Merge Sort
Outline and Reading

Divide-and-conquer paradigm (§10.1.1)

Merge-sort (§10.1)
- Algorithm
- Merging two sorted sequences
- Merge-sort tree
- Execution example
- Analysis

Generic merging and set operations (§10.2)

Summary of sorting algorithms
Divide-and-Conquer

- **Divide-and conquer** is a general algorithm design paradigm:
  - Divide: divide the input data $S$ in two disjoint subsets $S_1$ and $S_2$
  - Recur: solve the subproblems associated with $S_1$ and $S_2$
  - Conquer: combine the solutions for $S_1$ and $S_2$ into a solution for $S$
- The base case for the recursion are subproblems of size 0 or 1

- **Merge-sort** is a sorting algorithm based on the divide-and-conquer paradigm
  - Like heap-sort
    - It uses a comparator
    - It has $O(n \log n)$ running time
  - Unlike heap-sort
    - It does not use an auxiliary priority queue
    - It accesses data in a sequential manner (suitable to sort data on a disk)
Merge-Sort

Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:

- **Divide**: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
- **Recur**: recursively sort $S_1$ and $S_2$
- **Conquer**: merge $S_1$ and $S_2$ into a unique sorted sequence

**Algorithm** $\text{mergeSort}(S, C)$

- **Input** sequence $S$ with $n$ elements, comparator $C$
- **Output** sequence $S$ sorted according to $C$

if $S$.size() > 1

$(S_1, S_2) \leftarrow \text{partition}(S, n/2)$

$\text{mergeSort}(S_1, C)$

$\text{mergeSort}(S_2, C)$

$S \leftarrow \text{merge}(S_1, S_2)$
Merging Two Sorted Sequences

The conquer step of merge-sort consists of merging two sorted sequences \( A \) and \( B \) into a sorted sequence \( S \) containing the union of the elements of \( A \) and \( B \).

Merging two sorted sequences, each with \( n/2 \) elements and implemented by means of a doubly linked list, takes \( O(n) \) time.

Algorithm \textit{merge}(A, B)

\begin{itemize}
    \item \textbf{Input} sequences \( A \) and \( B \) with \( n/2 \) elements each
    \item \textbf{Output} sorted sequence of \( A \cup B \)
    \item \( S \leftarrow \) empty sequence
    \item while \( \neg A.\text{isEmpty()} \land \neg B.\text{isEmpty()} \)
        \item if \( A.\text{first()}.\text{element()} < B.\text{first()}.\text{element()} \)
            \item \( S.\text{insertLast}(A.\text{remove}(A.\text{first()})) \)
        \item else
            \item \( S.\text{insertLast}(B.\text{remove}(B.\text{first()})) \)
    \item while \( \neg A.\text{isEmpty()} \)
        \item \( S.\text{insertLast}(A.\text{remove}(A.\text{first()})) \)
    \item while \( \neg B.\text{isEmpty()} \)
        \item \( S.\text{insertLast}(B.\text{remove}(B.\text{first()})) \)
\end{itemize}

return \( S \)
Merge-Sort Tree

An execution of merge-sort is depicted by a binary tree:
- each node represents a recursive call of merge-sort and stores:
  - unsorted sequence before the execution and its partition
  - sorted sequence at the end of the execution
- the root is the initial call
- the leaves are calls on subsequences of size 0 or 1

![Merge-Sort Tree Diagram]

7 2 | 9 4 \rightarrow 2 4 7 9

7 | 2 \rightarrow 2 7
9 | 4 \rightarrow 4 9
7 \rightarrow 7
2 \rightarrow 2
9 \rightarrow 9
4 \rightarrow 4
Execution Example

Partition

7 2 9 4 | 3 8 6 1

Sets
Execution Example (cont.)

Recursive call, partition

Sets
Execution Example (cont.)

Recursive call, partition

Sets 9

| 7 2 9 4 | 3 8 6 1 |

| 7 2 | 9 4 |

| 7 | 2 |

Sets
Execution Example (cont.)

Recursive call, base case

7 2 9 4 | 3 8 6 1

7 2 | 9 4

7 | 2

7 → 7
Execution Example (cont.)

Recursive call, base case

7 2 9 4 | 3 8 6 1

Sets
Execution Example (cont.)

Merge

Sets
Execution Example (cont.)

Recursive call, ..., base case, merge

7 2 9 4 | 3 8 6 1
Execution Example (cont.)

Merge

\[
\begin{align*}
7 & 2 & 9 & 4 & | & 3 & 8 & 6 & 1 \\
\end{align*}
\]
Execution Example (cont.)

Recursive call, ..., merge, merge

7 2 9 4 | 3 8 6 1

Sets
Execution Example (cont.)

Merge

Sets 16
Analysis of Merge-Sort

- The height $h$ of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth $i$ is $O(n)$
  - we partition and merge $2^i$ sequences of size $n/2^i$
  - we make $2^{i+1}$ recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>#seqs</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$n/2^i$</td>
</tr>
</tbody>
</table>

... ... ...
## Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for large data sets (1K — 1M)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for huge data sets (&gt; 1M)</td>
</tr>
</tbody>
</table>
Sets
Storing a Set in a List

- We can implement a set with a list.
- Elements are stored sorted according to some canonical ordering.
- The space used is $O(n)$.

Nodes storing set elements in order:

- List
- Set elements
Generic Merging (§10.2)

- Generalized merge of two sorted lists $A$ and $B$
- Template method `genericMerge`
- Auxiliary methods
  - `aIsLess`
  - `bIsLess`
  - `bothEqual`
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

```
Algorithm genericMerge($A$, $B$)
    $S$ ← empty sequence
    while $¬A.isEmpty()$ ∧ $¬B.isEmpty()$
        $a ← A.first().element(); \ b ← B.first().element()$
        if $a < b$
            $aIsLess(a, S); A.remove(A.first())$
        else if $b < a$
            $bIsLess(b, S); B.remove(B.first())$
        else { $b = a$ }
            $bothEqual(a, b, S)$
            $A.remove(A.first()); B.remove(B.first())$
    while $¬A.isEmpty()$
        $aIsLess(a, S); A.remove(A.first())$
    while $¬B.isEmpty()$
        $bIsLess(b, S); B.remove(B.first())$
    return $S$
```
Using Generic Merge for Set Operations

Any of the set operations can be implemented using a generic merge.

For example:

- For **intersection**: only copy elements that are duplicated in both list.
- For **union**: copy every element from both lists except for the duplicates.

All methods run in linear time.
Set Operations

- We represent a set by the sorted sequence of its elements.
- By specializing the auxiliary methods, the generic merge algorithm can be used to perform basic set operations:
  - union
  - intersection
  - subtraction
- The running time of an operation on sets $A$ and $B$ should be at most $O(n_A + n_B)$

Set union:
- $aIsLess(a, S)$
  - $S.insertFirst(a)$
- $bIsLess(b, S)$
  - $S.insertLast(b)$
- $bothAreEqual(a, b, S)$
  - $S.insertLast(a)$

Set intersection:
- $aIsLess(a, S)$
  - \{ do nothing \}
- $bIsLess(b, S)$
  - \{ do nothing \}
- $bothAreEqual(a, b, S)$
  - $S.insertLast(a)$
Quick-Sort

Sets 24

Quick-Sort

{7, 4, 9, 6, 2} → {2, 4, 6, 7, 9}

{4, 2} → {2, 4}

{7, 9} → {7, 9}

{2} → {2}

{9} → {9}
Outline and Reading

Quick-sort (§10.3)
- Algorithm
- Partition step
- Quick-sort tree
- Execution example

Analysis of quick-sort (§10.3.1)

In-place quick-sort (§10.3.1)

