CSE 12 Week 5 Discussion

2-2-21

Focus: Sorting

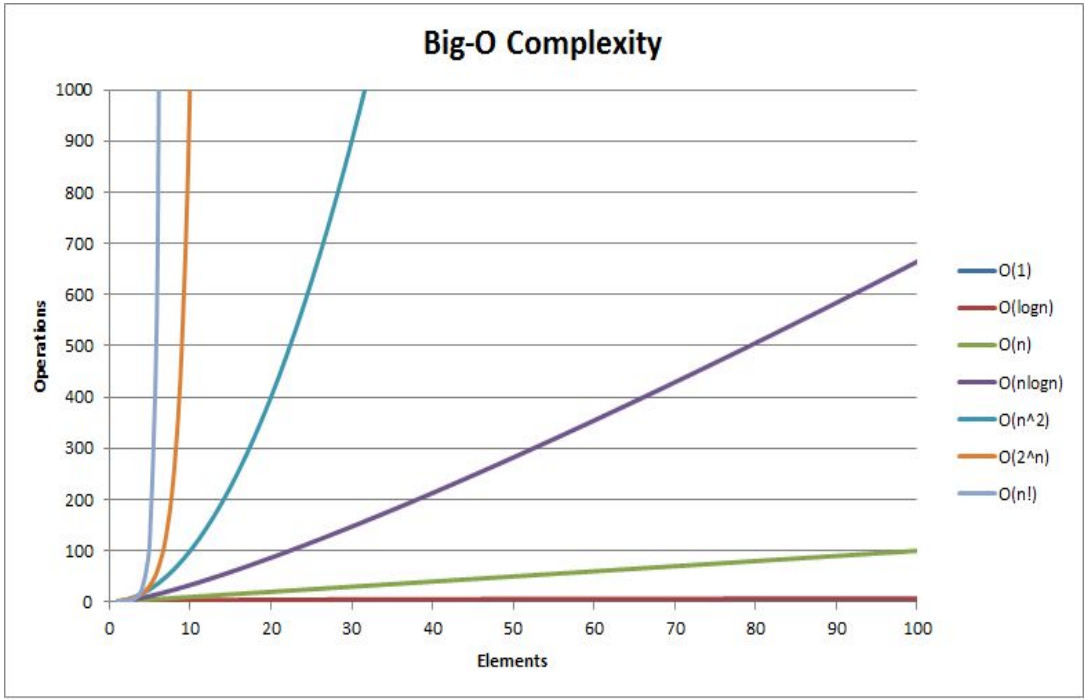
Reminders

- PA5 is a **closed** assignment - no collaborating!

- PA2 Resubmission due Friday, February 5th 11:59 PM
- PA3 Resubmission due Friday, February 12th 11:59 PM

Sorting

Big-O review



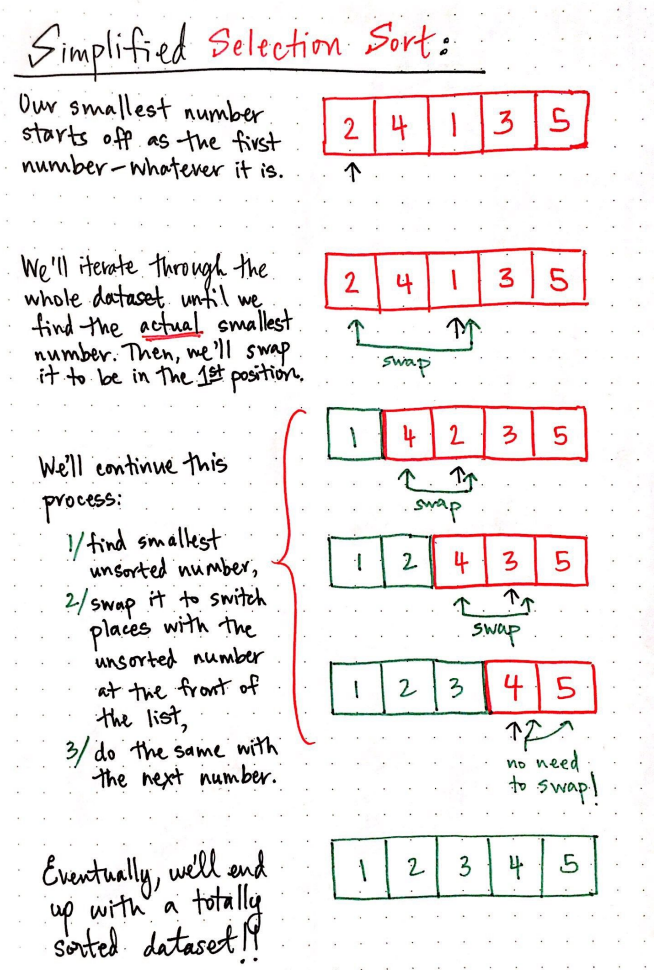
- Relative to input n
- Constants do not matter
  - O(3n)=O(n)
- Higher order values dominate
  - O(n^2 + n) = O(n^2)

