Introduction and Big-O notation

Bronson Rudner

South African Programming Olympiad
Training Camp

December 12, 2020

What do I need to know to solve programming problems?

What do I need to know to solve programming problems?

- Basic coding skills and fluency

What do I need to know to solve programming problems?

- Basic coding skills and fluency
- Knowledge of algorithms and data structures

What do I need to know to solve programming problems?

- Basic coding skills and fluency
- Knowledge of algorithms and data structures
- Ability to problem solve

Todays agenda

We are focusing on algorithms and data structures. Why?

Todays agenda

We are focusing on algorithms and data structures. Why?

- Learning to code these will ensure you develop your basic coding skills and fluency

Todays agenda

We are focusing on algorithms and data structures. Why?

- Learning to code these will ensure you develop your basic coding skills and fluency
- Many problems reduce to one of, or some subset of these

Todays agenda

We are focusing on algorithms and data structures. Why?

- Learning to code these will ensure you develop your basic coding skills and fluency
- Many problems reduce to one of, or some subset of these
- They provide inspiration for further solutions

Todays agenda

We are focusing on algorithms and data structures. Why?

- Learning to code these will ensure you develop your basic coding skills and fluency
- Many problems reduce to one of, or some subset of these
- They provide inspiration for further solutions
- Serve as a guideline for solving problems efficiently

Types of problems in programming contests

1. Ad-Hoc
2. Greedy
3. Computational Geometry
4. Dynamic Programming
5. BigNums
6. Two-Dimensional
7. Eulerian Path
8. Minimum Spanning Tree
9. Knapsack
10. Network Flow
11. Flood Fill
12. Shortest Path
13. Approximate Search
14. Complete Search
15. Recursive Search Techniques
16. Heuristic Search

How fast are computers?

- Computers are fast - but their speed is still finite

How fast are computers?

- Computers are fast - but their speed is still finite
- On the order of 100 000 000 operations per second (C++)

How fast are computers?

- Computers are fast - but their speed is still finite
- On the order of 100 000 000 operations per second (C++)
- Java about 2x slower, Python about 10x slower (and Scratch is about 100x slower)

How fast are computers?

- Computers are fast - but their speed is still finite
- On the order of 100 000 000 operations per second (C++)
- Java about 2x slower, Python about 10x slower (and Scratch is about 100x slower)
- It isn't enough to find an algorithm that solves a problem

How fast are computers?

- Computers are fast - but their speed is still finite
- On the order of 100 000 000 operations per second (C++)
- Java about 2x slower, Python about 10x slower (and Scratch is about 100x slower)
- It isn't enough to find an algorithm that solves a problem
- It needs to solve it within the time limit

E.g.

- The Fibonacci series is given by

$$1, 1, 2, 3, 5, 8, \ldots$$

  where the next number in the sequence is given by the sum of the two preceding numbers

E.g.

- The Fibonacci series is given by

$$1, 1, 2, 3, 5, 8, \ldots$$

  where the next number in the sequence is given by the sum of the two preceding numbers
- Find the n^{th} Fibonacci number, given that $1 \leq n \leq 1\,000$

E.g.

- The Fibonacci series is given by

$$1, 1, 2, 3, 5, 8, \ldots$$

  where the next number in the sequence is given by the sum of the two preceding numbers
- Find the n^{th} Fibonacci number, given that $1 \leq n \leq 1\,000$
-
```python
def slow_fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return slow_fibonacci(n-1) + slow_fibonacci(n-2)
```

E.g.

- The Fibonacci series is given by

$$1, 1, 2, 3, 5, 8, \ldots$$

  where the next number in the sequence is given by the sum of the two preceding numbers
- Find the n^{th} Fibonacci number, given that $1 \leq n \leq 1\,000$
-
```python
def slow_fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return slow_fibonacci(n-1) + slow_fibonacci(n-2)
```
- Increasing n by 1, (roughly) doubles the total number of calls of `slow_fibonacci`.

E.g.

- The Fibonacci series is given by

$$1, 1, 2, 3, 5, 8, \ldots$$

  where the next number in the sequence is given by the sum of the two preceding numbers
- Find the n^{th} Fibonacci number, given that $1 \leq n \leq 1\,000$
-
```python
def slow_fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return slow_fibonacci(n-1) + slow_fibonacci(n-2)
```
- Increasing n by 1, (roughly) doubles the total number of calls of `slow_fibonacci`.
- $n = 40$ calls `slow_fibonacci` about 200 000 000 times!

Big-O notation

- Gives a rough idea of the runtime of a program / function

Big-O notation

- Gives a rough idea of the runtime of a program / function
- Is expressed in relation to the size of the input (n).

Big-O notation

- Gives a rough idea of the runtime of a program / function
- Is expressed in relation to the size of the input (n).
- If a program is $\mathrm{O}(f(n))$, we mean it takes no more than $C \cdot f(n)$ steps in total, for suitably large n (for some constant C).

Big-O notation

- Gives a rough idea of the runtime of a program / function
- Is expressed in relation to the size of the input (n).
- If a program is $\mathrm{O}(f(n))$, we mean it takes no more than $C \cdot f(n)$ steps in total, for suitably large n (for some constant C).
- In particular, if a program takes $T(n)$ steps, it is $\mathrm{O}(T(n))$.