Summary of sorting algorithms
Quick-Sort

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- **Divide**: pick a random element \( x \) (called **pivot**) and partition \( S \) into
  - \( L \) elements less than \( x \)
  - \( E \) elements equal to \( x \)
  - \( G \) elements greater than \( x \)
- **Recur**: sort \( L \) and \( G \)
- **Conquer**: join \( L \), \( E \) and \( G \)
Partition

We partition an input sequence as follows:
- We remove, in turn, each element $y$ from $S$ and
- We insert $y$ into $L$, $E$ or $G$, depending on the result of the comparison with the pivot $x$

Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time

Thus, the partition step of quick-sort takes $O(n)$ time

**Algorithm partition($S$, $p$)**

<table>
<thead>
<tr>
<th>Input</th>
<th>sequence $S$, position $p$ of pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>subsequences $L$, $E$, $G$ of the elements of $S$ less than, equal to, or greater than the pivot, resp.</td>
</tr>
</tbody>
</table>

$L$, $E$, $G$ ← empty sequences

$x$ ← $S$.remove($p$)

while $\neg S$.isEmpty()

$y$ ← $S$.remove($S$.first())

if $y < x$

$L$.insertLast($y$)
else if $y = x$

$E$.insertLast($y$)
else

$G$.insertLast($y$)

return $L$, $E$, $G$
Quick-Sort Tree

An execution of quick-sort is depicted by a binary tree:

- Each node represents a recursive call of quick-sort and stores:
  - Unsorted sequence before the execution and its pivot
  - Sorted sequence at the end of the execution
- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1
Execution Example

Pivot selection

7 2 9 4 3 7 6 1

Diagram showing the process of pivot selection.
Execution Example (cont.)

Partition, recursive call, pivot selection

Sets

30
Execution Example (cont.)

Partition, recursive call, base case
Execution Example (cont.)

Recursive call, ..., base case, join

Sets
Execution Example (cont.)

Recursive call, pivot selection

[Diagram showing recursive calls and pivot selection process]
Execution Example (cont.)

Partition, ..., recursive call, base case
Execution Example (cont.)

