Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 2

Dr Nick Hayward

Algorithms and problem solving

- for a *search problem*
  - *some initial, general questions we might consider as we review algorithms to solve a problem*
- e.g.
  - *what is it meant to do?*
  - *does it actually do what it is meant to do?*
  - *how efficient is the algorithm?*
- Or, more formally, we define the following
  - *Specification*
  - *Verification*
  - *Performance analysis*

Algorithms and problem solving

specification

- specification should formalise the essential or pertinent details
  - *i.e. relative to the problem that the algorithm is meant to solve*
- it might be based on a particular representation of the associated data
  - *sometimes it will be presented in a more abstract manner*
- customarily need to define relationship between inputs and outputs of the algorithm
  - *n..b. there is no general requirement that the specification is complete or non-ambiguous*
- for simple problems
  - *often obvious or easy to see that a particular algorithm will always work*
  - *i.e. it will satisfy its specification*

Algorithms and problem solving

verification

- the fact that an algorithm satisfies its specification may not be as obvious
  - *e.g. for more complicated specifications and algorithms*
- need to consider formal verification
  - *to determine whether the algorithm is indeed correct*
- testing on a few particular inputs may be enough to show that the algorithm is incorrect
- as the number of potential inputs, and variety, for most algorithms is infinite
  - *infinite, in theory, and a tad large in practice…*
  - *need to test more than just sample cases to ensure the algorithm satifies the specification*
- need what is commonly known as *correctness proofs*
- we'll briefly discuss *proofs*
  - *and useful relevant ideas such as invariants*
- formal verification techniques are complex
  - *may be considered as an extra topic towards the end of the course*

Algorithms and problem solving

performance

- efficiency or *performance* of a given algorithm may relate to the defined *resources* it requires
- e.g. might be relative to how quickly the algorithm runs
  - *or the system resources, such as memory, it requires*
- commonly depends on defined *instance size* of the problem
  - *the chosen representation of data*
  - *the various details of the algorithm itself*
- commonly acted as a useful driving force for development of new data structures and algorithms
- efficiency will be considered in more detail later in the course

Fun Exercise

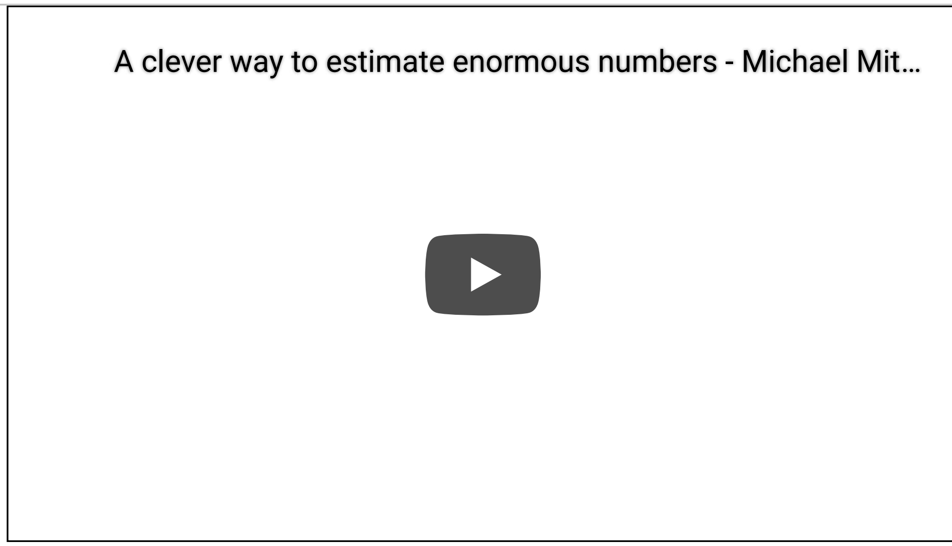
example of simple search

- consider examples where *simple search* might be necessary or useful
  - *why?*
  - *where?*
  - *how?*
  - *expected output?*
- then, consider the conditions or information necessary to avoid using simple search…

