Optimal Online and Offline Algorithms for Finding Longest and Shortest Subsequences with Length and Sum Constraints

1. Introduction

In a DNA sequence (a string of A, C, G, and T) it is often required to find CG-rich subsequences, in which C and G appear more than, say 55%, to locate CpG islands [5]. Since the ratio of C and G is computed by dividing the sum of the numbers of C and G by the length of the sequence, it is necessary to find subsequences whose lengths are between two bounds and whose sums satisfy a certain range condition.

Let \(A[1 \ldots n]\) be a sequence of \(n\) real numbers. For \(1 \leq i \leq j \leq n\), \(A[i \ldots j]\) is called a subsequence of \(A\). For \(A[i \ldots j]\), its sum is \(A[i] + \cdots + A[j]\) and its length is \(j - i + 1\). For two positive integers \(I, J\) with \(I \leq J\) and two real numbers \(X, Y\) with \(X \leq Y\), a subsequence is feasible if its length is between \(I\) and \(J\) and its sum is between \(X\) and \(Y\).

Given the sequence \(A\), and four parameters \(I, J, X\) and \(Y\), we are interested in the problems of locating the longest feasible subsequence and the shortest feasible subsequence in \(A\). In the offline version of the problem the elements of the sequence \(A\) are known before the start of the algorithms, while in the online version the elements of \(A\) arrive one by one from \(A[1]\) and we know only \(A[1 \ldots i]\) after \(A[i]\) arrives; and for each \(1 \leq i \leq n\) after \(A[i]\) arriving the algorithms are to solve the subproblem on \(A[1 \ldots i]\). Offline versions are appropriate for cases where data necessary are all available to be processed, while online versions are good for streaming data which are generated one by one or for applications which require responses in real time.

Chen and Chao [2] addressed the problems: Given a sequence \(A\) and a parameter \(X\), locate the longest and shortest subsequences whose sum is at least \(X\). They presented linear time algorithms for both of the problems, and also raised a question if linear time algorithms are possible for the problems: Given a sequence \(A\) and two parameters \(X \leq Y\), locate the longest and shortest subsequences whose sums are between \(X\) and \(Y\).

Hsieh, Yu and Wang [4] answered the question negatively by showing that the problems have an \(\Omega(n \log n)\) time lower bound. They also presented optimal online algorithms for the following problems: Given a sequence \(A\) and four parameters \(I, J, X\) and \(Y\), locate the longest and shortest subsequences such that their lengths are between \(I\) and \(J\), and their averages are between \(X\) and \(Y\), where the average of \(A[i \ldots j]\) is its sum divided by its length, i.e., \(\frac{A[i] + \cdots + A[j]}{j - i + 1}\).

In this paper, we are dealing with the problem of finding the longest feasible subsequence and the problem of finding the shortest feasible subsequence. For each problem, we are presenting two algorithms, an online and an offline algorithm. In Section 2, online algorithms for finding the longest and shortest feasible subsequences are given, which run in \(O(n \log n)\) time. In Section 3, offline algorithms for finding the longest and shortest feasible subsequences are given. The algorithms also run in \(O(n \log n)\) time, and, except one sorting of \(n\) numbers, they run in \(O(n \alpha(n))\) time, which is considered linear for all practical purposes, where \(\alpha(n)\) is the inverse of the Ackermann function [3, 6].

2. The Online Algorithms

In this section, we present online algorithms for the problem of finding the longest and shortest feasible subsequences. In Section 2.1, we transform the problem of finding the longest feasible subsequence into a geometric one, and shows the geometric problem can be solved in \(O(n \log n)\) time. In Section 2.2, the problem of finding the shortest feasible subsequence is considered.
2.1 Finding the Longest Feasible Subsequence

Our algorithm for finding the longest feasible subsequence will compute \( \lambda_j \) for all \( 1 \leq j \leq n \), where \( \lambda_j \) is the smallest integer, if exists, such that \( A[\lambda_j + 1 \ldots j] \) is feasible. In other words, \( A[\lambda_j + 1 \ldots j] \) is the longest feasible subsequence when the right boundaries of subsequences are fixed at \( j \). If no such integer exists, then \( \lambda_j = \infty \). If we have \( \lambda_j \) for all \( 1 \leq j \leq n \), then the longest feasible subsequence can be located by computing \( k \) such that \( k - \lambda_k = \max\{j - \lambda_j \mid 1 \leq j \leq n\} \). \( A[\lambda_k + 1 \ldots k] \) is the longest feasible subsequence. If \( k - \lambda_k = -\infty \), then there is no feasible subsequence and NIL is output.

