Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 2

Dr Nick Hayward

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.

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
    - ...

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*
  - *i.e.* `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]`

```
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);
```

Algorithms and Data Structures

Linked list - memory

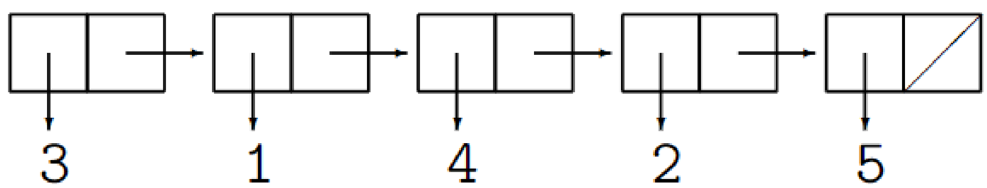
- a linked list
    - *data may be stored anywhere in memory*
    - *each item stores address of the next item in the list*
    - *i.e. link together random memory addresses as a contiguous structure*

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

- request an item from the list
    - *also returns the address of the next available item*
    - *like following a trail of clues to find the answers*
- with a linked list
    - *do not need to move items in memory*
- may also add a new item anywhere in memory
    - *then save address to previous item in the list*
- with a linked list, we do not need to move items
- i.e. if there’s space available in memory, there is space for a linked list

linked list - representation

- also consider a *linked list* data structure to store our list of items in memory
- represent non-empty lists as *two-cells*
- each cell defined as follows
  - *first cell - contains pointer to a list element*
  - *second cell - contains pointer to empty list or another two-cell*
- structure is commonly represented graphically as follows



- pointer to empty list may be shown with a diagonal line
  - *crossing out the cell*
- list is a representation of `[3, 1, 4, 2, 5]`

linked list - inductive construct - part 1

- now consider a *linked list* using two *constructors*
  - *`EmptyList` - constructs the required empty list*
  - *`MakeList(element, list)` - puts element at top of existing, defined list*
- e.g. we may use these constructors as follows

```
MakeList(3, MakeList(1, MakeList(4, MakeList(2, MakeList(5, EmptyList)))))
```

- use to construct our previous list
- use this pattern of constructor execution to construct any list
- *inductive* approach to the creation of data structures is particularly useful
  - *easy to reason and follow.*
- e.g. start with *base case*, `EmptyList`, and
  - *then build more complex lists by repeating induction step, `MakeList(element, list)`*

linked list - inductive construct - part 2

- as with example array
  - *need a pattern to allow us to retrieve the list's elements*
  - *retrieve in a predictable and repeatable manner*
- unlike the array
  - *we do not have an item index*
- we may use known pattern of a list's construction
- i.e. a list is constructed
  - *from first element*
  - *the rest of the list*
- now know that for a non-empty list
  - *always get the first element and the rest...*
- now define two *accessor methods* for our lists
  - `first(list)`
  - `rest(list)`
- *accessors* or *selectors* only useful for non-empty lists
  - *throw an error for an empty list*
- add a condition to check if a given list is empty
  - `isEmpty(list)`
- then call it for each list before passing it to an *accessor* or *selector*

linked list - inductive construct - part 3

- now define a list constructed with `EmptyList` and `MakeList`
  - *including accessors `first` and `rest`*
  - *and the condition `isEmpty`*
- such that the following relationsips may be `true`
  - *`isEmpty(EmptyList)`*
  - *`not isEmpty(MakeList(x,l))` - holds for any `x` and `l` (l = list)*
  - *`first(MakeList(x,l)) = x`*
  - *`rest(MakeList(x,l)) = l`*
- also need to *destructively* change lists
- *Mutators* used to modify either first element or rest of a non-empty list
- e.g.
  - *`replaceFirst(x,l)`*
  - *`replaceRest(r,l)`*
- e.g. for `l = [3, 1, 4, 2, 5]` test the following
  - *`replaceFirst(7, l)` - modifies `l` to `[7, 1, 4, 2, 5]`*
  - *`replaceRest([3, 2, 6, 4], l)` - modifies `l` to `[7, 3, 2, 6, 4]`*
- predicated on expected patterns for first and rest with a list data structure
- concepts for *constructors*, *selectors*, and *conditions*
  - *common place for almost all data structures*
  - *help with abstraction of data types and algorithms*