Approx. 10 minutes and then discuss…

Video - Mathematics

fun estimations



A clever way to estimate enormous numbers - Ted-Ed

Source - Ted-Ed - YouTube

Running time for algorithms

- first option for timing algorithms is *simple search*
- in effect
  - *100 items has a potential maximum number of guesses of 100*
- if we increase this number exponentially
  - *potential maximum time will continue to grow at the same rate*
  - *e.g. 4 billion items may take 4 billion guesses to reach the end of the list*
  - *known as linear time*
- if we compare this performance with *binary search*
  - *we quickly see the performance benefits*
- e.g. for a list of 100 items
  - *we require at most 7 guesses*
- larger datasets see a marked improvement in performance
  - *e.g. 4 billion results will now require a maximum of 32 guesses*
- so, we have a comparative result
  - `O(n)` *for linear time*
  - `O (Log n)` *for logarithmic time*

Logarithms

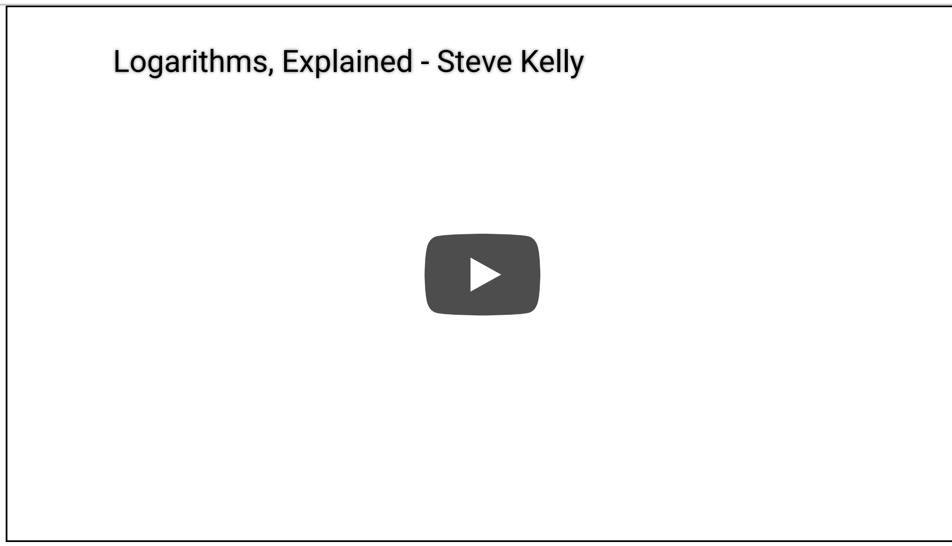
- a brief, but useful segue, on *logarithms*
- commonly consider logs as a flipped implementation of *exponentials*
- e.g.
  - *$log_{10}100$*
- represents an exponential of `2`. or `10 x 10`
- in effect,
  - *"how many 10s do we multiply together to get 100?"*
- e.g. we may consider the following examples

exponential	logarithm
$10^2 = 100$	$\log_{10}100 = 2$
$10^3 = 1000$	$\log_{10}1000 = 3$
$2^3 = 8$	$\log_2 8 = 3$
$2^4 = 16$	$\log_2 16 = 4$
$2^5 = 32$	$\log_2 32 = 5$

- running time in Big O notation is commonly referenced as $\log_2$
  - *e.g. `log 8 = 3` because 2^3 gives us 8.*
- for a list of `1024` elements
  - *test running time as `log 1024`*
  - *the same as 2^{10}*
- a search of `1024` elements will require a maximum of 10 queries

Video - Mathematics

Logarithms



Logarithms Explained - Ted-Ed

Source - Logarithms Explained - YouTube

Basic algorithms - binary search

- let's consider an initial example and problem
- Binary search algorithm is a common option
  - *e.g. for finding individual items in a larger dataset*
- we might use this algorithm to find a person in a directory
  - *or, perhaps, find a user in a broader network*
- instead of progressing from A to B to C &c. within a defined directory
  - *we may start in the middle and then divide the data in half*
- division is predicated on an sorted list of data for the binary search algorithm
- as binary search progresses through the dataset
  - *returns index position for a matched result or* `null` *for no match.*
- helps to eliminate possible results, and continually focus the dataset to find the search criteria

Conceptual example

search for a number

- start with a simple example for guessing a given number from the ordered sequence `1 to 100`
- e.g. pseudocode

```
* first guess is `54`
  * this guess is too low
  * remove all numbers from `1 to 54`
  * number sequence is updated to `55 to 100`
* second guess is `75`
  * this guess is too high
  * remove all numbers from `100 to 75`
  * number sequence is updated to `55 to 74`
* third guess is `65`
  * this guess is too high
  * remove all numbers from `65 to 74`
  * number sequence is updated to `55 to 64`
* fourth guess is `60`
  * this guess is *correct*
```

