MA2252 Introduction to computing

lectures 13-14

More on complexity and

Matias Ruiz

October 2023

Complexity: visually inspecting

The following is a useful code for inspecting the average complexity of an algorithm

```
N = 20; K = 30;
T = zeros(N,K);
for k=1:K
    for n = 1:N
        arr = rand(1,n)
        tic,
        myFunction(arr);
        T(n,k) = toc;
    end
end
T = 1/K*sum(T,2);
plot(T(1:end), '*')
```

It gives an average of `K` samples of the time it takes to run the function for each input of size 1 to `N`.

Complexity: complexity of the binary search algorithm

Best case: When the element is the middle pointer, so the element is found in one iteration $\Rightarrow O(1)$

Average case:

- **Case 1:** the key is present in the array $\rightarrow$ n possible cases
- **Case 2:** the key is not in the array $\rightarrow$ 1 possible case
- A key at index $n/2$ is found in 1 comparison
- A key at index $n/4$ and $3n/4$ is found in 2 comparisons
- A key at index $n/8$, $3n/8$, $5n/8$, and $7n/8$ is found in 3 comparisons, etc

$$
\begin{aligned}
\Rightarrow \frac{\text{Tot. comparisons}}{\text{numb. cases}} &= \frac{1 \times 1 + 2 \times 2 + 3 \times 4 + \cdots + \log n \times 2^{\log n - 1}}{n+1} \\
&= \frac{n(\log n - 1) + 1}{n+1} \\
&= O(\log n)
\end{aligned}
$$

Worst case: The key is not in the array (or it is in the first element). Following the same analysis as before we need to do $O(\log n)$ comparisons.

Complexity: complexity of the quick-sort algorithm

Best case: We select the pivot as the mean

$$
\begin{aligned}
T(n) &= 2 \times T\left(\frac{n}{2}\right) + n \times constant \\
&= 2 \times \left(2 \times T\left(\frac{n}{4}\right) + \frac{n}{2} \times constant\right) + n \times constant \\
&= 4 \times T\left(\frac{n}{4}\right) + 2 \times constant \times n. \\
&= 2^k \times T\left(\frac{n}{2^k}\right) + k \times constant \times n \\
&= n \times T(1) + n \times \log n.
\end{aligned}
$$

So in the best case the complexity is $O(n \log n)$

Complexity: complexity of the quick-sort algorithm

Worst case: The array gets divided into one part consisting of $n-1$ elements and that one into $n-2$ elements, so on and so forth.

$$\begin{aligned} T(n) &= T(n-1) + n \times constant \\ &= T(n-2) + (n-1) \times constant + n \times constant \\ &= T(n-k) + k \times n \times constant - constant \times (k \times (k-1))/2 \end{aligned}$$

For $k = n$ we have

$$T(n) = T(0) + n \times n \times constant - constant \times (n \times (n-1))/2 = O(n^2).$$

