Intro to Computer Science II

Chapter 12
Searching and Sorting Algorithms

Review from Chapter 8
ALGORITHM FindMaximum(⟨a₀, …, aₙ₋₁⟩)

index ← 0
FOR k ← 1 TO n − 1 DO
    IF a_k > a_index THEN
        index ← k
    END IF
END FOR
RETURN index

>= finds the last one
public class MaxFinder
{
    public int findMaximum(double[] a)
    {
        int index = 0;
        for (int k = 1; k <= a.length - 1; k++)
        {
            if (a[k] > a[index])
            {
                index = k;
            }
        }
        return index;
    }
}
public int findMaximum(BankAccount[] a) {
    int index = 0;
    for (int k = 1; k <= a.length - 1; k++) {
        if (a[k].getBalance() > a[index].getBalance()) {
            index = k;
        }
    }
    return index;
}
public int findMaximum(String[] a) {
    int index = 0;
    for (int k = 1; k <= a.length - 1; k++) {
        if (a[k].compareTo(a[index]) > 0) {
            index = k;
        }
    }
    return index;
}

FROM CHAPTER 8
Lexicographically largest string
Linear search

Given the array \( \langle a_0, a_1, \ldots, a_{n-1} \rangle \) and a value of \( x \) to find, determine the index \( i \) such that \( a_i = x \) and \( 0 \leq i \leq n-1 \).

If such an index cannot be found let the index be -1.
ALGORITHM LinearSearch(⟨a₀, ..., aₙ₋₁⟩, x)

index ← 0

WHILE (index ≤ n − 1) ∧ (a_index ≠ x) DO

    index ← index + 1

END WHILE

IF index > n − 1 THEN

    RETURN −1

ELSE

    RETURN index

END IF
Linear search method

```java
public class LinearSearcher {
    public int search(double[] a, double x) {
        int index = 0;
        int n = a.length;
        while (index < n && a[index] != x) {
            index = index + 1;
        }
        if (index > n-1) {
            return -1;
        } else {
            return index;
        }
    }
}
```
Bubble sort algorithm

Sort array \( \langle a_0, a_1, \ldots, a_{n-1} \rangle \) in increasing order:

Pass 1: Process the array elements \( a_0 \) to \( a_{n-1} \) exchanging elements that are out of order:

- if \( a_0 > a_1 \) swap them, if \( a_1 > a_2 \) swap them, \ldots, if \( a_{n-2} > a_{n-1} \), swap them.
- At end of this pass the largest array element will be in the last position, its correct position

Pass 2: process \( a_0 \) to \( a_{n-2} \) in the same way

Pass \( n-1 \): process \( a_0 \) to \( a_1 \)
Pseudo-code loop for passes

(1) Compare elements at positions \((0,1), (1,2), (2,3), \ldots, (n-2, n-1)\)
(2) Compare elements at positions \((0,1), (1,2), (2,3), \ldots, (n-3, n-2)\)
(3) Compare elements at positions \((0,1), (1,2), (2,3), \ldots, (n-4, n-3)\)

\ldots

\((p)\) Compare elements at positions \((0,1), (1,2), \ldots, (n-1-p, n-p)\)

\ldots

\((n-1)\) Compare elements at positions \((0,1)\)

**FOR** \(p \leftarrow 1\) **TO** \(n-1\) **DO**

Compare pairs at positions \((0,1), (1,2), \ldots, (n-1-p, n-p)\)
swapping elements that are out of order

**END FOR**
## Bubble sort example

<table>
<thead>
<tr>
<th>Pass</th>
<th>$a_0$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$a_4$</th>
<th>$a_5$</th>
<th>$a_6$</th>
<th>$a_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of pass 1</td>
<td>44</td>
<td>55</td>
<td>12</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
</tr>
<tr>
<td>Start of pass 2</td>
<td>44</td>
<td>12</td>
<td>42</td>
<td>55</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>94</td>
</tr>
<tr>
<td>Start of pass 3</td>
<td>12</td>
<td>42</td>
<td>44</td>
<td>18</td>
<td>6</td>
<td>55</td>
<td>67</td>
<td>94</td>
</tr>
<tr>
<td>Start of pass 4</td>
<td>42</td>
<td>12</td>
<td>18</td>
<td>6</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
</tr>
<tr>
<td>Start of pass 5</td>
<td>12</td>
<td>18</td>
<td>6</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
</tr>
<tr>
<td>Start of pass 6</td>
<td>12</td>
<td>6</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
</tr>
<tr>
<td>Start of pass 7</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
</tr>
<tr>
<td>End of pass 7</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
</tr>
</tbody>
</table>
ALGORITHM bubbleSort(⟨a₀, ..., aₙ₋₁⟩)
FOR p ← 1 TO n − 1 DO
    FOR j ← 0 TO n − 1 − p DO
        IF aⱼ > aⱼ₊₁ THEN
            swap(aⱼ, aⱼ₊₁)
        END IF
    END FOR
END FOR
END FOR
public class BubbleSorter
{
    public double[] bubbleSort(double[] a)
    {
        int n = a.length;
        for (int p = 1; p <= n - 1; p++)
        {
            for (int j = 0; j <= n-1-p; j++)
            {
                if (a[j] > a[j + 1])
                {
                    double temp = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = temp;
                }
            }
        }
        return a; // so we can test in BlueJ
    }
}
public void bubbleSort(String[] a)
{
    int n = a.length;
    for (int p = 1; p <= n - 1; p++)
    {
        for (int j = 0; j <= n-1-p; j++)
        {
            if (a[j].compareTo(a[j+1]) > 0)
            {
                String temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}

Note that we are not swapping String objects here. We are swapping references to String objects.
Chapter 12
Searching and Sorting Algorithms

With an introduction to algorithm efficiency
Min and Max algorithms

We consider minimum and maximum algorithms using subarrays. This is more general.

A subarray of an array \( \langle a_0, a_1, \ldots, a_{n-1} \rangle \) is the sequence \( \langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \) with \( 0 \leq \text{start} \leq \text{end} \leq n-1 \).

We consider only the min algorithms since the max algorithms are easily obtained by reversing inequalities in comparisons.
**Problem:** Given the array \( \langle a_0, a_1, \ldots, a_{n-1} \rangle \) and the index values \( start \) and \( end \) defining a subarray, determine an index \( i \) such that \( start \leq i \leq end \) and \( a_i \leq a_k \) for all \( k \) such that \( start \leq k \leq end \).

The index \( i \) is not unique: the minimum value can occur several times.

Do we return the minimum value or the index at which it occurs?
**Minimum algorithms (2)**

**ALGORITHM** \( \text{FindMinimum}(\langle a_0, \ldots, a_{n-1} \rangle, \text{start}, \text{end}) \)

\[
\text{index} \leftarrow \text{start} \\
\text{FOR } k \leftarrow \text{start} + 1 \text{ TO } \text{end} \text{ DO} \\
\quad \text{IF } a_k < a_{\text{index}} \text{ THEN} \\
\quad \quad \text{index} \leftarrow k \\
\quad \text{END IF} \\
\text{END FOR} \\
\text{RETURN } \text{index}
\]
Return index of first occurrence of minimum value

```c
int findMinimum(int[] a, int start, int end) {
    int index = start;
    for (int k = start + 1; k <= end; k++) {
        if (a[k] < a[index]) {
            index = k;
        }
    }
    return index;
}
```
Minimum algorithms (4)

Return index of last occurrence of minimum value

```c
int findMinimum(int[] a, int start, int end) {
    int index = start;
    for (int k = start + 1; k <= end; k++) {
        if (a[k] <= a[index]) {
            index = k;
        }
    }
    return index;
}
```

Only difference
Return minimum value itself instead of index

```c
int findMinimum(int[] a, int start, int end)
{
    int min = a[start];
    for (int k = start + 1; k <= end; k++)
    {
        if ( a[k] <= min )
            index = a[k];
    }
    return min;
}
```
Minimum algorithms (6)

Example using `findMinimum` (index version)

```java
int[] scores = {56,32,27,98,27,57,68,28,45,65};
int pos = findMinimum(score, 0, score.length - 1);
System.out.println("Position of minimum is \" + pos);
System.out.println("Minimum value is \" + score[pos]);
```
Can easily modify algorithm for other types of arrays

```java
int findMinimumBalance(BankAccount[] a, int start, int end)
{
    int index = start;
    for (int k = start + 1; k <= end; k++)
    {
        if (a[k].getBalance() < a[index].getBalance())
            index = k;
    }
    return index;
}
```

returns index of BankAccount with minimum balance
Let $T(n)$ be the time to find the minimum in an $n$ element subarray ($n = \text{end} - \text{start} + 1$).

