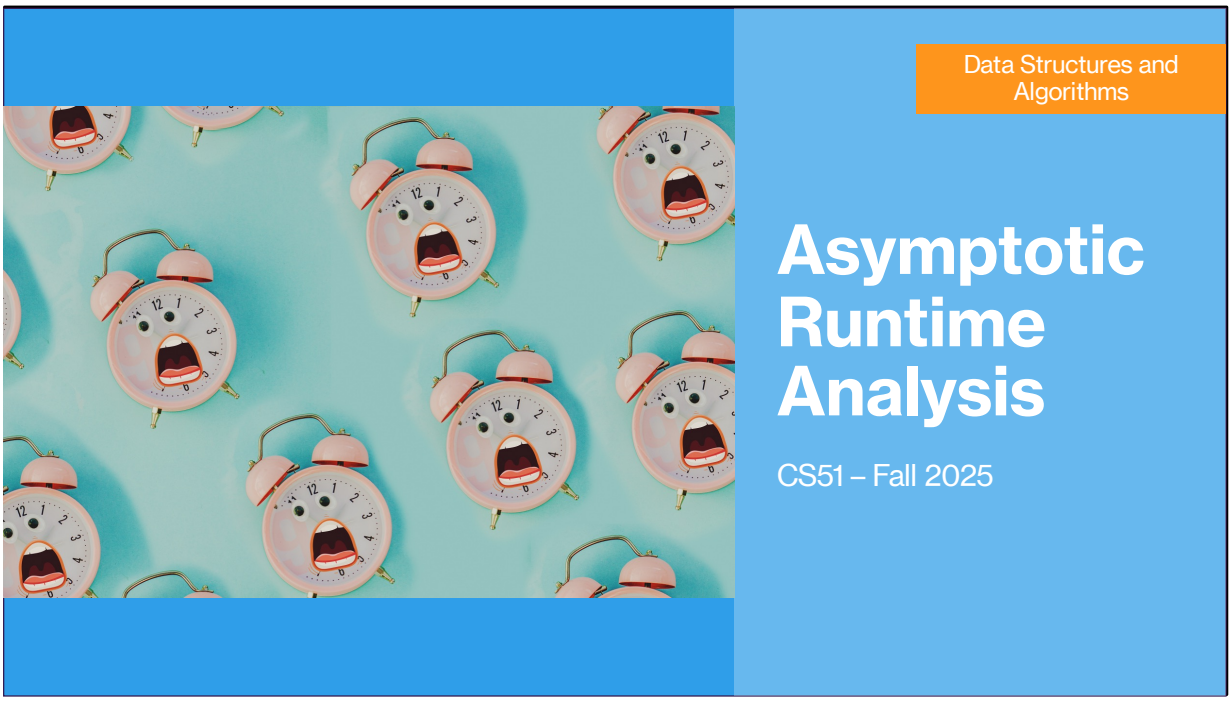


We made it to the last lecture of this unit! Today we will talk about asymptotic runtime analysis. So far, we have tackled a lot of problems, writing Python code to implement our algorithms, the step by step solution to the problems. Computer scientists spend considerable time analyzing algorithms.

Main parts to analyzing algorithms

- Does the algorithm work (**correctness**)?
- How fast does the algorithm run or how much memory does it need (**efficiency**)?
- We will focus on analyzing the runtime of algorithms.
- Our ultimate goal is to pick the best performing algorithm.

2

When analyzing algorithms, computer scientists typically focus on two things. Is the algorithm correct, meaning it works and does the work it is supposed to do (see, for example, proofs for iterative algorithms using loop invariants) and is the algorithm efficient (both in terms of how fast it runs and how much memory it needs). Together, we will focus on analyzing the runtime of algorithms, that is understanding how fast they can run. If we have multiple algorithms for the same problem that we know are all correct, runtime analysis will allow us to pick the fastest one (similarly, we might have cared about the one that requires the least amount of memory).

