Finding the \( k \)-th smallest of the merge of two sorted arrays

**Problem:** Given two sorted arrays \( A[1:m] \) and \( B[1:n] \) of distinct numbers and an integer \( 1 \leq k \leq m+n \), find the \( k \)-th smallest element in the merge of \( A \) and \( B \) in \( O(\log k) \) time.

**Solution**

**Idea**

Suppose \( \lfloor \frac{k}{2} \rfloor \leq m+n \) and consider comparing \( A[\lfloor \frac{k}{2} \rfloor] \) and \( B[\lfloor \frac{k}{2} \rfloor] \):

\[
\begin{align*}
A & \quad \lfloor \frac{k}{2} \rfloor \quad m \\
B & \quad \lfloor \frac{k}{2} \rfloor \quad n \\
\end{align*}
\]

Assuming w.l.o.g. \( A[\lfloor \frac{k}{2} \rfloor] < B[\lfloor \frac{k}{2} \rfloor] \), we can throw out \( \alpha := A[1: \lfloor \frac{k}{2} \rfloor] \) and \( \beta := B[\lfloor \frac{k}{2} \rfloor+1:n] \):

(i) if the \( k \)-th smallest were in region \( \alpha \), the first \( k \) elements must include \( B[\lfloor \frac{k}{2} \rfloor] \), yet this element must come after all elements in \( \alpha \); and

(ii) if the \( k \)-th smallest were in region \( \beta \), the first \( k \) elements could not include \( A[\lfloor \frac{k}{2} \rfloor] \), yet this element must come before all elements in \( \beta \).

Since by (i) we can ignore \( A[1: \lfloor \frac{k}{2} \rfloor] \) and by (ii) these elements must appear in the first \( k \), we can recursively search for the \( \lfloor \frac{k}{2} \rfloor \)-th smallest in \( A[\lfloor \frac{k}{2} \rfloor+1:m] \) and \( B[1: \lfloor \frac{k}{2} \rfloor] \).
If \( m < \lfloor \frac{k}{2} \rfloor \), we can throw out \( \beta := B[1 : k-m-1] \):

\[
\begin{align*}
A & \quad \underline{m} \quad \underline{\lfloor \frac{k}{2} \rfloor} \\
B & \quad \underline{k-m} \quad \underline{k} \quad \underline{n} \\
\end{align*}
\]

if the \( k \)th smallest were in region \( \beta \), there would not be enough elements in \( A \) to make a total of \( k \) elements. This also shows that all of region \( \beta \) must appear in the first \( k \) elements. Thus we can recursively search for the \( k-(k-m-1) = (m+1) \)th smallest of \( A[1:m] \) and \( B[k-m:n] \). (Notice that \( m+1 \leq \lfloor \frac{k}{2} \rfloor \), and that the other invariants also hold.)

**Implementation**

```plaintext
function MergedKthSmallest(A, B, k, PA, QA, PB, QB) begin
    m := QA - PA + 1
    n := QB - PB + 1
    if m \neq n then
        Swap A and B, m and n, PA and PB, QA and QB.
    T(k) = T(\frac{k}{2}) + T(\frac{k}{2}) + \Theta(\log k)
    if m = 0 then
        return B[PB + k - 1]
    else if k = 1 then
        return min \{ A[PA], B[PB] \}
    else if \( \lfloor \frac{k}{2} \rfloor \neq m \) then
        return MergedKthSmallest(A, B, m+1, PA, QA, PB + k - m - 1, QB)
    else if A[\lfloor \frac{k}{2} \rfloor] < B[\lfloor \frac{k}{2} \rfloor] then
        return MergedKthSmallest(A, B, \lfloor \frac{k}{2} \rfloor, PA + \lfloor \frac{k}{2} \rfloor, QA, PB, PB + \lfloor \frac{k}{2} \rfloor)
    else
        return MergedKthSmallest(A, B, \lfloor \frac{k}{2} \rfloor, PA, PA + \lfloor \frac{k}{2} \rfloor - 1, PB + \lfloor \frac{k}{2} \rfloor - 1, QB)
end
```

Find the \( k \)th smallest in the merge of sorted arrays \( A[PA:QA] \) and \( B[PB:QB] \). Assumes \( PA \leq QA \), \( PB \leq QB \),
\( 1 \leq k \leq QA - PA + QB - PB + 2 \), and that the numbers in \( A \) and \( B \) are all distinct.
Problem (Finding a pair of close elements)

Given an unsorted array \( A[1:n] \) of distinct numbers, find a pair of elements \( x > y \) in \( A \) such that

\[
x - y \leq \frac{1}{n-1} \left( \max_{1 \leq i \leq n} A[i] - \min_{1 \leq i \leq n} A[i] \right),
\]

in \( O(n) \) time.

