Announcements

• Assignment 4: Question 1b omitted

• Assignment 5: Question 1b clarification
  /* initialize and keep track of the last processed element smaller than pivot;
    use the mid variable from lecture slides on the partition function */
int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while(first<=last) {
        assert(arr[first-1]<target && arr[last+1]>target);

        int mid = (first+last)/2;
        printf("first = %d, mid = %d, last = %d\n", first, mid, last);

        if (target==arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }

    return -1;
}
int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while(first<=last) {
        assert(arr[first-1]<target && arr[last+1]>target);

        int mid = (first+last)/2;
        printf("first = %d, mid = %d, last = %d\n", first, mid, last);

        if (target==arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }
    return -1;
}
Assignment 4: Question 1b

int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while(first<=last) {
        assert(arr[first-1]<target && arr[last+1]>target);

        int mid = (first+last)/2;
        printf("first = %d, mid = %d, last = %d\n", first, mid, last);

        if (target==arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }

    return -1;
}
Assignment 4: Question 1b

```c
int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while(first<=last) {
        assert(arr[first-1]<target && arr[last+1]>target);

        int mid = (first+last)/2;
        printf("first = %d, mid = %d, last = %d\n", first, mid, last);

        if (target==arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }
    return -1;
}
```

int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while(first<=last) {
        assert((first>0 ? arr[first-1]<target : 1) && (last<0 ? arr[last+1]>target : 1));

        int mid = (first+last)/2;
        printf("first = %d, mid = %d, last = %d\n", first, mid, last);

        if (target==arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }
    return -1;
}
?: Ternary Operator

(first>0 ? arr[first-1]<target : 1)

- if first>0, evaluates to arr[first-1]<target (left side of colon ("::"))
- Otherwise, evaluates to 1 (right side of colon)
Assignment 4: Question 1b

int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while(first<=last) {
        assert((first>0 ? arr[first-1]<target : 1) && (last<0 ? arr[last+1]>target : 1));

        int mid = (first+last)/2;
        printf("first = %d, mid = %d, last = %d\n", first, mid, last);

        if (target==arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }
    return -1;
}
Lecture 16

Today:

- Quick sort
- Introduction to Generics
- Library Sorting
Sorting by Recursion (Review)

Use Divide and Conquer to sort recursively.

1. Split the array into two roughly equal pieces.
2. Recursively sort each half.
   • This works because each piece is smaller.
3. Join the two pieces together to make one sorted array.

Two famous sorts behave this way: mergesort and quicksort.
Quick Sort

Strategy: Divide and Conquer

1. Split the array into two roughly equal pieces.
   - partition by a pivot element, x

2. Recursively sort each half.
   - two recursive calls to sort()
   - assume smaller cases are sorted correctly

3. Join the two pieces together to make one sorted array.
   - trivial
Example

pivot

15  -4  18  22  0  5  49  42  25  23  -8  -3  6

≤ 15

-4  0  5  -8  -3  6

15

≥ 15

18  22  49  42  25  23

Recursively Sort

-8  -4  -3  0  5  6

15

Recursively Sort

18  22  23  25  42  49
Quick Sort Code

// Post: arr[first..last] are sorted
void quickSort(int arr[], int first, int last) {

    /* Base case */
    if (last <= first) return;

    /* Split array into two roughly equal pieces */
    int mid = partition(arr, first, last);

    /* Recursively sort each piece */
    quickSort(arr, first, mid-1);
    quickSort(arr, mid+1, last);
}

- **Base case**
  - return if fewer than 2 elements

- **Split array into two roughly equal pieces**
  - partition around a pivot element
  - pivot element in correct position → mid

- **Recursively sort each piece**
Quick Sort Code

// Post: arr[first..last] are sorted
void quickSort(int arr[], int first, int last) {
    // Base case
    if (last <= first) return;

    // Split array
    int mid = partition(arr, first, last);

    // Recursively sort
    quickSort(arr, first, mid-1);
    quickSort(arr, mid+1, last);
}
Q. How long does it take to partition \( N \) elements?

Strategy: Compare pivot with each element
- If less than pivot, put on left piece
- If greater than pivot, put on right piece
How to develop each piece?

There are many implementations of partition.

- To make our own, visualize a partially partitioned array:

```
6 -4 0 5 -8 -3 15 42 25 23 18 22 49
```

Need two indices:
- index to scan through the array of indices (sweep)
  - marks end of the second piece
- index to mark end of the first piece (mid)

Algorithmic Strategy:
- if \( \text{arr}[\text{sweep}] > \text{pivot} \) then
  - add it to the second piece
  - (do nothing)
- if \( \text{arr}[\text{sweep}] < \text{pivot} \) then
  - add it to the first piece
  - swap with \( \text{arr}[\text{mid}+1] \)
  - \( \text{mid}++ \)

Q. What's the last step?
- Place the pivot
- swap with \( \text{arr}[\text{mid}] \)
Running Time Analysis

What’s the worst case running time?

- depends on the partition
- if it’s an even split, then $O(N \log N)$ like Merge Sort.
- Q. What if it’s a uneven split on every partition?
- $O(N^2)$ like Insertion Sort

It turns out that Quick Sort works well over all possible permutations of arrays

- $O(N \log N)$ in the average case
- Most implementations pick a random pivot
Generic Sorts

There is a function `qsort()` in `<stdlib.h>`

Parameters:

- a comparator function
- an arbitrary array of data

Remember that arrays are specified by:

- base address
- type
- number of elements

Generics promote code reuse by generalizing algorithms over different types

C++ uses the template construct to make generic typing easier
Sorting Algorithms: Summary

- For great sound effects:
  https://youtu.be/92BfxHn2XE
Sorting Algorithms: Summary

- For great sound effects: https://youtu.be/8oJS1BMKE64
Sorting Algorithms: Summary

- For great sound effects:
  https://youtu.be/ZRPoEKHXTJg
Sorting Algorithms: Summary

- For great sound effects:
  https://youtu.be/9IqV6ZSjual
Sorting Algorithms: Summary

- [https://youtu.be/ZZuD6iUe3Pc](https://youtu.be/ZZuD6iUe3Pc)