Asymptotic runtime analysis

- How does the runtime of our algorithm grow as we increase the size of the problem?
  - Size of the problem could be the number of elements in a list we want to sort or number of characters in a string we want to reverse, etc.
- For example, if we double the input, what will happen to the runtime? Will it...
  - stay unchanged?
  - double?
  - triple?
  - quadruple?
  - ...
- Asymptotic runtime analysis relies on math to see how the work that the algorithm does scales in respect to input size.
- It does NOT provide an exact running time nor is an experimental approach where we measure how long our code takes to run using a stopwatch.

3

The key idea when thinking of the runtime of our algorithms is to see how much slower they become when the problem they try to solve becomes bigger. For example, if we double the input size, what happens to the runtime? It could stay unchanged (wouldn't that be great) or become twice as slow, or three times, or four times, and so on. To figure that out, we will use asymptotic analysis which relies on Math to see how the work that the algorithm does scales in respect to input size. We won't know the exact runtime; we don't have a stopwatch. It's just math on the number of operations the algorithm has to perform.

Big O notation

- Big O notation characterizes functions according to their growth rates.
- The **growth rate** of a function is also referred to as the **order of the function**.
- Definition: $f(n) = O(g(n))$ if $|f(n)| \leq c \times |g(n)|$, for all $n > n_0$.
  - We read this as "$f(n)$ is big O of $g(n)$" or "$f(n)$ is of the order of $g(n)$"
- The analysis is asymptotical because we refer to very large sizes for n. In such settings, if a function consists of multiple terms only the one with the largest growth rate is kept, the others are omitted.
- For example, $f(n) = 6n^3 + 2n^2 + 1$. We can say that $f(n) = O(n^3)$ as the one with the largest exponent is the one that contributes the highest growth as the problem size increases.

https://en.wikipedia.org/wiki/Big_O_notation

4

In general, for asymptotic runtime analysis we will borrow the Big O notation from Math which characterizes functions according to their growth rates. The growth rate of a function is also referred to as the order of the function and its mathematical definition is that a function f on is big of g(n) or of the order of g(n) if there is a constant c that when multiplied with the absolute value of g(n), it is greater or equal to f(n) for all sufficiently large ns. The analysis is asymptotical because we refer to very large sizes for n. In such settings, if a function consists of multiple terms only the one with the largest growth rate is kept, the others are omitted. For example, $f(n) = 6n^3 + 2n^2 + 1$. We can say that $f(n) = O(n^3)$ as the one with the largest exponent is the one that contributes the highest growth as the problem size increases.

