Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 7

Dr Nick Hayward

Video - Algorithms and Data Structures

Recursion & Divide and Conquer - part 3

Efficiency of Recursion and Divide and Conquer - UP TO 13:59

Source - Divide and Conquer - YouTube

sorting - quicksort - part 1

- a brief segue into a consideration of a sorting algorithm, *quicksort*
- faster search option than *selection sort*
  - *a common option for many real-world uses...*
- e.g. implementations of a `qsort` funtion in the C standard library.
- Quicksort also uses a pattern of *divide and conquer*
- e.g. use quicksort to sort our previous array of data
  - *`[6, 9, 13, 5, 11, 16]`*
- consider our last example of divide and conquer
  - *identified base case as simplest array we could sum*
- same may initially be considered relative to sorting
  - *i.e. clearly identify some arrays that do not need sorting*
- may define *base* case for such sorting as follows
  - *`[]` - empty array*
  - *`[16]` - array with one element*
- empty arrays and arrays with one element
  - *become base case for this sorting*
  - *i.e. return arrays as is without need to sort*

sorting - quicksort - part 2

- then consider an array with three elements
  - *again, use divide and conquer…*
- want to break this array down until we reach base case
  - *i.e. define quicksort as follows*
- *1. choose an element in array*
  - *element is **pivot***
- *2. **partition** the array*
  - *find elements less than pivot*
  - *find elements greater than pivot*
- now have two sub-arrays
  - *sub-array of all elements less than the pivot*
  - *sub-array of all elements greater than the pivot*
- these arrays will not initially be sorted
  - *just partitioned*
- when sub-arrays have been sorted
  - *combine them with pivot to return required sorted array*
- i.e.

```
sub_array[less_than] + pivot + sub_array[greater_than]
```

sorting - quicksort - part 3

- need a way to sort the sub-arrays
  - *where base case is useful again*
- i.e. Quicksort already knows how to sort arrays of two elements
- if we use quicksort with two sub-arrays
  - *then combine results*
  - *we now have a sorted array*
- e.g.

```
quicksort([less_than]) + [pivot] + quicksort([greater_than])
```

- this approach will work with any chosen pivot
- now define quicksort for an array of three elements
  - *choose a pivot*
  - *partition array into two sub-arrays*
    - elements less than pivot
    - elements greater than pivot
  - *recursively call quicksort on the two sub-arrays*

Video - Algorithms and Data Structures

quicksort - part 1

Quicksort - UP TO 3:25

Source - Quicksort - Java - YouTube

sorting - quicksort - part 4

- what happens if we now need to sort an array of four elements...
- use a similar, known pattern
- e.g. for an array of `[37, 12, 17, 9]` follow expected steps
  - *choose a pivot*
    - e.g. `37`
  - *select elements less than pivot*
    - e.g. `[12, 17, 9]`
  - *select elements greater than pivot*
    - e.g. `[]`
- we know how to sort an array of three elements
  - *may call quicksort recursively for this array*
- simply combine results to return sorted array
- we may now sort an array of four elements
- if we can sort an array of four elements
  - *may also sort an array of five elements*
  - *then six elements*
  - *& seven elements*
  - *...*

sorting - quicksort - part 5

- i.e. if we consider an array of five elements
  - *[6, 10, 4, 2, 8]*
- we may partition this array as follows
  - *then call quicksort for sub-arrays*
- e.g.

```
         []   2   [6, 10, 4, 8]
        [2]   4   [6, 10, 8]
     [2, 4]   6   [10, 8]
  [2, 6, 4]   8   [10]
[2, 6, 4, 8]  10   []
```

- clearly see how each sub-array has between zero and four elements
  - *already know how to sort arrays of these sizes using quicksort*
- regardless of chosen pivot
  - *recursively call quicksort on two sub-arrays*
- continue this logic for six elements, &c.

sorting - quicksort - part 6

- example implementation in Python is as follows

```
def quicksort(data):
    if len(data) < 2:
        # base case - 0 or 1 elements already sorted...
        return data
    else:
        # recursive case
        pivot = data[0]
        # sub-array of elements less than pivot
        less_than = [i for i in data[1:] if i <= pivot]
        # sub-array of elements greater than pivot
        greater_than = [i for i in data[1:] if i > pivot]
        # return sorted data
        return quicksort(less_than) + [pivot] + quicksort(greater_than)

print(quicksort([6, 10, 4, 2, 8]))
```

Video - Algorithms and Data Structures

sorting algorithms

Algorithms and Sorting - UP TO 22:03

Source - Algorithms - YouTube

Algorithms and Data Structures

inductive proofs - part 1

- just seen an example of **inductive proofs**
- use such proofs to show an algorithm will work in theory
- each **inductive proof** has two familiar steps
  - *a base case*
  - *an inductive case*
- e.g. we want to prove that a test robot can climb steps
- **inductive case** may define the following
  - *if robot's legs are on a step*
  - *it may put its legs on the next step...*
    - e.g. if it's on the second step, it may now move to the third step, and so on...
- **base case** will define the following
  - *robots legs are on first step*
  - *it can now climb all of the steps*
    - i.e. progressing one step at a time...

