Homework 6

1. Prove that the greedy algorithm for the coin change problem always yields the optimal answer when using denominations 1, 7, 7^2 = 49, and 7^3 = 343.

A1: Proof by Induction: Base step: N = 0. The function returns 0. Inductive step: Suppose that this greedy algorithm works for 0 <= N < n, for some arbitrary value n. Prove that this is true for N = n. Let N > 0 and the largest coin denomination d ∈ {7^3, 7^2, 7, 1}, where d <= N. We can assume the claim that there exists some optimal solution O for N that must contain a coin denomination of d. Based on our algorithm, we take away this largest coin denomination, yielding some optimal solution O' → O' = O - {d}, which is signified by the recursive call in the function N - d. The recursive call for N - d returns the size of O', and so the function call for N returns the size of O, with the denomination d glued back in.

Pf of Claim: Let 1 <= N < 7 → O must contain a penny at most

Let 7 <= N < 49 → O must contain a 7 somewhere, because it encompases at most the denomination of 6 pennies

Let 49 <= N < 343 → O must contain a 49 coin somewhere, because it encompases at most 6 pennies, and at most six 7 coins

Let N >= 343 → O must contain a 343 coin somewhere, because it encompases at most the combination of six 49 coins, six 7 coins, and 6 pennies, or seven 49 coins.

2. Design and analyze asymptotically a **greedy** algorithm to solve the following problems. Also give a proof that your algorithm always produces an optimal answer.
    - input: an array $L[1..n]$ containing the sizes of n print jobs to be processed by two identical printers;
- output: two queues of jobs ordered by processing time such that the total wait time is minimized.

A2: Print evens on one printer, and odds on other printer

```
Void p2 (int L[1..n])
{
        //Presort the array
        merge_Sort(L);
        //Create two arrays, one for printer 1 and the other for printer 2, size n / 2
        Int array1[L.size() / 2];// Evens
        Int array2[L.size() / 2 + 1];      // Odds
        //Zero fill the arrays
        For (int i = 0; i <= L.size() / 2 - 1; ++i)
        {
                array1[i] = 0;
```

```
            array2[i] = 0;
        }
        array2[array2.size() - 1] = 0;
        //Copy corresponding elements to the arrays
        For (int i = 0; i <= L.size() - 1; ++i)
        {
            //If it is an even element
            If (i % 2 == 0)
            {
                Array1[i] = L[i];
            } else
            {
                Array2[i] = L[i];
            }
        }
        Cout << “Printer 1: ”;
        For (int i = 0; i <= array1.size() - 1; ++i)
        {
            Cout << array1[i] << “ “;
        }
        Cout << endl;
        Cout << “Printer 2: ”;
        For (int i = 0; i <= array2.size() - 1; ++i)
        {
            Cout << array2[i] << “ “;
        }
        Cout << endl;
        return;
}
```

Analysis: I will analyze the amount of comparisons

$M(n) = \Theta(n \lg n) + 1 + \sum_{i=0}^{n/2-1}(1) + 1 + \sum_{i=0}^{n-1}(2) + 1 + \sum_{i=0}^{n/2-1}(1) + 1 + \sum_{i=0}^{n/2}(1) = \Theta(n \lg n) + 4 + n/2 + 2n + n/2 + (n/2 + 1) \rightarrow M(n) \in \Theta(n \lg n)$

Proof: Suppose that the optimal solution is not in sorted order. Then there must be two adjacent elements L[i] and L[i + 1], for some integer i, that are not in order. This means that L[i] > L[i + 1]. If we partition array L into two subarrays, array1 and array2, then this phenomenon still applies to both arrays, i.e. array1[j] and array1[j + 1] are out of order for some integer j and array1[j] > array1[j + 1]. In the context of array1, switching these two values will not affect the rest of the elements, since their positions do not directly change and therefore retain their total wait time. So, the jobs in front of j will not be affected and the jobs behind j + 1 will also not be affected. However, the change in wait time is array1[j + 1] - array1[j] < 0, since array1[j] > array1[j + 1]. The result is a decrease in wait time when you sort the adjacent elements. This happens in array2 as well, since it is unsorted.

