CS 2110
Lecture 11

Loop invariants
Searching
Sorting
Administrivia

- A4 due this week
- Test2 is tomorrow
- If you haven’t turned in A3, there’s a problem
Loops are getting harder to write
/** Returns: 1 + ... + `n`. Requires: `n` is positive. */

```c
int sumRange(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```
/** Returns: the frequency of `item` in `items`. Requires: (Omitted: bag invariant on `items` and `size`.) */

int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (items[i].equals(item)) {
            count++;
        }
    }
    return count;
}
/** ??? */
int mystery(int[] b, int v) {
    int k = -1;
    int t = b.length;
    while (k != t - 1) {
        int e = (k + t) / 2;
        if (b[e] <= v) {
            k = e;
        } else {
            t = e;
        }
    }
    return k;
}
// Pseudocode for an algorithm in week 14

frontier = new PriorityQueue();
root.dist = 0;
frontier.push(root);

while (frontier not empty) {
    g = frontier.pop();
    foreach (edge (g \rightarrow_d v) in E) {
        if (v.dist == \infty) {
            v.dist = g.dist + d;
            frontier.push(v);
        } else {
            if (g.dist + d < v.dist) {
                v.dist = g.dist + d;
                frontier.increasePriority(v);
            }
        }
    }
}

🤯
Recall: rewrite for as while loop

```plaintext
for (init; guard; update) {
    body;
}
```

```plaintext
init;
while (guard) {
    body;
    update;
}
```
Exercise

**Explain** in your own words why this loop works.

```java
/** Returns: the frequency of `item` in `items`. 
   Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```
Exercise

**Explain** in your own words why this loop works.

```java
/** Returns: the frequency of `item` in `items`. Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```
Exercise

**Exercise**

*Explain* in your own words why this loop works.

```java
/** Returns: the frequency of `item` in `items`. 
Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```

`count` is freq of `item` in first `size` elements of `items`
Exercise

**Exercise**: Explain in your own words why this loop works.

```java
/**
 * Returns: the frequency of `item` in `items`.
 * Requires: `items` has `size` items at indices 0..`size`-1.
 */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```

`count` is freq of `item` in first 0 elements of `items`
Exercise

**Exercise** in your own words why this loop works.

```java
/** Returns: the frequency of `item` in `items`. Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```