For findMinimum the running time increases linearly with $n$: $T(n) = an + b$ for some constants $a$ and $b$.

The constants $a$ and $b$ depend on the particular computer system running an implementation of the algorithm.
Running time (2)

★ We need a way to measure the running time of an algorithm in a machine independent way.

★ We need to eliminate the constants $a$ and $b$ that are machine independent.

★ We also need to find a measure of time that doesn't use physical units such as seconds.
For findMinimum we can choose the number of times the for-loop is executed (number of times the if statement is executed). In this case \( T(n) = n - 1 \)

We say that all algorithms whose running time is \( T(n) = an + b \) are \( O(n) \) algorithms (Big O notation) and we write \( T(n) = O(n) \)
The best, average, and worst case behaviour of an algorithm can also be used to describe the running time.

For finding the minimum they are all $O(n)$. 
Mathematical Definition

Given two functions \( f \) and \( g \) we say that \( f(n) = O(g(n)) \) if there are constants \( c > 0 \) and \( N > 0 \) such that \( 0 \leq f(n) \leq cg(n) \) for all \( n > N \).

There are other measures of the rate of growth of a function but we will not consider them here.

Example: The functions \( an + b, n, n / 2 \) are all \( O(n) \) and the functions \( an^2 + bn + c \) are all \( O(n^2) \).
We consider three searching algorithms

- Linear search (version done in Chapter 8)
- Recursive binary search
- Non-recursive binary search

We develop algorithms for integer arrays which can easily be modified for other array types.

We also use array slices
Linear search algorithm (1)

Problem: given the array $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ and the index values $\text{start}$ and $\text{end}$ defining a subarray $\langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle$ and a value $x$ to find, determine and index $i$ such that $a_i = x$ and $\text{start} \leq i \leq \text{end}$. If such an index cannot be found let the index be -1,
ALGORITHM LinearSearch(⟨a₀,…,aₙ₋₁⟩, x, start, end)

index ← start
WHILE index ≤ end ∧ a_index ≠ x DO
    index ← index + 1
END WHILE
IF index > end THEN
    RETURN −1
ELSE
    RETURN index
END IF

so that we don't go off the end of the slice

harder to understand than the FOR loop version with a RETURN statement
**Linear search algorithm (2b)**

**ALGORITHM** LinearSearch($\langle a_0, \ldots, a_{n-1} \rangle, x, start, end$)

$index \leftarrow start$

**WHILE** $index \leq end$ **DO**

   **IF** $a_{index} = x$ **THEN**

       **RETURN** $index$

   **END IF**

   $index \leftarrow index + 1$

**END WHILE**

**RETURN** $-1$
ALGORITHM $LinearSearch(\langle a_0, \ldots, a_{n-1} \rangle, x, \text{start}, \text{end})$

FOR $index \leftarrow \text{start}$ TO $\text{end}$ DO
  IF $a_{\text{index}} = x$ THEN
    RETURN $\text{index}$
  END IF
END FOR
RETURN $-1$

another version using a RETURN statement in a FOR loop
Linear search algorithm (3)

- The best case (\( x \) is the first array element)
  \[ T(n) = O(1) \] (doesn't depend on \( n \))

- The worst case (\( x \) is the last array element)
  \[ T(n) = O(n) \]

- The average case can be shown to have
  \[ T(n) = O(n) \]

- Here \( n = end - start + 1 \)
For testing we can put Java implementations of our search algorithms, as we develop them, in a class as static methods

```java
public class IntArraySearch {
    public static int linearSearch(int[] a, int x, int start, int end) {
        int i = start;
        while (i <= end && a[i] != x) {
            i++;
        }
        if (i <= end) return i;
        else return -1;
    }
}
```
A for loop can be used to do linear search. This version may be easier to understand.

```java
public static int linearSearch(int[] a, int x, int start, int end) {
    for (int k = start; I <= end; i++)
    {
        if (a[i] == x) return i;
    }
    return -1;
}
```
package chapter12.searching;
import java.util.Scanner;

public class IntArraySearchTester
{
    public void doTest()
    {
        Scanner input = new Scanner(System.in);

        // continued next slide
    }
}
// First read the array size

System.out.print("Enter array size: ");
int size = input.nextInt();
int[] testArray = new int[size];

// Now read the array elements

for (int k = 0; k < testArray.length; k++)
{
    System.out.print("Enter element "+k+": ");
    testArray[k] = input.nextInt();
    input.nextLine();
}
// Read element to find

System.out.print("Enter element to find: ");
int x = input.nextInt();
input.nextLine();

// Read the start and end indices

System.out.print("Enter start index: ");
int start = input.nextInt();
input.nextLine();

System.out.print("Enter end index: ");
int end = input.nextInt();
input.nextLine();
// Search array and display result

int pos;
pos = IntArraySearch.linearSearch(
    testArray, x, start, end);
displayResult(pos, x);

pos = IntArraySearch.rBinarySearch(
    testArray, x, start, end);
displayResult(pos, x);

pos = IntArraySearch.nrBinarySearch(
    testArray, x, start, end);
displayResult(pos, x);

} // end doTest
public void displayResult(int pos, int x) {
    if (pos > 0)
        System.out.println("Element " + x + " was not found");
    else
        System.out.println("Element " + x + " was found at position " + pos);
}

public static void main(String[] args) {
    new IntArraySearchTester().doTest();
}
} // end of class
Recursive binary search (1)

- Array must be sorted in some order
- Sometimes called the phone book algorithm
- Open book in middle
- Decide if you should search the left or right half and then search this half
- **Recursive description:** to search an array search one of the two subarrays obtained by dividing the array in half
Recursive binary search (2)

Start with an array slice \( \langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \)
where \( a_{\text{start}} \leq a_{\text{start}+1} \leq \ldots \leq a_{\text{end}} \)

We want to determine if \( x \) is found in this subarray

To begin each bisection step we choose the middle subarray element \( a_{\text{mid}} \) where \( \text{mid} = (\text{start} + \text{end})/2 \)
There are two base cases:

- $\text{start} > \text{end}$ (subarray empty, $x$ not found)
- $x = a_{mid}$ ($x$ has been found)

There are two recursive cases:

- $x < a_{mid}$ (search left subarray $\langle a_{\text{start}}, \ldots, a_{mid-1} \rangle$)
- $x > a_{mid}$ (search right subarray $\langle a_{mid+1}, \ldots, a_{\text{end}} \rangle$)
Example for integer array

Find 56 in <3,5,7,11,21,47,56,63,84,89>

<table>
<thead>
<tr>
<th>Step</th>
<th>Left subarray</th>
<th>Middle</th>
<th>Right subarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;3,5,7,11&gt;</td>
<td>21</td>
<td>&lt;47,56,63,84,89&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;47,56&gt;</td>
<td>63</td>
<td>&lt;84,89&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;&gt;</td>
<td>47</td>
<td>&lt;56&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;&gt;</td>
<td>56</td>
<td>&lt;&gt;</td>
</tr>
</tbody>
</table>
Running time estimate (1)

Running time is proportional to the number of bisections that are made and this is proportional to the number of comparisons of \( x \) with \( a_{mid} \).

