Homework 5

Due date: 2/10/2021 right before class

Problem 1

Consider a blob of text in need of formatting. Suppose our text consists of a sequence of words, $W = w_1, w_2, ..., w_n$, where w_i consists of c_i characters. We have a maximum line length of L . A formatting of W consists of a partition of the words in W into lines. In the words assigned to a single line, there should be a space after each word except the last; and so if $w_j, w_{j+1}, ..., w_k$ are assigned to one line, then we should have

$$\sum_j^{k-1} + c_k \leq L.$$

Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the squares of the slacks of all lines (including the last line) is minimized. (Chapter 6 question 6 in our book).

0.1 Solution 1

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

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

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 $i, \ldots, j$.

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

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 $O(n^2)$.

Problem 2

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by

sawing up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

Analyze both time and space complexity of your algorithm.

0.2 Solution 2

```
def cutRod(price, n):
    if(n <= 0):
        return 0
    max_val = -infinity

    # Recursively cut the rod in different pieces
    # and compare different configurations
    for i in range(0, n):
        max_val = max(max_val, price[i] + cutRod(price, n - i - 1))
    return max_val
```

Listing 1: algo

Time complexity involves calling cutRod on $n-1, n-2 \ldots 1$ chunks, and this happens n times. Overall this is an $O(n^2)$ algorithm. We are keeping the max val plus a recursive call tree of n elements at each point in time, making the space complexity $O(n)$

The algorithm involves recursive calls that span all options of cutting the rod into pieces. At each recursion we keep the maximum value of cutting the rod into one of the possible configurations, eventually finding the best one.

Problem 3

We are given a string of letters $x_1, x_2, ..., x_n$ from the English alphabet. For each x_j we would like to find the longest proper prefix of $x_1, x_2 ... x_j$ such that it is also a suffix of it, i.e., find a number $P(j) < j$ such that $x_1, x_2 ... x_{P(j)} = x_{j-P(j)+1}, x_{j-P(j)+2} ... x_j$ and $P(j)$ is the largest one with this property. For example, the longest proper prefix of *ababa* that is also a suffix is *aba*. Give an algorithm to calculate the array P whose j'th entry is $P(j)$, and analyze its complexity.

For example, given the input *ababaca*, $P = [0, 0, 1, 2, 3, 0, 1]$.

(a) Design a "brute force" algorithm and argue it's time complexity.

Solution:

The brute force algorithm is check every possible value of $P(i)$ for each $x_1x_2\dots x_i$. The algorithm runs in $O(n^2)$ time.

```
BRUTEFORCE(x_1, x_2, ..., x_n)
  for i ← 1 to n
      P[i] ← 0
      for j ← 1 to i − 1
          if x_1 x_2 ... x_j = x_{i−j+1} x_{i−j+2} ... x_i
              P[i] ← max(P[i], j)
```

(b) Design an $O(n)$ algorithm using *dynamic programming*. You need to write the recursion first.

Solution:

The problem is called to calculating "the failure function" of a pattern string, which is a part of the KMP string matching algorithm. The idea of the recursion is, to calculate $P[i]$, we try to extend the longest previous valid prefix $x_{1..}x_{p[i-1]}$, if it fail, we try to extend the longest proper suffix of $x_{1..}x_{p[i-1]}$, which is $x_{1..}x_{p[p[i-1]]}$. If it fails again, we repeat the process until no previous prefix can be extend and so, it returns 0. Define $P^c[i]$ inductively: $P^0[i] = i$, and $P^c[i] = P[P^{c-1}[i]] = \overbrace{P[P[..[P[\,i]]..]]}^{c}$. The recursion of the problem is

$$P[i] = \begin{cases} 0 & \text{if } i = 0 \\ max \begin{cases} P[i-1]+1 & \text{if } x_i = x_{p[i-1]+1} \\ P[P[i-1]]+1 & \text{if } x_i = x_{p[p[i-1]]+1} \\ .. & .. \\ P^c[i-1]+1 & \text{if } x_i = x_{p^c[i-1]+1} \end{cases} & \text{otherwise} \end{cases}$$

The recursion is calculated by the following algorithm:

```
PREFIXFUNCTION(x_1, x_2, ..., x_n)
  P[1] ← 0
  k ← 0
  for i ← 2 to n
      while k > 0 and x_i ≠ x_{k+1}
          k ← P[k]
      if x_i = x_{k+1}
          k ← k + 1
      P[i] ← k
```

(c) Argue that the dynamic programming algorithm you wrote can actually be executed in linear time.

Solution:

We show that the above algorithm runs in linear time. The only difficulty of the analysis is to show that the while loop will be executed at most n times in total. We first notice that the number of k will increased at most $n-1$ times according the for loop. Moreover, since k can not be smaller than 0, the number time while k is decreasing is bounded by $n-1$, and so the number of entering the while loop to decrease k is bounded by $n-1$. Since the while loop will be executed at most $n-1$ times, the algorithm runs in $O(n)$ time.

Problem 4

There are $n = 2^k$ athletes to play tennis in a tournament. You are asked to design a competition schedule that meets the following requirements:

1. (1) Each player must play once with each of the other $n-1$ players;
2. (2) Each player can only participate in one play a day;
3. (3) The tournament ends in $n-1$ days.

According to this requirement, the game schedule is designed as a table with n rows and $n-1$ columns. In the i-th row of the table, the j-th column is filled with the player that the i-th player encountered on the jth day. where $1 \leq i \leq n$, $1 \leq j \leq n-1$. Design an algorithm to construct this schedule.

Design an algorithm (if possible) when the number of players n is any number. Not answering this (hard question) will not distract from the credit of this problem. So not to worry if you struggle.

0.3 Solution 4

Recursive algorithm: If $n = 2$, schedule the one match: $A[1,1] = (1,2)$. Else, schedule all matches among the first $\frac{n}{2}$ players and place those in block $A[1 \cdots (\frac{n}{2}-1), 1 \cdots \frac{n}{4}]$, and schedule all matches among the second $\frac{n}{2}$ players and place those in block $A[1 \cdots (\frac{n}{2}-1), (\frac{n}{4}+1) \cdots \frac{n}{2}]$.

The remaining matches to be played each have one player from the first half and one player from the second half. Match these up one-to-one and use these matches on day $\frac{n}{2}$, then, for each next day, rotate the matching.

Programatically: Call these player halves $x_0, \ldots, x_{(\frac{n}{2}-1)}$ and $y_0, \ldots, y_{(\frac{n}{2}-1)}$. For $i = 0 \cdots (\frac{n}{2}-1)$ and $j = 0 \cdots (\frac{n}{2}-1)$, set $A[\frac{n}{2}+i, j+1] = (x_j, y_{(j+i) \mod n/2})$. The only thing left to check is that no player appears twice in some row, and every player plays every other player. We can see this using the induction hypothesis and the the fact that for the last $\frac{n}{2}$ rows: player x_j always plays in column j and player y_j always plays in column $(j-i) \mod \frac{n}{2}$.

Both algorithms have $O(nlogn)$ time complexity.