`count` is freq of `item` in first `i` elements of `items`
This comment, the **loop invariant**, is the core idea of why the loop works.

```java
/**
 * Returns: the frequency of `item` in `items`.
 * Requires: `items` has `size` items at indices 0..`size`-1.
 */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    // inv: `count` is freq of `item` in first `i` elements of `items`
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count; }
```
Some helpful, concise notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>When</th>
<th>Means</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>i..j</td>
<td>i &lt; j</td>
<td>i, i+1, ..., j-1, j</td>
<td>2..4 is 2, 3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i = j</td>
<td>i</td>
<td>2..2 is 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i &gt; j</td>
<td><em>empty</em></td>
<td>2..1 is empty</td>
</tr>
</tbody>
</table>
### Some helpful, concise notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>When</th>
<th>Means</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Range</strong></td>
<td>i..j</td>
<td>i &lt; j</td>
<td>i, i+1, …, j-1, j</td>
<td>2..4 is 2, 3, 4</td>
</tr>
<tr>
<td></td>
<td>i = j</td>
<td>i</td>
<td>i</td>
<td>2..2 is 2</td>
</tr>
<tr>
<td></td>
<td>i &gt; j</td>
<td>empty</td>
<td>empty</td>
<td>2..1 is empty</td>
</tr>
<tr>
<td><strong>Closed range</strong></td>
<td>[i..j]</td>
<td></td>
<td>Same as i..j</td>
<td>[2..4] is 2, 3, 4</td>
</tr>
<tr>
<td><strong>Open range</strong></td>
<td>(i..j)</td>
<td></td>
<td>Same as [(i+1)..(j-1)]</td>
<td>(2..4) is 3</td>
</tr>
<tr>
<td><strong>Half-open range</strong></td>
<td>[i..j)</td>
<td></td>
<td>Same as [i..(j-1)]</td>
<td>[2..4) is 2, 3</td>
</tr>
<tr>
<td></td>
<td>(i..j]</td>
<td></td>
<td>Same as [(i+1)..j]</td>
<td>(2..4] is 3, 4</td>
</tr>
</tbody>
</table>
Some helpful, concise notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>When</th>
<th>Means</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>i..j</td>
<td>i &lt; j</td>
<td>i, i+1, …, j-1, j</td>
<td>2..4 is 2, 3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i = j</td>
<td>i</td>
<td>2..2 is 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i &gt; j</td>
<td>empty</td>
<td>2..1 is empty</td>
</tr>
<tr>
<td>Closed range</td>
<td>[i..j]</td>
<td></td>
<td>Same as i..j</td>
<td>[2..4] is 2, 3, 4</td>
</tr>
<tr>
<td>Open range</td>
<td>(i..j)</td>
<td></td>
<td>Same as [(i+1)..(j-1)]</td>
<td>(2..4) is 3</td>
</tr>
<tr>
<td>Half-open range</td>
<td>[i..j)</td>
<td></td>
<td>Same as [i..(j-1)]</td>
<td>[2..4) is 2, 3</td>
</tr>
<tr>
<td></td>
<td>(i..j]</td>
<td></td>
<td>Same as [(i+1)..j]</td>
<td>(2..4] is 3, 4</td>
</tr>
<tr>
<td>Omitted first</td>
<td>[..j]</td>
<td>0-based</td>
<td>Same as [0..j]</td>
<td>[..2] is 0, 1, 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1-based</td>
<td>Same as [1..j]</td>
<td>[..2] is 1, 2</td>
</tr>
<tr>
<td>Omitted last</td>
<td>[i..]</td>
<td>Some last # implied</td>
<td>Same as [i../last]</td>
<td>Depends on context</td>
</tr>
</tbody>
</table>
Some helpful, concise notation

<table>
<thead>
<tr>
<th>Context</th>
<th>Application</th>
</tr>
</thead>
</table>
| Arrays (0-based) | a[2..4] is a[2], a[3], a[4]  
                   | a[..3] is a[0], a[1], a[2], a[3]  
                   | a[1..] is a[1], …, a[a.length – 1] |
| Strings (0-based)| “Hello”[2..4] is “llo”  
                   | “Hello”[..2] is “Hel” |
| Lists            | lst[3..4] are elements lst.get(3), list.get(4)  
                   | (Beware 0-based vs 1-based) |

⚠ Ranges are not the same as the Python slice operator.
Answer, Part 1 (part 2 will come much later)

This comment, the loop invariant, is the core idea of why the loop works.

```java
/** Returns: the frequency of `item` in `items`. Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    // inv: `count` is freq of `item` in `items[..i)`
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count; }
```
What is a loop invariant?

• A *specification*. Just as functions need them, so do loops.
What is a loop invariant?

- A **specification**. Just as functions need them, so do loops.
- Comment that describes what is always true before and after loop body, i.e., invariant

```plaintext
// inv: I
while (guard) {
  // body
  // body
}
```
What is a loop invariant?

• A **specification**. Just as functions need them, so do loops.

• Comment that describes what is always true **before** and **after** loop body, i.e., **invariant**

• States the **relationship** between the important **variables** in the loop

```java
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
What is the purpose of a loop invariant?

• Debugging:
  • Assert loop invariant like you would a class invariant

• Documentation:
  • Understand meaning of loop after it has been written

• Specification:
  • Aid in development of a new loop
  • Understand whether existing loop is correct…
Loop checklist

- Does it start right?
- Does it maintain the invariant?
- Does it end right?
- (Does it make progress?)

**Goal:** answer these questions with aid of loop invariant to understand whether loop is correct, or what might be wrong with it.
Q: Does it start right?

- Loop invariant must be true just before while
- Aka establishment or initialization
- Accomplished by assigning to loop variables to truthify invariant

```java
int count = 0;
int i = 0;
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
Q: Does it maintain the invariant?