Example: 10 element array (4 bisections)
\[
10 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1
\]

Example: 1000 element array (10 bisections)
\[
1000 \rightarrow 500 \rightarrow 250 \rightarrow 125 \rightarrow 63 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1
\]
## Comparison of linear and binary search

<table>
<thead>
<tr>
<th>$n$</th>
<th>Linear</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^6$</td>
<td>20</td>
</tr>
<tr>
<td>$10^9$</td>
<td>$10^9$</td>
<td>30</td>
</tr>
<tr>
<td>$10^{40}$</td>
<td>$10^{40}$</td>
<td>133</td>
</tr>
</tbody>
</table>

Binary search is much more efficient than linear search

\[
\left\lceil \log_2 n \right\rceil = \left\lceil \frac{\ln n}{\ln 2} \right\rceil
\]
Binary Search Algorithm

ALGORITHM binarySearch(⟨a_{start}, …, a_{end}⟩, x)
IF start > end THEN RETURN −1
ELSE
    mid ← (start + end) / 2
    IF x = a_{mid} THEN
        RETURN mid
    ELSE IF x < a_{mid} THEN
        RETURN binarySearch(⟨a_{start}, …, a_{mid-1}⟩, x)
    ELSE
        RETURN binarySearch(⟨a_{mid+1}, …, a_{end}⟩, x)
    END IF
END IF
public static int rBinarySearch(int[] a, int x, 
    int start, int end)
{
    if (start <= end)
    {
        int mid = (start + end) / 2;
        if (x < a[mid]) // search left half
            return rBinarySearch(a, x, start, mid-1);
        else if (x > a[mid]) // search right half
            return rBinarySearch(a, x, mid+1, end);
        else // x found and x = a[mid]
            return mid;
    }
    return -1;
}
Put the Java implementation in the `IntArraySearch` class along with the linear version.

```java
public class IntArraySearch {
    public static int linearSearch(int[] a, int x, int start, int end) {
        ...
    }

    public static int rBinarySearch(int[] a, int x, int start, int end) {
        ...
    }
}
```
Non-recursive binary search

- Keep track of two indices $low$ and $high$ that define the subarray $\langle a_{low}, \ldots, a_{high} \rangle$ to search.

- Initially let $low = start$ and $high = end$.

- If left half is chosen adjust $high$ to $mid - 1$.

- If right half is chosen adjust $low$ to $mid + 1$.

- As the bisection process proceeds, $low$ increases and $high$ decreases. When $low > high$ the algorithm terminates.
Example

Find 56 in the array \(<1,3,5,7,11,21,47,56,63,84,89>\)
Initially low = 0 and high = 10

<table>
<thead>
<tr>
<th>1 3 5 7 11 21 47 56 63 84 89</th>
</tr>
</thead>
<tbody>
<tr>
<td>low=0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>high=10</td>
</tr>
<tr>
<td>low=6</td>
</tr>
<tr>
<td>high=10</td>
</tr>
<tr>
<td>low=6 high=7</td>
</tr>
<tr>
<td>low=high=7</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
|                               |<--- found it
public static int nrBinarySearch(int[] a, int x, int start, int end) {
    int low = start;
    int high = end;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (x < a[mid]) { // left half
            high = mid - 1;
        } else if (x > a[mid]) { // right half
            low = mid + 1;
        } else {
            return mid;
        }
    }
    return -1; // x not found
}
IntArraySearch class

Put the Java implementation in the `IntArraySearch` class along with the linear and recursive versions

```java
public class IntArraySearch {
  public static int linearSearch(int[] a, int x, int start, int end) {
      {...}
      
  public static int rBinarySearch(int[] a, int x, int start, int end) {
      {...}
      
  public static int nrBinarySearch(int[] a, int x, int start, int end) {
      {...}
}
Binary search is $O(\log n)$ (1)

Assume that $n$ is a power of 2: $n = 2^m$

We can search a subarray of $n$ elements in $T(n)$ time

From the recursive version of the algorithm this should be $1 + T(n/2)$ since we do one comparison to determine which half to search and $T(n/2)$ comparisons to search this half. Therefore we obtain the recurrence relation

$T(n) = T(n/2) + 1$
Binary search is $O(\log n)$ (2)

\[ T(n) = T(n/2) + 1, \text{ where } T(1) = 1 \]
\[ T(n) = T(n/2^2) + 1 + 1 = T(n/2^2) + 2 \]
\[ T(n) = T(n/2^3) + 1 + 2 = T(n/2^3) + 3 \]
\[ \ldots \]
\[ T(n) = T(n/2^m) + m \]
\[ T(n) = T(1) + m, \text{ since } n = 2^m \]
\[ T(n) = 1 + m \]
\[ T(n) = 1 + \log_2 n \]
\[ T(n) = O(\log_2 n) \]
Sorting Algorithms
Sorting is a very common operation in data processing. There are many algorithms.

We consider four algorithms for integer arrays:

- selection sort (average case $O(n^2)$) inefficient
- insertion sort (average case $O(n^2)$) inefficient
- bubble sort (average case $O(n^2)$) inefficient
- quicksort (average case $O(n \log n)$) efficient
- mergesort (average case $O(n \log n)$) efficient
Each sorting method starts with an array slice \( \langle a_{start}, \ldots, a_{end} \rangle \) where \( 0 \leq start \leq n - 1 \).

The slice is sorted in increasing order so that when the algorithm finishes we have

\[
    a_{start} \leq a_{start+1} \leq a_{start+2} \leq \cdots \leq a_{end}
\]

The algorithms are easily modified to sort other array types in another order such as decreasing order.
Selection sort process (1)

Find smallest element in \( \langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \) and swap (exchange) it with element at position \( \text{start} \) so \( a_{\text{start}} \) is now the smallest element.

Find smallest element in \( \langle a_{\text{start}+1}, \ldots, a_{\text{end}} \rangle \) and swap (exchange) it with element at position \( \text{start} + 1 \) so \( a_{\text{start}}, a_{\text{start}+1} \) are now the smallest two elements.

Repeat until \( \langle a_{\text{end}−1}, a_{\text{end}} \rangle \) is the last subarray
Selection sort process (2)

sorted part of array

\[
\langle \rangle \\
\langle a_{\text{start}}^{(1)} \rangle \\
\langle a_{\text{start}}, a_{\text{start}+1}^{(2)} \rangle \\
\vdots \\
\langle a_{\text{start}}^{(n-2)}, \ldots, a_{\text{end} - 2}^{(n-2)} \rangle \\
\langle a_{\text{start}}, \ldots, a_{\text{end} - 1}^{(n-1)} \rangle
\]

unsorted part of array

\[
\langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \\
\langle a_{\text{start}+1}, \ldots, a_{\text{end}}^{(1)} \rangle \\
\langle a_{\text{start}+2}, \ldots, a_{\text{end}}^{(2)} \rangle \\
\vdots \\
\langle a_{\text{end} - 1}, a_{\text{end}}^{(n-2)} \rangle \\
\langle a_{\text{end}}^{(n-1)} \rangle
\]

sorted part of array

unsorted part of array
### Selection sort example

Sort the array \(<44, 55, 12, 42, 94, 18, 6, 67>\)

<table>
<thead>
<tr>
<th>step</th>
<th>(a_0)</th>
<th>(a_1)</th>
<th>(a_2)</th>
<th>(a_3)</th>
<th>(a_4)</th>
<th>(a_5)</th>
<th>(a_6)</th>
<th>(a_7)</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44</td>
<td>55</td>
<td>12</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>swap 6 with 44</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>55</td>
<td>12</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>44</td>
<td>67</td>
<td>swap 12 with 55</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>12</td>
<td>55</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>44</td>
<td>67</td>
<td>swap 18 with 55</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>94</td>
<td>55</td>
<td>44</td>
<td>67</td>
<td>swap 42 with itself</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>94</td>
<td>55</td>
<td>44</td>
<td>67</td>
<td>swap 44 with 94</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>94</td>
<td>67</td>
<td>swap 55 with itself</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>94</td>
<td>67</td>
<td>swap 67 with 94</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
<td>done</td>
</tr>
</tbody>
</table>
Top level pseudo-code

Top level pseudo-code algorithm for selection sort is

FOR $i \leftarrow \text{start}$ TO $\text{end} - 1$ DO
    Find index $k$ of smallest element in $\langle a_i, \ldots, a_{\text{end}} \rangle$
    Exchange (swap) elements at positions $i$ and $k$
END FOR
Selection sort algorithm

ALGORITHM selectionSort(⟨a\(_\text{start}\), ..., a\(_\text{end}\), start, end⟩)

FOR i ← start TO end − 1 DO
  k ← i
  FOR j ← i + 1 TO end DO
    IF a\(_j\) < a\(_k\) THEN
      k ← j
    END IF
  END FOR
  temp ← a\(_k\)
  a\(_k\) ← a\(_i\)
  a\(_i\) ← temp
END FOR

This is just the find minimum algorithm applied to the slice beginning at index \(i\) and ending at index \(end\).
Java selectionSort method