inductive proofs - part 2

- we may see a similar logic for our earlier **quicksort** algorithm
- **base case** shown to work as expected for arrays of size `0` and `1`
- **inductive case** - proved that if quicksort worked with an array of size `1`
  - *it would also work with an array of size `2`*
- if it works for an array of size `2`
  - *it will also work for an array of size `3`...*
- by inductive reasoning
  - *the algorithm for **quicksort** will work with an array of any size*
- for real-world usage
  - *obviously making assumptions regarding memory usage, scale, &c.*
  - *but inductive proofs still remain true*

Algorithms and Data Structures

Big O revisited - part 1

- briefly return to a consideration of Big O notation
- comparison of runtimes for various *search* and *sort* algorithms
  - *may help provide some context for quicksort &c...*
  - *e.g.*

binary search	simple search	quicksort	selection sort	traveling salesman
O(log n)	O(n)	O(n log n)	O(n^2)	O(n!)
logarithmic	linear	linearithmic	quadratic	factorial

Video - Algorithms and Data Structures

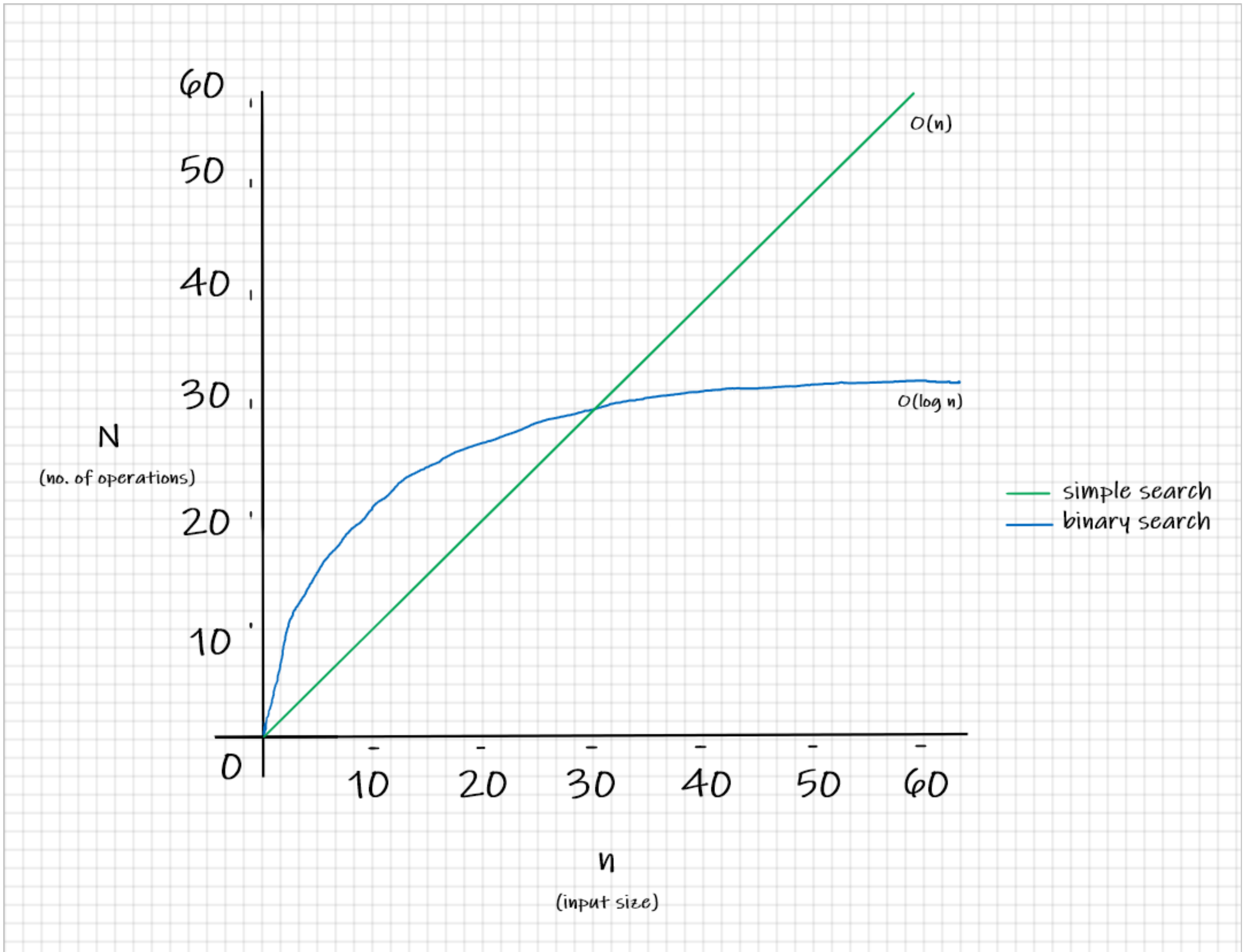
