Algorithms & Data Structures (M)

Tutorial Exercises

These exercises have been selected (and in some cases adapted) from the *Java Collections* textbook. Harder exercises are marked * or **.

Sample solutions will be posted at the course's Moodle site. *Attempt each exercise before consulting the sample solution.*

Exercises 1 (Introduction)

1A Use Euclid's GCD algorithm to find the GCDs of the following pairs of numbers: 6 and 9; 12 and 18; 15 and 21; 11 and 15.

1B Consider Newton's square-root algorithm:

> To compute approximately the square root of a positive real number a:
>
> 1. Set r to the mean of 1 and a.
> 2. While r^2 is not a good enough approximation to a, repeat:
>   2.1. Set r to the mean of r and a/r.
> 3. Terminate with answer r.

(a) Use this algorithm to calculate the square roots of the following numbers: 4, 6, 8, 9. In each case, calculate your answer to an accuracy of two decimal places, i.e., the absolute difference between a and r^2 should be less than 0.01.

(b) Write a Java program to implement the algorithm and use it to check your answers to part (a) above.

(c) What would happen if step 2 of the algorithm were simply "While r^2 is not equal to a, repeat:"?

1C Give some examples of algorithms used in everyday life, not requiring a calculator or computer.

1D Devise an algorithm to find the roots of the general quadratic equation $ax^2 + bx + c = 0$. The roots are the two values of the formula $(-b \pm \sqrt{(b^2 - 4ac)}) / 2a$.

Exercises 2 (Algorithms and Complexity)

2A Hand-test the simple and smart power algorithms. Use the test case $b = 2$, $n = 11$. How many multiplications are performed by each algorithm?

2B What is the time complexity of the midpoint algorithm (course notes §1)?

2C Create a spreadsheet to reproduce the table of growth rates (course notes §2), and extend it to $n = 100$.

2D The following Java methods implement matrix addition and multiplication. Each matrix is represented by an $n \times n$ 2-dimensional array of **`float`** numbers.

```
static void matrixAdd (
            int n, float[][] a,
            float[][] b, float[][] sum) {
// Set sum to the sum of the n×n matrices a and b.
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      sum[i][j] = a[i][j] + b[i][j];
    }
  }
}
static void matrixMult (
            int n, float[][] a,
            float[][] b, float[][] prod) {
// Set prod to the product of the n×n matrices a and b.
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      float s = 0.0;
      for (int k = 0; k < n; k++) {
        s += a[i][k] * b[k][j];
      }
      prod[i][j] = s;
    }
  }
}
```

Analyze these methods in terms of the number of **`float`** additions and multiplications performed. What is the time complexity of each method?

2E Analyze the time complexity of the recursive integer rendering algorithm.

2F Devise a non-recursive algorithm to print a given integer i to base r. What is your algorithm's time complexity?

2G The factorial of a positive integer n is $n \times (n-1) \times \ldots \times 2 \times 1$. A factorial can be calculated using the following recursive algorithm:

To calculate the factorial of n:

1. If $n = 0$:
   1.1. Terminate with answer 1.
2. If $n \neq 0$:
   2.1. Let f be the factorial of $n-1$.
   2.2. Terminate with answer $(n \times f)$.

(a) What is the time complexity of this algorithm?

(b) Devise a non-recursive version of this algorithm.

(c) Implement both algorithms as Java methods.

2H Write a Java program to implement the Towers of Hanoi algorithm, printing out the moves as it runs. Use your program to count the number of moves required, and thus verify the time complexity of the algorithm.

Exercises 3 (The Array Data Structure)

3A Write an algorithm to test whether an array *a*[*left*…*right*] is sorted (i.e., in ascending order).

In terms of the number of comparisons required, determine the time efficiency of your algorithm: in the best case; in the worst case; and on average.

Implement your algorithm as a Java method, assuming that the array elements are `Comparable` objects.

3B A *palindrome* is a sequence that is identical to the reverse of itself. For example, the sequence «'m', 'a', 'd', 'a', 'm'» is a palindrome.

Write an algorithm to test whether a character array *a*[*left*…*right*] is a palindrome.

What are the time efficiency and space efficiency of your algorithm?

Implement your algorithm as a Java method.

3C Modify the algorithm of Exercise 3B to ignore spaces and punctuation, and to treat corresponding upper-case and lower-case letters as equivalent. For example, the sentence "Madam, I'm Adam." is a palindrome by this definition.

