Problem (Finding $k$ elements closest in value to the median)

Given an unsorted array $A[1:n]$ of distinct numbers, find the $k$ elements of $A$ that are closest in value to the median of $A$, in $\Theta(n)$ time.

Solution

Do the following:

1. Find the median of $A[1:n]$; call it $x$.


4. Output the first $k$ elements of the reordered $A$.

Each step takes $O(n)$ time, so the whole algorithm takes $\Theta(n)$ time.
Problem (Finding quantiles)

Given an unsorted array of numbers $A[1:n]$ and an integer $k$ where $1 \leq k \leq n$, 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 $\left\lfloor \frac{k}{2} \right\rfloor$th $k$-quantile.
2. Find the $i$th-smallest element in the array; call it $x$. (This is the $\left\lfloor \frac{k}{2} \right\rfloor$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 $\left\lfloor \frac{n}{k} \right\rfloor$. 
procedure Quantiles (A, p, q, k) begin
    k := k - 1
    n := q - p + 1
    r := n mod k
    if \( \lfloor \frac{k}{2} \rfloor \leq r \) then
        i := \( \lfloor \frac{k}{2} \rfloor \times \frac{n-1}{k} \)
    else
        i := r \times \frac{n}{k} + \left( \lfloor \frac{k}{2} \rfloor - r \right) \times \frac{n}{k} 
    \end{if}

    \( \Theta(n) \) \{ Select the \( i^{th} \) smallest element, call it \( x \), of \( A[p..q] \) \}

    \( T(i, \frac{k}{2}) \) \{ Partition \( A[p..q] \) around element \( x \) \}

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

    \underbrace{\text{output } A[i]} \}

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

\end{end}
Problem cont'd

Analysis

We get the recurrence

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

Suppose

\[ T(n, k) \leq an \ lg k. \]

Substituting,

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

\[ \leq \max \ \left\{ ai \ lg \frac{k}{2} + a(n-i) \ lg \frac{k}{2} \right\} + \Theta(n) \]

\[ = \max \ \left\{ an \ lg \frac{k}{2} \right\} + \Theta(n) \]

\[ = an \ lg k - an + \Theta(n) \]

\[ \leq an \ lg k \text{ if } a \text{ is chosen large enough.} \]

So

\[ T(n, k) = O(n \ lg k). \]
Approach using dynamic programming

Consider how an LLS (longest increasing subsequence) ends. Suppose the last character that it uses in A is \( a_k \):

\[
A = a_1, a_2, \ldots, a_{k-1}, a_k, \ldots, a_n.
\]

Then the LLS minus character \( a_k \) must be an LLS over \( a_1, a_2, \ldots, a_{k-1} \) where every character chosen is \( \leq a_k \). Hence let us compute

\[
L(i) \equiv \text{length of an LLS over } a_1, a_2, \ldots, a_i \text{ where every char. chosen is } \leq a_{i+1}.
\]

Then the solution value is

\[
\max \left\{ L(k-1) + 1 \right\}.
\]

To compute \( L(i) \), the form of a subproblem is

\[
a, a_2, \ldots, a_{k-1}, a_k, \ldots, a_i
\]

Subprob. corrsp. Must have \( a_k \leq a_{i+1} \).

\[+ L(k-1).\]
Hence our recurrence relation is

\[ L(i) = \begin{cases} 
\max_{1 \leq k \leq i} \left\{ L(k-1) + 1 \right\}, & 1 \leq i \leq n; \\
0, & i = 0.
\end{cases} \]

This gives the following procedure.

\begin{align*}
\text{LISLength (A, n)} & \text{ begin} \\
L[0] & := 0 \\
\text{for } i := 1 \text{ to } n-1 \text{ do begin} \\
L[i] & := 0 \\
\text{for } k := 1 \text{ to } i \text{ do} \\
& \text{ if } A[k] \leq A[i+1] \text{ then} \\
& \quad L[i] := \max_{1 \leq j \leq n} \left\{ L[i], L[k-1] + 1 \right\} \\
\text{end} \\
\text{return } L \\
\text{ end}
\end{align*}

\( \Theta(n^2) \text{ time} \)
We recover an LIS as follows.

```
PRINT LIS (A, n) begin
    L := LIS LENGTH (A, n)
    Scan L[0] to L[n-1] to find the index k-1 at which the array attains its maximum value.
    Recursive PRINT LIS (A, k-1, L)
    print A[k]
end

Recursive PRINT LIS (A, i, L)
for k := i down to 1 do
    if L[i] = L[k-1] + 1 and A[k] ≤ A[i+1] then
        Recursive PRINT LIS (A, k-1, L)
        begin
            print A[k]
            break
        end
end
```

The time to recover an LIS is

\[ T(n) \leq \max \{ T(n-k) + \Theta(k) \} \]

which is \( \Theta(n) \), as can be shown by substitution.
Problem: (Edit distance with delete, insert, replace, copy, twiddle, and kill.)

