CS32 Dis 1F Week 7

Spring 2021

TA: Manoj Reddy

LA: Katherine Zhang

Outline

- Complexity Analysis
- Sorting

Big O

- Motivation: Measure the complexity of an algorithm
- Complexity:
  - Time: Number of operations
  - Space: Memory (RAM)
- Simply comparing the run-time is NOT useful (???)
- 2 reasons:
  - Different computer speeds
  - Must be a function of the size of the input data
- Big-O measures an algorithm by the gross number of steps it requires to process an input of size N in the WORST CASE scenario

Big-O (contd…)

- Measures the number of operations:
  - Accessing an item (e.g. an item in an array)
  - Evaluating a mathematical expression
  - Traversing a single link in a linked list

```
int arr[n][n];

for ( int i = 0; i < n; i++ )
   for ( int j = 0; j < n; j++ )
       arr[i][j] = 0;
```

- Algorithm is $O(n^2)$
- Only the most significant term is used

(e.g. n^2 from $2n^2+3n+1$)

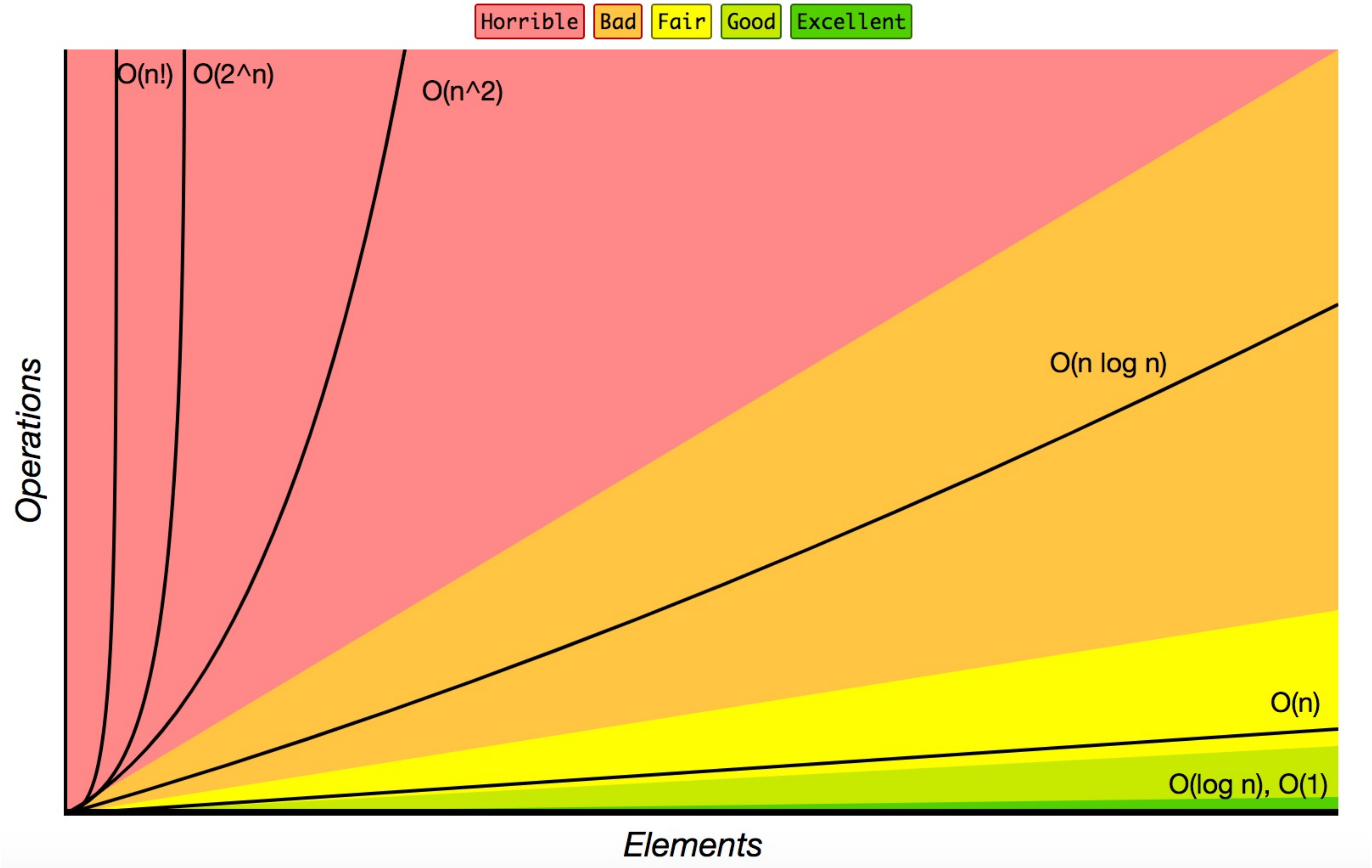
Calculating Big-O

- Step 1:
  - Locate all loops that don't run for a fixed number of iterations and determine the maximum number of iterations each loop could run for.
- Step 2:
  - Turn these loops into loops with a fixed number of iterations, using their maximum possible iteration count.
- Step 3:
  - Finally, do your Big-O analysis.
- For multi-input algorithms, include each independent variable ex: O(m+n)

Ex: Checking if 2 linked lists are of same length

```
void func1(int n)
{
  for ( int i = 0; i < n; i++ )
    for (int j=0; j < i ;j++)
      cout << j;
}
```

$O(n^2)$



http://bigocheatsheet.com/

Data Structure					Space Complexity
	Worst				Worst
	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked Lis	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked L	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tr	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(n)	O(n)	O(n)	O(n)	O(n)

http://bigocheatsheet.com/

Sorting

- Process of ordering a bunch of items based on a comparison operator
- Many sorting algorithms each with different space/time complexities
- Sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted
- Comparison-based sort cannot perform better than *n*log(n)*
  - Proof: Search for “Comparison-based Lower Bounds for Sorting CMU” on Google
  - Lecture notes by Prof. Avrim Blum

Selection Sort

- Select the smallest item from array and swap it with the first position
- Find the next smallest item and swap it in the 2^{nd} position
- And so on…
- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$
- Not Stable Sorting Algorithm
