Objectives:

- learn general strategies for problems about order statistics
- learn how to find the median (or $k$-th largest) in linear average-case number of comparisons (and time)
- learn how to find the median in linear worst-case time
- learn about dynamic programming as a general algorithmic technique
- see dynamic programming applied to a variety of problems
- learn the principle of optimality and how to verify that it applies in a given situation.

Order statistics

Problem is to find the $i$-th smallest of a set of $n$ keys. This is easy to do in linear time if $i = 1$ or $i = n$ (min or max), and not too hard if $i$ is close to either end. The obvious general approach is to sort the $n$ keys and extract the one in position $i$ of the sorted order. This takes $\Theta(n \lg n)$ using any comparison-based approach, and is overkill.

Let’s look at a divide-and-conquer approach similar to Quicksort.

```plaintext
function Select(A, p, r, i)
    \( \triangleright \) returns a key $x$ such that $i - p$ keys of $A[p], \ldots, A[r]$ (not including $x$)
    \( \triangleright \) are $\leq x$ and $r - i$ are $\geq x$
    \( \triangleright \) Called initially as Select($A, 1, n, i$)
    if $r - p > 1$ then
        choose a pivot $y$ (at random)
        partition $A[p], \ldots, A[r]$ (as in Quicksort) so that:
        $A[j] \leq y$ for all $p \leq j \leq q - 1$
        $A[q] = y$
        $A[j] > y$ for all $q + 1 \leq j \leq r$
        if $i = q$ then return $y$
        else if $i < q$ then return Select($A, p, q - 1, i$)
        else return Select($A, q + 1, r, i$) endif
    else return $A[p]$ endif
end Select
```

If partitioning is done choosing a random pivot, we can assume that the pivot is equally likely to be in any one of the $n$ positions of $A[1], \ldots, A[n]$. We can do a worst average-case analysis — think about a randomized algorithm “playing against” a deterministic adversary who can do everything to make the algorithm behave badly except “load the dice”. In other words, we can simplify the analysis by assuming that the key we’re looking for is always in the larger of the two partitions (and is never exactly in position $q$). The expected number of comparisons under this assumption is given by the recurrence:

$$T(n) \leq n - 1 + \frac{1}{n} \sum_{j=1}^{n} \max(T(j - 1), T(n - j)),$$

$T(1) = 0$
Assuming further that $T(n)$ is nondecreasing, we have

\[
T(n) \leq n - 1 + \frac{1}{n} \sum_{j=1}^{n} T(\max(j - 1, n - j))
\]

\[
\leq n - 1 + \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)
\]

History elimination to get an exact solution is tricky in this case. Instead, we prove by induction that $T(n) \leq cn$ for a specific constant $c$ that will be chosen later.

The basis, $n = 1$, is trivial, since $T(1) = 0$ (we’re counting comparisons; if we were measuring overall execution time, we would have $T(1) = c_0$ and would have to ensure that $c$ was chosen to be $\geq c_0$). For the induction step, assume $T(k) \leq ck$ for all $k < n$. Our goal is to prove that $T(n) \leq cn$.

\[
T(n) \leq n - 1 + \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck
\]

\[
\leq n - 1 + 2c \left( \frac{n(n-1)}{2} - \frac{(n/2 - 1)(n/2 - 1)}{2} \right)
\]

\[
= n - 1 + c \left( \frac{3n}{4} + c' + \frac{c''}{n} \right)
\]

where $c'$ and $c''$ are constants

\[
\leq \left( 1 + \frac{3c}{4} \right) n + d
\]

where $d$ is yet another constant

\[
\leq cn
\]

as long as $c > 4$ and $n \geq 4d/(c - 4)$

We can assume that $T(n)$ is constant when $n < 4d/(c - 4)$ and redo the proof with $n = 4d/(c - 4) - 1$ as the basis.

It does not make sense to do an induction proof using the assertion “$T(n) \in O(n)$” — that’s a statement about the value of $T(n)$ for infinitely many $n$. An induction proof must make a specific assertion that holds for a single $n$ — the proof then shows the assertion to be true for all $n$. In this case, the assertion is “$T(n) \leq 4n$”.

Let’s look at what happens when we try to use a similar induction for the Quicksort recurrence — the only difference is that the summation starts at $k = 1$ instead of $\lceil n/2 \rceil$:

\[
T(n) \leq n - 1 + \frac{2}{n} \sum_{k=1}^{n-1} ck
\]

\[
\leq n - 1 + 2c \left( \frac{n(n-1)}{2} \right)
\]

\[
= n - 1 + c(n-1) = (c + 1)n - (c + 1)
\]

At this point it’s tempting to say that $T(n) \leq (c + 1)n$ and therefore clearly $O(n)$. But the inductive assertion is that $T(n) \leq cn$, where $c$ is a particular constant. As $n$ gets large enough (i.e. $n \geq c + 1$), $(c + 1)n - (c + 1) > cn$ no matter what $c$ is.

