COL106
Data Structures and Algorithms

Subodh Sharma and Rahul Garg

Introduction to Algorithms, Data Structures and Complexity Analysis

What is an Algorithm?

- A high-level description of computational process
- In an abstract notation, similar to a programming language
- Algorithm need not be always about computer programs
- A way of thinking to solve complex problems
- Decomposing solution into a series of small steps
- Has “real-life” applications
- Management, organization etc.

What is an Algorithm?

- A high-level description of computational process
- In an abstract notation, similar to a programming language
- Focusing on the essential elements of the computation
- Abstracting away unnecessary and obvious details
- We often write algorithms in the form of pseudo codes
  - A mixture of natural language and programming concepts
  - Describes the main ideas in a general manner and high-level manner
  - Implementation of a data structure and/or algorithm.

Finding Maximum of Elements in a List

Pseudo Code

```
function findmax(list)
        max ← -infinity
        for i = 1 to size(list)
                if (list[i] > max)
                        max ← list[i]
        end for
        return max
end function
```

Pseudocode vs. C Code

Pseudo Code

```
function findmax(list)
        max ← -infinity
        for i = 1 to size(list)
                if (list[i] > max)
                        max ← list[i]
        end for
        return max
end function
```

The code in C

```
int findmax(int *list, int size) {
        int max;
        max = (1 << 31)  // 32 bit int
        for (I = 0; I < size; i++) {
          if (list[i] > max)
                max = list[i];
        }
        return max;
}
```

Program

- Implementation of Algorithm in a programming language

Pseudocode implemented in Python

Pseudo Code

```
function findmax(list)
        max ← -infinity
        for i = 1 to size(list)
                if (list[i] > max)
                        max ← list[i]
        end for
        return max
end function
```

The code in Python

```
def findmax(list):
        max = -float('inf’)
        for item in list:
                if item > max:
                        max = item
        return max
```

Pseudocodes

- Expressions: Standard BODMAS notations
- Procedure/Functions
  - Defining functions or methods
  - Calling / invoking functions or methods
- Program control flow
  - if-then-else
  - Loops
    - for, while .. do, repeat .. until
- Variables (a), structures (a.name) and arrays (a[i])

Data Structures

- Logical organization of data to solve a given problem
- Why data structures?
  - Time Efficiency: Program takes less time
  - Encapsulation: Logical organization of elements of data
  - Space Efficiency: Program takes less memory
- Data structures implemented using a combination of abstract data types (ADTs)

What are Abstract Data Types (ADTs)?

- A high-level description of some data-types
- A set of operations of those data types
- Implementation plan
  - Class structure for implementation
  - Methods to implement the set of operations on the ADTs

Example: Unlimited Length Integers

What does the data type represent?

What are the possible operations on this data type?

Example: Unlimited Length Integers

What does the data type represent?

What are the possible operations on this data type?

Addition, subtraction, multiplication, division, integer division, remainder, comparison, increment, decrement, conversion from int, conversion to int, conversion from string, conversion to string, conversion from double, conversion to double….

Unlimited Length Integers Class Structure

Unlimited Length Integers Class Structure

```
class unlimitedInt {
private:
    int size; // Size of data
    unsigned int *data; // Actual values

public:
    unlimitedInt(int size) {
        data = malloc(sizeof (int) * size);
        // Need to check for malloc failure
    }
    class unlimitedInt add(unlimitedInt op1, unlimitedInt op2);
    class unlimitedInt multiply(unlimitedInt op1, unlimitedInt op2);
    ...
};
```

Algorithm

- A sequence of steps to carry out a computation
- Represented using easy to understand pseudo code notation
- Using a high-level description, while omitting unnecessary details
- Not only relevant for computer programs
- Has many real-life applications

Adding two Unlimited Length Integers

Example 2

- Counting the number of students in the class (parallel algorithm)
- How many students are there in this class?
- I take 3 seconds per student
- Total time taken?

Counting Number of Students in Class

- Can we count faster?

Counting Students: A Parallel Algorithm

- Every student will implement
- Ready?

