SPEL Technologies, Inc.

Assignment 1

Total points: 40

3. (a) $1/2\ n^2 - 3n = \Theta(n^2)$

We need to prove that the following equation is true:

$c_1\ n^2 \le 1/2\ n^2 - 3n \le c_2\ n^2$

Lower-bound

First, we check the lower bound as follows:

$c_1\ n^2 \le 1/2\ n^2 - 3n$

Dividing by n^2 throughout gives:

$c_1 \le 1/2 - 3/n$ …………….(1)

Choose a value for n (called n_0) for which (1) is true.

Suppose that we select $n_0 = 12$. Substituting this value back in (1) gives:

=> $c_1 \le 1/2 - 3/12$

=> $c_1 \le 1/4$

Let's choose $c_1 = 1/4$ and substitute this value of c_1 in (1). We can see that (1) is true for all $n >= n_0$. ....(2)

Upper-bound

Next, we check the upper bound as follows:

$1/2\ n^2 - 3n \le c_2\ n^2$

=> $1/2 - 3/n \le c_2$ …………….(3)

Use the same value of n_0 in (3) for which (1) is true.

=> $1/2 - 3/12 \le c_2$

=> $1/4 \le c_2$

Choose $c_2 = 1$ and substitute it in (3). We can check that (3) is true for all $n >= n_0$ for these values of $c_2 = 1$ and $n_0 = 12$.. ....(4)

From (2), $1/2\ n^2 - 3n = \Omega(n^2)$

From (4), $1/2\ n^2 - 3n = O(n^2)$

Therefore, $1/2\ n^2 - 3n = \Theta(n^2)$

(b) n^3 **!= O(n²) ?**

Assume $n^3 = O(n^2)$
$0 \le n^3 \le c_1 n^2$
$0 \le n \le c_1$
We cannot find any constant c_1 that can satisfy $n \le c_1$ for all values of $n > n_0$
Therefore, $n^3 \neq O(n^2)$
=> $n^3 ! \neq O(n^2)$

(c) n log n - 2n + 13 = Ω(n log n)

c_1 n log n ≤ n log n - 2n + 13
$c_1 \le 1 - 2/(\log n) + 13/(n \log n)$ —–- (1)
$c_1 \le 1 - 2/(\log n) + 13/(n \log n)$
$c_1 \le 1 - 2/3 + 13/3000$ (n_0 = 10^3)
$c_1 \le 1/3$
Let's choose $c_1 = 1/3$
(1) is satisfied for any value $n > n_0$ for $c_1 = 1/3$ and n_0 = 10^3.
=> n log n - 2n + 13 = Ω(n log n)

Ans : Ω(n log n)

(d) $n^{1/2} = \Theta(n^{2/3})$

This is false. For powers of n, compare the powers. As 1/2 < 2/3, $n^{1/2} = O(n^{2/3})$

(e) Prove that n! = ω(2^n)

For n > 3, $2^n = 2*2*2....*2 < 1*2*3*...*n = n!$
=> n! = w(2^n)

(f) (Additional problem) Is $2^{n+1} = O(2^n)$? is $2^{2n} = O(2^n)$?

$0 \le 2^{n+1} \le c_1 2^n$
$0 \le 2\ 2^n \le c_1\ 2^n$ ……………… (1)
Choose $n_0 = 1$

$0 \le 2 \le c_1$
(1) is satisfied for all values of n >= n_0 for $c_1 = 2$
Therefore, $2^{n+1} = O(2^n)$

$0 \le 2^{2n} \le c_1\, 2^n$
$0 \le 2^n\, 2^n \le c_1\, 2^n$
$0 \le 2^n \le c_1$
$0 \le 2 \le c_1 \quad n_0 = 1$

We cannot find any constant c_1 to satisfy $2^n \le c_1$ for any value $n > n_0$
$2^{2n} \ne O(2^n)$

4. **Find the recurrence relation for the following algorithm. what is the running time of the algorithm (that is, find the time complexity)**

```
  // assume n is even
arraySum(A, n){
      if (n=1) return;
      for (i=1 to n/2){
      A[i]=A[i]+A[n/2];
}

arraySum(A, n/2);
}
```

