Asymptotic Notation and C

Oliver-Matis Lill

February 21, 2018

The Problem

```
void add_to_result(int i) {
    for(int j = 0; j < n; j++)
        for(int k = 0; k < n; k++)
            result += dp[i][j][k];
}
void calculate() {
    for(int i = 0; i < n; i++) {
        if(i * i < n)
            add_to_result(i);
        dp[i][0][0] += prev_dp[i][0][0];
    }
}
```

- Your program has a time limit of 1s
- How do we know if this function "calculate" will finish in time?
- We need some way to estimate the running time of a program

Big O

Definition

$f(x) \in O(g(x))$ if and only if there exist constants M and x_0 such that:

$$|f(x)| \leq M|g(x)| \text{ for all } x > x_0$$

- Note that Big O notation tends to be abused. Oftentimes people write $f(x) = O(g(x))$ instead, but keep in mind that $O(g(x))$ is a set of functions.

- Big O is generally used to estimate a complicated function $f(x)$ with some much simpler function $g(x)$
- Running time of a program can be considered a complicated function

Big O

Examples

$$2x^3 + 5x^2 + 4x + 9 \in O(x^3)$$
$$x^2 + x + 1 \in O(x^3)$$
$$x^2 + x + 1 \in O(x^2)$$
$$x^3 + 2x^2 + 3x + 1 \notin O(x^2)$$

In can become misleading when constants are huge:

$$10^9 \cdot x^2 \in O(x^2)$$
$$0.5 \cdot x^3 \in O(x^3)$$

But fortunately these cases almost never happen

Big O

Exercises

1. Find a good Big O estimate for: $f(n) = 1 + 2 + 4 + \ldots + 2^n$
2. Find a good Big O estimate for: $f(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$
3. Prove that if $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then $f_1(n)f_2(n) \in O(g_1(n)g_2(n))$
4. Given that $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ and $g_1(n) \in O(g_2(n))$, what would be a good Big O estimate for $f_1(n) + f_2(n)$

Big O

Hints

1. Look up "Geometric Series"
2. Use Integrals
3. Recall the definition. Write things out.

Big O in the Real World

Exercise

```
void add_to_result(int i) {
    for(int j = 0; j < n; j++)
        for(int k = 0; k < n; k++)
            result += dp[i][j][k];
}
void calculate() {
    for(int i = 0; i < n; i++) {
        if(i * i < n)
            add_to_result(i);
        dp[i][0][0] += prev_dp[i][0][0];
    }
}
```

- Give a Big O estimation to how many addition operations in total a call to "calculate" will cover with respect to *n*.

Exercise

```
void add_to_result(int i) {
    for(int j = 0; j < n; j++)
        for(int k = 0; k < n; k++)
            result += dp[i][j][k];
}
void calculate() {
    for(int i = 0; i < n; i++) {
        if(i * i < n)
            add_to_result(i);
        dp[i][0][0] += prev_dp[i][0][0];
    }
}
```

- "add_to_result" does $O(n^2)$ additions and it's called $O(\sqrt{n})$ times. It dominates everything else, therefore we can give the program an estimate $O(n^2\sqrt{n})$

Big O in the Real World

- In programs we can estimate:
  1. Processor cycles used
  2. The total number of lines covered
  3. Amount of arithmetic operations
- Thanks to attributes of Big O, all of them will generally give the same estimate
- In fact we generally estimate something vague like "Operations done", where operation can be anything that takes $O(1)$ cycles, from simple subtraction to adding a node to a graph
- The golden rule of competitive programming: your program can do roughly 10^8 to $5 \cdot 10^8$ "operations" per second

Big O in the Real World

Golden Rule: 10^8 to $5 \cdot 10^8$ "operations" per second

Exercise

- You have to solve the problem in 1s on a single thread. You have $n = 10^6$
- Which of the following running times would be suitable for a solution:
  1. $O(1)$
  2. $O(n)$
  3. $O(n \log n)$
  4. $O(n \log^2 n)$
  5. $O(n\sqrt{n})$
  6. $O(n^2)$
  7. $O(2^n)$

Big O in the Real World

Golden Rule: 10^8 to $5 \cdot 10^8$ "operations" per second

Exercise

- Hint: Constant factor M will probably be around 1. Try evaluating the functions. Note that log is in base 2.

