Algorithms

January 23, 2020

Algorithm is a step-by-step procedure, which defines a set of instructios tobe executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages.

1 Greedy algorithms

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, grenerally greedy algorithms do not provide globally optimized solutions.

So greedy algorithms look for an easy solution at that point in time without considering how it impacts the future steps. It is similar how humans solve problems without going through the complete details of the inputs providees.

Most networking algorithms use the greedy approach. Example: * Traveling salesman problem * Prim's minimal spanning tree algorithm * Kruskal's minimal spanning tree algorithm * Dijkstra's minimal spanning tree algorithm

2 Dynamic programming

Dynamic programming involves dividing the bigger problem into smaller ones but unlike divide and conquer it does not involve solving each sub-problem independently. Rather the results of smaller sub-problems are remembered and used for similar or overlapping sub-problems. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems.

Example: * Fibonacci number series * Knapsack problem * Tower of Hanoi

3 Divide and conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. The solutions of all sub-problems are finally merged in order to obtain the solution of an original problem.

Brodly, we can understand divide-and-conquer in a three steps

3.0.1 Divide/break

This step involves breaking the problem into smaller sub-problesms. Sub-problems should represent a part of the origial problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible.

3.0.2 conquer/solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

3.0.3 Merge/combine

When the smaller sub-problems are solved, theis stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Example * Merge sort * Quick sort * Kruskal's minimal spanning tree algorithms * Binary serach

An example of divide and conquer programming approach is the binary search.

Binary search we take a sorted list of elements and start looking for an element at the middle of the list. If athe search value matches with the middle value in teh list we complete the search. Otherwise we eleminate half of the list of elements by choosing whether to process with the right or left half of the list depending on the vlaue of the item searched. This is possible as the list is sorted and it is much quicker than linea search.

```
In [1]: def binary_search(items, value):
            length = len(items)
            minimum = 0
            maximum = length

            while minimum <= maximum:
                midpoint = (minimum+maximum)//2
                print("Midpoint",midpoint)
                print("Midpoint Number",items[midpoint])

                if value == items[midpoint]:
                    return midpoint

                elif value > items[midpoint]:
                    minimum = midpoint + 1
                    print("Minimum:",minimum)
                else:
                    maximum = midpoint - 1
                    print("Maximum",maximum)

                if minimum > maximum:
                    return
```

```
In [2]: items = [2,7,19,34,53,72]
        # print(binary_search(items, 72))
        print(binary_search(items, 11))
```

```
Midpoint 3
Midpoint Number 34
Maximum 2
Midpoint 1
Midpoint Number 7
Minimum: 2
Midpoint 2
Midpoint Number 19
Maximum 1
None
```

4 Recursion

Recursion allows a function to call itself. