# 1.6.1 Performance Analysis

# 1.6.1.1 Space Complexity

## function to compute simple expression : program 1.12

### 설명: prog 1

## iterative function for sum: program 1.13



## recursive function for sum: program 1.14



## the space needed by each program (Reading Assignment)

### a fixed part

#### space for the code, space for simple variables and fixed-size component variables, space for constant

### a variable part

#### the space needed by component variables whose size is dependent on the particular problem instance,

#### the space needed by referenced variables(depend on instance),

#### the recursion stack space

## the space requirement S(P) of any program P

###  S(P) = c + Sp

#### need to determine which instance characteristics to use to measure the space requirements

# Example 1.6

## For program 1.12, Sp = 0

# Example 1.7

## For program 1.13, Ssum(n) = 0

### The problem instances are characterized by n

### The space needed by the function is independent of n

# Example 1.8

## For program 1.14, each call to rsum requires 4 words(n, a, the returned value, the returned address)

### the depth of recursion = n + 1

### the recursion stack space = 4(n+1)

#### n은 instance

# 1.6.1.2 Time Complexity

## the time, T(P) = the sum of the compile time and the run time

### the compile time : independent of the instance characteristics

## tp: the run time

### tp(n) = caADD(n) + csSUB(n) + cmMUL(n) + cdDIV(n) +

#### n denotes the instance characteristics

#### ca, cd denote the time needed for an addition, division, . . .

#### an impossible task to obtain an exact formula

## count only the number of program steps

### define a program step as a segment of a program (independent of the instance characteristics)

### return a+b+b\*c+ (a+b-c)/(a+b)+4.0; -> regarded as a step

## the complexity of various statements

### expressions: a step count of one

### assignments <variable> = <expr>

#### have a step count equal to that of <expr>

#### if the size of <variable> is a function of the instance characteristics

#### then the step count := the size of <variable> + the step count of <expr>

### iteration statements

#### consider the step counts only for the control part of loop statements

### switch statement, if-then-else, ....

## 1st method to determine the number of steps needed by a program

### introduce a new variable, count, into the program

# Example 1.10

## introduce a new variable count into program 1.14

## trsum(n) = the increase in the value of count when program 1.17 terminates

###  trsum(0) = 2

###  trsum(n) = 2 + trsum(n-1)

#### referred to as recurrence relations

 float rsum(float \*a, const int n)

 {

 *count++*; // +1

 if (n<= 0) {

 *count++*;

 return 0;

 }

 else {

 *count++*; // +1

 return (rsum(a, n-1) + a[n-1]); // trsum(n-1)

 }

 }

 Program 1.17: Program 1.14 with count statements added

####  trsum(0) = 2

####  trsum(n) = 2 + trsum(n-1) = 2\*2+ trsum(n-2)

####  = 2n + trsum(0) = 2n+2

#### expect the run time to grow linearly in n

#### the time complexity is linear

## 2nd method to determine the step count of a program

### determine the number of steps per execution of the statement and the total number of times(frequency) each statement is executed

### an important difference between the step count of a statement and its steps per execution (s/e)

#### the step count: not reflect the complexity of the statement

###  x = sum(a, m)

#### 1) a step count = 1

#### 2) the total change in count

####  = 1 + the change resulting from the invocation of sum: (2m+3)



### Table 1.2: Step table for Program 1.14

#### line 2(a) : the if condition of line 2

#### line 2(b): the statement in the if clause

#### line 3: 1+trsum(n-1), under the s/e(steps per execution)



### have to analyze the time complexity of the program fibonacci

#### should be able to solve the similar problems (recursions)

# 1.6.1.3 Asymptotic notation

## a difficult task to determine the exact step count

## two programs with a complexity of c1n2 + c2n and c3n respectively

### if c1 =1, c2 = 2, c3 = 100

### then c1n2 + c2n <= c3n for m<= 98

###  c1n2 + c2n > c3n for n>98

#### the break-even point : the value of n 98

# Def. of Big "oh"

### f(n) = O(g(n)) iff f(n) <= cg(n) for all n, n>= n0, constant c

#### g(n) is an upper bound on the value of f(n) for all n

#### not say anything about how good this bound is

##### f(n) = O(n2), f(n)=O(n2.5), f(n) = O(n3)

.