We transform the problem of finding the longest feasible subsequence of \( A \) into a geometric problem. Define the cumulative sums, \( c_0 = 0 \) and \( c_i = c_{i-1} + A[i] \) for \( 1 \leq i \leq n \). For each \( 0 \leq i \leq n \), define a vertical segment \( l_i = (x_i, y^h_i, y^l_i) = (i, c_i + X, c_i + Y) \) in the plane. A vertical segment \( l \) with its bottom point \((x, y^l)\) and its top point \((x, y^h)\) is denoted by \( l = (x, y^h, y^l)\).

Let \( Q = \{l_i \mid 0 \leq i \leq n\} \). Let \( Q_j = \{l_i \mid \max\{0, j-I\} \leq i \leq j \leq n\} \). Define a point \( p_j = (-\infty, c_j) \) for \( I \leq j \leq n \). From \( p_j \), we draw a horizontal rightward ray toward \( Q_j \). Let \( l_k \), if exists, be the first segment in \( Q_j \) that is hit by the ray. Then, \( \lambda_j = k \). If the ray does not hit any segment in \( Q_j \), then \( \lambda_j = \infty \). See Fig. 1.

Consider \( i < j \). Then, \( c_j - c_i \) is the sum of \( A[i + 1 \ldots j] \). For \( A[i + 1 \ldots j] \) to be feasible, it has to satisfy the sum constraint \( X \leq c_j - c_i \leq Y \), or equivalently \( c_i + X \leq c_j \leq c_i + Y \); and the length constraint \( I \leq j - i \leq J \), or equivalently \( j - J \leq i \leq j - I \). The length constraint requires that \( l_{\lambda_j} \) should be in \( Q_j \), and the sum constraint requires that \( l_{\lambda_j} \) should be hit by the ray. The objective of finding the longest one requires that \( l_{\lambda_j} \) should be the first one hit by the ray. Refer to Fig. 1.

Two points are horizontally visible if the two points have the same \( x \)-coordinate and the horizontal segment connecting them does not intersect any other in-between object. Let \( l_\infty \) (resp., \( l_\infty \)) be the vertical line whose \( x \)-coordinate is \(-\infty \) (resp., \( \infty \)). For a set of vertical segments of the same length, the left envelope (resp., right envelope) of the set consists of the portions of the segments that are horizontally visible from \( l_\infty \) (resp., \( l_\infty \)). See Fig. 2. As seen in the figure, an envelope consists of fragments, each of which is a segment or a part of a segment in the set. Walking upwards an envelope starting at its bottom point, we may meet several discontinuities, where the envelope jumps from a fragment to another, and arrive at its top point. An envelope may be defined by giving its sequence of its fragments.

We shall describe data structures that dynamically maintain the left and right envelopes of a set of vertical segments of the same length with the restrictions that only the leftmost segment can be deleted from the set and a segment can be inserted into the set only as its rightmost segment. Note that a red-black tree [3] is a binary search tree that can implement operations such as insert, delete, search, predecessor, and successor in \( O(\log n) \) time, where \( n \) is the number of keys in the red-black tree. For a red-black tree \( T \), \( \max(T) \) denotes the largest key in \( T \), and for each key \( y \in T \), \( \text{pred}(y) \) (resp., \( \text{succ}(y) \)) is the largest (resp., smallest) among the keys that are less (resp., greater) than \( y \).

Let \( T^L \) (resp., \( T^R \)) denote the data structure for
maintaining the left (resp., right) envelope. $T^L$ (resp., $T^R$) will be implemented with a red-black tree whose keys are the $y$-coordinates of the bottom point, and the top point, and the discontinuities of the left (resp., right) envelope.

For ease of explanation, we shall assume that the keys in $T^L$ and $T^R$ are all distinct, i.e., that $c_i + X$ and $c_i + Y$ for $0 \leq i \leq n$ are distinct. Allowing multiple keys with the same value does not increase time complexity of our algorithms, but makes implementations more complicated. For example, we may use the primary and secondary keys, where the $y$-coordinates are the primary key and the $x$-coordinates are the secondary key. If two primary keys are equal, then their secondary key are compared.

In $T^R$, every key $y$ except max($T^R$) is associated with an integer $FR(y)$, which denotes the fragment of the $y$-range (or simply range) $(y, succ(y))$. $FR(y)$ is the index of the segment from which the fragment comes. In other words, between $y$ and $succ(y)$, a part of $l_{FR(y)}$ is visible from $l_{\infty}$. For ease of explanation, we assume that keys $-\infty$ and $\infty$ are in $T^R$, and $FR(-\infty) = FR(\infty) = -\infty$ for $y'$ such that $succ(y') = \infty$. The tuple $(y, FR(y))$ will be used to denote both $y$ and $FR(y)$ at the same time, and $(y, *)$ will be used to denote a tuple whose $FR(y)$ is not necessary to be specified. Since $y$ is the key field, the tuples can be identified without the $FR(y)$ fields.