3. Design and analyze asymptotically a **greedy** algorithm to solve the activity selection problem *that is different than the one discussed in lecture*. Also give a proof that your algorithm always produces an optimal answer.

A3: Problem: *You are given an array L[1..n] containing the sizes of n tasks. You have 3 identical printers, where each printer can only do 1 task at a time. All tasks must be completed. Minimize the total net unhappiness.*

Print evens on one printer, and odds on other printer

```
Void p3 (int L[1..n])
{
	//Presort the array
	merge_Sort(L);
	//Create three arrays, disregard gaps
	Int array1[L.size()];	// Elements by multiples of 1
	Int array2[L.size()];	// Elements by multiples of 2
	Int array3[L.size()];	// Elements by multiples of 3
	//Zero fill the arrays
	For (int i = 0; i <= L.size() - 1; ++i)
	{
		array1[i] = 0;
		array2[i] = 0;
		array3[i] = 0;
	}
	//Copy corresponding elements to the arrays
	For (int i = 0; i <= L.size() - 1; ++i)
	{
		//If it is an even element
		If (i % 1 == 0)
		{
			Array1[i] = L[i];
		} else if (i % 2 == 0)
		{
			Array2[i] = L[i];
		} else if (i % 3 == 0)
		{
			Array2[i] = L[i];
		}
	}
	Cout << “Printer 1: ”;
	For (int i = 0; i <= array1.size() - 1; ++i)
	{
		Cout << array1[i] << “ “;
	}
	Cout << endl;
	Cout << “Printer 2: ”;
```

```
        For (int i = 0; i <= array2.size() - 1; ++i)
        {
                Cout << array2[i] << " ";
        }
        Cout << endl;
        Cout << "Printer 3: ";
        For (int i = 0; i <= array3.size() - 1; ++i)
        {
                Cout << array3[i] << " ";
        }
        Cout << endl;
        return;
}
```

Analysis: I will analyze the amount of comparisons

$M(n) = \Theta(n \lg n) + 1 + \sum_{i=0}^{n-1}(1) + 1 + \sum_{i=0}^{n-1}(4) + 1 + \sum_{i=0}^{n-1}(1) + 1 + \sum_{i=0}^{n-1}(1) + 1 + \sum_{i=0}^{n-1}(1) = \Theta(n \lg n) + 5 + 8n \rightarrow M(n) \in \Theta(n \lg n)$

Proof: Suppose that the optimal solution is not in sorted order. Then there must be two adjacent elements L[i] and L[i + 1], for some integer i, that are not in order. This means that L[i] > L[i + 1]. If we partition array L into two subarrays, array1 and array2, then this phenomenon still applies to both arrays, i.e. array1[j] and array1[j + 1] are out of order for some integer j and array1[j] > array1[j + 1]. In the context of array1, switching these two values will not affect the rest of the elements, since their positions do not directly change and therefore retain their total wait time. So, the jobs in front of j will not be affected and the jobs behind j + 1 will also not be affected. However, the change in wait time is array1[j + 1] - array1[j] < 0, since array1[j] > array1[j + 1]. The result is a decrease in wait time when you sort the adjacent elements. This happens in array2 as well, since it is unsorted. There exists two elements, array2[k] and array2[k + 1], for some integer k. Then array2[k] > array2[k + 1], since for it to be sorted, the previous element must be smaller than its predecessor, i.e. array2[k] should be < array2[k + 1] for it to be considered sorted. If you switch these two elements, then its change in wait time will decrease, for array2[k + 1] - array2[k] < 0. All elements in before element k and all elements after element k + 1 remain unaffected. The same can be argued for array3. Let there be two unsorted elements array3[l] and array3[l + 1], such that array3[l] > array3[l + 1]. If you sort it, then the change in wait time is negative, which signifies a decrease in wait time. The wait times of the other elements do not change.