CSE 12 Week 8 Discussion

5-21-21

Focus: Exam 2 Review

Reminders

- PA7 **(closed)** due Tuesday, May 26th @ 11:59 PM

- PA5 Resubmission due TODAY @ 11:59 PM
- PA6 Resubmission due Friday, May 28th @ 11:59 PM

Questions?

Midterm 2 Concepts

- Time Complexities
- Sorting
- Maps and HashTables

Focus on **PAs**, **lecture**, and Quizzes

Go over Review Slides?

- Yes
- No

Runtime Analysis

Common Time Complexities in Order

ordering of functions from slowest-growing (indeed, the first two *shrink* as n increases) to fastest-growing that you might find helpful:

- $f(n) = 1/(n^2)$
- $f(n) = 1/n$
- $f(n) = 1$
- $f(n) = \log(n)$
- $f(n) = \text{sqrt}(n)$
- $f(n) = n$
- $f(n) = n^2$
- $f(n) = n^3$
- $f(n) = n^4$
- … and so on for constant polynomials …
- $f(n) = 2^n$
- $f(n) = n!$
- $f(n) = n^n$

Summary

Big-O

- **Upper bound** on a function
- f(n) = O(g(n)) means that we can expect f(n) will always be **under** the bound g(n)
  - But we don’t count n up to some starting point n_0
  - And we can “cheat” a little bit by moving g(n) up by multiplying by some constant c

Big-Ω

- **Lower bound** on a function
- f(n) = Ω(g(n)) means that we can expect f(n) will always be **over** the bound g(n)
  - But we don’t count n up to some starting point n_0
  - And we can “cheat” a little bit by moving g(n) down by multiplying by some constant c

Big-θ

- **Tight bound** on a function.
- If f(n) = O(g(n)) *and* f(n) = Ω(g(n)), then f(n) = θ(g(n)).
- Basically it means that f(n) and g(n) are interchangeable
- Examples:
  - $3n+20 = \theta(10n+7)$
  - $5n^2 + 50n + 3 = \theta(5n^2 + 100)$

Review Question

Which is the upper bound in this statement?

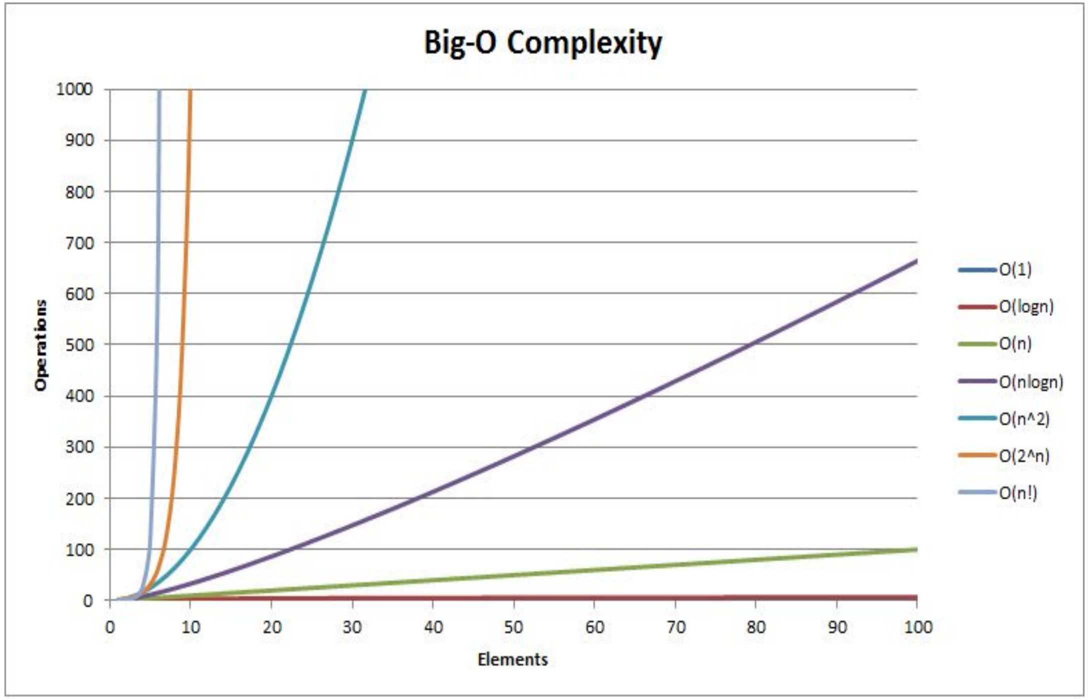
$$g(n) = O(\boxed{f(n)})$$

Review Question

Which is the upper bound in this statement?

$$\boxed{g(n)} = \Omega(f(n))$$

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)

True or False?

