Problem 1 (10 points).

In the lecture, we learned a $O(mn)$ dynamic programming (DP) algorithm to compute edit distance of two strings, aka sequence alignment. Similar algorithms are widely used in bioinformatics where scientists often compare similarity of two given DNA sequences.

Here you are required to design a DP algorithm to compute the edit penalty of the optimal alignment of two give strings, but with a higher gap penalty. The biological implication is that mutations (*i.e.* mismatches) happens much more frequently than insertions/deletions (*i.e.* gaps) in a DNA sequence. Also, extending a gap (*i.e.* a longer gap) is more favored than opening a new gap (*i.e.* two short gaps). More specifically, in the edit distance problem, the penalty (i.e., the edit distance) is incremented by 1 for a mismatch or a gap between S_1 and S_2. In this problem with refined gap model, the penalty is still incremented by 1 for a mismatch, but by a when opening a new gap, and by b when extending an existing gap ($a > b \geq 3$). For example, the total penalty for alignment `AA__CC` $\times$ `AAGGCT` is $(a+b+1)$ where a is for opening the gap at position 3, b for extending the gap at position 4, and 1 for a mismatch at position 6.

Problem: Sequence alignment with gap penalty.

Input: Two strings $S_1[1\cdots n]$ and $S_2[1\cdots m]$ over alphabet $\Sigma = \{A,C,G,U\}$, a, b, $a > b \geq 3$.

Output: the alignment between S_1 and S_2 with minimized total penalty.

Design a dynamic programming algorithm for above problem. Show correctness and complexity analysis of your algorithm. Your algorithm should run in $O(mn)$ time.

Hint: Use three DP tables. One for ending with a gap in S_1, one for ending with a gap in S_2, one for match/mismatch only.

Solution: Define $F(i,j)$ as the edit distance between $S_1[1\cdots i]$ and $S_2[1\cdots j]$ ending with a match or mismatch at $S_1[i]$ and $S_2[j]$.
Define $X(i,j)$ as the edit distance between $S_1[1\cdots i]$ and $S_2[1\cdots j]$ ending with a gap at S_1.
Define $Y(i,j)$ as the edit distance between $S_1[1\cdots i]$ and $S_2[1\cdots j]$ ending with a gap at S_2.

Define mismatch penalty function $\delta(x,y) = \begin{cases} 0, x = y \\ 1, x \neq y \end{cases}$.

Hence, we have the following recurrence:

$$F[i,j] = \delta(S_1[i], S_2[j]) + min \begin{cases} F[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases} \tag{1}$$

$$X[i,j] = min \begin{cases} F[i,j-1]+a & \text{// opening a gap} \\ X[i,j-1]+b & \text{// extending a gap} \\ Y[i,j-1]+a & \text{// actually won't use} \end{cases} \tag{2}$$

$$Y[i,j] = min \begin{cases} F[i-1,j]+a & \text{//opening a gap} \\ X[i-1,j]+a & \text{//actually won't use} \\ Y[i-1,j]+b & \text{//extending a gap} \end{cases} \tag{3}$$

Use P_f, P_x, P_y to store the traceback directions to get F, X, Y respectively.

function Alignment-affine-penalty $(S_1[1\cdots n], S_2[1\cdots m])$

initiate array F, X, Y, P_f, P_x, P_y of size $(n+1)\times(m+1)$

$F(0,0)=0$, $X(0,0)=0$, $Y(0,0)=0$

$F(i,0)=-\infty$, $X(i,0)=-\infty$, $Y(i,0)=a+(i-1)\times b$, $1\le i\le n$

$F(0,j)=-\infty$, $X(0,j)=a+(j-1)\times b$, $Y(0,j)=-\infty$, $1\le j\le m$

$P_f[i,0]=-1$, $P_x[i,0]=-1$, $P_y[i,0]=-1$, $0\le i\le n$

$P_f[0,j]=-1$, $P_x[0,j]=-1$, $P_y[0,j]=-1$, $0\le j\le m$

for $i=1$ to n

for $j=1$ to m

$$k = min \begin{cases} F[i-1,j-1] \\ X[i-1,j-1] \\ Y[i-1,j-1] \end{cases}$$

$$F[i,j] = \delta(S_1[i], S_2[j]) + k$$

$$X[i,j] = min \begin{cases} F[i,j-1]+a \\ X[i,j-1]+b \\ Y[i,j-1]+a \end{cases}$$

