Big-Oh Notation

2014 IOI Camp 1

Robert Spencer

December 11, 2013

Introduction

How long will the following code take to run?

How long will the following code take to run?

```
p = 0
for i = 1 to 10**6 do
  for j = 1 to i do
    for k = 1 to j do
      p += k
print p
```

Introduction

How long will the following code take to run?

```
p = 0
for i = 1 to 10**6 do
  for j = 1 to i do
    for k = 1 to j do
      p += k
print p
```

A minute? An hour? How about 31 millenia?

Competition Application

- Obviously, in competitions we are interested in how quickly a solution runs.

```
1 second
2 seconds
```

Competition Application

- Obviously, in competitions we are interested in how quickly a solution runs.

  `1 second`

  `2 seconds`

- Different languages differ by small amounts. On average, can do the same number of operations a second.

Competition Application

- Obviously, in competitions we are interested in how quickly a solution runs.

  `1 second`
  `2 seconds`

- Different languages differ by small amounts. On average, can do the same number of operations a second.
- Good rule of thumb: If you are doing more than 1 000 000 things a second, you have a problem.

Rules for Calculating Order of Magnitude

- Ignore constants:
$$O(3n^2) = O(n^2)$$

Rules for Calculating Order of Magnitude

- Ignore constants:

$$O(3n^2) = O(n^2)$$

  - This means that all logarithms are whatever base you want because they differ by a constant. E.g. binary searches (complexity $O(\log_2 n)$) are the same order as ternary searches (complexity $O(\log_3 n)$).

Rules for Calculating Order of Magnitude

- Ignore constants:

$$O(3n^2) = O(n^2)$$

  - This means that all logarithms are whatever base you want because they differ by a constant. E.g. binary searches (complexity $O(\log_2 n)$) are the same order as ternary searches (complexity $O(\log_3 n)$).

- Take the biggest term:

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

Rules for Calculating Order of Magnitude

- Ignore constants:

$$O(3n^2) = O(n^2)$$

  - This means that all logarithms are whatever base you want because they differ by a constant. E.g. binary searches (complexity $O(\log_2 n)$) are the same order as ternary searches (complexity $O(\log_3 n)$).

- Take the biggest term:

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

- Worst case bound: linear search is $O(n)$ even though it may only take one step.

Constant Time

Some operations take “constant time”: they are independant on the size of any parameter.

Constant Time

Some operations take "constant time": they are independant on the size of any parameter.
Example: accessing values from an array

```
array[i] += array[3]
```

Constant Time

Some operations take "constant time": they are independant on the size of any parameter.
Example: accessing values from an array

```
array[i] += array[3]
```

Some problems can even be solved in constant time!

- Problems with formulae (eg "Find the number of numbers less than n with their third bit set to 1 in binary expansion").
- Problems that can be hard coded (eg "Find the nth prime where $1 \leq n \leq 1\ 000\ 000$").

Classes of Big-Oh

From this we can work out various "classes" of Big-Oh values, and reasonable values of n.

Order	Reasonable value for N
$O(N)$	1 000 000
$O(N^2)$	1 000
$O(N^3)$	100
$O(N \log N)$	50 000

Example 1

What is the complexity of this algorithm?

```
for i = 1 to n do
  for j = 1 to i do
    if a[j] > a[j+1] then
      swap(a[j], a[j+1])
```

Example 1

What is the complexity of this algorithm?

```
for i = 1 to n do
  for j = 1 to i do
    if a[j] > a[j+1] then
      swap(a[j], a[j+1])
```

Answer: $O(n^2)$

Example 2

What is the complexity of this algorithm?

```
begin = 1
end = 2
count = 0
while end < n:
  if a[begin] == a[end] then
    count++
  if a[begin] > a[end] then
    end++
  else
    begin++
```

Example 2

What is the complexity of this algorithm?

```
begin = 1
end = 2
count = 0
while end < n:
  if a[begin] == a[end] then
    count++
  if a[begin] > a[end] then
    end++
  else
    begin++
```

Answer: $O(n)$

Problem Solving with Big-Oh

How can we use Big-Oh to solve problems?

Problem Solving with Big-Oh

How can we use Big-Oh to solve problems?
The evaluator has to run in time. Thus we can see what Big-Oh class the evaluator uses from the constraints. This gives us hints on the problem solution.

Problem Solving with Big-Oh

How can we use Big-Oh to solve problems?
The evaluator has to run in time. Thus we can see what Big-Oh class the evaluator uses from the constraints. This gives us hints on the problem solution.

- If we are given $1 < n < 50\ 000$, we can reasonably assume a solution with $O(n \log n)$, e.g. sorting

Problem Solving with Big-Oh

How can we use Big-Oh to solve problems?
The evaluator has to run in time. Thus we can see what Big-Oh class the evaluator uses from the constraints. This gives us hints on the problem solution.

- If we are given $1 < n < 50\ 000$, we can reasonably assume a solution with $O(n \log n)$, e.g. sorting
- If all constraints are small, e.g. $1 < n, m < 20$, then a very inefficient solution is possible (think $O(n^2 m^3)$).

Example 3

What is the complexity of this algorithm?

```
hi = n
lo = 0
while guess((hi+lo)/2) is false
  if (hi+lo)/2 too high
    hi = (hi+lo)/2 - 1
  else
    lo = (hi+lo)/2 + 1
```

Example 3

What is the complexity of this algorithm?

```
hi = n
lo = 0
while guess((hi+lo)/2) is false
  if (hi+lo)/2 too high
    hi = (hi+lo)/2 - 1
  else
    lo = (hi+lo)/2 + 1
```

Answer: $O(\log n)$

Example 4

What is the complexity of this algorithm?

```
def recurse (left, right)
    for i = left to right do
      a[i]++
    middle = (left+right)/2
    if left < middle then
      recurse(left, middle)
```

Example 4

What is the complexity of this algorithm?

```
def recurse (left, right)
    for i = left to right do
      a[i]++
    middle = (left+right)/2
    if left < middle then
      recurse(left, middle)
```

Answer: $O(n)$

Example 5

What is the complexity of this algorithm?

```
def recurse (left, right)
    for i = left to right do
      a[i]++
    middle = (left+right)/2
    if left < middle then
      recurse(left, middle)
    if middle + 1 < right then
      recurse(middle + 1, right);
```

Example 5

What is the complexity of this algorithm?

```
def recurse (left, right)
    for i = left to right do
      a[i]++
    middle = (left+right)/2
    if left < middle then
      recurse(left, middle)
    if middle + 1 < right then
      recurse(middle + 1, right);
```

Answer: $O(n \log n)$

Well Known Algorithms

Here are the complexities of some well known algorithms:

Algorithm	Complexity
Sorting (in general)	$O(n \log n)$
Binary Search	$O(\log n)$
DFS (visit once only)	$O(V + E)$
BFS (visit once only)	$O(V + E)$
Basic Dijkstra's	$O((E + V) \log V)$
Kruskal's and basic Prim's	$O(E \log E)$
Naive substring finding	$O(nk)$
Knuth-Morris-Pratt substring finding	$O(n + k)$
Rabin-Karp substring finding	$O(n + k)$