CS180 HW6

Xilai Zhang

TOTAL POINTS

100 / 100

QUESTION 1

1 Problem 1 **25 / 25**

✓ **- 0 pts** Correct

QUESTION 2

Problem 2 25 pts

2.1 2.a **5 / 5**

✓ **- 0 pts** Correct

2.2 2.b **10 / 10**

✓ **- 0 pts** Correct

2.3 2.c **10 / 10**

✓ **- 0 pts** Correct

QUESTION 3

3 Problem 3 **25 / 25**

✓ **- 0 pts** Correct

QUESTION 4

4 Problem 4 **25 / 25**

✓ **- 0 pts** Correct

1.

```
Array<int> compare(array A){
If(A.size()==2)
    if array[i]<array[i+1]:
        array<int> C;
        C.append(1);
        C.append(A[0]);
        Return C;
    Else
        array<int> C;
        C.append(0);
        C.append(A[1]);
        Return C;
Array <int> B;
For(int i=0;i<array.size();i=i+2)
    if array[i]<array[i+1]:
        B.append(array[i+1])
    Else
        B.append(array[i])
Array<int> D= compare( array B);
    if B[D[0]*2]<B[D[0]*2+1]:
        if B[D[0]*2]>D[1]:
            D[1]=B[D[0]*2]
        D[0]=D[0]*2+1;
    Else
        if B[D[0]*2+1]>D[1]:
            D[1]=B[D[0]*2+1]
        D[0]=D[0]*2;
Return D;
}
Array <int> E= compare(A);
Max=A[E[0]];
Second_max=E[1];
```

Correctness:

We use a recursion function. In each layer we compare every two adjacent numbers, append the bigger into an array, and pass the array that contains all the bigger numbers of the two adjacent numbers to the next layer. We also back track the index of the maximum number and the value of the second maximum number. From the bottom layer up, we return a two element array, in which the first element is the index of the maximum number in array of current layer, and the second element is the value of the second largest number. From all the two number groups in each layer that contain the maximum number, we compare the number which is not the maximum number, (they are the potential candidates of the second largest number), and find the

maximum, which is the second largest number.

Complexity:
The comparisons in each layer will add up O(N/2)+O(N/4)+O(N/8)+…to be O(N). And to back track the second largest number we need O(1) comparison in each layer. Since there are in total O(logN) layers, the total complexity will be O(N)+O(logN).

1 Problem 1 25 / 25

✓ - 0 pts Correct

2.(a)
For example, if we have coins of values 10,7 and 1, and we want them add up to be 14. The greedy algorithm will take 10+1+1+1+1 which is 5 coins, will we can do 7+7 which is only 2 coins.

(b)
The maximum number of coins we can take for value C^k is C-1. If we take more than C coins of value C^k, we will have a total value greater than C^{k+1}, which can be replaced partially by taking coins of value C^{k+1}. Now, suppose the value we want to take is in between C^n and C^{n+1}, namely, $C^n <= V < C^{n+1}$. We know we cannot take coin of value greater than C^{n+1} because that will exceed the value. If we do not take coin of value C^n, then the maximum value we can have is $(1+C+C^2+C^3+\cdots+C^{n-1})$ *(C-1) which is C^n-1 because we can take at most C-1 coins of each value as stated previously. Since C^n-1 < C^n<= V, we will never get the expected value. Thus we have to take coins of value C^n until the expected value decreases to less or equal to C^n -1 , which requires us to take the maximum number of coins of value C^n. After that we will be left with some remainder value M such that C^m<= remainder value < C^{m+1} for some m, and the same arguments follows to require us to take maximum number of C^m. If we repeat this procedure, we will have the exact same greedy algorithm.

(c)
Initialize array B of integers of length C+1, where C is our expected value.
B[0]=0, all other entries are infinity.

```
For(int i=1;i<=C;i=i+1)
    For(int j=1;j<n+1;j=j+1)
        If(i<A[j])
            break
        If (B[i-A[j]]+1<B[i])
            B[i]=B[i-A[j]]+1
Return B[C]
```

Correctness:
This algorithm is dynamic programming based on the recursion that opt(C)=min[opt(C- A[n])+1] for all A[n]<=C. In this recursion, opt(C) means the minimum number of coins to take to reach value C. To find the minimum number of coins to take to reach C, we look at all the minimum number of coins to take to reach C-A[i], (1<=i<=n), from there we can take a single coin of value A[i] to reach V. Of all the V-A[i] that can reach V, we find the one that requires the minimum number of coins.

Complexity:
Array B is of length C, to fill each entry of B, we have to iterate through n items of A. So in total it is O(C)*O(n)=O(nC).

2.1 2.a 5 / 5

✓ - 0 pts Correct

2.(a)
For example, if we have coins of values 10,7 and 1, and we want them add up to be 14. The greedy algorithm will take 10+1+1+1+1 which is 5 coins, will we can do 7+7 which is only 2 coins.