$n + 5n^3 + 8n^4 = O(n)$

a) True
b) False

True or False?

$n + 5n^3 + 8n^4 = O(n)$

a) True
[✓ b) False]

True or False?

n! + n^2 = O(nlog(n))

a) True
b) False

True or False?

$n! + n^2 = O(n\log(n))$

a) True

b) False

True or False?

$2^n + n\log(n) = O(n!)$

a) True
b) False

True or False?

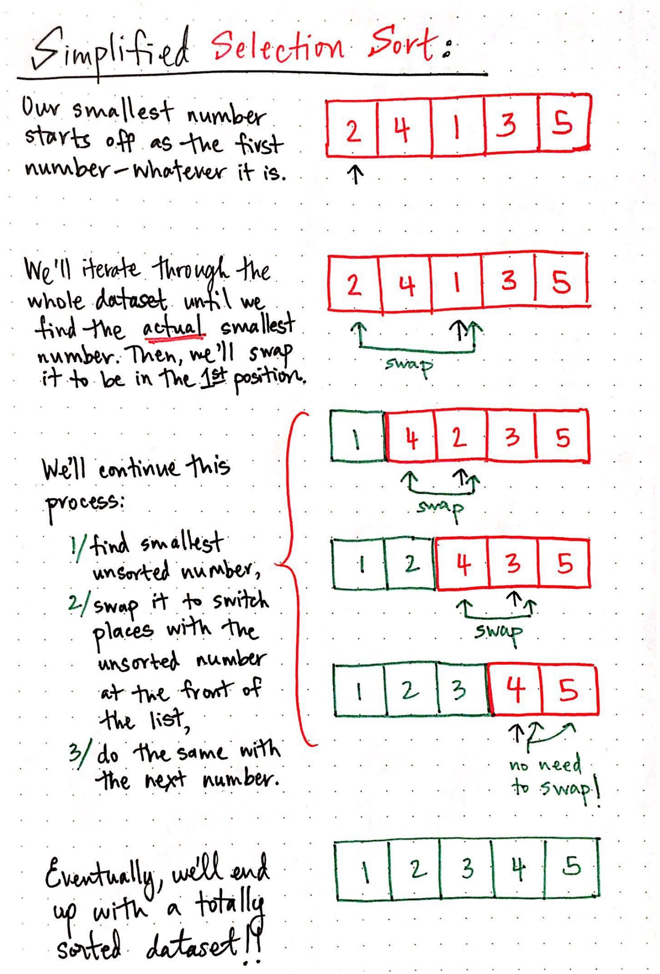
$2^n + n\log(n) = O(n!)$

a) True
b) False

Sorting

Sorting Algorithms

- Selection Sort: finds the minimum element in a list and moves it to the end of a sorted prefix in the list
- Insertion Sort: repeatedly takes the next element in a list, inserts it into the correct ordered position within a sorted prefix of the list
- Quicksort: picks an element as pivot and partitions the given array around the picked pivot
- Merge sort: recursively sorts half of the array



Simplified Selection Sort:

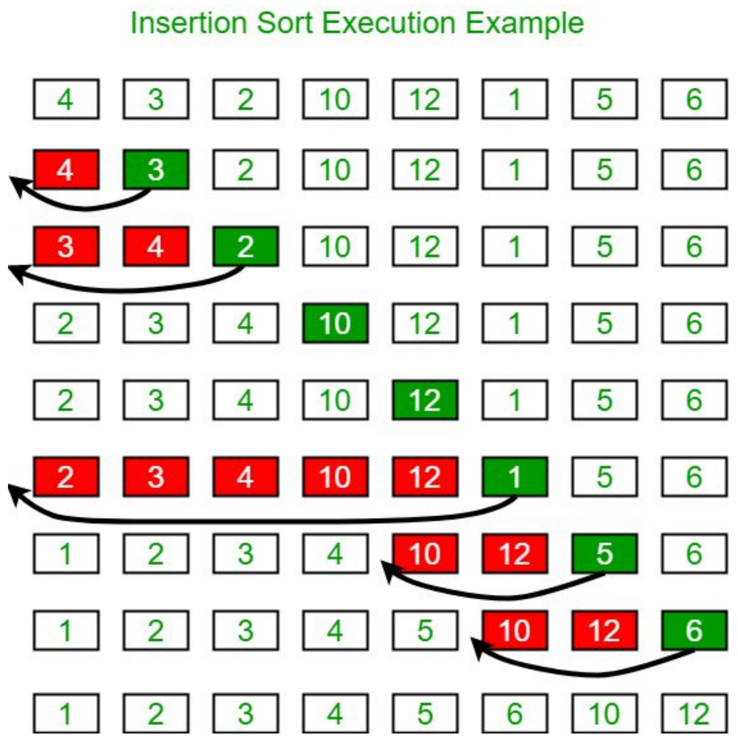
Our smallest number starts off as the first number – whatever it is.

