Algorithms and their complexity
Lecture 12

Waterford Institute of Technology

March 1, 2016

John Fitzgerald

Presentation Outline

Estimated duration presentation

Questions at end presentation

Topics discussed:

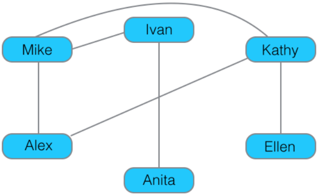
- Methods to measure program complexity
- Example algorithms
  - Sorting
  - Searching
- Relative performance of algorithms

Algorithm

Description

What is a software algorithm?

- Program that solves problem
- Algorithm examples:
  - Sorting data
  - Searching data
  - Graphs (social networks)
  - Cryptography
  - Image processing
  - From labs:
    - Generate PIN
    - Validate PIN



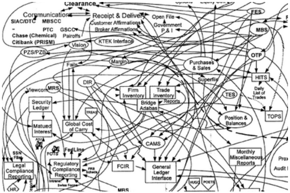
Modelling social network

Algorithm

Requirements

In developing an algorithm we are interested in:

- Does it produce correct answer for all valid inputs?
- How long to solve task?
- How much computer memory used?
- Is code easy to understand & thus maintain?
  - Conceptual complexity
  - Computational complexity



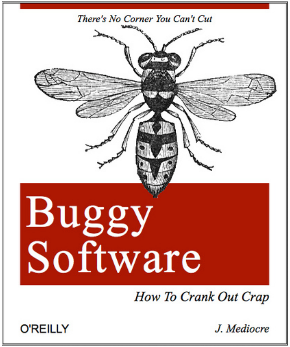
Algorithm output

Expect correct result for all valid inputs

Easier said than done

Examples of incorrect output:

- Timsort
  - Extensively used on Android
  - Bug recently discovered
- Binary search
  - Small & conceptually simple
  - Yet took years to eliminate all bugs for all inputs



Algorithm performance

How long to run?

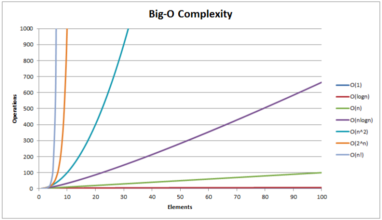
Goal not measurement run time because of variations in

- Computer speeds
- Language implementation
- Inputs

Instead, determine inherent complexity

- Classes of complexity exist
- Provide comparison of growth of number computing operations

Big-O Complexity Chart



Linear search algorithm

How long to run?

Consider the linear search algorithm below

Running times could vary significantly:

- If *String search* near start of *String[] target*
  - method returns immediately
- But if towards end of list
  - return might be much later
  - providing very different run times for same algorithm

```
boolean linearSearch(String search, String[] target)
{
  for (int i = 0; i < target.length; i += 1)
    if (target[i].contains(search))
      return true;
  return false;
 }
```

Algorithm efficiency

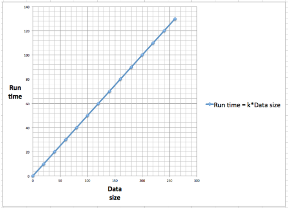
Choices available

We have a choice to make:

- Best case
  - Considering all possible inputs
  - Linear search runs in constant time
  - Returns immediately
- Worst case
  - Search time proportional input size
  - Linear search is *linear* in size list
- Average

We will choose worst case.

- We will seek an upper bound on number operations in algorithm.

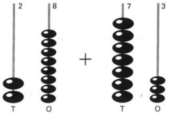


Counting operations

Agree some rules

Basic assumptions

- Concept of random access machine (RAM)
- An abstract computer
- Measure size of input
- Sequential execution operations
- Operations take constant time
  - Assignment (`a = 10`)
  - Comparison (`a==10`)
  - Arithmetic (`a/10`)
  - Memory access (`b[i]`)



Counting operations

Simple example

Approximate number operations or steps in *countOperations*:

- $2001 + 2n + 2n^2$

```
static int countOperations(int n)
{
  int ans = 0;
  for (int i = 0; i < 1000; i += 1)
    ans += 1;
  for (int i = 0; i < n; i += 1)
     ans +=1;
  for (int i = 0; i < n; i += 1)
    for (int j = 0; j < n; j += 1)
      ans += 1;
  return ans;
}
```

Number operations

1

2000

$2n$

$2n^2$

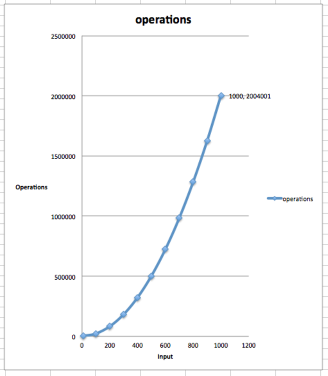
Count operations

Number steps or operations

$2001 + 2n + 2n^2$ (ignoring *for* overhead)

