CS180 Solution 5

1. Consider the problem of printing a paragraph with a mono-spaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1, l_2, ..., l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. If a given line contains words i through j, where $i \le j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be non-negative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the *cube* of sum of the numbers of extra space characters at the ends of lines. Give an algorithm to print a paragraph of n words on a printer in a way that minimizes the above sum. Analyze the running time and space requirements of your algorithm.

   Solution:

   Define the cost of including a line containing words i through j in the sum:

$$LC[i,j] = \begin{cases} \infty & \text{if words } i,\dots,j \text{ don't fit} \\ 0 & \text{if } j = n \text{ (because the last line costs 0)} \\ (M - j + i - \sum_{k=i}^{j} l_k)^3 & \text{otherwise} \end{cases}$$

   We want to minimize the sum of LC over all lines of the paragraph. Let $OPT[j]$ denote the cost of an optimal arrangement of words $i, \dots, j$.

$$OPT[j] = \begin{cases} \min_{1 \le i \le j}\{OPT[i-1] + LC[i,j]\} & \text{if } j > 0 \\ 0 & \text{if } j = 0 \end{cases}$$

   This allows us to have a dynamic programming algorithm. The algorithm tries to compute the elements of a one-dimensional array $OPT[0...n]$, where $OPT[0] = 0$, and for i from 1 to n the algorithm computes $OPT[i]$ using the above recursive formula. The final output of the algorithm is $OPT[n]$. The time complexity is $O(n^2)$ since calculating each $OPT[i]$ needs $O(N)$ time.

2. The longest ascending subsequence problem is defined as follows. Given an array $A[1..n]$ of natural numbers, find the length of the longest ascending subsequence of A. (A subsequence is a list $A[i_1]$,$A[i_2]$,...,$A[i_m]$ for some $1 \le i_1 < i_2 < ... < i_m \le n$. The value m is called the length of the subsequence. Such a subsequence is called *ascending* if $A[i_1] \le A[i_2] < ... < A[i_m]$. For simplicity assume all the values in the array are distinct.

   (a) Design a divide-and-conquer algorithm for solving the longest ascending subsequence problem in time $O(n^2)$. (Hint: Solve a tougher problem. For each position in the array find the cardinality of the longest sequence that ends up with it, and the longest sequence that starts with it. Notice that $n^2 + 2(n/2)^2 + 4(n/4)^2 + \dots$ is $O(n^2)$.)

   Solution:

```
FINDMAXSEQLEN(A[1..n], start, end)
  endingSeqMaxLen = [1, 1, ..., 1]
  for i ← 1...n
      for j ← 1...i-1
          max = 0
          if A[j]<A[i] and max < endingSeqMaxLen[j]
              max = endingSeqMaxLen[j]
          endingSeqMaxLen[i] = max + 1
  return max(endingSeqMaxLen)
```

(b) Design a dynamic programming algorithm in time $O(n \log n)$. (Hint: Assume you have a data structure in which *insert(value)* and *find(value)* is $O(\log n)$ operations where the latter operation returns the highest value that was inserted less or equal to $value$.)

Solution:

```
LAS(A[1..n])
  tails ← [0, 0, ..., 0] // length: n+1
  tails[1] ← A[1]
  $l$ ← 1 // max subsequence length
  for i ← 2...n
      if A[i] < tails[1]
          tails[1] ← A[i]// update the smallest value
      elif A[i] > tails[n]
          $l$ += 1
          tails[$l$] = A[i]
      else
          tails[BINARYSEARCH(tails, $l$, A[i])] ← A[i]

BINARYSEARCH($B[1..n+1]$, $r$, $v$)
  // Find the index of the ceil of the $v$ by binary search
  $l$ ← 0 // search in $B[l...r]$
  while $r > l$
      $m = l + (r-l)//2$
      if $A[m] \geq v$
          $r \leftarrow m$
      else
          $l \leftarrow m$
      return $r$
```

Time complexity: $O(n)$ iterations. Each iterations takes $O(n \log n)$ for binary search. $O(n \log n)$ in total.

3. In parallel computing, to solve a problem, processors communicate with others round by round. In each round, processors can concurrently read or exclusively write(CREW) on shared memory, i.e., multiple processors can read a shared memory cell, but only one can write to a given cell at a time. Given a weighted undirected graph $G = (V, E)$, where $|V| = n$ and edge weight are distinct. There are $|E|$ processors and each processor P_i has the input of a unique edge $E_i = \{u, v\}$ and its weight $w(E_i)$. Every shared memory cell can hold $O(\log n)$ bits. To solve the minimum spanning

tree of G, because every processor has only partial information of G, it needs to communicate with others and then decide whether its input edge is in the MST or not. Design an algorithm to solve MST in $O(n \log n)$ *rounds*. Values in the input to the problem are of size $O(\log n)$ bits. A cell can hold at most constant number of such values, i.e., the whole description of G cannot be written in a single cell. The output is each processor determines whether its edge is in the MST or not. (Hint: Notice that in this setting if we have a polynomial of n number of processor each holding a value then the minimum value can be found in $O(\log n)$ time. To use this assumption you have to show how to do it.)