Solution

If we consider the elements \( x_1 < x_2 < \ldots < x_n \) of \( A \) in sorted order, there are \( n-1 \) pairs of successive numbers \( (x_i, x_{i+1}) \), whose differences add up to \( x_n - x_1 \):

\[
\sum_{1 \leq i < n} (x_{i+1} - x_i) = x_n - x_1 = \max_{1 \leq i \leq n} A[i] - \min_{1 \leq i \leq n} A[i]
\]

Some pair \( (x_i, x_{i+1}) = (y, x) \) must have a difference \( x - y \) that is at most the average of the \( n-1 \) differences. Thus the problem always has a solution (for \( n \geq 2 \)).

To find such a pair, we can determine the median element

\[
x_m := x_{\left\lfloor \frac{n+1}{2} \right\rfloor}
\]

partition the array around \( x_m \) into two halves, and search in the half whose average difference is at most (1). (At least one of the halves must satisfy this.)

function \text{Close Pair}(A, n) begin

\( \Theta(n) \)\{

Compute the median element \( a_{\text{med}} \) of \( A \), which has rank \( \left\lfloor \frac{n+1}{2} \right\rfloor \), and the maximum \( a_{\text{max}} \) and minimum \( a_{\text{min}} \) elements of \( A \).

Partition \( A[1:n] \) into two halves \( L[1: \left\lfloor \frac{n+1}{2} \right\rfloor] \) and \( R[1:n - \left\lceil \frac{n+1}{2} \right\rceil] \) such that \( L[i] \leq a_{\text{med}} \) and \( R[j] > a_{\text{med}} \) for all \( i, j \) in \( L, R \).

Compute the minimum \( l_{\text{min}} \) and maximum \( l_{\text{max}} \) of \( L \).

if \( \frac{1}{\left\lceil \frac{n+1}{2} \right\rceil - 1} (l_{\text{max}} - l_{\text{min}}) \leq \frac{1}{n-1} (a_{\text{max}} - a_{\text{min}}) \) then

return \text{Close Pair}(L, \left\lfloor \frac{n+1}{2} \right\rfloor)

else return \text{Close Pair}(R, n - \left\lceil \frac{n+1}{2} \right\rceil)

end

Analysis

\( T(n) = T(\frac{n}{2}) + \Theta(n) \)

= \( \Theta(n) \) time.
**Problem (Generalized k-th smallest)**

Given $n$ distinct unsorted numbers $x_1, \ldots, x_n$ with associated positive weights $w_1, \ldots, w_n$ where $W := \sum_{i=1}^{n} w_i$, and a number $k$ with $0 \leq k \leq W$, find element $x_j$ such that

$$\sum_{x_i < x_j} w_i \leq k \quad \text{and} \quad \sum_{x_i > x_j} w_i \leq W - k,$$

in $\Theta(n)$ time.

**Algorithm**

Our algorithm proceeds as follows:

1. Find the median $X$ of the $x_i$ using the worst-case linear-time algorithm for finding the (ordinary) $k$-th smallest.

2. Partition the $x_i$ around the median $X$ into a left portion of $x_i < X$ and a right portion of $x_i > X$.

3. Evaluate $L := \sum_{x_i \leq X} w_i$ and $R := \sum_{x_i > X} w_i$.

4. If $L \leq k$ and $R \leq W - k$ then return $X$.
   
   Else if $L > k$ then recursively find the generalized $k$-th smallest over the left portion with $W' := L$.
   
   Otherwise recursively find the generalized $(k - L)$-th smallest over the right portion with $W' := R$.

**Analysis**

This takes time

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n).$$
(Longest balanced subsequence) (50 points) A string of parentheses is said to be balanced if the left- and right-parentheses in the string can be paired off properly. For example, the strings “(())” and “()()” are both balanced, while the string “(()” is not balanced.

Given a string S of length n consisting of parentheses, suppose you want to find the longest subsequence of S that is balanced. Using dynamic programming, design an algorithm that finds the longest balanced subsequence of S in O(n³) time.

(1) (Structure of optimal solution)

The longest balanced subsequence (LBS) of S[1:n], call it W, must end by choosing both S[1] and S[n] or not, which leads to the following four cases:

Case (i) W uses both S[1] and S[n], and these parentheses are paired with each other:

```
<table>
<thead>
<tr>
<th>S</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>n-1</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Must be an LBS of S[2:n-1].

Parentheses are paired off.

Case (ii) We use both S[1] and S[n], but these parentheses are not paired with each other:

```
<table>
<thead>
<tr>
<th>S</th>
<th>1</th>
<th>...</th>
<th>k</th>
<th>k+1</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Must be an LBS of S[1:k].