- If n small output dominated by 2000 term
- For large n 2000 term insignificant
  - If n is 10 output is 2221
  - If n is 10,000 output exceeds 200 million (2.0002E+08)

```java
static int countOperations(int n)
{
  int ans = 0;
  for (int i = 0; i < 1000; i += 1)
    ans += 1;
  for (int i = 0; i < n; i += 1)
     ans +=1;
  for (int i = 0; i < n; i += 1)
    for (int j = 0; j < n; j += 1)
      ans += 1;
  return ans;
}
```

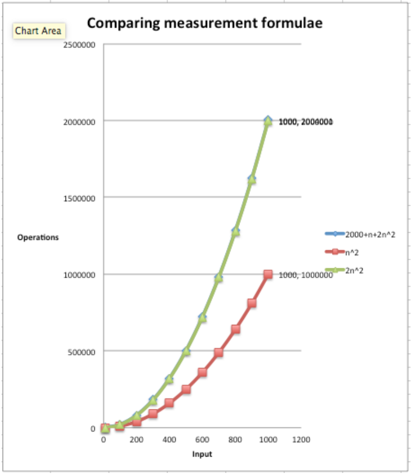


Algorithm complexity

Comparing measurement formulae

In the case of $2001 + 2n + 2n^2$ we use n^2

- We use the highest order variable
- And disregard any coefficient
- Provides approximate upper bound measurement
- Importantly, it provides representation of growth of complexity as input becomes very large.



Algorithm complexity

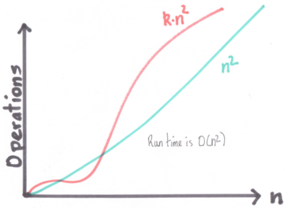
Use of asymptotic notation

Last example: Some constant times n^2 provides upper bound on number operations

Asymptotic notation used to describe growth of algorithm operations as input size approaches infinity.

We now introduce **Big O** notation

- Running time of $\mathbf{2001 + 2n + 2n^2}$
  - $\mathbf{O(n^2)}$ or
  - **Big-O**$\mathbf{(n^2)}$

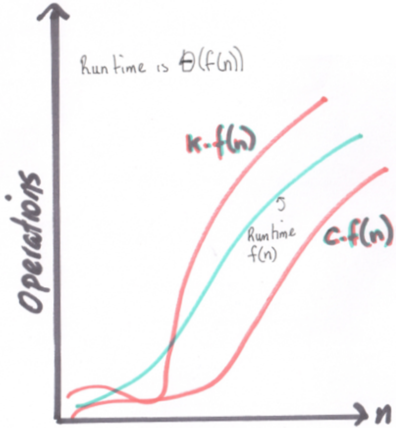


Algorithm complexity

Further asymptotic notation

Big Θ (Big Theta)

- For large n :
  - $k \cdot f(n)$ provides upper bound
  - $c \cdot f(n)$ provide lower bound
- k and c are some constants.

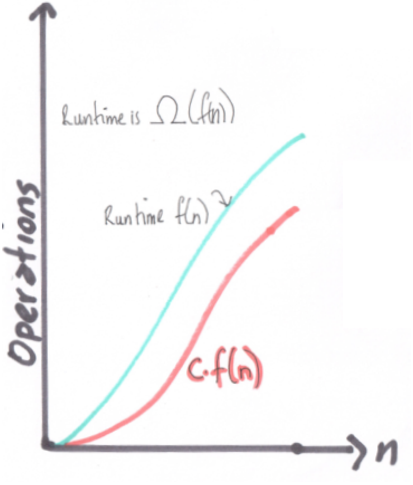


Algorithm complexity

Further asymptotic notation

Big Ω (Big Omega)

- For large n, $c \cdot f(n)$ provides lower bound run time
- c is some constant.