3D Devise array algorithms to solve the following problems, and analyze their time complexities:

(a) Delete *val* from an unsorted array *a*[*left*…*right*].

(b) Delete *val* from a sorted array *a*[*left*…*right*].

(c) Insert *val* in a sorted array *a*[*left*…*right*].

(d) Find the least component of an unsorted array *a*[*left*…*right*].

3E A simple "phone-book" can be represented by an array of `Contact` objects:

```
class Contact {
  public String name;
  public String number;
}
```

Assume that names are unique but phone-numbers are not unique. The following methods are to be implemented:

```
static String searchByName (
            Contact[] book,
            String targetName);
// Return the phone-number of the (unique) entry in book
// whose name is targetName, or null if there is no such
// entry.
static String[] searchByNumber (
            Contact[] book,
            String targetNumber);
// Return an array of all names in book whose phone-
// number is targetNumber, or null if there are no such
// entries.
```

Suppose that `searchByName` will be called frequently, but `searchByNumber` will be called only occasionally. How would you organize the directory entries? Implement the two methods accordingly.

3F A simple way to represent a *set* of words is by a sorted array of words with no duplicates. You are given two sets of words, s_1 and s_2, represented in this way. By modifying the array merging algorithm, devise algorithms for the following problems:

(a) Compute the *union* of s_1 and s_2. The union is the set of those words found in s_1 or s_2 or both.

(b) Compute the *intersection* of s_1 and s_2. The intersection is the set of those words found in both s_1 and s_2.

3G Consider the problem of reading a file of (unsorted) values into an array, where the array must be sorted. There are *n* values in the file.

(a) Write an algorithm to read all of the unsorted values into the array, and then sort the array using the selection sort algorithm. What is the time efficiency of your algorithm?

(b) Write an algorithm to read each value in turn, and insert it into a sorted array (initially empty). What is the time efficiency of your algorithm? How does this compare with your answer to part (a)?

(c) Implement your algorithms from parts (a) and (b), and compare them by timing their execution on files with a range of sizes.

*** 3H** The Dutch national flag problem is as follows. You are given an array of colors (reds, whites, and blues), in no particular order. Sort them into the order of the Dutch national flag (reds followed by whites followed by blues).

(a) Devise an efficient algorithm to solve this problem.

(b) What is your algorithm's time complexity?

3J You are given two unsorted arrays of values. You are required to obtain a sorted array containing all these values. Suggest *two* different ways of achieving this. Compare their time efficiency. (*Note:* Assume that suitable merging and sorting algorithms are already available.)

3K Devise an algorithm to solve the following problem. Given an array $a[0\ldots n-1]$, and a shorter array $b[0\ldots m-1]$, find the position of the leftmost subarray of *a* whose elements equal (pairwise) all the elements of *b*. In other words, if the answer is *p*, then $a[p]$ must equal $b[0]$, $a[p+1]$ must equal $b[1]$, …, and $a[p+m-1]$ must equal $b[m-1]$.

In the following example, the elements are colors, and the answer should be 3:

a: red, orange, yellow, green, blue, indigo, violet
b: green, blue, indigo

On the other hand, if $b[2]$ were violet, the answer should be *none*, indicating that there is no complete match. Likewise, if the colors in *b* were in a different order, the answer should be *none*.

What is the time complexity of your algorithm?

3L Consider the implementation and analysis of the selection-sort algorithm in the course notes. Modify the selection-sort method to count comparisons:

```
int sort (Comparable[] a,
          int left, int right);
// Sort a[left...right] into ascending order.
// Return the number of comparisons performed.
```

Use your method to sort arrays of length 10, 20, …, 60, and print

out the number of comparisons in each case. Do the numbers correspond to those predicted by the analysis of the selection-sort algorithm?

3M Consider the quick-sort method in the course notes. Test it, using a suitable implementation of the partitioning algorithm.

Then modify the quick-sort method to count comparisons:

```
int sort (Comparable[] a,
          int left, int right);
// Sort a[left...right] into ascending order.
// Return the number of comparisons performed.
```

Use your method to sort *randomly-ordered* arrays of length 10, 20, …, 60, and print out the number of comparisons in each case. Do the numbers correspond to those predicted by the analysis of the quick-sort algorithm?

Repeat with *already-sorted* arrays. What difference do you observe?

