CIS 5511. Programming Techniques

Basic Concepts

1. Algorithm and data structure

Similar to a mathematical problem, a *computational problem* is normally defined as a mapping (or function) from an *input domain* to an *output range* (*codomain*), with a specific *input/output relationship*.

An *algorithm* is a procedure consisting of computational steps that transforms an *input value* into an *output value*, where

- the steps are directly executable,
- the procedure is predetermined and has a finite-length description,
- the process will terminate for every valid input with an output.

Therefore, an algorithm provides a solution for the problem by specifying how to actually accomplish the mapping.

For example, "sorting" is a problem where the input is a sequence of items of a certain type, with a total order defined between any pair of them, and the output should be a sequence of the same items, arranged according to the order. A sorting algorithm should specify, step by step, how to turn any valid input into the corresponding output in finite time, then stop (halt) there.

An algorithm is *correct* if for every valid input instance, it halts with the output as specified in the problem. An algorithm is incorrect if there is a valid input instance for which the algorithm produces an incorrect answer or no answer at all (i.e. does not halt).

For a given problem, if there are multiple candidate algorithms, which one should be used? There are several factors to be considered:

- correctness
- time and space efficiency
- conceptual simplicity

They are usually considered in the above order, though very often a compromise is needed among these factors.

When the (input, output, or intermediate) data of a problem contain multiple components, they are usually organized in a *data structure*, which represents both the *data items* and the *relations* among them.

A data structure can be specified either *abstractly*, in terms of the operations (as computations) that can be carried out on it, or *concretely*, in terms of the storage

organization and the algorithms accomplishing the operations. An abstract data structure often corresponds to multiple concrete ones.

In the design and selection of data structures, the analysis of the algorithms involved is a central topic.

Programming means to code an algorithm in a computer language. A program is *language-specific*, while an algorithm is *language-independent*.

2. Time efficiency analysis

Traditionally, *algorithm analysis* has been focused on the time efficiency of (correct) algorithms, though space efficiency and the correctness of an algorithm can also be analyzed.

For a given program, the actual time it takes to solve a given problem instance depends on

- the algorithm implemented in the program,
- the given problem instance,
- the software and hardware in which the program is executed.

Since the aim of algorithm analysis is to compare algorithms, the other factors need to be somehow removed from the discussion.

As a starting point, the following instructions are usually assumed to be directly executable and each takes a constant amount of time.

- **arithmetic**: add, subtract, multiply, divide, remainder, floor, ceiling;
- **comparison**: equal to, larger than, smaller than;
- **data movement**: load, store, copy, assign;
- **control**: conditional and unconditional branch, subroutine call and return.

More complicated blocks, such as loops, can be built from the above instructions, though the time taken by a block is not necessarily a constant anymore.

To measure the time complexity of algorithms, the common practice is to define a *size* (usually using *n*) for each instance of the problem (which intuitively measures the relative difficulty of processing the instance), then to represent the running time as a *function* of this instance size (as *T(n)* in the following). Finally, the increasing rate of the function, with respect to the increasing of the instance size, is used as the indicator of the efficiency or complexity of the algorithm.

With such a procedure, algorithm analysis becomes a mathematical problem, which is well-defined, and the result has universal validity.

Though it is a powerful technique, we need to keep in mind that many relevant factors have been ignored in this process, and therefore, if for certain reason some of the factors have to be taken into consideration, the traditional algorithm analyzing approach may become improper to use.

3. Conventions about pseudocode

To be independent of programming languages, the algorithms in the textbook are written in *pseudocode*, according to the following conventions (the 4th edition and the 3rd edition of the textbook are slightly different):

- The looping and the conditional constructs are similar to those in C/Java/Python/ Pascal.
- Indentation indicates block structure, without delimiters.
- Assignment sign is equal sign, "=", and multiple assignment is allowed.
- Double equal sign, "==", is for equality.
- The boolean operators *and* and *or* are *short circuiting*.
- Variables are defined within an algorithm and do not require declaration.
- Array elements are represented as *A[index]*, and index usually starts at 1. *A[i:j]* is a subarray consisting *A[i]* ... *A[j]*.
- A field in an object is represented as *object.field* (array length is *Array.length* in the 3rd edition).
- Parameters are passed to a procedure by value.
- A return statement transfers control back to the calling procedure.
- Double slash ("//") is used for comments.
- Lines are numbered.

Your algorithms in homework and exam should follow the above format as closely as possible.

4. Example: sort using an incremental approach

Sorting: the problem and its input/output.

Insertion sort works by repeatedly insert an element into the sorted part of the array.

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

The correctness of the algorithm is proven by checking the *loop invariant*, which is a proposition about a relation among certain variables, such as

> *At the start of each iteration of the for loop of line 1-8, the subarray A[1 : i–1] consists of the elements originally in A[1 : i–1] but in sorted order.*

We must show three things about a loop invariant:

- **Initialization**: It is true prior to the first iteration of the loop.
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination**: When the loop terminates, the invariant helps showing the correctness of the algorithm.

These properties hold for the above loop invariant in the insertion sort algorithm.

For sorting, it is natural to use the number of keys to be sorted as input size, and it is assumed that a constant amount of time is required to execute each line of the pseudocode (except comments).

Now let us mark the cost and the number of execution times of each line:

INSERTION-SORT(A, n)		*cost*	*times*
1	**for** $i = 2$ **to** n	c_1	n
2	$key = A[i]$	c_2	$n-1$
3	// Insert $A[i]$ into the sorted subarray $A[1 : i-1]$.	0	$n-1$
4	$j = i-1$	c_4	$n-1$
5	**while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6	$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8	$A[j+1] = key$	c_8	$n-1$

In line 5-7, t_i is the number of executions of Line 5 for the i value.

The running time of the algorithm is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

For a given n, T(n) depends on the values of t_j, which changes from instance to instance.

For a given size n, the best case of the algorithm happens when the array is already sorted, so that $t_i = 1$ for i = 2, 3, ..., n, and the function becomes

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

which is a linear function of n.

The worst case of the algorithm happens when the array is reverse sorted, so that t_i = i

for i = 2, 3, ..., n, and the function becomes

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

which is a quadratic function of n.

Usually, analysis of algorithm concentrates on the worst case.

5. Example: sort using a divide-and-conquer approach

The divide-and-conquer approach solves a problem by

1. divide the problem into a number of subproblems,
2. conquer the subproblems,
3. combine the solutions to the subproblems into the solution for the original problem.

If a subproblem is of the same type but a smaller size, it can be solved in the same way, which leads to a *recursive* solution, in which the algorithm calls itself until the problem size is small enough to be directly solved.

Merge sort is an algorithm that sorts an array by cutting it into two halves, sorting them recursively, then merging the two sorted subarrays.

The merge procedures Merge(A, p, q, r) first moves the two sorted subarrays A[p : q] and A[q+1 : r] into two separate arrays L and R, put two special sentinel values at the end of each of them, then merge the two back into the original array A[p : r].

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Its time expense is a linear function of n, because every data item is moved 2 times, and compared less than 1 time in average.

The following is the merge-sort algorithm.

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

The correctness and efficiency of this algorithm can be analyzed similarly.