- Same number of comparisons regardless of input

Insertion Sort

- Inspired from playing cards
- Start from second item
- Insert it into the right place
- Continue doing the same with items to the right
- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$
- Stable sort
- Number of comparisons depends on input

Bubble Sort

- Largest value '*bubbles*' to the top
- Compare the first 2 elements: A[0] and A[1]
  - If they are out of order, then swap them
- Advance one element in array A:
  - Compare A[1] and A[2]
  - If they are out of order, swap them
- Repeat process till end of the array
- If at least one swap is made then repeat the whole process again
- Stable Sort
- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$
- Number of operations depends on the input

Divide-and-Conquer

Merge Sort

- Merge step:
    - Takes two-presorted arrays as inputs and outputs a combined sorted array
- Algorithm:
    - If array is size 1, return
    - Split the array into 2 halves and call merge sort recursively
    - Merge the 2 sorted array into a combined sorted array
- Time Complexity: $O(nlogn)$
- Space Complexity: $O(n)$
- Same number of operations regardless of input
- Stable sort
- Can be parallelized using a multi-core processor

Quick Sort

- Algorithm:
  - If array contains 0 or 1 element, return
  - Select a random item from the array called the pivot (p)
  - Move all elements less than or equal to p to the left of array and all elements greater than p to the right
  - Recursively repeat above process on left and right sub-array
- Time Complexity: $\mathrm{O}(n^2)$
- Space Complexity: $\mathrm{O}(n)$
- Not Stable Sort
- Can be parallelized
- Works very well in practice

Array Sorting Algorithms

	Algorithm	Time Complexity			Space Complexity	
		Best	Average	Worst	Worst	
Not stable	Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))	Divide and Conquer
	Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)	
	Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)	
Not stable	Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)	
	Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)	
	Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)	
Not stable	Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)	
	Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)	
Not stable	Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)	
	Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)	
	Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)	
	Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)	
	Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)	