Introduction to C++ and Algorithm Analysis

CS 202 – Fundamental Structures of Computer Science

Bilkent University

Computer Engineering Department

Writing Programs

- In order to make a computer to do some work, you first design an algorithm.
- It is not enough that your algorithm works and functionally correct.
  - It should also practical in terms of run-time: For large input sizes, it should complete in a reasoble amount of time.
- There may be different algorithms that are solving the same problem, but they require much different time and space during run-time.
- Therefore, an algorithm should be designed for
  - 1) Operational correctness: It should solve the problem correctly.
  - 2) Time efficiency: It should solve the problem as quickly as possible.
  - 3) Space efficiency: It should requires reasonable amount of memory, disk space (computer system resources).
  - There may be trade-offs in achieving the goals 2) and 3)

Some Basic Mathematics Review

In computer science, all logarithms are to the base 2 unless specified otherwise.

$$\sum_{i=0}^{N} 2^{i} = 2^{N+1} - 1$$

More generally

$$\sum_{i=0}^{N} A^{i} = \frac{A^{N+1} - 1}{A - 1}$$

If $0 < A < 1$, then

$$\sum_{i=0}^{N} A^{i} \le \frac{1}{1 - A};\ \text{as n tends to } \infty,\ \sum_{i=0}^{N} A^{i} = \frac{1}{1 - A}$$

$$\sum_{i=0}^{N} \frac{i}{2^{i}} = 2$$

Some Basic Mathematics Review

$$\sum_{i=0}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$\sum_{i=0}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

C++ Classes

- In this course, we will write many data structures.
- We will use C++ to define and manipulate data structures.
- In C++, classes are used to define data structure and the operations (methods) that manipulate them

Class syntax

- A class consists of members
  - A member can be Data or Function.
- The functions are called member functions.
- Each instance of a class is an object.
  - Each object contains data components
  - The function of the class of the object are used to act (operate) on the data components.

Class syntax - example

```
            /* A class for simulating an integer memory cell */

class  IntCell
{
            public:
                        IntCell( )
                        { storedValue = 0; }

                        IntCell(int initialValue )
                        { storedValue = initialValue;}

                        int read( ) {
                        { return storedValue; }

                        void write( int x )
                                    { storedValue = x;}
            private:
                        int storedValue;
};
```

constructors

Class syntax

- Private members are not visible outside of the class (provides information hiding).
  - By use of private members the internal representation of data can be changes without changing the interface, hence without affecting other classes that make use of this class.
- Public members are visible to all other classes.
- Usually,
  - The data members are defined as private.
  - Member functions are defined as public.
- A constructor is a method
  - that has the same name with the class, and
  - that describes how an instance of the class (objects) is constructed.

  That may be more than one constructors defined.

Extra Constructor Syntax

```
          /* A class for simulating an integer memory cell */

class  IntCell
{
          public:
                    explicit IntCell( int initialValue = 0 )
                              : storedValue( initialValue) {}

                    int read( ) const {
                    { return storedValue; }

                    void write( int x )
                              { storedValue = x; }
          private:
                    int storedValue;
};
```


Extra Constructor Syntax - explanation

- Here, we are defining one constructor function that can be called either with or without parameter initialValue.
  - Thereby, we just define a single constructor as opposed to two constructors in the initial example.
  - If we omit the parameter in the call to the constructor, then the default value is used (which is 0 in this case).
- : storedValue(initialValue) is *the initialized list. Here we have just one element in the list*.
  - Sometimes it is mandatory to initialize data members of a class in the initializer list;
    - If the data member is const (can not be changed after object construction)
    - If the data member is of type some other class which has complex initialization.
    - The data member is of type some other calss which has not zero-parameter constructor.

Extra Constructor Syntax - explanation

- Explicit constructor
  - Is used for type checking at compile time.
  - All one parameter constructors should be defined explicit.

```
IntCell obj;   /* onj is an object of class IntCell */
obj = 37;      /* should not compile: type mismatch */
```

If there is not explicit, C++ compiler may convert the above code to the following for one-parameter constructor:

```
IntCell obj;

IntCell temporary = 37;
obj  = temporary;
```

Use of explicit make the compile to complain at the line: `obj = 37;`

Extra Constructor Syntax - explanation

- const keyword after the closing paranthesis of a member function is used:
  - To define a member function that can examine but not modify/change the state of its object.
  - These kind of member functions are called accessor.
  - Member functions that do change the state of its object called mutators.

Separation of Interface and Implementation

- It is sometimes useful the separate the definition of the interface of a class from the implementation of its members.
    - The interface remains same for a long time.
    - The function implementations can be modified more frequently.
    - The writers of other classes and modules have to only know the interfaces of classes.
- An interface lists the class and its members (data and function signatures).
- An implementation is coding of the member functions.