Algorithms review

Which algorithm finds the minimum element in a list and moves it to the end of a sorted prefix in the list?

a. Selection sort
b. Insertion sort
c. insert()
d. transform()

Answer - A (selection sort)



Selection sort

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the worst case runtime for selection sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. o(n!)
e. None of the above

Answer - A

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

In worst case, outer loop is order of O(n) and inner loop is the order of O(n).

O(n) * O(n) = O(n^2)

Selection sort

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the best case runtime for selection sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - A

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

Even in best case, outer loop runs through all iterations while going through entire inner loop each time

O(n) * O(n) = O(n^2)

Selection sort

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

How can we optimize selection sort?

a. Start inner loop at j=0 instead of j=i
b. Swap elements inside the inner loop
c. Make outer loop bound arr.length-1 instead of arr.length
d. Start outer loop at i=1 instead of i=0
e. Cannot be optimized any further for correct solution

Answer - C

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length-1; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

Last element (arr[arr.length-1]) must already be in correct place, so cut down algorithm by 1 iteration

Cannot compare values with other indices because at end of list anyways

Selection sort

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length-1; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

By changing outer loop bound to arr.length-1, what is the new worst case runtime of selection sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - A

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length-1; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

No change in runtime (still O(n^2 for worst and best cases)

Reducing outer loop by 1 iteration results in O(n-1) for outer loop, which is same as O(n)

Inner loop still runs O(n)

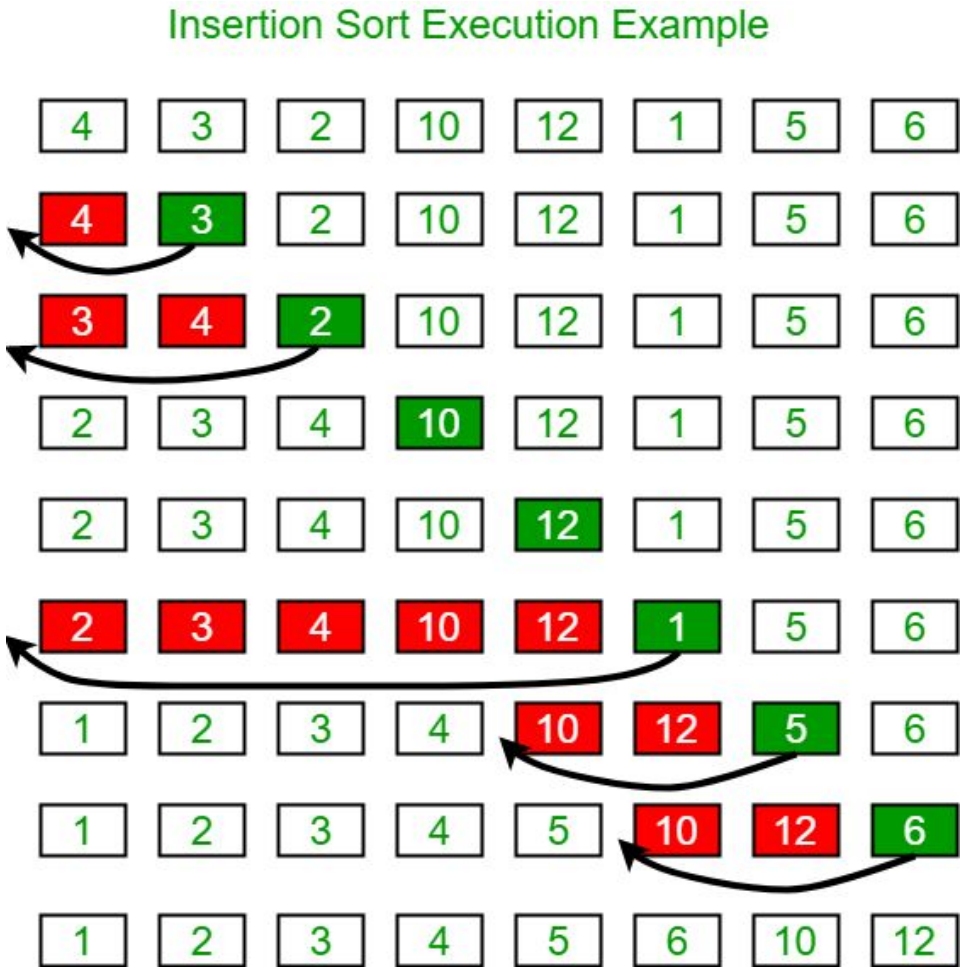
O(n) * O(n) = O(n^2)

