Algorithm Analysis and Complexity

CMSC 201 Section 60
November 20, 2024

Administrative Notes

Class will NOT meet next Wednesday, November 27

- Enjoy your Thanksgiving holiday

What should you take away from the last two lectures?

Asymptotic run-time

Definition: The limiting behavior of the execution time of an *algorithm* when the size of the problem goes to infinity. This is usually denoted in *big-O notation*.

Asymptotic literally means “the value of an expression as the value tends toward infinity.”

A common mathematical concept

Taken From: https://xlinux.nist.gov/dads/HTML/asymptoticTimeComplexity.html#:~:text=Definition%3A%20The%20limiting%20behavior%20of,denoted%20in%20big%2DO%20notation.

Notation:

“Big O” notation: use a capital “O” to describe how long it takes for an algorithm to execute in the worst case. E.g., how many comparisons it takes.

We usually express this in terms of “n” because we assume a list of size n.

In the previous example: linear search is O(n). Because in the worst case - like with 83 - you have to check every element in the list. All n of them

Big Omega - **Ω**(n) - describes how long the algorithm takes to run in the best case. Linear search is **Ω**(1) because in the best case it only takes one comparison to find what you’re looking for

Comparison of algorithms

There are two ways to compare sorting and searching algorithms: by the number of comparisons and swaps they involve; or by the time they take

We'll look at our algorithms both ways

- Comparisons and swaps
- time

Linear Search:

We have the following 120-element list:

[22, 75, 67, 24, 49, 65, 96, 81, 96, 36, 66, 100, 73, 30, 23, 32, 89, 5, 8, 70, 71, 9, 71, 77, 48, 45, 6, 73, 42, 71, 55, 98, 19, 47, 71, 21, 43, 75, 5, 72, 78, 53, 72, 89, 60, 79, 43, 89, 84, 81, 14, 31, 44, 54, 41, 91, 78, 71, 24, 24, 42, 30, 57, 55, 26, 26, 48, 65, 28, 95, 74, 93, 89, 49, 92, 86, 14, 62, 36, 15, 51, 27, 36, 6, 24, 41, 69, 54, 14, 24, 50, 6, 27, 58, 100, 45, 35, 9, 91, 57, 22, 3, 50, 72, 89, 13, 64, 0, 68, 52, 20, 16, 52, 40, 6, 74, 34, 34, 15, 71]

How many comparisons does it take to see if the number 83 is in the list, using linear search? 120, because it's not there and we have to check each element to confirm that.

How many comparisons does it take to see if the number 22 is in the list? 1, because we find it on the first comparison

HUMONGOUS Data Sets

- About 500 million tweets per day
- 30,000 hours of video uploaded to YouTube per hour; 720,000 hours of video per day
- 95 million Instagram posts per day

… you get the picture. There are some big data sets

We really need algorithms that run fast on these big data sets

Binary search

Worst case - $O(\log_2 n)$ - you might not find it

Best case - $\Omega(1)$ - you might find it on the first value

Explanation:

$\log_2 n$ is the power that you have to raise 2 to in order to get n. Also, the number of times you can successively divide n in half.

If n = 256, log 256 = 8. N = 1024, log 1024 = 10

Now, the sorting algorithms

Bubble sort:

- List is n elements long
- You have to go through all remaining unsorted elements of the list each time:
  - n elements the first time; n-1 the second time; and so on.
  - This equals n * (n+1) / 2 from your calculus classes. Or, $(n^2 + n) / 2$
- We round this off to n^2.
- Bubble sort is $O(n^2)$ - WORST CASE BEHAVIOR
- Bubble sort is $\Omega(n)$ - if the list is already sorted and you stop when you don't swap anything, you only have to go through the list once

SIgma(i) from 1 to n -the sum of the first n numbers = n *(n+1) /2

Don’t sweat the small stuff

Bubble sort is $O((n^2 - n)/2)$. How come we rounded that off to $O(n^2)$?

Think if n is really, really large. Say 1 million.

1 million squared is 1 trillion - 10 to the 12th power.

When n = 1 million, $(n^2 + n)/2$ = (1 trillion + 1 million)/2. Or 999 billion, 999 million /2. We just round that off to 1 trillion - it’s close enough.

Selection sort

Selection sort is almost like bubble sort

- List is n elements long
- You have to go through all remaining unsorted elements of the list each time:
  - n elements the first time; n-1 the second time; and so on.
  - This equals n * (n-1) / 2 from your calculus classes. Or, $(n^2 - n) / 2$
- We round this off to n^2.
- Selection sort is $O(n^2)$
- Selection sort is $\Omega(n^2)$
- So we say that selection sort is $\Theta(n^2)$ - if O and Ω are equal, Θ is the same value

What about quicksort? It's different

There's an area of risk. When we pick the pivot, we have no idea whether the pivot is somewhere in the middle of the values to be sorted.

If we get really, really unlucky, each time we pick a pivot it's the smallest number left, or the largest. All remaining values go to one side, and we just make the list one element smaller.

In that case, quicksort is $O(n^2)$ just like the other two algorithms.

You will often see that quicksort is $O(n * \log_2 n)$ - that's the more likely, average behavior

And the best case is $\Omega(n)$

But realistically...

On the average, you're not going to randomly pick the worst possible value for the pivot every time. Sometimes you're going to have good luck.

In that case, you'll divide the list to be sorted into halves, and this becomes like binary search.

So you'll often see that quicksort is O(n * log n)

If the list has n elements, n/2 will be less and n/2 will be greater

So now your recursive calls each have n/2

- n = 120; recursive calls will have 60 and 60. Then call with 30 and 30. Then 15 and 15; then 7 and 7; then 3 and 3; then 2 and 2; then 1.

What's 2 to the 7th power? 128 - approximately 120. $\text{Log}_2 120$ is approximately 7

Times n recursive calls - that's where you get (n * log n)

n * log n is always less than n^2 for any positive integer n -> quicksort is faster.

An illustration

https://www.youtube.com/watch?v=ZZuD6iUe3Pc