Announcements

• A1 due in one week.

• Quiz 1 scores will be increased 1 pt due to vague wording in question 1(a).

• Happenings around the department:
  • Monday, 10/8 – CSCI Resume Workshop presented by Filip Jagodzinski! – 5 pm in CF 110
  • Tuesday, 10/9 – ACM Ice Cream Social – 5 pm in CF 316
  • Tuesday, 10/9 – First Whiteboard Coders Meeting – 5 pm in CF 420
  • Wednesday, 10/10 – The Game of Cybersecurity, presented by Shay Colson – 5 pm in CF 125
  • Wednesday and Thursday, 10/10 & 10/11 – Google is on Campus! Check their agenda here!
Goals:

- Thoroughly understand the mechanism of mergesort and quicksort.

- Be prepared to implement **merge** and **partition** helper methods.

- Know how to determine whether a sort is **in-place** and **stable**.
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (end-start < 2):
    return
  mid = (end-start)/2
  Divide
  mergeSort(A,start,mid)  Conquer (left)
  mergeSort(A,mid, end)   Conquer (right)
  merge(A, start, mid, end)  Combine
Quicksort

Divide

Conquer (left)

Conquer (right)

(done!)

Mergesort

Combine
```c
/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return
    mid = partition(A, st, end)
    quickSort(A, st, mid)
    quickSort(A, mid, end)
```

Unsorted:
Smaller things left
bigger things right:
Sort left things:
Sort right things:
Quicksort

Key issues:

1. Implementing `partition`

```python
/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return

    mid = partition(A, st, end)

    quickSort(A, st, mid)

    quickSort(A, mid, end)
```
/** rearrange A so all negative values are to *
* the left of all non-negative values */

```java
public void separateSign(int[] A) {

    // Precondition: A
    // Precondition: A

    // Invariant: A
    // Postcondition: A
    // Postcondition: A
}
```
/** partition A around the pivot A[pivIndex].
  * return the pivot's new index.
  * precondition: start <= pivIndex < end
  * where i is the return value */

public int partition(int[] A, int start, int end, int pivIndex) {

i

j

Pre: A p ?

Inv: A <= p p ? >= p

Post: A <= p p >= p

Four concerns:
1. Initialization
2. Termination
3. Progress
4. Maintenance
QuickSort

Key issues:

1. Implementing \texttt{partition}

2. Runtime?

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A,st,end)
  quickSort(A,st,mid)
  quickSort(A,mid, end)
```
Quicksort

/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return
    mid = partition(A, st, end)
    quickSort(A, st, mid)
    quickSort(A, mid, end)

Key issues:
1. Implementing partition
2. Runtime?
QuickSort

Key issues:

1. Implementing `partition`
2. Runtime?

```python
/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return
    mid = partition(A, st, end)
    quickSort(A, st, mid)
    quickSort(A, mid, end)
```

Runtime?

- O(1)
- O(hmm)
- O(huh?)
/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small): return
    mid = partition(A, st, end)
    quickSort(A, st, mid)
    quickSort(A, mid, end)
Quicksort

Key issues:

1. Implementing partition

2. Runtime?

3. Picking the pivot
   - First, middle, or last
   - Median of first, middle, or last

```c
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A, st, end)
  quickSort(A, st, mid)
  quickSort(A, mid, end)
```
In-Place

• Time complexity: how many operations?

• Space complexity: how much (extra) memory?

  • Usually don’t count the size of the input, because we have no choice but to store it.
In-Place

• Time complexity: how many operations?

• Space complexity: how much (extra) memory?

• Usually don’t count the size of the input, because we have no choice but to store it.

**ABCD:**
How much extra space does insertion sort use?

A. $O(1)$
B. $O(\log n)$
C. $O(n)$
D. $O(n^2)$

```java
insertionSort(A):
    i = 0;
    while i < A.length:
        j = i;
        while j > 0 and A[j] > A[j-1]:
            swap(A[j], A[j-1])
            j--
        i++
```
In-Place

A sort is considered **in-place** if it requires $O(1)$ storage space in addition to the input.

**ABC: How much extra space does insertion sort use?**

- A. $O(1)$
- B. $O(\log n)$
- C. $O(n)$
- D. $O(n^2)$

```python
insertionSort(A):
    i = 0;
    while i < A.length:
        j = i;
        while j > 0 and A[j] > A[j-1]:
            swap(A[j], A[j-1])
            j--
        i++
```
Stability

Objects can be sorted on keys - different objects may have the same value.

- e.g., sorting on first name only.

A stable sort maintains the order of distinct elements with the same key.
Stability

A **stable** sort maintains the order of elements with the same value.

Stably sorted: \([ 2^* \ 2^+ \ 3 \ 4 \ 6^* \ 6^+]\)

Unstably sorted: \([ 2^+ \ 2^* \ 3 \ 4 \ 6^* \ 6^+]\)
Stability

A **stable** sort maintains the order of elements with the same value.

In groups: determine stability of insertionSort and selectionSort

\[
[6^* 2^* 6^+ 2^+ 3 4]
\]
Stability

A **stable** sort maintains the order of elements with the same value.

**Homework**: Sort this list using insertion and selection sort. For each sort, write the state of the list after each iteration of the **outer** loop. For each sort, write whether it is stable or not.

```
[ 6* 2* 6+ 2+ 3 4 ]
```

**insertionSort(A)**:

```python
code
insertionSort(A):
    i = 0;
    while i < A.length:
        // push A[i] to its sorted position in A[0..i]
        // increment i
```

**selectionSort(A)**:

```python
code
selectionSort(A):
    i = 0;
    while i < A.length:
        // find min of A[i..A.length]
        // swap it with A[i]
        // increment i
```