Video - Algorithms and Data Structures

linked list - part 1

Data Structures: Linked Lists

Source - Linked Lists- YouTube

linked list - implementation

- as we use and design data structures for various algorithms
  - *may find differing implementations of same underlying conceptual outline*
- e.g. *lists*
  - *implementations may depend on primitives offered by a given programming language*
- *list* data structure in Python, Lisp, &c.
  - *considered important primitive data structure*
  - *filling same basic role as arrays in JavaScript...*
- often see same conceptual design either core to a language
  - *or based upon other data structures*
- issues with latter approach for some languages
- e.g. need to ensure that use of an array
  - *i.e. as a construct for a list*
  - *does not limit its size*
- role of defined selectors, mutators &c.
  - *ensure algorithm is implemented correctly with constructed list*

linked list - JS example - part 1

- consider an example implementation in JavaScript for a *linked list*
  - *& constructs required for an algorithm to ensure it functions as expected*
- begin by considering initial criteria for our custom *linked list*
  - *multiple values stored in a linear pattern*
  - *each value stored in a node*
  - *& a link to the next node in the list*
  - `null` *if no next node pointer required*
- initially consider a *singly linked list*
  - *i.e. a node has just one pointer to another node, or* `null`
- for JavaScript development
  - *might consider an array as a suitable data structure*
  - *perhaps for approximating linked list usage*
- whilst array size is dynamic
  - *still requires customisation to provide expected functionality of a linked list*
- example may use an array as the foundation
  - *and construct the linked list...*

linked list - JS example - part 2

- begin with initial design of node structure for a *linked list*
- each node in the list must contain some data and the pointer to the next node
- e.g. following code is a simple JS representation of this pattern

```
class LinkedNode {
  // instantiate with default props for linked list node
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
```

- constructor for this class
  - *defines default properties required for a Linked List node*
  - *data may be defined upon instantiation*
  - `next` *pointer is initially* `null` *- no pointer is yet available for this property*

linked list - JS example - part 3

- use this class as follows to create the first node in a list, which is customarily named `head`

```
// create first node in list
const head = new LinkedNode(13);
// create second node and assign to pointer
head.next = new LinkedNode(9);
```

Algorithms and Data Structures

linked list - JS example - part 4

- initial traversal follows an algorithm
  - *defined using the inherent structure of a linked list*
- algorithm allows an app to traverse all of the list’s data
  - *simply by following `next` pointer defined for each node*

```
let currentNode = head;

while (currentNode !== null) {
  // get data for current node - output, send to DB &c...
  let data = current.data;
  // update pointer for current node
  current = current.next;
}
```

- traversal follows a simple pattern
  - *informed by structure of the linked list itself*
- traversal will continue until we reach end of current list
  - *and `current` is set to `null`*
- algorithm is a common example, regardless of langauge, for initial traversal of a linked list

linked list - JS example - part 5

- also define a class to work with overall Linked List
  - *not just individual nodes*

```
class LinkedList {
  constructor() {
    ...
  }
}
```

- to ensure `head` node is always unique to this Linked List
  - *use a recent JS data type `Symbol`*
- `Symbol` added to JavaScript with ES2015 (ES6)
  - *other benefit is useful description for variable as part of declaration*

```
const head = Symbol('head');

class LinkedList {
  constructor() {
    // set initial pointer to first node in list
    this[head] = null;
  }
}
```

- description is useful for debugging and monitoring variable
  - *and its usage*
- may not be used to access the Symbol itself
- we’ve now set initial pointer for linked list

linked list - JS example - part 6

- after creating initial empty Linked List
  - *need to define a method to allow us to add a new node*
- adding some new data to a linked list
  - *requires traversing structure to find a suitable location*
  - *create the node*
  - *insert in identified location in list*
- if list is empty
  - *simply create new node and assign it to head of list*

```
addNode(data) {
  // create new node
  const newNode = new LinkedNode(33);
  // handle empty list - no items
  if (this[head] === null) {
    // head set to new node
    this[head] = newNode;
  } else {
    // look at first node
    let current = this[head];
    // follow pointers to the end...
    while (current.next !== null) {
      // update current
      current = current.next;
    }
    // update node for next pointer
    current.next = newNode;
  }
}
```

linked list - JS example - part 7

- in previous example code
  - *`addNode()` method defines a single parameter - data for node*
  - *then adds it to end of list*
- we check list - if it is empty
  - *i.e. `head is null`*
  - *assign new node as `head` of list*
- if list has existing nodes
  - *need to traverse it to reach end - the last node*
- uses a simple `while` loop
  - *starting at `head`*
  - *following `next` pointers until we find last node*
- last node will have its `next` pointer set to `null`
  - *stop traversal at this point*
  - *ensure we don't update `current` to `null`*
- allows us to assign new node to `next` pointer
  - *adding data to current list*

