Algorithms & Data Structures (M)

Solutions to Tutorial Exercises

Here are sample solutions to most of the tutorial exercises. For some of the exercises (especially coding exercises), alternative solutions are possible.

Attempt each exercise before consulting the sample solution.

Solutions to Exercises 1

1A The GCD of 6 and 9 is 3; the GCD of 12 and 18 is 6; the GCD of 15 and 21 is 3; the GCD of 11 and 15 is 1.

1B Using Newton's algorithm to calculate the square root of a number accurate to two decimal places, the square root of 4 is 2.00; the square root of 6 is 2.45; the square root of 8 is 2.83; and the square root of 9 is 3.00.

Method to calculate the square root of `a` to 2 decimal places:

```
static float squareRoot (float a) {
   float r = (1 + a)/2;
   while (Math.abs(r*r - a) > 0.01)
      r = (r + a/r)/2;
   return r;
}
```

If step 2 of the algorithm continued while $r^2 \neq a$, the algorithm would be unlikely to terminate, since two approximately-computed numbers are unlikely to be exactly equal.

1D To find the (real) roots of the general quadratic equation $ax^2 + bx + c = 0$:

1. Let d be $b^2 - 4ac$.
2. If $d > 0$:
   2.1. Let r be the square root of d.
   2.2. Terminate with answers $(-b + r) / 2a$ and $(-b - r) / 2a$.
3. Else, if $d = 0$:
   3.1. Terminate with answer $-b / 2a$.
4. Else, if $d < 0$:
   4.1. Terminate with no answer.

Solutions to Exercises 2

2A In the test case $b = 2$, $n = 11$, the simple power algorithm performs 11 multiplications, while the smart power algorithm performs 7 multiplications.

2B The midpoint algorithm has time complexity $O(1)$.

2D The `matrixAdd` method performs n^2 additions. Its time complexity is $O(n^2)$.

The `matrixMult` method performs n^3 additions and n^3 multiplications. Its time complexity is $O(n^3)$.

2E To analyze the recursive integer rendering algorithm, count the number of characters required to render i to base r. If i is positive, the number of characters is $\log_r i + 1$. If i is negative, the number of characters is $\log_r(\text{abs}(i)) + 2$ (the extra character being '–'). The time complexity is $O(\log(\text{abs}(i)))$.

2F To print a given integer i to base r:

1. Set s to the empty string "".
2. Set p to the absolute value of i.
3. Repeat the following until $p = 0$:
   3.1. Let d be the digit corresponding to (p modulo r).
   3.2. Insert d at the front of s.
   3.3. Divide p by r.
4. If $i < 0$, insert '–' at the front of s.
5. Print s.
6. Terminate.

This algorithm's time complexity is $O(\log(\text{abs}(i)))$.

2G The factorial algorithm (recursive version) performs n multiplications. Its time complexity is $O(n)$.

Method to calculate the factorial of `n` (recursive version):

```
static int factorial (int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

To calculate the factorial of n (non-recursive version):

1. Set f to 1.
2. For $i = 1, \ldots, n$, repeat:
   2.1. Multiply f by i.
3. Terminate with answer f.

Method to calculate the factorial of `n` (non-recursive version):

```
static int factorial (int n) {
  int f = 1;
  for (int i = 1; i <= n; i++)
    f *= i;
  return f;
}
```

2H Outline of program:

```
static void moveTower (int n,
                       int source, int dest) {
  if (n == 1)
    moveDisk(source, dest);
  else {
    int spare = 6 - source - dest;
    moveTower(n-1, source, spare);
    moveDisk(source, dest);
    moveTower(n-1, spare, dest);
  }
}
static void moveDisk (int source, int dest) {
  System.out.println("Move disk from " + source
       + " to " + dest);
}
```

To make the program count the moves, modify `moveTower` to return the required number of moves, as follows:

```
static int moveTower (int n,
                      int source, int dest) {
  if (n == 1) {
    moveDisk(source, dest);
    return 1;
  } else {
    int spare = 6 - source - dest;
    int moves1 = moveTower(n-1, source, spare);
    moveDisk(source, dest);
    int moves2 = moveTower(n-1, spare, dest);
    return moves1 + 1 + moves2;
  }
}
```

Solutions to Exercises 3

3A To test whether the array *a*[*left*…*right*] is sorted in ascending order:

1. For *p* = *left*+1, …, *right*, repeat:
   1.1. If *a*[*p*–1] is greater than *a*[*p*], terminate with answer *false*.
2. Terminate with answer *true*.

The number of comparisons is between 1 and *n*–1, i.e., *n*/2 on average.

Method to test whether the array `a[left…right]` is sorted in ascending order:

```
static boolean isSorted (Comparable[] a,
                         int left, int right) {
   for (int p = left+1; p <= right; p++) {
      if (a[p-1].compareTo(a[p]) > 0)
         return false;
   }
   return true;
}
```

3B To test whether the character array *a*[*left*…*right*] is a palindrome:

1. Set *l* to *left*, and set *r* to *right*.
2. While *l* < *r*, repeat:
   2.1. If *a*[*l*] ≠ *a*[*r*], terminate with answer *false*.
   2.2. Increment *l* and decrement *r*.
3. Terminate with answer *true*.

The number of comparisons is between 1 and *n*/2, i.e., about *n*/4 on average. Therefore the algorithm's time complexity is *O*(*n*). Its space complexity is *O*(1).

Method to test whether the character array `a[left…right]` is a palindrome:

```
static boolean isPalindrome (char[] a,
                             int left, int right) {
   int l = left, r = right;
   while (l < r) {
      if (a[l] != a[r])  return false;
      l++;  r--;
   }
   return true;
}
```

3C To test whether the character array *a*[*left*…*right*] is a palindrome, ignoring spaces and punctuation:

1. Set *l* to *left*, and set *r* to *right*.
2. While *l* < *r*, repeat:
   2.1. If *a*[*l*] is a space or punctuation:
      2.1.1. Increment *l*.
   2.2. Else, if *a*[*r*] is a space or punctuation:
      2.2.1. Decrement *r*.
   2.3. Else (if neither *a*[*l*] nor *a*[*r*] is a space or punctuation):
      2.3.1. If *a*[*l*] ≠ *a*[*r*], terminate with answer *false*.
      2.3.2. Increment *l* and decrement *r*.
3. Terminate with answer *true*.

3D To delete *val* from the unsorted array *a*[*left*…*right*]:

1. For *p* = *left*, …, *right*, repeat:
   1.1. If *val* is equal to *a*[*p*]:
      1.1.1. Copy *a*[*p*+1…*right*] into *a*[*p*…*right*–1].
      1.1.2. Make *a*[*right*] unoccupied.
      1.1.3. Terminate.
2. Terminate.

To delete *val* from the sorted array *a*[*left*…*right*]:

1. For *p* = *left*, …, *right*, repeat:
   1.1. If *val* is equal to *a*[*p*]:
      1.1.1. Copy *a*[*p*+1…*right*] into *a*[*p*…*right*–1].
      1.1.2. Make *a*[*right*] unoccupied.
      1.1.3. Terminate.
   1.2. Else, if *val* is less than *a*[*p*], terminate.
2. Terminate.

To insert *val* in the sorted array *a*[*left*…*right*]:

1. For *p* = *right*, …, *left*, repeat:
   1.1. If *val* is less than *a*[*p*]:
      1.1.1. Copy *a*[*p*] into *a*[*p*+1].
   1.2. Else (if *val* is greater than or equal to *a*[*p*]):
      1.2.1. Copy *val* into *a*[*p*+1].
      1.2.2. Terminate.
2. Copy *val* into *a*[*left*].
3. Terminate.

(*Note:* This algorithm overwrites *a*[*right*+1], assuming that it exists.)

To find the least component of the unsorted array *a*[*left*…*right*]:

1. Set *least* to *a*[*left*].
2. For *p* = *left*+1, …, *right*, repeat:
   2.1. If *a*[*p*] is less than *least*, set *least* to *a*[*p*].
2. Terminate with answer *least*.

All these algorithms have time complexity $O(n)$.

3E The phone-book should be sorted by name, allowing the most frequently-called method `searchByName` to be implemented using binary search. Methods:

```
static String searchByName
               (Contact[] book,
                String targetName) {
  int l = 0, r = book.length - 1;
  while (l <= r) {
    int m = (l + r)/2;
    int comp =
         targetName.compareTo(book[m].name);
    if (comp == 0)
      return book[m].number;
    else if (comp < 0)
      r = m - 1;
    else  // comp > 0
      l = m + 1;
  }
  return null;
}
static String[] searchByNumber
               (Contact[] book,
                String targetNumber) {
  String[] names1 = new String[book.length];
  int count = 0;
  for (int p = 0; p < book.length; p++) {
    if (targetNumber.equals(book[p].number))
      names1[count++] = book[p].name;
  }
  if (count == 0)
    return null;
  else {
    String[] names2 = new String[count];
    System.arraycopy(names1, 0, names2, 0,
         count);
    return names2;
  }
}
```

3F To compute the union of *s1*[*l1*…*r1*] and *s2*[*l2*…*r2*] in *s3*[*l3*…*r3*]:

1. Set *i* to *l1*, set *j* to *l2*, and set *k* to *l3*.
2. While $i \le r1$ and $j \le r2$, repeat:
   2.1. If *s1*[*i*] is equal to *s2*[*j*]:
      2.1.1. Copy *s1*[*i*] into *s3*[*k*], then increment *i*, *j*, and *k*.
   2.2. Else, if *s1*[*i*] is less than *s2*[*j*]:
      2.2.1. Copy *s1*[*i*] into *s3*[*k*], then increment *i* and *k*.
   2.3. Else (if *s1*[*i*] is greater than *s2*[*j*]):
      2.3.1. Copy *s2*[*j*] into *s3*[*k*], then increment *j* and *k*.
3. If $i \le r1$, copy *s1*[*i*…*r1*] into *s3*[*k*…*r3*].
4. If $j \le r2$, copy *s2*[*j*…*r2*] into *s3*[*k*…*r3*].
5. Terminate.

To compute the intersection of *s1*[*l1*…*r1*] and *s2*[*l2*…*r2*] in *s3*[*l3*…*r3*]:

1. Set *i* to *l1*, set *j* to *l2*, and set *k* to *l*.
2. While $i \le r1$ and $j \le r2$, repeat:
   2.1. If *s1*[*i*] is equal to *s2*[*j*]:
      2.1.1. Copy *s1*[*i*] into *s3*[*k*], then increment *i*, *j*, and *k*.
   2.2. Else, if *s1*[*i*] is less than *s2*[*j*]:
      2.2.1. Increment *i*.
   2.3. Else (if *s1*[*i*] is greater than *s2*[*j*]):
      2.3.1. Increment *j*.
3. Terminate.

3G To read values from the unsorted file *f* into a sorted array *a*[0…] (version 1):

1. Set *m* to 0.
2. While not at end of file *f*, repeat:
   2.1. Read *val* from *f*.
   2.2. Copy *val* into *a*[*m*].
   2.3. Increment *m*.
3. Sort *a*[0…*m*–1].
4. Terminate.

Step 2 performs 0 comparisons. If step 3 uses (say) selection sort, it performs about $n^2/2$ comparisons. Version 1 therefore performs about $n^2/2$ comparisons.

To read values from the unsorted file *f* into a sorted array *a*[0…] (version 2):

1. Set *m* to 0.
2. While not at end of file *f*, repeat:
   2.1. Read value *val* from *f*.
   2.2. Insert *val* in the sorted array *a*[0…*m*–1].
   2.3. Increment *m*.
3. Terminate.

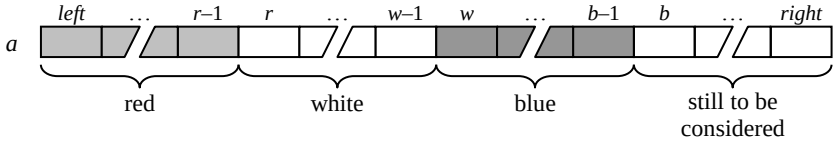
Step 2.2 would use the sorted-array insertion algorithm of Exercise 3D(c). This performs about *m*/2 comparisons. Since *m* ranges from 0 to *n*–1, the total number of comparisons is $0 + 1/2 + ... + (n–1)/2 = n(n–1)/4 \approx n^2/4$.

Both versions have time complexity $O(n^2)$, but version 2 is about twice as fast as version 1.

3H To sort an array of colors *a*[*left*…*right*] into the order red–white–blue:

1. Set *r* to *left*, set *w* to *left*, and set *b* to *left*.
2. While *b* ≤ *right*, repeat:
   2.1. If *a*[*b*] is blue:
      2.1.1. Increment *b*.
   2.2. If *a*[*b*] is white:
      2.2.1. If *b* > *w*, swap *a*[*b*] with *a*[*w*].
      2.2.2. Increment *w* and *b*.
