Estruturas de Dados / Programação 2
Eficiência de Algoritmos e Notação Big O

Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

Binary search

Array size	Time to find an element
1	1
3	2
7	1 + 2 = 3
15	1 + 3 = 4
31	1 + 4 = 5
2^t - 1	t

Given a array with n elements…

- … how long the binary search takes to find one particular element in this array? This is the interesting question!!!

Array size	Time to find an element
1	1
3	2
7	1 + 2 = 3
15	1 + 3 = 4
31	1 + 4 = 5
2^t - 1	t

Phonebook with 200.000 numbers…

- How many numbers we should look at?

$$\log_a(x) = \log_b(x) / \log_b(a)$$

$$\log_2 200000 = \log 200000 / \log 2 = 18$$

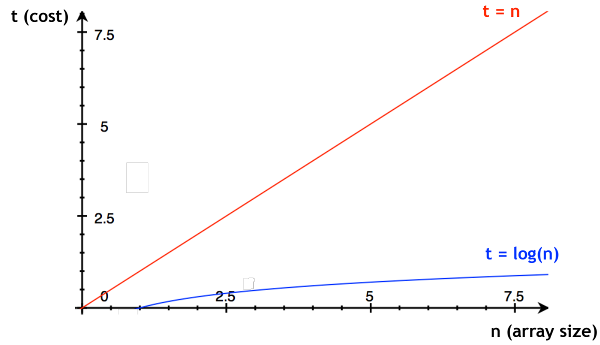
- Suppose it takes 30 seconds…
- What about 400.000 numbers?
  - It's NOT 60 seconds! This happens for Linear search!
  - 30 seconds + time to look at one name…

Binary search efficiency

$$t = \log(n)$$

- The time is proportional to the **logarithm** of the number of elements
- What about the linear search?

Linear search *versus* Binary search



Measures of Efficiency

What does "better algorithm" mean?

1 + 2 + 3 + … + 98 + 99 + 100

```
int sum_integers(int n);
```

```
int sum_integers(int n)
{
  int sum = 0;

  for (int i = 1; i <= n; i++) {
    sum = sum + i;
  }
  return sum;
}
```

What do you think about…

- … the following solution proposed by Karl Friedrich Gauss?

```
int sum_integers(int n)
{
  return (n + 1) * n / 2;
}
```

- One algorithm is "better" than another if it requires **less time** to execute!
- We must be aware of two resources: **time** and **space**!

T(n)

- Running time, where "n" is the size of the input
- Suppose a program that multiplies two n-digit numbers
  - $117.65n^2$ milliseconds on a microcomputer
  - $5.08n^2$ milliseconds on a mainframe
- Seems impossible to estimate timing for every computer!
- We just say that the algorithm is proportional to n^2…
- … and each machine only change the running time by a constant multiple, as a rule

Now, we can analyze T(n)

- In such a manner that
  - We will not concern ourselves with constant multiples
  - We will be concerned only with the largest estimates (worst cases)
- Notice that our function depends on the size input "n"!

So…

- Suppose that the running time of a statement is constant = 1
- The following statement does not depend on the array size

```
a = v[100];
```

- Linear search depends on the array size… The running time is proportional to the array size!

Big O notation

- Notation used to describe the effect of the input size on an algorithm performance

a = v[100]	**O(1)**
Linear Search	**O(n)**
Binary Search	**O(log(n))**

If $f(n)$ and $g(n)$ are two functions of n whose values are always positive, we say that $f(n) = O(g(n))$ if there is some constant c greater than zero for which $f(n) \leq c\ g(n)$ for all $n \geq n_0$ (i.e., for all n sufficiently large)

New algorithm!

- Someone has created a new algorithm
- She claims that…

t = f(n) = n + 1

- How to announce the efficiency in a formal way?

O(n)?

n + 1 = O(n)

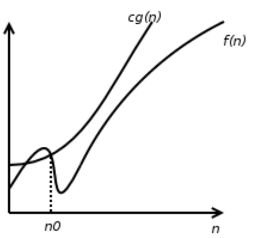
$f(n) = n + 1$ is $O(n)$; $g(n) = n$

$n + 1 \leq c.n$
$n + 1 \leq 2n$

$f(n) = O(n)$,
$c = 2$ and $n_0 = 1$

Therefore…

- *f(n)* is the exact complexity of an algorithm as a function of the problem size *n*, and that *g(n)* is an upper-bound on that complexity (i.e., the time for a problem of size *n* will be no worse than *g(n)*).



Try yourself!

$2n^2 + 2 = O(n^2)$

$f(n) = 2n^2 + 2$ is $O(n^2)$; $g(n) = n^2$

$2n^2 + 2 \leq c.n^2$
$2n^2 + 2 \leq 3n^2$

$f(n) = O(n^2)$,
$c = 3$ and $n_0 = 2$

Worst case and constants

- Usually describes the worst case running time…
- … guaranteeing that the algorithm performance will not be worse than the performance suggested
- No constants!
  - Big O expressions do not have constants
  - When “n” gets large enough, constants do not matter

Rules for Big O

Rule 1.
For any k, $k\ f(n) = O(f(n))$

Rule 2.
If $f(n) = O(g(n))$ and $g(n) = O(h(n))$,
then $f(n) = O(h(n))$

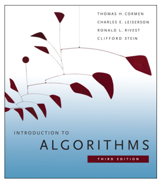
Rule 3.
$f_1(n) + g_1(n) = O(\max\{f_2(n),\ g_2(n)\})$

Exercise: prove rule 2

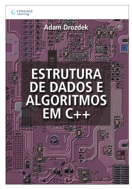
Rule 2.
If $f(n) = O(g(n))$ and $g(n) = O(h(n))$,
then $f(n) = O(h(n))$

$f(n) = O(g(n))$, there are constants c_1 and n_1,
So, $f(n) \le c_1.g(n)$ [1]
$g(n) = O(h(n))$, there are constants c_2 and n_2,
So, $g(n) \le c_2.h(n)$ [2]
Substituting [2] in [1], $f(n) \le c_1.(c_2.h(n))$
n_3 is the maximum of n_1 and n_2
Let $c_3 = c_1.c_2$, then for all $n > n_3$, we have
$f(n) \le c_3.h(n)$, which is, by definition,
$f(n) = O(h(n))$

References



Chapters 1, 2, and 3



Chapter 2