Video - Algorithms and Data Structures

linked list - part 2

Data Structures: Linked Lists

Source - Linked Lists- YouTube

issues with linked lists

- a linked list may seem a preferable solution
  - *we may still encounter noticeable issues with such lists*
- e.g. if we need to access item 10 in a linked list
  - *need to know the address location in memory.*
  - *need to get the address from the previous item in the linked list*
- that item needs to get the address from the previous item
  - *and so on to the first item in the linked list...*
- quickly see that a linked list is great
  - *if we need to access each item one at a time*
  - *may read one item, then move to the next item, and so on...*
- if we need to access items out of order
  - *on a regular basis*
  - *a linked list is a poor choice*

benefits of arrays

- accessing an array is a noticeable benefit compared to a linked list
- address known for every item in the array
  - *quickly and easily access an indexed item*
- arrays are a good option if we need to access random items on a regular basis
  - *easily calculate the position of an array item*
  - *contrasts strongly with rigid pattern of access for a linked list*

runtime comparison - part 1

- comparative run times for common operations on arrays and lists

	Arrays	Lists
reading	O(1)	O(n)
insertion	O(n)	O(1)

- key:
  - *O(n) = linear time*
  - *O(1) = constant time*
- *linear time* for *array* insertion and *list* reading
- constant time for *array* reading and *list* insertion

insertion in the middle

- may need to modify our data storage for an app
  - *e.g. to allow insertion in the middle of the data structure*
- our choice of array or linked list will also affect this option
  - *and the efficiency of insertion*
- e.g. if we consider a linked list
  - *it's as easy as modifying address reference of previous element to point to inserted data item*
- for arrays
  - *need to shift all of the remaining elements down to create space for the inserted items*
  - *if there is not sufficient space in the current address location*
    - may also need to move whole array before we can insert new items
- may see a performance benefit for insertion to middle of a linked list compared to an array

deletions

- which option is preferable for deletion?
- for most use cases
  - *simpler to delete an item from a linked list*
  - *only need to modify address reference for previous item in the list*
- for an array
  - *again, need to move the whole array to accommodate the deletion*

runtime comparison - part 2

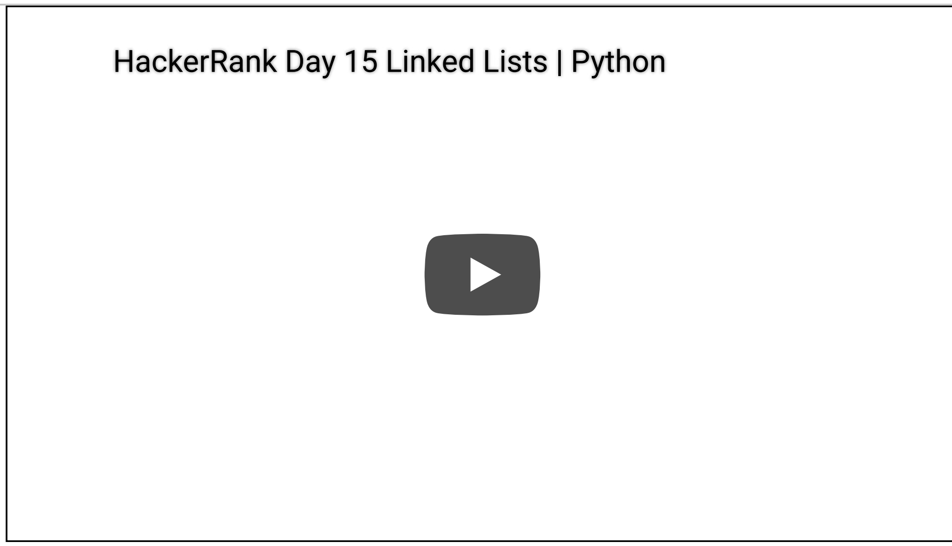
- update our comparison table to now include *delete* operations for both arrays and linked lists

	Arrays	Lists
reading	O(1)	O(n)
insertion	O(n)	O(1)
deletion	O(n)	O(1)

- key:
  - *O(n) = linear time*
  - *O(1) = constant time*
- it's worth noting
  - *both insertions and deletions may be* `O(1)` *run time - only if we may access the element instantly*
- e.g. common practice in such algorithms to maintain a record of the first and last items in a linked list
  - *then only take* `O(1)` *run time to delete such items*

Video - Algorithms and Data Structures

sample linked list question



HackerRank Day 15 - Linked Lists - Python

Source - Linked Lists - YouTube

Resources

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