Homework 2

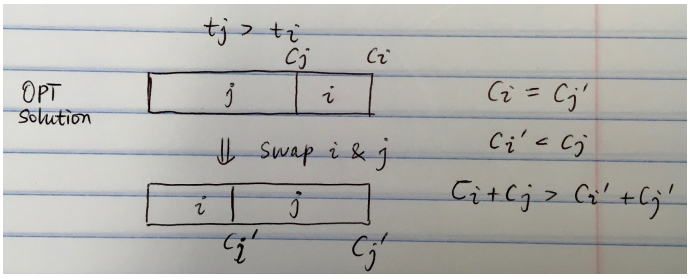
Problems:

1. Job scheduling: given n jobs where job i requires time t_i and has a weight w_i characterizing the importance of the job, we want to schedule the jobs on a single machine without overlap that minimizes

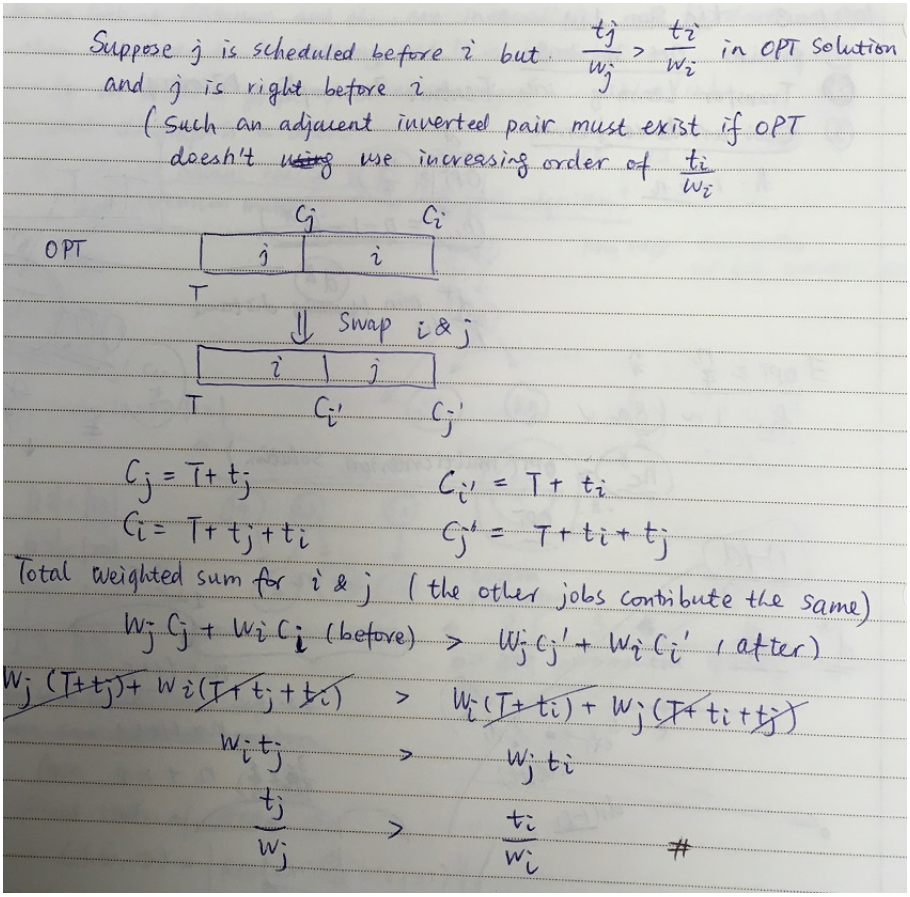
$$\sum_i w_i C_i,$$

where C_i is the finishing time of job i. Suppose we start at time 0.

Solution: When w_i's are the same the optimal solution is to take the shortest job first. To see that, consider the optimal solution in which the ordering does not follow "shortest job first", say there is a pair i, j such that $t_i < t_j$ and j is placed right before i. Now if we swap the two jobs. Then we have the total sum of completion time to be smaller – all other jobs have the same completion time, and job j's completion time *after* the swap is the same as job i's completion time *before* the swap but the job i's completion time after the swap is *strictly* smaller than the completion time of job j before the swap. Thus the total time is smaller after the swap. See the picture below.