```java
public static void selectionSort(int[] a, int start, int end) {
    for (int i = start; i < end; i++) {
        int k = i;
        for (int j = i + 1; j <= end; j++) {
            if (a[j] < a[k]) k = j;
        }
        int temp = a[k];  // swap
        a[k] = a[i];
        a[i] = temp;
    }
}
```
Put the Java implementation in the `IntArraySort` class

```java
public class IntArraySort {
    public static void selectionSort(int[] a, int start, int end) {
        ... }
}
```

This method is tested in the `SortTester` class
### Selection sort running time (1)

<table>
<thead>
<tr>
<th>outer loop index</th>
<th>inner loop index</th>
<th>inner loop executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start + 1, …, end</td>
<td>end – start = n − 1</td>
</tr>
<tr>
<td>start + 1</td>
<td>start + 2, …, end</td>
<td>end – start – 1 = n − 2</td>
</tr>
<tr>
<td>start + 2</td>
<td>start + 3, …, end</td>
<td>end – start – 2 = n − 3</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>end – 1</td>
<td>end, …, end</td>
<td>…</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 1</td>
</tr>
</tbody>
</table>
Selection sort running time (2)

\[
T(n) = 1 + 2 + 3 + \ldots + n-1
\]
\[
T(n) = n-1 + n-2 + n-3 + \ldots + 1
\]

\[
2T(n) = n + n + n + \ldots + n
\]

Therefore

\[
T(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)
\]
Insertion sort process (1)

- Start with the array slice $\langle a_{start}, \ldots, a_{end} \rangle$

- Divide it into two parts: a left part $L$ which is already sorted and a right part $R$ which is unsorted.

- Initially we let $L = \langle a_{start} \rangle$ and $R = \langle a_{start+1}, \ldots, a_{end} \rangle$
Starting with

\[
L = \langle \alpha_{\text{start}} \rangle \quad \text{and} \quad R = \langle \alpha_{\text{start}+1}, \ldots, \alpha_{\text{end}} \rangle
\]

take the first element of \( R \) and move it to its proper place in \( L \) to give

\[
L = \langle \alpha_{\text{start}}^{(1)}, \alpha_{\text{start}+1}^{(1)} \rangle \quad \text{and} \quad R = \langle \alpha_{\text{start}+2}, \ldots, \alpha_{\text{end}} \rangle
\]

where

\[
\alpha_{\text{start}}^{(1)} \leq \alpha_{\text{start}+1}^{(1)}
\]
After several steps we obtain

\[ L = \langle a_{\text{start}}^{(i-\text{start}-1)}, \ldots, a_{i-1}^{(i-\text{start}-1)} \rangle \]

and

\[ R = \langle a_i, \ldots, a_{\text{end}} \rangle \]

where

\[ a_{\text{start}}^{(i-\text{start}-1)} \leq \ldots \leq a_{i-1}^{(i-\text{start}-1)} \]
Insertion sort process (4)

For

\[ L = \langle a_{\text{start}}^{(i-\text{start}-1)}, \ldots, a_{i-1}^{(i-\text{start}-1)} \rangle \quad R = \langle a_i, \ldots, a_{\text{end}} \rangle \]

we need to move \( a_i \) to its correct position in \( L \)

This is done, beginning at the right end of \( L \), by moving the elements of \( L \) to the right one place until the correct "hole" is found for \( a_i \)

This is called a "move and drop" technique
## Insertion sort example

<table>
<thead>
<tr>
<th>step</th>
<th>(a_0)</th>
<th>(a_1)</th>
<th>(a_2)</th>
<th>(a_3)</th>
<th>(a_4)</th>
<th>(a_5)</th>
<th>(a_6)</th>
<th>(a_7)</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44</td>
<td>55</td>
<td>12</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>put 55 in (L)</td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td>55</td>
<td>12</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>put 12 in (L)</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>44</td>
<td>55</td>
<td>42</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>put 42 in (L)</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>put 94 in (L)</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>94</td>
<td>18</td>
<td>6</td>
<td>67</td>
<td>put 18 in (L)</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>94</td>
<td>6</td>
<td>67</td>
<td>put 6 in (L)</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>94</td>
<td>67</td>
<td>put 67 in (L)</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>42</td>
<td>44</td>
<td>55</td>
<td>67</td>
<td>94</td>
<td>done</td>
</tr>
</tbody>
</table>

The bold numbers are in \(L\) and the other numbers are in \(R\)
Insertion sort algorithm

ALGORITHM insertionSort(⟨a₀,…,aₙ₋₁⟩, start, end)

FOR i ← start + 1 TO end DO
    x ← aᵢ \[
    \text{Left element of unsorted part}
    \]
    j ← i − 1 \[
    \text{Right index of sorted part}
    \]
    WHILE j ≥ start ∧ x < aᵢ DO
        aⱼ₊₁ ← aⱼ
        j ← j − 1
    END WHILE
    aⱼ₊₁ ← x
END FOR
Java insertionSort method

```java
public static void insertionSort(int[] a, int start, int end) {
    // sorted part is a[start], ..., a[i-1]
    // unsorted part is a[i] to a[end]

    for (int i = start + 1; i <= end; i++) {
        int x = a[i]; // left element of unsorted part
        int j = i - 1; // right index of sorted part
        // move element right to find position for x
        while ( (j >= start) && (x < a[j]) ) {
            a[j+1] = a[j]; // move a[j] to right
            j--;
        }
        a[j+1] = x; // drop x into hole found
    }
}
```
Put the Java implementation in the IntArraySort class

```java
public class IntArraySort {
    public static void selectionSort(int[] a, int start, int end)
    {
        ... }