Solution:

We consider an distributed implementation of Kruska's algorithm. Processor compute the MST in n rounds, in each round, there is exactly one processor decide its input is added in the set S, i.e., a subset of the MST during the construction. In Kruska's algorithm, we sort all edges at the beginning and process them one by one. In distributed setting, we can achieve this by asking every possible edges jointly compute the minimum edge, and then add it into S. To implement the algorithm, we need a distributed $O \log n$ union find data structure and an algorithm that find the minimum value among n^2 processes in $O(\log n)$ rounds.

The implement of distributed Union is almost the same as classical version except we use n shared memory cells $P[1..n]$ where $P[i] = (\text{parent}(i), \text{count}(i))$. Parent($i$) stores the parent node of node i and count(i) stores the number of nodes of subtree with root i. The implementation of UNION and FIND is same. Since we use CREW shared memory, multiple process can find their roots at the same time but only one process can execute the UNION operation for safety. The idea of the implementation of DISTRIBUTEDKRUSKA is: in each round, each processor P_E checks whether its edge E has end points from two different component. If so, P_E participate FindMin with input $w(E)$ to check whether it is the minimum of all edges that connects two different compoent. Otherwise, E is an edge in the same component, and P_E just skip and return its status. In each round, there is exactly one process with minimum weights connects two different component. Hence, after $n-1$ rounds the algorithm terminates and we have a MST.

```
DISTRIBUTEDKRUSKA($E = (i, j)$)     // code for processor $P_E$
  for round ← 1...$n-1$
      if status($E$) = unexplored   //E is not explored. Initially, every edge has the status unexplored
          $r_i$ ← FIND($i$)     //the component that $i$ belongs to
          $r_j$ ← FIND($j$)
          if $r_i = r_j$
              status($E$) ← false        //E is not in MST
          else
              res← FindMin($w(E)$)    //find the min edge that connects two different components
              if res = $w(E)$    //E is the min edge connects two different components
                  UNION($i, j$)
                  status($E$) ← true    //E is in MST
      return status($E$)
```

Next, we show how to implement FindMin to calculate the minimum edge among at most n^2 processes in $O(\log n)$ rounds. The idea is to use a perfect binary decision tree with n^2 leaves. At the beginning, each process P_E is assigned to a unique leaf. The algorithm contains $\log(n^2) = O(\log n)$

rounds. At round i, the process P_E writes its weights $w(E)$ on the node it belongs to (the node is a leaf in round 0), and then check its sibling on round i, if the sibling location is empty or has smaller weights, P_E proceeds to round $i+1$. Otherwise, P_E loose the comparison and return an empty value. So each round, the number of inputs is reduced by half, and the final winner after $\log(n^2)$ rounds that writes on the root of the tree has the the minimum edge among all processors. Since Find and Union also takes $O(\log n)$ rounds, each iteration of DISTRIBUTEDKRUSKA takes $O(\log n)$ rounds, and the total rounds of this algorithm is $O(n \log n)$.

4. After class on Wednesday a student asked me why not use the "obvious" greedy algorithm to solve the Knapsack problem. What will the obvious greedy algorithm be? Give a counter example to show that Knapsack is not amenable to a greedy solution (In Dynamic programming we do not fill the Knapsack incrementally making commitments as we go. We either know the whole solution or not. We cannot determine a partial solution that we know can be extended.). Show that nevertheless if the Hiker is in a hurry and she fills the Knapsack using the greedy algorithm then the value she carries is greater equal than half of the optimal value.

   Solution:

```
GREEDYKNAPSACK($v_1, v_2, ..., v_n, W$)
    $v_1, v_2, ..., v_n \leftarrow$ sort items in increasing value
    add $v_1, v_2, ..., v_k$ into bag until $v_{k+1}$ can not be added.
    if $\sum_{i=1}^{k} v_i > v_{k+1}$
        return $\{v_1, v_2, ..., v_k\}$
    else
        return $v_{k+1}$
```

   Given the input of items: $(\epsilon, W/2, W/2)$ and weight capacity of W, where ϵ is an arbitrary small positive small constant. The GREEDYKNAPSACK will give a $\frac{1}{2} + \frac{\epsilon}{W}$ of the optimal solution. Next we explain GREEDYKNAPSACK always give a $\frac{1}{2}$ approximation of the optimal solution. Let opt be the optimal solution and $V = \sum_{i=1}^{k} v_i$, Since we have volume equals to the value, we have $W = V + (W - V) \geq$ opt. $(W - V)$ is the empty space in the knapsack after we add $v_1, v_2, ..., v_n$. Therefore, $v_{k+1} > (W - V)$ and so $V + v_{k+1} >$ opt. Therefore, either V or v_{k+1} is greater than $\frac{1}{2}$opt and the greedy algorithm will use the max one. And the proof does not depends on the number of items we have for each type.