CS 32 Week 7 Discussion 1B

this week's topics:
big-O, sorting, midterm review

TA: **Yiyou Chen** / LA: **Ian Galvez**

today’s agenda

- Review of Big-O / Sorting Algorithms (20 min)
- Selected LA Worksheet Problems (30-40 min)
- Break (10 min)
- Kahoot! Time (30-40 min)
- Project 3 Workshop (Rest of class)

big-O, algorithms, sorting *review!*

worksheet practice problems!

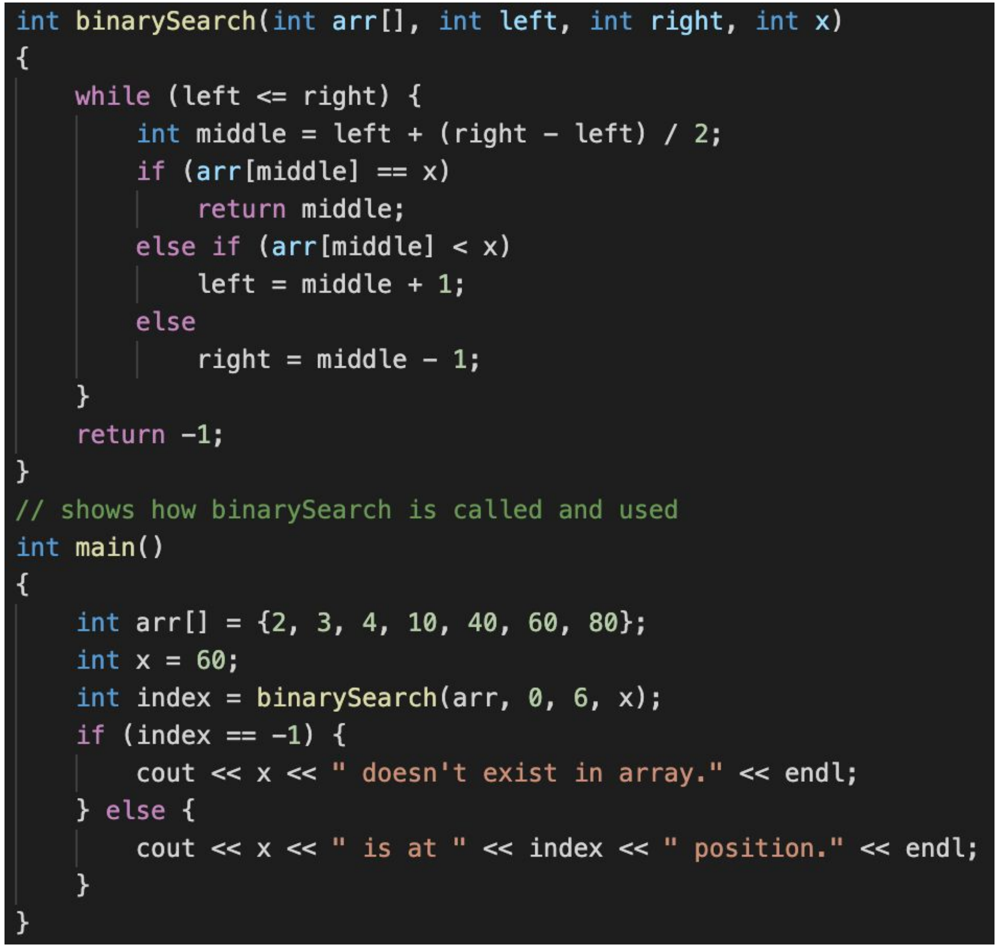
a couple things before we start…

- ***Please find a partner / form a group!***
- We'll be doing practice problems throughout the class.
- ***You should go to TA/LA office hours*** if you're ever having any questions about projects/homeworks! There are usually lots of other students in the Boelter room that may be stuck on the same thing, and we'll be able to help you and you'll be able to help each other!
- Discussion section is more for reviewing and practicing concepts that were gone over in class (i.e. doing worksheet practice problems, reviewing for the midterm, etc.)