In $T^L$, every key $y$ except max($T^L$) is associated with an integer $FL(y)$, which denotes the fragment of the range $(y, succ(y))$. We assume that $-\infty$ and $\infty$ are in $T^L$, and $FL(-\infty) = FL(\infty) = \infty$ for $y'$ such that $succ(y') = \infty$. The notations $(y, FL(y))$ and $(y, *)$ will also be used.

Only one operation INSERT$_S(l_i)$ will be used on $T^R$, which inserts the segment $l_i$ (actually, $y_i^k$ and $y_i^l$) into $T^R$, while two operations DELETE$_S(l_i)$ and SEARCH$_L(c_j)$ will be used on $T^L$. DELETE$_S(l_i)$ deletes the segment $l_i$ (actually, $y_i^k$ and $y_i^l$) from $T^L$ and SEARCH$_L(c_j)$ locates $y$ such that $y < c_j < succ(y)$ with the help of $T^L$.

Our algorithm runs in an online manner. Initially, $maxlen = \infty$, $c_0 = 0$, both $T^L$ and $T^R$ have only two keys $-\infty$ and $\infty$. Whenever $A[j]$ for each $1 \leq j \leq n$ arrives, the algorithm in Fig. 3 is executed.

When $A[j]$ arrives, we first compute the cumulative sum $c_j$. For $1 \leq j \leq I$, we are done as $Q_j = \emptyset$. For $I < j \leq n$, we insert $l_{j-1}$ into $T^R$ as $Q_j = Q_{j-1} \cup \{l_{j-1}\}$, and search $c_j$ in $T^L$ to get $\lambda_j$. For $j > I$, we insert $l_{j-1}$ into $T^R$, delete $l_{j-1}$ from $T^L$ as $Q_j = Q_{j-1} \cup \{l_{j-1}\}$, and search $c_j$ in $T^L$ to get $\lambda_j$.

We shall give detailed descriptions of the three operations INSERT$_S$, DELETE$_S$, and SEARCH$_L$. Fig. 4 shows INSERT$_S(l_i)$. Before explaining the operation, the definition of $B_i$ will be given. $B_i$, $1 \leq i \leq n$, is the set of tuples $(y, FL'(y))$. The $y$'s are the $y$-coordinates of the top and bottom points of the segments $l_{i+1}, \ldots, l_{i+j-1}$ that are visible from $l_i$. $FL'(y)$ is similar to $FL(y)$ with the restriction that the visibility from $l_i$ instead of from $l_{\infty}$ is considered. $FL'(y)$ represents the fragment in $B_i$ of the range $(y, succ(y))$.

A queue will be employed to implement each $B_i$.

If $l_i$ is inserted, $l_i$ immediately becomes a fragment of the new right envelope as it is the rightmost segment, and fragments or parts of fragments of the current envelope that will be hid by $l_i$ must be deleted from the envelope. We first locate two ranges $(y_1, succ(y_1))$ and $(y_2, succ(y_2))$ in $T^R$ satisfying $y_i^k \in (y_1, succ(y_1))$ and $y_i^l \in (y_2, succ(y_2))$ in (1). Refer to Fig. 5.

In (2), we check if the bottom point of $l_i$ is visible from $l_{\infty}$ through the range $(y_1, succ(y_1))$. If so, $y_i^k$ with $FL(y_i^k) = i$ is inserted into $T^L$ in (3) to be a part of the left envelope. This is shown in Fig. 6(a). If not, $(y_i^k, i)$ is inserted into $B_{FR(y_1)}$ in (4) as the bottom point of $l_i$ is visible from $l_{FR(y_1)}$.

In (5), we check if the top point of $l_i$ is visible from $l_{\infty}$ through the range $(y_2, succ(y_2))$. If so, $y_i^l$ with $FL(y_i^l) = i$ is inserted into $T^L$ in (6) to be a part of the left envelope. In (7), we change $FL(y_2)$ from $\infty$ to $i$, depicted in Fig. 6(b). If not, $(y_i^l, i)$ is inserted into $B_{FR(y_2)}$ in (8) as the top point of $l_i$ is visible from $l_{FR(y_2)}$.
In (9)–(18), we delete these part of the right envelope that is hid by \( l_i \). Those \( (y, *) \in T^R \) satisfying \( y^t_y < y < y^t_{y'} \) are to be deleted. As shown in Fig. 5, the part of the right envelope that will be deleted forms two “stairways,” one of which starts from \( \text{succ}(y_1) \) and goes upward, and the other starts from \( y_2 \) and goes downward. This is obvious from the fact that the segments are all of the same length. In (9)–(11) and (13), we remove the “upward stairway” one “step” at a time.

