3. (Finding elements near the median) (35 points) Given an unsorted array $A$ of $n$ distinct numbers, and an integer $k$ where $1 \leq k \leq n$, design an algorithm that finds the $k$ numbers in $A$ that are closest in value to the median of $A$ in $\Theta(n)$ time.

(Note: the position of the elements in the array with respect to the median is irrelevant; only their is important. The numbers that are closest in value to the median may be larger or smaller than the median.)

Consider the following algorithm:

1. Find the median of $A$, call it $x$.
2. Form another array $B[1:n]$ where
3. Find the $k$th smallest element in $B$, call it $y$.

Using the linear-time $k$th smallest algorithm for Steps (1) and (3), the entire algorithm runs in $\Theta(n)$ time.
Problem (Finding quantiles)

Given an unsorted array of numbers $A[1:n]$ and an integer $k$ where $1 < k \leq n$, find $k-1$ elements of $A$ whose ranks divide the sorted array into $k$ pieces that are of equal size (to within one unit), in $O(n \log k)$ time.

Solution

Idea

We use the following strategy:

1. Compute the index $i$ of the $\left\lfloor \frac{n}{k} \right\rfloor$th $k$-quantile.
2. Find the $i$th-smallest element in the array; call it $x$. (This is the $\left\lfloor \frac{n}{k} \right\rfloor$th $k$-quantile.)
3. Partition the array around pivot element $x$.
4. Recurse on both halves.

To compute the index $i$, we consider an apportionment into pieces of size $\left\lfloor \frac{n}{k} \right\rfloor$ and $\left\lceil \frac{n}{k} \right\rceil$. In this division, the first $n \mod k$ pieces have size $\left\lfloor \frac{n}{k} \right\rfloor + 1$, and the remainder of the $k$ pieces have size $\left\lfloor \frac{n}{k} \right\rfloor$. 
Problem cont'd

Implementation

procedure Quantiles (A, p, q, k) begin

if \( k > 1 \) then begin
    \( n := q - p + 1 \)
    \( r := n \mod k \)
    if \( \lfloor \frac{k}{2} \rfloor \leq r \) then
        \( i := \lfloor \frac{k}{2} \rfloor \frac{n}{k} \)
    else
        \( i := r \lfloor \frac{n}{k} \rfloor + (\lfloor \frac{k}{2} \rfloor - r) \frac{n}{k} \)
end

\( \Theta(n) \) \{ Select the \( i \)th-smallest element, call it \( x \), of \( A[p..q] \) \}
\( T(i, \frac{k}{2}) \) \{ Partition \( A[p..q] \) around element \( x \) \}
\( T(n-i, \frac{k}{2}) \) \{ Quantiles (\( A, p+i, q, \lfloor \frac{k}{2} \rfloor \)) \}
\( T(n-i, \frac{k}{2}) \) \{ Quantiles (\( A, p+i, q, \lfloor \frac{k}{2} \rfloor \)) \}
end

Find the \( k \)th quantiles of \( A[p..q] \).
Analysis

We get the recurrence

\[ T(n, k) = T\left(i, \frac{k}{2}\right) + T\left(n-i, \frac{k}{2}\right) + \Theta(n) \]

Suppose

\[ T(n, k) \leq a \cdot n \log k \]

Substituting,

\[
\begin{align*}
T(n, k) &\leq \max_{1 \leq i < n} \left\{ \frac{n}{2} T\left(i, \frac{k}{2}\right) + T\left(n-i, \frac{k}{2}\right) + \Theta(n) \right\} \\
&\leq \max_{1 \leq i < n} \left\{ a \cdot i \cdot \log \frac{k}{2} + a \cdot (n-i) \cdot \log \frac{k}{2} + \Theta(n) \right\} \\
&= \max_{1 \leq i < n} \left\{ a \cdot n \cdot \log \frac{k}{2} \right\} + \Theta(n) \\
&= a \cdot n \cdot \log k - a \cdot n + \Theta(n) \\
&\leq a \cdot n \cdot \log k \text{ if } a \text{ is chosen large enough.}
\end{align*}
\]

So

\[ T(n, k) = \Theta(n \log k). \]
Problem (Longest Palindromic Subsequence)
Given string $S[1:n]$, find the longest subsequence of $S$ that is a palindrome in $O(n^2)$ time.