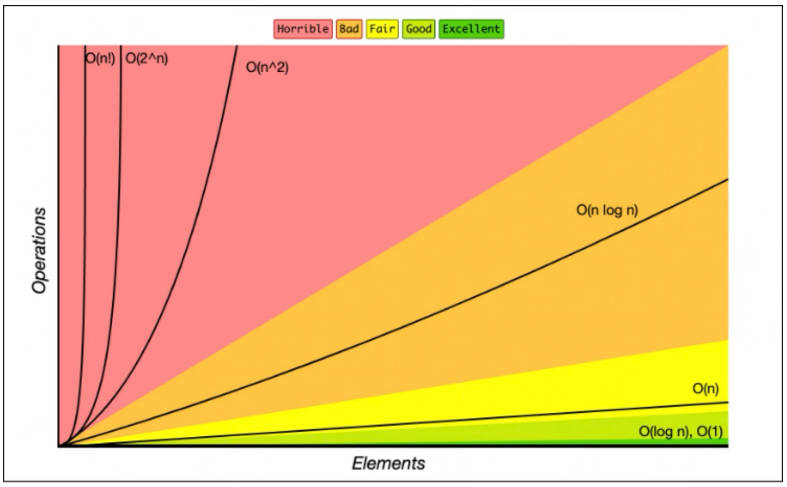
Common classes of runtime complexities

- $O(1)$ = constant
  - Doubling the input size, won't affect the runtime. Holy-grail but often pretty hard to accomplish.
- $O(\log n)$ = logarithmic
  - Doubling the input size, will increase the runtime by a constant.
- $O(n)$ = linear
  - Doubling the input size, will result to double the runtime.
  - Rule of thumb: try to keep your programs running at or below this range.
- $O(n^2)$ = quadratic
  - Doubling the input size, will result to quadrupling the runtime.
- $O(n^3)$ = cubic
  - Doubling the input size, will result to eight times longer runtime.
- Even faster growing (slower) algorithms such as exponential (c^n) or factorial ($n!$).

5

In practice, g is often one of a few simple functions. From slowest growing (i.e. fastest algorithm) to fastest growing (i.e. slowest algorithm), we would have constant (same performance), logarithmic (slower by a constant), linear (twice as slow), quadratic (four times as slow), cubic (eight times slower) all the way to exponential or even factorial.

Common classes of runtime complexities



https://www.bigocheatsheet.com/

6

This is a very useful figure that shows how different classes of algorithms behave as the number of elements they work on increases. In green is the ideal scenario, constant or logarithmic. Linear is acceptable but once we get into things like linearithmic (nlogn), quadratic (n^2), exponential (2^n) or factorial (n!) we get abysmal performance for large problems.

Practice time

Using big O notation, simplify the following functions:

- $3n^3 + 2n + 7$
- $2^n + n^2$
- 1000
- $n + 1000$
- $1 + \frac{1}{n}$

7

Using big O notation, simplify the following functions

Answer

Using big O notation, simplify the following functions:

- $3n^3 + 2n + 7 = O(n^3)$
- $2^n + n^2 = O(2^n)$
- $1000 = O(1)$
- $n + 1000 = O(n)$
- $1 + \frac{1}{n} = O(1)$

8

Remember, big O keeps the largest factor only.

Revisiting `linear_search` function

```
def linear_search(lst, element):
  n = len(lst)
  for i in range(n):
    if lst[i]==element:
      return i
  return -1
```

- What is the runtime for `linear_search` counting number of comparisons (==)?
  - In the worst case, $O(n)$. We have to go through the entire list to either not find the element at all or find it in the last position.
  - In the average case, the element will be somewhere in the middle and we will have to make about n/2 comparisons $O\left(\frac{n}{2}\right) = \mathrm{O(n)}$.
  - In the best case, the element will be in the first index and we will only need to make one comparison, thus $O(1)$.

9

Let's revisit the linear search function we worked on last week. The big O for this function would be linear for the worst case where the element is either not in the list at all or in the last index. Imagine what happens as your problem size increases, meaning you are given increasingly larger lists to search the element through. If your list is now twice as large as it was before, the code will be twice as slow because you will have to do twice as much work, your for loop when from 0…n to 0…2n elements. On average, we expect that the element would be somewhere half way. That is still linear time, as the 2 denominator gets thrown out. The best case scenario is that the element will be in the first index and we will only need to make one comparison, thus O(1).

Revisiting `bubblesort`

```
def bubblesort (lst):
    n = len(lst)
    for i in range(n):
        for j in range(n-i-1):
            if lst[j]>lst[j+1]:
                exchange(lst, j, j+1)
```

- What is the big O for `bubblesort` in the worst case?
  - In first outer loop, we make n-1 comparisons.
  - In second outer loop, we make n-2 comparisons.
  - In third outer loop, we make n-3 comparisons.
  - ...
  - In last outer loop, we make 1 comparison.
  - Hence, the number of comparisons is $(n-1)+(n-2)+(n-3)+\cdots+1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
  - This pattern is common in nested if you have two nested for loops.
- What is the big O for the average and best case?
  - Still $O(n^2)$.