In (12), we insert \( (y, i) \) into \( B_{FR(y)} \). For each \( y \) of the “upward stairway” the tuple \( (y, \infty) \) is already in \( B_{FR(y)} \) as the tuple was inserted in (8) when \( \text{INSERT}_S \) (segment with \( y^t_y = y \)) was called. At that time, \( L(y) = \infty \), but it has to change to \( i \) as \( l_i \) is now visible from \( l_{FR(y)} \) through the range \( (y, \text{succ}(y)) \). Instead of changing \( L(y) \), we insert the tuple \( (y, i) \) into \( B_{FR(y)} \) as this is easy to implement, which will be explained later when \( B_{FR(y)} \) is used in \( \text{DELETE}_S(l_i) \). \( B_{FR(y)} \) has both \( (y, \infty) \) and \( (y, i) \), which have the same key \( y \).

In (15)–(18), we remove the “downward stairway” one “step” at a time, starting from \( y_2 \).

Finally, in (19) two tuples \( (y^t_y, i) \) and \( (y^t_{y'}, f) \) are inserted into \( T^R \).

The operation \( \text{DELETE}_S(l_i) \) is shown at the top side of Fig. 8. We first find \( y_1 = y^t_y \) in (1) and delete both \( (y_1, *) \) and \( (\text{succ}(y_1), *) \) from \( T^L \) in (2). After \( l_i \) is removed, the left envelope has to be updated as the area that was hid by \( l_i \) is now visible from \( l_{-\infty} \). See Fig. 7. In (3)–(6), this update of the left envelope is performed by inserting the tuples in \( B_r \). Each \( B_r \) is a queue containing the tuples whose \( y \)-coordinates of the top and bottom points of the segments \( l_{i+1}, \ldots, l_{i+j-1} \) that are visible from \( l_i \). As mentioned in the description of (12) of \( \text{INSERT}_S(l_i) \), \( B_r \) may have two tuples with the same key \( y \), i.e., \( (y, \infty) \) and \( (y, k) \) for some \( k \). Since \( (y, \infty) \) was inserted into \( B_r \) before \( (y, k) \), \( (y, \infty) \) is considered in (3) before \( (y, k) \) as \( B_r \) is a queue. When \( (y, \infty) \) is considered, it is inserted into \( T^L \) in (6). When \( (y, k) \) is considered, since \( (y, \infty) \) is already in \( T^L \), \( (y, k) \) replaces it in (5).

The operation \( \text{SEARCH}_S(c_j) \), shown at the bottom side of Fig. 8, finds the range \( (y_1, \text{succ}(y_1)) \) containing \( c_j \) and returns \( \lambda_j = FL(y_1) \).

Time complexity will be analyzed. Since both \( T^R \) and \( T^L \) have at most \( J - I \) keys, operations on \( T^R \) and \( T^L \) such as search, insert, delete, predecessor, and successor can be executed in \( O(\log(J - I)) \) time per operation. Since each \( B_r \) is a queue, insertion and deletion takes constant time. During the execution of the algorithm on \( A[1], \ldots, A[n] \), each of the keys that are the \( y \)-coordinates of the top and bottom points of the segments in \( Q \) is inserted into \( T^R \) (resp., \( T^L \)) at most once and deleted from \( T^R \) (resp., \( T^L \)) at most once. Each of the keys is inserted into one of the queues \( B_r \) at most twice in \( \text{INSERT}_S(\cdot) \) and is considered at most twice in \( \text{DELETE}_S(\cdot) \). The algorithm runs in \( O(n \log(J - I)) = O(n \log n) \) time.

We have proved the following theorem:

**Theorem 1:** The longest feasible subsequence can be found by an online algorithm in \( O(n \log n) \) time.
In this section, we present offline algorithms for the problem of finding the longest and shortest subsequences. In Section 3.1, we present the nearest friend finding problem and provides a solution for it, which will be used in solving the longest and shortest subsequence problems. In Section 3.2, the problem of finding the longest feasible subsequence is addressed, and in Section 3.3, the problem of finding the shortest feasible subsequence is considered.

3.1 The Nearest Friend Finding Problem

The nearest friend finding problem is: Given a set of blue points, \( B \), and a set of red points, \( R \), in the plane, find, for each red point, its nearest blue friend (for short, friend) such that the \( x \)-coordinate (resp., \( y \)-coordinate) of the friend is less than the \( x \)-coordinate (resp., \( y \)-coordinate) of the red point and the \( x \)-distance between them is minimum. See Fig. 10. We shall show that the problem can be solved in \( O(n \log n) \) if both a \( x \)-sorted list and a \( y \)-sorted list of the points are given, where \( n = |B \cup R| \).

Let \( G = \{y_i = (x_i, y_i) \mid 1 \leq i \leq n\} \) be a list of points in \( B \cup R \) such that \( x_1 \leq \cdots \leq x_n \). Let \((g_{a_1}, \ldots, g_{a_n})\) be the \( y \)-sorted list of \( G \) such that \( y_{a_1} \leq \cdots \leq y_{a_n} \). An integer \( i \) is blue (resp., red) if \( g_i \) is blue (resp., red).

