1 Submission Instructions

Create a document using your favorite word processor and type your exercise solutions. At the top of the document be sure to include your name and the homework assignment number, e.g. HW3. Convert this document into **Adobe PDF format** and name the PDF file *<asuriteid>.pdf* where *<asuriteid>* is your ASURITE user id (for example, my ASURITE user id is *kburger2* so my file would be named *kburger2.pdf*). To convert your document into PDF format, Microsoft Office versions 2008 and newer will export the document into PDF format by selecting the proper menu item from the File menu. The same is true of Open Office and Libre Office. Otherwise, you may use a freeware PDF converter program, e.g., *CutePDF* is one such program.

Next, create a folder named *<asuriteid>* and copy *<asuriteid>.pdf* to that folder. Copy any requested Java source code files to this folder (note: Java source code files are the files with a *.java* file name extension; **do not** copy the *.class* files as we do not need those).

Next, compress the *<asuriteid>* folder creating a **zip archive** file named *<asuriteid>.zip*. Upload *<asuriteid>.zip* to the Homework Assignment 3 dropbox by the assignment deadline. The deadline is 11:59pm Mon 21 Apr. Consult the online syllabus for the late and academic integrity policies.

Note: not all of these exercises will be graded, i.e., random ones will be selected for grading.

2 Learning Objectives

1. To apply recursion solutions to solve problems.
2. To use the linear search and binary search algorithms to search a list for a key.
3. To analyze an algorithm to determine the asymptotic time complexity.
4. To determine the order of growth of a function and express it in Big O notation.
5. To implement sorting algorithms to sort a list of elements.
6. To analyze the time complexity of sorting algorithms.

3 Recursion

3.1 The sum of the first n positive integers is:

$$sum(n) = \sum_{i=1}^{n} i$$

and can easily be computed using a for loop:

```
public int sum1toN(int n) {
    int sum = 0;
    for (int i = 1; i <= n; ++i) {
        sum += i;
    }
    return sum;
}
```

It is also possible to recursively compute the sum by recognizing that $sum(n) = n + sum(n - 1)$. For this exercise, write a recursive version of *sum1toN*().

3.2 Name your source code file *Hw3_1.java*. Write a recursive method `double power(double x, int n)` that computes and returns x^n where $n \geq 0$. Hint: remember that $x^n = x \cdot x^{n-1}$. Write a driver routine that computes (and prints) 2^p for $p = 0, 2, 3, ..., 10$ by calling *power*(). This driver routine will test your method to ensure it is working properly.

3.3 Modify *Hw3_1.java* to implement a recursive method `double powerFaster(double x, int n)` that computes and returns x^n. However, write the code so that when n is even the method will return $(x^{n/2})^2$. Modify your driver routine to compute and print 2^p for $p = 0, 2, 3, ..., 10$ by calling *powerFaster*().

3.4 Modify *Hw1_1.java* and declare two int counter variables (as private instance variables) named *calls* and *callsFaster*. In the loop of the test driver, set both instance variables to 0 before calling *power*() and *powerFaster*(). Increment the corresponding counter variable as the first statement in each of the power methods. After each method returns the computed power 2^p back to the driver routine, print the values of *calls* and *callsFaster* (if you have not figured it out yet, *calls* and *callsFaster* are counting the number of times each method is called during the computation of 2^p). Explain why *powerFaster*() is faster in some cases than *power*().

3.5 Write a recursive method `String reverse(String s)` that returns the reverse of *s*. For example, if *s* is "Hello world" then the method would return "dlrow olleH". Hint: remove the first character *c* at index 0 of *s* forming a substring *t* which consists of the characters at indices 1, 2, 3, ..., *s.length*()-1 of *s*; then concatenate the reverse of *t* and *c* and return this new string.

4 Linear and Binary Search

4.1 Write a recursive method `int recLinearSearch(ArrayList<String> pList, String pKey, int pBeginIdx, int pEndIdx)` that searches *pList* elements *pBeginIdx* up to and including *pEndIdx* for *pKey* and returns the index of *pKey* in *pList* if found and -1 if not found. The method will be called in this manner to search the entire list:

```
ArrayList<String> list = new ArrayList<>();
// populate list with some Strings...
int idx = recLinearSearch(list, "some key", 0, list.size() - 1);
```

Hint: the base case is reached when *pBeginIdx* is greater than *pEndIdx* (what does this mean?). Otherwise, check to see if the element at *pBeginIdx* is *pKey*. If it is, then return *pBeginIdx*. If it is not, then make a recursive method call.

4.2 Suppose *list* is an *ArrayList* of *Integer*s and contains these elements:

```
list = { 2, 3, 5, 10, 16, 24, 32, 48, 96, 120, 240, 360, 800, 1600 }
```

and we call the recursive binary search method from the lecture notes:

```
int idx = recursiveBinarySearch(list, 10, 0, list.size() - 1);
```

Trace the method and show the values of *pLow* and *pHigh* on entry to each method call. Show the value of *pMiddle* that is computed and state which clause of the if-elseif-elseif statement will be executed, i.e., specify if `return middle;` will be executed, or if `return recursiveBinarySearch(pList, pKey, pLow, middle - 1);` will be executed, or if `return recursiveBinarySearch(pList, pKey, middle + 1, pHigh);` will be executed. Finally, at the end, specify the value assigned to *index* and the total number of times that *recursiveBinary Search*() was called (including the original call).