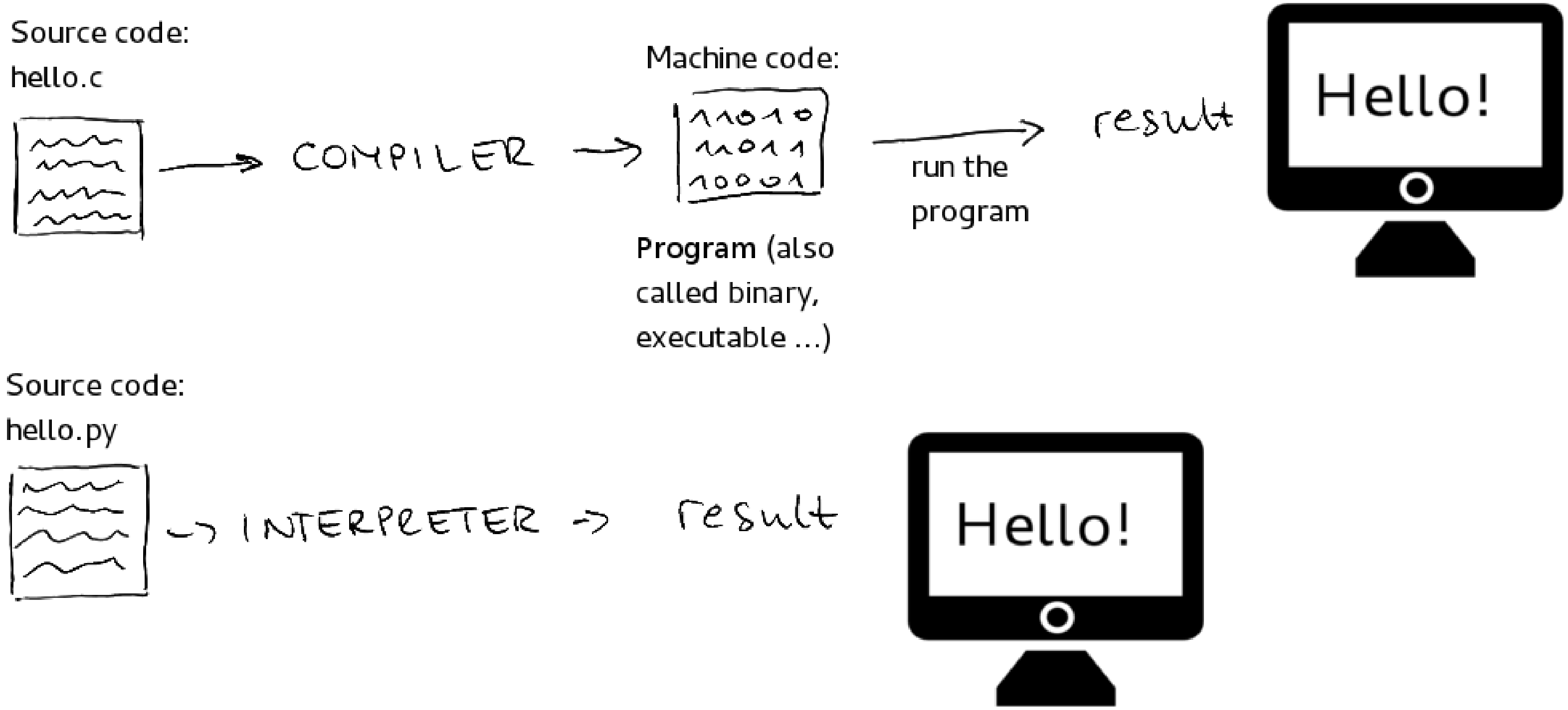
Average case: It can be shown that in the average case the complexity is $O(n \log n)$.

Complexity: a good online reference

`https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/`

Efficient coding in MATLAB: intepreted v/s compiled languages

There are (roughly) two types of programming languages: **interpreted** and **complied**



`MATLAB` is an interpreted programming language, but it has built-in pre-compiled functions $\Rightarrow$ try to use them as much as you can.

Representation of Numbers

Base-N numbers

Base: Number of unique digits to represent a number.

Example:

Decimal number (Base-10 number)

1. Uses digits from 0 to 9. So, base is 10.
2. Decimal number can be expanded in powers of 10.

   Examples:

   $582 = 5 \text{x} 10^2 + 8 \text{x} 10 + 2 \text{x} 10^0$,

   $25.36 = 2 \text{x} 10^1 + 5 \text{x} 10^0 + 3 \text{x} 10^{-1} + 6 \text{x} 10^{-2}$.

Base-N numbers (contd.)

Other examples:

- **Binary number:** Uses only two digits: 0 and 1.
- **Octal number:** Uses digits from 0 to 7.
- **Hexadecimal number:** Uses 16 digits: 0-9 followed by A-F or a-f.

Binary numbers

- Binary numbers are represented by only 0s and 1s. So, the base is 2.
- A digit in binary number is called a **bit**.
- **Example:** The number 1001000110.

  This number can be converted to a decimal number as follows:

$$1\text{x}2^9+0\text{x}2^8+0\text{x}2^7+1\text{x}2^6+0\text{x}2^5+0\text{x}2^4+0\text{x}2^3+1\text{x}2^2+1\text{x}2^1+0\text{x}2^0 = 582$$

Binary numbers (contd.)

Pros

- Easier for a computer to store and work with.
- A computer can do all arithmetic operations with them.
- A 32-bit computer can represent upto 2^{23} binary numbers.

Binary numbers (contd.)

Cons

- Harder for humans to do binary algebra.
- Limited **range** and **precision** to perform all mathematical calculations.

Floating Point Numbers

MATLAB uses **floating point** numbers or **floats** to achieve the desired range and precision required to perform mathematical calculations.

Note: Floating point numbers are always **rational**. MATLAB approximates irrational numbers (e.g. π) by rational numbers.

Types of float:

- single precision (32 bits)
- double precision (64 bits)

Single precision float

In the **IEEE754** standard for single precision, a float which is represented as

$$n = -1^s 2^{e-127}(1 + f)$$

Here, s, e and f are sign indicator, exponent and fraction respectively.