Must be an LBS of S[k+1:n].
Case (iii) \( W \) does not use \( S[i] \).

\[
\begin{array}{c}
S \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qu
The value of the solution to the input problem is:
\[ L(1, n). \]

(3) (Evaluation phase)

We evaluate the recurrence in a table \( L[1:n, 0:n] \). The dependencies among entries are:

These dependencies can be satisfied by filling in the upper triangle of table \( L \) in diagonal-major order:

There are \( O(n^2) \) entries in the table, and each takes \( O(n) \) time to evaluate by the recurrence, so the evaluation phase runs in \( O(n^3) \) time.
(4) (Recovery phase)

To recover the LBS, we start at the goal entry \( L[1,n] \), determine which of cases (i)-(iv) gave its value, and recurse (possibly on two subproblems in case (iii)).

The recursive calls in this process can be mapped to the nodes of a binary tree (in which nodes have 1 child in cases (i), (iii), (iv), and 2 children in case (ii)) that has \( O(n) \) total nodes. Each node takes \( O(n) \) time to determine which case gave its value, so the recovery phase takes \( O(n^2) \) total time.

The entire algorithm using both phases takes total time \( O(n^3) + O(n^2) = O(n^3) \).
Problem (Two-finger dialing)

Given an \( k \)-digit telephone number \( T[1:n] = t_1 t_2 \ldots t_n \),
dial it with two fingers starting on the `*` and `#` keys
so as to minimize the total distance the fingers travel
in \( O(n) \) time.

Solution

(i) Consider how an \underline{optimal} dialing of \( T \) ends. Prior to
pressing the key \( T[n] \), the two fingers must have
been in some final state

\[(f_1, f_2) \in \mathcal{F} (a, b) : a, b \in K_3^2,\]

where \( K = \{0, 1, \ldots, 9, *, #\} \).

Furthermore, the sequence that dialed the prefix
\( T[1:n-1] \) must be an \underline{optimal} dialing that ends
in state \( (f_1, f_2) \) (as can be shown using proof
by contradiction).

This leads to the subproblem

\[(i, a, b) := \text{find an \underline{optimal} dialing of the prefix } T[1:i], \text{ that}
\quad \text{ends in state } (a, b) \].

Let

\[D_{a, b} (i) := \text{total distance travelled in an}
\quad \text{optimal dialing of } T[1:i], \text{ that}
\quad \text{ends in state } (a, b), \]

where \( D_{a, b} (i) = \infty \) if it's impossible to end at \( (a, b) \)
after dialing \( T[1:i] \).

Then the solution value for the input problem is

\[\min_{a, b \in K} \left\{ D_{a, b} (n) \right\}.\]
Solution cont'd

(ii) For \( a, b \in K \) let

\[ d(a, b) := \text{Euclidean distance between keys } a, b. \]

A recurrence for \( D \) for \( i \geq 1 \) is

\[
D_{a, b}(i) = \min_{c \in K} \begin{cases} d(a, c) + D_{c, b}(i-1) & \text{if } a = T[i], \\ d(c, b) + D_{a, c}(i-1) & \text{if } b = T[i], \\ \infty & \text{otherwise} \end{cases}
\]

where for \( i = 0 \),

\[
D_{a, b}(0) = \begin{cases} 0, & (a, b) = (\ast, \ast), \\ \infty, & \text{otherwise}. \end{cases}
\]

(iii, iv) We can evaluate recurrence (*) in a 3-dimensional table \( D[a, b, i] \) that has \( O(|K|^2 n) \) entries.

Filling in a given entry using (*), for increasing \( i \), takes \( O(|K|) \) time using table lookup, for a total time of

\( O(|K|^2 n) = O(n) \),

since \( |K| = 12 = O(1) \).

Recovering the optimal dialing from the \( D \) table also takes \( O(|K| n) = O(n) \) time.
Exercise

longest increasing subsequence in $O(n \log n)$ time

Given a sequence $A = a_1, a_2, \ldots, a_n$, we wish to find a longest strictly monotonically increasing subsequence. We first develop a $O(n^2)$ time algorithm, and then speed it up to $O(n \log n)$ time using a balanced search tree.

Let

$$L(i) := \text{length of a longest strictly monotonically increasing subsequence over } a_1, \ldots, a_i \text{ that ends with } a_i.$$ 

Then the solution value is $\max \{ L(i) \}$. A recurrence for $L(i)$ is

$$L(i) = 1 + \max \left\{ L(j) \right\},$$

where the maximum of an empty set is taken to be zero. (If a subsequence that is not strict is sought, replace "$a_j < a_i$" by "$a_j \leq a_i$" in the above.)