Join, join

Sets 35
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of $L$ and $G$ has size $n - 1$ and the other has size 0.
- The running time is proportional to the sum $n + (n - 1) + \ldots + 2 + 1$.
- Thus, the worst-case running time of quick-sort is $O(n^2)$.

```
<table>
<thead>
<tr>
<th>depth</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n - 1$</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Sets
Expected Running Time

Consider a recursive call of quick-sort on a sequence of size $s$

- **Good call**: the sizes of $L$ and $G$ are each less than $3s/4$
- **Bad call**: one of $L$ and $G$ has size greater than $3s/4$

A call is good with probability $1/2$

- $1/2$ of the possible pivots cause good calls:
Expected Running Time, Part 2

- **Probabilistic Fact:** The expected number of coin tosses required in order to get \( k \) heads is \( 2k \)
- For a node of depth \( i \), we expect
  - \( i/2 \) ancestors are good calls
  - The size of the input sequence for the current call is at most \((3/4)^{i/2}n\)
- Therefore, we have
  - For a node of depth \( 2\log_{4/3} n \), the expected input size is one
  - The expected height of the quick-sort tree is \( O(\log n) \)
- The amount or work done at the nodes of the same depth is \( O(n) \)
- Thus, the expected running time of quick-sort is \( O(n \log n) \)
In-Place Quick-Sort

Quick-sort can be implemented to run in-place.

In the partition step, we use replace operations to rearrange the elements of the input sequence such that:
- the elements less than the pivot have rank less than \( h \)
- the elements equal to the pivot have rank between \( h \) and \( k \)
- the elements greater than the pivot have rank greater than \( k \)

The recursive calls consider:
- elements with rank less than \( h \)
- elements with rank greater than \( k \)

Algorithm \textit{inPlaceQuickSort}(S, l, r)

\begin{itemize}
\item Input sequence \( S \), ranks \( l \) and \( r \)
\item Output sequence \( S \) with the elements of rank between \( l \) and \( r \) rearranged in increasing order
\end{itemize}

\begin{algorithmic}
\If {\( l \geq r \)}
\State \textbf{return}
\EndIf
\State \( i \leftarrow \text{a random integer between } l \text{ and } r \)
\State \( x \leftarrow S\.elemAtRank(i) \)
\State \( (h, k) \leftarrow \text{inPlacePartition}(x) \)
\State \textit{inPlaceQuickSort}(S, l, h - 1)
\State \textit{inPlaceQuickSort}(S, k + 1, r)
\end{algorithmic}
In-Place Partitioning

Perform the partition using two indices to split S into L and EYG (a similar method can split EYG into E and G).

Repeat until j and k cross:
- Scan j to the right until finding an element > x.
- Scan k to the left until finding an element < x.
- Swap elements at indices j and k.
# Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, randomized, fastest (good for large inputs)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>sequential data access, fast (good for huge inputs)</td>
</tr>
</tbody>
</table>
Bucket-Sort and Radix-Sort
Bucket-Sort (§10.5.1)

Let be $S$ be a sequence of $n$ (key, element) items with keys in the range $[0, N - 1]$

Bucket-sort uses the keys as indices into an auxiliary array $B$ of sequences (buckets)

Phase 1: Empty sequence $S$ by moving each item $(k, o)$ into its bucket $B[k]$

Phase 2: For $i = 0, ..., N - 1$, move the items of bucket $B[i]$ to the end of sequence $S$

Analysis:
- Phase 1 takes $O(n)$ time
- Phase 2 takes $O(n + N)$ time
Bucket-sort takes $O(n + N)$ time

Algorithm $bucketSort(S, N)$

Input sequence $S$ of (key, element) items with keys in the range $[0, N - 1]$

Output sequence $S$ sorted by increasing keys

$B \leftarrow$ array of $N$ empty sequences

while $\neg S.isEmpty()$
    $f \leftarrow S.first()$
    $(k, o) \leftarrow S.remove(f)$
    $B[k].insertLast((k, o))$

for $i \leftarrow 0$ to $N - 1$
    while $\neg B[i].isEmpty()$
        $f \leftarrow B[i].first()$
        $(k, o) \leftarrow B[i].remove(f)$
        $S.insertLast((k, o))$
Example

Key range $[0, 9]$
Properties and Extensions

**Key-type Property**
- The keys are used as indices into an array and cannot be arbitrary objects
- No external comparator

**Stable Sort Property**
- The relative order of any two items with the same key is preserved after the execution of the algorithm

**Extensions**
- Integer keys in the range \([a, b]\)
  - Put item \((k, o)\) into bucket \(B[k - a]\)
- String keys from a set \(D\) of possible strings, where \(D\) has constant size (e.g., names of the 50 U.S. states)
  - Sort \(D\) and compute the rank \(r(k)\) of each string \(k\) of \(D\) in the sorted sequence
  - Put item \((k, o)\) into bucket \(B[r(k)]\)
Lexicographic Order

- A \(d\)-tuple is a sequence of \(d\) keys \((k_1, k_2, \ldots, k_d)\), where key \(k_i\) is said to be the \(i\)-th dimension of the tuple.

Example:
- The Cartesian coordinates of a point in space are a 3-tuple.

The lexicographic order of two \(d\)-tuples is recursively defined as follows:

\[(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d) \iff x_1 < y_1 \lor (x_1 = y_1 \land (x_2, \ldots, x_d) < (y_2, \ldots, y_d))\]

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.
Lexicographic-Sort

Let $C_i$ be the comparator that compares two tuples by their $i$-th dimension.

Let $\text{stableSort}(S, C)$ be a stable sorting algorithm that uses comparator $C$.

Lexicographic-sort sorts a sequence of $d$-tuples in lexicographic order by executing $d$ times algorithm $\text{stableSort}$, one per dimension.

Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of $\text{stableSort}$.

Algorithm \text{lexicographicSort}(S)

Input sequence $S$ of $d$-tuples
Output sequence $S$ sorted in lexicographic order

for $i \leftarrow d$ downto 1
    $\text{stableSort}(S, C_i)$

Example:

(7,4,6) (5,1,5) (2,4,6) (2,1,4) (3,2,4)
(2,1,4) (3,2,4) (5,1,5) (7,4,6) (2,4,6)
(2,1,4) (5,1,5) (3,2,4) (7,4,6) (2,4,6)
(2,1,4) (2,4,6) (3,2,4) (5,1,5) (7,4,6)
Radix-Sort (§10.5.2)

- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension.
- Radix-sort is applicable to tuples where the keys in each dimension \( i \) are integers in the range \([0, N - 1]\).
- Radix-sort runs in time \( O(d(n + N)) \).

Algorithm \( \text{radixSort}(S, N) \)

Input sequence \( S \) of \( d \)-tuples such that \((0, ..., 0) \leq (x_1, ..., x_d)\) and \((x_1, ..., x_d) \leq (N - 1, ..., N - 1)\) for each tuple \((x_1, ..., x_d)\) in \( S \).

Output sequence \( S \) sorted in lexicographic order.

for \( i \leftarrow d \) downto 1

\( \text{bucketSort}(S, N) \)
Radix-Sort for Binary Numbers

Consider a sequence of \( n \)
\( b \)-bit integers
\[
x = x_{b-1} \ldots x_1 x_0
\]
We represent each element as a \( b \)-tuple of integers in the range \([0, 1]\) and apply radix-sort with \( N = 2 \)
This application of the radix-sort algorithm runs in \( O(bn) \) time
For example, we can sort a sequence of 32-bit integers in linear time

Algorithm \( \text{binaryRadixSort}(S) \)

\textbf{Input} sequence \( S \) of \( b \)-bit integers
\textbf{Output} sequence \( S \) sorted
replace each element \( x \) of \( S \) with the item \((0, x)\)
for \( i \leftarrow 0 \) to \( b - 1 \)
replace the key \( k \) of each item \((k, x)\) of \( S \) with bit \( x_i \) of \( x \)
\[ \text{bucketSort}(S, 2) \]
Example

Sorting a sequence of 4-bit integers

1001 0010 1001 1001 0001
0010 1110 1101 0001 0010
1101 1001 0010 1101 1101
0001 0110 0010 1101 1110
0001 0010 1001 1101 1110
Sorting Lower Bound
Comparison-Based Sorting (§10.4)

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort $n$ elements, $x_1, x_2, \ldots, x_n$.

Is $x_i < x_j$?

- yes
- no

Sets
Counting Comparisons

Let us just count comparisons then.

Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree.
Decision Tree Height

- The height of this decision tree is a lower bound on the running time.
- Every possible input permutation must lead to a separate leaf output.
  - If not, some input \(4...5...\) would have same output ordering as \(5...4...\), which would be wrong.
- Since there are \(n! = 1*2*...*n\) leaves, the height is at least \(\log(n!)

\[
\begin{align*}
&x_a < x_b? \\
&x_c < x_d? \\
&x_e < x_f? \\
&x_k < x_l? \\
&x_m < x_o? \\
&x_p < x_q? \\
\end{align*}
\]

minimum height (time)

\[
\log(n!)
\]
The Lower Bound

- Any comparison-based sorting algorithms takes at least \( \log (n!) \) time
- Therefore, any such algorithm takes time at least

\[
\log (n!) \geq \log \left( \frac{n}{2} \right)^{\frac{n}{2}} = (n/2) \log (n/2).
\]

That is, any comparison-based sorting algorithm must run in \( \Omega(n \log n) \) time.
Selection
The Selection Problem

Given an integer \( k \) and \( n \) elements \( x_1, x_2, \ldots, x_n \), taken from a total order, find the \( k \)-th smallest element in this set.

Of course, we can sort the set in \( O(n \log n) \) time and then index the \( k \)-th element.

Can we solve the selection problem faster?
Quick-Select (§10.7)

Quick-select is a randomized selection algorithm based on the prune-and-search paradigm:

- **Prune**: pick a random element $x$ (called pivot) and partition $S$ into
  - $L$ elements less than $x$
  - $E$ elements equal $x$
  - $G$ elements greater than $x$

- **Search**: depending on $k$, either answer is in $E$, or we need to recur on either $L$ or $G$

$$k \leq |L|$$

$$k > |L| + |E|$$

$$k' = k - |L| - |E|$$

$$|L| < k \leq |L| + |E|$$

(done)
### Partition

We partition an input sequence as in the quick-sort algorithm:
- We remove, in turn, each element $y$ from $S$ and
- We insert $y$ into $L$, $E$, or $G$, depending on the result of the comparison with the pivot $x$.

Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time.

Thus, the partition step of quick-select takes $O(n)$ time.

