(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

$$B[i] := |A[i] - x|.$$

(3) Find the k^{th} smallest element in B, call it y.

(4) Scan A and output all $A[i]$ where $B[i] \leq y$.

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

$\square$

Problem (Finding quantiles)

Given an unsorted array of numbers $A[1:n]$ and an integer k where $1 \le k \le n$, 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 $\lfloor \frac{k}{2} \rfloor^{th}$ k-quantile.

(2) Find the i^{th}-smallest element in the array; call it x. (This is the $\lfloor \frac{k}{2} \rfloor^{th}$ 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 $\lfloor \frac{n}{k} \rfloor$ and $\lceil \frac{n}{k} \rceil$. In this division, the first $n \bmod k$ pieces have size $\lfloor \frac{n}{k} \rfloor + 1$, and the remainder of the k pieces have size $\lfloor \frac{n}{k} \rfloor$.

Problem cont'd

Implementation

<u>procedure</u> Quantiles (A, p, q, k) <u>begin</u>

Find the $k-1$ k^{th} quantiles of $A[p..q]$.

<u>if</u> $k > 1$ <u>then</u> <u>begin</u>

$n := q - p + 1$

$r := n \bmod k$

<u>if</u> $\lfloor \frac{k}{2} \rfloor \leq r$ <u>then</u>

$i := \lfloor \frac{k}{2} \rfloor \lceil \frac{n}{k} \rceil$

<u>else</u>

$i := r \lceil \frac{n}{k} \rceil + (\lfloor \frac{k}{2} \rfloor - r) \lfloor \frac{n}{k} \rfloor$

$\Theta(n)$ $\left\{ \begin{array}{l} \text{Select the } i^{th}\text{-smallest element, call it } x, \text{ of } A[p..q] \\ \text{Partition } A[p..q] \text{ around element } x. \end{array} \right.$

$T(i, \frac{k}{2})$ $\{$ Quantiles $(A, p, p+i-1, \lfloor \frac{k}{2} \rfloor)$

<u>output</u> $A[i]$

$T(n-i, \frac{k}{2})$ $\{$ Quantiles $(A, p+i, q, \lceil \frac{k}{2} \rceil)$

<u>end</u>

<u>end</u>

Problem contd

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

$$T(n,k) \le a n \lg k.$$

Substituting,

$$\begin{aligned}
T(n,k) &\le \max_{1 \le i < n} \left\{ T\left(i, \frac{k}{2}\right) + T\left(n-i, \frac{k}{2}\right) \right\} + \Theta(n) \\
&\le \max_{1 \le i < n} \left\{ a i \lg \frac{k}{2} + a(n-i) \lg \frac{k}{2} \right\} + \Theta(n) \\
&= \max_{1 \le i < n} \left\{ an \lg \frac{k}{2} \right\} + \Theta(n) \\
&= an \lg k - an + \Theta(n) \\
&\le an \lg k \quad \text{if } a \text{ is chosen large enough.}
\end{aligned}$$

So

$$T(n,k) = O(n \log k).$$

$\square$

Problem (Longest Palindromic Subsequence)

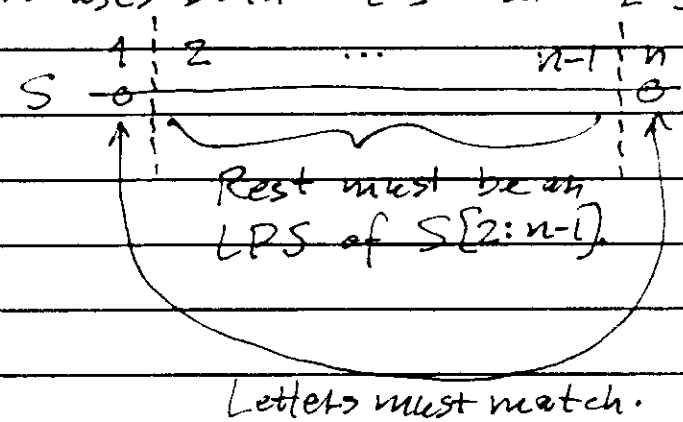
Given string $S[1:n]$, find the longest subsequence of S that is a palindrome in $O(n^2)$ time.

