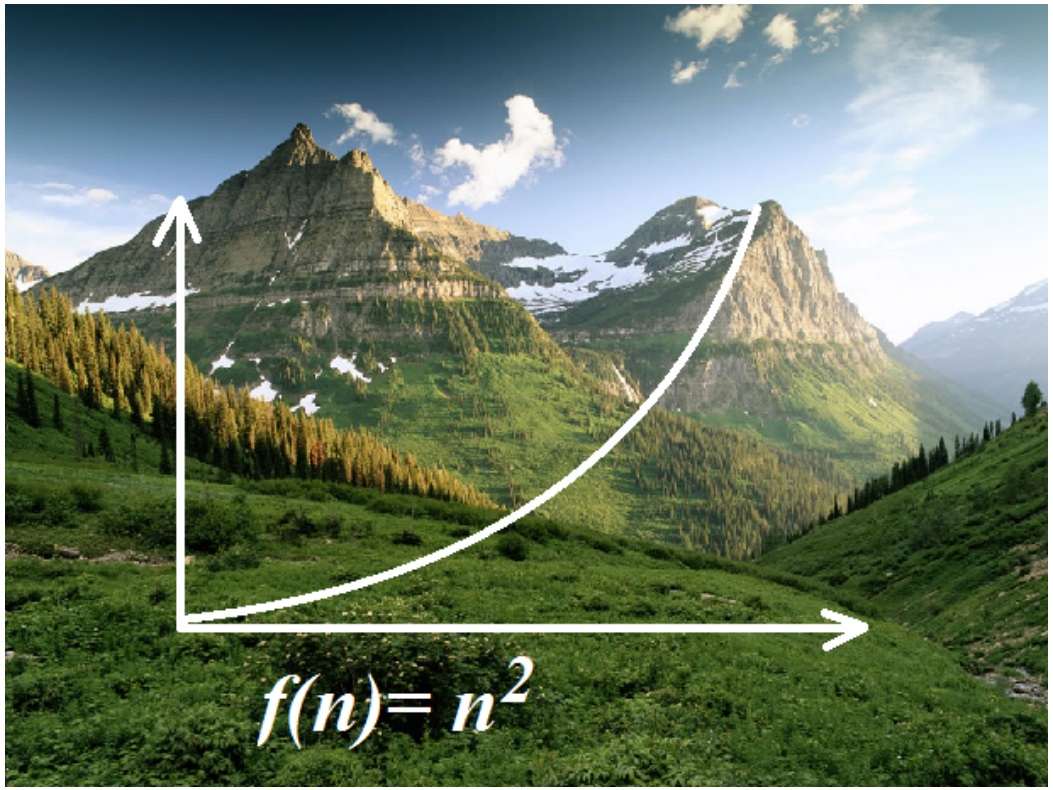
Logarithms and Linear Functions

Facts.

- $\log n$ is $O(n)$, but is not $\Omega(n)$
- For any $c > 1$, $\log^c n$ is $O(n)$, but is not $\Omega(n)$
- n is $O(n \log n)$ but not $\Omega(n \log n)$

Polynomials

Polynomials



Polynomials

Polynomials

- n^c is $O(n^d) \Leftrightarrow c \leq d$
- n^c is $\Omega(n^d) \Leftrightarrow c \geq d$
- n^c is $\Theta(n^c) \Leftrightarrow c = d$

Polynomials

Polynomials

- n^c is $O(n^d) \Leftrightarrow c \leq d$
- n^c is $\Omega(n^d) \Leftrightarrow c \geq d$
- n^c is $\Theta(n^c) \Leftrightarrow c = d$

Therefore $f(n) = n$ is $O(n^2)$, but is not $\Omega(n^2)$.
$f(n) = n^{1/2} = \sqrt{n}$ is $O(n)$, but is not $\Omega(n)$.

Polynomials v.s. Logarithms

Question

How to compare $n^{0.1}$ and $\log^{100} n$?

Polynomials v.s. Logarithms

Question

How to compare $n^{0.1}$ and $\log^{100} n$?

Fact

For any $c \geq 1$ and $d > 0$, we have

$$\log^c n \text{ is } O(n^d), \text{ but is not } \Omega(n^d)$$

Polynomials v.s. Logarithms

Question

How to compare $n^{0.1}$ and $\log^{100} n$?