Solution: We derive a dynamic programming algorithm using the 4-part framework.

1. (Structure)
A longest palindromic subsequence (LPS) of $S[1:n]$, call it $W$, must end in 3 possible ways:

(i) $W$ uses both $S[1]$ and $S[n]$:

$S[1 \ldots n-1]$ $\quad$ Rest must be an LPS of $S[2:n-1]$

Letters must match.

This can only occur if $S[1] = S[n]$.
The rest of $W$ must be an LPS of $S[2:n-1]$
(which can be proved by contradiction).

(ii) $W$ does not use $S[1]$:

$S[1 \ldots n]$ $\quad$ Must be LPS of $S[2:n]$.

(iii) $W$ does not use $S[n]$:

Similar to Case (ii), $W$ must be an LPS of $S[1:n-1]$. 
Solution, cont'd

(2) (Recurrence)

The general form of a subproblem that arises is to compute an LPS over a substring of $S$, say $S[i:j]$, which can be described by the pair $(i,j)$. Let

$$L(i,j) := \text{length of an LPS of } S[i:j].$$

Then by Part (1),

$$L(i,j) = \begin{cases} 
L[i+1:j-1]+1, & \text{if } S[i]=S[j]; \\
\max \{ L[i+1:j], L[i:j-1] \}, & \text{otherwise.} 
\end{cases}$$

The length of an LPS for the input string $S$ is $L(1,n)$.

(3) (Evaluation)

We evaluate the recurrence in a two-dimensional table $L[1:n+1, 0:n]$.
Solution, cont'd

The entries \((i, i-1)\) on the main diagonal contain the boundary values.

In general, entry \((i, j)\) depends on the 3 entries \((i, j-1), (i+1, j-1), (i+1, j)\).

Filling in the table in its upper triangle in diagonal-major order

or a kind of upward-row-major order

satisfies the dependencies.

There are \(\Theta(n^2)\) entries to evaluate, and each entry takes \(\Theta(1)\) time, using the recurrence.

So the evaluation phase takes \(\Theta(n^2)\) total time.
Solution, cont'd

(4) (Recovery)

We can recursively recover the LPS of $S$ from table $L$ by calling the following procedure:

Recover $(L, S, i, j)$, which determines which of cases (i), (ii), or (iii) gave the optimal solution:

\[
\text{procedure } \text{Recover} \ (L, S, i, j) \ \text{begin}
\]

\[
\text{outputs an LPS of } S[i:j]
\]

\[
\text{using table } L.
\]

\[
\text{if } i = j \ \text{then}
\]

\[
\text{return}.
\]

\[
\text{else}
\]

\[
\text{if } S[i] = S[j] \text{ and } L[i,j] = L[i+1,j-1] + 1
\]

\[
\text{then begin}
\]

\[
\text{Case (i)}
\]

\[
\text{begin}
\]

\[
\text{output } S[i]
\]

\[
\text{Recover } (L, S, i+1, j-1)
\]

\[
\text{output } S[j]
\]

\[
\text{end}
\]

\[
\text{else if } L[i,j] = L[i+1,j] \text{ then}
\]

\[
\text{Case (ii)}
\]

\[
\text{Recover } (L, S, i, j+1)
\]

\[
\text{Case (iii)}
\]

\[
\text{Recover } (L, S, i, j-1)
\]

\[
\text{end}
\]

This spends $\Theta(1)$ time per call and recurses on one subproblem of size $S[n-1]$, which takes $O(n)$ total time.

The entire algorithm from parts (3), (5) takes total time $\Theta(n^2) + O(n) = \Theta(n^2)$.
Prob. --- Bitonic Euclidean travelling salesman tour in $O(n^2)$ time.