Counting Students: A Parallel Algorithm

- Every student will implement
- Ready?
- Take out a sheet of paper and a pen
- Write your name and entry number on the top left of the paper

Algorithm: Parallel Count

```
function initialize()
     Stand up with a paper and a pen
     Write the following on your paper
          my_number = 1
          my_entry = <last five digits of
                           your entry number>
end initialize()
```

Main Step

```
function add_my_number()
     Partner with a standing student
     if (my_entry > partner.my_entry)
          my_number = my_number +
partner.my_number
     else {
          my_number = 0;
          Sit down
     }
end add_my_number()
```

Main loop

```
function main()
      initialize()
      while (I am standing AND
        there is a standing student)
      do
          add_my_number()
      end while
      if (I am standing)
           report my_number
      end if
end main()
```

Algorithm Parallel Count

```
function main()
        initialize()
        while (I am standing AND
          there is a standing student)
        do
            add_my_number()
        end while
        if (I am standing)
                report my_number
        end if
end main()
```

```
function initialize()
        Stand up with a paper and a pen
        Write the following on your paper
                my_number = 1
                my_entry = <last five digits
                         of your entry number>
end initialize

function add_my_number()
        Partner with a standing student
        if (my_entry > partner.my_entry)
                my_number = my_number +
                             partner.my_number
        else {
                my_number = 0;
                Sit down
        }
end add_my_number()
```

Need Eight Volunteers

[Balanced tree case]

Analysis: Parallel Time Complexity

[Unbalanced tree case]

Analysis: Parallel Time Complexity

Analysis: Parallel Time Complexity

Time taken is related to then height of the tree

Analysis: Time Complexity

Saliant points

- Different people may have different speeds
- We do not wish to find the exact time taken
- We wish to estimate the time taken
- As a function of the size of input
  - How the time taken grows as the input size increases
- Assume the size of input is N
- Time taken is T(N)

Asymptotic Analysis

Balanced Tree

$T(N) = \log_2(N)$

Unbalanced Tree

$T(N) = N - 1$

Plot and compare T(N) for different values of N

Asymptotic Analysis

Balanced Tree

$T(N) = 10 \log_2(N)$

Unbalanced Tree

$T(N) = N - 1$

Plot and compare T(N) for different values of N

Asymptotic Analysis

- Assume that you wish to make a Chat GPT
- You have N documents
- w_i words in the i^{th} document
- Total words W
- Suppose your algorithm take W^2 steps
- Computers today can execute C steps per second
- Will you ever be able to implement it?

Estimation of Chat GPT Runtime

- Estimate
  - N, $w_{i,}$ W, C
- Estimate W^2 / C
- Compare with another algorithm that takes 1000 W steps

Analysis: Time Complexity

Estimate T(N)

- But there can be multiple inputs of size N
- Runtime could be dependent on the specific input

T(N): maximum steps needed on all input of size N

It is difficult to estimate T(N)

We try to estimate an upper bound on T(N)

In a very specific way

Asymptotic Time Complexity: Big-O

- The big-O notation
- Asymptotic upper bound

Definition: A function $f(n)$ is $O(g(n))$ if there exists constants c, n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

f and g are functions defined over whole numbers

Asymptotic Time Complexity: Big-O

Let us look at some examples

- $f(n) = 1000 * n^2; g(n) = n^2$
- $f(n) = 1000 * n^2; g(n) = n^{1.5}$
- $f(n) = 10000 * n^2 + 300 * n^3; g(n) = n^3$
- $f(n) = 10000 * n^2 + 300 * n^3; g(n) = n^4$
- $f(n) = 10000 * n^2 + n^3 * \log(n); g(n) = n^3$
- $f(n) = 200 * n * \log(n); g(n) = n^{1.5}$
- $f(n) = n^2 \sin(2\pi\, n / T); g(n) = n^2$