quicksort - part 2

Quicksort - UP TO 4:40

Source - Quicksort - Java - YouTube

Image - Big O revisited

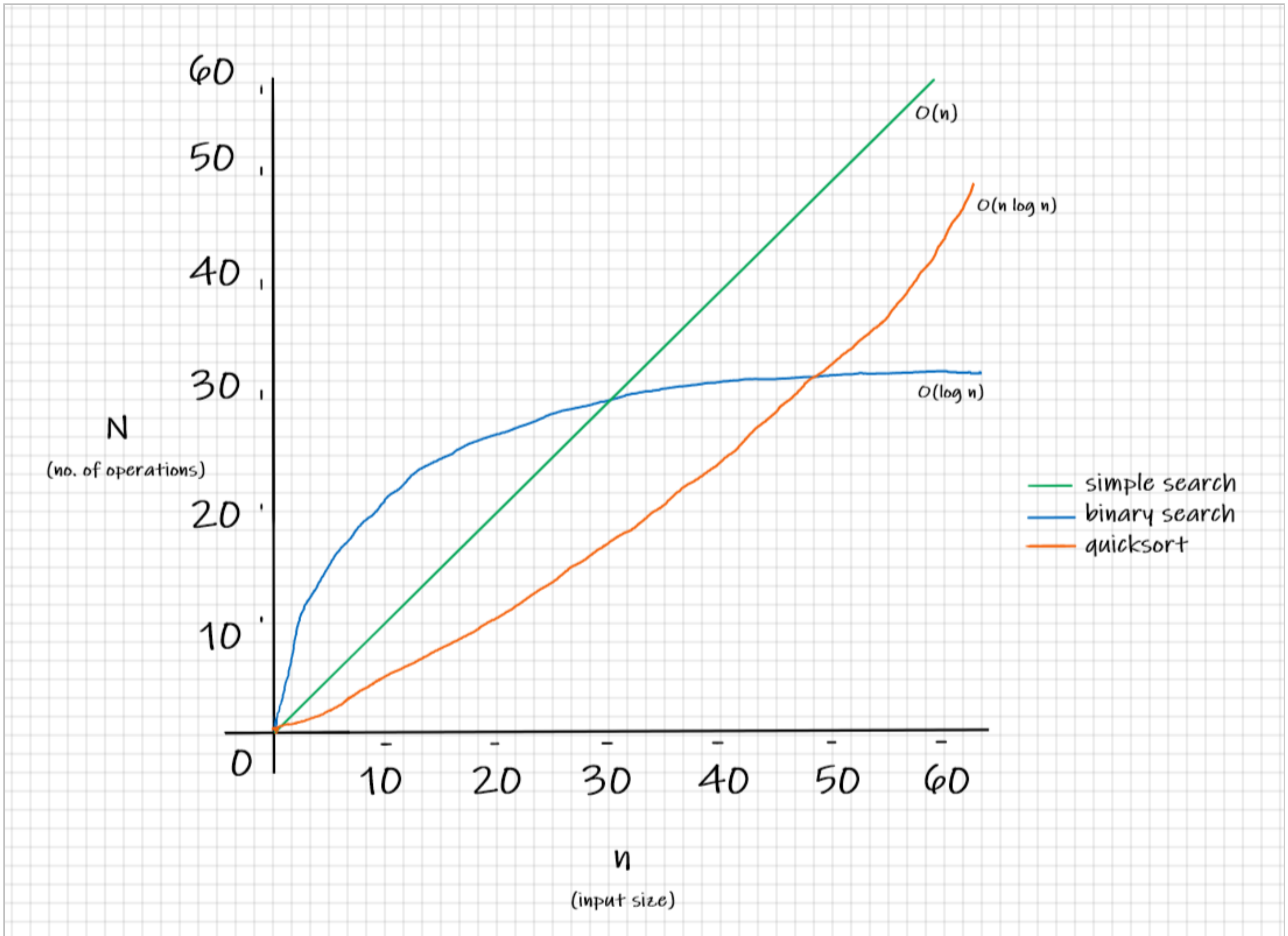
simple search and binary search



Big O Complexity - Simple Search and Binary Search

Image - Big O revisited

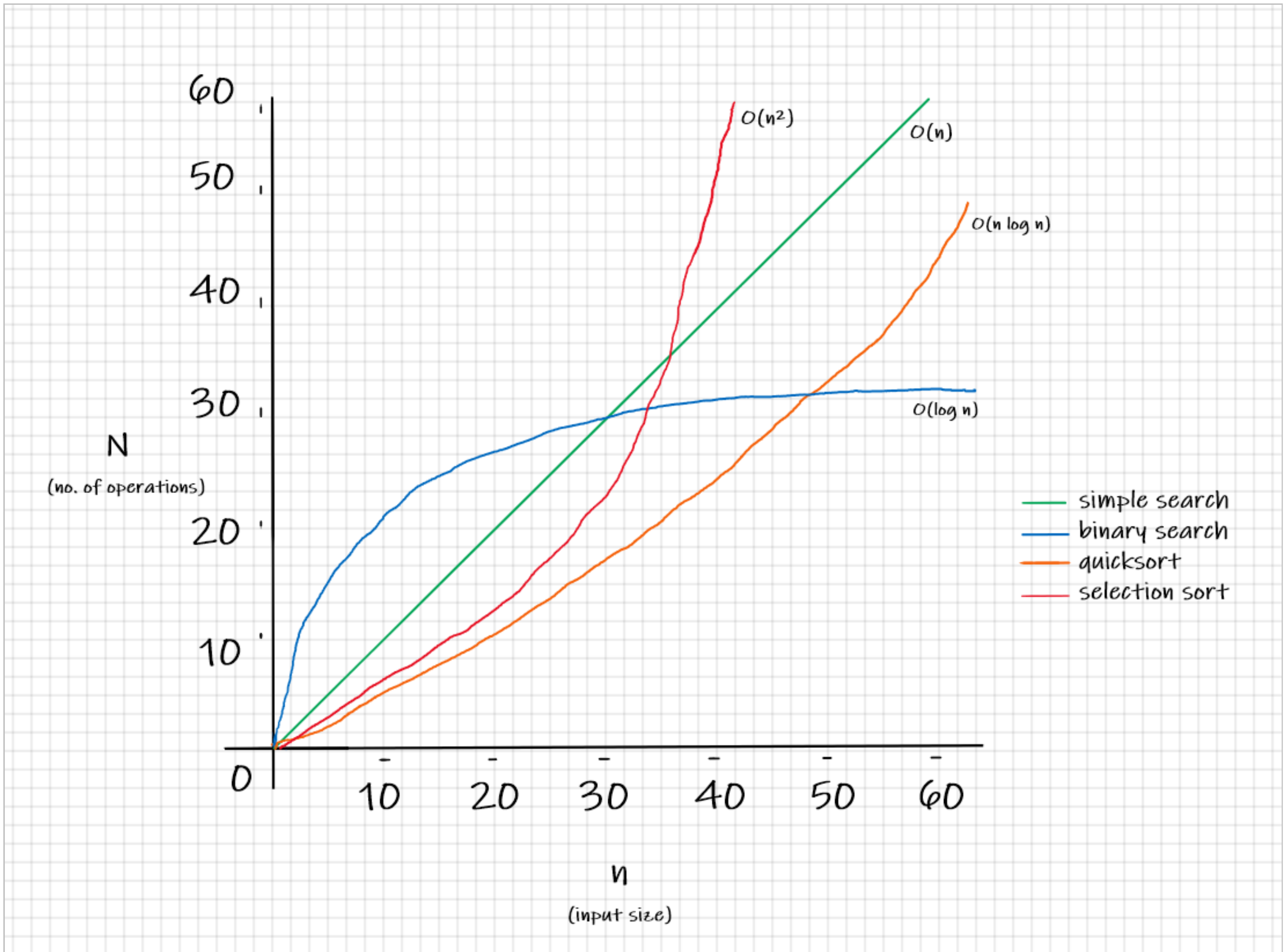
simple search, binary search, and quicksort



Big O Complexity - Simple Search, Binary Search, and Quicksort

Image - Big O revisited

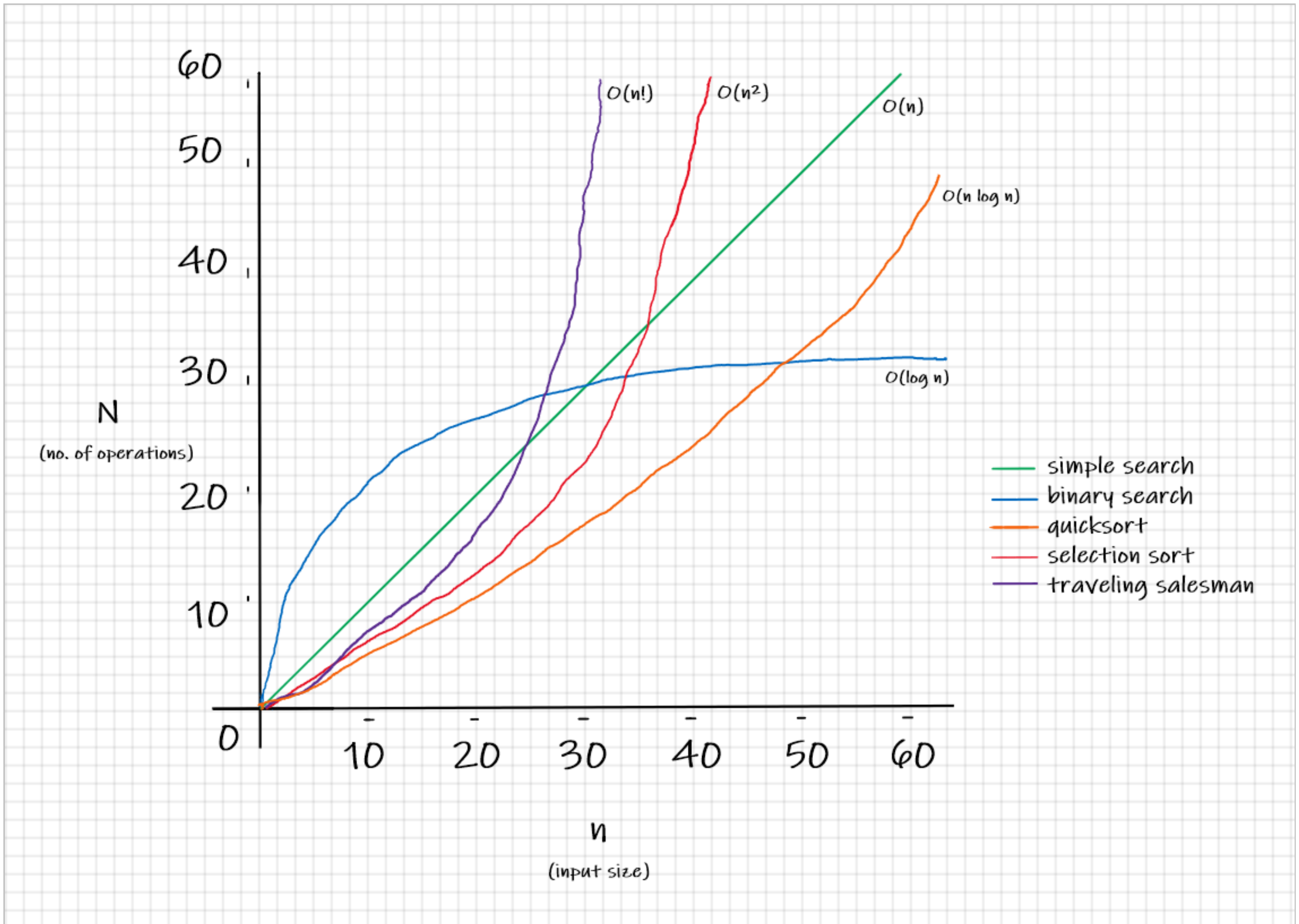
simple search, binary search, quicksort, and selection sort



