CE0825a: Object Oriented Programming II
11: Algorithms, Structures, Searching

James A Sutherland

Abertay University

Monday, 21st March 2016

Algorithms

- The basic tools of programming: sorting, searching...
- How do we compare them? Which is 'best'?
    - CPU usage
    - Memory usage
    - Quality (Zopfli takes 1s to shave 35k off this PDF)
    - Note: Often a trade-off between those
    - LZMA2 ('Ultra', 8 threads): over 4Gb!

Algorithms

- The basic tools of programming: sorting, searching. . .
- How do we compare them? Which is 'best'?
  - CPU usage
  - Memory usage
  - Quality (Zopfli takes 1s to shave 35k off this PDF)
  - Note: Often a trade-off between those
  - LZMA2 ('Ultra', 8 threads): over 4Gb!

Algorithms

- The basic tools of programming: sorting, searching. . .
- How do we compare them? Which is 'best'?
  - CPU usage
  - Memory usage
  - Quality (Zopfli takes 1s to shave 35k off this PDF)
  - Note: Often a trade-off between those
  - LZMA2 ('Ultra', 8 threads): over 4Gb!

Algorithms

- The basic tools of programming: sorting, searching. . .
- How do we compare them? Which is 'best'?
  - CPU usage
  - Memory usage
  - Quality (Zopfli takes 1s to shave 35k off this PDF)
  - Note: Often a trade-off between those
  - LZMA2 ('Ultra', 8 threads): over 4Gb!

Algorithms

- The basic tools of programming: sorting, searching...
- How do we compare them? Which is 'best'?
  - CPU usage
  - Memory usage
  - Quality (Zopfli takes 1s to shave 35k off this PDF)
  - Note: Often a trade-off between those
  - LZMA2 ('Ultra', 8 threads): over 4Gb!

Algorithms

- The basic tools of programming: sorting, searching. . .
- How do we compare them? Which is 'best'?
  - CPU usage
  - Memory usage
  - Quality (Zopfli takes 1s to shave 35k off this PDF)
  - Note: Often a trade-off between those
  - LZMA2 ('Ultra', 8 threads): over 4Gb!

Algorithms

- The basic tools of programming: sorting, searching. . .
- How do we compare them? Which is 'best'?
    - CPU usage
    - Memory usage
    - Quality (Zopfli takes 1s to shave 35k off this PDF)
    - Note: Often a trade-off between those
    - LZMA2 ('Ultra', 8 threads): over 4Gb!

Big-O Notation

- Quick short-hand way of describing *roughly* how an algorithm scales
- Form of "Bachmann-Landau notation", but name "Big O" stuck
- Memory/workspace
- Time (usually CPU)
- If the problem size doubles, resource needs . . . ?
- For example: "$O(n)$ time, $O(1)$ space"

Big-O Notation

- Quick short-hand way of describing *roughly* how an algorithm scales
- Form of "Bachmann-Landau notation", but name "Big O" stuck
- Memory/workspace
- Time (usually CPU)
- If the problem size doubles, resource needs . . . ?
- For example: "$O(n)$ time, $O(1)$ space"

Big-O Notation

- Quick short-hand way of describing *roughly* how an algorithm scales
- Form of "Bachmann-Landau notation", but name "Big O" stuck
- Memory/workspace
- Time (usually CPU)
- If the problem size doubles, resource needs . . . ?
- For example: "$O(n)$ time, $O(1)$ space"

Big-O Notation

- Quick short-hand way of describing *roughly* how an algorithm scales
- Form of "Bachmann-Landau notation", but name "Big O" stuck
- Memory/workspace
- Time (usually CPU)
- If the problem size doubles, resource needs ...?
- For example: "$O(n)$ time, $O(1)$ space"

Big-O Notation

- Quick short-hand way of describing *roughly* how an algorithm scales
- Form of "Bachmann-Landau notation", but name "Big O" stuck
- Memory/workspace
- Time (usually CPU)
- If the problem size doubles, resource needs ...?
- For example: "$O(n)$ time, $O(1)$ space"

Big-O Notation

- Quick short-hand way of describing *roughly* how an algorithm scales
- Form of "Bachmann-Landau notation", but name "Big O" stuck
- Memory/workspace
- Time (usually CPU)
- If the problem size doubles, resource needs ...?
- For example: "$O(n)$ time, $O(1)$ space"

Example: Calculate Mean

From back in week 2:

1. Iterate through each item in list
2. Add to running total and counter
3. Divide those values and return

Example: Calculate Mean

From back in week 2:

1. Iterate through each item in list
2. Add to running total and counter
3. Divide those values and return

Example: Calculate Mean

From back in week 2:

1. Iterate through each item in list
2. Add to running total and counter
3. Divide those values and return

Analysing Mean Algorithm

How does that algorithm perform if we have twice, or ten times, as many items?