(Since the kill operation is somewhat special, as it can be applied only at the end of A, while the other operations can be applied anywhere, we first solve the problem without kill, and later add it back in.) Given strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$, let

$$D(i, j) := \text{cost of best way to edit } A_i \text{ into } B_j, \text{ without using the kill operation},$$

where $A_i = a_1 a_2 \ldots a_i$ and $B_j = b_1 b_2 \ldots b_j$.

Then the solution value is

$$\min \{ D(m, n), \min_{0 \leq i < m} \{ D(i, n) + C(\text{kill}) \} \},$$

where $C(\text{kill})$ is the cost of kill-to-end-of-line.

Given a solution, we can order the operations so they occur left-to-right across strings A and B. Then the optimal solution for editing $A_i$ into $B_j$ (without kill) must end in one of the following ways:

- optimally edit $A_{i-1}$ into $B_{j-1}$ and copy, if $a_i = b_j$, or
- " " $A_{i-1}$ " $B_{j-1}$ " replace, if $a_i \neq b_j$, or
- " " $A_{i-1}$ " $B_j$ " delete, or
- " " $A_i$ " $B_{j-1}$ " insert, or
- " " $A_{i-2}$ " $B_{j-2}$ " twiddle, if $a_i a_{i-1} = b_{j-1} b_j$. 
Problem cont'd

So, ignoring boundary conditions,

\[
D(i, j) = \min \begin{cases} 
D(i-1, j-1) + C(\text{copy}) & \text{if } a_i = b_j, \\
D(i-1, j-1) + C(\text{replace}) & \text{if } a_i \neq b_j, \\
D(i-1, j) + C(\text{delete}), \\
D(i, j-1) + C(\text{insert}), \\
D(i-2, j-2) + C(\text{twiddle}) & \text{if } a_{i-1} = b_{j-1}, b_j
\end{cases}
\]

The boundary conditions are,

\[
D(i, j) = \begin{cases} 
0, & \text{if } i = 0 \text{ or } j = 0; \\
\text{above } \min \{\ldots\} \text{ w/o the twiddle term,} & \text{if } i = 1 \text{ or } j = 1.
\end{cases}
\]

Storing \(D(i, j)\) in a table \(D[0..m, 0..n]\), and filling it in by the recurrence in row-major order, gives the following algorithm:

\[
\text{Evaluate Edit Distance } (A, B, D, m, n) \begin{array}{@{}l@{}} \text{begin} \\
A[1..m] \{\text{input strings}\} \\
B[1..n] \\
D[0..m, 0..n] \{\text{table of edit distances}\}
\end{array}
\]

\[
\begin{array}{@{}l@{}} \text{for } i := 0 \text{ to } m \text{ do } D[i, 0] := 0 \\
\text{for } j := 0 \text{ to } n \text{ do } D[0, j] := 0 \\
\text{for } i := 1 \text{ to } m \text{ do} \\
\quad \text{for } j := 1 \text{ to } n \text{ do } \begin{array}{@{}l@{}} \text{begin} \\
D(i, j) := \infty \\
\text{if } A(i) = B(j) \text{ then} \\
D(i, j) := \min \{D(i, j), D(i-1, j-1) + C(\text{copy})\} \\
\text{else} \\
D(i, j) := \min \{D(i, j), D(i-1, j-1) + C(\text{replace})\} \\
\text{if } i > 1 \text{ and } j > 1 \text{ and } A(i) = B(j-1) \text{ and } A(i-1) = B(j) \text{ then} \\
D(i, j) := \min \{D(i, j), D(i-1, j-2) + C(\text{twiddle})\} \\
D(i, j) := \min \{D(i, j), D(i-1, j) + C(\text{delete}), D(i, j-1) + C(\text{insert})\}
\end{array}
\end{array}
\]

\(O(mn)\) time, space.

\(O(i)\) time.

end
We can recover the optimal edit script from the D table as follows.

**Print Edit Script** \((A, B, D, m, n)\) begin

\[ i := \min_{0 \leq i < m} \left\{ D[i, n] + C[kill] \right\} \]

if \(D[i, n] + C[kill] < D[m, n]\) then begin

Print Helper \((A, B, D, i, n)\)

print "kill"

end else Print Helper \((A, B, D, m, n)\)

end.