- Body must **maintain** invariant:
  - Invariant must be true **after** body
  - In reasoning about that, we get to assume that invariant is true **before** body
  - We also get to assume guard holds before body executes

- Aka **preservation**

```java
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
Q: Does it end right?

- Loop invariant must be true just after while
- And postcondition must be established by loop invariant being true and guard becoming false

```java
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
Loop checklist

```java
int count = 0;
int i = 0;
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```

☑ Does it start right?
☑ Does it maintain the invariant?
☑ Does it end right?
❑ (Does it make progress?)
Answer, Part 2

```java
/** Returns: the frequency of `item` in `items`. Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    // inv: `count` is freq of `item` in `items[..i)`
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```

i must be in bounds to avoid exception! Add another invariant:

inv: 0 <= i <= size
Q: Does it make progress?

• Body must make **progress** toward **termination**
  • Every repetition of body must do something to get closer to making the guard false
  • Typically by strictly decreasing some quantity to zero, at which point guard becomes false
• That quantity is called the **loop variant**

```java
int count = 0;
int i = 0;
// inv: 0 <= `i` <= `size`
// variant: `size` - `i`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++; }
```
Loop checklist

```java
int count = 0;
int i = 0;
// inv: `count` is freq of `item` in `items[..i)`
// inv: 0 <= `i` <= `size`
// variant: `size` - `i`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```

- Does it start right? ✔
- Does it maintain the invariant? ✔
- Does it end right? ✔
- Does it make progress? ✔
Flowchart

Loop precondition: code before loop

Loop initialization

Invariant, guard, variant: ?

Loop guard:
- True:
  - Loop invariant is true
  - Guard is true
  - Variant is non-zero

- False:
  - Loop invariant is true
  - Guard is false
  - Variant is zero

Loop body

Loop postcondition

Code after loop
Loop invariants: are they worth the trouble?

“I really don’t understand why we are talking about loops in this fashion. It is totally unintuitive and cumbersome.”

“This ain’t it, chief.”
—Surely some anonymous CS 2110 student, Summer 2023.
Loop invariants: are they worth the trouble?

They are Cornell lore.

They are tools. Use them when needed.

They are part of next-gen *program verification* tools in research at major companies.
Demo: Dafny
Java-like language developed at Microsoft
Search Algorithms
Linear Search

**Key idea:** search linearly through array from front to back to find item

```c
/** Returns: the smallest index \( i \) such that \( a[i] == v \).
   Requires: \( v \) is in \( a \). */
int linear_search(int[] a, int v) {
    int i = 0;
    while (a[i] != v) i++;
    return i;
}
```
Exercise

State the loop invariant.

```c
/** Returns: the smallest index i such that a[i] == v.
   Requires: v is in a. */
int linear_search(int[] a, int v) {
    int i = 0;
    // inv: TODO
    while (a[i] != v) i++;
    return i;
}
```
Discovering the loop invariant

/** Returns: the smallest index i such that a[i] == v. 
Requires: v is in a. */

Post: a
\[ \begin{array}{c|c|c}
0 & i & a.length \\
\hline
\text{v not here} & v & ?
\end{array} \]

Pre: a
\[ \begin{array}{c}
0 & a.length \\
\hline
v \text{ in here}
\end{array} \]

Inv: a
\[ \begin{array}{c|c|c}
0 & i & a.length \\
\hline
\text{v not here} & v \text{ in here}
\end{array} \]

Rule: We never draw indices directly above a line in diagram! Always to left or right.

Post: v not in a[0..i] 
a[i] == v

Pre: v in a[0..]

Inv: v not in a[0..i] 
v \notin a[i..]
Discovering the loop invariant

/** Returns: the smallest index i such that a[i] == v.
Requires: v is in a. */

Discovering an invariant from the pre and post conditions requires creativity and practice.

Theorem. There is no algorithm that can do it for you.
Corollary: ChatGPT can’t replace human programmers yet!

Inv:
v not in a[0..i)
v in a[i..]
Linear Search: with invariant

/** Returns: the smallest index i such that a[i] == v.
   Requires: v is in a. */
int linear_search(int[] a, int v) {
  int i = 0;
  // inv: v not in a[0..i), and v in a[i..]
  while (a[i] != v) i++;
  return i;
}
Linear Search: loop checklist

☐ Does it start right?

☐ Does it maintain the invariant?

☐ Does it end right?

☐ Does it make progress?

/** Returns: the smallest index i such that a[i] == v. Requires: v is in a. */

int linear_search(int[] a, int v) {
    int i = 0;
    // inv: v not in a[0..i), and v in a[i..]
    while (a[i] != v) i++;
    return i;
}
Binary Search

**Key idea:** maintain upper and lower bounds on where value could be.