```markdown
<table>
<thead>
<tr>
<th>Algorithm</th>
<th><code>partition(S, p)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>sequence $S$, position $p$ of pivot</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>subsequences $L$, $E$, $G$ of the elements of $S$ less than, equal to, or greater than the pivot, resp.</td>
</tr>
<tr>
<td>$L$, $E$, $G$</td>
<td>← empty sequences</td>
</tr>
<tr>
<td>$x$</td>
<td>← $S$.remove($p$)</td>
</tr>
<tr>
<td><strong>while</strong> $¬S$.isEmpty()</td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>← $S$.remove($S$.first())</td>
</tr>
<tr>
<td><strong>if</strong> $y &lt; x$</td>
<td></td>
</tr>
<tr>
<td>$L$.insertLast($y$)</td>
<td></td>
</tr>
<tr>
<td><strong>else if</strong> $y = x$</td>
<td></td>
</tr>
<tr>
<td>$E$.insertLast($y$)</td>
<td></td>
</tr>
<tr>
<td><strong>else</strong> ${ y &gt; x }$</td>
<td></td>
</tr>
<tr>
<td>$G$.insertLast($y$)</td>
<td></td>
</tr>
<tr>
<td><strong>return</strong> $L$, $E$, $G$</td>
<td></td>
</tr>
</tbody>
</table>
```
Quick-Select Visualization

An execution of quick-select can be visualized by a recursion path

- Each node represents a recursive call of quick-select, and stores k and the remaining sequence