- by using binary search
  - *may see a stark contrast with the algorithm for simple search*
  - *e.g. compare with linear progression through the numbers until we hit upon the required number or answer*
- e.g. if we consider the above number search
  - *we can easily see how the algorithm optimises performance*
  - *`100 -> 56 -> 20 -> 10 -> 0 - answer found...`*
- binary search has helped us find the correct number in four turns
  - *instead of iterating through each number sequentially*
- a key part of working with binary search is the need to start with an ordered list of data...

Conceptual example

benefits of scale

- a noted benefit of this type of algorithm
  - *the potential to scale for larger datasets*
- as the dataset grows exponentially
  - *the search algorithm is able to keep pace for simple queries*

Working example - binary search

- conceptual design and use of a binary search algorithm
  - *may be implemented in many different programming languages*
- e.g. we might consider the following sample for a Python application

sample - Python binary search

- `binary_search` function
  - *takes a sorted array of items, and a single item*
  - *if item is in the defined array - search function will commonly return its position*
- we may keep a record of where to find a given value

Code example - Python binary search

- start by defining how to track high and low values in a given data set
  - *e.g.*

```
low = 0
high = len(list) -1
```

- as the example searches for a value
  - *keep a record of where to search in the passed array for a given value*
- we may also query the middle of the array
  - *e.g.*

```
mid = (low + high) / 2
guess = list(mid)
```

- then modify these values as we use binary search with the passed dataset
- e.g. if we guess a value for an item
  - *it may be higher, lower, or a known value*
- for a lower value
  - *simply check the current stored value of* `low`
  - *if the guess is too low, update the current low value accordingly*

```
if (guess < item) {
    low = mid + 1
}
```

Big O notation

- Big O is special notation we may use
  - *to test and define the comparative performance of an algorithm*
- e.g. commonly use this notation
  - *to test the performance of a third party algorithm*
- then compare and contrast various algorithms
  - *compare relative to project requirements*

A practical example - part 1

- an example of choosing between simple and binary search
  - *n.b. this may seem like an obvious choice, but there may be contexts where linear time may be acceptable*
- in many examples, we need an algorithm that is both fast and correct
  - *e.g. Landing on Mars…*
- we need to quickly choose an algorithm
  - *usually in 10 seconds or less*
  - *to allow a spaceship to land on Mars*
- for this test
  - *binary search will be quicker for most tests*
  - *simple search is easier to write - may reduce errors due to its inherent simplicity…*
- as we’re performing mission critical tasks, we can’t have any bugs

A practical example - part 2

- begin by running each algorithm 100 times
  - *each task may take 1 millisecond to execute*
- if we run initial tests, we get the following results
  - *simple search = 100 ms (100 x 1ms)*
  - *binary search = 7 ms ($\log_2 100 = 7$)*
- 100 ms vs 7 ms.
- real-world usage difference is minimal
  - *actual program will likely require a billion plus tasks and executions*
- perform a quick initial scaling of timings, e.g.
  - *binary search = ~30 ms ($\log_2 1,000,000,000$)*
- back of the enveleope, panicked calculation...
  - *binary search was initially ~15 times faster, so simple search will scale to* `30 x 15`
- seems reasonable, and is within tolerances for the program

A practical example - part 3

- there's a major issue with this cursory calculation
    - *it's based on an assumption that both search algorithms grow at the same rate*
- run times grow at different rates
    - *thereby impacting performance relative to each dataset*
- if we consider this specific example
    - *Big O notation shows us that binary search is closer to 33 million times faster than simple search*
- so, we *cannot* use simple search for our Mars lander…

Big O usage - part 1

initial consideration

- Big O notation tells us
  - *the relative operations for each algorithm and dataset*
- in effect
  - *how fast the algorithm is per task*
- Big O notation
  - *defines binary search, for example, as* `O Log(n)`

Video - Big O usage

What is Big O?

What Is Big O? (Comparing Algorithms)

What is Big O? Comparing algorithms.