   2.3. If *a*[*b*] is red:
      2.3.1. If *b* > *r*, swap *a*[*b*] with *a*[*r*].
      2.3.2. If *b* > *w*, swap *a*[*b*] with *a*[*w*].
      2.3.3. Increment *r*, *w*, and *b*.
3. Terminate.

The loop invariant is:



This algorithm performs 1 color comparison and at most 4 copies per iteration, i.e., *n* color comparisons and at most 4*n* copies in total. Its time complexity is $O(n)$.

3J Let $n_1 = r1 - l1 + 1$, $n_2 = r2 - l2 + 1$, and $n = n_1 + n_2$.

To copy all values from unsorted arrays *a1*[*l1*…*r1*] and *a2*[*l2*…*r2*] into sorted array *a3*[*l3*…*r3*] (version 1):

1. Concatenate *a1*[*l1*…*r1*] and *a2*[*l2*…*r2*] into *a3*[*l3*…*r3*].
2. Sort *a3*[*l3*…*r3*].
3. Terminate.

Step 1 performs 0 comparisons. If step 2 uses (say) selection sort, it performs about $n^2/2$ comparisons. The total number of comparisons is therefore about $n^2/2 = (n_1 + n_2)^2/2 = n_1^2/2 + n_2^2/2 + n_1 n_2$.

To copy all values from unsorted arrays *a1*[*l1*…*r1*] and *a2*[*l2*…*r2*] into sorted array *a3*[*l3*…*r3*] (version 2):

1. Sort *a1*[*l1*…*r1*].
2. Sort *a2*[*l2*…*r2*].
3. Merge *a1*[*l1*…*r1*] and *a2*[*l2*…*r2*] into *a3*[*l3*…*r3*].
4. Terminate.

If steps 1 and 2 use selection sort, they perform about $n_1^2/2$ and $n_2^2/2$ comparisons, respectively. Step 3 performs about $n = n_1 + n_2$ comparisons. The total number of comparisons is therefore about $n_1^2/2 + n_2^2/2 + n_1 + n_2$.

For all but small values of n_1 and n_2, $n_1 + n_2 < n_1 n_2$. Therefore version 2 is faster than version 1.

3K To find the position of the leftmost subarray of $a[0\ldots n-1]$ that matches $b[0\ldots m-1]$ (assuming that $m \le n$):

1. For $p = 0, \ldots, n-m$, repeat:
   1.1. If $a[p\ldots p+m-1]$ matches $b[0\ldots m-1]$, terminate with answer p.
2. Terminate with answer *none*.

To determine whether $a[p\ldots p+m-1]$ matches $b[0\ldots m-1]$:

1. For $d = 0, \ldots, m-1$, repeat:
   1.1. If $a[p+d]$ is unequal to $b[d]$, terminate with answer *false*.
2. Terminate with answer *true*.

The auxiliary algorithm performs between 1 and m comparisons, i.e., $(m+1)/2$ comparisons on average. The main algorithm performs between 1 and n iterations, i.e., $(n+1)/2$ iterations on average. Therefore it performs $(m+1)(n+1)/4$ comparisons on average. Its time complexity is $O(mn)$.

Solutions to Exercises 4

4A To access the *k*th element of the SLL headed by *first*, counting the first element as 0:

1. Set *curr* to *first*.
2. Repeat *k* times:
   2.1. If *curr* is null, terminate with answer *none*.
   2.2. Set *curr* to node *curr*'s successor.
3. Terminate with answer *curr*.

This algorithm follows up to *k* links. Its time complexity is $O(k)$.

4B To access the *k*th element of the DLL headed by (*first*, *last*), counting the first element as 0:

1. Let *n* be the length of the DLL headed by (*first*, *last*).
2. If $2k < n$:
   2.1. Set *curr* to *first*.
   2.2. Repeat *k* times:
      2.2.1. If *curr* is null, terminate with answer *none*.
      2.2.2. Set *curr* to node *curr*'s successor.
3. Else, if $2k \geq n$:
   2.1. Set *curr* to *last*.
   2.2. Repeat $n–1–k$ times:
      2.2.1. If *curr* is null, terminate with answer *none*.
      2.2.2. Set *curr* to node *curr*'s predecessor.
4. Terminate with answer *curr*.

If the DLL's length is immediately available, step 1 follows 0 links. Step 2 follows *k* links, while step 3 follows $(n–1–k)$ links. This algorithm's time complexity is $O(\max(n, n–k))$.

