Algorithms and data structures 1: introduction, O-notation and some examples

Boris Kirikov

17.9.2015

Outline

1. Introduction
2. Fibonacci numbers
3. Estimating time complexity
4. Sorting: count sorting
5. Binary search
6. Two pointer method

Introduction

Fibonacci numbers

Def 1:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & , n > 1 \\ 1 & , n = 1 \\ 0 & , n = 0 \end{cases}$$

It can be easily proved, that F_n grows very fast: as fast as exponent.

Computing fibonacci numbers

```
int fib1(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib1(n-1) + fib1(n-2);
}
```

Computing fibonacci numbers

```
int fib1(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib1(n-1) + fib1(n-2);
}
```

- Is it correct?
- How much time it works?
- Is there a better solution?

Estimating time

Let $T(n)$ be a number of *computer steps* needed to compute F_n. If $n < 2$, than $T(n) < 2$ and for other n:

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

This yelds $T(n) > F_n$ which means exponential speed. Example: $T(200) \geq 2^{138}$, which will take millions of years for the most powerful machines.

Polynomial solution

Let's study how it works. A lot of exactly same calls. Let's keep results and not do same work again and again:

Polynomial solution

Let's study how it works. A lot of exactly same calls. Let's keep results and not do same work again and again:

```
int fib2(int n) {
  f[0] = 0;
  f[1] = 1;
  for (int i = 2; i <= n; ++i)
    f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

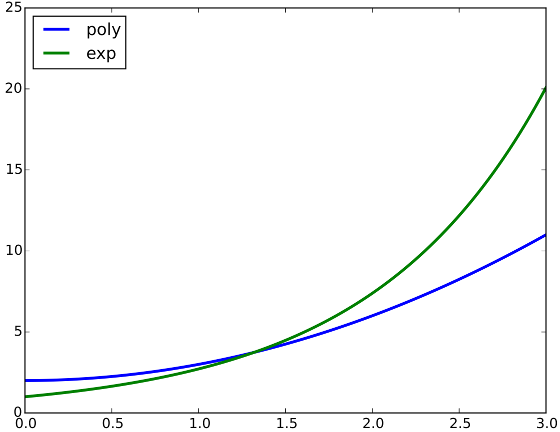
Polynomial solution

Let's study how it works. A lot of exactly same calls. Let's keep results and not do same work again and again:

```
int fib2(int n) {
  f[0] = 0;
  f[1] = 1;
  for (int i = 2; i <= n; ++i)
    f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

$T(n) = 2$ if $n < 2$ and $T(n) = n$ if $n \geq 2$.

Estimating working time



Big O and others

Def 1: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = O(g)$ if and only if exists constant C: $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$.

NOTE: f can be at the same time $O(g_1)$ and $O(g_2)$.

Big O and others

Def 1: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = O(g)$ if and only if exists constant C: $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$.

NOTE: f can be at the same time $O(g_1)$ and $O(g_2)$.

Def 2: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = \Omega(g)$ if and only if $g = O(f)$.

Big O and others

Def 1: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = O(g)$ if and only if exists constant C: $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$.

NOTE: f can be at the same time $O(g_1)$ and $O(g_2)$.

Def 2: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = \Omega(g)$ if and only if $g = O(f)$.

Def 3: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = \Theta(g)$ if and only if $f = O(g)$ and $g = O(f)$.

Big O and others

Def 1: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = O(g)$ if and only if exists constant C: $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$.

NOTE: f can be at the same time $O(g_1)$ and $O(g_2)$.

Def 2: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = \Omega(g)$ if and only if $g = O(f)$.

Def 3: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = \Theta(g)$ if and only if $f = O(g)$ and $g = O(f)$.

Def 4: $f, g : \mathbb{N} \to \mathbb{N}$. We say $f = o(g)$ if and only if $f/g \to 0$.