I Sawed This Array in Half

(taken from Wk 7 LA Worksheet, Problem 3)

What's the time complexity of *binarySearch*?



```
int binarySearch(int arr[], int left, int right, int x)
{
    while (left <= right) {
        int middle = left + (right - left) / 2;
        if (arr[middle] == x)
            return middle;
        else if (arr[middle] < x)
            left = middle + 1;
        else
            right = middle - 1;
    }
    return -1;
}
// shows how binarySearch is called and used
int main()
{
    int arr[] = {2, 3, 4, 10, 40, 60, 80};
    int x = 60;
    int index = binarySearch(arr, 0, 6, x);
    if (index == -1) {
        cout << x << " doesn't exist in array." << endl;
    } else {
        cout << x << " is at " << index << " position." << endl;
    }
}
```

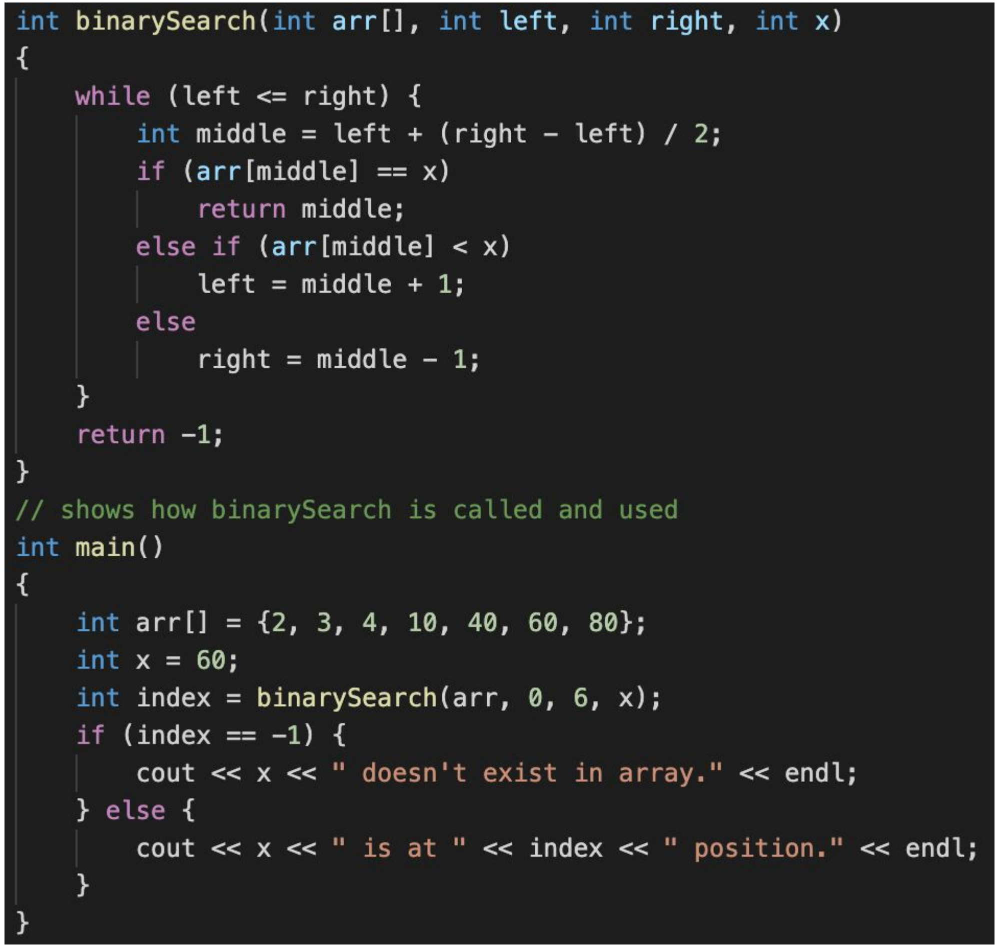
solution

(to I Sawed This Array In Half)

The time complexity of *binarySearch* is *O(log n)*.