4.3 Repeat Exercise 4.2 but this time let *pKey* be 150.

5 Analysis of Algorithms and Big O Notation

5.1 Using the formal definition of big O, prove mathematically that $f(n) = 2.5n + 4$ is $O(n)$.

5.2 Using the formal definition of big O, prove mathematically that $f(n)$ = -4 $\times$ $10^{6,00,000}$ is O(1).

5.3 An important concept to know regarding big O notation is that for logarithmic complexity the base is irrelevant. In other words, if $f(n)$ is a function that counts the number of times the key operation is performed as a function of n and $f(n)$ is $O(log_a\ n)$ then it it also true that $f(n)$ is $O(log_b\ n)$. For example, binary search—which is usually stated as being $O(lg\ n)$—is also $O(ln\ n)$, $O(log_{10}\ n)$, and $O(log_{3.14159265}\ n)$. Using the formal definition of big O, prove mathematically that if $f(n)$ is $O(log_a\ n)$ then $f(n)$ is also $O(log_b\ n)$.

5.4 Consider this *split*() method where: *pList* is an *ArrayList* of *Integer*s containing zero or more elements; *pEvenList* is an empty *ArrayList* of *Integer*s; and *pOddList* is an empty *ArrayList* of *Integer*s. On return, *pEvenList* will contain the even *Integer*s of *pList* and *pOddList* will contain the odd *Integer*s.

```
void split(ArrayList<Integer> pList, ArrayList<Integer> pEvenList, ArrayList<Integer> pOddList) {
    for (int n : pList) {
        if (n % 2 == 0) pEvenList.add(n);
        else pOddList.add(n);
    }
}
```

To analyze the worst case time complexity of an algorithm we first identify the "key operation" and then derive a function which counts how many times the key operation is performed as a function of the size of the input. What is the key operation in this algorithm? Explain.

5.5 Continuing with the previous exercise, derive a function *f*(n) which equates to the number of times the key operation is performed as a function of n, where n is the size of *pList*. State the worst case time complexity of *split*() in big O notation.

5.6 Would the time complexity of *split*() change if the elements of *pList* were sorted into ascending order? Explain.

5.7 Binary search is such an efficient searching algorithm because during each pass of the loop (in the iterative version) or in each method call (in the recursive version) the size of the list is essentially halved. If reducing the size of the list to be searched by one half is so effective, it may seem that reducing it by two thirds each time would be even more effective. To that end, consider this iterative ternary search method. Rewrite it is as a recursive ternary search method named `int recTernarySearch(ArrayList<Integer> pList, Integer pKey, int pLow, int pHigh)`.

```
int ternarySearch(ArrayList<Integer> pList, Integer pKey) {
    int low = 0, high = pList.size() - 1;
    while (low <= high) {
        int range = high - low;
        int oneThirdIdx = (int)Math.round(low + range / 3.0);
        int twoThirdIdx = (int)Math.round(low + range / 1.33);
        if (pKey.equals(pList.get(oneThirdIdx))) {
            return oneThirdIdx;
        } else if (pKey.equals(pList.get(twoThirdIdx))) {
            return twoThirdIdx;
        } else if (pKey < pList.get(oneThirdIdx)) {
            high = oneThirdIdx - 1;
        } else if (pKey > pList.get(twoThirdIdx)) {
            low = twoThirdIdx + 1;
        } else {
            low = oneThirdIdx + 1;
            high = twoThirdIdx - 1;
        }
    }
    return -1;
}
```

5.8 For *recTernarySearch*() the key operations are the four comparisons of *pKey* to the the elements of *pList*. Treat these four comparisons as one comparison and provide an informal proof of the worst case time complexity of ternary search. To simplify the analysis, assume that the size of the list on entry to *recTernarySearch*() is always a power of 3, e.g., on the first call assume the size of the list is 3^p, on the second call the size of the list is 3^{p-1}, on the third call 3^{p-2}, and so on.

6 Sorting

6.1 Consider the *Point* class discussed in *Objects and Classes : Section 1* (in *burger-cse205-note-objs-classes-01.pdf* in the *Week 1 Notes* zip archive; the *Point.java* source code file can be found in the *Week 1 Source Code* zip archive). Modify this class so it implements the *Comparable<Point>* interface. We define *Point p1* to be less than *Point p2* if the distance from the origin to *p1* is less than the distance from the origin to *p2*; *p1* is greater than *p2* if the distance from the origin to *p1* is greater than the distance from the origin to *p2*; otherwise, if they distances are equal then *p1* is equal to *p2*. Submit your modified source code file in your homework submission zip archive.

6.2 Consider this *ArrayList* of *Integer*s, which is to be sorted into ascending order:

`list = { 13, 75, 12, 4, 18, 6, 9, 10, 7, 14, 15 }.`

Trace the *insertionSort*() method and show the contents of *list*: (1) on entry to *insertionSort*(); (2) after the for j loop terminates each time but before the for i loop is repeated; and (3) after the for i loop terminates. The objective of this exercise is to help you understand how the insertion sort algorithm sorts a list.

6.3 Your after-hours research finally pays off: you discover a way to build a computer system that is one billion times faster than the fastest system currently in existence. Therefore, on your new computer system each comparison of selection sort requires only 25×10^{-18} seconds, which is 25 attoseconds. Create a table similar to the one in *Sorting Algorithms : Section 6* showing how long selection sort will require to sort various sized lists.

6.4 If we would like to require *insertionSort*() to sort a list of 10-billion elements in no more than one minute, how many times faster would your new computer system need to be than the fastest system currently in existence?