An algorithm that’s linear in the worst case

The recursive partitioning algorithm presented above has quadratic worst-case behavior just like Quicksort. In order to guarantee a linear number of comparisons in the worst case, we need to make sure a fixed fraction of the input keys are eliminated from consideration before each recursive call.
How? The solution to a much smaller selection problem guarantees a “good enough” pivot. The following algorithm, called the big five algorithm, both because of the cardinality and stature of its inventors (Blum, Floyd, Pratt, Rivest, and Tarjan) and because of the importance of the number 5 in its design, implements this idea.

```plaintext
function BigFive(S, k) is
  ▷ returns the k-th smallest key in the set of keys S
  Put the keys of S into groups of 5 and sort each group
  Let M = \{y | y is the 3rd smallest key (median) in its group \}
  Let x = BigFive(S, |S|/10 + 1)
  ▷ x is the median of M, i.e. the “median of medians”
  Use x as a pivot to divide S − \{x\} into two subsets:
    L = \{y ∈ S − \{x\} | y ≤ x\}
    U = \{y ∈ S − \{x\} | y > x\}
  if |L| ≥ k then return BigFive(L, k)
  else if |L| = k − 1 then return x
  else return BigFive(U, k − |L| − 1) endif
end BigFive
```

The picture below shows the results of the comparisons after x has been chosen.

Suppose, to keep the analysis simple, that the original set has \(n = 5(2r + 1)\) keys, for some \(r\). Regardless of where \(k\) falls in relation to the pivot, at least \(3(r + 1)\) keys can always be eliminated from further consideration, which means that at most \(7r + 2\) keys can be involved in the next recursive call. This means the set of keys is reduced to at most 70% of its original size, a process that by itself would yield a linear number of comparisons.

There is, however, another recursive call, the one that computes the median of the set \(M\). So the recurrence ends up being

\[
T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \frac{c_5 n}{5} + n - 1
\]

with \(T(n)\) constant when \(n\) is small (for example, the recurrence no longer makes sense if \(n \leq 25\) — at that point \(T(n/5)\) is going to require at most \(c_5\) comparisons). Here \(c_5\) is the number of comparisons required to find the third largest of 5 keys (because mergesort can do this in 8 comparisons, we know \(c_5 \leq 8\)).

Again, let’s prove by induction that the solution is \(O(n)\) — I mean (let’s be sure to phrase this carefully) that there exist \(c\) and \(n_0\) such that \(T(n) \leq cn\) when \(n \geq n_0\). Appropriate choices for \(c\) and \(n_0\) will become clear after working through the details of the proof.
For the basis, we only need to ensure that $c$ is chosen large enough so that $T(n) \leq cn$ for all $n < n_0$. This can easily be done since there are only a finite number of $n$ for which the inequality must be satisfied.

The induction hypothesis is that $T(i) \leq ci$ for all $i < n$ and we need to show that $T(n) \leq cn$.

Using the recurrence relation and substituting the hypothesis we have

$$T(n) \leq \frac{c}{5}n + \frac{7c}{10} + \frac{c5n}{5} + n - 1$$

$$= \left(\frac{9c}{10} + \frac{c5}{5} + 1\right) n - 1$$

$$\leq cn \quad \text{when } c \geq 2c5 + 10$$

So $c = 26$ and $n_0 = 25$ will work (the number of comparisons for mergesort when $n = 25$ is far less than $16 \cdot 25 = 400$; even insertion sort with $12 \cdot 25 = 300$ comparisons will work for the basis).

The analysis in the text differs in the details in two important ways:

1. It does not assume that $n$ is an odd multiple of 5. This introduces $\lceil n/5 \rceil$ in place of $n/5$ and $7n/10 + 6$ in place of $7n/10$ for the recursive calls. The +6 is an overestimate — it assumes that any group with less than 5 elements contributes nothing and that the middle group contributes nothing (in case the number of groups is even rather than odd); it also does not account for the fact that $n$ is smaller when there is an incomplete groups. But this does not matter.

2. It takes the total time for the partitioning combined with that for sorting groups of 5 to be an.

The only effect on our analysis would be that we would get

$$T(n) \leq \left(\frac{9c}{10} + \frac{c5}{5} + 1\right) n + 7c - 1.$$  

(the $7c$ comes from substituting $n/5 + 1$ for $\lceil n/5 \rceil$ plus the $6c$ from the $7n/10 + 6$). This is $\leq cn$ as long as $c > 2c5 + 10$ and $n \geq (70c - 10)/(c - 2c5 - 10)$. It’s clear that we can choose $c$ and $n_0$ to make this happen.

### Dynamic programming as an alternative to recursion

Recursive formula for the Fibonacci numbers:

$$f_0 = f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \text{ for } n > 1$$

and the obvious recursive algorithm for computing them:

```haskell
function FIB(n) is
    if n = 0 or n = 1 then
        return 1
    else return FIB(n-1)+FIB(n-2)
end FIB
```

Time bound analysis (using deliberate underestimate to simplify):

$$T(n) \geq 2T(n-2) + c_1, \quad T(0) = T(1) = c_0$$

$$T(n) \geq c_0 \cdot 2n/2 + c_1 \cdot (2n/2 - 1) \in \Omega(2n/2)$$

Actually, $T(n) \approx (c_0 + c_1)(1.6)^n$, the 1.6 being the Golden ratio $(1 + \sqrt{5})/2$.

But there are lots of duplicate recursive calls.
The duplication can be avoided if we “remember” that the call for a particular value has already been done and retrieve the return value (saved earlier). The text calls this technique memoizing. Here’s the algorithm that does it.

Assume \( \text{done}[i] = \text{false} \) for \( i = 0, \ldots, n \)

\[
\begin{align*}
\text{function Fib(n) is} \\
\quad &\text{if } n = 0 \text{ or } n = 1 \text{ then} \\
\quad &\phantom{\text{function Fib(n) is}}fib[n] \leftarrow 1 \\
\quad &\text{else if not done[n] then} \\
\quad &\phantom{\text{function Fib(n) is}}fib[n] \leftarrow \text{Fib}(n-1)+\text{Fib}(n-2) \\
\quad &\phantom{\text{function Fib(n) is}}\text{done}[n] \leftarrow \text{true}; \text{return fib[n]} \\
\end{align*}
\]

Instead of using augmenting a recursive algorithm by filling in a table to avoid duplicated calls, why not skip the recursion altogether.

This works as long as the table can be filled in such a way that each value depends only on previously computed values. This is the essence of dynamic programming (see Fig. 1).

\[
\begin{align*}
\text{function Fib(n) is} \\
\quad &fib[0] \leftarrow fib[1] \leftarrow 1 \\
\quad &\text{for } i \leftarrow 2 \text{ to } n \text{ do} \\
\quad &\phantom{\text{function Fib(n) is}}fib[i] \leftarrow fib[i-1] + fib[i-2] \\
\quad &\text{return fib}[n] \\
\end{align*}
\]

**Longest common subsequence**

Given two strings \( x \) and \( y \), find the longest sequence of symbols that occurs as a subsequence of both.

For example, let \( x = \text{abcbaabbcb}a \) and \( y = \text{bacabcbba} \). Then \( \text{acaba} \) is a subsequence:

\[
\begin{align*}
\text{b} &\text{a} \text{c} \text{ab} \text{a} \text{c} \text{ab} \text{b} \text{a} \\
\text{b} &\text{a} \text{c} \text{a} \text{b} \text{a} \text{c} \text{ab} \text{b} \text{a} \\
\text{b} &\text{a} \text{c} \text{a} \text{b} \text{a} \text{c} \text{ab} \text{b} \text{a} \\
\end{align*}
\]

but is it a longest subsequence? We can’t make it longer on either end, but consider
Figure 2: Computing the longest common subsequence of \textit{babcbba} and \textit{bbcaba}.

Brute force algorithm takes $\Theta(\min(m, n) \cdot 4^{\min(m, n)})$.
— try all possible subsequences of length up to $\min(m, n)$ (there are $2^j = 4^j$ ways to choose a subsequence of length $j$ in each string), and compare them (comparing two subsequences of length $j$ requires $j$ comparisons of symbols). So we have (letting $k = \min(m, n)$):

$$
\sum_{j=1}^{k} j \cdot 4^j = \frac{4(1 - 4^k)}{(1 - 4)^2} - \frac{k \cdot 4^{k+1}}{1 - 4} = \Theta(\min(m, n) \cdot 4^{\min(m, n)})
$$

A recursive formulation of a simpler problem: finding the length of the subsequence

\begin{align*}
C[i, j] & = \text{length of LCS of } x[1 \ldots i], y[1 \ldots j] \\
C[i, j] & = 0 \quad \text{if } i = 0, \quad j = 0, \ldots, n \\
C[i - 1, j - 1] + 1 & \quad \text{if } x[i] = y[j] \\
\max(C[i, j - 1], C[i - 1, j]) & \quad \text{otherwise}
\end{align*}

There are only $\Theta(mn)$ possible subproblems (the number of possible $i, j$ pairs). This suggests a solution derived from a table in which each entry depends only on those of the previous row or of the previous column in the current row. See, for example, the table in Fig. 2.