When the weights are not the same, we will sort the jobs by t_i/w_i and schedule them in increasing order. This can be proved to be optimal by the same argument above. Details below:



2. Consider the following problem. The input consists of n skiers with heights $p_1, \cdots, p_n$, and n skies with heights $s_1, \cdots, s_n$. The problem is to assign each skier a ski to to minimize the average difference between the height of a skier and his/her assigned ski. That is, if the ith skier is given the $\alpha(i)$th ski, then you want to minimize

$$\sum_{i=1}^{n} |p_i - s_{\alpha(i)}|.$$

   (a) Consider the following greedy algorithm. Find the skier and ski whose height difference is minimized. Assign this skier this ski. Repeat the process until every skier has a ski. Prove of disprove that this algorithm is correct.

   (b) Consider the following greedy algorithm. Give the shortest skier the shortest ski, give the second shortest skier the second shortest ski, give the third shortest skier the third shortest ski, etc. Prove of disprove that this algorithm is correct.

Answer:

   (a) Not correct. A counter example is shown below:

   (b) Correct.
   Proof: Sort P and S to be ascending order, Then our solution O would be $(p_1, s_1), (p_2, s_2), \ldots, (p_n, s_n)$. Suppose there is an optimal solution O^* with the first i-1 pairs the same as O, and they differ on the ith pair. Therefore, in O^* it would be $(p_1, s_1), (p_2, s_2), (p_{i-1}, s_{i-1}), (p_i, s_j), \ldots, (p_k, s_i), \ldots$, and $p_i \leq p_k, s_i \leq s_j$. If we swap it to be $(p_1, s_1), (p_2, s_2), (p_{i-1}, s_{i-1}), (p_i, s_i), \ldots, (p_k, s_j), \ldots$,

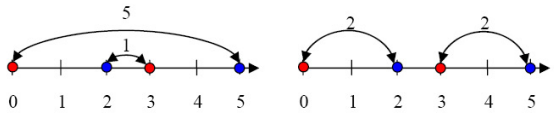


Figure 1: A counterexample

then we get a new solution O^{**} with the first i pairs the same as O. Next let's show O^{**} is not worse than O^*:
Claim: $|p_i - s_i| + |p_k - s_j| \leq |p_i - s_j| + |p_k - s_i|$
Proof: do case analysis. because $p_i \leq p_k, s_i \leq s_j$, there are only $P_4^4/4 = 6$ cases.

$p_i \leq p_k \leq s_i \leq s_j$	left=$s_i + s_j - p_i - p_k$, right=$s_i + s_j - p_i - p_k$	$\Rightarrow left = right$
$s_i \leq s_j \leq p_i \leq p_k$	left=$p_i + p_k - s_i - s_j$, right=$p_i + p_k - s_i - s_j$	$\Rightarrow left = right$
$s_i \leq p_i \leq p_k \leq s_j$	left=$p_i + s_j - s_i - p_k$, right=$s_j + p_k - s_i - p_i$	$\Rightarrow left \leq right$
$p_i \leq s_i \leq s_j \leq p_k$	left=$s_i + p_k - p_i - s_j$, right=$s_j + p_k - s_i - p_i$	$\Rightarrow left \leq right$
$p_i \leq s_i \leq p_k \leq s_j$	left=$s_i + s_j - p_i - p_k$, right=$s_j + p_k - s_i - p_i$	$\Rightarrow left \leq right$
$s_i \leq p_i \leq s_j \leq p_k$	left=$p_i + p_k - s_i - s_j$, right=$s_j + p_k - s_i - p_i$	$\Rightarrow left \leq right$

To sum up, solution O^{**} is no worse than O^*, and has the first i pairs the same as our solution O. After finite such steps, we can swap an optimal solution O^* to our solution, and still maintain optimality. Therefore, our solution O is optimal.

3. The input to this problem consists of an ordered list of n words. The length of the ith word is w_i, that is the ith word takes up w_i spaces. (For simplicity assume that there are no spaces between words.) The goal is to break this ordered list of words into lines, this is called a layout. Note that you can not reorder the words. The length of a line is the sum of the lengths of the words on that line. The ideal line length is L. No line may be longer than L, although it may be shorter. The penalty for having a line of length K is $L - K$. There are two ways to define the total penalty as shown below. The problem is to find a layout that minimizes the total penalty.

   (a) In the first definition, the total penalty is the sum of the line penalties.

   (b) In the second definition, the total penalty is the maximum of the line penalties.