Big O Complexity - Simple Search, Binary Search, Quicksort, and Selection Sort

Image - Big O revisited

simple search, binary search, quicksort, selection sort, and traveling salesman



Big O Complexity - Simple Search, Binary Search, Quicksort, Selection Sort, and Traveling Salesman

Big O revisited - part 2

- consider the following comparative run times
  - *computer capable of a basic 10 operations per second*
  - *(such a slow computer helps visualise the comparative performance)*

data size	quicksort	selection sort	traveling salesman
10 items	3.3 seconds	10 seconds	4.2 days
100 items	66.4 seconds	16.6 minutes	2.9×10^{149} years
1000 items	996 seconds	27.7 hours	1.27×10^{2559} years

- previous graphs indicative of expected performance
  - *not accurate reflections of performance times*
- show difference in expected performance for each algorithm
  - *relative to scale...*
- e.g quickly see that *selection sort*, `O(n²)`, is a slow algorithm
  - *in particular compared with quicksort...*

Big O revisited - part 3

- compare with another sorting algorithm
  - *e.g merge sort - a time of* `O(n log n)`
  - *much faster than selection sort*
- current algorithm *quicksort* is a tad harder to pin down
- for worst case
  - *time is* $O(n^2)$
  - *potentially as slow as selection sort*
- for average case
  - *define a time of* `O(n log n)`
  - *comparable with faster algorithm merge sort*
- if *merge sort* is considered faster with a time of `O(n log n)`
  - *why not use this algorithm all the time instead of quicksort?*

Algorithms and Data Structures

Big O revisited - part 4

- consider a comparison of *quicksort* and *merge sort*
  - *should helps choose a preferred algorithm to use...*
- start with the following simple usage
  - *a Python function to iterate a list*

```
def print_list(data):
  for val in list:
    print val
```

- as this iteration loops the whole list
  - *runs with a time of* `O(n)`
- what happens if we need to introduce a pause per iteration...
  - *e.g. perhaps to check an external data store, API &c.*
  - *add a test pause of one second per iteration*
- both use cases need to loop through data
  - *each may be defined with a time of* `O(n)`
- even though both functions return same time using Big O notation
  - *first iteration without pause will return faster real-world performance and time...*

Algorithms and Data Structures

Big O revisited - part 5

- consider apparent contradiction for a moment
  - *start to understand actual meaning of Big O notation*
- consider a time of `O(n)` as follows

```
`constant` x `n`
```

- or

```
`c` x `n`
```

- `c` is a fixed amount of time algorithm will take
  - *or the constant*
- comparative times for basic iteration and iteration with a pause
  - *e.g. `10ms * n` vs `1 sec * n`*

Algorithms and Data Structures

Big O revisited - part 6

- usually ignore such constants
  - *if comparative algorithms have different Big O time*
  - *i.e. for most instances - constant doesn't matter*
- e.g. compare *simple search* to *binary search* for previous usage

```
simple search = 10ms * n
binary search = 1 sec * log n
```

- simple search initially seems faster
- if we scale query to **four billion** elements
  - *disparity in performance becomes clear...*

```
simple search = 10ms * 4 billion = 463 days
binary search = 1sec * 32 = 32 seconds
```