Solution : Find the recurrence relation by finding the running time for different values of n:

```
n = 1, T(1) = 1  // only the first if statement is executed
n = 2, T(2) = T(1) + 1
n = 4  T(4)= T(2) + 2
T(n) = T(n/2) + n/2
```

Solve the recurrence relation using back substitution:

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + \frac{n}{4}$$
$$=> T(n) = T\left(\frac{n}{4}\right) + \frac{n}{4} + \frac{n}{2}$$
$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + \frac{n}{8}$$
$$=> T(n) = T\left(\frac{n}{8}\right) + \frac{n}{8} + \frac{n}{4} + \frac{n}{2}$$
$$T(n) = T\left(\frac{n}{k}\right) + n\left[\tfrac{1}{2} + \tfrac{1}{2^2} + \ldots\ldots + \tfrac{1}{2^{lg\ k}}\right]$$

Note :

$$(2^x = k => x = lg\ k)$$
$$2^x = k\ for\ k = n$$
$$T(n) = T(1) + n\left[\tfrac{1}{2} + \tfrac{1}{2^1} + \ldots.. + \tfrac{1}{2^{lg\ k}}\right]$$
$$= \frac{1}{2} + \frac{1}{2^2} + \ldots. + \frac{1}{2^{log\ n}} \le 1$$

$$T(n) = O(n)$$

5. **Given an array of *n* integers and an integer *S*, write an algorithm that only returns true if the array contains two numbers whose sum is *S*.**
   (a) Find the recurrence relation of an algorithm that uses an exhaustive search and solve it to find the (best-case, worst-case, and average-case) running time of the algorithm.

   The exhaustive search sums all possible combinations of the numbers. If any sum is equal to S, it returns *true*; otherwise, it returns *false*.

```
bool checkSum_exhaustive(int *inputData, int size, int S) {
    bool flag = false; // S not found

 for (int i = 0; i  < size && flag == false; i++) {
   for (int j = i+1; j < size && flag == false; j++) {
       if (inputData[i] + inputData[j] == S)
            flag = true;
        }
    }

    return flag;
}
```

Analysis to find the running time
Best case : the first two values sum to S **O(1)**

Worst case : The last two numbers sum to S.

```
       for(i=0; i<n && flag == false; i++){
            for(j = i+1; j<n && flag == false; j++){
```

T(1) = time for iteration with n =1: Number of comparisons = 0
T(2) = time for iteration with n = 2: Number of comparisons = 1
T(3) = time with n = 3 , Number of comparisons = 3

Find the recurrence relation
Worst case:

n =1	X	Number of comparisons = 0
n = 2	ij = 01	Number of comparisons = 1
n = 3	ij = **01** 02 12	T(3) = T(2)+2