$$Y[i,j] = min \begin{cases} F[i-1,j]+a \\ X[i-1,j]+a \\ Y[i-1,j]+b \end{cases}$$

$$P_f[i,j] = \begin{cases} 1, & \text{if } F[i-1,j-1]=k \\ 2, & \text{if } X[i-1,j-1]=k \\ 3, & \text{if } Y[i-1,j-1]=k \end{cases}$$

$$P_x[i,j] = \begin{cases} 1, & \text{if } X[i,j]=F[i,j-1]+a \\ 2, & \text{if } X[i,j]=X[i,j-1]+b \\ 3, & \text{if } X[i,j]=Y[i,j-1]+a \end{cases}$$

$$P_y[i,j] = \begin{cases} 1, & \text{if } Y[i,j]=F[i-1,j]+a \\ 2, & \text{if } Y[i,j]=X[i-1,j]+a \\ 3, & \text{if } Y[i,j]=Y[i-1,j]+b \end{cases}$$

End for

End for

// trace back using P

$max\_score = max(F[n,m], X[n,m], Y[n,m])$

$$P' = \begin{cases} P_f, \text{ if } max\_score = F[n,m] \\ P_x, \text{ if } max\_score = X[n,m] \\ P_y, \text{ if } max\_score = Y[n,m] \end{cases}$$

$p_0 = P'[n,m]$

$i = n,\ j = m$

// make alignment from end to start

while $p_0 != -1$

  if $P' = P_f$: consume $S_1[i]$ and $S_2[j]$ for alignment, $i--$, $j--$

  if $P' = P_x$: consume a gap in S_1 and $S_2[j]$ for alignment, $j--$

  if $P' = P_y$: consume $S_1[i]$ and a gap in S_2 for alignment, $i--$

  $$P' = \begin{cases} P_f, \text{ if } p_o = 1 \\ P_x, \text{ if } p_o = 2 \\ P_y, \text{ if } p_o = 3 \end{cases}$$

  $p_0 = P'[i,j]$

End while

if $i \neq 0$ or $j \neq 0$: consume any remaining chars of S_1 or S_2 to align with gaps.

End **function**

Correctness: To fill in F, X ,and Y, they should be build upon the minimum of previous penalty plus the new penalty. For F the new penalty is $\delta(S_1[i], S_2[j])$. For X, the penalty is a for using optimal cases from F and Y, and the penalty is b for using optimal cases from X. Since the gap penalty is affine penalty (a linear function $a + (x-1) \times b$), we can safely use $F[i-1,j]$ and $X[i-1,j]$ for computing $X[i,j]$ without considering the length of the gap. (If the penalty function is a general one, for example log function, we have to consider the length of the gap and consequently the complexity becomes $O(n^3)$). The traceback step works similar to that when using a linear penalty, but we need to switch between three DP tables. Then the alignment can be constructed from end to the start.

Kudos: Why you actually will never compute $X[i,j]$ directly from $Y[,]$ or *vice versa*? Recall that it is an affine penalty where opening a gap costs a and extending a gap costs b ($a > b \geq 3$). Suppose you compute $X[i,j]$ from $Y[i,j-1]$. In other words, the last position of the alignment is $S_2[j]$ and a gap in S_1 while the 2nd last position is $S_1[i]$ and a gap in S_2 (recall it is computed from Y meaning you end a gap in S_2). For example, `XXXXAG_` × `XXXXA_C`. You can always eliminate the two adjacent gaps in opposite strings by making them a mismatch, because its penalty is always less than two gaps, *i.e.* $1 < a+b$ or $1 < a+a$. For example, `XXXXAG_` × `XXXXA_C` ⇒ `XXXXAG`× `XXXXAC`. So in practice, you will not have an alignment which has two adjacent gaps in opposite strings.

Complexity: All tables F, X, Y, P_f, P_x, P_y are of size $(n+1) \times (m+1)$, and filling them takes constant time, so filling all the tables take $O(mn)$ time.
In the traceback procedure, at least one of i or j is decremented by 1, so in total the *while* loop can have at most (m+n+1) iterations. Each iteration takes constant time.
Hence the total time complexity is $O(mn)$.