Fact

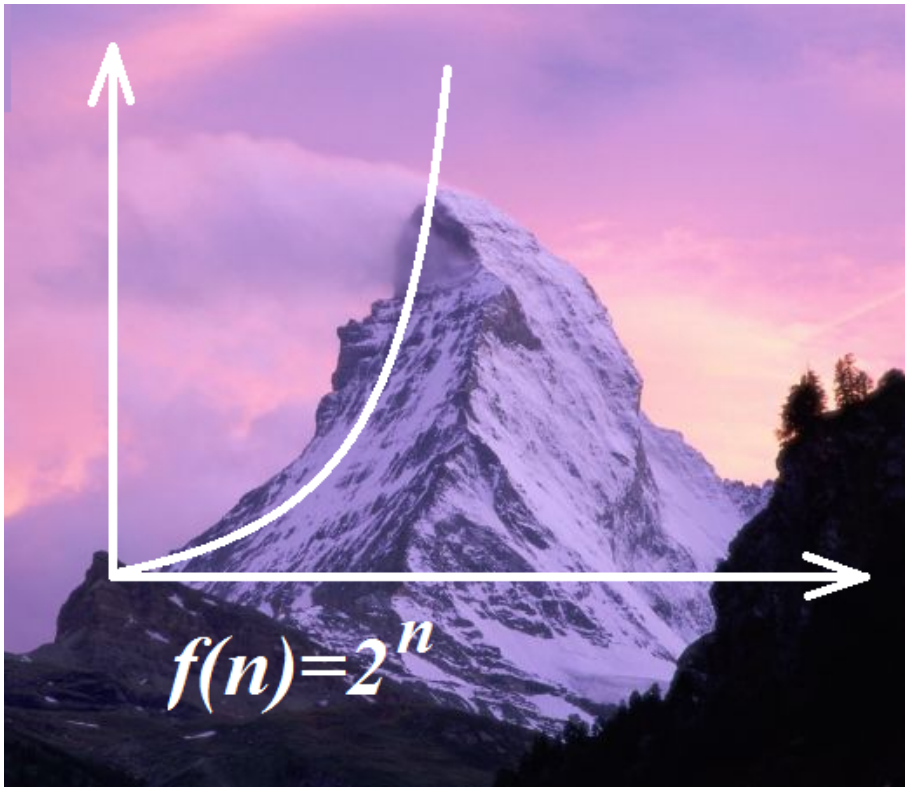
For any $c \geq 1$ and $d > 0$, we have

$$\log^c n \text{ is } O(n^d), \text{ but is not } \Omega(n^d)$$

Therefore $\log^{100} n$ is $O(n^{0.1})$, but is not $\Omega(n^{0.1})$.

Exponentials

Exponentials



Polynomials v.s. Exponentials

Fact.

For any $c > 0$, $d > 1$, we have

$$n^c \text{ is } O(d^n), \text{ but is not } \Omega(d^n)$$

Polynomials v.s. Exponentials

Fact.

For any $c > 0$, $d > 1$, we have

$$n^c \text{ is } O(d^n), \text{ but is not } \Omega(d^n)$$

Therefore n^{1000} is $O(2^n)$, but is not $\Omega(2^n)$.

Polynomials v.s. Exponentials

Fact.

For any $c > 0$, $d > 1$, we have

$$n^c \text{ is } O(d^n), \text{ but is not } \Omega(d^n)$$

Therefore n^{1000} is $O(2^n)$, but is not $\Omega(2^n)$.

Fact.

For any $c, d > 1$, we have

$$c^n \text{ is } O(d^n) \text{ if and only if } c \leq d$$

Polynomials v.s. Exponentials

Fact.

For any $c > 0$, $d > 1$, we have

$$n^c \text{ is } O(d^n), \text{ but is not } \Omega(d^n)$$

Therefore n^{1000} is $O(2^n)$, but is not $\Omega(2^n)$.

Fact.

For any $c, d > 1$, we have

$$c^n \text{ is } O(d^n) \text{ if and only if } c \leq d$$

Therefore 2^n is $O(3^n)$, but is not $\Omega(3^n)$.

Question

- $f_1(n) = n + 100$
- $f_2(n) = 4n^{1/2} + 3\log n$
- $f_3(n) = n^2 + 3n\log n$
- $f_4(n) = n^4 + \log^5 n$
- $f_5(n) = n + \log^3(2n)$
- $f_6(n) = 2^n + n^3$
- $f_7(n) = n^{1000} + 1000n^3$
- $f_8(n) = n^2\log n + n\log^2 n$

How do the above functions compare asymptotically?

Question

- $f_1(n) = n + 100$
- $f_2(n) = 4n^{1/2} + 3\log n$
- $f_3(n) = n^2 + 3n\log n$
- $f_4(n) = n^4 + \log^5 n$
- $f_5(n) = n + \log^3(2n)$
- $f_6(n) = 2^n + n^3$
- $f_7(n) = n^{1000} + 1000n^3$
- $f_8(n) = n^2\log n + n\log^2 n$

How do the above functions compare asymptotically?
Labwork.