Binary search halves the search interval after every iteration; it either traverses through the right half of the original interval or the left half.

- At the start, left = 0 and right = length of array - 1
- Iteration stops when left > right
- If the exact match is not found, either left or right will be assigned value of middle
- So next time in iteration, the new left and right pair interval is half of the original interval

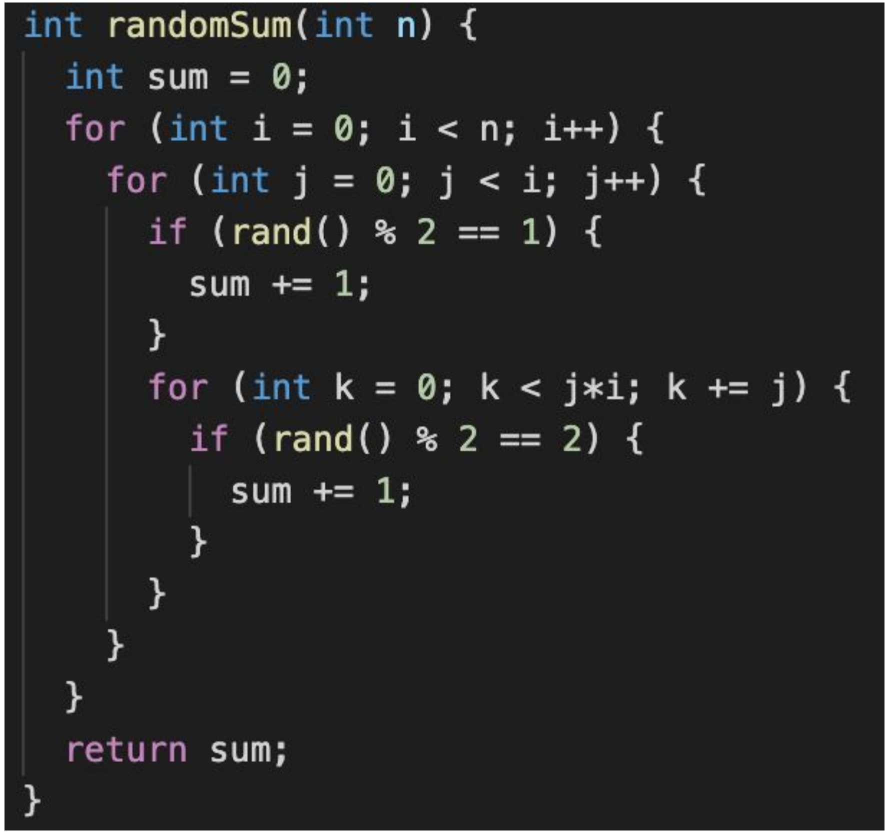


```cpp
int binarySearch(int arr[], int left, int right, int x)
{
    while (left <= right) {
        int middle = left + (right - left) / 2;
        if (arr[middle] == x)
            return middle;
        else if (arr[middle] < x)
            left = middle + 1;
        else
            right = middle - 1;
    }
    return -1;
}
// shows how binarySearch is called and used
int main()
{
    int arr[] = {2, 3, 4, 10, 40, 60, 80};
    int x = 60;
    int index = binarySearch(arr, 0, 6, x);
    if (index == -1) {
        cout << x << " doesn't exist in array." << endl;
    } else {
        cout << x << " is at " << index << " position." << endl;
    }
}
```

it do be like that Sum Times

(taken from Wk 7 LA Worksheet, Problem 1)

What's the time complexity of *randomSum*?