```
k=5, S=(7 4 9 3 2 6 5 1 8)
k=2, S=(7 4 9 6 5 8)
k=2, S=(7 4 6 5)
k=1, S=(7 6 5)
5
```
Expected Running Time

Consider a recursive call of quick-select on a sequence of size $s$

- **Good call**: the sizes of $L$ and $G$ are each less than $3s/4$
- **Bad call**: one of $L$ and $G$ has size greater than $3s/4$

A call is **good** with probability $1/2$

- $1/2$ of the possible pivots cause good calls:
Expected Running Time, Part 2

Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two

Probabilistic Fact #2: Expectation is a linear function:
  - \( E(X + Y) = E(X) + E(Y) \)
  - \( E(cX) = cE(X) \)

Let \( T(n) \) denote the expected running time of quick-select.

By Fact #2,

- \( T(n) \leq T(3n/4) + bn^* \) (expected # of calls before a good call)

By Fact #1,

- \( T(n) \leq T(3n/4) + 2bn \)

That is, \( T(n) \) is a geometric series:

- \( T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2n + 2b(3/4)^3n + ... \)

So \( T(n) \) is \( O(n) \).

We can solve the selection problem in \( O(n) \) expected time.
Deterministic Selection

- We can do selection in $O(n)$ worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
  - Divide $S$ into $n/5$ sets of 5 each
  - Find a median in each set
  - Recursively find the median of the “baby” medians.

See Exercise C-4.24 for details of analysis.
Master Method

Many divide-and-conquer recurrence equations have the form:

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:

1. If \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).