To recover the subsequence solution, we compute

$$P(i) := \text{index of the preceding element on a longest increasing subsequence ending with } a_i,$$

where, if there is no preceding element, the index is taken to be zero.
The following algorithm evaluates $L$ via the recurrence bottom-up, left to right across $A$.

\[
\text{Evaluate } L1S (A, L, P, n) \begin{align*}
\text{begin} & \quad \forall v \in A[1..n], \\
\forall i : 1 + n \text{ do } \begin{align*}
L[i] & := 1 \\
P[i] & := 0 \\
\text{for } j : 1 \text{ to } i-1 \text{ do } \begin{align*}
& \text{if } A[j] < A[i] \text{ and } L[j] + 1 > L[i] \text{ then begin } \\
& \quad L[i] := L[j] + 1 \\
& \quad P[i] := j \\
& \text{end} \\
& \text{end} \\
& \text{end} \\
\end{align*} \\
\end{align*} \\
\text{end}
\]

$O(n^2)$ time

\[
\text{Print } L1S (A, L, P, n) \begin{align*}
& \text{begin} \\
& \quad i := \text{argmax} \{ L[j] \} \\
& \quad \text{Print Helper } (A, L, P, i) \\
& \text{end}
\end{align*} \\
\]

\[
\text{Print Helper } (A, L, P, i) \begin{align*}
& \text{begin} \\
& \quad \text{if } k > 0 \text{ then begin } \\
& \quad \quad \text{Print Helper } (A, L, P, P[k]) \\
& \quad \quad \text{print } A[k] \\
& \text{end}
\end{align*} \\
\]

\[
\text{end}
\]
Next observe that, to evaluate \( \max \{ L(j) \} \) for \( 1 \leq j < i \) a fixed \( i \), it suffices to record, for a given element value \( v \), the index \( j < i \) for which \( a_j = v \) and \( L(j) \) is largest. Let the set of these (element, index, length) triples over \( a_1 \ldots a_i \) be

\[
\left\{ (a_{j_1}, j_1, l_1), (a_{j_2}, j_2, l_2), \ldots, (a_{j_t}, j_t, l_t) \right\}
\]

where \( a_{j_1} < a_{j_2} < \ldots < a_{j_t} \) (i.e., the triples are in order of increasing element).

Second, observe that, for two triples \((a_{j_p}, j_p, l_p)\), \((a_{j_q}, j_q, l_q)\) where \( a_{j_p} < a_{j_q} \), if \( l_p > l_q \), we can throw out triple \((a_{j_q}, j_q, l_q)\). (Any solution extending \( a_{j_q} \) also extends \( a_{j_p} \) and the \( a_{j_p} \)-extension will be at least as long.) Thus, for this reduced set of triples, \( l_1 < l_2 < \ldots < l_t \) (i.e., as the elements increase, so do the associated lengths).

So, to evaluate \( \max \{ L(j) \} \), it suffices to find the immediate predecessor of element \( a_i \) in a search tree over the reduced triples, where triples are ordered by increasing element. As there are \( O(n) \) triples, this takes \( O(\log n) \) time.
This gives the following algorithm.

EvaluateLIS \( (A, L, P, n) \) begin

\[ T := \text{Tree}() \]

\[ \text{for } i := 1 \to n \text{ do begin} \]

\[ \text{Find max } \{ L(j) \} \text{ where } i < j \quad \rightarrow \quad (a, j, l) := \text{Predecessor} (A[i], T) \]

\[ L[i] := l + 1 \]

\[ P[i] := j \]

\[ (a, j, l) := \text{Find} (A[i], T) \]

\[ \text{if } L[i] > l \text{ then} \]

\[ \text{Insert} (A[i], i, L[i], T) \]

\[ (a, j, l) := \text{Successor} (A[i], T) \]

\[ \text{while } L[i] > l \text{ do begin} \]

\[ \text{Delete} (a, T) \]

\[ (a, j, l) := \text{Successor} (A[i], T) \]

\[ \text{end} \]

\[ \text{end} \]

\[ \text{end} \]

The total time for all calls to Predecessor and Find is \( O(n \log n) \).

The total time for all calls to Successor and Delete in the while-loop is also \( O(n \log n) \): each call deletes a triple, any triple can be deleted only once, and there are \( O(n) \) triples in total (one for each position in \( A \)). Thus the algorithm runs in \( O(n \log n) \) time.