Time Visits each item exactly once, so clearly *linear time* ($O(n)$)

Space Still just one counter and one running total: *constant space* ($O(1)$)

Analysing Mean Algorithm

How does that algorithm perform if we have twice, or ten times, as many items?

Time Visits each item exactly once, so clearly *linear time* ($O(n)$)

Space Still just one counter and one running total: *constant space* ($O(1)$)

Typical Complexities

- **Constant** $O(1)$ Fixed: resources unaffected by input size.
- **Log** $O(\ln n)$ Slightly more for each doubling of input size.
- **Linear** $O(n)$ Proportional: $10 \times data \rightarrow 10 \times resources$
- **Quasilinear** $O(n \ln n)$ $10 \times data \rightarrow c12 \times resources$
- **Quadratic** $O(n^2)$ $10 \times data \rightarrow 100 \times resources$
- **Cubic** $O(n^3)$ $10 \times data \rightarrow 1,000 \times resources$
- **Exponential** $O(e^n)$

Typical Complexities

- **Constant** $O(1)$ Fixed: resources unaffected by input size.
- **Log** $O(\ln n)$ Slightly more for each doubling of input size.
- **Linear** $O(n)$ Proportional: $10 \times \textit{data} \rightarrow 10 \times \textit{resources}$
- **Quasilinear** $O(n \ln n)$ $10 \times \textit{data} \rightarrow c12 \times \textit{resources}$
- **Quadratic** $O(n^2)$ $10 \times \textit{data} \rightarrow 100 \times \textit{resources}$
- **Cubic** $O(n^3)$ $10 \times \textit{data} \rightarrow 1,000 \times \textit{resources}$
- **Exponential** $O(e^n)$

Typical Complexities

Constant $O(1)$ Fixed: resources unaffected by input size.

Log $O(\ln n)$ Slightly more for each doubling of input size.

Linear $O(n)$ Proportional: $10 \times \mathit{data} \to 10 \times \mathit{resources}$

Quasilinear $O(n \ln n)$ $10 \times \mathit{data} \to c12 \times \mathit{resources}$

Quadratic $O(n^2)$ $10 \times \mathit{data} \to 100 \times \mathit{resources}$

Cubic $O(n^3)$ $10 \times \mathit{data} \to 1,000 \times \mathit{resources}$

Exponential $O(e^n)$

Typical Complexities

Constant	$O(1)$ Fixed: resources unaffected by input size.
Log	$O(\ln n)$ Slightly more for each doubling of input size.
Linear	$O(n)$ Proportional: $10 \times data \to 10 \times resources$
Quasilinear	$O(n \ln n)$ $10 \times data \to c12 \times resources$
Quadratic	$O(n^2)$ $10 \times data \to 100 \times resources$
Cubic	$O(n^3)$ $10 \times data \to 1,000 \times resources$
Exponential	$O(e^n)$

Typical Complexities

Constant	$O(1)$ Fixed: resources unaffected by input size.
Log	$O(\ln n)$ Slightly more for each doubling of input size.
Linear	$O(n)$ Proportional: $10 \times data \to 10 \times resources$
Quasilinear	$O(n \ln n)$ $10 \times data \to c12 \times resources$
Quadratic	$O(n^2)$ $10 \times data \to 100 \times resources$
Cubic	$O(n^3)$ $10 \times data \to 1,000 \times resources$
Exponential	$O(e^n)$