n = 4	`ij` = **01 02** 03 **12** 13 23	T(4) = T(3) + 3

$T(n) = T(n-1) + n - 1$

$T(n-1) = T(n-2) + (n-1) - 1$

$\Rightarrow T(n) = T(n-2) + n + (n-1) - 2$

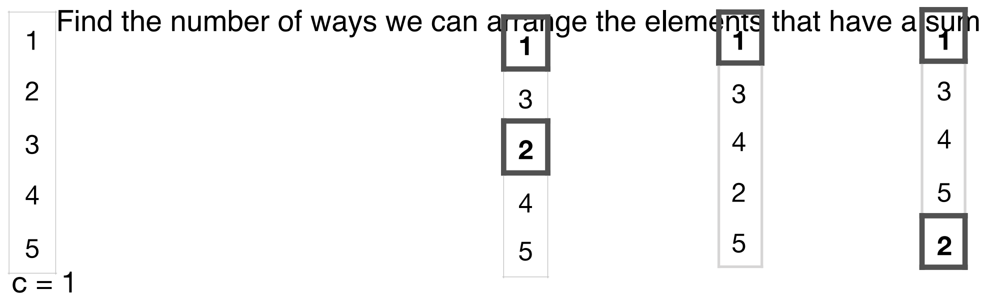
$T(n-2) = T(n-3) + (n-2) - 1$

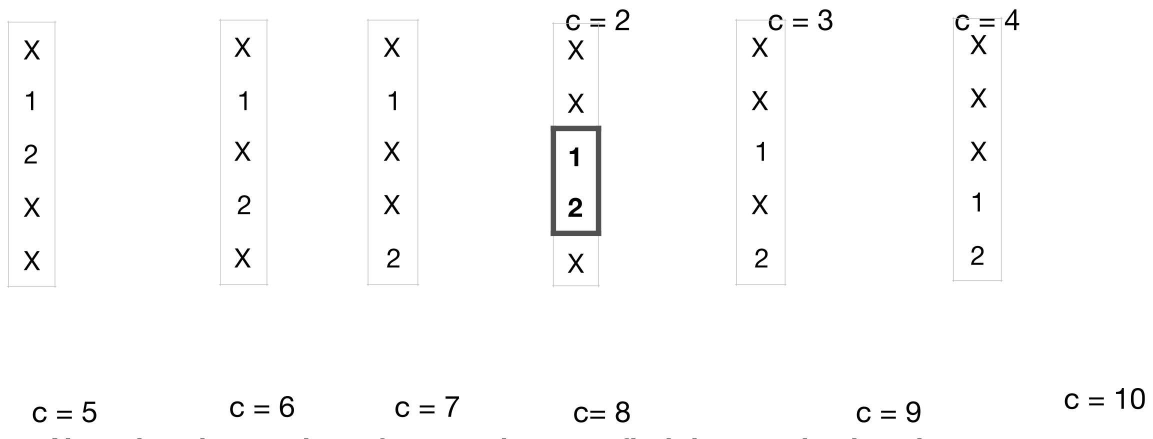
$\Rightarrow T(n) = T(n-3) + (n) - 1 + (n-1) - 1 + (n-2)-1 + ..$
$= T(n-3) + (n) + (n-1) + (n-2)-3$

$T(n) = T(2) + [n + n(n-1) + (n-2)+ \dots +3] - n$
$$= 1 + \left[\frac{n(n+1)}{2} - 3\right] - n$$

$T(n) = \Theta(n^2)$

Average case (Assume that c is number of comparisons)
Find the number of ways we can arrange the elements that have a sum equal to 3.





Note that the number of comparisons to find the sum is given by c
Number of ways we can rearrange the elements that sum to 3 = 3! X 2

Number of comparisons = 1, 2, 3, …, n(n-1)/2

Number of permutations for each comparison : `(n-2)! X 2`

`E(X)` = number of comparisons `(c)` X $P_r(c)$

`E(X)` = expected number of comparisons
`c` = number of comparisons
$P_r(c)$ = probability that c occurs

$$
\begin{aligned}
E(X) &= \sum cP_r(c) \\
&= 1\times\left[\frac{(n-2)!\times 2}{n!}\right]+2\left[\frac{(n-2)!\times 2}{n!}\right]+3\left[\frac{(n-2)!\times 2}{n!}\right]+ \\
&\quad\ldots+\left[\frac{n(n-1)}{2}\right]\left[\frac{(n-2)!\times 2}{n!}\right] \\
&= \frac{(n-2)!\times 2}{n!}\left[1+2+3+4+\ldots+\frac{n(n-1)}{2}\right] \\
&= \frac{2}{n(n-1)}\left[\frac{1}{2}\left(\frac{n^2-n}{2}\right)\left(\frac{n^2-n}{2}+1\right)\right] \\
&= \frac{2}{n(n-1)}\left[\frac{n}{4}(n-1)\left(\frac{n^2-n+2}{2}\right)\right] \\
&= \frac{n^2-n+2}{4} \\
&= O(n^2)
\end{aligned}
$$

Space complexity :O(n)

(b) Can you find a better algorithm to solve the problem? What is the time complexity (best case, worst case, and average case) of the algorithm? What is the space complexity of the algorithm?