Problem 2 (10 points).

Given two strings $S_1[1\cdots n]$ and $S_2[1\cdots m]$, design an $O(nm)$ algorithm to decide the optimal alignment between $S_1[1\cdots n]$ and $S_2[1\cdots m]$ is unique.

Solution: Define $F(i,j)$ as the edit distance between $S_1[1\cdots i]$ and $S_2[1\cdots j]$.
Define $H(i,j)$ as the tracing back set of how to compute $F(i,j)$.
Define $\delta(x,y) = \begin{cases} 0, x = y \\ 1, x \neq y \end{cases}$.

```
function Alignment-and-traceback (S1[1···n], S2[1···m])
    initiate array F of size (n+1) × (m+1)
    initiate array H of size (n+1) × (m+1)
    F(i,0) = i, 0 ≤ i ≤ n
    F(0,j) = j, 0 ≤ j ≤ m
    H(i,j) = {}, 0 ≤ i ≤ n and 0 ≤ j ≤ m
    // Main body: compute edit distance F and traceback set H
    for i = 1 to n
        for j = 1 to m
            a = F(i−1, j−1) + δ(S1[i], S2[j])
            b = F(i−1, j) + 1
            c = F(i, j−1) + 1
            F(i,j) = min(a,b,c)
            if F(i,j) == a:  H(i,j).add(H(i−1, j−1))
            if F(i,j) == b:  H(i,j).add(H(i−1, j))
            if F(i,j) == c:  H(i,j).add(H(i, j−1))
        End for
    End for
    // traceback
    p = H(n,m)
    while(p.size() ≠ 0)
        if p.size() > 1:
            return False
        else
            p = p.pop()
        End if
    End while
    return True
End function
```

Correctness: The main body this algorithm (lines before traceback) resembles the edit distance algorithm we introduced in the lecture. We can compute $F(i,j)$ correctly, while $H(i,j)$ is a set storing how the optimal edit distance $F(i,j)$ is achieved – *e.g.* match/mismatch, gap in S_1, or gap in S_2. If $H(i,j).size() > 1$, it means

we can achieve the optimal edit distance $F(i,j)$ from at least two subproblems from

$$\begin{cases} F(i-1,j-1)+\delta(S_1[i],S_2[j]) \\ F(i-1,j)+1 \\ F(i,j-1)+1 \end{cases}$$

i.e. multiple optimal alignments of $S_1[1\cdots i]$ and $S_2[1\cdots j]$.

The traceback part examines from the end to the start of the optimal alignment, whether such $H(i,j)$ is present. If all $H(i,j)$ in the optimal alignment has size 1, there is only one optimal alignment. Otherwise there are multiple optimal alignments. When we reach an $H(i,j)$ of size 0, we terminate the traceback because we exhausted at least one of S_1 or S_2, and then we can only align the rest of the other string with gaps.

Complexity: Initializing F and H takes $O(mn)$ time. Main body of the algorithm runs in $O(mn)$. Since for i from 1 to n and j from 1 to n, each $F(i,j)$ and $H(i,j)$ is only computed once and computing each $F(i,j)$ and $H(i,j)$ takes constant time.

We terminate traceback *while* loop if size of $H(i,j)$ is not 1, so in each iteration $H(i,j)=\begin{cases} H(i-1,j-1) \\ H(i-1,j) \\ H(i,j-1) \end{cases}$, index of $H(i,j)$ must decrement by at least 1, so we have at most $m+n$ iterations. The *if* statement inside the loop takes constant time. Thus, traceback takes $O(n+m)$ time.

The total time complexity is $O(mn)+O(m+n)=O(mn)$

Problem 3 (10 points).

Recall from the class we talked about the sequence alignment problem where we use dynamic programming to calculate the edit distance between two strings. Given two strings A and B have the same length of n characters, and a threshold k, $0<k<n$, design an algorithm to check whether the edit distance between A and B is less than k. Your algorithm should run in $O(nk)$ time. See Figure 1.

Solution: Instead of calculating the value for *all* the cells in the $n\times n$ dynamic programming matrix M, here, we only need to calculate the value in the diagonal with the width of $2k-1$ (Figure 1) using exactly the same recursion we introduced during lecture. We can prove that, $M[n,n]<k$ if and only if the edit distance between A and B is less than k.