Deriving a recurrence

We ask, how does a solution end?

Either

The last points are in the upper half of the tour.

or

The last points are in lower half.

In either case, the prefix of the solution over points 1,2,...,k must be as short as possible.

Hence let us compute

$$C_U(k) \equiv \text{length of a shortest tour-prefix over points } 1,2,...,k \text{ that ends at } k$$

and

$$C_L(k) \equiv \text{len. of shortest tour-prefix over pts } 1,2,...,k \text{ ending at } k$$
Then the value of the solution is

\[
\min_{1 \leq k \leq n} \left\{ \min \left\{ C_U(k), C_L(k) \right\} + d(k-1,n) + \sum_{k \leq j < n} d(j,j+1) \right\}
\]

Distance between points \(k-1\) and \(n\).

We next derive a recurrence for \(C_U(i)\):

\[
C_U(i) = \begin{cases} 
\min_{1 < k < i} \left\{ C_L(k) + d(k-1,i) \right\}, & 2 < i < n; \\
\min_{1 < k < i} \left\{ C_U(k) + d(k-1,i) \right\}, & i = 2; \\
d(1,2), & i = 1.
\end{cases}
\]

Similarly for \(C_L(i)\):

\[
C_L(i) = \begin{cases} 
\min_{1 < k < i} \left\{ C_U(k) + d(k-1,i) \right\}, & 2 < i < n; \\
\min_{1 < k < i} \left\{ C_L(k) + d(k-1,i) \right\}, & i = 2; \\
d(1,2), & i = 1.
\end{cases}
\]
Evaluating the recurrence

Computing $c_u(i)$ and $c_l(i)$ side-by-side for increasing $i$ is a valid evaluation order. To obtain an $O(n^2)$-time algorithm we must evaluate the sum $\sum_{k \leq j < i} d(j, j+1)$ quickly. To do this, compute

$$S(i) = \sum_{1 \leq j < i} d(j, j+1)$$

by

$$S(i) = \begin{cases} S(i-1) + d(i-1, i), & 1 < i \leq n; \\ 0, & i = 1. \end{cases}$$

Then

$$\sum_{k \leq j < i} d(j, j+1) = S(i) - S(k).$$

The next page gives the full procedure.
Bitonic Tour Length \((X, Y, n)\) begin

\[
\begin{align*}
S[1] &:= 0 \\
\text{for } i &:= 2 \text{ to } n \text{ do} \\
S[i] &:= S[i-1] + \text{DISTANCE}(X, Y, i-1, i) \\
U[2] &:= \text{DISTANCE}(X, Y, 1, 2) \\
L[2] &:= \infty \\
\text{for } i &:= 3 \text{ to } n-1 \text{ do begin} \\
U[i] &:= \infty \\
L[i] &:= \infty \\
\text{for } k &:= 2 \text{ to } i-1 \text{ do begin} \\
U[i] &:= \min \{ U[i], \} \\
L[k] &:= \text{DISTANCE}(X, Y, k-1, i) + S[i] - S[k] \\
L[i] &:= \min \{ L[i], \} \\
U[k] &:= \text{DISTANCE}(X, Y, k-1, i) + S[i] - S[k] \\
\text{end} \\
\text{end} \\
\text{return } U, L, S \\
\end{align*}
\]

Uses auxiliary arrays:
- \(U[2..n-1]\) for \(C_u(i)\),
- \(L[2..n-1]\) for \(C_l(i)\),
- \(S[1..n]\) for \(\Sigma d(j, j+1)\).

Away of \(x\) and \(y\)-coordinates of points. We assume \(x[i] < x[i+1] < \ldots < x[2n]\).
We can recover the optimal tour from arrays \( U \) and \( L \). We represent a tour by a string of \( U \)'s and \( L \)'s specifying, for each point from \( 2 \) to \( n-1 \), whether it is in the upper or lower half.