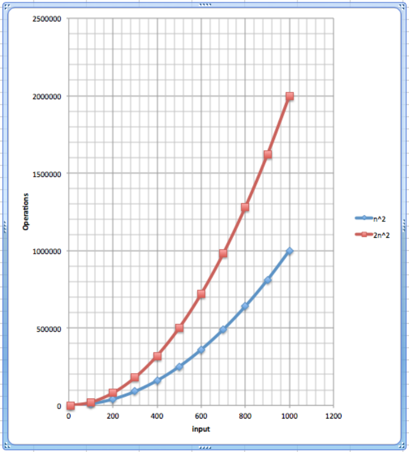


Algorithmic complexity

Alternative measurement approach

Tilde notation

- Proposed by Robert Sedegwick
- Denoted by: ~f(n)
- Differs from Big-O
  - Coefficient of term with highest exponent retained
- Consider: $f(x) = 2000 + 2n + 2n^2$
  - $\sim 2n^2$ is considered the complexity of this algorithm



Big-O complexity

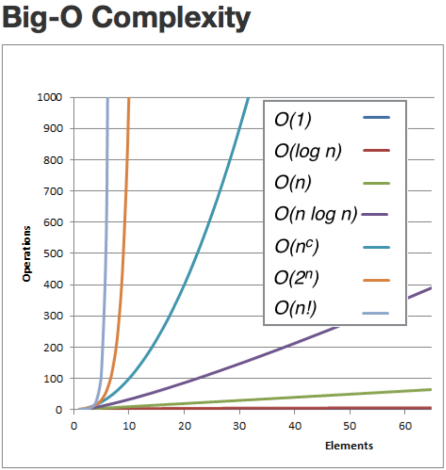
Categories

- Constant O(1) : independent of input
- Linear O(n): find smallest item unsorted array
- Logarithmic O(log n) : binary search
- Linearithmic O(n log n) : mergesort
- Polynomial $O(n^c)$, $c > 0$: selection sort quadratic ($c == 2$)
- Exponential $O(2^n)$: Binary exponential backoff (networking)
- Factorial O(n!) : Brute force search travelling salesman problem

Big-O complexity

Categories

- Constant O(1)
- Logarithmic O(log n)
- Linear O(n)
- Linearithmic O(n log n)
- Polynomial O(n^c)
- Exponential O(2^n)
- Factorial O(n!)



Algorithm complexity

Acceptable categories

Very large performance differences between categories

Does this really matter?

- For small problems, not really
- For very large problems, definitely yes.

Size array	Merge sort	Selection sort
20,000	2ms	868ms
100,000	10ms	13s 230ms
1,000,000	117ms	21m 55s 254ms
500,000,000	2m 2s 652ms	10.25 years

Sort algorithms

Selection sort

{20, 24, 17, 12, 11, 14, 22, 19}

{11, 12, 14, 17, 20, 24, 22, 19}

Sort algorithms

Merge sort

This implementation comprises two methods:

- public method **sort**
- private method **merge**

```
public sort
loop
  subdivide array into pairs blocks
  loop all pairs blocks
    invoke merge on each pair
  endloop
endloop
```

```
private merge
Successively compare elements
in left and right blocks.
Incrementally populate target sorted
     array.
```

Merge sort

Method sort

```
Before merge call
20 24 17 12 11 14 22 19
After merge call
20 24 17 12 11 14 22 19
Before merge call
20 24 17 12 11 14 22 19
After merge call
20 24 12 17 11 14 22 19
Before merge call
20 24 12 17 11 14 22 19
After merge call
20 24 12 17 11 14 22 19
Before merge call
20 24 12 17 11 14 22 19
After merge call
20 24 12 17 11 14 19 22
```

Block size 1

Invoke `merge` to sort each pair of blocks.

Legend

20 24 unsorted

20 24 sorted

Merge sort

Method sort

```
Before merge call
20 24|17 12 11 14 22 19
After merge call
12 17|20 24 11 14 22 19
Before merge call
12 17 20 24 11 14|22 19
After merge call
12 17 20 24 11 14|19 22
```

Block size 2

Invoke `merge` to sort each pair of blocks.

Legend

unsorted

11 14|22 19

sorted

11 14|19 22

Merge sort

Method sort

Before merge call

12	17	20	24	11	14	19	22

After merge call

11	12	14	17	19	20	22	24

Block size 4

Invoke `merge` to sort each pair of blocks.

Merge sort

Method merge

12	17	20	24	11	14	19	22

11	0	0	0	0	0	0	0
11	12	0	0	0	0	0	0
11	12	14	0	0	0	0	0
11	12	14	17	0	0	0	0
11	12	14	17	19	0	0	0
11	12	14	17	19	20	0	0
11	12	14	17	19	20	22	0
11	12	14	17	19	20	22	24

11	12	14	17	19	20	22	24

Block size 4

Invoking merge

Indices

i : left blocks
j : right blocks
k : sorted array

```
for (int k = lo; k <= hi; k++)
{
  if (i > mid)                a[k] = aux[j++];
  else if (j > hi)            a[k] = aux[i++];
  else if (aux[j] < aux[i])   a[k] = aux[j++];
  else                        a[k] = aux[i++];
}
```