    public static void insertionSort(int[] a, int start, int end)
    {
        ... }
}
```

This method is tested in the SortTester class
Insertion sort running time (1)

<table>
<thead>
<tr>
<th>outer loop index</th>
<th>inner loop index</th>
<th>inner loop executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$start + 1$</td>
<td>$start, \ldots, start$</td>
<td>1</td>
</tr>
<tr>
<td>$start + 2$</td>
<td>$start, \ldots, start + 1$</td>
<td>2</td>
</tr>
<tr>
<td>$start + 3$</td>
<td>$start, \ldots, start + 2$</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$end$</td>
<td>$start, \ldots, end - 1$</td>
<td>$end - start$</td>
</tr>
</tbody>
</table>
The worst case behaviour (while loop executes the maximum number of times when array is sorted in reverse order) is:

\[ T(n) = 1 + 2 + \cdots + \text{end} - \text{start} = \frac{n(n-1)}{2} = O(n^2) \]

where \( n = \text{end} - \text{start} + 1 \) is the array size.

Can be shown that the average time is also \( O(n^2) \)
Comparing running times

Both selection and insertion sort are $O(n^2)$

By timing these algorithms on a real computer system it can be shown that insertion sort is about twice as fast as selection sort.

On one particular system we got the estimates

$$T(n) = 2.1 \times 10^{-8} n^2$$ for selection sort
$$T(n) = 1.2 \times 10^{-8} n^2$$ for insertion sort
package chapter12.sorting;
import java.util.Scanner;
import java.util.Random;

public class QuadraticSortTimer {
    public void doTest() {
        Scanner input = new Scanner(System.in);
        String sortType = ""

        System.out.println("Selection sort: 1");
        System.out.println("Insertion sort: 2");
        System.out.println("Enter 1 or 2");

        int choice = input.nextInt(); input.nextLine();
System.out.print("Enter size of array: ");
int size = input.nextInt();
input.nextLine();

System.out.print(
    "Enter number of trials to average: ");
int numTrials = input.nextInt();
input.nextLine();

Random rand = new Random(1234);
int[] array = new int[size];

double timeSum = 0.0;
for (int trial = 1; trial <= numTrials + 1; trials++){
    for (int k = 0; k < array.length; k++)
        array[k] = rand.nextInt();
long endTime = 0L; startTime = 0L;
if (choice == 1)
{
    sortType = "Selection type"; 
    startTime = System.nanoTime(); // nano sec
    IntArraySort.selectionSort(array, 0,
                                array.length - 1);
    endTime = System.nanoTime();
}
else
{
    sortType = "Insertion type";
    startTime = System.nanoTime(); // nano sec
    IntArraySort.insertionSort(array, 0,
                                array.length - 1);
    endTime = System.nanoTime();
}
// skip first r=trial for the just in time
// compiler and do numTrials trials

if (trial > 1)
{
    // calculate seconds from nano seconds

double sortTime = (endTime - startTime)/ 1E9;
System.out.println("Sort time: "+sortTime+" seconds");
timeSum = timeSum + (double) sortTime;
}
} // end for loop

// compute time average over all trials
double averageTime = timeSum / (double) numTrials;
System.out.println();
System.out.println(sortType);
System.out.println("Array size (n): " + size);
System.out.println("Number of trials: " + numTrials);
System.out.println("Average running time: " + averageTime + " seconds");
double nSquared = (double) size * (double) size;
double ratio = averageTime / nSquared;
System.out.println("average / n^2: " + ratio);
}  // end of doTest method

public static void main(String[] args)
{  QuadraticSortTimer timer = new QuadraticSortTimer();
    timer.doTest();
}
}  // end of class QuadraticSortTimer
Comparison results (2001)

<table>
<thead>
<tr>
<th>size $n$</th>
<th>selection sort</th>
<th>$T(n)/n^2$</th>
<th>insertion sort</th>
<th>$T(n)/n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>0.41 sec</td>
<td>$1.64 \times 10^{-8}$</td>
<td>0.25 sec</td>
<td>$1.01 \times 10^{-8}$</td>
</tr>
<tr>
<td>10,000</td>
<td>2.07 sec</td>
<td>$2.07 \times 10^{-8}$</td>
<td>1.02 sec</td>
<td>$1.02 \times 10^{-8}$</td>
</tr>
<tr>
<td>20,000</td>
<td>8.50 sec</td>
<td>$2.12 \times 10^{-8}$</td>
<td>4.36 sec</td>
<td>$1.09 \times 10^{-8}$</td>
</tr>
<tr>
<td>30,000</td>
<td>19.5 sec</td>
<td>$2.16 \times 10^{-8}$</td>
<td>10.0 sec</td>
<td>$1.11 \times 10^{-8}$</td>
</tr>
<tr>
<td>40,000</td>
<td>34.7 sec</td>
<td>$2.17 \times 10^{-8}$</td>
<td>18.8 sec</td>
<td>$1.17 \times 10^{-8}$</td>
</tr>
<tr>
<td>50,000</td>
<td>53.6 sec</td>
<td>$2.15 \times 10^{-8}$</td>
<td>29.0 sec</td>
<td>$1.16 \times 10^{-8}$</td>
</tr>
<tr>
<td>100,000</td>
<td>213.0 sec</td>
<td>$2.13 \times 10^{-8}$</td>
<td>125.0 sec</td>
<td>$1.25 \times 10^{-8}$</td>
</tr>
</tbody>
</table>
Comparison results (2005)

<table>
<thead>
<tr>
<th>size $n$</th>
<th>selection sort</th>
<th>$T(n)/n^2$</th>
<th>insertion sort</th>
<th>$T(n)/n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>0.068 sec</td>
<td>$2.40 \times 10^{-9}$</td>
<td>0.044 sec</td>
<td>$1.76 \times 10^{-9}$</td>
</tr>
<tr>
<td>10,000</td>
<td>0.242 sec</td>
<td>$2.40 \times 10^{-9}$</td>
<td>0.172 sec</td>
<td>$1.72 \times 10^{-9}$</td>
</tr>
<tr>
<td>20,000</td>
<td>0.949 sec</td>
<td>$2.37 \times 10^{-9}$</td>
<td>0.679 sec</td>
<td>$1.70 \times 10^{-9}$</td>
</tr>
<tr>
<td>30,000</td>
<td>2.141 sec</td>
<td>$2.38 \times 10^{-9}$</td>
<td>1.524 sec</td>
<td>$1.69 \times 10^{-9}$</td>
</tr>
<tr>
<td>40,000</td>
<td>3.796 sec</td>
<td>$2.37 \times 10^{-9}$</td>
<td>2.706 sec</td>
<td>$1.69 \times 10^{-9}$</td>
</tr>
<tr>
<td>50,000</td>
<td>5.943 sec</td>
<td>$2.38 \times 10^{-9}$</td>
<td>4.244 sec</td>
<td>$1.70 \times 10^{-9}$</td>
</tr>
<tr>
<td>100,000</td>
<td>23.73 sec</td>
<td>$2.37 \times 10^{-9}$</td>
<td>16.92 sec</td>
<td>$1.69 \times 10^{-9}$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2370 sec</td>
<td></td>
<td>1690 sec</td>
<td></td>
</tr>
<tr>
<td>5,000,000</td>
<td>59,250 sec (16 hours)</td>
<td></td>
<td>42,250 sec (12 hours)</td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td>237,000 sec (66 hours)</td>
<td></td>
<td>169,000 sec (38 hours)</td>
<td></td>
</tr>
</tbody>
</table>
Mergesort (1)

A naturally recursive algorithm for sorting the subarray \( \langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \)

Split it into two halves

\[
L = \langle a_{\text{start}}, \ldots, a_{\text{mid}} \rangle \quad R = \langle a_{\text{mid}+1}, \ldots, a_{\text{end}} \rangle
\]

where \( \text{mid} = (\text{start} + \text{end})/2 \)

Apply mergesort to each half and merge the sorted halves into a sorted subarray.
Mergesort (2)

We need a merge function that will merge two sorted halves together into a sorted subarray.

For example if \( L = \langle 1,8,12,15,17 \rangle \) is the left half and \( R = \langle 2,9,10,19,21,23,25 \rangle \) is the right half then \( L \) and \( R \) are separately sorted.

The merge function should produce the sorted subarray \( \langle 1,2,8,9,10,12,15,17,19,21,23,25 \rangle \).
public static void mergeSort(int[] a, int start, int end) {
    if (start == end) {
        return; // one element subarray already sorted
    }
    int mid = (start + end) / 2;
    mergeSort(a, start, mid); // sort left half
    mergeSort(a, mid+1, end); // sort right half
    merge(a, start, mid, end);
}
Given a subarray $\langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle$ partitioned by an index $\text{split}$ with $\text{start} \leq \text{split} \leq \text{end}$, such that the subarrays $\langle a_{\text{start}}, \ldots, a_{\text{split}} \rangle$ and $\langle a_{\text{split}+1}, \ldots, a_{\text{end}} \rangle$ are each sorted in increasing order, sort the entire subarray $\langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle$ into a temporary array $\langle t_0, \ldots, t_{\text{end} - \text{start} + 1} \rangle$ such that $t_0 \leq t_1 \leq \cdots \leq t_{\text{end} - \text{start} + 1}$.
The array $\langle 1,8,12,15,17,2,9,10,19,21,23,25 \rangle$ is not sorted

If we choose $\text{split} = 4$ then the subarrays:

$L = \langle 1,8,12,15,17 \rangle$

$R = \langle 2,9,10,19,21,23,25 \rangle$

are separately sorted in increasing order.

The entire array can be sorted by a pairing technique called merging.
Merge algorithm

ALGORITHM merge(⟨a₀,…, aₙ₋₁⟩, start, split, end)

\[ t ← ⟨t₀, …, t_{end−start+1}⟩ \]

WHILE neither \( ⟨a_{start}, …, a_{split}⟩ \) nor \( ⟨a_{split+1}, …, a_{end}⟩ \) is exhausted DO

\[ \text{Compare pairs of elements and copy smallest to temp array } t \]

END WHILE

WHILE elements remain in \( ⟨a_{start}, …, a_{split}⟩ \) DO

\[ \text{Copy them to } t \]

END WHILE

WHILE elements remain in \( ⟨a_{split+1}, …, a_{end}⟩ \) DO

\[ \text{Copy them to } t \]

END WHILE

\[ ⟨a_{start}, …, a_{end}⟩ ← ⟨t₀, …, t_{end−start+1}⟩ \]
public static void merge(int[], int start, int split, int end) {

Define indices into subarrays and temporary array

    int n = end - start + 1; // number of elements
    int[] t = new int[n];    // temporary storage
    int i = start;           // index into left subarray
    int j = split + 1;      // index into right subarray
    int k = 0;               // index into temporary storage
merge elements from left and right subarrays to temp array until one or both of the subarrays are exhausted

while (i <= split && j <= end)
{
    if (a[i] < a[j]) // left element is smaller
    {
        t[k] = a[i];  // move ot to temp array
        i++;          // index of next left element
    }
    else // right element is smaller
    {
        t[k] = a[j];  // move it to temp array
        j++;          // index of next right element
    }
    k++;  // set index to next temp position
}

Java merge method (2)
Java merge method (3)

Copy any remaining elements from the left subarray to the temporary array \( t \)

