CMPT 125

Introduction to Computing Science and Programming II

October 06, 2021

Assignment 2

- Assignment 2 is due to October 15, 23:59
- https://www.cs.sfu.ca/~ishinkar/teaching/fall21/cmpt125/assignments.html

- You need to submit one file to Canvas – *assignment2.c*

- This assignment is more challenging than assignment 1.
- Start working on it earlier.

Big-O notation

Searching in array

Goal: Search for a given element in an array.

Q: do we have any information about the array?

Searching in an unsorted array

If we do not know how the elements are arranged in the array, we can use a linear search (*for loop*), iterating over all elements in the array one after another.

In the worst case, the element we are looking for may be the last element in the array, or may not be in the array at all.

Therefore, if the array contains *N* elements the algorithm will read each of the N elements at most ones.

We will say the running time of the search is *linear - O(N)*.

Why isn't the runtime exactly N?

Searching in a sorted array

If the array is sorted, then searching for an element in the array becomes easier.

For example, when looking for a word in a dictionary, we do not go through the entire dictionary.

We use the fact that the dictionary is sorted to quickly find the word we are looking for.

Idea: **divide and conquer**

Divide: cut the array into 2 roughly equal pieces

Use the fact that the array is sorted:
search in only one of the halves.

More on this next week

Three more examples

Summation:

Input: Two numbers each of length n
Outputs: The sum of the two numbers

Multiplication:

Input: Two numbers each of length n
Outputs: The product of the two numbers

Decomposing a number into primes

Input: A number N of length n that is a product of two primes
Outputs: find primes p,q such that N = p*q

Question: Suppose the basic operations are on one digit at a time. How many operations are required to solve each of the problems?

Big-O notation

We will use the Big-O notation to express the *time complexity* (running time) of algorithms.

Denote by f(N) the running time of a program for inputs of size N.
Examples:

$f(N) = 1.5N^2 + 4N + 3$
$f(N) = 2N^4 + 10N^3 + 4N^2 + 900\ log(N) + 3000$
$f(N) = 2^N - 100$ (for $N>10$)

The big-O notation will be the term that increases the fastest for large inputs.

We are ignoring:
1) low order terms
2) multiplicative constants

Example: if $f(N) = 2N^4 + 10N^3 + 4N^2 + 900\ log(N) + 3$ we will say that the *time complexity* is $O(N^4)$.

Big-O notation

Q: Why are we ignoring low order terms?

A: Because they become negligible as N grows

Example: $f(N) = 4N^4 + 100N^3 + 9000N^2 + N$

	$4N^4$	$100N^3$	$9000N^2$	N
N = 100	$4*10^8$	10^8	$9*10^7$	100
$N = 10^8$	$4*10^{32}$	10^{26}	$9*10^{19}$	10^8
$N = 10^{12}$	$4*10^{48}$	10^{38}	$9*10^{27}$	10^{12}

Big-O notation

Q: Why are we ignoring multiplicative constants?

A: Because if we buy a computer that runs twice as fast, then the running time decreases accordingly.

But it will not change the **order of magnitude**

In practice constants do matter!

In theory we ignore them.

Big-O notation

Let *f(N)* and *g(N)* be two functions on positive integers.

f = O(g) will mean that f is "essentially smaller" than g.

A naive attempt:
Wrong definition: f = O(g) if f(N) <= g(N) for all N.

Example: f(N) = 2N, g(N) = N.
Then, we want f = O(g), but it is not true that f(N) <= g(N) for all N.

Wrong definition2: f = O(g) if f(N) <= 2g(N) for all N.

Example: f(N) = 3N, g(N) = N.
Then, we want f = O(g), but it is not true that f(N) <= g(N) for all N.

Correct definition: We say that *f = O(g)* if
there exists a large enough constant C (e.g. *C = 1000*)
such that *f(N) <= C*g(N)* for all N.

Big-O notation

Formally: Let *f(N)* and *g(N)* be two functions on positive integers.

We say that *f = O(g)* if there exists a large enough constant C (e.g. *C = 1000*) such that *f(N) <= C*g(N)* for all sufficiently large N.

f = O(g) if there C>0 (e.g. C = 1000) such that f(N)/g(N) <= C for all N large enough.

Example: *f(N) = $1.5N^2 + 4N + 3$. Want to show f = $O(N^2)$*

Let ***C=8.5***. *Then f(N) = $1.5N^2+4N+3 <= 1.5N^2+4N^2+3N^2 = 8.5N^2 = CN^2$*

Therefore f = $O(N^2)$

Big-O notation

Formally: Let *f(N)* and *g(N)* be two functions on positive integers.

We say that *f = O(g)* if there exists a large enough constant C (e.g. *C = 1000*) such that *f(N) <= C*g(N)* for all sufficiently large N.

f = O(g) if there C>0 (e.g. C = 1000) such that f(N)/g(N) <= C for all N large enough.

Example: *f(N) = $1.5N^2 + 4N + 3$. Want to show f = $O(N^3)$*

Let ***C=8.5***. *Then f(N) = $1.5N^2+4N+3 <= 1.5N^3+4N^3+3N^3 = 8.5N^3 = CN^3$*

Therefore f = $O(N^3)$

Big-O notation

Formally: Let *f(N)* and *g(N)* be two functions on positive integers.

We say that *f = O(g)* if there exists a large enough constant C (e.g. *C = 1000*) such that *f(N) <= C*g(N)* for all sufficiently large N.

f = O(g) if there C>0 (e.g. C = 1000) such that f(N)/g(N) <= C for all N large enough.