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

(1) (Structure)

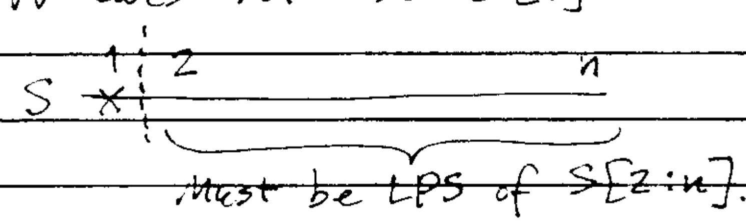
A longest palindromic subsequence (LPS) of $S[1:n]$, call it W, must end in 3 possible ways:

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



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 $S[1]$:



(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),

$$L(i,j) = \begin{cases} \max\begin{cases} L[i+1:j-1]+1, & \text{if } S[i]=S[j]; \\ L[i+1:j], \\ L[i:j-1] \end{cases}, & 1 \le i \le j \le n; \\ 0, & j = i-1, \\ & 1 \le i \le n+1. \end{cases}$$

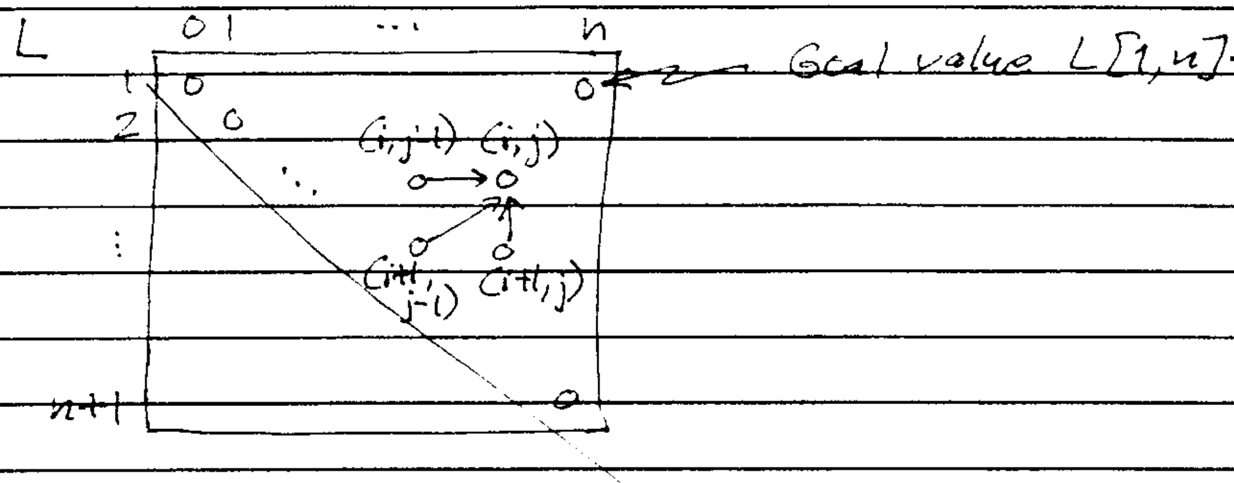
The length of an LPS for the input string S is

$$L(1,n).$$

(3) (Evaluation)

We evaluate the recurrence in a two-dimensional table $L[1:n+1,\ 0:n]$:

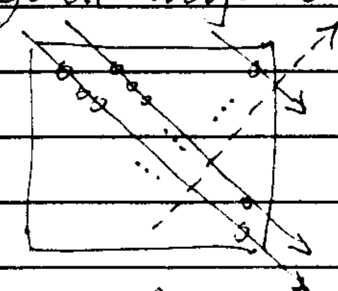
Solution, cont'd



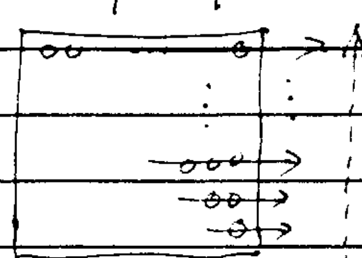
The entries $(i, i-1)$ on the main diagonal contain the boundary values.