```c
/** Returns: an index i such that a[i] == v. 
   Requires: v is in a, and a is sorted in ascending order. */
int bin_search(int[] a, int v) {
    int l = 0;
    int r = a.length - 1;
    // inv: 0 <= l <= r < a.length, and v in a[l..r]
    while (l != r) {
        int m = (l + r) / 2;
        if (v <= a[m]) { r = m; }
        else { l = m + 1; }
    }
    return l;
}
```
Understanding the loop invariant

/** Returns: an index $i$ such that $a[i] == v$. Requires: $v$ is in $a$, and $a$ is sorted in ascending order. */

<table>
<thead>
<tr>
<th>Pre:</th>
<th>0</th>
<th>$i$</th>
<th>$a.length$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$</td>
<td>$v$ in here</td>
<td>$sorted$</td>
</tr>
</tbody>
</table>

Post: $a <= v \quad v \quad >= v$

Inv: $a <= v \quad v$ in here $\quad >= v$

$sorted$

Nothing about the loop invariant requires halving the search space!

**Efficiency** is distinct from **correctness.**
### Binary Search: loop checklist

- **Does it start right?**
- **Does it maintain the invariant?**
- **Does it end right?**
- **Does it make progress?**

```c
/** Returns: an index i such that a[i] == v.
   Requires: v is in a, and a is sorted in ascending order.
*/
int bin_search(int[] a, int v) {
    int l = 0;
    int r = a.length - 1;
    // inv: 0 <= l <= r < a.length, and v in a[l..r]
    while (l != r) {
        int m = (l + r) / 2;
        if (v <= a[m]) { r = m; }
        else { l = m + 1; }
    }
    return l;
}
```

<table>
<thead>
<tr>
<th>then</th>
<th>0</th>
<th>l</th>
<th>m</th>
<th>r</th>
<th>a.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>&lt;= v</td>
<td>...</td>
<td>&gt;= v</td>
<td>...</td>
<td>&gt;= v</td>
</tr>
<tr>
<td>sorted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>else</th>
<th>0</th>
<th>l</th>
<th>m</th>
<th>r</th>
<th>a.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>&lt;= v</td>
<td>...</td>
<td>&lt; v</td>
<td>...</td>
<td>&gt;= v</td>
</tr>
<tr>
<td>sorted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Loops (in)variants are your friends.
CS 2110
Lecture 12?

Sorting
• Selection sort
• Insertion sort
• Merge sort
• Quicksort
Why sort things?

• Makes looking things up faster
  • Binary search

• Compute robust statistics
  • Median, quantiles
  • Top-10 lists

• Prioritize/optimize
  • Search results
  • Drawing order
Why multiple algorithms?

• Tradeoffs: no one “best” algorithm
  • Speed
  • Memory
  • Expected vs. worst case
  • Stability
  • R/W locality

• You will be responsible for choosing appropriate methods

I think the bubble sort would be the wrong way to go.
Setting: arrays

• Why arrays?
  • Data to be sorted is often in an array (or ArrayList)
  • Arrays are familiar
  • Good opportunity to visualize loop invariants with array diagrams

• Implications
  • Fast to read/write arbitrary locations, iterate in reverse
  • Swaps are cheap
  • Insertions are expensive

• Most algorithms generalize to linked structures
Selection sort
Analogy: bookshelf

• Find the shortest remaining (unsorted) book
• Move it just after all the already sorted (and shorter) books

• How to “move” it?
  • Push subsequent books out of the way
    • Difficult; analogous to insertion
  • Trade places with book in desired position
    • Easy; analogous to swapping
## Selection sort example

<table>
<thead>
<tr>
<th>i=0</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selection sort invariant

Pre: $\emptyset \leq i \leq a.length$

Inv: $a[0..i]$ is sorted and $\geq a[0..i)$

Post: $a[0..i]$ is sorted
Selection sort code

// Invariant: a[0..i) is sorted, a[i..] >= a[0..i)
int i = 0;
while (i < a.length - 1) {
    // Find index of smallest element in a[i..]