Example: $f(N) = 1.5N^2 + 4N + 3.$

Then

$f = O(N^2)$ – *TRUE*
$f = O(N^3)$ – *TRUE*
$f = O(2^N)$ – *TRUE*

f = O(N) – *FALSE*
f = O(log(N)) – *FALSE*
f = O(1) – *FALSE*

Common orders of magnitude

O(1) – bounded by some absolute constant (e.g., <= 100)

O(log N) – logarithmic complexity – very fast

O(N) – linear: That is constant times the size of the input. We consider it very efficient

O(N log N) – Almost as good as linear.

Q: Explain what is log(N) in simple words

O(N^2) – quadratic time

O(N^3), O(N^4), ...O(N^7)... O(N^{const}) – polynomial

O(2^N) – exponential: considered as very inefficient

O(4^N) – exponential: even worse than *2^N*

O(N!) – more than exponential

O(2^{2^N}) – double exponential ☹ [too much even for N = 10]

Big-O notation – simple rules

Fix $0 < a < b$ (e.g., $a=2, b=4$)

Then $N^a = O(N^b)$
But $N^b = O(N^a)$ ***does not hold***

$log(N)$ is smaller than any power of N: $log(N) = O(N^{0.1})$

$log_2(N) = log_{10}(N) * log_2(10)$

Absolute constant:
$3 < log_2(10) < 4$

No need to specify the base of log

If $f = O(g)$ and $g = O(h)$, then $f = O(h)$

If $f_1 = O(g)$ and $f_2 = O(g)$, then $f_1 + f_2 = O(g)$

$f+g <= 2max(f,g) = O(max(f,g))$

Big-O notation – simple rules

Claim: If $f_1 = O(g)$ and $f_2 = O(g)$, then $f_1+f_2 = O(g)$
Proof:

$f_1 = O(g)$ ➔

There is some C_1 such that $f_1(N)<=C_1g(N)$ for all N large enough.

I.e., there is some n_1 (e.g. $n_1=10$) such that $f_1(N)<=C_1g(N)$ for $N>n_1$.

$f_2 = O(g)$ ➔ *There is some C_2 n_2 such that $f_2(N)<= C_1g(N)$ for $N>n_2$.*

For f_1+f_2: let $C = C_1+C_2$. Then $f_1(N) + f_2(N) < C_1g(N)+ C_1g(N) =C*g(N)$.

All this assuming that $N>max(n_1,n_2)$.

This implies that $f_1+f_2=O(g)$

Big-O notation

Sort the functions in the increasing order:

- $f_1(N) = N^2 + 100N$
- $f_2(N) = 2^N + N^6 + 100N$ - $O(2^N)$
- $f_3(N) = N^3 \log(N) + 400N^2$
- $f_4(N) = 2N^3 + 100N + 10^8$
- $f_5(N) = (\log(N))^{10}$
- $f_6(N) = 99N + \log(N) + 4^N$ - $O(4^N)$
- $f_7(N) = N \log(N) + 100N$ [$\log(N) < N^{0.1}$ ➔ $\log(N)^{10} < N$]
- $f_8(N) = \log(N/2)$

$f_8 < f_5 < f_7 < f_1 < f_4 < f_3 < f_2 < f_6$

Go to https://www.desmos.com/ and draw all these functions

Searching algorithms

Searching in array

Goal: Search for a given element in an array.

Q: do we have any information about the array?

Searching in an unsorted array

If we do not know how the elements are arranged in the array, we can use a linear search (*for loop*), iterating over all elements in the array one after another.

In the worst case, the element we are looking for may be the last element in the array, or may not be in the array at all.

Therefore, if the array contains *N* elements the running time of the search is *linear- O(N)*.

Why isn't the runtime exactly N?

Searching in a sorted array

If the array is sorted, then searching for an element in the array becomes easier.

For example, when looking for a word in a dictionary, we do not go through the entire dictionary.

We use the fact that the dictionary is sorted to quickly find the word we are looking for.

Idea: **divide and conquer**

Divide: cut the array into 2 roughly equal pieces

Use the fact that the array is sorted:
search in only one of the pieces.

Binary search

Running time for array of length N:
Denote the running time by T(N)

Input: A sorted array, an element

Output: Index of the element in the array (or N/A)

1. Compute the midpoint of the array

2. If array[midpoint] == element
2.1 Return midpoint

3. Else

3.1 If array[midpoint] > element
3.1.1 Search in the left half of the array

3.2 Else // array[midpoint] < element
3.2.1 Search in the right half of the array

}

Checking if statements: O(1)

Recursion: T(N/2)

Total: T(N) = T(N/2) + O(1)

Binary search

Analyzing the running time T(N):

The recursion formula is $T(N) = T(N/2) + O(1)$.

How do we solve it?

$$
\begin{aligned}
T(N) &= T(N/2) + O(1) \\
&= T(N/4) + 2*O(1) \\
&= T(N/8) + 3*O(1) \\
&= \ldots \\
\text{[For } i > 0\text{]} \quad &= T(N/2^i) + i*O(1) \\
\text{[For } i = \log_2(N)\text{]} \quad &= T(1) + \log_2(N)*O(1) \\
&= O(\log(N))
\end{aligned}
$$

Binary search

Advantages: really fast!

Requirements: the array must be sorted

- If the array is dynamic, this can be expensive.
- For example, if we need to insert new values into the array

Homework: write a recursive version of binary search.

Questions?

Comments?