Separation of Interface and Implementation

- It is a good programming practice for large-scale projects to put the interface and implementation of classes in different files.
    - For small amount of coding it may not matter.
- A file that contains the interface of a class usually ends with .h (an include file)
- A file that contains the implementation of a class usually ends with .cpp (.cc or .C)
    - .c file includes the .h file with preprocessor command #include.
        - Example: #include<myclass.h>

Separation of Interface and Implementation

- In a big project, there will be a lot files (may be in the order of thousands), that may including other files.
  - There is a danger that an include file (.h file) may be read more than once during the compilation process.
    - It should be read once and only once to let the compiler learn the definition of the classes.
- To prevent a .h file to be read multiple times, we use preprocessor commands #ifndef and #define in the following way.

Separation of Interface and Implementation

```
#ifndef  _IntCell_H_
#define _IntCell_H_

class  IntCell
{
        public:
                explicit IntCell( int initialValue = 0 )
                        : storedValue( initialValue) {}

                int read( ) const;
                void write( int x );
        private:
                int storedValue;
};
#endif
```

Interface in *IntCell.h* file

Separation of Interface and Implementation

```
#include "IntCell.h"

explicit IntCell( int initialValue) : storedValue( initialValue) {}

int IntCell::read( ) const {
{
        return storedValue;
}

void IntCell::write( int x )
{
        storedValue = x;
}
```

Implementation in *IntCell.cpp* file

Separation of Interface and Implementation

```
#include "IntCell.h"


int main()
{
        IntCell m; /* or IntCell m(0); */
        m.write (5);
        cout << "Cell content : " << m.read << endl;

        return 0;
}
```

A program *TestIntCell.cpp* that uses IntCell class. We only include the Interface of the class.

Object declaration

Similar to primitive types.

```
int main()
{
        /* correct declarations */
        IntCell m1;
        IntCell m2 ( 12 );

        /* incorrect declarations */
        Intcell m3 = 37;    /* constructor was defined explicit:
                               meaning that when you declare an
                               object using this constructor you have
                               to call the constructor with parenthesis like
                               m3( 37 );
        Intcell m4();    /* this is a function declaration, not object!*/}
```


Algorithm Analysis

What is an algorithm

- Clearly specified set of simple instructions to be followed to solve a problem.
- Once you have a correct algorithm for a problem, you have determine how much resource (time and space) the algorithm will require.
- Now we will focus:
  - How to estimate the time required for an algorithm (program)
  - How to reduce the time required

Mathematical Background

- Analysis required to estimate the resource use of an algorithm is generally a theoretical issue.
  - A formal framework is required.
- Definitions:
  - DEFINITION: T(N) = O(f(N)) if there are positive constants c and n_0 such that T(N) <= cf(N) when N >= n_0
  - DEFINITION: T(N) = Ω(g(N)) if there are positive constants c and n_0 such that T(N) >= cg(N) when N >= n_0
  - DEFINITION: T(N) = θ(h(N)) if and only if T(N) = O(h(N)) and T(N) = Ω(h(N)).
  - DEFINITION: T(N) = o(p(N)) if T(N) = O(p(N)) and T(N) ≠ θ(h(N)).

- The running time of an algorithm is expressed with function T(N).
  - N is the input size.
- The bound is given with f(N)
- We say that T(N) is O(f(N)).
  - T(N) = O(f(N))
  - f(N) is an upper bound for the running time for sufficiently big N.
- Examples:
  - T(N) = 1000N = $O(N^2)$ (correct)
  - T(N) = 1000N = O(N) (better)
  - T(N) = 1000N = θ(N) (tight bound expression)

Rules

- If $T_1(N)$ = O(f(N)) and $T_2(N)$ = O(g(N)), then
  - a) $T_1(N) + T_2(N)$ = max(O(f(N)), O(g(N)))
  - b) $T_1(N) * T_2(N)$ = O(f(N)) * O(g(N))
- If T(N) is a polynomial of degree k then T(N) = $\theta(N^k)$.
- $\log^k N = O(N)$ for any constant k.

Common Growth Rates

Function	Growth Rate Name
C	Constant
log N	Logarithmic
$\log^2 N$	Log-squared
N	Linear
N log N	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Computation Model

- Before analyzing an algorithm, it is important over what kind of machine the algorithm will run (computer, parallel machine, ….)
- We will assume that the algorithms we will design will be running on a computer
- The computation model in this case is:
  - Computer has standard set of basic instructions (add, mulyiply, …) and algorithms are using them to do a job.
  - All instructions take one unit of time.
  - No fancy basic instructions such sorting which require more than one unit of time.
  - We assume infinite memory (since we want to focus on running time).
  - We have fixed size integers (32 bit).