Print Helper \((A, B, D, i, j)\) begin

if \(i \leq 0\) or \(j \leq 0\) then return

else if \(A[i] = B[j]\) and \(D[i, j] = D[i-1, j-1] + C[copy]\) then begin

Print Helper \((A, B, D, i-1, j-1)\)

print "copy" A[i]

end else if \(A[i] \neq B[j]\) and \(D[i, j] = D[i-1, j-1] + C[replace]\) then begin

Print Helper \((A, B, D, i-1, j-1)\)

print "replace" A[i] "with" B[j]

end else if \(i > 1\) and \(j > 1\) and \(A[i] = B[j-1]\) and \(A[i-1] = B[j]\) and \(D[i, j] = D[i-2, j-2] + C[twiddle]\) then begin

Print Helper \((A, B, D, i-2, j-2)\)

print "twiddle" A[i-1] A[i]

end else if \(D[i, j] = D[i-1, j] + C[delete]\) then begin

Print Helper \((A, B, D, i-1, j)\)

print "delete" A[i]

end
else begin
  Print Helper (A, B, D, i, j-1)
  print "insert" B[j]
end

In printing, each call does $\Theta(1)$ work, and then decrements $i$, or $j$, or both. The total number of decrements is $\Theta(m+n)$, so the total time to recover the edit script is $\Theta(m+n)$. 
The 0-1 knapsack problem is, given $n$ items labeled $\{1, 2, \ldots, n\}$, each with an integer weight $w_i$ and a value $v_i$, find a subset of the items weighing at most $k$ that has maximum total value.

Suppose the optimal solution ends by putting item $i$ into the knapsack, to reach total weight $x \leq k$. 

The prefix must be a solution of greatest value that has weight $x-w_i$ and does not contain item $i$. Let

$$C(i, x) := \text{value of best knapsack over items } \{1, 2, \ldots, i\} \text{ that has weight } x.$$ 

Then the solution value is

$$\max_{0 \leq x \leq k} \{ C(n, x) \}.$$ 

In $C(i, x)$, either item $i$ is included, or not, which gives the recurrence,

$$C(i, x) = \begin{cases} 
0, & i \leq 0 \text{ or } x \leq 0; \\
\max \left\{ C(i-1, x-w_i) + v_i, \right. & i > 0 \text{ and } x > 0. \\
\left. C(i-1, x) \right\} & i \text{ is not included; } \quad \left. C(i-1, x) \right\} & i \text{ is included.}
\end{cases}$$
Storing \( C(i,x) \) in a table \( C[0..n, 0..k] \) and computing its value by the recurrence in row-major order gives the following algorithm.

\[
\text{Evaluate Knapsack} \ (W, V, C, n, k) \begin{align*}
C[0,0] &:= 0 \\
\text{for } i := 1 \text{ to } n \text{ do} &
\quad C[i,0] := 0 \\
\text{for } x := 1 \text{ to } k \text{ do} &
\quad C[0,x] := 0 \\
\quad \quad \quad \text{for } i := 1 \text{ to } n \text{ do} &
\quad \quad \quad \quad \text{for } x := 1 \text{ to } k \text{ do begin} \\
\quad \quad \quad \quad \quad C[i,x] &:= C[i-1,x] \\
\quad \quad \quad \quad \quad \text{if } x - W(i) > 0 \text{ then} &
\quad \quad \quad \quad \quad C[i,x] &:= \max \left\{ C[i-1,x], \ C[i-1,x - W(i)] + V[i] \right\} \\
\quad \quad \quad \quad \text{end} \\
\quad \quad \text{end} \\
\quad \text{end} \\
\end{align*}
\]

\( \Theta(nk) \) time, 
\( \Theta(nk) \) space.
Exercise contd.

We can recover a solution from the C table as follows.

\[
\text{Print Knapsack} \ (W, C, n, k) \ \text{begin} \\
\quad x := \arg\max \ \{ C[n, x] \} \\
\quad 0 \leq x \leq k \\
\text{Print Helper} \ (W, C, n, x) \\
\text{end} \\
\text{Print Helper} \ (W, C, i, x) \\
\quad \text{if } i \leq 0 \text{ or } x \leq 0 \text{ then} \\
\quad \quad \text{return} \\
\text{is not} \quad \rightarrow \quad \text{else if } C[i, x] = C[i-1, x] \text{ then} \\
\text{chosen} \quad \quad \quad \text{Print Helper} \ (W, C, i-1, x) \\
\text{is} \quad \rightarrow \quad \text{else begin} \\
\text{chosen} \quad \quad \quad \quad \text{print } i \quad \text{end} \\
\text{Can remove} \quad \rightarrow \quad \text{Print Helper} \ (W, C, i-1, x-W[i]) \\
\text{the tail} \quad \quad \quad \text{end} \\
\text{recursion} \\
\]

In printing the solution, each call takes \( \Theta(i) \) time, and decrements \( i \), so the total time is \( \Theta(n+k) \).