Details of the algorithm are in Fig. 3. The actual longest subsequence is easy to compute from the table (the extra table of arrows in the text is unnecessary since the direction of the arrow can be deduced from the numbers in the table).

Execution time is $\Theta(mn)$ (pretty obvious) - a constant amount of work per table entry.

The number of permutations with a specific number of inversions

\begin{align*}
P(n, k) & = \text{no. of permutations of } n \text{ items} \\
& \quad \text{with exactly } k \text{ inversions} \\
& = 0 \text{ if } k < 0 \text{ or } k > n(n - 1)/2 \\
& \sum_{i=0}^{n-1} P(n - 1, k - i) \text{ otherwise}
\end{align*}
function \text{Lcs}(x, y) \text{ is}
  \triangleright x \text{ is a string of } m \text{ characters, } y \text{ has } n \text{ characters}
  \triangleright \text{returns } z, \text{the longest common subsequence of } x \text{ and } y
  \text{for } j \leftarrow 0 \text{ to } n \text{ do } C[0, j] \leftarrow 0
  \text{for } i \leftarrow 1 \text{ to } m, j \leftarrow 1 \text{ to } n \text{ do}
    \text{if } x[i] = y[j] \text{ then } C[i, j] \leftarrow C[i - 1, j - 1] + 1
    \text{else } C[i, j] \leftarrow \max(C[i, j - 1], C[i - 1, j])
  \text{Get-Seq}(z, m, n, C[m, n]); \text{return } z
\text{end Lcs}

procedure \text{Get-Seq}(z, i, j, k) \text{ is}
  \triangleright \text{uses } C[1 \ldots i, 1 \ldots j] \text{ table to figure out } z[1 \ldots k]
  \text{while } i > 0 \text{ and } j > 0 \text{ and } k > 0 \text{ do}
    \text{if } x[i] = y[j] \text{ then}
      z[k] \leftarrow x[i]; i \leftarrow i - 1; j \leftarrow j - 1; k \leftarrow k - 1;
    \text{else if } C[i, j - 1] = k \text{ then } j \leftarrow j - 1 \text{ else } i \leftarrow i - 1
\text{end Get-Seq}

Figure 3: Algorithms for computing the length of a longest common subsequence and retrieving the actual sequence.

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{P}[n, k] & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 2 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 3 & 5 & 6 & 5 & 3 & 1 & 0 & 0 & 0 & 0 \\
5 & 1 & 4 & & & & & & & & & \\
6 & 1 & 5 & & & & & & & & & \\
\hline
\end{tabular}

Figure 4: Computing all permutations of 6 or fewer elements with specific numbers of inversions.
function Get-P(n, k) is
   if k < 0 or k > n \cdot (n - 1)/2 then
      return 0
   else return P[n, k]
end Get-P

function Permutations(n, k) is
   P[1, 0] ← 1
   for i ← 2 to n do
      for j ← 0 to min(k, i \cdot (i - 1)/2) do
         sum ← 0
         for h ← 0 to i - 1 do
            sum ← sum + Get-P(i - 1, j - h)
         P[i, j] ← sum
      end
   end
   return Get-P(n, k)
end Permutations

Figure 5: Algorithm for computing number of permutations having a particular number of inversions.

Figure 6: A balanced search tree for apple, cherry, melon, peach, and pear.

The table for computing P(6, 10) is shown in Fig. 4 (partially filled in).

The algorithm, as shown in Fig. 5, is pretty simple — Get-P is used to avoid any out-of-bounds array references.

**Optimal binary search trees**

Unlike the book, we assume here that every search is successful. We are given a set of n keys and access probabilities p_1, \ldots, p_n so that the probability that the i-th key will be accessed is p_i. The search tree has to organize the keys so that the key at the root of each subtree is greater than (with respect to the ordering of keys) all keys in its left subtree and less than all keys in its right subtree. The goal is to minimize the expected number of comparisons for a search.

For example, suppose the words apple, cherry, melon, peach, and pear had access probabilities 0.3, 0.2, 0.1, 0.2, and 0.2, respectively. We might try to construct a balanced tree — see Fig. 6.

Or we might try a “greedy” approach by putting the most frequently accessed item at the root, as in Fig. 7.

