CMPT 125

Introduction to Computing Science and Programming II

October 11, 2021

Midterm – Oct 25

- Midterm will be on October 25, during the Monday class.
- The exam will be pen and paper.
- Format: 4-5 question each with several items
- Closed books. Only pens/pencils are allowed.
- The material includes everything learned before the midterm (including Oct 18-20)
- For previous exams go to https://www.cs.sfu.ca/~ishinkar/teaching/fall21/cmpt125/exams.html
  (In some offerings the midterm was a bit later, and covered more material)
- Solve all practice problems on piazza: The links are under Resources

Lecture recordings

- I posted on piazza links to videos from CMPT125 I taught in 2020.
- The material is exactly the same
- The links are under Resources

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*
- Please make sure it compiles with the provided makefile

>> make

>> ./run_test2

- This assignment is more challenging than assignment 1.
- But it allows you to practice pointers

Assignment 3,4,5

- Assignment 3,4 have been moved one week forward
- https://www.cs.sfu.ca/~ishinkar/teaching/fall21/cmpt125/assignments.html

Big-O notation

Big-O notation - Definition

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 - Definition

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=10***. *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: $f(N) = 1.5N^2 + 4N + 3$.

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:

1) write a recursive version of binary search.

2) write a recursive version of linear search.

More on Big-O notation

Running time of Fib

Recall the recursive algorithm for Fib(n)

```
Fib(n):
    if (n <=1)
        return n;
    else
        return Fib(n-1) + Fib(n-2)
```

On the other hand, the running time of Fib using an array is O(n)

What is the running time of the algorithm?

$T(n) = T(n-1) + T(n-2) + O(1)$

$T(n-1) = T(n-2) + T(n-3) + O(1)$

→ $T(n) > 2*T(n-2) + O(1)$

Exponential time!

…

→ $T(n) > 2^{n/2}$

Prove it!

Running time of Fib

Recall the recursive algorithm for Fib(n)

```
Fib(n):
    if (n <=1)
        return n;
    else
        return Fib(n-1) + Fib(n-2)
```

What is the running time of the algorithm?

$T(n) = T(n-1) + T(n-2) + O(1)$

$T(n-1) = T(n-2) + T(n-3) + O(1)$

…

Q: Compute the correct running time of Fib()?

Big-O notation - Examples

Let $f(N) = 1.5N^2 + 4N + 3$. Prove that $f(N) = O(N^3)$

Proof: Let $C = 9$. We want to prove that $f(N) < CN^3$ for all $N >= 1$.

Indeed, $f(N) = 1.5N^2 + 4N + 3$

$< 1.5N^3 + 4N^3 + 3N^3$

$< 9N^3$

Therefore $f = O(N^3)$

Big-O notation - Examples

Let $f(N) = 10*2^N + 4N^4 + 3$. Prove that $f(N) = O(2^N)$

Use the fact that $N^4 < 100 * 2^N$ for all $N > 1$

Big-O notation - Examples

Let $T(N)$ satisfy the recursive formula
$T(N) = T(N-1) + O(1)$ and $T(1) = O(1)$.

Prove: $T(N) = O(N)$.

Proof:

1) By definition, there exists C such that $T(N) < T(N-1) + C$.

Intuition: this implies that $T(N-1) < T(N-2) + C$...

2) Claim: for all i=0,1,…N-1 it holds that $T(N) < T(N-i) + i*C$.

Proof: by induction on i

3) Therefore, by plugging in $i = N-1$ we get
$T(N) < T(N-(N-1)) + (N-1)*C = T(1) + (N-1)*C = O(N)$.

Big-O notation - Examples

Let $T(N)$ satisfy the recursive formula $T(N) = T(N/2)+O(1)$ and $T(1) = O(1)$.

Prove that $f(N) = O(log(N))$

Big-O notation - Examples

Let $T(N)$ satisfy the recursive formula $T(N) = T(N-1)+O(N)$ and $T(1) = O(1)$.

Prove that $f(N) = O(N^2)$

Big-O notation - Examples

Let $T(N)$ satisfy the recursive formula $T(N) = T(N/2)+O(N)$ and $T(1) = O(1)$.

Prove that $f(N) = O(N*log(N))$

Big-O notation - Examples

Let $T(N)$ satisfy the recursive formula
$T(N) = k*T(N-1)+O(N)$ and $T(1) = O(1)$. (Think of $k=7$)

Prove that $f(N) = O(k^N)$

More examples

How many iterations are performed in the following for loop?

Foo(N):

```
for (int i = 1 ; i < N ; i = i*2) {
    sum = sum+i;
}
return sum;
```

Rewrite the function so that the running time is O(1)

More examples

Express the run time of the following program using Big-O notation.

```
foo(int N)
{
        if (N > 1)
                foo(N/2);
 }
```

More examples

Express the run time of the following program using Big-O notation.

```
foo(int N) {
        if (N > 1)
        {
                foo(N-1);
                foo(N-1);
                foo(N-1);
        }
 }
```

Questions?

Comments?