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$(L, S, 1, n)$, which determines which of Cases (i), (ii), or (iii) gave the optimal solution:

```
procedure Recover (L, S, i, j) begin
                            . outputs an LPS of S[i:j]
                              using table L.
    if i > j then
       return
    else
       if S[i] = S[j] and L[i,j] = L[i+1, j-1] + 1
       then begin                                   . Case (i)
          output S[i]
          Recover (L, S, i+1, j-1)
          output S[j]
       end else if L[i,j] = L[i+1, j] then
          Recover (L, S, i+1, j)                    . Case (ii)
       else
          Recover (L, S, i, j-1)                    . Case (iii)
    end
```

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

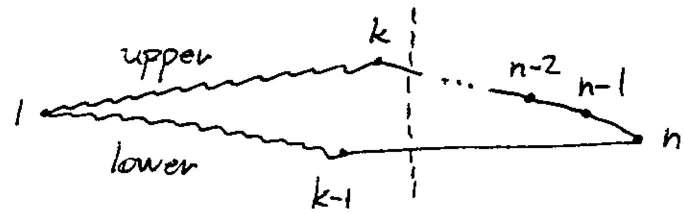
The entire algorithm from Parts (3), (4) takes total time $\Theta(n^2) + \Theta(n) = \Theta(n^2)$.

Prob. Bitonic Euclidean travelling salesman tour in $O(n^2)$ time.

Deriving a recurrence

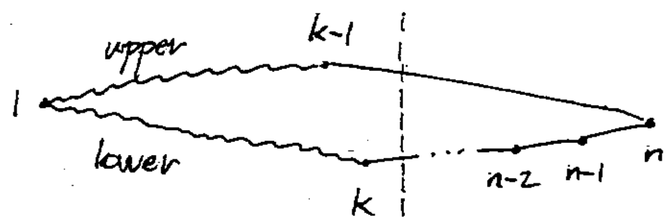
We ask, how does a solution end?

Either



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

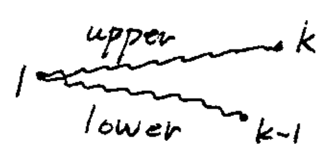
or



The last points are in lower half.

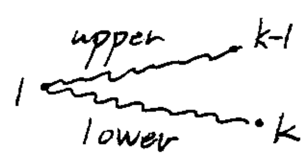
In either case, the prefix of the solution over points $1, 2, \cdots, k$ must be as short as possible. Hence let us compute

$C_U(k) \equiv$ length of a shortest tour-prefix over points $1, 2, \cdots, k$ that ends



and

$C_L(k) \equiv$ len. of shortest tour-prefix over pts $1, 2, \cdots, k$ ending

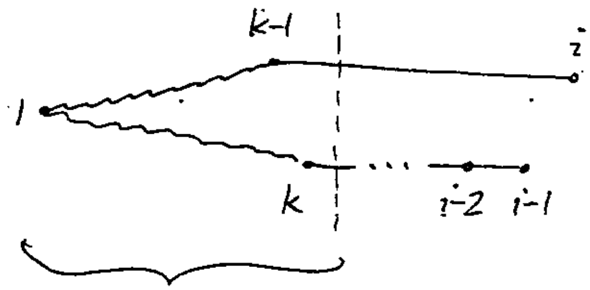


Pr. cont'd

Then the value of the solution is

$$\min_{1<k<n} \left\{ \min\{C_U(k), C_L(k)\} + \underbrace{d(k-1,n)}_{\substack{\text{Distance between} \\ \text{points } k-1 \text{ and } n.}} + \sum_{k \le j < n} d(j,j+1) \right\}$$

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



Must be a shortest tour-prefix as well.

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

Similarly for $C_L(i)$:

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

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 $\sum_{k \le j < i} d(j, j+1)$ quickly. To do this, compute

$$S(i) \equiv \sum_{1 \le j < i} d(j, j+1)$$

by

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

Then

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

The next page gives the full procedure.

Pr contd.

Array of x- and y-coordinates of points. We assume $X[1] < X[2] < \cdots < X[n]$.

```
BITONIC TOUR LENGTH (X, Y, n) begin
    S[1] := 0
    for i := 2 to n do
        S[i] := S[i-1]
           + DISTANCE(X, Y, i-1, i)
    U[2] := DISTANCE(X, Y, 1, 2)
    L[2] :=        "
    for i := 3 to n-1 do begin
        U[i] := ∞
        L[i] := ∞
        for k := 2 to i-1 do begin
            U[i] := min { U[i],
                L[k] + DISTANCE(X, Y, k-1, i) + S[i-1] - S[k] }
            L[i] := min { L[i],
                U[k] + DISTANCE(X, Y, k-1, i) + S[i-1] - S[k] }
        end
    end
    return U, L, S
end
```

Uses auxiliary arrays:
$U[2..n-1]$ for $C_U(i)$,
$L[2..n-1]$ for $C_L(i)$,
$S[1..n]$ for $\sum d(j, j+1)$.

Compute $S(i)$. $\Theta(n)$ time.