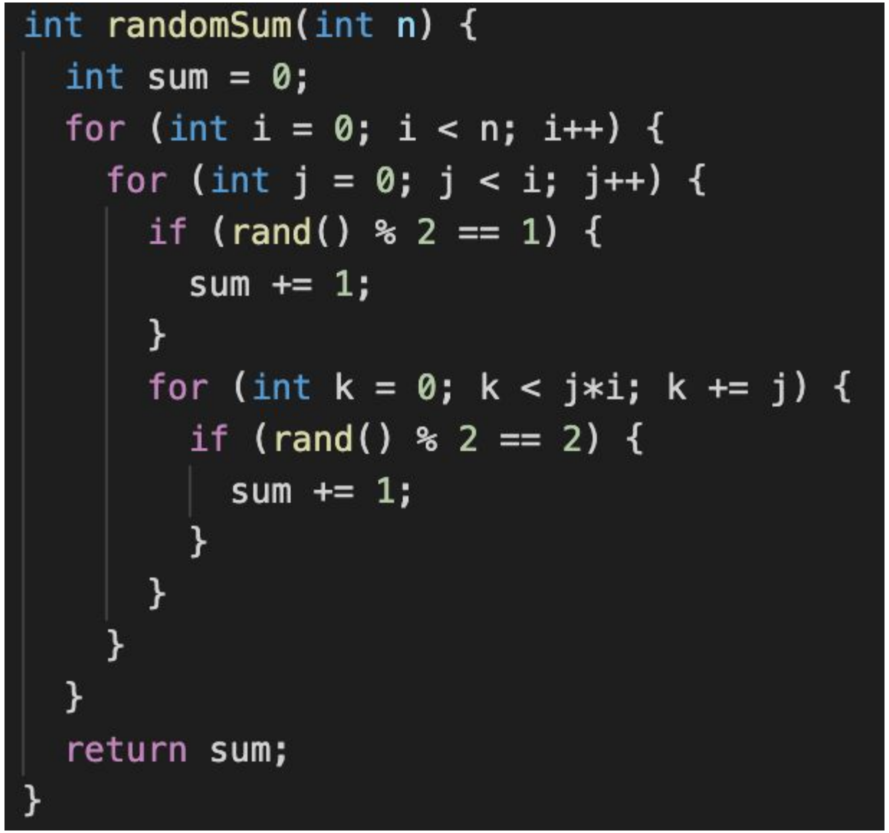
```
int randomSum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      if (rand() % 2 == 1) {
        sum += 1;
      }
      for (int k = 0; k < j*i; k += j) {
        if (rand() % 2 == 2) {
          sum += 1;
        }
      }
    }
  }
  return sum;
}
```

solution

(to it do be like that Sum Times)

The time complexity of *randomSum* is *O(n^3)*.

- The inner loop is ***O(i)***
  - Even though ***rand() % 2 == 2*** is always false, we still call ***rand()*** every iteration.
- The middle loop is ***O(i^2)***
  - It runs the inner loop ***i*** times,
  - So ***i * i = i^2***
- The outer loop (and thus the function) is ***O(n^3)***.
  - When i = 1, the middle loop runs 1^2 times.
  - When i = 2, the middle loop runs 2^2 times.
  - Then, the middle loop runs 1^2 + 2^2 + ... + (n-1)^2 times, or (n-1)(n)(2n-1)/6 times, which is on the order of ***O(n^3)***

