Basic Concepts

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

Overview: System Life Cycle

- Tools and techniques necessary to design and implement large-scale computer systems
  - Data abstraction
  - Algorithm specification
  - Performance analysis and measurement
  - Recursive programming
- The system life cycle -- the development process of programs; five highly interrelated phases

Overview: System Life Cycle (contd.)

- Requirements
  - Describing the information that we are given (input) and the results that we must produce (output)
- Analysis
  - Breaking the problem down into manageable pieces
  - Bottom-up & top-down
- Design
  - The creation of abstract data types
  - The specification of algorithms and a consideration of algorithm design strategies
  - Coding details are ignored!

Overview: System Life Cycle (contd.)

- Refinement and coding
  - Choosing representations for our data objects and writing algorithms for each operation on them
- Verification
  - Correctness proofs
    - The same techniques used in mathematics; time-consuming
  - Testing
    - Good test data should verify that every piece of code runs correctly.
  - Error removal
    - The ease with which we can remove errors depends on the design and coding decisions made earlier.

Algorithm Specification

- Definition: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task and must satisfy the following criteria:
  - Input
  - Output
  - Definiteness
  - Finiteness
  - Effectiveness

Algorithm Specification (contd.)

- cf. a program
  - A program does not have to satisfy finiteness condition.
- How to describe an algorithm?
  - In a natural language
    - No violation of definiteness is allowed.
  - By flowcharts
    - Working well only if the algorithm is small and simple

Algorithm Specification (contd.)

- Example: Selection Sort (p. 5)
  - Description statements; not an algorithm

    From those integers that are currently unsorted, find the smallest and place it next in the sorted list.
  - A selection sort algorithm (p. 5, Program 1.1)

```
for (i=0; i<n; i++) {
  Examine list[i] to list[n-1] and
  suppose that the smallest integer is
  at list [min];
  Interchange list[i] and list[min];
}
```

Algorithm Specification (contd.)

- Example: Binary search (p. 6)
  - Given a sorted array *list* with $n \geq 1$ distinct integers, figure out if an integer *searchnum* is in list.
  - Binary search algorithm (p. 6, Program 1.4)

```
while (there are more integers to check) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
    right = middle - 1;
  else if (searchnum == list[middle])
         return middle;
       else left = middle + 1;
}
```

Recursive Algorithms

- Direct recursion
  - Functions call themselves.
- Indirect recursion
  - Functions may call other functions that invoke the calling function again.
- Any function that we can write using assignment, **`if-else`**, and **`while`** statements can be written recursively.
  - Easier to understand

Recursive Algorithms (contd.)

- When should we express an algorithm recursively?
  - The problem itself is defined recursively.
  - Example: factorials, Fibonacci numbers, and binomial coefficients
- Example: Binary search
  - Recursive version (p. 11, Program 1.7)

```
int binsearch(int list[], int searchnum, int left,
                                        int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
   int middle;
   if (left <= right) {
       middle = (left + right)/2;
       switch (COMPARE(list[middle], searchnum)) {
           case -1: return
               binsearch(list, searchnum, middle + 1, right);
           case 0 : return middle;
           case 1 : return
               binsearch(list, searchnum, left, middle - 1);
       }
   }
   return -1;
}
```

Program 1.8: Recursive implementation of binary search

Data Abstraction

- Definition: A *data type* is a collection of objects and a set of operations that act on those objects.
  - Example: **`int`** and arithmetic operations
- All programming languages provide at least a minimal set of predefined data types, plus the ability to construct *user-defined types*.
- Knowing the representation of the objects of a data type can be useful and dangerous.

Data Abstraction (contd.)

- Definition: An *abstract data type* (*ADT*) is a data type whose specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.
- Specification vs. Implementation (of the operations of an ADT)
  - The former consists of the names of every function, the type of its arguments, and the type of its result.

Data Abstraction (contd.)

- Categories of functions of a data type
  - Creator/constructor
  - Transformers
  - Observers/reporters
  - Example: p. 17, Structure 1.1

Performance Analysis

- Criteria of performance evaluation can be divided into two distinct fields.
  - *Performance analysis* -- Obtaining estimates of time and space that are machine-independent
  - *Performance measurement* -- Obtaining machine-dependent times

Performance Analysis -- Space Complexity

- Definition: The *space complexity* is the amount of memory that it needs to run to completion.
- Equal to the sum of the following components
  - Fixed space requirements
    - Do not depend on the number and size of the program's inputs and outputs
    - Including the instruction space, space for simple variables, fixed-size structured variables, and constants

Performance Analysis -- Space Complexity (contd.)

- Variable space requirements
  - The space needed by structured variables whose size depends on the particular instance, I, of the problem and the additional space required when a function uses recursion
  - $S_P(I)$: The variable space requirement of a program P working on an instance I
    - Usually a function of some characteristics of the instance I
      - The number, size, and values of the inputs and outputs associated with I

- The total space requirement $S(P)$
  - $S(P) = c + S_P(I)$, where c is a constant representing the fixed space requirements

Performance Analysis -- Time Complexity

- The time, $T(P)$, taken by a program P is the sum of its *compile time* and its *run/execution time*.
  - Compile time
    - Similar to the fixed space component
    - Does not depend on the instance characteristics
  - Execution time T_P
    - Machine-independent estimate
    - Counting the number of operations performed in P
    - A problem: How is P divided into distinct steps?

Performance Analysis -- Time Complexity (contd.)

- Definition: A *program step* is a syntactically meaningful program segment whose execution time is independent of the instance characteristics.
  - The amount of computing represented by one program step may be different from that represented by another step.
- How to determine the number of steps?
  - Creating a global variable (p.23~25)
  - A tabular method (p.26~27)

Performance Analysis -- Time Complexity (contd.)

- The *best case step count*
  - The minimum number of steps that can be executed for the given parameters
- The *worst case step count*
  - The maximum number of steps that can be executed for the given parameters
- The *average step count*
  - The average number of steps executed on instances with the given parameters

Performance Analysis -- Asymptotic Notation (O, Ω, Θ)

- Because of the inexactness of what a step stands for, the exact step count isn't very useful for comparative purposes.
- Definition: $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all n, $n \ge n_0$.
  - p.31, Example 1.15
  - $O(1)$ ⇨ constant computing time, $O(n)$ ⇨ linear, $O(n^2)$ ⇨ quadratic, $O(2^n)$ ⇨ exponential

Performance Analysis -- Asymptotic Notation (O, Ω, Θ) (contd.)

- $f(n) = O(g(n))$ only states that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$ instead of implying how good this bound is.
  - So, $n = O(n^2)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, etc.
  - To be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$.
- Theorem 1.2: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$.
  - <proof> p. 31

Performance Analysis -- Asymptotic Notation (O, Ω, Θ) (contd.)

- Definition: $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.
  - To be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true.
- Theorem 1.3: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Performance Analysis -- Asymptotic Notation (O, Ω, Θ) (contd.)

- Definition: $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$.
- Theorem 1.4: If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.
- Example: Figure 1.5, p. 33
  - Since the number of lines is a constant, then we can take the maximum of the line complexities as the asymptotic complexity of the function