PRINT BITONIC TOUR \((X, Y, n)\) begin

\[ U, L, S := \text{BITONIC TOUR LENGTH} (X, Y, n). \]
Scan \([U[2] \text{ to } U[n-2]] \) and \([L[2] \text{ to } L[n-1]] \)

to find the index \( k \) at which

\[
\min \{ U[k], L[k] \} + \text{DISTANCE} (X, Y, k-l, n) \\
+ S[n] - S[k]
\]

attains its minimum value.

If minimum is attained with \( U[k] \) then begin

RECURSIVE UPPER PRINT TOUR \((U, L, S, k)\)

print \( n-k \) "U"s

end else begin

RECURSIVE LOWER PRINT TOUR \((U, L, S, k)\)

print \( n-k \) "L"s

end

end
The time to recover the optimal bitonic tour is $T(n) = \Theta(n) + T(n-1) = O(n^2)$

end

print i-k

Recursive {lower}

Print Tour (U, S, k)

Recursive {upper}

Print Tour (U, S, k)

\[
\{ \text{lower} \} + \text{distance}(k-1) + S_{(k-j)} - S_{(k)}
\]

atts its minimum value.

the index \( k \) at which

Scan array \{ U[k] \} to find

\{ U[2] \} to \{ U[i-1] \}

Recuive {upper}

PrintTour (U, S, i)

begin
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 $\Theta(n^2)$ 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 $\max_{1 \leq i \leq n} \{L(i)\}$. A recurrence for $L(i)$ is

$$L(i) = 1 + \max_{1 \leq j < i} \{L(j)\}, \quad a_j < a_i$$

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

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

where, if there is no preceding element, the index is taken to be zero.
Ex. cont’d.

The following algorithm evaluates $L$ via the recurrence bottom-up, left to right across $A$.

Evaluate $LIS (A, L, P, n)$ begin
Always $A[1..n]$;
$L[1..n]$;
$P[1..n]$.

for $i := 1$ to $n$ do begin
  $L[i] := 1$
  $P[i] := 0$

  for $j := 1$ to $i-1$ do
      $L[i] := L[j] + 1$
      $P[i] := j$
    end
  end
end

Print $LIS (A, L, P, n)$ begin
  $i := \arg\max \{ L[j] \}$ for $1 \leq j \leq n$
  Print $Helper (A, L, P, i)$
end

Print $Helper (A, L, P, k)$
if $k > 0$ then begin
  Print $Helper (A, L, P, P[k])$
  print $A[k]$
end
Next observe that, to evaluate \( \max \{ L(j) \} \) for \( 1 \leq j < i \) \( a_j < a_i \),
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 \( a_i \ldots a_i \) be
\[
\{ (a_{j_1}, j_1, l_1), (a_{j_2}, j_2, l_2), \ldots, (a_{j_t}, j_t, l_t) \}
\]
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 \( (a_{jp}, j_p, l_p) \),
\( (a_{j_q}, j_q, l_q) \) where \( a_{jp} < a_{j_q} \), 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
\( a_{jp} \), and the \( a_{jp} \)-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 \((A, L, P, r, n)\) begin
\[ T := \text{Tree}(\) \]
\[ \text{for } i := 1 \text{ to } n \text{ do begin} \]
\[ \text{Find max } \{L(j)\} \rightarrow (a, j, i) := \text{Predecessor } (A[i], T) \]
\[ L[i] := i + 1 \]
\[ P[i] := j \]
\[ (a, j, i) := \text{Find } (A[i], T) \]
\[ \text{if } L[i] > i \text{ then} \]
\[ \text{Insert } (A[i], i, L[i], T) \]
\[ (a, j, i) := \text{Successor } (A[i], T) \]
\[ \text{while } L[i] \geq i \text{ do begin} \]
\[ \text{Delete } (a, T) \]
\[ (a, j, i) := \text{Successor } (A[i], T) \]
\[ \text{end} \]
\[ \text{end} \]
end

Returns \((a, 0, 0)\) if no predecessor.

Returns \((a, 0)\) if not found.

Returns \((a, \infty, \infty)\) if no successor.

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.