   Prove or disprove that the following algorithm gives the correct solution for each of the problems above.

```
For i = 1 to n
      Place the ith word on the current line if it fits
            else place the ith word on a new line
```

   Answer:

   (a) Correct.
   We prove the correctness of the algorithm by saying that it always stays ahead. First we should prove that: after placing n words to a page, this algorithm requires no more space than other algorithms. We use mathematical induction to prove this claim.
   Basis step is trivial because this algorithm doesn't have a whitespace before the first word. The inductive hypothesis is that suppose after placing k words to a page, this algorithm require no more space than other algorithms and we consider when we add the $(k+1)$ word. If we can place the $(k+1)$ word in the same line, because there is no whitespace

between the k and the $(k+1)$ words, we can conclude that after placing $k+1$ words, this algorithm requires no more space than other algorithms. Otherwise, if we have to place the $(k+1)$ word in a new line, and we should consider two cases,

i. When after placing k words to a page, the number of lines of this algorithm is the same as other algorithms, in this case, other algorithms should also place the $(k+1)$ word in a new line, so after placing $k+1$ words, this algorithm requires no more space than other algorithms.

ii. If the number of lines of this algorithm is less than other algorithms, we should start a new line when placing the $(k+1)$ word, but for other algorithm it may be not required to do that. However, the $(k+1)$ word starts at the beginning of the new line, while for other algorithms there must be some words in that line, thus we can also say that after placing $k+1$ words, this algorithm requires no more space than other algorithms.

The sum of the line penalties is equal to all spaces subtract the spaces taken up by word. So this algorithm is optimal.

(b) Not correct. A counter example is shown below:

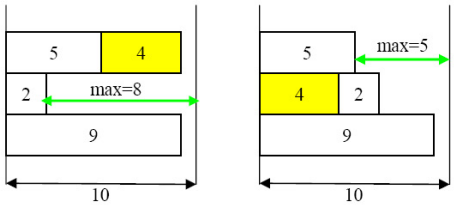


Figure 2: A counterexample

4. Given a graph G, each edge (i, j) has a weight $r_{ij} > 0$, the goal is to find whether there is a cycle such that the multiplication of the weights on the edges of the cycle is greater than 1. Find a polynomial time algorithm.

   Solution: This is similar to the all pairs shortest paths problem. But one must be careful not to fall into chicken and egg problem. To do that, we add another parameter i as the length of the trading cycle we find. Define $R[i, j, r]$ as the highest ratio of trading for a simple path from i to j of length at most r. The subproblem structure is that

$$R[i, j, r] = \max_{\forall(i,k)} r_{ik} \cdot R[k, j, r-1]$$

   The original problem asks for $\max_{\forall(i,j)} r_{ij} \cdot R[j, i, n-1]$.

5. A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, 'bananaananas' is a shuffle of 'banana' and 'ananas' in several different ways depending on the way the two strings are overleaved. Given three strings $A[1..m]$, $B[1..n]$ and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

   Solution: We use dynanmic programming. Define $S[i, j]$ to be 1 if $A[1..i]$, $B[1..j]$ can be used to get $C[1..i+j]$ by a shuffle and 0 otherwise. We can verify $S[0, j]$ for all j and $S[i, 0]$ for all i easily. Further, we have the following recursive structure:

$$S[i, j] = 1 \text{ if either } S[i-1, j] = 1 \text{ and } A[i] = C[i+j] \text{ or } S[i, j-1] = 1 \text{ and } B[i] = C[i+j],$$

   and 0 otherwise. The running time of the algorithm is $O(n^2)$.

6. Assume that you have a subroutine IsWord that takes an array of characters as input and returns True if and only if the string is a valid English word. Design efficient algorithms for the following problems and bound the number of calls to IsWord.

   (a) Given an array $A[1 \cdots n]$ of characters, compute the number of partitions of A into words. For example, given the string ARTISTOIL, your algorithm should return 2, for the partitions ARTIST·OIL and ART·IS·TOIL.

   Answer: Define subproblem $S[i, j]$, $i \le j$, to be the number of partitions of $A[i \cdots j]$ into words. Now we have $W[i, j]$, $i \le j$, to be 1 if IsWord on $A[i \cdots j]$ is True and 0 otherwise. Also $S[i, i] = W[i, i]$. The recursive structure is now defined as

$$S[i, j] = \sum_{k=i}^{j-1} W[i, k] \cdot S[k+1, j] + W[i, j].$$

   The running time is $O(n^3)$.

   Remark that if one uses

$$S[i, j] = \sum_{k=i}^{j-1} S[i, k] \cdot S[k+1, j] + W[i, j].$$

   There are issues with double counting.

(b) Given two arrays $A[1 \cdots n]$ and $B[1 \cdots n]$ of characters, decide whether A and B can be partitioned into words at the same indices.