Asymptotic Time Complexity: Big-O

$f(n) = 1000 * n^2$; $g(n) = n^2$

Asymptotic Time Complexity: Big-O

$f(n) = 1000 * n^2; g(n) = n^{1.5}$

Asymptotic Time Complexity: Big-O

$f(n) = 10000 * n^2 + 300 * n^3$; $g(n) = n^3$

Asymptotic Time Complexity: Big-O

$f(n) = 10000 * n^2 + 300 * n^3$; $g(n) = n^4$

Asymptotic Time Complexity: Big-O

$f(n) = 10000 * n^2 + n^3 * \log(n); g(n) = n^3$

Asymptotic Time Complexity: Big-O

$f(n) = 200 * n * \log(n)$; $g(n) = n^{1.5}$

Asymptotic Time Complexity: Big-O

$f(n) = n^2 \sin(2\pi\, n / T)$; $g(n) = n^2$

Asymptotic Time Complexity: Big-O

General rules of thumb

- Constants multipliers can be removed
- If polynomial terms are added, only need to take the largest power

Classes of Algorithm

- Logarithmic: $T(n)$ is $O(\log(n))$
- Linear: $T(n)$ is $O(n)$
- Quadratic: $T(n)$ is $O(n^2)$
- Polynomial: $T(n)$ is $O(n^k)$ for some $k \geq 1$
- Exponential $T(n)$ is $O(a^n)$ for some $a > 1$

Time Complexity Notations: Ω big-Omega and Θ (theta)

Definition: A function f(n) is Ω(g(n)) if there exists constants c, n_0 such that $f(n) \geq c\, g(n)$ for all $n \geq n_0$.

In other words f(n) is Ω(g(n)) if and only if g(n) is O(f(n))

Definition: A function f(n) is Θ(g(n)) if f(n) is O(g(n)) and f(n) is Ω(g(n))

In other words, there exist constants c_1, c_2 and n_0 such that $c_1\, g(n) \geq f(n) \geq c_2\, g(n)$ for all $n \geq n_0$

Time Complexity Notations: Little-o

Represents strict < asymptotic inequality

Definition: A function $f(n)$ is $o(g(n))$ if for every constant c, there is a n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Most Important Words in CS

- For all
- There exist
- Such that

Time Complexity Notations: Little-o

Represents strict < asymptotic inequality

Definition [little-o]: A function f(n) is o(g(n)) if for every constant c, there is a n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Definition [Big-O]: A function f(n) is O(g(n)) if there exists constants c, n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Little o vs. Big O

$f(n) = 1000 * n^2; g(n) = n^2$

$f(n) = 1000 * n^2; g(n) = n^{1.5}$

Definition [little-o]: A function $f(n)$ is $o(g(n))$ if for every constant c, there is a n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Definition [Big-O]: A function $f(n)$ is $O(g(n))$ if there exists constants c, n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Time Complexity Notations: Little-Omega

- Non-tight analogue of big-omega

Asymptotic Notations: Analogy

Functions Asymptotic Analysis	Numbers
f(n) is O(g(n))	f ≤ g
f(n) is Ω(g(n))	f ≥ g
f(n) is Θ(g(n))	f = g
f(n) is o(g(n))	f < g
f(n) is ω(g(n))	f > g

Now Let Us Count

Let Us Count

```
function main()
        initialize()
        while (I am standing AND
          there is a standing student)
        do
            add_my_number()
        end while
        if (I am standing)
                report my_number
        end if
end main()
```

```
function initialize()
        Stand up with a paper and a pen
        Write the following on your paper
                my_number = 1
                my_entry = <last five digits
                        of your entry number>
end initialize
```

```
function add_my_number()
        Partner with a standing student
        if (my_entry > partner.my_entry)
                my_number = my_number +
                             partner.my_number
        else {
                my_number = 0;
                Sit down
        }
end add_my_number()
```

Thank You

Unlimited Length Integers Class Structure

```
Class unlimitedInt {
private:
   int size; // Size of data
   unsigned int *data; // Actual values

public:
   unlimitedInt(int size) {
          data = malloc(sizeof (int) * size)
          // Need to check for malloc failure
   }
   class unlimitedInt add(unlimitedInt op1, unlimitedInt op2);
  class unlimitedInt multiply(unlimitedInt op1, unlimitedInt op2);
};
```