What to analyze

- Given the computation model
- Given the input size (N)
- Compute for an algorithm (as part of algorithm analysis)
  - Average running time for the algorithm for inputs of size N: $T_{avg}(N)$. (reflecst the typical behaviour of the algorithm)
  - Worst-case running time for the algorithm for inputs of size N: $T_{worst}(N)$ (reflects a guarantee on the performance)
  - Best case running time for the algorithm for inputs of size N: $T_{best}(N)$

$T_{best}(N)$, <$T_{avg}(N)$, $T_{worstt}(N)$

What to analyze

- $T_{avg}(N)$ reflect the typical behavior of the algorith,
- $T_{worst}(N)$ reflects a guarantee for performance on any possible input.
- Generally we will be interested in computing (or estimating) the worst case running time $T_{worst}(N)$.
  - It is much difficult to compute the average running time.
  - Sometimes, the definition of average may also be nor very clear.

Running Time Calculations

- Given a set of algorithms that solve a problem, we want to figure which one is better.
  - We want to eliminate bad ones.
  - We want to find out the bottlenecks, so that we can be very careful in coding these parts very efficiently.
- There is no particular units of time in our calculations
- We will throw away the following from the running time estimations (bounds)
  - Leading constants: O(7N) → O(N)
  - Low-order terms: $O(N^3 + N^2) \rightarrow O(N^3)$.
- In big-Oh running estimation, overestimation is OK, but we should never underestimate the running time.

Example

```
int sum( int n )
{
        int partialSum;                      → no time

        partialSum = 0;                      → 1 unit
        for (int i = 1; i <=n;  i++)         → 1 + (N+1) + N units
                partialSum += i * i * i;     → 4 units
        return partialSum;                   → 1 unit
}
```

T(N) = 1 + 1 + (N+1) + N + N*(4) + 1 = 6N + 4 = **0(N)**

So our running time estimate is O(N).

General Rules for estimation

- **For loops**: The running time of for loops is at most the running time of the statements inside for loop times the number of iterations.
- **Nested Loops**: Running time of nested loops containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.
- **Consecutive Statements:** Just add the running times.
- **If/Else**: never more than the running of the test plus the larger of running times of S1 and S2.

Recursion

```
long fib( int n )
{
        if ( n <= 1 )
                return 1;
        else
                return fib( n-1 ) + fib( n-2 )
}
```

T (N) = T (N-1) + T(N-2) + 2
Solving this recurrence yields that T(N) grows exponentially.

Max Subsequence Problem

Given (possibly negative) integers $A_1, A_2, \ldots, A_N$,

find the maximum value of $\sum_{k=i}^{j} A_k$

For convenience, the maximum subsequence sum is 0 if all integers are negative.

Example :
For input -2, 11, -4, 13, -5, -2, the answer is 20
(A_2 through A_4).

Algorithm 1

```
int  maxSubSum1(const vector<int> & a)
{
        int maxSum = 0;

        for ( int i =0; i < a.size(); ++i) {
                for (int j = i; j < a.size(); j++)
                {
                        int thisSum = 0;

                        for (int k = i; k <=j; k++) {
                                thisSum += a[k];
                        }

                        if (thisSum > maxSum)
                                maxSum = thisSum;
                }
        return maxSum;
}
```

Buried inside 3 loops

Algorithm 1 - Analysis

- Running time is $O(N^3)$ due to lines shown previously (O(1)) that are bried inside 3 for loops.
  - For loop has size of N
  - Second loop has size of N-I (max value of N)
  - Third loop has size of j-i+1 (max value of N)
- Therefore, the upper bound is O(1 x N x N x N) = $O(N^3)$.

Algorithm 1 – more precise analysis

$$sum = \sum_{i=0}^{N} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1$$

$$\sum_{k=i}^{j} 1 = j - i + 1$$

$$\sum_{j=i}^{N-1} \sum_{k=i}^{j} 1 = \sum_{j=i}^{N-1} j - 1 + 1 = \frac{(N-i+1)(N-i)}{2}$$

$$\sum_{i=0}^{N} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1 = \sum_{i=1}^{N} \frac{(N-i+1)(N-i)}{2}$$

$$= \frac{N^3 + 3N^2 + 2N}{6} = \Theta(N^3)$$

Algorithm 2

```
int  maxSubSum2(const vector<int> & as)
{
        int maxSum = 0;

        for ( int i =0; i < a.size(); ++i) {
                int thisSum = 0;
                for (int j = i; j < a.size(); j++)
                {
                        thisSum += a[j];

                        if (thisSum > maxSum)
                                maxSum = thisSum;
                }
        return maxSum;
}
```


Algorithm 2 - Analysis

- We have 2 for loops.
- The statements inside the second for loop are executed $O(N^2)$ times and this is the biggest contribution to the running time.
- Therefore the running time is: $O(N^2)$
- There are two more algorithms in the book. You should study them.

Algoritm 3

```
int  maxSubSum3(const vector<int> & as)
{
        int maxSum = 0; thisSum = 0;

        for ( int j =0; j < a.size(); ++j) {

                thisSum += a[j];

                if ( thisSum > maxSum ) {
                        maxSum = thisSum;
                }
                else if ( thisSum < 0 ) {
                        thisSum = 0;
                }
                return maxSum;
}
```


Algorithm 3 - Analysis

- We have one for loop.
- The running time is O(N).