Source - What is Big O? - YouTube

Big O usage - part 2

visualising

- we may start with various algorithms to draw a grid of 16 squares
- effectively trying to determine the best algorithm to use
- decision is often predicated on many disparate conditions
  - *e.g. may reflect priorities in a given project*
- e.g. we may consider the following algorithm
  - *draw each box until we have the required 16 boxes...*
    - big O notation tells us that a linear pattern will produce 16 grid squares
    - one box is one operation...
- Big O notation produces the following results

```
O(n)
```

Big O usage - part 3

- there are better and more efficient options for algorithms
- e.g. a second algorithm option uses folding to optimise the creation of squares in the grid.
- if we start by folding a large square
  - *e.g. a piece of paper for example*
  - *immediately create two boxes*
- each fold is an operation in the algorithm
- continue folding the square, the piece of paper
  - *until we create our grid of 16 squares*
  - *only takes four operations to complete*
- this algorithm produces a performance of `O(log n)` time

Big O usage - part 4

common runtimes

- as we use various algorithms for projects
  - *consider common performance times as calculated using Big O notation*
- e.g.

big O notation	description	algorithm
O(n)	also known as linear time	simple search
O(log n)	also known as log time	binary search
O(n * log n)	a faster sorting algorithm	e.g. quicksort
0(n$^{2)}$	a slow sorting algorithm	e.g. selection sort
O(n!)	a very slow algorithm	e.g traveling salesman problem

- if we applied each algorithm to the above creation of a grid of 16 squares
  - *we may choose appropriate algorithm to solve the defined problem*
  - *n.b. this is a tad simplified representation of Big O notation to the number of required operations...*
- a fun resource - Big O Algorithmic Complexity Cheatsheet

Big O usage - part 5

runtime and performance

We may consider the runtime and performance of an algorithm as follows,

- algorithm speed is measured using the growth in the number of required operations
  - *not time in seconds*
- consider the speed of increase in the runtime for an algorithm as the size of the input increase
- runtime for algorithms is defined using Big O notation
- `O(log n)` is faster than `O n`
  - *continues to get faster as the list of search items continues to increase*

Big O usage - part 6

Traveling Salesman problem

- an algorithm with a renowned bad running time
- become a famous problem in Computer Science
  - *many believe it will be very difficult to improve its performance*
- the problem is as follows,
  - *the salesman has to visit various cities, e.g. 5*
  - *salesman wants to visit all cities in the minimum distance possible*
- we might simply review each possible route and order to and from the cities
- e.g. or 5 cities
  - *there are 120 permutations*
  - *this will scale as follows*

cities	permutations
6	720
7	5040
8	40320
15	1307674368000
30	265252859812191058636308480000000

- for n items
  - *it will take* n! *(n factorial) operations to compute the result*
- known as *factorial time* or O(n!) time
- as soon as the number of cities passes 100
  - *we do not have enough time to calculate the number of permutations*
  - *our sun is forecast to collapse sooner...*

Video - Algorithms

Efficiency & the Traveling Salesman Problem

Algorithms.

Source - Algorithms - YouTube

Algorithms and Data Structures

intro

- as we use algorithms in applications and systems
  - *need to store, retrieve, and manipulate data*
- a fundamental and key part of working with algorithms
- each piece of data is stored with an address in the computer's available memory
  - *ready for access by the system and application*
- e.g. we might store some data as follows

.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	X	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

- a defined address for `X`, e.g. `ff0edfbe`
  - *allows the system to reference and recall the stored data*
- whenever we need to store some data
  - *the computer will allocate some space in memory and assign an address*
- to store multiple items in an organised structure
  - *consider a data structure*
- create an app to store notes, to-do items, and other data records
  - *might store these items in a list in memory*

- many different data structures we might consider
  - *e.g. array or linked list*

Arrays - part 1

- we'll consider an *array* data structure for this list of items
- from a conceptual perspective
  - *an array will store each list item contiguously in data*
  - *i.e. they are stored next to each other*
  - *one indexed value after another*
- arrays are implemented in different configurations
  - *with varied limitations*
  - *relative to the chosen programming language*
- e.g. we might consider the following scenario for a basic array

```
* store the initial list items in contiguous blocks of memory - e.g. 5 items stored
* add a 6th item to array
* 6th block of memory is already allocated to data
  * move 5 blocks of data for array to empty memory and add 6th block
* add 7th item to array
* add 8th item to array
* 8th block of memory is already allocated to data
  * move 7 blocks of data for array to empty memory and add 8th block
* ...
```

Algorithms and Data Structures

Arrays - part 2

- with this simple pattern
  - *now able to manage a basic array*
- predicated on available memory blocks
  - *and efficiency of algorithms*
  - *ensure it works smoothly for the application and system*
- i.e. it becomes reliant on the following
  - *array data structure algorithm*
    - add data
    - move data
    - manage data - including index, size, &c.
  - *memory management algorithm for underlying system*
    - read data
    - move data
    - resize data
    - ...