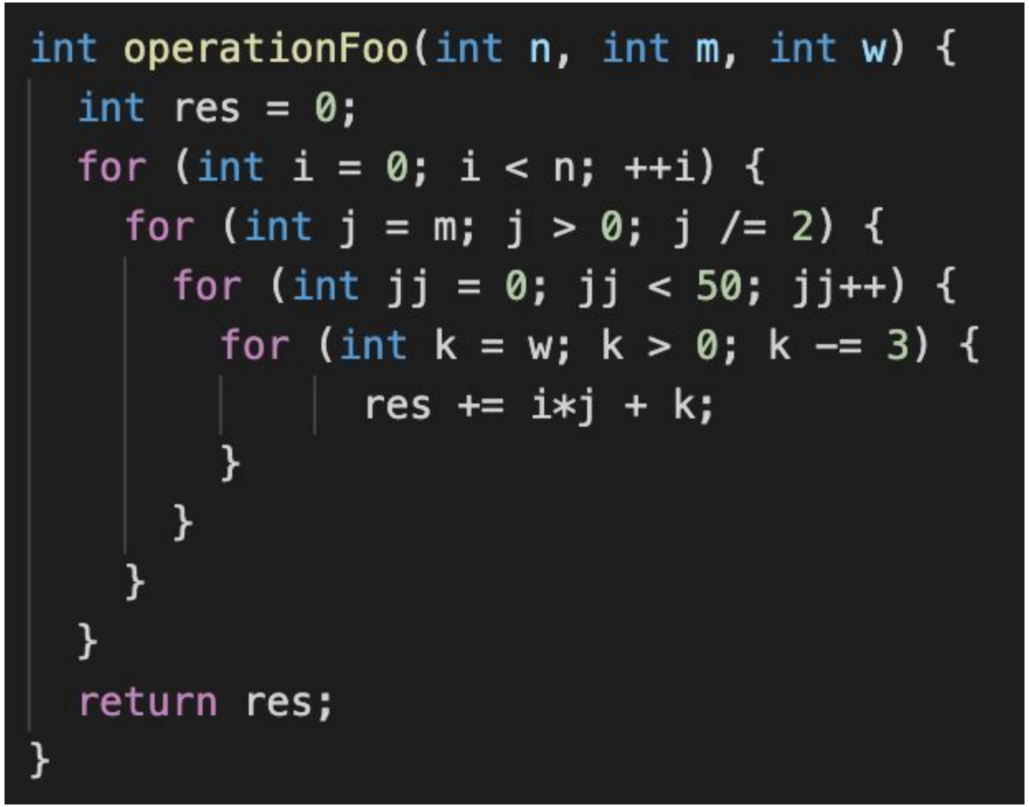


```
int randomSum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      if (rand() % 2 == 1) {
        sum += 1;
      }
      for (int k = 0; k < j*i; k += j) {
        if (rand() % 2 == 2) {
          sum += 1;
        }
      }
    }
  }
  return sum;
}
```

Foo Loops

(taken from Wk 7 LA Worksheet, Problem 2)

What's the time complexity of *randomSum*?



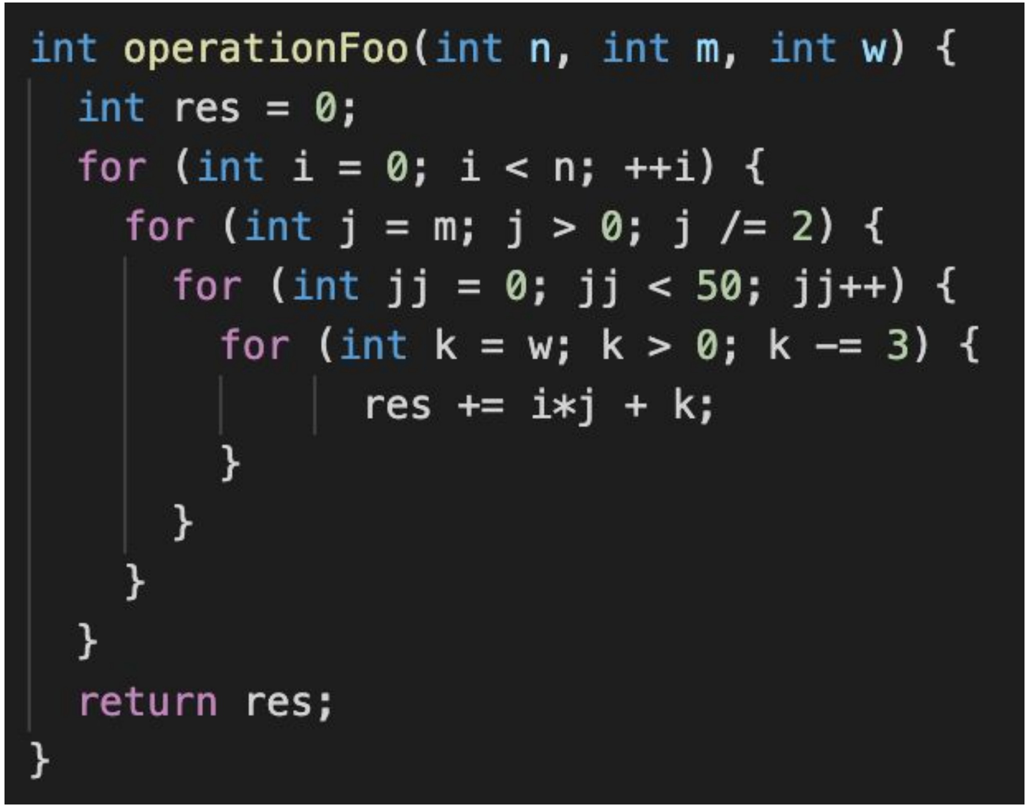
```
int operationFoo(int n, int m, int w) {
  int res = 0;
  for (int i = 0; i < n; ++i) {
    for (int j = m; j > 0; j /= 2) {
      for (int jj = 0; jj < 50; jj++) {
        for (int k = w; k > 0; k -= 3) {
            res += i*j + k;
        }
      }
    }
  }
  return res;
}
```