    int jSmallest = i;
    for (int j = i + 1; j < a.length; ++j) {
        if (a[j] < a[jSmallest]) {
            jSmallest = j;
        }
    }
    // Swap smallest element to extend sorted portion
    swap(a, i, jSmallest);
    i += 1;
}

• Time complexity analysis
  \(N = a.length\)
  • i=0: \(N-1\) comparisons
  • i=1: \(N-2\) comparisons
  • i=2: \(N-3\) comparisons
  • ...
  • i=\(N-2\): 1 comparison
  • Total comparisons:
    \(1 + 2 + ... + (N-1)\)
    \(O(N^2)\)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best case time complexity</th>
<th>Worst case time complexity</th>
<th>Space complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Insertion sort
Analogy: a hand of playing cards

- Left hand holds cards that have already been sorted
- Take next card from right hand, insert it where it belongs in left hand
- How to “insert”
  - Push all bigger cards out of the way
  - Swap with cards to left until in position
## Insertion sort example

<table>
<thead>
<tr>
<th>i=0</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Insertion sort invariant

Pre: \( a_0 \) ? \( a_{\text{length}} \)

Inv: \( a_0 \) \( a_{\text{length}} \)

Post: \( a_0 \) \( \text{sorted} \) \( a_{\text{length}} \)
// Invariant: a[0..i) is sorted
int i = 0;
while (i < a.length) {
    // Slide a[i] to its sorted position in a[0..i]
    // Invariant: a[j] < a[j+1..i]
    int j = i;
    while (j > 0 && a[j - 1] > a[j]) {
        swap(a, j - 1, j);
        j -= 1;
    }
    i += 1;
}

• Time complexity analysis
  (N = a.length)
  • i=1: 1 comparison
  • i=2: < 2 comparisons
  • i=3: < 3 comparisons
  • ...
  • i=N-1: < N-1 comparisons

• Total comparisons (worst-case):
  1 + 2 + ... + (N-1)

  \( O(N^2) \)
Poll: Complexity if array is already sorted?

• How many comparisons does Insertion Sort evaluate if the array is already sorted?

(N is the number of elements in the array)

A. O(1) 
B. O(log N) 
C. O(N) 
D. O(N^2)
Insertion sort extras

- What if array is already sorted?
  - Each “insert” requires only 1 comparison
  - Overall complexity (best case) is $\Omega(N)

- Fast in practice for small $N$
  - Often used as a “base case” in implementations of other algs

- What if there are duplicates?
  - E.g. sorting Students by last name
  - **Stable**: relative order of equal elements is preserved

  - Insertion sort is stable because elements only move right-to-left and stop when they hit a duplicate
  - Selection sort is *not* stable because long-range swaps can change order
## Algorithm properties

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best case time complexity</th>
<th>Worst case time complexity</th>
<th>Space complexity</th>
<th>Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td></td>
<td></td>
<td></td>
<td>Unstable</td>
</tr>
<tr>
<td>Insertion sort</td>
<td></td>
<td></td>
<td></td>
<td>Stable</td>
</tr>
</tbody>
</table>
Merge sort
Merging sorted subarrays

• Given two sorted sequences, how hard is it to merge them?
  • Easy! Repeatedly take the smaller of what’s left of the two sequences
  • Complexity: $O(N)$ – easier than sorting (but requires $O(N)$ scratch space)

• What if, when tasked to sort, you outsourced the job to two assistants, who each sorted half of the list
  • Their jobs are easier (maybe much easier), since their lists are smaller
  • Your job is easier, since you only have to merge

• What if your assistants outsourced their tasks…?
Divide and conquer

• Divide task into *multiple* smaller subtasks, then assemble results into solution

• Natural fit for recursion
Merge example

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Merge sort (high level)

1. Sort left half of array (using merge sort)