Problem 4 (10 points).

We introduced longest increasing subsequence (LIS) problem in the lecture, and defined an DP algorithm runs in $O(n^2)$ time. Now we want to further improve it: design an algorithm runs in $O(n\log n)$ time to solve this problem but only finds the length of the longest increasing subsequence of the given array $S[1\cdots n]$, i.e., this algorithm don't need to find the actual longest increasing subsequence.

Here are some hints: we introduce a new array D, where $D[i]$ represents the minimum value of the last element in the increasing subsequence of length i. For example, if we have $S=[2,1,4,5]$, then we'll have $D[1]=1$, $D[2]=4$ and $D[3]=5$. (There are four increasing subsequences of length 1, and 1 is the smallest; among the four increasing subsequences of length 2, i.e., $[2,4]$, $[2,5]$, $[1,4]$ and $[1,5]$, 4 is the minimum last element; and both the two increasing subsequences of length 3 end with 5.) In your algorithm, you can

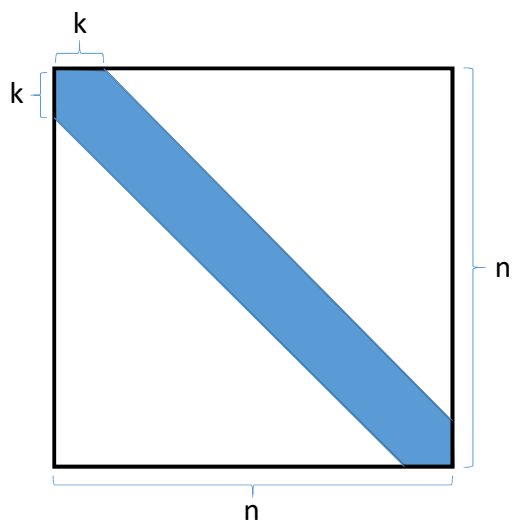


Figure 1: Diagonal area required to fill in the dynamic programming matrix when only edit distance of less than k is considered.

enumerate the index i from 1 to n, and use the value of $S[i]$ to update the array D. Meanwhile, we also need to keep updating the length of LIS. *Further hint*: binary search might be needed.

Solution: Similar to the $O(n^2)$ algorithm for LIS problem introduced in class, here we also need to enumerate from the array S from 1 to n. But now we will not calculate $L[i]$, instead we will keep updating the values in array D. According to the definition of array D, it's easy to find that $D[1] \leq D[2] \leq D[3] \cdots \leq D[longest]$. Because if there exists i, j that $i < j$ and $D[i] > D[j]$, we can easily find an increasing sequence of length i ended with $D[j]$, which is contradictory to the definition of D.

When enumerating the index i of array S, it's also needed to store the length of LIS so far in $S[1 \cdots i]$. Let $maxL$ be the length of LIS so far, which is also the answer of the problem when we finish the enumeration. When it comes to index i, there are two situations. The first one is that $S[i] > D[maxL]$, apparently we can find a new increasing sequence of length $maxL+1$ ended with $S[i]$. Thus here we let $maxL = maxL+1$, and $D[maxL] = S[i]$. The other one is that there exists j that $S[i] < D[j]$ and $S[i] \geq D[j-1]$, then we can update $D[j]$ by letting $D[j] = S[i]$. We can use binary search to find the j which would take $O(\log n)$ each time. After enumerating all the index of S, $maxL$ would be the length of LIS. And the overall running time would be $O(n \log n)$.

Problem 5 (10 points).

Let $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$ be two strings. Design a dynamic programming algorithm to compute the *longest common subsequence* between A and B, i.e., to find the *largest* k and indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and $1 \leq j_1 < j_2 < \cdots < j_k \leq m$ such that $A[i_1] = B[j_1], A[i_2] = B[j_2], \cdots, A[i_k] = B[j_k]$. Your algorithm should run in $O(mn)$ time.