If the DLL's length is not immediately available, step 1 would have to follow *n* links, so it would be better just to mimic the algorithm of Exercise 4A.

4C To reverse the elements of the SLL headed by *first*:

1. Set *curr* to *first* and set *pred* to null.
2. While *curr* is not null, repeat:
   2.1. Let *succ* be *curr*'s successor.
   2.2. Set *curr*'s successor to *pred*.
   2.3. Set *pred* to *curr*.
   2.4. Set *curr* to *succ*.
3. Set *first* to *pred*.
4. Terminate.

This algorithm follows *n* links, so its time complexity is $O(n)$. Its space complexity is $O(1)$.

4D To reverse the elements of the DLL headed by (*first*, *last*):

1. Set *curr* to *first*.
2. While *curr* is not null, repeat:
   2.1. Let *succ* be *curr*'s successor.
   2.2. Swap *curr*'s predecessor and successor links.
   2.2. Set *curr* to *succ*.
3. Swap *first* and *last*.
4. Terminate.

This algorithm follows n links, so its time complexity is $O(n)$. Its space complexity is $O(1)$.

4E To test whether the SLL headed by *first* is a palindrome:

1. Let n be the length of the SLL headed by *first*.
2. Copy characters in reverse order from the first $n/2$ nodes of the SLL headed by *first* into another SLL headed by *prefix*, and let *suffix* be a link to the next node of the SLL headed by *first*.
3. If n is odd, set *suffix* to *suffix*'s successor.
4. Let *matched* be the result of testing whether the SLL headed by *prefix* matches the SLL headed by *suffix*.
5. Terminate with answer *matched*.

To copy characters in reverse order from the first k nodes of the SLL headed by *first* into another SLL headed by *prefix*, and let *suffix* be a link to the next node of the SLL headed by *first*:

1. Set *curr* to *first*, and set *prefix* to null.
2. Repeat k times:
   2.1. Insert *curr*'s character before the first node of the SLL headed by *prefix*.
   2.2. Set *curr* to *curr*'s successor.
3. Set *suffix* to *curr*.
4. Terminate with answers *prefix* and *suffix*.

To test whether the SLL headed by *prefix* matches the SLL headed by *suffix*:

1. Set *p* to *prefix*, and set *s* to *suffix*.
2. While *p* and *s* are not null, repeat:
   2.1. If node *p*'s character ≠ node *s*'s character, terminate with answer *false*.
   2.2. Set *p* to node *p*'s successor, and set *s* to node *s*'s successor.
3. Terminate with answer *true*.

The main algorithm performs $n/2$ character comparisons. Step 1 follows either 0 or n links, depending on whether the SLL's length is immediately available or not. Step 2 follows $n/2$ links. Step 4 follows $n/2$ links in each of two SLLs. In total, the algorithm follows either $3n/2$ or $5n/2$ links.

4G If a sorted DLL contains words in alphabetical order, it would be advantageous to search the DLL right-to-left when the target word's initial letter is in the second half of the alphabet.

4H To find which if any node of the unsorted DLL headed by (*first*, *last*) contains an element equal to *target* (version that searches simultaneously from both ends):

1. If *first* and *last* are null, terminate with answer *none*.
2. Set *p* to *first*, and set *s* to *last*.
3. Repeat:
   3.1. If *target* is equal to node *p*'s element, terminate with answer *p*.
   3.2. If *target* is equal to node *s*'s element, terminate with answer *s*.
   3.3. If *p* and *s* are the same node, or node *p* is node *s*'s predecessor, terminate with answer *none*.