```java
while (i <= split) {
    t[k] = a[i];
    i++;
    k++;
}
```
Copy any remaining elements from the right subarray to the temporary array \( t \)

```java
while (j <= end)
{
    t[k] = a[j];
    j++;
    k++;
}
```
Copy all elements from the temporary array `t` to the array `a`. For increased efficiency we can also use

```java
System.arraycopy(t, 0, a, start, end - start + 1)
```

```java
for (k = 0; k < n; k++)
{
    a[start + k] = t[k];
}
} // end merge method
```
Compact version of loops

Here is a compact version of the while loops

```c
while (i <= split && j <= end)
    if (a[i] < a[j]) t[k++] = a[i++];
    else t[k++] = a[j++];
while (i <= split) t[k++] = a[i++];
while (j <= end) t[k++] = a[j++];
for (k = 0; k < nl k++) a[start + k] = t[k];
```

Here \( t[k++] \) means first use \( k \) as an index, then increment \( k \). Therefore a statement such as \( t[k++] = a[i++] \) is equivalent to the three assignment statements
\[
t[k] = a[i]; k = k + 1; i = i + 1;
\]
Put `merge` and `mergeSort` methods in the `IntArraySort` class as we did for other algorithms

```java
public class IntArraySort {
    public static void selectionSort(int[] a, int start, int end) {...}
    public static void insertionSort(int[] a, int start, int end) {...}
    public static void mergeSort(int[] a, int start, int end) {...}
    public static void merge(int[] a, int start, int split, int end) {...}
}
```

`mergeSort` is tested in class `SortTester`
Mergesort running time (1)

Let $T(n)$ be the running time for array size $n$.

The top level mergesort algorithm shows that $T(n) = T(n/2) + T(n/2) + \text{time for merge}$.

The merge algorithm is $O(n)$ so let's assume that its running time is $an + b$. Therefore $T(n) = 2T(n/2) + an + b$. 

For simplicity we assume that \( n = 2^m \) for some \( m \).

Results hold for any \( n \) but proof is more complicated.

Now use the recurrence relation
\[
T(n) = 2T(n/2) + an + b
\]
to express \( T(n/2) \) in terms of \( T(n/2^2) \).

Then express \( T(n/2^2) \) in terms of \( T(n/2^3) \) and so on ....
Mergesort running time (3)

\[ T(n) = 2 T\left(\frac{n}{2}\right) + an + b \]
\[ = 2 \left[ 2 T\left(\frac{n}{2^2}\right) + a \frac{n}{2} + b \right] + an + b \]
\[ = 2^2 T\left(\frac{n}{2^2}\right) + 2 an + (1+2)b \]
\[ = 2^2 \left( 2 T\left(\frac{n}{2^3}\right) + an/2^2 + b \right) + 2 an + (1+2)b \]
\[ = 2^3 T\left(\frac{n}{2^3}\right) + 3 an + (1+2+2^2)b \]

... 
\[ = 2^m T\left(\frac{n}{2^m}\right) + anm + (1 + 2 + \cdots + 2^{m-1})b \]
\[ = 2^m T\left(\frac{n}{2^m}\right) + anm + (2^m - 1)b \]
\[ = n + an \log_2 n + (n-1)b \]
\[ = O(n \log_2 n) = O(n \log n) \]
## Running time comparison

<table>
<thead>
<tr>
<th>( \log_2 n )</th>
<th>( n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.32</td>
<td>10</td>
<td>33.2</td>
<td>( 10^2 )</td>
</tr>
<tr>
<td>6.64</td>
<td>( 10^2 )</td>
<td>664</td>
<td>( 10^4 )</td>
</tr>
<tr>
<td>9.97</td>
<td>( 10^3 )</td>
<td>9.97( \times )10^3</td>
<td>( 10^6 )</td>
</tr>
<tr>
<td>13.3</td>
<td>( 10^4 )</td>
<td>1.33( \times )10^5</td>
<td>( 10^8 )</td>
</tr>
<tr>
<td>16.6</td>
<td>( 10^5 )</td>
<td>1.66( \times )10^6</td>
<td>( 10^{10} )</td>
</tr>
<tr>
<td>19.9</td>
<td>( 10^6 )</td>
<td>1.99( \times )10^7</td>
<td>( 10^{12} )</td>
</tr>
<tr>
<td>23.3</td>
<td>( 10^7 )</td>
<td>2.33( \times )10^8</td>
<td>( 10^{14} )</td>
</tr>
<tr>
<td>26.6</td>
<td>( 10^8 )</td>
<td>2.66( \times )10^9</td>
<td>( 10^{16} )</td>
</tr>
<tr>
<td>29.9</td>
<td>( 10^9 )</td>
<td>2.99( \times )10^{10}</td>
<td>( 10^{18} )</td>
</tr>
</tbody>
</table>
Mergesort is thousands to millions of times faster than either selection sort or insertion sort.

Its only disadvantage is the extra storage space required for the temporary array.
The file merge problem (1)

★ The merge algorithm that is part of mergesort is useful by itself for sorting files of data

★ PROBLEM: Given two files of records that are each sorted on some field in a specified order, produce a new file that contains all the records of both files in sorted order

★ Example: merge two files of bank accounts sorted in order of increasing account number
accounts1.dat (sorted)

175:Linda Kerr:8008.43
424:Mary Binch:35135.5
932:Harry Garfield:4723.1
1134:Alfred Vaillancourt:51914.93
1345:Amy Flintstone:81507.31
2489:Barney Lafreniere:66568.5
7123:Jean Ebert:53361.25
7845:Marc Gardiner:7541.34
9243:Dan Sinclair:4151.34
9546:Peter Jensen:16146.29
## accounts2.dat (sorted)

<table>
<thead>
<tr>
<th>Account</th>
<th>Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>310</td>
<td>Don Laing</td>
<td>12337.39</td>
</tr>
<tr>
<td>417</td>
<td>Marc Tyler</td>
<td>3455.42</td>
</tr>
<tr>
<td>811</td>
<td>Amanda Schryer</td>
<td>3541.15</td>
</tr>
<tr>
<td>1219</td>
<td>Gilles Olivier</td>
<td>84285.23</td>
</tr>
<tr>
<td>1765</td>
<td>Patricia Innes</td>
<td>4494.43</td>
</tr>
<tr>
<td>9623</td>
<td>Remi Martin</td>
<td>19732.12</td>
</tr>
<tr>
<td>9754</td>
<td>Patricia Schneider</td>
<td>40066.75</td>
</tr>
<tr>
<td>9912</td>
<td>Mike Laforge</td>
<td>5798.0</td>
</tr>
</tbody>
</table>
merge.dat (sorted)

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>175</td>
<td>Linda Kerr</td>
<td>8008.43</td>
</tr>
<tr>
<td>310</td>
<td>Don Laing</td>
<td>12337.39</td>
</tr>
<tr>
<td>417</td>
<td>Marc Tyler</td>
<td>3455.42</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7123</td>
<td>Jean Ebert</td>
<td>53361.25</td>
</tr>
<tr>
<td>7845</td>
<td>Marc Gardiner</td>
<td>7541.32</td>
</tr>
<tr>
<td>9243</td>
<td>Dan Sinclair</td>
<td>4151.34</td>
</tr>
<tr>
<td>9546</td>
<td>Peter Jensen</td>
<td>16146.29</td>
</tr>
<tr>
<td>9623</td>
<td>Remi Martin</td>
<td>19732.12</td>
</tr>
<tr>
<td>9754</td>
<td>Patricia Schneider</td>
<td>40066.76</td>
</tr>
<tr>
<td>9912</td>
<td>Mike Laforge</td>
<td>5798.0</td>
</tr>
</tbody>
</table>
Read two input bank account files that are sorted and merge them to produce a sorted output file

```java
public class FileMerger {
    private File inFile1;
    private File inFile2;
    private File outFile;

    public FileMerger(File inFile1, File inFile2, File outFile) {
        this.inFile1 = inFile1;
        this.inFile2 = inFile2;
        this.outFile = outFile;
    }
}
```
Merge two input files according to increasing account number into a single sorted file. We can use our PrintAccountWriter and BufferedAccountReader classes.

