Problem Set 1

Big O

1. Exercise 3.7 of the textbook.

   An algorithm prints the following pattern:

   ```
   *
   * *
   * * *
   * * * *
   * * * * *
   ```

   a. What are the basic operations performed by the algorithm that you would count towards its running time?
   b. Count the number of these basic operations for the specific output shown above.
   c. The number of lines printed in the preceding pattern is 5. Assume that the algorithm can extend this pattern for any number of lines (line number *i* has *i* stars). If the number of lines, *n* is the input to the algorithm, how many basic operations are performed as a function of *n*?
   d. Write your answer to the above question as a big *O* order.

   SOLUTION

   a. Printing a star, printing a blank space, printing a newline
   b. 15 print stars, 10 print blank spaces, 5 print newlines
   c. $n*(n+1)/2$ print stars, $n*(n-1)/2$ print spaces, n print newlines
   d. $O(n^2)$

2. Exercise E3.10, page 117 of the textbook.

   A spreadsheet keeps track of student scores on all the exams in a course. Each row of the spreadsheet corresponds to one student, and each column in a row corresponds to his/her score on one of the exams. There are *r* students and *c* exams, so the spreadsheet has *r* rows and *c* columns.

   Consider an algorithm that computes the total score on all exams for each student, and the average class score on each exam. You need to analyze the running time of this algorithm.

   a. What are the basic operations you would count toward the running time?
   b. What is the worst-case running time as a total count (not big *O*) of these basic operations?
   c. What is the big *O* running time?
   d. Is your algorithm linear, quadratic, or some other order?

   SOLUTION

   a. Basic operations are addition and division.
   b. Each student total (which starts at zero) will be added to *c* times. Since there are *r* student totals, there are $r*c$ additions for the totals.

      Each exam average will added to *r* times. Since there are *c* exams, this gives another $c*r$ additions.

      Each exam average will be divided exactly once by the number of students, *c*. So, *c* divisions.

   c. O(*rc*)
   d. LINEAR (Note: The Big O looks like a quadratic value, but the INPUT SIZE is *rc*, and the running time is linearly proportional to the input size.)

3. Exercise 3.14, pg 118 of the textbook

   A card game program keeps a deck of cards in an array. Give an algorithm to "unshuffle" the deck so that all the cards of a suit are grouped together, and within each suit, the cards are in ascending order or rank--consider the ace as the lowest ranked card. Since there are only 52 cards, space is not an issue so you can use extra space if needed. The only constraint is that your algorithm be as fast as possible.

   What is the worst case big O running time of your algorithm? What are the basic operations you used in your analysis? Is the average big O running time different from the worst case?

   SOLUTION

   Allocate four new arrays, each of length 13, one for each suit. Go through the original deck from front to end, and slot each card in its appropriate position in the suit array to which it belongs. So for example the queen of hearts will go to index position 11 in the hearts array. Note that since the indexes of the arrays are from 0 through 12, the array position of a card will be one less than its face value. (Queen's face value is 12, so its array position will be 11.)

   The big O running time is O(1)!!

   The basic operations are (a) looking up a card in the deck array, and (b) writing it into a suit array. Each of these takes unit time. (Writing takes unit time because the suit array is directly indexed with the card's face value.) Since there are 52 look ups and 52 writes, the total number of basic operations, and therefore the total units of time, is 104. This is a constant, so the big O time is O(1).

   Pay attention to this result, because what it says is that no matter how many actual operations you perform, because the input size is always the same, 52, the number of operations does not vary. Hence O(1).

---

4. * Exercise E3.11, page 118 of the textbook.

   Two people compare their favorite playlists for matching songs. The first playlist has *n* songs, and the second has *m*. Each is stored in an array, in no particular order.

   1. Describe an algorithm to find the common songs in these lists (intersection), WITHOUT sorting either list.
      a. What is the worst-case big *O* running time of your algorithm? Make sure to state the basic operations used in your analysis of running time.
      b. What is the best-case big *O* running time of your algorithm? Explain clearly, including all book-keeping needed to achieve this best case.
   2. Now suppose you could sort either or both arrays (as part of your algorithm). Repeat the worst-case and best-case analysis for the big *O* running time. (The running time must include the time to sort.)

   SOLUTION

   1. Algorithm: For each song in the first playlist, do a linear search in the second and if a match is found, add it to the output list.

      Basic operation is a comparison between a pair of songs.

      1. In the worst case, there are no common songs. Every song in the first list gets compared with every song in the second, for a total of *n*m* comparisons, and therefore, O(*n*m*)
      2. In the best case, there is a maximum number of common songs, which would be min(*m*,*n*) (number of common songs cannot exceed the length of the smaller list). Also, the least number of comparisons is made to find a match. Again, doing a linear search for each song in the *n* list against the songs in the *m* list, the first song in the *n* list matches the first song in the *m* list (1 comparison), the second song in the *n* list matches the second song in the *m* list (2 comparisons), and so on. This gives a total of:

         1+2+3+...+min(m,n)

comparisons, which is O(min(m,n)^2)

2. Use mergesort to sort each array. Then use the sorted lists merge algorithm to compare songs pair-wise and only keeping in the output the pairs that match.
    1. Worst case: O(mlog m) + O(nlog n) + O(m+n) = O(mlog m + nlog n)
    2. Best case: O(mlog m) + O(nlog n) + O(min(m,n)) = O(mlog m + nlog n)

---

5. ** Exercise E3.15, page 118 of the textbook.

There is a highway with *n* exists numbered 0 to *n* - 1. You are given a list of the distances between them. For instance:

```
Exits:     1  2  3  4  5  6
Distances: 5  3  4  2  8  6
```

The distance from the start of the highway to exit 1 is 5, from exit 1 to 2 is 3, and so on.

You have to devise an algorithm that would calculate the distance between any two exits. Imagine that this distance function is called millions of times by applications that need this information, so you need to make your algorithm as fast as possible.

Describe your algorithm. What is the worst-case big *O* running time? Make sure to state all the parameters you use in the analysis and relate them to the algorithm.

SOLUTION

As the function that calculates the distance between any two exits is going to be called millions of times, it should be as fast as possible, ideally *O(1)* We can achieve this speed by creating an array, S of distances of each exit from the start of the highway. Assume that the original distances between exits (as in the example given in the problem) are in an array D

```
// copy distances between pairs of exits from D to S
for (i=0; i < n; i++) {
  S[i] = D[i];
}
// compute distance ("sum") from start for each exit
for (i=1; i < n; i++) {
  S[i] = S[i-1] + S[i];
}

D: 5  3  4  2  8  6
S: 5  8 12 14 22 28
```

Creating the array takes linear time (*O(n)*) but using it will take constant time (*O(1)*). Once S is created, finding the distance from exit i to exit j is a simple matter of computing S[j] - S[i].

Running time Analysis

- The basic operation for creating the S array is calculating the distance of each exit from the start of highway. It takes O(1) time for each exit and there are *n* exits. Therefore, it takes *O(n)* time.
- The basic operation for finding the distance between two exits after we have the S array is a subtraction of two distance, both of which are obtained in constant time. So, *O(1)* time.