Typical Complexities

Constant	$O(1)$ Fixed: resources unaffected by input size.
Log	$O(\ln n)$ Slightly more for each doubling of input size.
Linear	$O(n)$ Proportional: $10 \times data \rightarrow 10 \times resources$
Quasilinear	$O(n \ln n)$ $10 \times data \rightarrow c12 \times resources$
Quadratic	$O(n^2)$ $10 \times data \rightarrow 100 \times resources$
Cubic	$O(n^3)$ $10 \times data \rightarrow 1,000 \times resources$
Exponential	$O(e^n)$

Typical Complexities

Constant	$O(1)$ Fixed: resources unaffected by input size.
Log	$O(\ln n)$ Slightly more for each doubling of input size.
Linear	$O(n)$ Proportional: $10 \times data \rightarrow 10 \times resources$
Quasilinear	$O(n \ln n)$ $10 \times data \rightarrow c12 \times resources$
Quadratic	$O(n^2)$ $10 \times data \rightarrow 100 \times resources$
Cubic	$O(n^3)$ $10 \times data \rightarrow 1,000 \times resources$
Exponential	$O(e^n)$

Example Complexities

Constant Space Finding mean of inputs

Log Time Searching an index

Linear Time Finding mean of inputs

Quasilinear Time Quicksort on most inputs

Quadratic Time Selection sort (find lowest, remove, repeat)

Example Complexities

Constant Space Finding mean of inputs

Log Time Searching an index

Linear Time Finding mean of inputs

Quasilinear Time Quicksort on most inputs

Quadratic Time Selection sort (find lowest, remove, repeat)

Example Complexities

Constant Space Finding mean of inputs

Log Time Searching an index

Linear Time Finding mean of inputs

Quasilinear Time Quicksort on most inputs

Quadratic Time Selection sort (find lowest, remove, repeat)

Example Complexities

Constant Space Finding mean of inputs

Log Time Searching an index

Linear Time Finding mean of inputs

Quasilinear Time Quicksort on most inputs

Quadratic Time Selection sort (find lowest, remove, repeat)

Example Complexities

Constant Space: Finding mean of inputs

Log Time: Searching an index

Linear Time: Finding mean of inputs

Quasilinear Time: Quicksort on most inputs

Quadratic Time: Selection sort (find lowest, remove, repeat)

Sorting

- Sorting is hard work, often needed, well studied
- Clearly no *sub-linear time* solution possible: must visit each item at least once
- Possibly *constant space* though: could sort input in-place

Sorting

- Sorting is hard work, often needed, well studied
- Clearly no *sub-linear time* solution possible: must visit each item at least once
- Possibly *constant space* though: could sort input in-place

Sorting

- Sorting is hard work, often needed, well studied
- Clearly no *sub-linear time* solution possible: must visit each item at least once
- Possibly *constant space* though: could sort input in-place

Big-O Limitations

- Generic/ideal abstract machine, no CPU specifics
- General case, ignores tiny/huge/corner cases
- Ignores parallelism: on 8 core CPU, multithreaded may be 8× faster, same big-O
- Ignores constants: multithreaded LZMA2 compression can be over 4 Gb, still 'constant'
- Tiny data sets cached, huge ones mean hitting disk

Big-O Limitations

- Generic/ideal abstract machine, no CPU specifics
- General case, ignores tiny/huge/corner cases
- Ignores parallelism: on 8 core CPU, multithreaded may be $8\times$ faster, same big-O
- Ignores constants: multithreaded LZMA2 compression can be over 4 Gb, still 'constant'
- Tiny data sets cached, huge ones mean hitting disk

Big-O Limitations

- Generic/ideal abstract machine, no CPU specifics
- General case, ignores tiny/huge/corner cases
- Ignores parallelism: on 8 core CPU, multithreaded may be $8\times$ faster, same big-O
- Ignores constants: multithreaded LZMA2 compression can be over 4 Gb, still ‘constant’
- Tiny data sets cached, huge ones mean hitting disk

Big-O Limitations

- Generic/ideal abstract machine, no CPU specifics
- General case, ignores tiny/huge/corner cases
- Ignores parallelism: on 8 core CPU, multithreaded may be $8\times$ faster, same big-O
- Ignores constants: multithreaded LZMA2 compression can be over 4 Gb, still 'constant'
- Tiny data sets cached, huge ones mean hitting disk

