**Problem 1**

1. *(2 points) Explain what it means for a function to be 𝑂(1). Be sure to reference the definition of Big-Oh in your explanation.*

A function is *𝑂(1)* if it can be executed in constant time. This means that the number of operations performed by the function does not depend on the number of inputs *n*. Big-Oh is a way to express how long an algorithm or function takes to run, and is not expressed in discrete units of time, but in terms of how quickly the runtime grows relative to the input *n*, as *n* gets arbitrarily large. Big-Oh notation is called “asymptotic notation” because it describes the ultimate upper bound of the runtime when *n* gets so large that all constant terms cease to be relevant in proportion to the magnitude of *n*. A function that is *𝑂(1)* will have the same worst-case runtime for an input of *n*=2 elements as it would for an input of *n*=10000 elements. An example of an *𝑂(1)* algorithm is indexing into an array.

1. *(10 points) [Weiss 2.1] Order the following functions by growth rate:*

𝑁, , 𝑁1.5, 𝑁2, 𝑁log𝑁, 𝑁log(log𝑁), 𝑁log2N, 𝑁log(𝑁2),, 2𝑁, , 37, 𝑁2logN, N3

*Indicate which functions grow at the same rate.*

1. *(5 points) [Weiss 2.5] Find two functions 𝑓(𝑛) and 𝑔(𝑛) such that neither*

*𝑓(𝑛)=𝑂(𝑔(𝑛)) nor 𝑔(𝑛)=𝑂(𝑓(𝑛)).*

For the specified conditions to be true, *𝑓(𝑛)* must NEVER act as the upper bound of *g(𝑛),* and vis versa, for all values of n greater than some arbitrary point. If both functions are orthogonal periodic functions like sine and cosine, and *C* is some constant to ensure the functions are nonnegative, then the range of both functions is [*C*-1, *C*+1] for all *n*. However, when 𝑓*(n)*=*C-1,* then *g(n)*=*C*+1; and when 𝑓*(n)*=*C+1,* then *g(n)*=*C*+1. Therefore, neither function ever bounds the other for all values of n greater than some point, as the relative magnitudes of both functions switch on every period.

1. *(10 points) [Weiss 2.7.6] For the following code fragment: Give an analysis of the running time (Big-Oh will do).*

O(1)🡪 sum = 0;

O(n)🡪 for(i = 1; i < n; i++)

O(n2)🡪 | for(j = 1; j < i\*i; j++)

O(1)🡪 | | if(j%i == 0)

O(n2)🡪 | | | for(k = 0; k < j; k++)

O(1)🡪 | | | | sum++;

Total runtime =

* **for(k<j)** loop is O( because it is nested within **for(j<i\*i**), so it is an *O*(*n*) algorithm that is executed i2 times
* we can ignore the conditional statements as a general rule, as they fall out for large *n*

1. *(10 points) [Weiss 2.15] Give an efficient algorithm to determine if there exists an integer i such that 𝐴𝑖 = 𝑖 in an array of integers 𝐴1 < 𝐴2 < 𝐴3 < ⋯ < 𝐴𝑁.*

*What is the running time of your algorithm in terms of Big-Oh?*

/\* PseudoCode algorithm \*/

**boolean** modifiedBinarySearch(**array** A) {

**int** length **=** A.length; // length of array

**int** lwr **=** **0**; // lower bound of BS

**int** upr **=** length; // upper bound of BS

**int** ind\_ans **=** **-1**; // index of answer

**while** (lwr **<=** upr) {

**int** midpoint **=** (lwr**+**upr)**/ 2**; // middle of array

// if value equals its index, index is solution

**if** (A[midpoint] **=** midpoint **+** **1**) {

ind\_ans **=** midpoint **+** **1**;

}

// if value **>** index, discard UPPER half of array

**else** **if** (A[midpoint] **>** midpoint **+ 1**) {

upr **=** midpoint **-** **1**; // toss out

}

// value < index, discard LOWER half of array

**else** {

lwr **=** midpoint **+** **1**;

}

} // END WHILE

**if** (ind\_ans **==** **-1**) {

**return** **false**; // value does not exist

} **else** {

**return** **true**;

}

} // END

Algorithm is ***O(log*N*)*** It is a binary search, so each iteration reduces the number of elements by ½

1. (15 points) Do a complete run-time analysis for the algorithms in Figures 2.6, 2.9, and 7.6 in the text [Weiss].

1 /\*\*Fig 2.6

3 \* Quadratic maximum contiguous subsequence sum algorithm. \*/

5 int maxSubSum2( const vector<int> & a) {

6 int maxSum = 0;

7 for(inti=0;i<a.size();++i) {

8 int thisSum = 0;

9 for(int j=i; j< a.size( ); ++j) {

10 thisSum += a[ j ];

11 if( thisSum > maxSum )

12 maxSum = thisSum;

13 } }

15 return maxSum;

16 }

O(n)🡪 iterate through every element

O(n) 🡪 iterate again through every element

in all cases case because the entire set n must be summed

/\* Fig 2.7

21 \* Performs the standard binary search using two comparisons per level.

22 \* Returns index where item is found or -1 if not found. \*/

24 template <typename Comparable>

25 int binarySearch(const vector<Comparable> & a, const Comparable & x) {

26 int low = 0, high = a.size() -1;

27 while( low <= high ) {

28 int mid = ( low + high ) / 2;

29 if( a[ mid ] < x )

30 low = mid + 1;

31 else if( a[ mid ] > x )

32 high = mid - 1;

33 else

34 return mid; // found

35 }

36 return NOT\_FOUND; // Defined as -1

37 }

O(log(n)) 🡪 set size halved each iteration

worst case: binary search reduces elements by ½ each iteration

O(1) best case: finds value on first try

/\*\* Fig 7.6

1 \* Shellsort, using Shell's (poor) increments \*/

2 template <typename Comparable>

3 void shellsort ( vector <Comparable> & a ) {

4 for (int gap = a.size()/2; gap > 0; gap /=2 )

5 for (int i=gap; i<a.size(); ++i) {

6 Comparable tmp = std::move( a[i] );

7 int j = i;

8 for(; j>=gap && tmp<a[j-gap]; j -= gap)

9 a[j] = std:move( a[j-gap] );

10 a[j] = std::move(tmp);

11 } } // END

O(log(n)) 🡪 set size halved each iteration

O(n) 🡪 iterate over all elements (insertion sort)

O(n/m) 🡪

iterate over m subsets of elements

worst case: must insertion sort (*O(n)* ) on every subset

O(n\*log(n)) best case: inner loop never invoked if list is presorted