   3.4. Set *p* to node *p*'s successor, and set *s* to node *s*'s predecessor.

On a successful search, this algorithm performs between 1 and n comparisons, i.e., $(n+1)/2$ comparisons on average. On an unsuccessful search, it performs n comparisons. Thus it is no better than the original unsorted DLL linear search algorithm.

This algorithm's time complexity is $O(n)$.

4J To find which if any node of the unsorted SLL headed by *first* contains an element equal to *target* (version that moves the node to the front of the SLL):

1. Set *pred* to null.
2. For each node *curr* of the SLL headed by *first*, repeat:
   2.1. If *target* is equal to *curr*'s element:
      2.1.1. If *pred* is not null:
         2.1.1.1. Set *pred*'s successor to *curr*'s successor.
         2.1.1.2. Set *curr*'s successor to *first*.
         2.1.1.3. Set *first* to *curr*.
      2.1.2. Terminate with answer *curr*.
   2.2. Set *pred* to *curr*.
3. Terminate with answer *none*.

If the same x is searched for 50 times out of the next 100 searches, x will be the first or second element in the SLL for most of the time, so each of the 50 searches for x will perform only 1 or 2 comparisons. Each of the remaining 50 searches (if successful) will perform about $n/2$ comparisons on average. The total number of comparisons for the 100 searches will be about $100 + 25n$.

If we use the original unsorted SLL search algorithm, each of the 100 searches (if successful) will perform about $n/2$ comparisons on average. The total number of comparisons will be about $50n$. Thus the above algorithm is faster for all but small values of n.

4K To delete the node containing element *elem* in the SLL headed by *first*:

1. Set *pred* to null.
2. For each node *curr* of the SLL headed by *first*, repeat:
   2.1. If *target* is equal to *curr*'s element:
      2.1.1. Let *succ* be *curr*'s successor.
      2.1.2. If *pred* is null, set *first* to *succ*.
      2.1.3. If *pred* is not null, set *pred*'s successor to *succ*.
      2.1.4. Terminate.
   2.2. Set *pred* to *curr*.
3. Terminate.

To delete the node containing element *elem* in the DLL headed by (*first*, *last*):

1. For each node *curr* of the DLL headed by (*first*, *last*), repeat:
   2.1. If *target* is equal to *curr*'s element:
      2.1.1. Let *pred* be *curr*'s predecessor, and let *succ* be *curr*'s successor.
      2.1.2. If *pred* is null, set *first* to *succ*.
      2.1.3. If *pred* is not null, set *pred*'s successor to *succ*.
      2.1.4. If *succ* is null, set *last* to *pred*.
      2.1.5. If *succ* is not null, set *succ*'s predecessor to *pred*.
      2.1.6. Terminate.
3. Terminate.

4L To insert *elem* after node *pred* in the SLL headed by *first*:

1. Let *succ* be *pred*'s successor.
2. Make *ins* a link to a newly-created node with element *elem* and successor *succ*.
3. Set *pred*'s successor to *ins*.
4. Terminate.

(*Note:* If we wish to insert *elem* before the SLL's first node, *pred* will be the dummy node.)

To delete node *del* in the nonempty SLL headed by *first*:

1. Let *succ* be *del*'s successor.
2. Let *pred* be *del*'s predecessor.
3. Set *pred*'s successor to *succ*.
4. Terminate.

(*Note:* Like the corresponding step of the SLL deletion algorithm, step 2 must find the predecessor by traversing the SLL from its first node.)

The above algorithms are neater than the original algorithms, but they have the same time complexities, $O(1)$ and $O(n)$ respectively.