Big O in the Real World

Golden Rule: 10^8 to $5 \cdot 10^8$ "operations" per second

Exercise

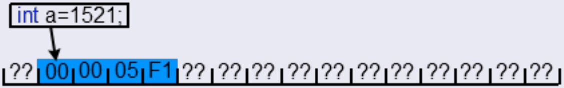
- Hint: Constant factor M will probably be around 1. Try evaluating the functions. Note that log is in base 2.
  1. $O(1)$ V
  2. $O(n)$ V
  3. $O(n \log n)$ V
  4. $O(n \log^2 n)$?
  5. $O(n\sqrt{n})$ X
  6. $O(n^2)$ X
  7. $O(2^n)$ X

The C language

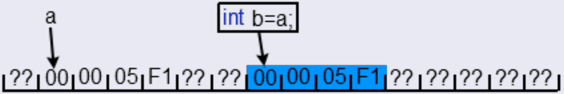
- C is a relatively simple language
- The vast majority of what C allows is close to what the machine itself does
- As such it doesn't allow highly abstractive features like Object Oriented Programming and doesn't have non-trivial structures like Vectors, Sets and Maps built-in
- The flip side of this is that the language should be much easier to learn than C++ or Java
- For a more complete picture, we will go over how memory works in C

C Variable Mechanics

- Each variable is associated with a fixed memory segment from the start of it's lifespan

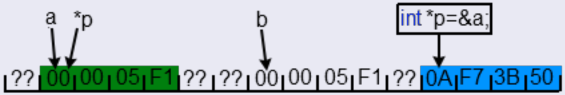


- The assignment operator copies data into the memory segment of the variable

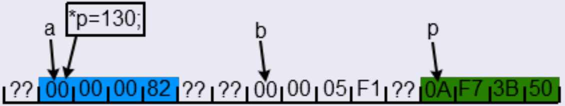


C Pointers

- A pointer is just a variable that stores some memory address:



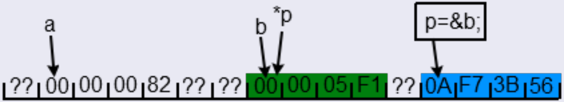
- You can dereference pointers with the * sign and use the dereferenced pointer as a variable to, for example, modify it:



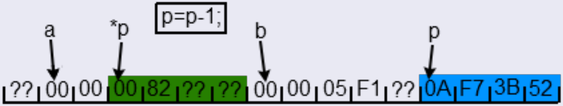
- The type specifier (here int) only helps the compiler know how to use the pointer, otherwise all pointers are the same, just variables that store memory addresses

C Pointers

- Pointers can change the address they point to:



- You can use addition and subtraction to move the pointer (useful for iterating over arrays):



- In fact you can perform all sorts of arithmetic on pointers. I recommend you spend some time learning it and playing with it

C Pointers

Exercise

```
int32_t a = 0; // 32-bit integer
int8_t* p = &a; // 8-bit integer
for(int i = 0; i < 4; i++) {
    *(p+i) = 1;
}
printf("%d", a); // Print the value of "a"
```

What would the above code print?

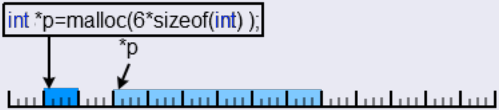
C Arrays

- In C, the array elements always occupy consecutive memory locations:
- In multidimensional arrays the data is consecutive as well and the ordering is fairly intuitive:

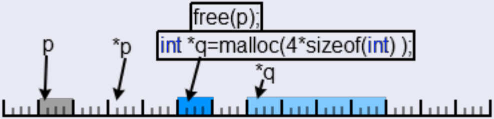
- With C arrays, their size must be known on compile time, so unless n is constant you can't declare an array like int a[n];

C Dynamic Memory

- You can use malloc to allocate a new chunk of memory. The allocated size can be calculated on runtime



- You can use free to deallocate previously allocated memory. That memory can be reused (even by other programs)



- malloc stores extra data on allocation size, so free will always know the size to deallocate

C Exercise

Exercise

Look at two implementations of a 2-D array:

```
// Implementation A
int **a = malloc(8*sizeof(int*));
for(int i = 0; i < 8; i++) {
    a[i] = malloc(11 * sizeof(int));
}
```

```
// Implementation B
int a[8][11];
```

1. How would memory be arranged in implementation A?
2. In A, how many memory locations would you have to read to get the value of some element a[i][j]? How many in B?
3. What are the advantages and disadvantages of A and B?