solution

(to Foo Loops)

The time complexity of *operationFoo* is *O(n * log(m) * w).*

- The outermost loop runs ***n*** iterations
- The 2nd loop runs ***$log_2(m)$*** iterations
- The 3rd loop runs ***50*** iterations, which is constant, or ***O(1)***
- The innermost runs ***1/3 * w*** iterations, or ***O(w)***
- The ***res += i*j + k*** line runs on the order of ***n*log(m)*w*** times



```
int operationFoo(int n, int m, int w) {
  int res = 0;
  for (int i = 0; i < n; ++i) {
    for (int j = m; j > 0; j /= 2) {
      for (int jj = 0; jj < 50; jj++) {
        for (int k = w; k > 0; k -= 3) {
            res += i*j + k;
        }
      }
    }
  }
  return res;
}
```

Complexity Time

(taken from Wk 7 LA Worksheet, Problem 6)

Fill out the following table with the correct time complexities:

Time complexity	**Doubly linked list (<u>given head, unless noted otherwise</u>)**	**Array / vector**
Inserting an element to the beginning		
Inserting an element to some position i		
Getting the value of an element at position i		
Changing the value of an element at position i		
Deleting an element given a reference to it		

solution

(to Complexity Time)

In general, linked lists are better at inserting and deleting elements at the beginning and end, since you don't need to shift each element.

In general, vectors are better at finding an element anywhere in the list, since you just access using an index.

Time complexity	Doubly linked list (given head, unless noted otherwise)	Array / vector
Inserting an element to the beginning	***O(1)***	***O(n)***
Inserting an element to some position i	***O(n)***	***O(n)***
Getting the value of an element at position i	***O(n)***	***O(1)***
Changing the value of an element at position i	***O(n)***	***O(1)***
Deleting an element given a reference to it	***O(1)***	***O(n)***

An Example Problem, of Sorts

(taken from Wk 7 LA Worksheet, Problem 9)

Yiyou used a mystery sorting algorithm to sort an array. After each pass through the array, his program printed out the elements of said array. What was the mystery sorting algorithm he used?

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

solution

(to an Example Problem, of Sorts)

This is insertion sort! Notice that the section to the left of the underlined digit is in sorted order. One of the most prominent insertion sort characteristics is having a section that is already in sorted order, and continuing to put the current element in the right place in the sorted section.

$\underline{3}$ 7 4 9 5 2 6 1

3 $\underline{7}$ 4 9 5 2 6 1

3 **7** $\underline{4}$ 9 5 2 6 1

3 **4** 7 $\underline{9}$ 5 2 6 1

3 4 7 **9** $\underline{5}$ 2 6 1

3 4 **5** 7 9 $\underline{2}$ 6 1

2 3 4 5 7 9 $\underline{6}$ 1

2 3 4 5 **6** 7 9 $\underline{1}$

1 2 3 4 5 6 7 9

break time!

feel free to ask any questions

midterm review
it’s time for a kahoot!

thank you all for coming!

the rest of the time will be for project/midterm questions