Consider the sequence \((a_1, \ldots, a_n)\). For each \( i \) such that \( a_i \) is red, find \( f_i = \max\{a_k \mid k < i, a_k < a_i, \text{ and } a_k \text{ is blue}\} \). Then, \( g_i \) is the friend of \( a_i \). The condition \( k < i \) implies that \( y_{a_k} < y_{a_i} \), and the condition \( a_k < a_i \) implies that \( x_{a_k} < x_{a_i} \). Computing the \( f_i \)'s can be done by the algorithm in Fig. 11.

In the algorithm, \( D \) is a sorted list of blue and red integers, initially \( D = (1, \ldots, n) \). In (1), we set \( D \) to include the integers \( 1, \cdots, n \). In (2)–(5), while scanning \((a_1, \ldots, a_n)\) backward, we compute \( f_i \) in (4) if \( a_i \) is red, and delete \( a_i \) in (5), otherwise.

For each red integer in \( D \), its neighbor is the largest one among the blue integers that are less than the red integer. If no such blue integer exists, then the neigh-

\[
(1) \quad \text{find } y_1 \text{ such that } y_1 = y_i^1; \\
(2) \quad \text{delete } (y_1, *) \text{ and } (\text{succ}(y_1), *) \text{ from } T^L; \\
(3) \quad \text{for each tuple } (y, *) \in B_i; \\
(4) \quad \text{if } (y, \infty) \text{ is already in } T^L \\
(5) \quad \text{replace the tuple } (y, \infty) \text{ in } T^L \text{ by the one from } B_i; \\
(6) \quad \text{insert } (y, *) \text{ into } T^L;
\]

Fig. 8 \( \text{DELETE}_S(l_i) \) at the upper side, and \( \text{SEARCH}_L(c_j) \) at the lower side.

\[
c_j = c_{j-1} + A[j]; \\
\text{if}(j < l) \quad \sigma_j = \infty; \\
\text{else} \quad \text{INSERT}_S(l_{i-j}); \\
\text{if}(j > J) \quad \text{DELETE}_S(l_{i-J-1}); \\
\sigma_j = \text{SEARCH}_R(c_j); \\
mind = \min(\text{mind}, j - \sigma_j); \\
\quad \text{Fig. 9} \quad \text{The code for finding the shortest feasible subsequence: this is executed whenever } A[j] \text{ arrives.}
\]

2.2 Finding the Shortest Feasible Subsequence

To find the shortest feasible subsequence we compute \( \sigma_j \), in stead of \( \lambda_j \), for all \( I \leq j \leq n \), where \( \sigma_j \) is the largest integer, if exists, \( A[\sigma_j + 1 \ldots j] \) is feasible. If no such integer exists, \( \sigma_j = \infty \). The shortest feasible subsequence of \( A \) can be located by computing \( k \) such that \( k - \sigma_k = \min\{j - \sigma_j \mid 1 \leq j \leq n\} \). \( A[\sigma_k + 1 \ldots k] \) is the shortest feasible subsequence. If \( k - \sigma_k = \infty \), then there is no feasible subsequence and NIL is output.

Let \( p_j^1 = (\infty, c_j) \) for \( I \leq j \leq n \) be a point. Draw a horizontal leftward ray from \( p_j^1 \) and find the first segment \( l_k \), if exists, in \( Q_j \) that is hit by the ray. Then, \( \sigma_j = k \). If no segment in \( Q_j \) is hit by the ray, then \( \sigma_j = \infty \).

\[ T^L \text{ and } T^R \text{ in Section 2.1 will be used to compute } \sigma_j \text{ for } I \leq j \leq n. \]

Initially, \( \text{mind} = \infty \). Whenever \( A[j] \) for each \( 1 \leq j \leq n \) arrives, the algorithm in Fig. 9 is executed. The following theorem can be proved in a similar manner as Theorem 1.

Theorem 2: The shortest feasible subsequence can be found by an online algorithm in \( O(n \log n) \) time.

3. The Offline Algorithms

In this section, we present offline algorithms for the problems of finding the longest and shortest feasible subsequences.)
1. \[D = (1, \ldots, n);\]
2. \[\text{for } (i = n \text{ to } 1)\]
3. \[\text{if } a_i \text{ is red } f_i = \text{neighbor of } a_i \text{ in } D;\]
4. \[\text{else } \text{delete } a_i \text{ from } D;\]

**Fig. 11** Computing the nearest blue neighbors

**Fig. 12** (a) The data structure for \(D = (1, 2, 3, 4, 5, 6, 7, 8)\) of Fig. 10, which has two blocks. The blue integers (\(\square\)) are stored in a doubly linked list and the red integers (\(\bigcirc\)) are stored into sets (or rooted trees). Each root points to its current blue neighbor.