Solution. Define $LCS[i, j]$ as the length of the *longest common subsequences* between $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$. We have the *recursion* formula below:

$$LCS[i,j] = \begin{cases} LCS[i-1, j-1] + 1 & \text{if} \quad a_i = b_j \\ max\{LCS[i-1, j], LCS[i, j-1]\} & \text{if} \quad a_i \neq b_j \end{cases}$$

The *pseudo-code*, which includes three steps, is as follows.

Initialization:

$LCS[i,0] = 0$, for all i in $1,2,\ldots n$
$LCS[0,j] = 0$, for all j in $1,2,\ldots m$

Iteration:

for $i = 1 \cdots n$:
  for $j = 1 \cdots m$:
    if $a_i = b_j$: $LCS[i,j] = LCS[i-1,j-1]+1$
    else: $LCS[i,j] = max\{LCS[i-1,j], LCS[i,j-1]\}$
  endfor
endfor

Termination: $LCS[n,m]$ gives the length of the *longest common subsequences* between A and B. (The corresponding indices can be fetched through introducing backtracing pointers.)

Running time: The initialization of $LCS[i,j]$ takes $O(m+n)$ time. The iteration step have two embedded *for* loops, and therefore takes $O(mn)$ time. The total running time of the algorithm is $O(mn)$.

Problem 6 (10 points).

Given a string S of length n and a positive integer k, find the longest continuous repeat of a substring which length is k. For example, when $S = ACGACGCGCGACG$, if $k = 2$, the longest continuous repeat would be $CGCGCG$ composed of CG repeating 3 times. And if $k = 3$, the longest continuous repeat would be $ACGACG$ composed of ACG repeating twice. Design a dynamic programming algorithm to solve this problem and analyze its running time.

Solution. Define $f[i]$ as the length of repeating character at position i with distance k. For example, $f[i] = 3$ represents $f[i] = f[i-k] = f[i-2k] \neq f[i-3k]$.

We have the *recursion* formula below:

$$f[i] = \begin{cases} f[i-k]+1 & \text{if} \quad S[i] = S[i-k] \\ 1 & \text{if} \quad S[i] \neq S[i-k] \end{cases}$$

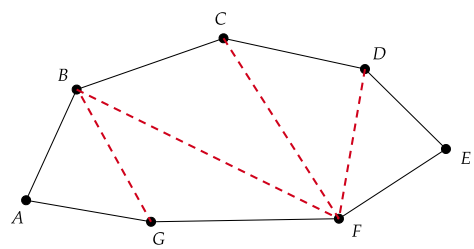
After computing $f[i]$ for every position in the array, we can use a sliding window to get the answer. Since for any substring $S[i, i+1, \cdots i+k-1]$ of length k, the repeating time of it is $min(f[i], f[i+1], f[i+2] \cdots, f[i+k-1])$. Therefore we only need to use a sliding window of length k moving from beginning of the string to the end. Every time we find the minimum value of $f[i]$ for all the indices inside the sliding window. And the maximum value of all the values we get from each sliding window will be our answer. The running time is $O(nk)$.

Problem 7 (10 points).

You have a wood plank in a shape of a convex polygon with n edges, represented as the (circular) list of the 2D coordinates of its n vertices. You want to cut it into $(n-2)$ triangles using $(n-3)$ non-crossing diagonals, such that the total length of the cutting trace (i.e., total length of the picked diagonals) is minimized. Design a dynamic programming algorithm runs in $O(n^3)$ time to solve this problem. Your solution should include

definition of subproblems, a recursion, how to calculate the minimized total length (i.e., the termination step), and an explanation of why your algorithm runs in $O(n^3)$ time.

For example, in the convex polygon given below, one possible way to cut it into triangles is illustrated with red dashed lines, and in this case the total length of cutting trace is $\overline{BG}+\overline{BF}+\overline{CF}+\overline{DF}$.



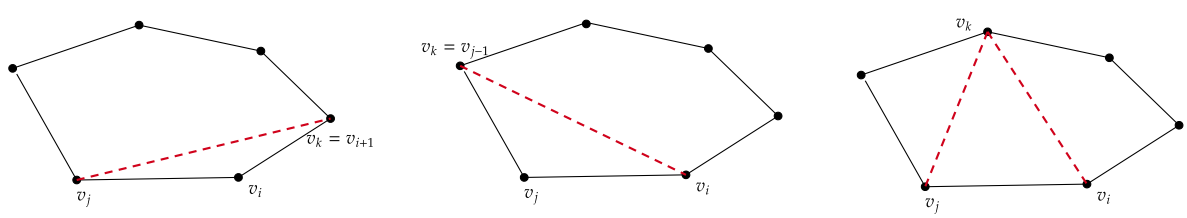
Solution. Note: this problem requires defining subproblems over intervals.