Exercises 4 (Linked-List Data Structures)

4A Devise an algorithm to access the k^{th} element of an SLL. What is your algorithm's time complexity?

4B Devise an algorithm to access the k^{th} element of a DLL. (*Hint:* You can do better than imitating your solution to Exercise 4A.) What is your algorithm's time complexity?

4C Devise an algorithm to reverse the elements of a SLL. What is your algorithm's time complexity and space complexity?

4D Devise an algorithm to reverse the elements of a DLL. What is your algorithm's time complexity and space complexity?

4E Devise an algorithm to check whether the elements of a SLL of characters is a palindrome. (See Exercise 3B for the definition of a palindrome).

What is the time efficiency of your algorithm?

Implement your algorithm as a Java method.

4F If an SLL is sorted, it is possible to speed up the SLL linear search algorithm. The algorithm traverses the SLL first-to-last, terminating if it reaches a node that contains an element equal to *or greater than* the target.

(a) Write this modified algorithm.

(b) How does this improvement affect the time efficiency of the linear search algorithm when the search is successful? when the search is unsuccessful?

(c) Write a Java method to implement this algorithm.

4G It is just as easy to search a DLL last-to-first as first-to-last. When might it be advantageous to do so? (*Hint*: consider a sorted DLL containing a large number of words.)

4H Devise a search algorithm that simultaneously searches from both ends of an unsorted DLL.

How many comparisons would this take to find a given element in the best case? in the worst case? on average?

What is the time complexity of this algorithm?

4J When repeatedly searching an unsorted SLL, it can be advantageous to move an item to the head of the list when it is found, if it is likely that the same item will be searched for again in the near future.

(a) Write this modified linear search algorithm.

(b) How does this modification affect the time efficiency of the algorithm when the same item is searched for 50 times out of the next 100 searches? (*Hint:* Consider the total time taken on average to perform all 100 searches with and without the modification.)

(c) Write a Java method to implement this algorithm.

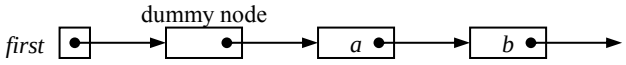
4K Devise algorithms to delete the node containing a given element *elem*:

(a) in an SLL;

(b) in a DLL.

4L Consider the SLL insertion and deletion algorithms.

These algorithms must check for the special case when we are inserting or deleting a node at the front of the SLL. These checks can be avoided if we create a *dummy node* at the front of the SLL, i.e., a node that does not contain an element. This ensures that even the first node in the SLL has a predecessor node. This is illustrated in the following diagram:



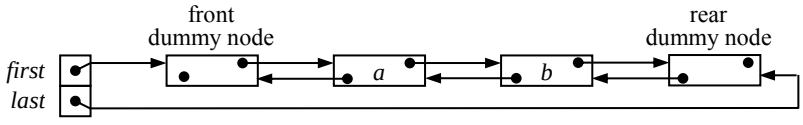
Even an empty SLL contains one node, i.e., the dummy node. When a new SLL is created, *first* is initialized with a link to the dummy node, and thereafter remains unchanged.

Modify the SLL insertion and deletion algorithms to use a dummy node. What changes are required to the other SLL algorithms?

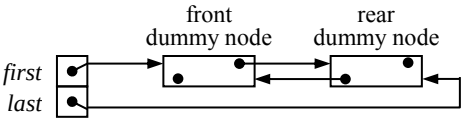
Implement a revised Java class for SLLs that uses your modified algorithms.

Compare the time efficiency of your modified algorithms with the original algorithms.

4M The DLL insertion and deletion algorithms can be improved along the lines of Exercise 4L by creating a dummy node at each end of the DLL. Call these the *front* and *rear* dummy nodes. The front dummy node's successor contains the first element of the DLL. The rear dummy node's predecessor contains the last element of the DLL. This is illustrated in the following diagram:



The empty DLL now contains just the two dummy nodes. When a new DLL is created, *first* is initialized with a link to a front dummy node, and *last* is initialized with a link to a rear dummy node. Both of these links remain unchanged thereafter. Initially the successor of the front dummy node is the rear dummy node, and the predecessor of the rear dummy node is the front dummy node. The empty DLL is illustrated in the following diagram:



Modify the DLL insertion and deletion algorithms to use dummy nodes. What changes are required to the remaining algorithms?

Implement a revised Java class for DLLs that uses your modified algorithms.

Compare the time efficiency of your modified algorithms with the original algorithms.