- clear improvement in times with binary search
  - *the **constant** did not make a difference*

Big O revisited - part 7

- still exceptions to this rule
- i.e. constant may sometimes make a difference
- *Quicksort* versus *merge sort* is one example where this holds true
- *Quicksort* has a smaller constant than *merge sort*
- if they're both `O(n log n)` time
  - *quicksort is faster...*
- *quicksort* is faster in practice
  - *it hits average case more frequently than worst case*

Big O revisited - part 8

- *average case* and *worst case*
- performance for *quicksort* predicated on chosen *pivot*
- e.g. if we choose a pivot and array is already sorted
  - *quicksort does not check if input array is already sorted*
  - *i.e. it will try to sort the passed array*
- if we compared two possible scenarios for an array
  - *1. first element is always chosen as the pivot*
  - *2. middle element is always chosen as the pivot*
- starting at middle element
  - *will not need to make as many recursive calls for this example*
  - *i.e. hits the base case more quickly, and required call stack will also be shorter...*

Big O revisited - part 9

- first example, choosing first element, is *worst case*
- second example, middle element selection, is *best case*
- for *worst case*
    - *stack size is* `O(n)`
- *best case* has a stack size of `O(log n)`
- e.g. we may see how best case is partitioning the array

```
[1, 2, 3, 4, 5, 6, 7, 8]


[1, 2, 3] 4 [5, 6, 7, 8]
[1] 2 [3]   [5] 6 [7, 8]
                [] 7 [8]
```

Algorithms and Data Structures

Big O revisited - part 10

- for *worst case*
  - *checking each element in array*
  - *e.g. eight in this example*
- first operation takes `O(n)`
  - *we actually check* `O(n)` *elements on every level of call stack*
- even if we partition array in a different manner
  - *e.g. with a different pivot*
- still checking `O(n)` elements every time
- i.e. each level of the stack currently takes `O(n)` time to complete

Big O revisited - part 11

- difference between worst case and best case
  - *seen when we consider height of call stack*
- e.g. best case will check `O(log n)` levels
  - *height of its call stack*
- each level takes `O(n)` time
  - *algorithm will take `O(n) * O(log n)`*
  - *i.e. `O(n log n)` time*
  - *best case for this algorithm*
- see difference when we calculate comparative worst case
  - *a time of `O(n)` for each level*
  - *but also `O(n)` levels*
  - *algorithm will take `O(n) * O(n)`*
  - *i.e. $O(n^2)$ time*
- also define best case as average case
  - *if we always choose a random element in array as defined pivot*
  - *quicksort algorithm will have average time of `O(n log n)`*

Video - Algorithms and Data Structures

quicksort - part 3

Quicksort - UP TO 8:42

Source - Quicksort - Java - YouTube

Algorithms and Data Structures

binary search tree - intro - part 1

- binary search tree (BST) is a binary tree
- each node has a Comparable key (and an associated value)
  - *satisfies a defined restriction*
- e.g. key in any node is
  - *larger than the keys in all nodes in that node's left subtree*
  - *smaller than the keys in all nodes in that node's right subtree*
- comparison is context specific relative to the current node
- binary search - need to work with sorted data sets
- when we add or update the data
  - *need to re-sort list before using binary search*
- if we're working with sorted lists of data
  - *e.g. an array of books*
- quickly encounter a problem
  - *list may need to be updated - item in the array is deleted*
  - *then need to add a value to index where last deleted item stored…*

binary search tree - intro - part 2

- may now update list to meet specific criteria
  - *removing need to repeatedly sort dataset*
- binary search tree data structure
  - *basic tree data structure*

```
          node
           |
          / \
         /   \
       node   node
        |       |
       / \     / \
      /   \   /   \
   node  node node  node
```

Video

Trees and parsing - part 1

How the browser renders a website - UP TO 7:40

Source - So how does the browser actually render a website - YouTube

Algorithms and Data Structures

binary search tree - intro - part 3

- sample binary search tree may be structured as follows

```
              daisy
                |
               / \
              /   \
           bea     emma
            |       |
           / \       \
          /   \       \
       anna  cat    rose
                      |
                     / \
                    /   \
              ophelia   yvaine
                          |
                         /
                        /
                   violet
```

- for every node in this tree
  - *nodes to left of current node are smaller*
  - *nodes to right of current node are larger*
- e.g. search for `violet` - begin at root then use following path
  - *V is after D - traverse right side of tree*
    - current node = emma
  - *V is after E - continue down right side*
    - current node = rose
  - *V is before Y - continue down left side*
    - `violet` node found...
- searching for a node in a binary search tree takes `O(log n)` on average
  - `O(n)` *for worst cases*
- sorted array, by contrast, takes `O(log n)` in worst case scenarios
- might initially consider arrays as preferable option
  - *sorted binary search tree is, on average, faster for insertions and deletions...*

binary search tree - issues

- binary search trees do not provide random access
- performance times are averages
  - *rely on a balanced tree*
- specialist trees may provide self-balancing mechanisms
- e.g. might use a *red-black* tree