```java
public void mergeFiles() throws IOException {
    BufferedAccountReader in1 =
        new BufferedAccountReader(
            new FileReader(inFile1));

    BufferedAccountReader in2 =
        new BufferedAccountReader(
            new FileReader(inFile2));

    PrintAccountWriter out =
        new PrintAccountWriter(
            new FileWriter(outFile));
```
BankAccount b1 = in1.readAccount();
BankAccount b2 = in2.readAccount();
while (b1 != null && b2 != null)
{
    if (b1.getNumber() < b2.getNumber())
    {
        out.writeAccount(b1);
        b1 = in1.readAccount();
    }
    else
    {
        out.writeAccount(b2);
        b2 = in2.readAccount();
    }
}
If there are accounts remaining in the first file write them to the output file until we find end of file on the first file. Then do the same for the second file.

```java
while (b1 != null) {
    out.writeAccount(b1);
    b1 = in1.readAccount();
}

while (b2 != null) {
    out.writeAccount(b2);
    b2 = in2.readAccount();
}
```
Class FileMerger (5)

close all the files

```java
in1.close();
in2.close();
out.close();
} // end of the fileMerge method
```
public static void main(String[] args) {
    if (args.length == 3) {
        File inFile1 = new File(args[0]);
        File inFile2 = new File(args[1]);
        File outFile = new File(args[2]);
        FileMerger merger =
            new FileMerger(inFile1, inFile2, outFile);
        merger.mergeFiles();
    } else {
        System.out.println(
            "args: inFileName1 inFileName2, outFileName");
    }
} // end of FileMerger class
Quicksort (1)

- A very efficient sorting algorithm
- Uses the recursive divide and conquer technique as in binary search and merge sort
- Subarray is partitioned into two parts in such a way that neither part is necessarily sorted but such that each element in the left part is less or equal to every element in the right part
A partition of the subarray \( \langle a_{\text{start}}, \ldots, a_{\text{end}} \rangle \) defined by \( x \) has the form

\[
L = \langle a_{\text{start}}, \ldots, a_{\text{split}} \rangle \quad R = \langle a_{\text{split}+1}, \ldots, a_{\text{end}} \rangle
\]

Here \( x = a_{\text{split}} \) is such that

For each \( y \) in \( L \) we have \( y \leq a_{\text{split}} \)

For each \( y \) in \( R \) we have \( y \geq a_{\text{split}} \)
Quicksort (3)

★ The division process into subarrays is $O(\log n)$

★ The partitioning process is $O(n)$

★ Then the average behaviour of quicksort is $O(n \log n)$ as in mergesort

★ Unlike mergesort, quicksort doesn't need a temporary array but its worst case behaviour is $O(n^2)$
Example 1

Consider the array $<8,1,10,4,6,5,3,2,22>$

Choose middle element 6 as pivot element

A partition is defined by $L = <2,1,6,4,5,3>$ and $R = <10,8,22>$

Here every element in $L$ is less or equal to 6 and every element in $R$ is greater or equal to 6 even though neither $L$ nor $R$ are sorted
Example 2

Consider the array \(<8,1,6,4,10,5,3,2,22>\)

Choose middle element 10 as pivot element

A partition is defined by
\[ L = <2,1,6,4,8,5,3,10> \text{ and } R = <22> \]

Here every element in \( L \) is less or equal to 10 and every element in \( R \) is greater or equal to 10 even though neither \( L \) nor \( R \) are sorted
Quicksort

8,1,6,4,10,5,3,2,22

2,1,6,4,8,5,3,10

3,1,2,4

1

2

3

3,2,4

5

4

8,5,6,10

8,6,10

8,10

8

10
Top level QuickSort method

Top level recursive algorithm

```java
public static void quickSort(int[] a, int start, int end) {
    if (start < end) {
        int split = partition(a, start, end);
        quickSort(a, start, split-1); // sort left half
        quickSort(a, split+1, end); // sort right half
    }
}
```

The pivot is in its correct position so it is not included in the subarrays for either of the recursive calls
Java partition method (1)

```java
public static int partition(int[] a, int start, int end) {
    choose middle element as pivot and move it to the start of the subarray temporarily
    swap(a, (start + end) / 2, start);
    int pivot = a[start];
}
```
Java partition method (2)

Partition elements $a[start + 1]$ to $a[end]$. 

$lastLeft$ is index of the last element in the left subarray. The elements $a[start]$ to $a[lastIndex]$ are less than or equal to the pivot value.

```java
int lastLeft = start;
for (int j = start + 1; j <= end; j++)
    if (a[j] < pivot)
        lastLeft++;
    swap(a, j, lastLeft);

// Move pivot back to its correct position
swap(a, start, lastLeft);
return lastLeft;
} // end of partition method
```
Java partition method (3)

The `swap` method exchanges two elements of array `a` with indices `i` and `j`.

```java
private static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```
Put the `partition`, `swap`, and `quickSort` methods in the `IntArraySort` class as we did for the other sorting algorithms.

```java
public class IntArraySort {
    public static void selectionSort(int[] a, int start, int end) {...}
    public static void insertionSort(int[] a, int start, int end) {...}
    public static void mergeSort(int[] a, int start, int end) {...}
    public static void merge(int[] a, int start, int split, int end) {...}
}```
private static int partition(int[] a, int start, int end) {...}

private static void quickSort(int[] a, int start, int end) {...}

private static void swap(int[] a, int start, int end) {...}

} // end of IntArraySort

quickSort is tested in class SortTester
<table>
<thead>
<tr>
<th>Point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>★</td>
<td>Average case behaviour is same as mergesort</td>
</tr>
<tr>
<td>★</td>
<td>Same recurrence relation is obtained</td>
</tr>
<tr>
<td>★</td>
<td>Disadvantage of quicksort is that the worst case behaviour is $O(n^2)$</td>
</tr>
<tr>
<td>★</td>
<td>This can be avoided by some tricks to choose the pivot element</td>
</tr>
</tbody>
</table>
Class FasterSortTimer

This class times mergesort and quicksort and is almost identical to \texttt{QuadraticSortTimer}
## Merge/Quick sort (2001)

<table>
<thead>
<tr>
<th>size $n$</th>
<th>merge</th>
<th>$T(n)/n \ln n$</th>
<th>quicksort</th>
<th>$T(n)/n \ln n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>0.03 sec</td>
<td>$8.0 \times 10^{-7}$</td>
<td>0.01 sec</td>
<td>$2.0 \times 10^{-7}$</td>
</tr>
<tr>
<td>10,000</td>
<td>0.07 sec</td>
<td>$7.6 \times 10^{-7}$</td>
<td>0.02 sec</td>
<td>$2.1 \times 10^{-7}$</td>
</tr>
<tr>
<td>100,000</td>
<td>0.65 sec</td>
<td>$5.7 \times 10^{-7}$</td>
<td>0.17 sec</td>
<td>$1.5 \times 10^{-7}$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>7.01 sec</td>
<td>$5.1 \times 10^{-7}$</td>
<td>2.09 sec</td>
<td>$1.5 \times 10^{-7}$</td>
</tr>
<tr>
<td>5,000,000</td>
<td>36.7 sec</td>
<td>$5.1 \times 10^{-7}$</td>
<td>12.1 sec</td>
<td>$1.6 \times 10^{-7}$</td>
</tr>
</tbody>
</table>
## Merge/Quick sort (2005)

<table>
<thead>
<tr>
<th>size $n$</th>
<th>merge</th>
<th>$T(n)/n \ln n$</th>
<th>quicksort</th>
<th>$T(n)/n \ln n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>0.004 sec</td>
<td>$9.39 \times 10^{-8}$</td>
<td>0.00 sec</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.006 sec</td>
<td>$6.51 \times 10^{-8}$</td>
<td>0.00 sec</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>0.052 sec</td>
<td>$4.52 \times 10^{-8}$</td>
<td>0.026 sec</td>
<td>$2.26 \times 10^{-8}$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.667 sec</td>
<td>$4.83 \times 10^{-8}$</td>
<td>0.322 sec</td>
<td>$2.33 \times 10^{-8}$</td>
</tr>
<tr>
<td>5,000,000</td>
<td>3.787 sec</td>
<td>$4.91 \times 10^{-8}$</td>
<td>1.785 sec</td>
<td>$2.31 \times 10^{-8}$</td>
</tr>
<tr>
<td>10,000,000</td>
<td>--------</td>
<td>--------</td>
<td>3.783 sec</td>
<td>$2.31 \times 10^{-8}$</td>
</tr>
</tbody>
</table>
package chapter12.sorting;
import java.util.Scanner;

public class SortTester
{
    public void doTest()
    {
        Scanner input = new Scanner(System.in);

        System.out.print("Enter number of integers in array: ");
        int size = input.nextInt(); input.nextLine();
    }
}
int[] testArray = new int[size];
for (int k = 0; k < testArray.length; k++)
{
    System.out.println("Enter element "+k+": ");
    testArray[k] = input.nextInt();
    input.nextLine();
}

System.out.print(
    "Enter start index for subarray: ");
int start = input.nextInt(); input.nextLine();
System.out.print(
    "Enter end index for subarray: ");
int end = input.nextInt(); input.nextLine();
SortTester (3)