10

What is the big O for bubblesort in the worst case if we count comparisons (the if statement that compares the j-th and j+1-th element).

Revisiting optimized `bubblesort`

- Worst case and average case analysis doesn't change.
- But we can think of a best-case scenario where only n comparisons are needed because the list is already sorted.
- This small modification changed our best case from $O(n^2)$ to $O(n)$!

```
def bubblesort (lst):
    n = len(lst)
    for i in range(n):
        exchanged = False
        for j in range(n-i-1):
            if lst[j]>lst[j+1]:
                exchange(lst, j, j+1)
                exchanged = True
        if not exchanged:
            break
```

11

Quadratic runtime is pretty bad, can we do any better? Last time we saw a small modification we could make in bubblesort so that if a list has been sorted at some point, we stop doing any unnecessary future steps. Our worst and average case analysis doesn't really change.

Revisiting `selectionsort`

```
def selectionsort (lst):
    n = len(lst)
    for i in range(n-1):
        min_index = i
        for j in range(i+1, n):
            if lst[j]<lst[min_index]:
                min_index = j
        exchange(lst, i, min_index)
```

- What is the big O for `seletionsort` in the worst case?
  - In first outer loop, we make n-1 comparisons.
  - In second outer loop, we make n-2 comparisons.
  - In third outer loop, we make n-3 comparisons.
  - ...
  - In last outer loop, we make 1 comparison.
  - Hence, the number of comparisons is $(n-1)+(n-2)+(n-3)+\cdots+1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
  - What is the big O for the average and best case?
    - Still $O(n^2)$.

12

Revisiting `insertionsort`

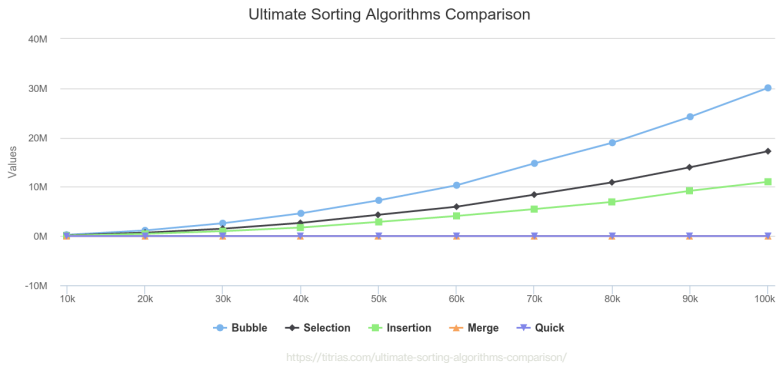
```
def insertionsort (lst):
    n = len(lst)
    for i in range(1, n):
        for j in range(i, 0, -1):
            if lst[j]<lst[j-1]:
                exchange(lst, j, j-1)
            else:
                break
```

- What is the big O for `insertionsort` in the worst case?
  - $O(n^2)$.
  - The list is sorted in reverse order.
- What is the big O for the average case?
  - Still $O(n^2)$.
  - The list elements are randomly allocated.
- What is the big O for the best case?
  - $O(n)$.
  - The list is already sorted and we make one comparison only for each of the n runs of the outer loop.

13

Is this the best we can hope for?

- mergesort and quicksort are two comparison-based sorting algorithms with $O(n \log n)$ average performance.
- This is the lower bound for how well we can expect any comparison-based sorting algorithms.



14

It seems all the algorithms we have seen so far on average have O(n^2) behavior which we know that isn't good. Is this the best we can hope for when sorting data? The good news is that there are other comparison based algorithms like mergesort and quicksort which you will learn about if you continue with CS62 that have linearithmic (O(nlogn)) performance. This is unfortunately the lower bound of what we can achieve with comparison based algorithms. To put it in perspective, look at this graph as the number of elements on the x axis grows. You can see that merge and quick sort are so much faster than the other quadratic algorithms