Groups

Christian
Zeke
Bruce
Joonwoo
Teal
Tenzin
Kai
Zach
Lille
Henry
Alderik
Chris

Scott
Dylan
Michael Ca
Caroline
Nadani
Ben
Peter
Annika
Jackson
Bryan
Jacqueline
Tommaso

Graham
Pierce
Angel
Eliza
Alex
Nick
Will
Michael Cz

Reflection: Use group outside class. Look up master memo?

Programming Assignment!

Goals

• Create + analyze loop invariants
• Analyze QuickSort
Loop Invariants: Prove loops are correct

setup
while (condition) {
    stuff
}
Great output

Induction tailored to loops

Parts of Loop Invariant Proof

1. State Invariant: thing(s) that is true before & after each loop iteration

2. Base Case: Show invariant is true before loop starts.

3. Maintenance: Show if invariant is true before an iteration, it is true after an iteration

**Input**: Array $A$ of integers of length $n$

**Output**: Array containing sorted elements of $A$

1. **for** $k = 1$ to $n - 1$ **do**
2. 2. **for** $j = n$ to $k + 1$ **do**
3. 3. **if** $A[j] < A[j - 1]$ **then**
5. 4. **end**
6. 3. **end**
7. **end**
8. **return** $A$;

Algorithm 2: BubbleSort($A$)
Bubble Sort:

1. **Inner Loop Invariant:**
   - \( A[j] \) is the smallest element of \( A[j:n] \)
   - The elements of \( A \) are same as input array

Base case: \( j = n \), \( A[n] \) is smallest of \( A[n:n] \)


**Termination:** The loop terminates at \( j = k \), so we have
   - \( A[k] \) is smallest element of \( A[k:n] \)
   - Elements of \( A \) preserved

2. **Outer Loop Invariant:**
   - \( A[1:k-1] \) is sorted
   - \( A[1: k-1] \) contains the smallest \( k-1 \) elements of array
   - Elements of \( A \) same as input

Base case: \( k = 0 \), no elements in \( A[1:k] \), \( A \) same as input

**Maintenance:** Start of loop, \( A[1:k-1] \) is sorted and contains \( k-1 \) smallest elements of \( A \). Then inner loop moves smallest of remaining \( A[k:n] \) to \( A[k] \) while preserving elements, so now \( A[1:k] \) contains smallest \( k \) elements of \( A \), sorted.

**Termination:** At \( k = n \), so \( A[1:n-1] \) is sorted smallest elements, but there is only one remaining element in \( A[n] \), which must be the largest element. Elements are same as input, so output is sorted array.
Loop Invariant for Heapify

Max Heap:

```
// The key of each node is larger than all of its descendants
```

Before creating a heap:

In red: indices of array where heap is stored:

```
1 2 3 4 5 ...
```

```
4 1 3 2 16 ...
```

Build - Max - Heap

```for i = [A.length/2] to 1```

```
Max-Heapify(A, i)
```
**Max Heapify** \((A, i)\)

**Input:**

*Full tree NOT max heap*

**Output:**

*Max heap*

---

Prove **Build-Max-Heap** works correctly:

**Invariant:**

**Initialization**

**Mainknance**

**Termination**
Max Heapify \((A, i)\)

**Input:**

\[
\begin{array}{c}
    V \\
    \text{Max} \\
    \text{Heap}
\end{array} \quad \begin{array}{c}
    V \\
    \text{Max} \\
    \text{Heap}
\end{array} \quad \text{Full tree NOT max heap}
\]

**Output:**

\[
\begin{array}{c}
    V \\
    \text{Max} \\
    \text{Heap}
\end{array} \quad \begin{array}{c}
    V \\
    \text{Max} \\
    \text{Heap}
\end{array} \quad \text{Max heap}
\]

Prove Build-Max-Heap works correctly:

**Invariant:**

Indeces \(i+1, i+2, \ldots, n\) are roots of max heaps

**Initialization**

Indeces \([A.\text{length}/2], \ldots, n\) are leaves, and trees with one node are max heaps.

**Maintenance**

By our invariant, the children of \(i\) are roots of heaps, so Max-Heapify creates heap at \(i\). Now \(i, i+1, \ldots, n\) are roots of heaps.

**Termination**

Loop ends at \(i=1\), so indeces \(1, 2, \ldots, n\) are roots of heaps but in particular, \(1\) is a root of a heap, so the whole tree is a heap.
QuickSort Review

Key subroutine: Partition

Input: Array $A$ of length $n$, no repeated elements
Output: Array with sorted elements

QuickSort(array $A$)

1. If $|A| = 1$: return $A$
2. $\text{pivot} = \text{ChoosePivot}(A)$
3. $\text{Partition}(A, \text{pivot})$
4. $A_L \mid \mid A_R$

5. $\text{QuickSort}(A_L)$
6. $\text{QuickSort}(A_R)$
Partition \((A, p)\)  

\[ A: \begin{bmatrix} 10 & 5 & 12 & 0 & 1 & 18 \end{bmatrix} \]

- If size = 1, return
- Move pivot to start
- Loop invariant: Array looks like

### Base Case:

Each step:
- Compares current to pivot
- Does swaps to maintain invariant
- Increases current

Termination:
Runtime of QuickSort is $O(\# \text{ of comparisons})$

**Proof:** Partition does most of the work, and runtime of partition is $O(\# \text{ of comparisons})$.

Q: How many comparisons are done by Partition on input array of size $n$?

A: $O(n^2)$  B: $O(n)$  C: $O(n \log n)$  D: $O(n^2)$

Q: What is the runtime of QuickSort when the pivot is always chosen to be the $\left(\frac{n}{2}\right)$ largest element of array?

A: $O(n^2)$  B: $O(n)$  C: $O(n \log n)$  D: $O(n^2)$