```java
int[] testArrayCopy;

testArrayCopy = arrayCopy(testArray);
IntArraySort.selectionSort(testArrayCopy, start, end);
System.out.println(
    "Selection sort: Sorted subarray is");
printArray(testArrayCopy, start, end);

testArrayCopy = arrayCopy(testArray);
IntArraySort.insertionSort(testArrayCopy, start, end);
System.out.println(
    "insertion sort: Sorted subarray is");
printArray(testArrayCopy, start, end);
```
testArratCopy = arrayCopy(testArray);
IntArraySort.mergeSort(testArrayCopy,
    start, end);
System.out.println(
    "Merge sort: Sorted subarray is");
printArray(testArrayCopy, start, end);

testArratCopy = arrayCopy(testArray);
IntArraySort.quickSort(testArrayCopy,
    start, end);
System.out.println(
    "Quick sort: Sorted subarray is");
printArray(testArrayCopy, start, end);
}
// end of doTest method
private void printArray(int[] a, int start, int end) {
    System.out.print("[");
    for (int k = start; k <= end; k++)
    {
        System.out.print(a[k]);
        if (k < end) System.out.print(",");
    }
    System.out.println("]");
    System.out.println();
}
private int[] arrayCopy(int[] a) {
    int[] copy = new int[a.length];
    for (int k = 0; k < a.length; k++) {
        copy[k] = a[k];
    }
    return copy;
}

public static void main(String[] args) {
    new SortTester().doTest();
}
Our sorting algorithms so far will only work with arrays of type `int[]`

What do we do to sort other kinds of arrays such as arrays of `BankAccount` objects?

Also we may want to sort in a different order: for example, decreasing instead of increasing
Generic arrays

We can write our sorting methods using a generic object type parameter instead of a specific type such as int. This does not work for primitive types.

We can use a generic type parameter $E$ and an array whose elements are of type $E$ using the following statement to construct the array:

$$E[] \text{ array } = (E[]) \text{ new Object}[n];$$

Here it is necessary to typecast the `Object` type to the type $E$. 

★

★

★
Generic comparison

We need a way to compare two objects of type E

For example in our `selectionSort` method for `int[]` arrays we have the statement

```java
if (a[j] < a[k]) k = j;
```

This will not work if we want a method to sort an array of strings. Then we would need to use the statement

```java
if (a[j].compareTo(a[k])) k = j;
```
Comparator\(<E>\) interface

One solution is to use the Comparator\(<E>\) interface for comparing objects of type \(E\). To implement this interface we need to provide an implementation of the following method

\[
\text{public int compare}(E \text{ obj1, E obj2});
\]

Here a negative value is returned if \text{obj1} "is less than" \text{obj2}, 0 if \text{obj1} "is equal to" \text{obj2} and a positive value if \text{obj1} "is greater than" \text{obj2}
Using Comparator<E> interface

We can now make a generic version of selection sort by changing the prototype to

```java
public static <E> void selectionSort(E[] a, int start, int end, Comparator<E> f)
{
...
}
```

Here we have added a `Comparator<E>` argument so for the comparison we can use

```java
if (f.compareTo(a[j], a[k]) < 0) k = j;
```

Here `f` is an object from any class that implements the `Comparator<E>` interface
Generic types

A generic type can be any object type

A primitive type such as int or double cannot be a generic type: only object types can be generic

If you want to sort an array of type int[] using a method of generic type it is necessary to use the wrapper class Integer. Similarly for double type use the wrapper class Double.

Autoboxing and unboxing simplify this
Generic selection sort

```java
public static <E> void selectionSort(E[] a, int start, int end, Comparator<E> f)
{
    for (int i = start; i < end; i++)
    {
        int k = i;
        for (int j = i+1; j <= end; j++)
        {
            if (f.compare(a[j],a[k]) < 0)
                k = j;
        }

        E temp = a[k];
        a[k] = a[i];
        a[i] = temp;
    }
}
```

Note the generic type here
public static <E> void insertionSort(E[] a, 
   int start, int end, Comparator<E> f)
{
    for (int i = start + 1; i < end; i++)
    {
      E x = a[i];
      int j = i-1;
      while ((j >= start) && (f.compare(x,a[j]) < 0))
      {
        a[j+1] = a[j];
        j--;
      }
      a[j+1] = x;
    }
}
public class GenericArraySort
{
    public static <E> selectionSort(E[] a,
            int start, int end, Comparator<E> f)
    {
        ...
    }

    public static <E> insertionSort(E[] a,
            int start, int end, Comparator<E> f)
    {
        ...
    }

    // generic merge sort and quick sort go here
To sort an array of strings in lexicographical order first define the following class which uses the `compareTo` method in the `String` class to implement the `compare` method of the `Comparator<E>` interface

```java
public class StringComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
}
```

The type `E` is `String` now.
To sort an array of strings in reverse lexicographical order first define the following class which uses the `compareTo` method in the `String` class to implement the `compare` method of the `Comparator<E>` interface.

```java
public class StringDecreasingComparator implements Comparator<String>
{
    public int compare(String s1, String s2)
    {
        return s2.compareTo(s1);
    }
}
```

Note the order.
Sorting an array of strings (3)

The following statements can be used to sort an array of strings in lexicographical order.

```java
String[] s = { "one", "two", "three", "four" };
GenericArraySort.selectionSort(s, 0, s.length - 1, new StringComparator());
```

To sort in reverse lexicographical order use

```java
String[] s = { "one", "two", "three", "four" };
GenericArraySort.selectionSort(s, 0, s.length - 1, new StringDecreasingComparator());
```
To sort an array of `BankAccount` objects in order of increasing account number define the following class which implements the `Comparator<BankAccount>` class:

```java
public class AccountNumberComparator implements Comparator<BankAccount>
{
    public int compare(BankAccount b1, BankAccount b2)
    {
        return b1.getNumber() - b2.getNumber();
    }
}
```
To sort an array of `BankAccount` objects in order of increasing balance define the following class which implements the `Comparator<BankAccount>` class.

```java
public class AccountBalanceComparator implements Comparator<BankAccount>
{
    public int compare(BankAccount b1, BankAccount b2)
    {
        return b1.getBalance() - b2.getBalance();
    }
}
```
Comparable<E> interface

A class can implement the Comparable<E> interface

The String class and many other classes implement this interface if there is some standard ordering of the elements of the class

To implement this interface a class must provide an implementation of the following method

```java
public int compareTo(E element)
```

The Comparator<E> interface is more general since it can be used to provide several different orderings of the objects of a class
The Arrays class

Java provides an \texttt{Arrays} class that contains searching and sorting methods for arrays. To use these methods the array elements must be from a class that implements the \texttt{Comparable\langle E \rangle} interface.

There are also versions of these methods that use the \texttt{Comparator\langle E \rangle} interface.

See textbook for more on these methods and examples.
public class Test {
    System.out.println("Hello");
}

public class Test {
    System.out.println("Hello");
}
Template 2

left justified text

- sub item
- sub item
- sub item