1. Subproblem: consider the convex polygon from vertex v_i to vertex v_j following couter-closewise order closed by a (possibly virtual) edge (v_j, v_i); define $M[i, j]$ as the minimized total cutting trace of this polygon. Notice that the subproblems we defined span *all* intervals of the given polygon in a circular manner; it could be that $i > j$.

2. Recursion. Consider the last edge (v_j, v_i) of above defined polygon: this edge must be in a triangle, and we enumerate the other vertex of this triange. Assume it's v_k. If v_k is next to v_i, i.e., $k = i+1$ (in circular manner, i.e., mod n), then we only need to cut $\overline{v_k v_j}$, and this leaves a smaller polygon from v_k to v_j. If v_k is next to v_j, i.e., $k = j-1$ (in circular manner, i.e., mod n), then we only need to cut $\overline{v_k v_i}$, and this leaves a smaller polygon from v_i to v_k. If v_k is not next to either one i.e., $i+1 < k < j-1$ (in circular manner, i.e., mod n), then we need to cut both $\overline{v_k v_i}$ and $\overline{v_k v_j}$, and this leaves two smaller polygons, one from v_i to v_k and one from v_k to v_j. See figure below. Formally,

$$M[i,j] = \min \begin{cases} \overline{v_{i+1} v_j} + M[i+1, j] \\ \overline{v_i v_{j-1}} + M[i, j-1] \\ \min_{i+1<k<j-1} (\overline{v_i v_k} + \overline{v_k v_j} + M[i,k] + M[k,j]) \end{cases}$$

   Note that we define $M[i, j] = 0$ for any $i+2 \leq j$ (mod n), i.e., any polygon with at most 3 vertices is free of cutting.



3. Termination: $M[1, n-1]$ answers the original question.

4. Time complexity: we define n^2 subproblems, and solving each takes $\Theta(n)$ time; hence the time complexity is $O(n^3)$.

Problem 8 (30 points).

Let $c_1, c_2, \cdots, c_n$ be various currencies. For any two currencies c_i and c_j, there is an exchange rate $r_{i,j}$; this means that you can purchase $r_{i,j}$ units of currency c_j in exchange for one unit of c_i. These exchange rates satisfy the condition that $r_{i,j} \cdot r_{j,i} < 1$, so that if you start with a unit of currency c_i, change it into currency c_j and then convert back to currency c_i, you end up with less than one unit of currency c_i (the difference is the cost of the transaction). Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$, and two currencies s and t, find the most advantageous sequence of currency exchanges for converting currency s into currency t. (Assume that, there does not exists sequence of currencies $c_1, c_2, \cdots, c_k$ such that $r_{1,2} \cdot r_{2,3} \cdot r_{3,4} \cdots r_{k-1,k} \cdot r_{k,1} > 1$.)

Solution. We can transform the currency exchange problem into the shortest path problem. We build a directed graph $G = (V, E)$, where $V = \{c_1, c_2, \cdots, c_n\}$ represents all currencies, and E contains all pairs (c_i, c_j), $1 \le i \ne j \le n$. We assign length for edge $(c_i, c_j) \in E$ as $-\log(r_{i,j})$. We now show that computing the shortest path from s to t in G actually gives the optimal sequence of currency exchanges for converting s into t. In fact, there is one-to-one correspondence between a path from s to t in G and a sequence of currency exchange for converting s into t. Moreover, the length of such a path p equals to $\sum_{(c_i,c_j)\in p} -\log r_{i,j} = -\log \prod_{(c_i,c_j)\in p} r_{i,j}$. Hence, the shortest path in G gives the path maximizes $\prod_{(c_i,c_j)\in p} r_{i,j}$, which is exactly the maximized amount of currency t that can be coverted from a unit of currency s. Based on the above analysis, the algorithm will be to compute the shorest path from s to t in G (using any of the single-source shortest path algorithm we introduced). The vertices along this optimal path gives the sequence of currencies following which we can get the maximized amount of currency t.