(b)
The maximum number of coins we can take for value C^k is C-1. If we take more than C coins of value C^k, we will have a total value greater than C^{k+1}, which can be replaced partially by taking coins of value C^{k+1}. Now, suppose the value we want to take is in between C^n and C^{n+1}, namely, $C^n <= V < C^{n+1}$. We know we cannot take coin of value greater than C^{n+1} because that will exceed the value. If we do not take coin of value C^n, then the maximum value we can have is $(1+C+C^2+C^3+\cdots+C^{n-1})$ *(C-1) which is C^n-1 because we can take at most C-1 coins of each value as stated previously. Since C^n-1 < C^n<= V, we will never get the expected value. Thus we have to take coins of value C^n until the expected value decreases to less or equal to C^n -1 , which requires us to take the maximum number of coins of value C^n. After that we will be left with some remainder value M such that C^m<= remainder value < C^{m+1} for some m, and the same arguments follows to require us to take maximum number of C^m. If we repeat this procedure, we will have the exact same greedy algorithm.

(c)
Initialize array B of integers of length C+1, where C is our expected value.
B[0]=0, all other entries are infinity.

```
For(int i=1;i<=C;i=i+1)
    For(int j=1;j<n+1;j=j+1)
        If(i<A[j])
            break
        If (B[i-A[j]]+1<B[i])
            B[i]=B[i-A[j]]+1
Return B[C]
```

Correctness:
This algorithm is dynamic programming based on the recursion that opt(C)=min[opt(C- A[n])+1] for all A[n]<=C. In this recursion, opt(C) means the minimum number of coins to take to reach value C. To find the minimum number of coins to take to reach C, we look at all the minimum number of coins to take to reach C-A[i], (1<=i<=n), from there we can take a single coin of value A[i] to reach V. Of all the V-A[i] that can reach V, we find the one that requires the minimum number of coins.

Complexity:
Array B is of length C, to fill each entry of B, we have to iterate through n items of A. So in total it is O(C)*O(n)=O(nC).

2.2 2.b 10 / 10

✓ - 0 pts Correct

2.(a)
For example, if we have coins of values 10,7 and 1, and we want them add up to be 14. The greedy algorithm will take 10+1+1+1+1 which is 5 coins, will we can do 7+7 which is only 2 coins.

(b)
The maximum number of coins we can take for value C^k is C-1. If we take more than C coins of value C^k, we will have a total value greater than C^{k+1}, which can be replaced partially by taking coins of value C^{k+1}. Now, suppose the value we want to take is in between C^n and C^{n+1}, namely, $C^n <= V < C^{n+1}$. We know we cannot take coin of value greater than C^{n+1} because that will exceed the value. If we do not take coin of value C^n, then the maximum value we can have is $(1+C+C^2+C^3+\cdots+C^{n-1})$ *(C-1) which is C^n-1 because we can take at most C-1 coins of each value as stated previously. Since C^n-1 < C^n<= V, we will never get the expected value. Thus we have to take coins of value C^n until the expected value decreases to less or equal to C^n -1 , which requires us to take the maximum number of coins of value C^n. After that we will be left with some remainder value M such that C^m<= remainder value < C^{m+1} for some m, and the same arguments follows to require us to take maximum number of C^m. If we repeat this procedure, we will have the exact same greedy algorithm.

(c)
Initialize array B of integers of length C+1, where C is our expected value.
B[0]=0, all other entries are infinity.

```
For(int i=1;i<=C;i=i+1)
    For(int j=1;j<n+1;j=j+1)
        If(i<A[j])
            break
        If (B[i-A[j]]+1<B[i])
            B[i]=B[i-A[j]]+1
Return B[C]
```

Correctness:
This algorithm is dynamic programming based on the recursion that opt(C)=min[opt(C- A[n])+1] for all A[n]<=C. In this recursion, opt(C) means the minimum number of coins to take to reach value C. To find the minimum number of coins to take to reach C, we look at all the minimum number of coins to take to reach C-A[i], (1<=i<=n), from there we can take a single coin of value A[i] to reach V. Of all the V-A[i] that can reach V, we find the one that requires the minimum number of coins.

Complexity:
Array B is of length C, to fill each entry of B, we have to iterate through n items of A. So in total it is O(C)*O(n)=O(nC).

2.3 2.c **10 / 10**

✓ **- 0 pts** Correct

3.

Let r_i be v_i/w_i for all i between 0 and n.

