Hermmy Wang
704978214
Discussion 1B

Homework 5

1.

Suppose we have an array of n words. The cube of extra spaces for printing word[i] through word[j] is defined as cost[i,j]. For the words printed on each line, we define print[i]=j, which indicates that the line that starts from words[i] will end at words[j].
If the sum of lengths plus all the spaces is equal to M, the cost is 0.
If the total lengths exceeds M, the cost is ∞.
In order to minimize the cost, we want the extra spaces to distribute as evenly as possible.
minCost[1] will contain the minimum cost of printing from the first word.

dp_printing(words[]):

```
print[n]
cost[n,n]

/* Assign costs for each i, j */
for i <-- 1 to n:
  for j <-- i to n:
    if j − i + ∑_{k=i}^{n} l_k > M:
      cost[i,j] <-- ∞
    else
      cost[i,j] <-- [M − (j − i + ∑_{k=i}^{n} l_k)]^3

/* Find the optimal cost */
for i  <-- n to 1:
  minCost[i] <-- cost[i,n]
  print[i] <-- words[n]
  for j <-- n to i:
    minCost[i] <-- min(minCost[i], minCost[j]+cost[i,j])
    if minCost[i] is changed:
      print[i] <-- j

/* Store words line by line */
lastword <-- print[1]
j <-- 1
for i <-- 1 to n:
  if print[i] != lastword:
    j < j+1
    lastword <-- print[i]

  append words[i] to line[j]
  if length(line[j]) < M:
```

```
      append space to line[j]
```

printing(line[]):

```
  for i <-- 1 to length(line):
    print(line[i] + linebreak)
```

Space complexity: $O(n^2)$
Time complexity: $O(n^2)$
If we have an array of n words "abc, a, bc, abcd, ... " and the page width of 6, the cost matrix is as follows:

	1	2	3	...	n
1	27	1	∞	∞	∞
2		125	4	∞	∞
3			64	∞	∞
...				...	...
n					...

Therefore, the space requirements is $O(n^2)$, dominated by the cost matrix of n*n.
$T(n)=T(n^2)+T(n^2)+T(n)$. Therefore, the running time complexity is $O(n^2)$

2(a).

A divide-and-conquer algorithm involves dividing the array into subarrays constantly. At each point of division i, we look at the length of the longest ascending subsequence in the left array that ends with A[i] and the length of LAS that starts with A[i] (the length does not include A[i] itself). The current longest length will be the maximum of the two plus 1, including the current element.

length_start(i):

```
    length <-- 0
    for k <-- i to n:
        if A[k] > A[i]
            length <-- length+1
            i <-- k
 return length
```

length_end(i):

```
    length <-- 0
    for k <-- i to 1:
      if A[k] < A[i]
        length <-- length+1
        i <-- k
 return length
```

las_length(A[]):

```
  length <-- 0
  for i <-- 1 to n:
    curr_length <-- max(length_start(i), length_end(i)) + 1
    length <-- max(length, curr_length)
  return length
```

Time complexity: $O(n^2)$
Looking for the length that starts or ends with an element takes O(n) time. Examining the length of LAS for every element calls length_start() and length_end() n times. Therefore, the total running time is $O(n^2)$.

2(b).

We construct the longest ascending subsequence las[] using the data structure given in the problem. Assume insert(value) will insert the value at the end of las[] and find(value) will return the element less or equal to value.
For every element in the array, we will compare it with the last element in LAS:
If the current element is greater than the last in one LAS, we append this element to LAS and increment the length of LAS.
If it is less than the last element of LAS, we search in LAS to find the greater least element (or the ceiling element) and replace the ceiling element with the current element.

dp_findLAS(A[])

```
  initialize las[n]
  las[1] <-- A[1]
  length <-- 1
  for i <-- 1 to n:
    curr <-- A[i]
    if curr < las[n]:
      find(curr) <-- curr
    else if curr > las[n]:
      insert(curr)
      length <-- length+1

  return las[], length
```

Time complexity: O(nlogn)
Insertion and searching take O(logn), and we will do either insertion or searching for n times. Therefore, the total running time complexity is O(nlogn).

3.

There are |E| processors and n = |V|. Since |E| is at most $|V|^2$, log(|E|)=2log(|V|)=O(log|V|). Therefore, we need to solve MST in O(|E|)log(n) rounds. When a processor needs to communicate with others, it will at most communicate with |E|-1 processors and read (|E|-1)log(n) bits from shared memory cells. We can use the Prim's algorithm with memoization to find the minimal spanning tree.
visited[v] is initialized to keep track of the nodes that we have visited. input[e] is initialized to null, representing the bits stored in each memory cell. Every time we read a memory cell in O(logn) time, we store the reading in the array. Thus, when we need to read the same memory cell again next time, the access time would be O(1).

findMST(G):

```
mst <-- null
for every edge e:
  input[e] <-- null
for every node v:
  visited[v] <-- false

for every node v:
  if visited[v] is false:
    for every adjacent edge e:
      if input[e] is not null
        mst <-- make decision based on input[e]
      else
        input[e] <-- read()
        mst <-- make decision based on input[e]
  visited[v] <-- true
```

Time complexity:
Essentially, the algorithm will take O(|E|log(n)), or equivalently O(nlogn), time. The loop appears that it will read |E||V| times; however, every time we have read a new memory cell, we will store it in an array. There are in total |E| processors, so the times of reading are at most |E|. The access time of repeated memory cells only takes O(1). Therefore, the time complexity is O(nlogn).