32 bits are allocated as follows:

- s has 1 bit so it takes values of 0 or 1.
- e has 8 bits so it can take 2^8 values.
- f has 23 bits so it can take 2^{23} values. $0 \leq f < 1$.

Floating Point Numbers (contd.)

Example:

Convert the number

1 10000011 1100000000000000000000 (IEEE754)

into decimal (base 10) number.

Floating Point Numbers (contd.)

Solution:

$s = 1$,
e=$1\text{x}2^7 + 0 + 0 + 0 + 0 + 0 + 1\text{x}2^1 + 1\text{x}2^0 = 131$,
f=$1\text{x}2^{-1} + 1\text{x}2^{-2} + \cdots = 0.75$.

$$n = -1^1 2^{131-127}(1 + 0.75) = -2^4\text{x}1.75 = -28.$$

Gaps between numbers

Not every decimal (base 10) number can be represented by double precision float. This causes gaps between numbers in MATLAB.

To find the gap, use MATLAB's eps function. eps(x) gives the gap between number x and next representable number.

Example:
eps(5)=8.881784197001252e-16.

The gap increases as numbers get large because the factor 2^{e-127} grows in size.

Special cases

1. e=0 (00000000(base 2))
   In this case, the float is calculated using

$$n = -1^s 2^{-126} f$$

2. e=255 (11111111(base 2))
   - $f \neq 0$: n=NaN (Not a Number)
   - f=0 and s=0: n=Inf
   - f=0 and s=1: n=-Inf

Floating Point Numbers (contd.)

Single precision float interesting facts

- Largest defined number
  0 11111110 11111111111111111111111
  (3.402823466385289e+38)

- Smallest defined positive ‘normal’ number
  0 00000001 00000000000000000000000
  (1.175494350822288e-38)
  (2^-126)

Note: Use MATLAB functions realmax(‘single’) and realmin(‘single’) to find the above results.

Floating Point Numbers (contd.)

- Smallest defined subnormal number
  0 00000000 00000000000000000001
  (1.401298464324817e-45)
  (2^-149)

Note: Use MATLAB single() function to represent a number in single precision.

Double precision float

MATLAB uses default double precision of **IEEE754** standard. A double precision is represented as

$$n = -1^s 2^{e-1023}(1+f)$$

Again, s, e and f are sign indicator, exponent and fraction respectively.

64 bits are allocated as follows:

- s has 1 bit so it takes values of 0 or 1.
- e has 11 bits so it can take 2^{11} values.
- f has 52 bits so it can take 2^{52} values. $0 \leq f < 1$.

Floating Point Numbers (contd.)

The special cases for double precision float are similar to single-precision float.

Double precision float interesting facts

- Largest defined number
  1.797693134862316e+308 (type realmax in command window)
- Smallest defined positive 'normal' number
  2.225073858507201e-308 (type realmin in command window)
- Smallest defined subnormal number
  4.940656458412465e-324
  (2^-1074)

Loss of precision

- Precision cannot be gained but can be lost.
- If you add the two quantities 0.171717 and 0.51, then the result should only have two significant digits.
- **Subtractive cancellation:** A loss of significance can be incurred if two nearly equal quantities are subtracted from one another!
  Example:

  $$0.177241 - 0.177589 = 0.348 \times 10^{-3}$$

  So three digits of accuracy have been lost.
- Can often be avoided by rewriting the expression.

Loss of precision

Example 1: Consider the stability of $\sqrt{x+1}-1$ when x is near zero. Say, $x = 1.2345678 \times 10^{-5}$, then $\sqrt{x+1} \approx 1.000006173$. In a 8 significant digits computer we'd have $\sqrt{x+1}-1 = 6.2 \times 10^{-6}$, so 6 digits of accuracy have been lost.

To fix this:

$$\sqrt{x+1}-1 = \sqrt{x+1}-1\frac{\sqrt{x+1}+1}{\sqrt{x+1}+1} = \frac{x}{\sqrt{x+1}+1}$$

No subtractions so no subtractive cancellation!

$$\frac{1.2345678 \times 10^{-5}}{2.0000062} \approx 6.17281995 \times 10^{-6}$$

Loss of precision

Example 2: Rewrite the roots of the quadratic equation $x^2 + bx + c = 0$, for when $b \gg c > 0$.

Loss of precision

Example 2: Rewrite $e^x - \cos x$ to be stable when x is near zero.