(b) The data structure after deleting 5 is shown. The trees with root 3 and with root 6 are merged into one.

The point is \(-\infty\). A maximal subsequence of \((1, \ldots, n)\) that consists of red integers only is called a block.

To implement \(D\), a doubly linked list combined with a data structure for disjoint sets will be used. The doubly linked list stores the blue integers of \(D\), while the red integers are stored in the disjoint set data structure. We build a set for each block, which contains the integers of the block. The sets, as in the UNION-FIND problem of disjoint sets \([1],[3],[6]\), are implemented with disjoint rooted trees. In each tree, one of the integers in the block becomes the root and every non-root integer points to its parent only. Each root points to a blue integer in the doubly linked list, which is the neighbor of every integer in the block. Fig. 12 shows our data structure for the points in Fig. 10.

In (4) in Fig. 11, \(f_i\) can be found by calling FIND\((a_i)\), which locates the root of the tree containing \(a_i\), which in turn points to the neighbor of \(a_i\), i.e., \(f_i\). In (5), we delete \(a_i\), which is blue, from the doubly linked list. If \(a_i\) is pointed to by a root as its neighbor, the neighbor of the root and its tree must be changed to the predecessor of \(a_i\) in the doubly linked list. This can be accomplished by UNIONing two trees: one whose root points to \(a_i\) and the other whose root points to the predecessor of \(a_i\).

With \((a_1, \ldots, a_8) = (2, 3, 7, 1, 5, 6, 4, 8)\), and the data structure in Fig. 12(a), a partial result of the execution of the algorithm is as follows:

| \(i\) | \(a_i\) | FIND\((a_i)=\) | Points to | \(f=\) |
|---|---|---|---|---|
| 8 | 8 | 6 | 5 | 5 |
| 7 | 4 | 3 | 2 | 2 |
| 6 | 6 | 6 | 5 | 5 |

**Fig. 13** Point \(p_j\) and points \(p_{j-1}, \ldots, p_{j-t}\) are shown. The segment \(v_j\) is moving rightward and hits one of the points, which is \(p_{\lambda_j}\).

5. \([5]\) delete 5 UNION\((3,6)\)

**Lemma 1:** The nearest friend finding problem can be solved in \(O(n\alpha(n))\) time if both a \(x\)-sorted list and a \(y\)-sorted list of the points are given.

**Proof:** Construction of the data structure for \(D = (1, \ldots, n)\) can be done in linear time. A sequence of FINDs and UNIONs of length \(n\) is applied to \(D\), which can be accomplished in \(O(n\alpha(n))\) time \([1],[3],[6]\). \(\square\)

### 3.2 Finding the Longest Feasible Subsequence

This problem is also transformed into a geometric one. Let \(P = \{(x_i, y_i) \mid 0 \leq i \leq n\}\). Let \(P_j = \{p_i \mid \max\{0, j - I\} \leq i \leq j - I\} \text{ for } I \leq j \leq n\). Define a vertical segment \(v_j = (\max\{-0.5, j - J - 0.5\}, c_j - Y, c_j - X)\) for \(I \leq j \leq n\). Note that \(x\)-coordinate of \(v_j\) is less than those of \(p_i\)'s in \(P_j\). We move \(v_j\) horizontally rightward toward \(P_j\). Let \(p_k\) exist, be the first point in \(P_j\) hit by the moving segment. Then, \(\lambda_j = k\). If no such point in \(P_j\) is hit by the moving segment, then \(\lambda_j = \infty\). See Fig. 13.

For a subsequence \(A(i+1\ldots j), i < j\), to be feasible, it has to satisfy \(X \leq c_j - c_i \leq Y\) and \(I \leq j - i \leq J\); or equivalently, \(c_j - Y \leq c_i \leq c_j - X\) and \(j - I \leq i \leq j - I\). The points in \(P_j\) satisfy the length constraint \(j - I \leq i \leq j - I\) and the points hit by the moving segment satisfy the sum constraint \(c_j - Y \leq c_i \leq c_j - X\). The first point hit by the moving segment gives the longest feasible subsequence. Refer to Fig. 13.

Notice that the roles of points and segments in Section 2.1 are reversed here. In Section 2.1 the segments are fixed and the points are moving, while here the points are fixed and the segments are moving.

A *strip* of width \(Y - X\) is a region in the plane bounded by two horizontal lines that are \(Y - X\) apart. A strip is assumed to include its bottom boundary (or its bottom horizontal line), but not its top boundary (or its top horizontal line). A strip *covers* points if the points are in the strip.