A slight improvement, but is it the best we can do? Even with such a small example, it would be daunting to enumerate all possible search trees.
Principle of optimality

To show that this problem is a candidate for dynamic programming we introduce the principle of optimality: each instance of the problem can be reduced to a collection of subinstances so that an optimal solution to the original instance implies that each subinstance must also be solved optimally. In case of the longest common subsequence problem, every subinstance asked for a longest common subsequence between a prefix of $x$ and a prefix of $y$. The longest common subsequence of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$ was either the lcs of $x_1x_2\ldots x_{i-1}$ and $y_1y_2\ldots y_{j-1}$ concatenated with $x_i$ (if $x_i = y_j$), or the lcs of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_{j-1}$, or the lcs of $x_1x_2\ldots x_{i-1}$ and $y_1y_2\ldots y_j$. In each case we use the lcs of prefixes whose combined length is shorter to deduce the lcs of the original instance.

If we have an lcs $z$ of $x$ and $y$ and we look at the prefix $z_1, \ldots, z_k$ of $z$ that is common to $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$, then $z_1, \ldots, z_k$ must be an lcs of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$ — otherwise we could replace $z_1, \ldots, z_k$ with a longer subsequence of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$ and obtain a longer subsequence for all of $x$ and $y$.

Principle of optimality applied to binary search trees

Suppose the root of the search tree has been chosen. This is a decision the algorithm must make, but we assume it when proving the principle of optimality. I claim that both the left and the right subtree must be optimal binary search trees with respect to the frequencies of the items in them (no longer probabilities because they don’t add up to 1). Otherwise, we could replace the subtree with one that had a smaller expected number of comparisons and get a smaller number for the whole tree. See Fig. 8.
The principle of optimality gives us a recursive formulation describing any instance in terms of subinstances. For binary search trees, it goes as illustrated in Fig. 9 (assuming the root \( r \) has been chosen).

\[
A(T) = \text{average access cost in tree } T
\]

\[
= \sum_{i=1}^{r-1} P(i) + \frac{r-1}{2} \cdot (1 + A(L)) + \sum_{i=r+1}^{n} P(i) (1 + A(G))
\]

Initially the left subtree begins with the first (least) key and the right subtree ends with the last (greatest). But once \( L \) and \( G \) are split into further subtrees, a general subinstance will be a tree containing the \( i \)-th through \( j \)-th key for arbitrary \( i \) and \( j \) (the keys in a subtree must be consecutive).

Since we don’t know how to choose the root, we must try out every possibility.

Let \( T_{i,j} \) be the optimal tree on keys \( i, \ldots, j \).

\[
P[i, j] = \sum_{k=i}^{j} p_k
\]

\[
A[i, j] = P[i, j] \cdot A(T_{i,j})
\]

\[
R[i, j] = \text{the root of } T_{i,j}
\]

\[
\]

\[
= P[i, j] + \min_{i \leq r \leq j} (A[i, r - 1] + A[r + 1, j])
\]

And \( A[i, j] = 0 \) when \( T_{i,j} \) is empty, i.e. when \( j < i \).

On the example, the table representing \( A[i, j] \) would be filled in as shown in Fig. 10. As usual we compute a single number, the average access cost of the optimal tree, instead of the structure. The \( R[i, j] \) array can be filled in as we go along to allow us to reconstruct the tree (these are the numbers in the second row of each table entry).

A detailed description of the algorithm is in Fig. 11.
function Opt-Bin-Tree(p[1, \ldots, n]) is
\begin{itemize}
\item $p[i]$ is the probability that the $i$-th item in a sorted sequence
\item of $n$ items is searched for. The function returns the expected
\item number of comparisons for an optimal search tree.
\item The $R$ matrix can be used to recover the actual tree.
\end{itemize}
for $i \leftarrow 1$ to $n$ do $A[i, i - 1] \leftarrow 0.0$; $P[i, i - 1] \leftarrow 0.0$
for $d \leftarrow 0$ to $n - 1$ do \hspace{1cm} $d$ represents $j - i$
\hspace{1cm} for $i \leftarrow 1$ to $n - d$ do
\hspace{2cm} $j \leftarrow i + d$
\hspace{2cm} $\text{Min} \leftarrow \min_{i \leq r \leq j}(A[i, r - 1] + A[r + 1, j])$
\hspace{2cm} $R[i, j] \leftarrow$ the $r$ that achieved the minimum
\hspace{2cm} $P[i, j] \leftarrow P[i, j - 1] + p[j]; A[i, j] \leftarrow \text{Min} + P[i, j]$
\hspace{1cm} return $A[1, n]$
end Opt-Bin-Tree

Figure 11: Algorithms for computing cost of optimal binary search tree and retrieving the tree.