Big-O Limitations

- Generic/ideal abstract machine, no CPU specifics
- General case, ignores tiny/huge/corner cases
- Ignores parallelism: on 8 core CPU, multithreaded may be $8\times$ faster, same big-O
- Ignores constants: multithreaded LZMA2 compression can be over 4 Gb, still 'constant'
- Tiny data sets cached, huge ones mean hitting disk

Algorithmic Complexity Summary

- Each algorithm has two key complexities: time, space
- Two algorithms could have similar complexity but one is faster
- Generally refers to typical case, ignores corner cases

Algorithmic Complexity Summary

- Each algorithm has two key complexities: time, space
- Two algorithms could have similar complexity but one is faster
- Generally refers to typical case, ignores corner cases

Algorithmic Complexity Summary

- Each algorithm has two key complexities: time, space
- Two algorithms could have similar complexity but one is faster
- Generally refers to typical case, ignores corner cases

Data Structures

Lists	Arrays, linked lists
Trees	Binary trees, red-black trees
Tables	Hash tables, lookup tables

Data Structures

Lists	Arrays, linked lists
Trees	Binary trees, red-black trees
Tables	Hash tables, lookup tables

Data Structures

Lists	Arrays, linked lists
Trees	Binary trees, red-black trees
Tables	Hash tables, lookup tables

Data Structure: Array

Insert	Linear time (rewrite)
Delete	Linear time (rewrite)
Search	Linear time

Data Structure: Array

Insert	Linear time (rewrite)
Delete	Linear time (rewrite)
Search	Linear time

Data Structure: Array

Insert	Linear time (rewrite)
Delete	Linear time (rewrite)
Search	Linear time

Data Structure: Linked List

Insert	Constant time *once found*
Delete	Constant time *once found*
Search	Linear time

Data Structure: Linked List

Insert	Constant time *once found*
Delete	Constant time *once found*
Search	Linear time

Data Structure: Linked List

Insert	Constant time *once found*
Delete	Constant time *once found*
Search	Linear time

Data Structure: Binary Tree

When (roughly) balanced

Insert $O(\ln n)$

Delete $O(\ln n)$

Search $O(\ln n)$

All variants of searching: find the point where the item is or should be.

Caveat: in-order insertions can turn the tree into a linked list, $O(n)$ for all!

Data Structure: Binary Tree

When (roughly) balanced

Insert $O(\ln n)$

Delete $O(\ln n)$

Search $O(\ln n)$

All variants of searching: find the point where the item is or should be.

Caveat: in-order insertions can turn the tree into a linked list, $O(n)$ for all!

Data Structure: Binary Tree

When (roughly) balanced

Insert	$O(\ln n)$
Delete	$O(\ln n)$
Search	$O(\ln n)$
	All variants of searching: find the point where the item is or should be.
	Caveat: in-order insertions can turn the tree into a linked list, $O(n)$ for all!

Data Structure: Binary Tree

When (roughly) balanced

Insert $O(\ln n)$

Delete $O(\ln n)$

Search $O(\ln n)$

All variants of searching: find the point where the item is or should be.

Caveat: in-order insertions can turn the tree into a linked list, $O(n)$ for all!

Data Structure: Binary Tree

When (roughly) balanced

Insert $O(\ln n)$

Delete $O(\ln n)$

Search $O(\ln n)$

All variants of searching: find the point where the item is or should be.

Caveat: in-order insertions can turn the tree into a linked list, $O(n)$ for all!

Data Structure: Red-Black Tree

- Binary tree
- Mark (colour) each node either red or black
- Red nodes have only black child nodes
- Every path contains the same number of black nodes
- Shortest has all black, longest alternates black-red, i.e. twice as long

Data Structure: Red-Black Tree

- Binary tree
- Mark (colour) each node either red or black
- Red nodes have only black child nodes
- Every path contains the same number of black nodes
- Shortest has all black, longest alternates black-red, i.e. twice as long

Data Structure: Red-Black Tree

- Binary tree
- Mark (colour) each node either red or black
- Red nodes have only black child nodes
- Every path contains the same number of black nodes
- Shortest has all black, longest alternates black-red, i.e. twice as long