2. Sort right half of array (using merge sort)
3. Merge left and right subarrays
Merge sort code

• Demo
Merge invariant

Pre: \(\begin{array}{ccc}
\text{begin} & \text{mi} & \text{end} \\
\text{a} & \text{sorted} & \text{d} & \text{sorted} \\
\end{array}\)

Inv: \(\begin{array}{cccc}
\text{a} & \text{copied} & \text{...} & \text{d} & \text{copied} & \text{...} \\
i & \text{mi} & j & \text{en} & \text{d} \\
\end{array}\)

Post: \(\begin{array}{ccc}
\text{a} & \text{copied} & \text{d} & \text{copied} & \text{...} \\
\text{O} & \text{i} & \text{mi} & \text{en} & \text{d} \\
\end{array}\)

\(\Omega\) sorted k

\(\emptyset\)

\(\emptyset\)

\(\emptyset\) sorted k

\(\emptyset\) sorted k
Analysis
## Algorithm properties

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best case time complexity</th>
<th>Worst case time complexity</th>
<th>Space complexity</th>
<th>Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td></td>
<td></td>
<td></td>
<td>Unstable</td>
</tr>
<tr>
<td>Insertion sort</td>
<td></td>
<td></td>
<td></td>
<td>Stable</td>
</tr>
<tr>
<td>Merge sort</td>
<td></td>
<td></td>
<td></td>
<td>Stable</td>
</tr>
</tbody>
</table>
Merge sort in practice

• Usually the go-to stable sort (default in many language libraries)
• Since merging is always left-to-right, can be performed on data that does not fit in RAM
Quicksort
Quicksort on one slide

\[ \text{sort}(a) = \left[ \text{sort}(a[a<p]), p, \text{sort}(a[a\geq p]) \right] \]  

1. Partition array about a “pivot”  
2. Sort the subarray of values less than the pivot  
3. Sort the subarray of values greater than the pivot

Sort via repeated partitioning  
• How efficient is partitioning?  
• How many times will you need to partition?
Partition example

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partition invariant

Pre:

\[ a \begin{array}{c} \beta \end{array} \leq p \]

Inv:

\[ a \begin{array}{cccc} n \leq p & p & ? & \geq p \end{array} \]

Post:

\[ a \begin{array}{c} n \leq p \end{array} \begin{array}{c} p \end{array} \geq p \]
Quicksort code

• Demo
Analysis

Best case
• Pivot is median value
• Each subarray is less than half the size of the original
• Depth of recursion: $O(\log N)$
  Cost of partitioning one level: $O(N)$
• Overall complexity: $\Omega(N \log N)$

Worst case
• Pivot is smallest (or largest) value
• One subarray is only 1 element shorter than original array
• Dept of recursion: $O(N)$
  Cost of partitioning one level: $O(N)$
• Overall complexity: $O(N^2)$
Choice of pivot

• Using first value is a bad choice!
  • In practice, many arrays are partially sorted
• Computing true median is not cost-effective
• Common heuristic: med3(a[begin], a[mid], a[end-1])

• Consequences of a bad pivot can be severe!
  • “Complexity attacks” to deny service
# Algorithm properties

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best case time complexity</th>
<th>Worst case time complexity</th>
<th>Space complexity</th>
<th>Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td></td>
<td></td>
<td></td>
<td>Unstable</td>
</tr>
<tr>
<td>Insertion sort</td>
<td></td>
<td></td>
<td></td>
<td>Stable</td>
</tr>
<tr>
<td>Merge sort</td>
<td></td>
<td></td>
<td></td>
<td>Stable</td>
</tr>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td>Unstable</td>
</tr>
</tbody>
</table>

* Naïve implementation requires $O(N)$ worst-case space, but can use tail recursion to reduce to $O(\log N)$. 
Quicksort in practice

• Despite poor worst-case complexity, Quicksort is often the fastest sort in practice (default unstable sort in many language libraries)
• Often augmented to detect and avoid worst-case behavior (e.g. fall back to heap sort)