4(1).

If we have a list of items, each item with a particular weight and value, and we would like to fill the Knapsack with items that maximize the value, the greedy algorithm would be calculating the value/weight and fill the Knapsack starting from the greatest value/weight ratio. If an item is too heavy to fit in the knapsack, we compare the current knapsack value and the maximum value item in the items. Change the knapsack to the more optimal one. The time complexity is dominated by quicksort, and is therefore O(nlogn).

```
compare_item(item a, item b):
  pw_a <-- a.value / a.weight
  pw_b <-- b.value / b.weight
  return pw_a > pw_b
```

```
greedy_Knapsack(items, capacity):
  initialize knapsack[]
  quicksort items by compare_item()
  for i <-- 1 to n:
    if items[i].weight > capacity:
      i <-- i + 1
      knapsack[] <-- max(knapsack[], max value in items)
    add items[i] to knapsack[i]
    capacity <-- capacity – items[i].weight

  return knapsack[]
```

4(2).

Knapsack is not amenable to a greedy solution because we might leave too much extra space in our Knapsack and thus not obtain an optimal solution.

For example,

item	**1**	**2**	**3**	**4**	**5**	**6**
weight	3	1	4	3	2	5
value	4	7	12	9	8	15
value/weight	4/3	7	3	3	4	3

Suppose we have a knapsack of capacity 10. We could fill the knapsack according to the value/weight ratio in the order of 2, 5, 6, with a total value of 30. When picking the third item in the Knapsack, there are three items with the same value/weight ratio.

Since the items are quicksorted by value/weight ratio and quicksort is not stable, there is no guarantee that the item with the minimum weight will be placed in the front.

In this case, the solution picks items 6 when there are three items with the same value/weight ratio, but it is not optimal. The actual optimal solution is to fill the knapsack in the order of 2, 5, 3, 4, with a total value of 36.

4(3).

The greedy algorithm will always get a value greater or equal to half of the optimal value when the item values are equal to item weights. Suppose we have a greedy algorithm that will take a fraction of the last item, $item_j$, then the solution obtained from this new greedy algorithm will always be at least equally optimal as the optimal solution. Since each item has the same value as its weight, then the optimal solution is bounded by the capacity or the total weight of the knapsack.

Let's denote the optimal dynamic programming solution has a value of S_{opt}, and the greedy algorithm has a final solution S_g. Suppose $2S_g$ is less than the total weight, or S_g is less than half the capacity. Either the capacity is large enough to hold twice the items, or there exists an item whose weight exceeds S_g and cannot fit in the Knapsack anymore. In the former case, the greedy algorithm equals the optimal solution. We will thus only consider the latter case.

Since the item weight equals the item value, the item value is larger than the current solution S_g, and the greedy algorithm should have picked this item and discard the original S_g. Therefore, we have reached a contradiction, and $2S_g$ has to be equal or larger than the total weight.

$$S_{opt} \leq \sum weight \leq 2S_g$$

$$S_g \geq \frac{1}{2} S_{dp}$$

Example yielding half of the optimal solution:

Imagine we have items with weight and value 1, 2, 4, 8, ... 2^n and a knapsack of capacity 2^{n+1}-2. The optimal solution is $\sum_{i=1}^{\infty} 2^i = 2^{n+1} - 2$. If the greedy algorithm first picks up 1, 2, 4, ..., 2^{n-1}, when it reaches 2^n, the knapsack can't fit 2^n anymore. Therefore, the greedy algorithm yields 1, 2, 4, ..., 2^{n-1}=2^n-1, which is half of the optimal solution is 2^{n+1}-2.

4(4).

If we have infinite duplicates of each item, the greedy algorithm will always pick the same item with the highest value/weight ratio. The reasoning process is similar to 4(3). Since each item has the same value as its weight, then the optimal solution is bounded by the capacity or the total weight of the knapsack. Each item has an infinite number of copies, so the greedy algorithm will pick the same item over and over again. We need to show $2S_g$ cannot be less than the capacity, S_g cannot be less than half the capacity. Imagine S_g is less than half of the capacity, the algorithm will not stop here: there are infinite copies of the current chosen items, so the knapsack can fit one more S_g. Therefore, we have reached a contradiction. S_g cannot be less than half the capacity.

$$S_{opt} \leq \sum weight \leq 2S_g$$

$$S_g \geq \frac{1}{2} S_{dp}$$