# Example 1.13

### 3n+2 = O(n) for all n>=2

### 10n\*\*2+4n+2 = O(n\*\*2) as

### 10n\*\*2+4n+2 <= 11n\*\*2

### 6\*2n+n\*n = O(2n) as

### 6\*2n+n\*n <= 7\*2n for n>=4

### 3n+3 = O(n2) as 3n+3 <= 3n\*\*2 for n>=2

### 10n\*\*2 + 4n+2 = O(n4)

### O(1): constant

### O(n): linear

### O(n2): quadratic, O(nlogn) is better than O(n2):

### O(2n): exponential

## f(n) = O(g(n))

### g(n) should be small (to be informative)

# Theorem 1.2:

## if f(n) = amnm+ ... + a1n + a0, then f(n) = O(nm)

# Def. of Omega

## f(n) = (g(n)) iff f(n) >= g(n)

# Example 1.14

### 3n+2 = (n) as 3n + 2 >= 3n for n>=1

### 10n\*\*2 +4n+2 =(n\*\*2)

### 3n+2 = (1)

### 6\*2n+n2 = (n)

### g(n) is only a lower bound on f(n)

#### g(n) should be as large a function of n as possible

# Theorem 1.3:

## if f(n) = amnm+ ... + a1n + a0,

##  then f(n) = (nm)

# Def. of Theta

## f(n) =(g(n)) iff c1g(n) <= f(n) <= c2g(n)

### g(n) is both an upper and lower bound on f(n)

# Example 1.15

### 3n+2 = (n)

### 10n\*\*2+4n+2 = (n\*\*2)

### 3n+2 != (1)

### 3n+2 != (n\*\*2)

# Theorem 1.4:

## if f(n) = amnm+ ... + a1n + a0, then f(n) = (nm)

## determine the asymptotic complexity of each statement and then add up these complexities

### Tab 1



##### line 3: 1 + trsum(n-1)

# Example 1.16[Permutation generator]

### assume that a is of size n

### the second for loop is entered n-k times

### each iteration of 2nd loop: (tperm (k+1, n-1))

### tperm(k, n-1) = ((n-k) tperm (k+1, n-1))

### when k < n-1

### tperm(0, n-1) = (n(n!)), n>=1

# 1.6.1.4 Practical Complexities

## complexity (n) is faster than complexity (n2) for sufficiently large n







### the time needed by a 1-billion-steps-per-second computer to execute a program of complexity f(n) instructions

## currently only the fastest computers can execute about 1 billion instructions per second

# 1.6.2 Perormance Measurement

## need a clocking function: time(hsec) that returns in the variable hsec the current time in hundredths of a second

## wish to measure the worst-case performance of the sequential search function(program 1.23)

### asymptotic analysis : (n)



## decide on the values of n for which the times are to be obtained

### need the times for more than two values of n

#### may not follow the asymptotic curve for smaller values of n

#### may not lie exactly on the predicted curve because low-order terms are discarded

## determine, for each of the above values of n, the data that exhibits the worst-case behavior

## exhibit worst-case behavior when x is chosen such that it is not one of the a[i]'s or set a[i] = i and x = 0





### inadequate precision of clock

### necessary to repeat to time a short event several times and divide the total time for the event by the number of repetitions



### r[j]: # of times the search is to be repeated when # of elements in the array is n[j]

# CHAPTER 2 ARRAYS

# 2.1 Abstraction Data Types and the C++ Class

## 중요 공부 내용

### ADT 정의할 수 있는 능력

### Representation

#### Efficiency를 위한 데이터 구조 표현

### 알고리즘

#### 성능 분석

# 2.1.1 An Introduction to the C++ Class

## the class

### support the distinction between specification and implementation

### hide the implementation of an ADT from its users

## four components of the C++ class

### a class name

### data members

### member functions

#### the set of operations that may be applied to the objects of a class

### levels of program access

#### three levels of access to class members: public, protected, private

## public data member(member function)

### can be invoked from anywhere in the program

## private data member(member function)

### invoked from within its class, or by a friend function, or a friend class

## protected data member(member function)

### invoked its class, or from its subclass, or by a friend



# 2.1.2 Data Abstraction and Encapsulation in C++

## data encapsulation is enforced

### by declaring all data members of a class to be private (or protected)

### external access to data memebers

#### achieved by defining member functions(public) that get and set data members

## separate the specification of the operations of a class from their implementation

### the specification -> program 2.1(Not yet ADT)

#### must be contained inside the public portion of the class definition

#### consists of the names of every public member function, the type of its arguments, and the type of its result

##### function prototype

### should be a description of what the function does

#### achieved in C++ by using comments

## place the specification of an operation in a named header file

### program 2.1 (placed in Rectangle.h)

## place the implementations of the functions in a source file of the same name



# 2.1.3 Declaring Class Objects and Invoking Member Functions

## declare class objects(instance objects) as variables

## invoke members of an object by using the component selection operators, a dot(.) and an arrow(->)



# 2.1.4 Special Class Operations

## constructor

### a member function which initializes data members of an object

### automatically executed when an object of that class is created

### if not define a constructor for a class

#### allocate memory for the data members of a class object

#### not initialize the data members

### the advantage of defining constructors for a class

#### eliminate errors that result from accessing an undefined object

#### all class objects are well-defined as soon as they are created

### must be declared as a public member function of its class



###  Rectangle r(1, 3, 6, 6);

###  Rectangle \*s = new Rectangle (0, 0, 3, 4);

###  Rectangle t;

##### undefined object

##### result in compile time error

###### the compiler requires a default constructor to initialize t

###### why: operator signature에 의한 binding

##### if not define a constructor

###### the compiler generate its default constructor

##### if users define a constructor, then user's responsibility to provide a default constructor for Rectangle t



#### program 2.4와의 차이

##### body empty

##### use member initialization list

#### initialize the data members by using a member initialization list

## destructor

### a member function which deletes data members before the object disappears

### automatically invoked when a class object goes out of scope or a class object is deleted

### must be declared as a public member of its class

### if a destructor is not defined for a class, then free memory associated with data members of the class for deleting an object of the class

#### if a data member is a pointer to some other object, then the object that it was pointing to is not deleted

# operator overloading

## allow the user to overload operators for user-defined data types

### done by providing a definition that implements the operator for the particular data type

### defined in the form of a class member function or an ordinary function



#### the C++ keyword this

##### represent a pointer to the particular class object that invoked it, when used inside a member function of a class

본 강의 자료의 그림 및 알고리즘 발췌

저자 : HOROWITZ

타이틀 : FUNDAMENTALS OF DATA STRUCTURES IN C++ 2nd Edition (2006)

공저 : SAHNI, MEHTA

 출판사 : Silicon Press