Algorithms Review

Which algorithm repeatedly takes the next element in a list inserts it into the correct ordered position within a sorted prefix of the list?

a. Selection sort
b. Insertion sort
c. insert()
d. transform()

Answer - B (insertion sort)



Insertion sort

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the worst case runtime for insertion sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - A

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

In worst case, outer loop is order of O(n) and inner loop is the order of O(n).

O(n) * O(n) = O(n^2)

Insertion sort

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the best case runtime for insertion sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - A

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

Even in best case, outer loop runs through all iterations while going through entire inner loop each time

O(n) * O(n) = O(n^2)

Insertion sort

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

How can we optimize insertion sort?

a. Compare arr[j] to arr[j-2] also
b. Break out of inner loop if arr[j] is >= arr[j-1]
c. Make outer loop bound arr.length-1 instead of arr.length
d. Break out of outer loop if arr[j] is >= arr[j-1]
e. Cannot be optimized any further for correct solution

Answer - B

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

Once we see that elements are in the right order, we do not need to look at earlier elements since they will already be ordered

Insertion sort

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

By adding the else statement with a break, what is the new worst case runtime of insertion sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - A

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

No change in runtime (still O(n^2) for worst case)

Will not call break statement in worst case because entire list is out of order

Inner and outer loops each run O(n)

O(n) * O(n) = O(n^2)

Insertion sort

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

By adding the else statement with a break, what is the new best case runtime of insertion sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - B

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

Best case: sorted list

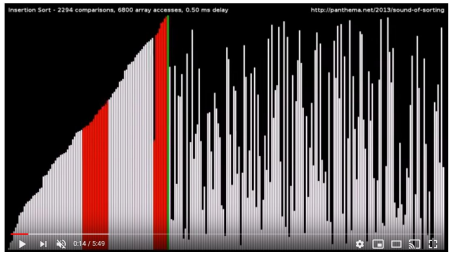
As a result, will call break statement each time inner loop is entered

Inner loop runs O(1)

Outer loop still runs O(n)

O(n) * O(1) = O(n)

Sorting Algorithm Visualizations



WARNING: flashing lights

https://www.youtube.com/watch?v=kPRA0W1kECg

If you do not have a hearing sensitivity, we recommend having the volume on during these visualizations. Lower pitches correspond to operations on smaller bars/values, and higher pitches correspond to operations on larger bars/values. Check out the top left corner for the current sort method and number of certain operations.

The first 4 sort methods are the 4 you've learned so far: insertion, selection, quick, and merge sort.

Note on bogo sort: this is a bit of a joke in CS. Bogo sort works by randomly moving all of the values until it finds the sorted solution by chance; it is the last sort method in the video. It has run time upper bound: O((n+1)!)