Algorithms and Data Structures

Arrays - part 3

- may not be the best option for each programming language and system.
- we might consider an initial reserved size for the array
  - *such as 15 slots in the array for data*
- with this option,
  - *we know we may now add up to 15 items to our data structure*
  - *without worrying about resizing or moving the array in data*
- there are also issues with this solution to array and memory management
  - *wasted memory allocation for unused slots*
    - e.g. add 12 items, and 3 slots are left empty and unused in memory
    - unused memory is still allocated to the data structure, and may not be used elsewhere by default
  - *more than 15 items will still require a move of array in memory*
    - also needs a resize of the underlying data structure...

Iteration and Access - part 1

invariant

- as we work with various iterable data structures
  - *i.e. in the context of algorithms*
- need to define various *invariants* (or *inductive assertions*)
- e.g. an *invariant* is a condition that is not modified or changed
  - *i.e. during the execution of a program or algorithm*
- e.g. usage may be simple

```
i < 13
```

- or more abstract

```
array items are sorted
```

- *invariants* are important and useful for both algorithms and data structures
- enable *correctness proofs* and *verification*

Iteration and Access - part 2

loop invariant

- a *loop-invariant* is a given condition
  - *`true` at the beginning and end of every iteration of a loop*
- e.g. a procedure to find the minimum of n numbers stored in a given array a
  - *e.g. minimum from 5 numbers in passed array...*

```
minimum(int n, array a[n]) {
  // set initial min for array 'a'
  min = a[0];
  // min equals minimum element in a[0],...,a[0]
  for (int i = 1; i != n; i++) {
    // min equals minimum element in a[0],...,a[i-1]
    if (a[i] < min) {
      // update min
      min = a[i];
    }
  }
  // min equals minimum element in a[0],...,a[i-1] & i==n
  return min;
}
```

Iteration and Access - part 3

loop invariant

- at the start of each iteration
  - *and end of previous iteration*
- the *invariant* we defined is `true`
  - *`min equals minimum element in a[0],...,a[i-1]`*
- starts as `true`
  - *repetition maintains this truth*
- as the loop terminates with `i==n`
  - *we know the invariant holds*
  - *`min equals minimum element in a[0],...,a[i-1]`*
- we can be certain that `min` can be returned
  - *as the required minimum value*
- this example is commonly referenced as a *proof by induction*
  - *the invariant is `true` at the beginning of the loop*
  - *the invariant is maintained by each iteration of the loop*
- it *must* be `true` at the end of the loop

Iteration and Access - part 4

loop invariant - example

- we may see this working with the following coded example
  - *check invariant*
  - *i.e.* `min equals minimum element in a[0],...,a[i-1]`

```javascript
function minimum(n, a) {
  // set initial min for array 'a'
  let min = a[0];
    // min equals minimum element in a[0],...,a[0]
  for (i = 1; i != n; i++) {
    // min equals minimum element in a[0],...,a[i-1]
    if (a[i] < min) {
      // update min
      min = a[i];
    }
  }
  // min equals minimum element in a[0],...,a[i-1] & i==n
  return min;
}

// test array 'a'
const a = [4, 8, 22, 13, 19, 7, 2, 49, 10];

// find min in array 'arr' for 'n' numbers
const minNum = minimum(7, a);
console.log(minNum);
```

Resources

- Algorithms - YouTube
- Asymptotic computational complexity
- Big O Algorithmic Complexity Cheatsheet
- Big O notation
- Big O Algorithmic Complexity Cheatsheet
- Logarithms Explained - YouTube
- MDN - JavaScript - Class
- MDN - JavaScript - Symbol
- Ted-Ed - A clever way to estimate enormous numbers - YouTube
- What is Big O? - YouTube