To compute strips that are needed to cover \(P\), we
first sort the points in $P$ in increasing order of the $y$-coordinates, i.e., the $c_i$ values.\footnote{This is the only step where our algorithm for finding the longest feasible subsequence requires $O(n \log n)$ time. The remaining part will take only $O(n\alpha(n))$ time.} Let $P' = (p_{a_1}, \ldots, p_{a_n})$ be the sorted list. The algorithm in Fig. 14 is a greedy one, which finds disjoint strips that cover $P$ (or equivalently $P'$). The algorithm scans $P'$, opens a new $j$-th strip whose bottom boundary is at $y$-coordinate $b_j$ in (5), and, in (7)–(10), adds $p_{a_1}$ into $P'_j$ as long as its $y$-coordinate is less than $b_j + Y - X$, which corresponds to the $y$-coordinate of the top boundary of the strip.

Let $b_1, \ldots, b_m$ be the list of $b_j$ computed by the algorithm. Let $S_i, 1 \leq i \leq m$, be the strip whose bottom boundary is at $y$-coordinate $b_i$. Obviously, $S_1, \ldots, S_m$ are disjoint. Then, $P'_i = P' \cap S_i$ for $1 \leq i \leq m$.

Consider a segment $v_j$ for some $1 \leq j \leq n$. Since $v_j$ is $Y - X$ long, it can intersect at most two consecutive strips. Suppose that $v_j$ intersects two strips $S_o$ and $S_{o+1}$ for some $o$. The bottom point of $v_j$ is in $S_o$, and its top point is in $S_{o+1}$. Let $v'_j = v_j \cap S_o$ and $v''_j = v_j \cap S_{o+1}$. Move the segment $v'_j$ horizontally rightward and let $p_{\alpha j}$, if exists, be the first point in $P'_o$ hit by the segment. If no point in $P'_o$ is hit by the segment, then $\lambda'_j = \infty$. Similarly, move the segment $v''_j$ horizontally rightward and let $p_{\alpha j}$, if exists, be the first point in $P'_{o+1}$ hit by the segment. If no point in $P'_{o+1}$ is hit by the segment, then $\lambda''_j = \infty$. Let $\lambda_j = \min\{\lambda'_j, \lambda''_j\}$. If $\max(0, j - J) \leq \lambda_j \leq j - I$, then $\lambda_j = \lambda_j$. Otherwise, $\lambda_j = \infty$.

For $1 \leq i \leq m$, let $V^b_i = \{v^b_j \mid v_j \text{ has its bottom point in } S_i\}$, and $V^t_i = \{v^t_j \mid v_j \text{ has its top point in } S_i\}$. With $V^b_i$ and $P'_i$, compute $\lambda^b_j$ for each $j$ such that $v^b_j$ is in $V^b_i$, and with $V^t_i$ and $P'_i$, compute $\lambda^t_j$ for each $j$ such that $v^t_j$ is in $V^t_i$.

For $1 \leq j \leq n$, compute $\lambda_j = \min\{\lambda^b_j, \lambda^t_j\}$, and set $\lambda_j = \lambda_j$ if $\max(0, j - J) \leq \lambda_j \leq j - I$, and $\lambda_j = \infty$, otherwise.

Computing $\lambda^b_j$ for all $j$ such that $v^b_j$ is in $V^b_i$ is an instance of the nearest friend problem in Section 3.1. If we let $B = \{(x, y) \mid (x, y) \in P'_i\}$ and $R = \{(-x, y) \mid (x, y) \text{ is the top point of a segment in } V^t_i\}$, and solve the problem, we have $\lambda^b_j$ for all $j$ such that $v^b_j$ is in $V^b_i$. Similarly, $\lambda^t_j$ for all $j$ such that $v^t_j$ is in $V^t_i$ can be computed.

**Theorem 3:** The longest feasible subsequence can be found by an offline algorithm in $O(n \log n)$ time. Except one sorting of $n$ numbers, the algorithm runs in $O(n\alpha(n))$ time.

**Proof:** Computing $(a_1, \ldots, a_n)$ takes $O(n \log n)$ time. Finding $S_i$ and $P'_i$, $1 \leq i \leq m$, by the algorithm in 14 can be done in $O(n)$ time. Computing $V^b_i$ and $V^t_i$, $1 \leq i \leq m$, can be accomplished in $O(n)$ time. Computing $\lambda^b_j$ for all $j$ such that $v^b_j$ is in $V^b_i$ can be done in $O(n \cdot \alpha(n))$ time using the algorithm in Section 3.1, where $n_i = |V^t_i \cup P'_i|$, $1 \leq i \leq m$. Thus, $\lambda^b_j$ for $1 \leq j \leq n$ can be obtained in $O(n \alpha(n))$ time as $n_1 + \cdots + n_m \leq 2n$. Thus, $\lambda^b_j$ for $1 \leq j \leq n$ also can be found in $O(n \alpha(n))$ time. Excluding the sorting for obtaining $(a_1, \ldots, a_n)$, the algorithm takes $O(n \alpha(n))$ time. \qed