Search algorithms

Binary search

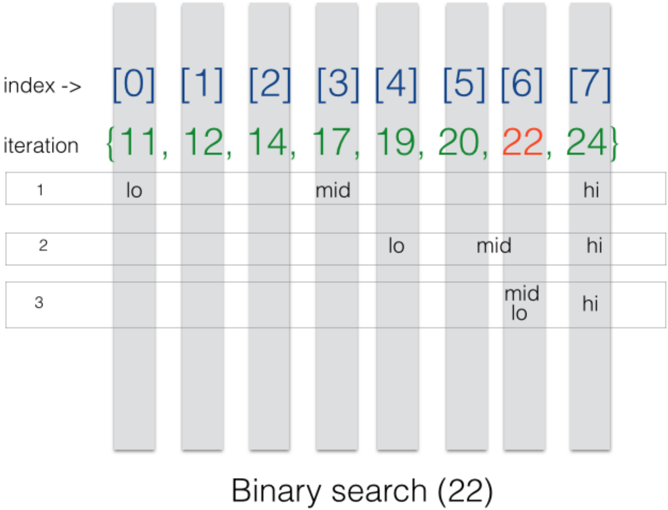
Linear search complexity is O(n).

A more efficient *Binary search* has complexity O(log n)

- Works with a sorted array
- Array considered as upper and lower half
- Check carried out: is item in upper or lower?
- Check repeated in the half array containing item.
- This search method repeated until item found if it exists.
- Each iteration reduces the search space in two.
- This explains the *O(log n)* order of complexity.

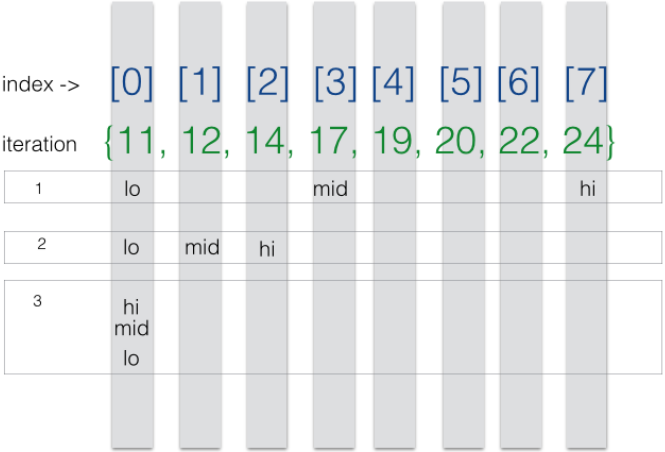
Search algorithms

Binary search



Search algorithms

Binary search



Binary search (0)

Big O Notation

Time classification of algorithms

- One method of categorizing algorithmic processing time

	constant	*logarithmic*	*linear*		*quadratic*	*cubic*
n	O(1)	O(log N)	O(N)	O(N log N)	$O(N^2)$	$O(N^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824
1,048,576	1	20	1,048,576	20,971,520	10^{12}	10^{16}

Big O Notation
Sorting

Important to have regard to

- Best
- Average
- Worst

Type of Sort	*Best*	*Worst*	*Average*	*Comments*
BubbleSort	O(N)	$O(N^2)$	$O(N^2)$	Not a good sort, except with ideal data.
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	Perhaps best of $O(N^2)$ sorts
QuickSort	O(N log N)	$O(N^2)$	O(N log N)	Good, but it worst case is $O(N^2)$
HeapSort	O(N log N)	O(N log N)	O(N log N)	Typically slower than QuickSort, but worst case is much better.

Summary

- Methods to measure complexity
  - Elapsed time
  - Big O
  - Big Θ
  - Big Ω
  - Tilde notation
- Example algorithms
  - Selection sort
  - Merge sort
  - Linear search
  - Binary search
- Relative performance of algorithms

Referenced Material

1. Algorithms (Khan Academy)

www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation

[Accessed 2015-03-03]

2. TimSort bug fixed with formal methods http://www.cwi.nl/news/2015/java-bug-fixed-formal-methods-cwi

[Accessed 2015-03-03]

3. MOOC: edX | MITx: 6.00.1x Grimson. Introduction to Computer Science and Programming Using Python

Massachusetts Institute of Technology

https://www.edx.org

[Accessed 2015-03-05]

Referenced Material

4. MOOC: Algorithms I: Sedgewick & Wayne. Princeton University.

https://www.coursera.org/course/algs4partI

[Accessed 2015-03-03]

5. Robert Sedgewick : Algorithms for the masses

http://osric.com/chris/accidental-developer/2012/04/robert-sedgewick-algorithms-for-the-masses/ [Accessed 2015-03-05]