Answer: Define subproblem $S[i, j]$, $i \leq j$, to be 1 if $A[i \cdots j]$ and $B[i \cdots j]$ can be partitioned into words at the same indices and 0 otherwise. $W_A[i, j]$, $i \leq j$, to be 1 if IsWord on $A[i \cdots j]$ is True and 0 otherwise. Similarly define $W_B[i, j]$.

Now $S[i, j] = 1$ if there is an k taking values between i to $j - 1$ such that

$$W_A[i, k] = 1 \wedge W_B[i, k] = 1 \wedge S[k + 1, j] = 1,$$

or

$$W_A[i, j] = 1 \wedge W_B[i, j] = 1.$$

The running time is $O(n^3)$.

(c) Given two arrays $A[1 \cdots n]$ and $B[1 \cdots n]$ of characters, compute the number of different ways that A and B can be partitioned into words at the same indices.

Answer: Define $H[i, j]$, $i \leq j$, to be the number of different partitions of $A[i \cdots j]$ and $B[i \cdots j]$ into words at the same indices.

Define an indicator variable with a predicate P as $I(P) = 1$ if the predicate P is true and 0 otherwise.

Now we have,

$$N[i, j] = \sum_{k=i}^{j-1} I(W_A[i, k] = 1 \wedge W_B[i, k] = 1) \cdot N[k + 1, j] + I(W_A[i, j] = 1 \wedge W_B[i, j] = 1).$$

The running time is $O(n^3)$.

Update Nov. 2nd, credit to Hatsune Miku.

(a) *Given an array $A[1 \ldots n]$ of characters, compute the number of partitions of A into words.*

Answer: Instead of a two–index DP $S[i, j]$, we use a *suffix* DP with a single index. Define $F[\ell]$ for $\ell \in \{1, \ldots, n + 1\}$ to be the number of partitions of $A[\ell \ldots n]$ into words. Set the base case $F[n + 1] = 1$ (the empty suffix has exactly one partition).

For each $\ell = n, n - 1, \ldots, 1$ we consider all possible first cuts at $i \in [\ell, n]$:

$$F[\ell] = \sum_{i=\ell}^{n} \mathbf{1}\big[\textsc{IsWord}(A[\ell \ldots i])\big] \cdot F[i + 1].$$

Return $F[1]$ as the solution.

Running time on # of IsWord calls: There are n states and each state scans at most n endpoints i, so the total number of distinct substrings $A[\ell \ldots i]$ queried is $\sum_{\ell=1}^{n}(n - \ell + 1) = \Theta(n^2)$. Each such substring triggers at most one call to IsWord. Hence the algorithm runs in $O(n^2)$ time aside from IsWord, and makes $O(n^2)$ calls to IsWord.

(b) and (c): can also be improved to $O(n^2)$ time complexity following a similar approach.

7. **Distances between polygonal curves** A polygonal curve of n vertices $x_1, x_2, \cdots, x_n$ is the concatenation of $n-1$ line segments $x_i x_{i+1}$, for $i = 1, 2, \cdots, n-1$. Given two polygonal curves X, Y, each with n vertices $\{x_i\}$, $\{y_i\}$ respectively, a natural question is to define the distance between them.

The Frechét distance is defined in the following way. Imagine that one person walks on the curve X and a dog walks on the curve Y. The dog is tied with a leash (or rope). Both the person and the dog can not walk backward. In addition, to simplify the problem, the person and the dog can only walk one at a time. When the person (or the dog) walks, the dog (or the person) needs to wait at a vertex. The Frechét distance is the minimum length of the leash for the person and the dog to possibly finish the walk.

Give an $O(n^2)$ algorithm to compute the discrete Frechét distance of the two curves.

Answer: First we observe that if the person stops at a vertex and the dog walks on a line segment $y_i y_{i+1}$, then the maximum leash length is realized at the endpoints y_i or y_{i+1}, by simple geometry.

We use dynamic programming on this problem. The induction step is as follows. Denote by $D[i,j]$ as the minimum leash length for the person to walk to x_i and the dog to walk to y_j. Now,

$$\begin{aligned} D[i,j] &= \min\{\max\{D[i-1,j], d(x_i,y_j)\}, \max\{D[i,j-1], d(x_i,y_j)\}\} \\ &= \max\{\min\{D[i,j-1], D[i-1,j]\}, d(x_i,y_j)\} \end{aligned}$$

In addition, we have $D[1,1] = d(x_1,y_1)$, $D[i,0] = D[0,j] = \infty$, for all $0 \le i,j \le n$. By using standard dynamic programming techniques we can solve this problem in $O(n^2)$ time.