3.3 Finding the Shortest Feasible Subsequence

Define a vertical segment $v' = (j - I + 0.5, c_j - Y, c_j - X)$ for $I \leq j \leq n$. Note that $x$-coordinate of $v'_j$ is larger than those of $p_i$’s in $P'_j$. We move $v'_j$ horizontally leftward toward $P'_j$. Let $p_{x_j}$ exists, be the first point in $P'_j$ hit by the moving segment. Then, $\sigma_j = k$. If no such point in $P'_j$ is hit by the moving segment, then $\sigma_j = -\infty$.

Let $b_1, \ldots, b_m$ be the list of $b_j$ computed by the algorithm in Fig. 14. $S_i$ and $P'_i$ for $1 \leq i \leq m$ are the same as in Section 3.2. Compute $v^b_j = v'_j \cap S_o$ and $v^t_j = v'_j \cap S_{o+1}$ for all $j$, as in Section 3.2. Move the segment $v^b_j$ horizontally leftward and let $p_{\alpha j}$, if exists, be the first point in $P'_o$ hit by the segment. If no point in $P'_o$ is hit by the segment, then $\sigma^b_j = -\infty$. Similarly, move the segment $v^t_j$ horizontally leftward and let $p_{\alpha j}$, if exists, be the first point in $P'_o$ hit by the segment. If no point in $P'_o$ is hit by the segment, then $\sigma^t_j = -\infty$. Let $\tilde{\sigma}_j = \max\{\sigma^b_j, \sigma^t_j\}$. If $\max(0, j - J) \leq \tilde{\sigma}_j \leq j - I$, then $\sigma_j = \tilde{\sigma}_j$. Otherwise, $\sigma_j = -\infty$.

For $1 \leq j \leq n$, compute $\sigma^b_j$ for each $j$ such that $v^b_j$ is in $V^b_i$, and with $V^t_i$ and $P'_i$, compute $\sigma^t_j$ for each $j$ such that $v^t_j$ is in $V^t_i$.

For $1 \leq j \leq n$, compute $\sigma_j = \max\{\sigma^b_j, \sigma^t_j\}$, and set $\sigma_j = \sigma_j$ if $\max(0, j - J) \leq \sigma_j \leq j - I$, and $\sigma_j = -\infty$, otherwise.

Computing $\sigma^b_j$ for all $j$ such that $v^b_j$ is in $V^b_i$ is an instance of the nearest friend finding problem in Section 3.1. If we let $B = \{(x, y) \mid (x, y) \in P'_i\}$ and $R = \{(x, y) \mid (x, y) \text{ is the top point of a segment in } V^t_i\}$, and solve the problem, we have $\sigma^b_j$ for all $j$ such that $v^b_j$ is in $V^b_i$. Similarly, $\sigma^t_j$ for all $j$ such that $v^t_j$ is in $V^t_i$ can be computed.

The proof of Theorem 3 can be applied to prove the following theorem.
Theorem 4: The shortest feasible subsequence can be found by an offline algorithm in $O(n \log n)$ time. Except one sorting of $n$ numbers, the algorithm runs in $O(n a(n))$ time.

4. Conclusions

We have addressed the following problems: Given a sequence $A$ of $n$ real numbers, and four parameters $I, J, X$ and $Y$ with $I \leq J$ and $X \leq Y$, find the longest (or shortest) subsequence of $A$ such that its length is between $I$ and $J$ and its sum is between $X$ and $Y$. We have presented online and offline algorithms for the problems, both run in $O(n \log n)$ time, which are optimal. A lower bound proof is in [4], which proved that in the comparison model an $\Omega(n \log n)$ time is necessary to determine whether there is a subsequence of $A$ whose sum is zero. Solving any one of the two problems with $I = 1$, $J = n$, and $X = Y = 0$ will check whether there is a subsequence of $A$ whose sum is zero.

References

[1] A. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
[2] K.Y. Chen and K.M. Chao, Optimal algorithms for locating the longest and shortest segments satisfying a sum or an average constraint, Info. Process. Lett., 96:197–201, 2005.
[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, Second Ed., MIT Press and McGraw-Hill, 2001.
[4] Y.H. Hsieh, C.C. Yu, and B.F. Wang, Optimal algorithms for the interval location problem with range constraints on length and average, IEEE Trans. Comp. Bio. Bioinfo., 5(2):281–290, 2008.
[5] D. Takai, and P.A. Jones, Comprehensive analysis of CpG islands in human chromosome 21 and 22, PNAS, 99(6), pp. 3740–3745, 2002.
[6] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM, 22(2):215–225, 1975.