Initialize array A with all values of r_i .

```
Int partion(array &A, int left, int right){
Int i=left-1;
For(int j=left;j<right-1;j=j+1)
    If A[j]<A[right]
        i=i+1
        int temp=A[i]
        A[i]=A[j]
        A[j]=temp
int temp=A[i+1]
A[i+1]=A[right]
A[right]=temp

Return i+1;
}
Int Recursive(array &A, int left, int right, int median){
Int index=partition(A, left, right)
If index==median
    Return A[index]
If index>median
    Return Recursive(A, left, index, median)
Else
    Return Recursive(A, index, right, median)

}
Initialize array B,C for return purpose.
Array<int> driver(array A, int left, int right, int midindex, int W){
Int midvalue=recursive(A, left, right, midindex);
Initialize Array D,E of values.
For(int i=1;i<=n;i=i+1)
    If  ri>midvalue:
        B.append(ri)
        D=D+wi
    Else if  ri<midvalue:
        C. append(ri)
        E=E+wi
If D<W:
    If D+wmidindex>W:
        B.append(median)
        Return B.
    Else:
        Driver(C,0,C.size()-1,C.size()/2, W-D-wmidindex)
```

```
Else if D>W:
    Driver(B,0,B.size()-1,B.size()/2, W)
Else:
    Return B
}
driver(A, 0, A.size()-1, A.size()/2,W);
return B.
```

Correctness:

This algorithm combines quick sort and greedy algorithm. We want to fill the knapsack with items that has highest ratio of value per weight. We keep taking items with highest value per weight, until the last item which we can only fill in a fraction of it due to capacity limit. First we create an array that stores value per weight of each item, then we find the median of the current array using quick sort. We add up the weights of all items that have a ratio at least bigger than the median, if this weight exceeds the capacity, we recur on these items that are bigger than the median. If this weight plus median is smaller than capacity, we store these items (because we will take them), and recur on the portion of items that are smaller than the median. If this weight plus a portion of the median is capacity, we return the array and the median.

Complexity:

To find median of an array we use quick sort and recursion. This takes O(N) complexity on average. After we find the median, we can use O(N) operation to find the weight of all items that have ratio at least bigger than the median. After that, we recur on half of the array according to different situations(as stated in the correctness section). This gives us a complexity of O(N)=O(N/2)+O(N). Using master theorem, we know that this gives us O(N) complexity in total.

3 Problem 3 25 / 25

✓ - 0 pts Correct

4.

```
Array<int> recursive(array A){
Array<int> B,C,D;
If A.size()==2
    If A[0]>A[1]:
        B.append(A[1]);
        B.append(A[0]);
    Else
        B.append(A[0]);
        B.append(A[1]);
    If(A[0]>2*A[1])
        B.append(1)
    Else
        B.append(0)
    Return B;
Array<int>E;
For(int i=index1;i<A.size()/2-1;i=i+1)
        E.append(A[i])
Array<int>F;
For(int i=index2;i<A.size();i=i+1)
        F.append(A[i])
E=recursive(E);
Int count1=E[E.size()-1]
E=E.erase(E.size()-1)
F=recursive(F);
Int count2=F[F.size()-1]
F=F.erase(F.size()-1)

Int index1=0;
Int index2=0
While(true){
If index1>=E.size()
    For(int i=index2;i<F.size();i=i+1)
        B.append(F[i])
    Break
Else if index2>=F.size()
    For(int i=index1;i<E.size();i=i+1)
        B.append(E[i])
    Break
If E[index1]<F[index2]
    B.append(E[index1])
    Index1=index1+1
Else if E[index1]>=F[index2]
    B.append(F[index2])
```

```
    Index2=index2+1
}
Int count=count1+count2;
For(int i=0;i<F.size();i=i+1)
    C.append(2*F[i])
Int index3=0;
Int index4=0;
While (true){
If index3>=E.size() or index4>=C.size()
    Break
If E[index3]<=C[index4]
    Index3=index3+1
Else if E[index3]>C[index4]
    Count=count+E.size()-index3

}
B.append(count);
Return B;
}
Array<int>G=recursive(A);
Return G[G.size()-1]
```

Correctness:

We use divide and conquer and recursion. We divide the array by 2 each time. Suppose the layer below generates two sorted arrays A and B, each has half of the length of array C in the current layer. A and B each has its count of significant inversions appended as the last element. We pop off the last element of A and B, and store them in variables count1 and count2. Now we first merge A and B into array D so that D is sorted in increasing order. Secondly we time each element of array B by 2 to generate array E, and then sort array A and E to count the number of significant inversions. We add this count to count1 and count2 to generate the total count of significant inversions up to the current layer. We append this count to D and return D. D is the result of current layer. We keep repeating this process and in the end, we will return an array that in its last element stores the total count of significant inversions. Thus we look at the last element and get the total count.

Complexity:

We have a total of log(N) layers in recursion. In each layer we merge arrays. Since we iterate through all the elements while merging, the total complexity of merge is O(N) complexity. (we have two merges as stated in the correctness section, each has O(N) complexity) So in total it is O(NlogN) complexity.

4 Problem 4 25 / 25

✓ - 0 pts Correct