Occasionally the exchange rates satisfy the following property: there is a sequence of currencies $c_1, c_2, \cdots, c_k$ such that $r_{1,2} \cdot r_{2,3} \cdot r_{3,4} \cdots r_{k-1,k} \cdot r_{k,1} > 1$. This means that by starting with a unit of currency c_1 and then successively converting it to currencies c_2, c_3, $\cdots$, c_k, and finally back to c_1, you would end up with more than one unit of currency c_1. Give an efficient algorithm for detecting the presence of such an anomaly.

Solution. We need to detect whether there is $r_{1,2} \cdot r_{2,3} \cdot r_{3,4} \cdots r_{k-1,k} \cdot r_{k,1} > 1$, which is equivalent to detect $-(\log(r_{1,2}) + \log(r_{2,3}) + \log(r_{3,4}) + \cdots + \log(r_{k-1,k}) + \log(r_{k,1})) < 0$. Since we assign length for edge $e = (c_i, c_j)$ as $-\log(r_{i,j})$, this exactly implies a negative cycle in G. In other words, such an anomaly exists if and only if G contains negative cycles. Therefore, we can use Bellman-Ford algorithm on G to identify whether G contains negative cycles, which also answers whether there exists anamaly.

Problem 9 (15 points).

There are n trains $(X_1, X_2, \cdots, X_n)$ moving in the same direction on parallel tracks. Train X_k moves at constant speed v_k, and at time $t = 0$, is at position s_k. Train X_k will be awarded, if there exists a time period $[t_1, t_2]$ of length at least δ (i.e., $0 \le t_1 < t_2$ and $t_2 - t_1 \ge \delta$) such that during this entire time peroid X_k is in front of all other trains (it is fine if X_k is behind some other train prior to t_1 or X_k is surpassed by some other train after t_2). Given v_k and s_k, $1 \le k \le n$, and $\delta > 0$, design an algorithm to list all trains that will be awarded. Your algorithm should run in $O(n \cdot \log n)$ time.

Solution. Let $y_k(t)$ be the position of X_k at time t. Clearly $y_k(t) = s_k + v_k \cdot t$. We first find those trains that are ahead of other trains at some time point (for now not necessarily spanning δ). This can be solved by transforming into the half-plane-intersection problem. Note that train X_k is in front of other trains at time t if and only if $y_k(t) > y_i(t)$ for any $i \ne k$. Therefore, we compute the *upper envelop* of these n

lines: $y_k(t) = s_k + v_k \cdot t$, $1 \leq k \leq n$, which is equivalent to finding the intersection of these n half planes: $y_k(t) \geq s_k + v_k \cdot t$, $1 \leq k \leq n$. Clearly, trains corresponding to those lines consisting of the upper envelop (i.e., on the boundary of the intersection) are trains that can be possibly awarded, as each such train is ahead of other trains at some time point. Trains do not correspond to any line on the boundary cannot be awarded.

We can use the divide-and-conquer algorithm (introduced in class) to find the intersection of above n half-planes. Recall that this algorithm returns two sorted lists of lines, representing the left boundary and right boundary of the intersecting region respectively. In fact in our case, the left boundary must be empty, as all lines have positive slope and all half-planes are upper part of the 2D space (i.e., in the form of $y \geq ax + b$). We now process the sorted list of lines in the right boundary to determine whether each train will be actually awarded. For each line L (corresponding to a train) we consider its left neighbor L_1 and right neighbor L_2 in the sorted list, and compute the interesecting point (t_1, y_1) between L and L_1 and the intersecting point (t_2, y_2) between L and L_2. (If L is the leftmost line in the sorted list, set $t_1 = 0$; if L is the rightmost line set $t_2 = \infty$.) If we have $0 \leq t_1 < t_2$ and $t_2 - t_1 \geq \delta$, then the train (corresponding to line L) will be awarded. Examine all lines in the sorted list in this way gives us the set of trains that will be awarded.

The running time of finding the upper envelop takes $O(n \cdot \log n)$ time and the postprocessing of the sorted list takes linear time. Therefore, the entire algorithm runs in $O(n \cdot \log n)$ time.