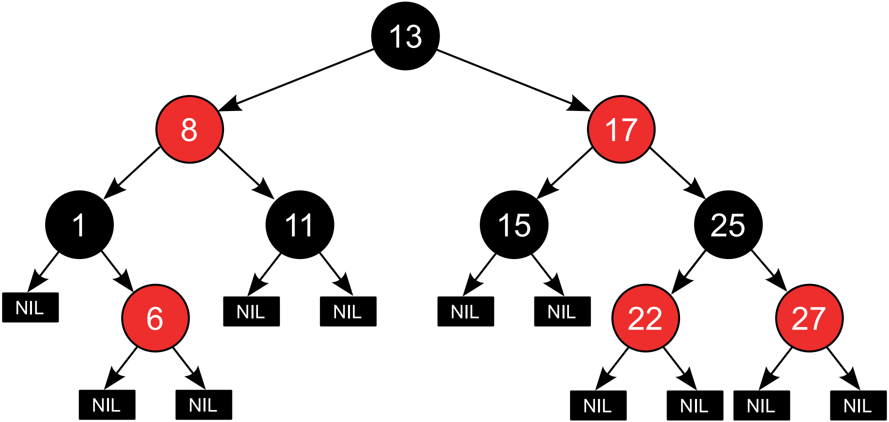
Data Structure: Red-Black Tree

- Binary tree
- Mark (colour) each node either red or black
- Red nodes have only black child nodes
- Every path contains the same number of black nodes
- Shortest has all black, longest alternates black-red, i.e. twice as long

Data Structure: Red-Black Tree

- Binary tree
- Mark (colour) each node either red or black
- Red nodes have only black child nodes
- Every path contains the same number of black nodes
- Shortest has all black, longest alternates black-red, i.e. twice as long

Red Black Tree Example[1]



[1] Source: `https://commons.wikimedia.org/wiki/File:Red-black_tree_example.svg`

Data Structure: Lookup Tables

1	AMG
2	SET
3	DBS
4	SHS
5	GS

Lookup Table Operations

Insert Linear time (rewrite)

Delete Linear time (rewrite)

Search Constant time: jump to offset (linear time for reverse)

Lookup Table Operations

Insert Linear time (rewrite)

Delete Linear time (rewrite)

Search Constant time: jump to offset (linear time for reverse)

Lookup Table Operations

Insert Linear time (rewrite)

Delete Linear time (rewrite)

Search Constant time: jump to offset (linear time for reverse)

Data Structure: Hash Tables

Split the data into 'buckets' by 'hash value'. Calculate the hash, search that bucket.

Bernstein CDB hash function:

$h = 5381$

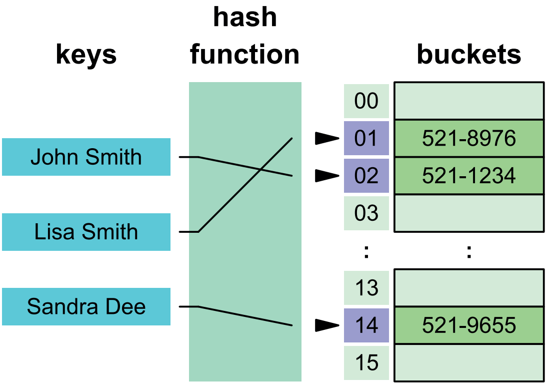
for each byte:

$h = h \times 33 \oplus \mathit{byte}$

End result is an array, but containing only some fraction of the total data.

With a predetermined list, like language keywords, you can precompute a *perfect hash* in which each valid word has a unique value.

Hash Table Example[2]



[2] Source: https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg

Searching and Soundex

Encode (English) words into alphanumeric strings so that words which sound the same have the same code.

- 1. Leave the first letter alone
- 2. Drop vowels, Y, H, W
- 3. Drop repetitions unless they were separated by a vowel-sound
- 4. Take 3 digits, truncate/zero-extend as needed

1	BFPW
2	CGJKQSXZ
3	DT
4	L
5	MN
6	R

e.g. their, they're, there = T600

Lab Task Week 11

1. Find and try out the built in Java data structures, classifying their time complexity
2. Write your own Soundex implementation in Java

Lab Task Week 11

1. Find and try out the built in Java data structures, classifying their time complexity
2. Write your own Soundex implementation in Java