Video - Algorithms and Data Structures

binary search trees - part 1

Binary Search Trees - UP TO 1:36

Source - Trees - Java - YouTube

binary search tree - basic logic implementation - part 1

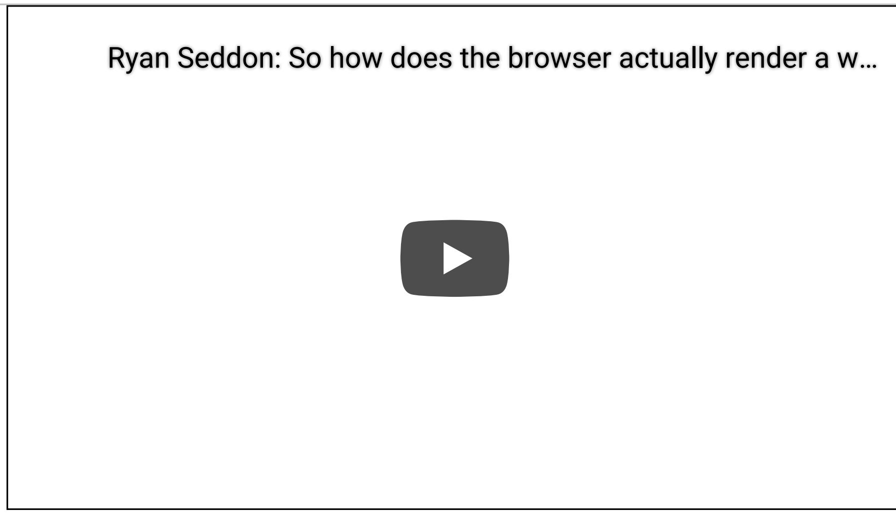
- *symbol-table API* for a binary search tree
  - *common option for implementing this type of traversal and search*
- symbol-table is also known as a map, dictionary, associative array &c.
  - *may vary by programming language*
- general concept is as follows
  - *abstraction of key/value pairs*
    - e.g. insert a value with a specified key
    - given key, search for corresponding value
- sample usage may include the following

application	search	key	value
dictionary	search for a definition	word	definition
index	search for a given reference, e.g book page	term	e.g. list of page numbers for a book
compiler	search for props of variables	variable name	type and value

- for binary search tree logic - may begin with custom function to define nodes
- function includes props required for a node, e.g.
  - *key*
  - *value*
  - *left link*
  - *right link*
  - *node count*

Video

Trees and rendering - part 2



A common working example - How the browser renders a website - UP TO 17:17

Source - So how does the browser actually render a website - YouTube

Algorithms and Data Structures

binary search tree - order-based methods - part 1

- common reason for working with binary search trees (BST)
  - *they keep the keys in order*
- may use BSTs in many disparate API contexts
  - *ensure consistent I/O structure*
- e.g. might consider a custom *symbol-table* API
- some sample methods and usage
  - *`min & max`*
    - if left link of root is `null`
      - smallest key in BST must be root node
    - if left link is not `null`
      - smallest key is in subtree referenced by left link
      - repeats for each left link in each subtree...
  - *`floor`*
    - if a given key is less than key at root of BST
      - floor of key must now be added to left subtree...
    - if key is greater than root
    - floor can be in right link but only if there is a smaller or equal existing key
      - if not, floor of key is root...
  - *`ceiling`*
    - same pattern as floor
      - except check relative to right link
  - *`selection`*
    - e.g. seek key of rank `k`
      - key such that precisely `k` other keys in BST are smaller
    - if number of keys `t` in left subtree is larger than `k`
      - look (recursively) for key of rank `k` in left subtree
    - if `t` is equal to `k`
      - return key at root
    - if `t` is smaller than `k`
      - look (recursively) for key of rank `k - t - 1` in right subtree

Algorithms and Data Structures

binary search tree - order-based methods - part 2

- `range search`
  - *to implement `keys()` method - returns all keys in a given range*
  - *begin with basic recursive BST traversal method - known as `inorder traversal`*
  - *e.g. to show this order traversal*
    - first print all keys in left side of BST - all less than root
    - then print root key
    - then print all keys in right side of BST
  - *`keys()` method*
    - define code to add each key that is in range to a *Queue*
    - skip recursive calls for subtrees that cannot contain keys in range
- `rank`
  - *if given key is equal to key at root*
    - return number of keys `t` in left subtree
  - *if given key is less than key at root*
    - return rank of key in left subtree
  - *if given key is larger than key at root*
    - return `t` plus one (to count key at root) plus rank of key in right subtree

