CMPT 125

Introduction to Computing Science and Programming II

October 4, 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.

We are done the syntax of C

One last example just for fun

See quine.c

Pseudo-code

Pseudo-code

Starting today we will write (mostly) pseudo-code

- High-level description of an algorithm
- Essential details needed to implement the algorithm
- Language independent
- No rules for syntax, should be readable by humans
- No need to worry about types of variables / pointers / memory allocation

Recursion

Recursion

Recursion is a powerful tool for solving certain kinds of problems.

Recursion breaks a problem into smaller sub-problems that are, in some sense, identical to the original.

Use the solution of the smaller sub-problems to solve the original problem.

Examples:

Factorial: Write a function that gets an integer n>=0, and computes n!

$$n! = 1*2*\ldots*(n-1)*n = (n-1)! * n$$

Fibonacci numbers: Write a function that gets an integer n>=0, and computes Fib(n) defined as follows:

$$Fib(0) = 0,\ Fib(1) = 1,\ Fib(n) = Fib(n-1)+Fib(n-2) \text{ for } n >=2$$

Recursion

Write a function that gets n and computes n!

```
long long factorial(unsigned int n) {
        long long result = 1;

        for ( int i = 1; i <= n ; i++)
                result = result*i;

        return result;

}
```

Recursion

Write a function that gets n and computes n!

```
long long factorial_rec(unsigned int n) {
        if (n <= 1)
                return 1;
        return n*factorial_rec(n-1);
}
```

Recursion

Write a function that computes the Fibonacci sequence:

Fib(0) = 0, Fib(1) = 1
Fib(N) = Fib(N-1) + Fib(N-2) for N >=2

Write a recursive implementation

Write an iterative implementation

Comment: when implementing in C use ***long long*** (not int)

See fib.c

Ex: Compute fib(n) in linear time (fast) using not more than 4 (5-6?) variables.

Fun fact about Fibonacci numbers

- Fib(n) has a closed formula.

$$Fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

- If you take linear algebra/discrete math, you should learn how to derive the closed formula.
- Fact: $((1-\sqrt{5})/2)^n / \sqrt{5} < 0.2$ for all $n>=2$.
- So computing only the first term suffices to compute Fib(n)

Fun fact about Fibonacci numbers

- Fib(n) has a closed formula.

$$Fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Fib(n)

- If n <=1 return n
- Otherwise
    - Let a=(1+√5)/2
    - Compute B = a^n
    - Return rounding(B/√5)

Try it: Compare Fib(n) to this algorithm

Computing a^n

Input: a real , n integer, the goal is to compute a^n

Naïve algorithm: run a for loop for n iterations

A more sophisticated algorithm: runs in only O(log(n)) iterations.

Example to compute : a^7

Given a

Compute $a^2 = a*a$

Compute $a^4 = a^2*a^2$

Output: $a^7 = a^4*a^2*a$

Conclusion: we used only 4 operations to compute a^7

Merge sort
(more on recursion)

Merge sort

Merge sort: sorting algorithm: gets an array of numbers, and sorts it.

1. Merge sort: Given an array of length n
2. Sort the first n/2 elements
3. Sort the last n/2 elements
4. Merge the two sorted halves

Merge sort

Using recursion

Example: Input: [4, 1, 8, 7, 10, 3]

- sort the left half: [1, 4, 8, 7, 10, 3]
- sort the right half [1, 4, 8, 3, 7, 10]
- merge the two halves

Merge 3: [1, 4, 3, 8, 7, 10] → [1, 3, 4, 8, 7, 10]

Merge 7: [1, 3, 4, 7, 8, 10]

Merge 10: [1, 3, 4, 7, 8, 10]

DONE!

This is an inefficient implementation of the merge part

Space Filling algorithm (more on recursion)

Space Filling algorithm

Given a picture, we want to color a specified bounded area.

Also known as flood fill algorithm or boundary fill algorithm

Example in paint.

Measuring performance of algorithms

Measuring performance of algorithms

How can we measure the running time of an algorithm?

Empirical Timing:

- Run the code on a real machine with various inputs
- Plot the graph of runtime vs size of input

Counting Operations:

Count the number of operations as a function of the input

- Assume for simplicity that all elementary operations run in 1 time unit

Actual performance depends on more than just the algorithm.

Measuring performance of algorithms

Actual performance depends on more than just the algorithm.

- CPU speed
- Operating system / configurations
- Amount of memory
- Other programs running at the same time
- Hardware
- Implementation of the algorithm
- More…

Checking duplicates

Length of the input

```
bool check_duplicates(const int* arr, int n) {
        int i = 0, j;
        bool found = false;
        while (i < n && !found)    {
            j = 0;
            while (j < i && !found)    {
                        if (arr[i] == arr[j]) {
                                found = true;
                        }
                        j++;
            }
            i++;
        }
        return found;
}
```

Outside the loop: 3 ops

Outer loop: 4*n ops

Inner loop: 4*i ops
for each i=0...n-1

Outside loop: 3
Outer loop: 4n
Inner loop = $4*1 + 4*2 + 4*3 + \ldots + 4*(n-1) \cong 2n^2 + 2n$

Total: $\cong 2n^2 + 6n + 3$

If n doubles, the running time is roughly multiplied by 4:: $2(2n)^2 + \ldots = 8n^2$

Two examples

Write a function with the following guarantees:

Input: An array of ints of length n

Output: Returns 0 if all entries are distinct. Returns 1 otherwise.

Write a function with the following guarantees:

Input: An array of ints of length n.

Promise: The array contains all but one number among {0...n}.

Output: Returns the missing number.

What is the running time of each algorithm on inputs of length n?

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

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

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

Questions?

Comments?