Sort the numbers in ascending order and add two pointers to the first and last elements of the array. If their sum is greater than S, decrement the pointer of the larger element; otherwise, increment the pointer of the smaller element.

```
// sort the array first in ascending order. Then, maintain two indices
i, j to the start and end of the array, respectively.
```

```
// Add array[i] + array[j]. If the sum is greater than S, decrement
index j. If sum is less than S, increment index j.
bool checkSum_sorting(vector<int> &inputData, int size, int S) {
    bool flag = false; // set to true if sum of two elements is equal
to S
    bool completed = false;
    sort(inputData.begin(), inputData.end()); // function to sort the
array


    // initialize the indexes i and j to the first and last element of
the sorted array
    int i = 0;
    int j = inputData.size() - 1;

    while (!completed){
      if (inputData.at(i) + inputData.at(j)> S) {
                // choose a smaller value
            j--;
        }
        else if (inputData.at(i) + inputData.at(j) < S) {
            // choose a greater value
            i++;
        }
        else {
           flag = true; // sum of elements is equal to S
           completed = true;
        }

       // all done?
       if (i >= j)
          completed = true;
    }

   return flag;

}
```

Mergesort has best, worst, average case running time of O(n lg n) and worst case space complexity of O(n). After the elements are sorted, a linear scan is required to determine the

// Complete program to check if the sum of two numbers in array is S
// This solution does not use STL
// Mergesort is used to sort the array

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
```

```
using namespace std;

// uses exhaustive search - add array[i] and array[j] for all i and j
bool checkSum_exhaustive(int *inputData, int size, int S) {
    bool flag = false; // S not found

  for (int i = 0; i  < size && flag == false; i++) {
 for (int j = i+1; j < size && flag == false; j++) {
   if (inputData[i] + inputData[j] == S)
            flag = true;
        }
    }

    return flag;
}


void merge(int arr[], int p, int q, int r)
{
int n1 = q - p + 1;
int n2 = r - q;

// create arrays for merging
int L[n1], R[n2];

// Copy data from arr to arrays L and R
for (int i = 0; i < n1; i++) {
    L[i] = arr[p + i];
 }

for (int j = 0; j < n2; j++) {
    R[j] = arr[q + 1 + j];
}

// merge the arrays L and R into arr in sorted order
int i = 0;
int j = 0;
int k = p;
  while (i < n1 && j < n2)
{
      if (L[i] <= R[j])
      {
            arr[k] = L[i];
            i++;
      }
      else
      {
            arr[k] = R[j];
            j++;
      }
      k++;
}
  while (i < n1)
    arr[k++] = L[i++];

  while (j < n2)
    arr[k++] = R[j++];

}

void mergeSort(int arr[], int p, int r)
{
   if (p < r) {
      int q = (p+r)/2;

      // Sort first and second halves
      mergeSort(arr, p, q);
      mergeSort(arr, q+1, r);

      merge(arr, p, q, r);
  }
}

// sort the array first in ascending order. Then, maintain two indices
i, j to the start and end of the array, respectively.
// Add array[i] + array[j]. If the sum is greater than S, decrement
index j. If sum is less than S, increment index j.
bool checkSum_sorting(int *inputData, int size, int S) {
    bool flag = false; // set to true if sum of two elements is equal
to S
    bool completed = false;
    mergeSort(inputData, 0, size-1);


    // initialize the indexes i and j to the first and last element of
the sorted array
    int i = 0;
    int j = size - 1;

    while (!completed){
      if (inputData[i] + inputData[j] > S) {
                // choose a smaller value
            j--;
        }
        else if (inputData[i] + inputData[j] < S) {
            // choose a greater value
            i++;
        }
```

```
         else {
            flag = true; // sum of elements is equal to S
            completed = true;
         }

        // all done?
        if (i >= j)
           completed = true;
     }

   return flag;

}

int main() {
  // generate the input Data and call the function checkSum
  int size = 10;
  int *v1 = new int[size];

  srand(time(NULL));

  cout << "The input array is: " << endl;
  for (int i = 0; i < size; i++){
      v1[i] = rand() % 10;
      cout << v1[i] << endl;
  }
   // run exhaustive search
   //cout << "The result is " << checkSum_exhaustive(v1, size, 3) <<
endl;

   // run sort-based search
   cout << "The result is " << checkSum_sorting(v1, size, 3) << endl;
   cout << endl;

}
```

// Complete program to check if the sum of two numbers in array is S
// This solution uses STL

```
#include <iostream>
#include <vector>
#include <array>
#include <algorithm>
#include <ctime>

using namespace std;


// sort the array first in ascending order. Then, maintain two indices
i, j to the start and end of the array, respectively.
```