Compute $C_U(i)$, $C_L(i)$. $\Theta(n^2)$ time.

Pr cont'd

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 := BITONIC TOUR LENGTH (X, Y, n).
  Scan U[2] to U[n-2] and L[2] to L[n-1]
   to find the index k at which
        min{U[k], L[k]} + DISTANCE (X, Y, k-1, 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
```

Pr contd

RECURSIVE $\left\{\begin{matrix} \text{UPPER} \\ \text{LOWER} \end{matrix}\right\}$ PRINTTOUR (U, L, S, i) **begin**

Scan array $\left\{\begin{matrix} L[2] \text{ to } L[i-1] \\ U[2] \text{ to } U[i-1] \end{matrix}\right\}$ to find

the index k at which

$$\left\{\begin{matrix} L[k] \\ U[k] \end{matrix}\right\} + \text{DISTANCE}(k-1, i) + S[i-1] - S[k]$$

attains its minimum value.

RECURSIVE $\left\{\begin{matrix} \text{LOWER} \\ \text{UPPER} \end{matrix}\right\}$ PRINTTOUR (U, L, S, k).

print $i-k$ $\left\{\begin{matrix} \text{"L"s} \\ \text{"U"s} \end{matrix}\right\}$.

end

The time to recover the optimal bitonic tour is

$$T(n) \le \Theta(n) + T(n-1) = O(n^2).$$

<u>Exercise</u> longest increasing subsequence in $O(n \lg n)$ time

Given a sequence $A = a_1 a_2 \cdots a_n$, 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 $O(n \lg n)$ time using a balanced search tree.

Let

$L(i)$:= length of a longest strictly monotonically increasing subsequence over $a_1 \cdots a_i$ that ends with a_i.

Then the solution value is $\max_{1 \le i \le n} \{ L(i) \}$. A recurrence for $L(i)$ is

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

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 "$a_j \le a_i$" in the above.)

To recover the subsequence solution, we compute

$P(i)$:= 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.

Ex contd

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

Arrays $A[1..n]$, $L[1..n]$, $P[1..n]$.

$\Theta(n^2)$ time

```
EvaluateLIS (A, L, P, n) begin
  for i := 1 to n do begin
    L[i] := 1
    P[i] := 0
    for j := 1 to i-1 do
      if A[j] < A[i] and L[j]+1 > L[i] then begin
        L[i] := L[j]+1
        P[i] := j
      end
  end
end
```

```
PrintLIS (A, L, P, n) begin
  i := argmax_{1≤j≤n} { L[j] }
  PrintHelper (A, L, P, i)
end
```

```
PrintHelper (A, L, P, k)
  if k > 0 then begin
    PrintHelper (A, L, P, P[k])
    print A[k]
  end
```

Ex contd.

Next observe that, to evaluate $\max\limits_{\substack{1 \le j < i \\ a_j < a_i}} \{L(j)\}$ for 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 <u>largest</u>. Let the set of these (element, index, length) triples over $a_1 \cdots a_i$ 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_{j_p}, j_p, l_p), (a_{j_q}, j_q, l_q) where $a_{j_p} < a_{j_q}$, if $l_p \geq l_q$, 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 $\max\limits_{\substack{1 \le j < i \\ a_j < a_i}} \{L(j)\}$, 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 $O(\lg n)$ time.

Ex. cont'd

This gives the following algorithm.

```
EvaluateLIS (A, L, P, n) begin
    T := Tree()
    for i := 1 to n do begin
        (a, j, l) := Predecessor (A[i], T)        Returns (?, 0, 0) if no predecessor.
        L[i] := l + 1
        P[i] := j
        (a, j, l) := Find (A[i], T)               Returns (?, 0, 0) if not found.
        if L[i] > l then
            Insert (A[i], i, L[i], T)
        (a, j, l) := Successor (A[i], T)          Returns (?, ∞, ∞) if no successor.
        while L[i] ≥ l do begin
            Delete (a, T)
            (a, j, l) := Successor (A[i], T)
        end
    end
end
```

Find $\max\limits_{\substack{1 \le j < i \\ a_j < a_i}} \{L(j)\}$. → (a, j, l) := Predecessor (A[i], T)

Update the triple for (A[i], i, L[i]). → if L[i] > l then

Throw out unneeded triples. → while L[i] ≥ l do begin

The total time for all calls to Predecessor and Find is $O(n \lg n)$. The total time for all calls to Successor and Delete in the while-loop is also $O(n \lg n)$: 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 $O(n \lg n)$ time.