CS180 Solution 5

1. You are given n songs $\{1, 2, \ldots, n\}$ and 2 preference lists in array $A[1..n]$ and $B[1..n]$. Each $A[i]$ or $B[i]$ contains a positive integer and $A[i] = k$ means the song k is ranked at i^{th} position in list A. An inversion between A and B is a pair (i, j) such that song i is ranked before song j in list A, but song j is ranked before song i in list B. To count the number of inversions, we can permute the songs in both lists according to the index of list A. More specifically, the permutation is defined by $A[i] \mapsto i$ for all i. For example, $A = \{1, 3, 2, 4\}$ and $B = \{2, 3, 1, 4\}$. The permutation is defined as follows:

$$\begin{tikzcd}
1 \arrow[r] & 1 \\
2 \arrow[dr] & 2 \\
3 \arrow[ur] & 3 \\
4 \arrow[r] & 4
\end{tikzcd}$$

The permutation maps $A[i]$ to i for every number

After the permutation, the lists become $A' = \{1, 2, 3, 4\}$ and $B' = \{3, 2, 1, 4\}$.

(a) Prove that the number of inversions does not change after this permutation

(b) Prove that the number of inversions does not change if we apply an *arbitrary* permutation. Especially the sequence that get all integers in array A to be in order.

Solution:

We just prove (b) because (a) is a special case of (b). Consider a simple permutation P_{i-j} which is the identity permutation except we swap i and j, i.e., $P_{i-j}(k) = k$ if $k \neq i, j$; $P_{i-j}(i) = j$ and $P_{i-j}(j) = i$.

Lemma 1. *The number of inversions does not change after applying the permutation P_{i-j}.*

Proof. The number of inversions that does not contain i or j is not change after the permutaion since the location of any pair is the same. An inversion (i, k) before applying the permutation implies an inversion (j, k) since the location of i holds the value j after the permutaion. Similarly, an inversion (j, l) before applying the permutation implies an inversion (i, l) after. Hence, the number of inversions including i or j does not change and so, the total number does not change too. □

Note that any permutation P' can be constructed from the identity permutation by applying $n-1$ swaps. Specifically,we swap the value of $P'(i)$ with $P(i)$ for $i \in \{1, \ldots n-1\}$ where P is the identity permutation after applying $(i-1)^{th}$ swap. Therefore, P' is equivalent to the composition of $(n-1)$ permutations with the form of P_{i-j} for some i and j. By lemma 1, the number of inversions will not be changed if we apply P'.

2. In the weighted interval scheduling problem, you are given three arrays $S[1..n]$, $F[1..n]$ and $W[1..n]$ listing the starting times, finish times and weights for each job. At the cost of sorting we assume that job i has the i^{th} lowest termination time so array F is a sorted array. We defined $p(j)$ to be the prefix of sorted jobs $\text{job}_1 \ldots, \text{job}_{p(j)}$ such that these are all the jobs between $\text{job}_1 \ldots, \text{job}_{p(j)}$ that do not intersect with the interval of job_j. Give an algorithm to calculate all $p(j)$'s in $o(n^2)$ time, where "little o" means a function that grows strictly slower than n^2.

   Solution:

   We use binary search to find the location of $p(j)$. Calculating each $p(j)$ needs $O(\log n)$ time and so it takes $O(n \log n)$ time to calculate all $p(j)$.

```
COMP($\text{job}_j$)
  $a \leftarrow 1, b \leftarrow n$
  $k \leftarrow (a+b)/2$
  while $S[j] \notin (F[k], F[k+1])$
      if $S[i] > F[k]$
          $a \leftarrow k/2$
      else
          $b \leftarrow k/2$
  return $p(j) \leftarrow k$
```

3. Given n items and each item i has a integer value v_i. Design an algorithm that determine the most fair division of the n items between two persons, i.e., a division such that the difference between the sum of the values each person obtains is minimized.

   Solution:

   Let the total value be $s = \sum v_i$. The problem can be reduced to the knapsack problem by set the desired value to $s/2$. We find a subset of items such that maximize the total value with the restriction it is smaller than $s/2$. This knapsack solution defines a division by give the subset to one person and the rest to the other. Moreover, such division gives us the best division, otherwise one of the person in the division will get a total value that is closer to $s/2$ than the knapsack solution, a contradiction. The algorithm runs in $O(ns)$ time.

4. 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 $i \leq j$, 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:

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

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 $1, \ldots, j$.

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

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 $O(n^2)$.

5. A subsequence is obtained from a string by extracting a subset of elements, but keeping them in the same order. The elements of the subsequence need not be contiguous in the original string. For example, `A`, `NA`, and `BAAA` are subsequences of the string `BANANA`. A common subsequence of two strings A and B is a subsequence of both A and B. Let arrays $A[1..m]$ and $B[1..n]$ represents two strings. Describe an algorithm to compute the length of the *longest* common subsequence of A and B. For example $A =$ `abazdc` and $B =$ `bacbad`, then the longest common subsequence is `abad` and has length of 4.

   Solution:

   Let LCS$[i,j]$ be the length of the longest common subsequence for the subarray $A[1..i]$ and $B[1..j]$. If we know $A[i] = B[j]$ then we can match these two and solve the subproblem for LCS$[i-1, j-1]$. Or we don't match these two location and remove the $A[i]$ or $B[j]$ to solve the subproblems. Hence, we have the following recurrence formula.

$$\text{LCS}[i,j] = \begin{cases} \text{LCS}[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ max \begin{cases} \text{LCS}[i-1,j] \\ \text{LCS}[i,j-1] \end{cases} & \\ 0 & \text{if } i = 0 \text{ or } j{=}0 \end{cases}$$

   The solution of the subproblem LCS$[i,j]$ depends on the location of LCS$[i-1,j-1]$, LCS$[i-1,j]$ and LCS$[i,j-1]$. Hence, we can update the subproblems from left to right and then from top to bottom. The algorithm takes $O(mn)$ time. To recover the actual subsequence, we can simply make LCS contains both the length of the sequence and sequence by change the formula to be LCS$[i,j] \leftarrow$(LCS$[i-1,j-1]+1, s+$"$A[i]$"), where s is subsequnce from the LCS$[i-1,j-1]$ and the second "+" is the concatenation operation. The time complexity of the algorithm is $O(mn)$.

```
LCS($A[1..m], B[1..n]$)
  for $i \leftarrow 0$ to $m$
      LCS$[i,0] \leftarrow 0$
  for $j \leftarrow 0$ to $n$
      LCS$[j,0] \leftarrow 0$
  for $i \leftarrow 1$ to $m$
      for $j \leftarrow 1$ to $n$
          if $A[i] = b[j]$
              LCS$[i,j] \leftarrow$LCS$[i-1,j-1]$+1
          else
              LCS$[i,j] \leftarrow max \begin{cases} \text{LCS}[i-1,j] \\ \text{LCS}[i,j-1] \end{cases}$
  return LCS$[m,n]$
```