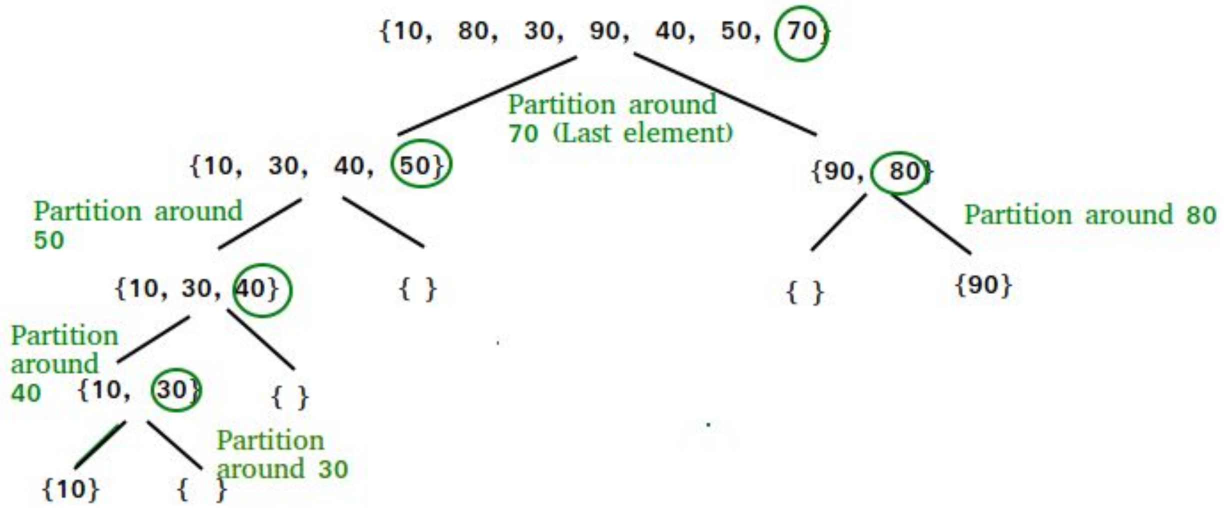
We'll iterate through the whole dataset until we find the actual smallest number. Then, we'll swap it to be in the 1st position.

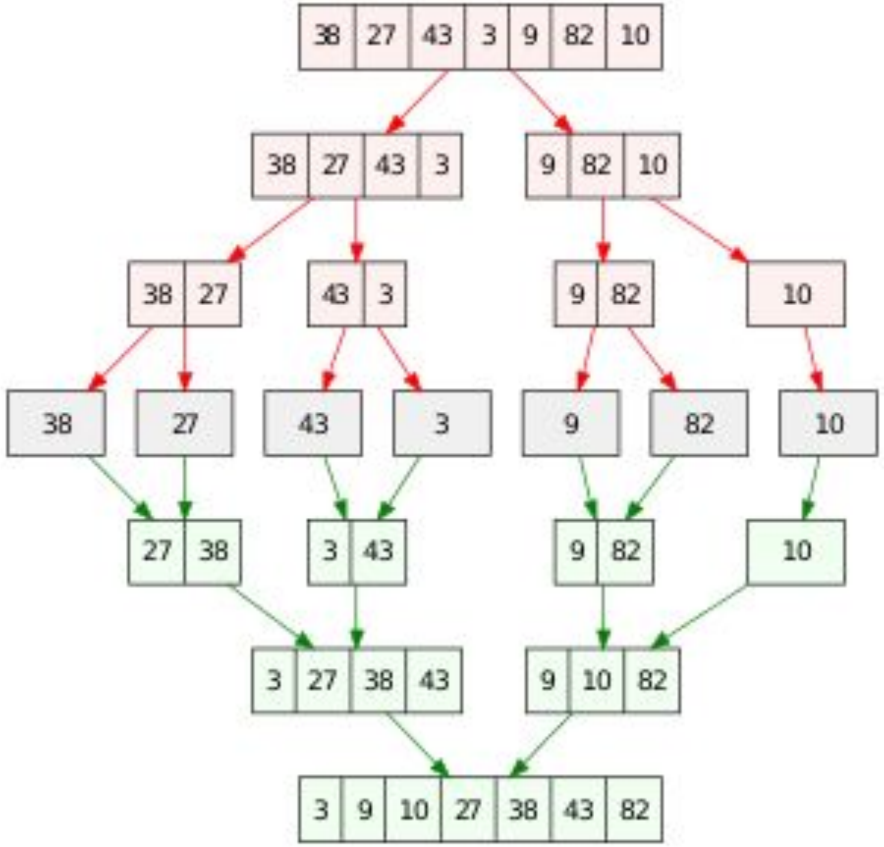
We'll continue this process:

1/ find smallest unsorted number,
2/ swap it to switch places with the unsorted number at the front of the list,
3/ do the same with the next number.

Eventually, we'll end up with a totally sorted dataset!!



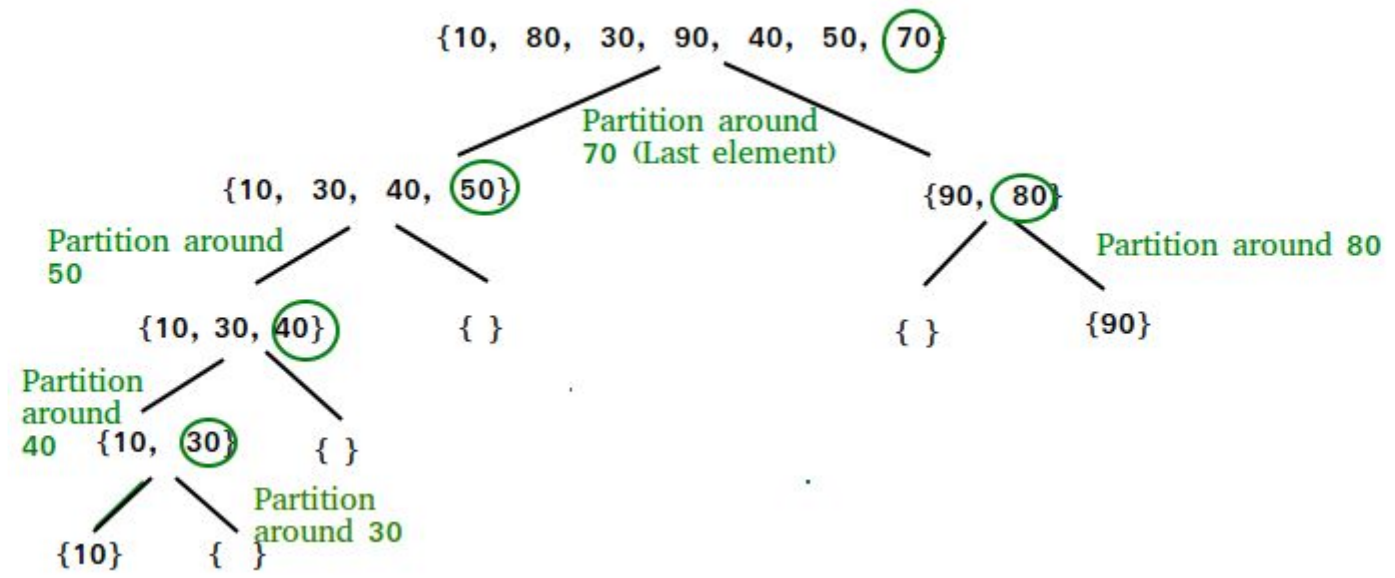

{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

What is a partition in quicksort?

- Partitioning is a component in quicksort and an algorithm that is called again and again until the original array elements are sorted as single-element arrays
- A pivot index is selected (for example, pick the last element), then the array is partitioned in such a way that the input array has its elements moved around so that element values less than or equal to the pivot index's value are to the left of the pivot index and element values greater than or equal to the pivot index's value are to the right
    - **Note:** the array is not necessarily completely sorted here (and probably isn't)!
- Now the array is divided into two at the point where the pivot index ended up
- Partition is called on each of these two subarrays using the "same" pivot point location (we picked the last element and do so again for each of these subarrays)
- We partition again, and this process is repeated until we have single-element arrays that are sorted between themselves; **see visualization in next slide**

Graphic visualization of quicksort



Resultant single-element arrays to combine: `{{10}, {30}, {40}, {50}, {70}, {80}, {90}}`

Review Quiz Questions

Which of the following will result in the most number of element comparisons using selection sort? Select all that apply:

A. {1, 2, 3, 4, 5}
B. {5, 4, 3, 2, 1}
C. {1, 3, 5, 2, 4}
D. {1, 4, 5, 3, 2}
E. {1, 1, 1, 1, 1}

Review Quiz Questions

Which of the following will result in the most number of element comparisons using selection sort? Select all that apply:

A. {1, 2, 3, 4, 5}
B. {5, 4, 3, 2, 1}
C. {1, 3, 5, 2, 4}
D. {1, 4, 5, 3, 2}
E. {1, 1, 1, 1, 1}

All of them!