Some properties

- Constants and signs can be ommited
- If $a < b$ then $n^a = O(n^b)$

Some properties

- Constants and signs can be ommited
- If $a < b$ then $n^a = O(n^b)$

- Other properties

Comparing common functions

$$\log(n) \leq \sqrt{n} \leq n \leq n \log n \leq n^a \leq e^n$$

Comparing common functions

$$\log(n) \leq \sqrt{n} \leq n \leq n \log n \leq n^a \leq e^n$$

Real working times:

	n	$n \log n$	n^2	2^n
$n = 20$	1 sec	1 sec	1 sec	1 sec
$n = 50$	1 sec	1 sec	1 sec	13 days
$n = 100$	1 sec	1 sec	1 sec	10^{13} years
$n = 10^6$	1 sec	1 sec	17 sec	
$n = 10^9$	1 sec	5 min	30 years	

Sorting: count sort

- How fast can you sort array of N?

Sorting: count sort

- How fast can you sort array of N?
- How fast it is possible to sort?

Sorting: count sort

- How fast can you sort array of N?
- How fast it is possible to sort?
- Can we do it faster?

Sorting: count sort: idea

3	3	2	3	1	2	3	2	1

2	3	4

Sorting: count sort: complexity

Sorting: count sort: complexity

$O(n + m)$

Linear search

Task: given array and number, return index of number in array if present or -1 if not.

Linear search

Task: given array and number, return index of number in array if present or -1 if not.

```
int linsearch(int * A, int n, int k) {
  for (int i = 0; i < n; ++i) {
    if (A[i] == k) return i;
  }
  return -1;
}
```

Linear search

Task: given array and number, return index of number in array if present or -1 if not.

```
int linsearch(int * A, int n, int k) {
  for (int i = 0; i < n; ++i) {
    if (A[i] == k) return i;
  }
  return -1;
}
```

$O(n)$

Devide-and-conquer methods: binary search

Task: given array and number, return index of number in array if present or -1 if not. And array is **sorted**.

Devide-and-conquer methods: binary search

Task: given array and number, return index of number in array if present or -1 if not. And array is **sorted**.

```
int binsearch(int * A, int nl int k) {
  int l = 0;
  int r = n - 1;
  while (l <= r) {
    int m = (l + r)/2;
    if (A[m] == k) return m;
    else if (A[m] > k) r = m - 1;
    else if (A[m] < k) l = m + 1;
  }
  return -1;
}
```

Devide-and-conquer methods: binary search

Task: given array and number, return index of number in array if present or -1 if not. And array is **sorted**.

```
int binsearch(int * A, int nl int k) {
  int l = 0;
  int r = n - 1;
  while (l <= r) {
    int m = (l + r)/2;
    if (A[m] == k) return m;
    else if (A[m] > k) r = m - 1;
    else if (A[m] < k) l = m + 1;
  }
  return -1;
}
```

$O(\log n)$

Sum search task

Task: in array find two elements that sums to given number.

Sum search task

Task: in array find two elements that sums to given number.

```
pair<int, int> sumsearch(int * A, int n, int S) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if (A[i] + A[j] == S) {
        return pair<int, int>(i, j);
      }
    }
  }
  return -1;
}
```

Sum search task

Task: in array find two elements that sums to given number.

```
pair<int, int> sumsearch(int * A, int n, int S) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if (A[i] + A[j] == S) {
        return pair<int, int>(i, j);
      }
    }
  }
  return -1;
}
```

$O(n^2)$

Two pointers method

Task: in array find two elements that sums to given number. And array is **sorted**.

Two pointers method

Task: in array find two elements that sums to given number. And array is **sorted**.

```
pair<int, int> twopointers(int * A, int n, int S) {
  int l = 0, r = n - 1;
  while (l < r) {
    if (A[l] + A[r] == S) return pair<>(i, j);
    else if (A[l] + A[r] > S) r--;
    else if (A[l] + A[r] < S) l++;
  }
  return -1;
}
```

Two pointers method

Task: in array find two elements that sums to given number. And array is **sorted**.

```
pair<int, int> twopointers(int * A, int n, int S) {
  int l = 0, r = n - 1;
  while (l < r) {
    if (A[l] + A[r] == S) return pair<>(i, j);
    else if (A[l] + A[r] > S) r--;
    else if (A[l] + A[r] < S) l++;
  }
  return -1;
}
```

$O(n)$

EOF