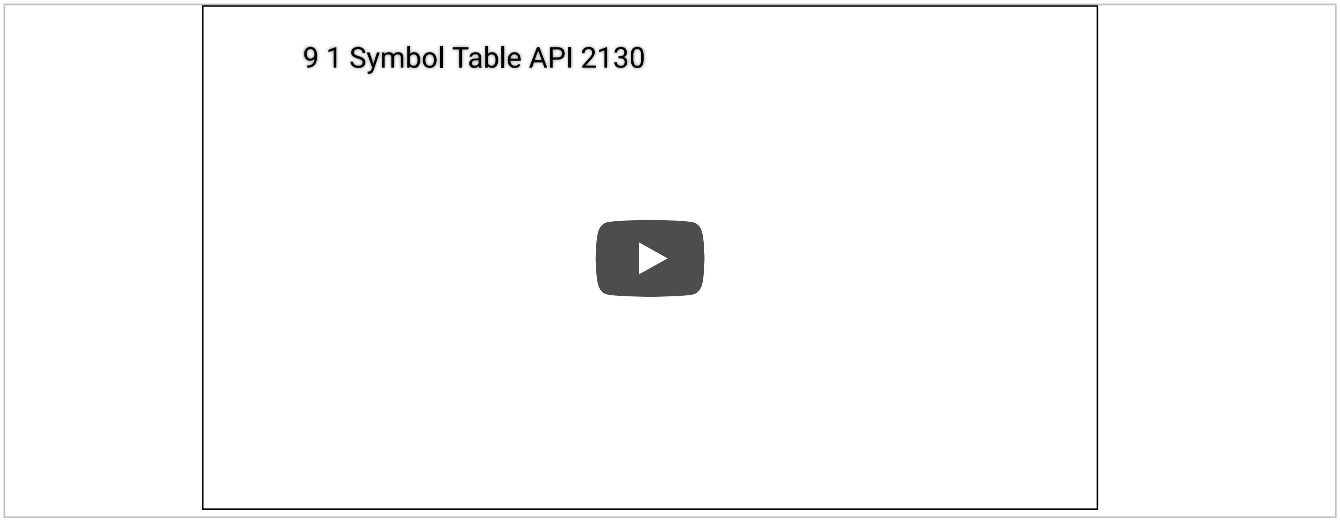
Algorithms and Data Structures

binary search tree - order-based methods - part 3

- `delete min & max`
  - *delete minimum*
  - *go left until finding a node that has a null left link*
  - *then replace link to that node by its right link*
  - *symmetric method works for delete maximum*
- `delete`
  - *proceed in a similar manner to delete a node with one or null childrem*
  - *for two or more - start by replacing current node with its successor*
  - *successor is node with smallest key in its right subtree*
- accomplish task of replacing x by its successor in four easy steps
  - *1. save a link to node to be deleted in `t`*
  - *2. set `x` to point to its successor `min(t.right)`*
  - *3. set right link of `x`*
    - supposed to point to BST containing all keys larger than `x.key`
    - to `deleteMin(t.right)`
      - the link to BST containing all keys that are larger than `x.key` after deletion
  - *4. set left link of `x` to `t.left`*
    - all keys that are less than both deleted key and its successor...

Video

symbol table API



Symbol table API - UP TO 5:40

Source - Symbol Table API - YouTube

binary search tree - usage - intro

- binary search tree (BST) has a non-linear insertion algorithm
- BST is similar in nature to a doubly-linked list
  - *a linked data structure*
  - *includes set of sequentially linked records, commonly known as nodes*
- each node defines three fields,
  - *link field - `previous` node in sequence of nodes*
  - *link field - `next` node in sequence of nodes*
  - *one `data` field*
- link fields may be represented using a common example of a convoy, a train &c.
  - *e.g. linked ships sailing in a convoy...*

binary search tree - usage - links and usage - part 1

- binary search tree defines its pointers as `left` and `right` to help indicate any duplication of logic &c.
- algorithm may be detected
  - *providing support for left and right traversal*
- binary search tree node's pointers are typically called *left* and *right*
  - *indicate subtrees of values relating to current value*
- simple JavaScript implementation of such a node is as follows

```
const node = {
    value: 123,
    left: null,
    right: null
}
```

- BST is a unique tree due to its inherent ordering of nodes based on value *binary search tree - usage - links and usage - part 2*

- any child nodes in a left subtree are always less than parent node's value
- converse holds for a right subtree
  - *values in subtree will always be greater*
- e.g.

```
         7
        / \
       5   9
      /     \
     2       11
    / \        \
   1   3        12
                  \
                   14
                  /
                 13
```

- traverse the tree and check key of current node
- if search key is less than current node's value
  - *follow left link*
  - *otherwise, follow right link*
- position of node values is based on a few factors
  - *value of node*
  - *value of root*
  - *order of insertion*
- e.g.
  - *root is set to 7*
  - *5 is less than root - insert as left link*
  - *9 is greater than root - insert as right link*
  - *2 is less than root - follow left link*
    - 2 is less than 5 - left link is null - insert as left link
  - *...*
- repeat for additional inserts...

Video - Algorithms and Data Structures

binary search trees - part 2

Binary Search Trees - Insert - UP TO 3:00

Source - Trees - Java - YouTube

Resources

Various

- Algorithms - YouTube
- Divide and Conquer - YouTube
- Memoisation - YouTube
- Quicksort - Java - YouTube
- Recursion & Fibonacci - YouTube
- Recursion and Fun - JavaScript - YouTube
- Recursion and the Call Stack - Java - YouTube
- So how does the browser actually render a website - YouTube
- Symbol Table API - YouTube
- Trees - Java - YouTube