Which of the following descriptions of pivot selection will result in the best case quicksort runtime?

A. Randomly choosing the pivot
B. Choosing the first value as the pivot
C. Choosing the median index as the pivot
D. Choosing the median value as the pivot
E. There is no definite pivot selection method that will always result in best case runtime

Which of the following descriptions of pivot selection will result in the best case quicksort runtime?

A. Randomly choosing the pivot
B. Choosing the first value as the pivot
C. Choosing the median index as the pivot
D. Choosing the median value as the pivot
E. There is no definite pivot selection method that will always result in best case runtime

Maps and HashTables

Maps

Assign a **key** to each **value** we are trying to keep track of.

Key 1 ---> Some value a

Key 2 ---> Some value b

Key 3 ---> Some value c

etc...

Map<K,V> Interface

- Implemented in Java by AbstractMap, HashMap, TreeMap etc.
- Index for entry is determined by a hash function that calculates index using key value (useful for quick lookup and insert)
- Contains methods get(Object key), put(K key, V value), size(), replace(K key, V value) etc.
- ***Keys need to be unique***
- Existing data structures we can use to implement this - ArrayList!

Hash Functions

```
int hash1(String s) {
  return s.length();
}
```

```
int hash2(String s) {
  int hash = 0;
  for(int i = 0; i < s.length(); i += 1) {
    hash += Character.codePointAt(s, i);
  }
  return hash;
}
```

```
public int hash3(String s) {
  int h = 0;
  for (int i = 0; i < s.length(); i++) {
    h = 31 * h + Character.codePointAt(s, i);
  }
  return h;
}
```

```
int hash(char key) {
   return (int) key;
}
```

Which of the following sequences of insertions would cause the most collisions for a hash table with **four** buckets and assuming expandCapacity is not called during the adds?

A. add('A', 56); add('B', 5); add('C', 65); add('D', 2);
B. add('E', 43); add('F', 7); add('K', 6); add('L', 160);
C. add('M', 58); add('Q', 14); add('U', 20); add('W', 37);
D. add('N', 7); add('R', 24); add('V', 92); add('Z', 100);
E. add('Z', 91); add('R', 604); add('P', 9); add('L', 5);

```
int hash(char key) {
    return (int) key;
}
```

Which of the following sequences of insertions would cause the most collisions for a hash table with **four** buckets and assuming expandCapacity is not called during the adds?

A. add('A', 56); add('B', 5); add('C', 65); add('D', 2);
B. add('E', 43); add('F', 7); add('K', 6); add('L', 160);
C. add('M', 58); add('Q', 14); add('U', 20); add('W', 37);
D. add('N', 7); add('R', 24); add('V', 92); add('Z', 100);
E. add('Z', 91); add('R', 604); add('P', 9); add('L', 5);

```
int hash(String key) {
    return key.length;
}

/*
 ----------
|          |
|          -
|          |
 ----------
|          |
|          - {"greetings" : 6}
|          |
 ----------
|          |
|          - {"hi" : 5}
|          |
 ----------
|          |
|          - {"bye" : 9}
|          |
 ----------
|          |
|          - {"happy week 7" : 3}
|          |
```

After expandCapacity is called, which of the following elements will have a different index in the new array after rehashing?

- {"greetings" : 6}
- {"hi" : 5}
- {"bye" : 9}
- {"happy week 7" : 3}
- {"hello" : 2}

```
int hash(String key) {
   return key.length;
}

/*
 ----------
|         |
|         -
|         |
 ----------
|         |
|         - {"greetings" : 6}
|         |
 ----------
|         |
|         - {"hi" : 5}
|         |
 ----------
|         |
|         - {"bye" : 9}
|         |
 ----------
|         |
|         - {"happy week 7" : 3}
|         |
```

After expandCapacity is called, which of the following elements will have a different index in the new array after rehashing?

- [✓ {"greetings" : 6}]
- {"hi" : 5}
- {"bye" : 9}
- [✓ {"happy week 7" : 3}]
- {"hello" : 2}