Big-O notation

It is an upper bound:

$$n = O(n^2), \quad n^2 = O(n^2), \quad 1 = O(n)$$

Big-O notation

It is an upper bound:

$$n = O(n^2), \quad n^2 = O(n^2), \quad 1 = O(n)$$

We ignore constant factors:

$$3n^2 = O(n^2), \quad \frac{1}{2}n = O(n)$$

Big-O notation

It is an upper bound:

$$n = O(n^2), \quad n^2 = O(n^2), \quad 1 = O(n)$$

We ignore constant factors:

$$3n^2 = O(n^2), \quad \frac{1}{2}n = O(n)$$

We only care about the largest term:

$$n^2 + n = O(n^2), \quad 2n^2 + 3n + 1000 = O(n^2)$$

Big-O notation

- We usually consider the worst case bound
  - E.g. linear search can be $O(1)$ in the best case, but in the worst case, and on average, it is $O(n)$

Big-O notation

- We usually consider the worst case bound
  - E.g. linear search can be $O(1)$ in the best case, but in the worst case, and on average, it is $O(n)$
- If you are sure the data is suitably random, you could use average time bound

Big-O notation

- We usually consider the worst case bound
  - E.g. linear search can be O(1) in the best case, but in the worst case, and on average, it is O(n)
- If you are sure the data is suitably random, you could use average time bound
- Can also describe memory of a program - but usually you are given more memory than the time bound anyway

E.g.

What is the Big-O of the following function?

```python
def triangular_nums(n):
    nums = []
    for i in range(n):
        num = 0
        for j in range(i+1):
            num += j+1
        nums.append(num)
    return nums
```

E.g.

What is the Big-O of the following function?

```
def triangular_nums(n):
    nums = []
    for i in range(n):
        num = 0
        for j in range(i+1):
            num += j+1
        nums.append(num)
    return nums
```

Answer: $O(n^2)$

E.g.

What is the Big-O of the following function?

```
def is_prime(n):
    if n % 2 == 0:
        return False
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True
```

E.g.

What is the Big-O of the following function?

```
def is_prime(n):
    if n % 2 == 0:
        return False
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True
```

Answer: $O(\sqrt{n})$

E.g.

What is the Big-O of the following function?

```python
def foo(nums1, nums2):
    total = 0
    for x in nums1:
        total += x
    for x in nums1:
        for y in nums2:
            total += x * y
    return total
```

E.g.

What is the Big-O of the following function?

```
def foo(nums1, nums2):
    total = 0
    for x in nums1:
        total += x
    for x in nums1:
        for y in nums2:
            total += x * y
    return total
```

Answer: $O(nm)$ (where n, m is the size of `nums1`, `nums2`, respectively)

E.g.

What is the Big-O of the following function?

```
def sum_powers_of_two(n):
    total = 0
    i = 1
    while i < n:
        total += i
        i *= 2
    return total
```

E.g.

What is the Big-O of the following function?

```python
def sum_powers_of_two(n):
    total = 0
    i = 1
    while i < n:
        total += i
        i *= 2
    return total
```

Answer: $O(\log n)$
$(\log 2^x = x \text{ (base 2)})$

E.g.

What is the Big-O of the following function?

```
def slow_fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return slow_fibonacci(n-1) + slow_fibonacci(n-2)
```

E.g.

What is the Big-O of the following function?

```
def slow_fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return slow_fibonacci(n-1) + slow_fibonacci(n-2)
```

Answer: $O(2^n)$

E.g.

What is the Big-O of the following function?

```
def slow_fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return slow_fibonacci(n-1) + slow_fibonacci(n-2)
```

Answer: $O(2^n)$
(can be tightened to $O(1.618^n)$)

E.g.

What is the Big-O of the following function?

```
def sum_values_in_binary_tree(node):
    sum = node.value
    if node.left is not None:
        sum += sum_values_in_binary_tree(node.left)
    if node.right is not None:
        sum += sum_values_in_binary_tree(node.right)
    return sum
```

E.g.

What is the Big-O of the following function?

```
def sum_values_in_binary_tree(node):
    sum = node.value
    if node.left is not None:
        sum += sum_values_in_binary_tree(node.left)
    if node.right is not None:
        sum += sum_values_in_binary_tree(node.right)
    return sum
```

Answer: $O(n)$ (where n is the number of nodes in the tree)

Will my program run in time?

Follow this procedure:

Will my program run in time?

Follow this procedure:

- Determine the Big-O of your algorithm. E.g. $O(nk \log n)$.

Will my program run in time?

Follow this procedure:

- Determine the Big-O of your algorithm. E.g. $O(nk \log n)$.
- Plug in the constraints of the question. E.g. $n \leq 10\,000, k \leq 50$. Then $nk \log n \approx (10\,000)(50)(13) = 6\,500\,000$

Will my program run in time?

Follow this procedure:

- Determine the Big-O of your algorithm. E.g. $O(nk \log n)$.
- Plug in the constraints of the question. E.g. $n \leq 10\,000$, $k \leq 50$. Then $nk \log n \approx (10\,000)(50)(13) = 6\,500\,000$
- Your result should be reasonable amount less than 100 000 000 - typically less than 10 000 000 is reasonable. You may multiply this 10 000 000 by the time limit in seconds.

Common classes of Big-O

Class	Big-O	Typical upper limit on n	$n = 1\,000\,000$
Constant	1		1
Logarithmic	$\log n$		20
Square root	$\sqrt{n}$	10^{13}	1000
Linear	n	5 000 000	
Linearithmic	$n \log n$	200 000	
Quadratic	n^2	5 000	
Cubic	n^3	200	
Exponential	2^n	24	
Factorial	$n!$	11	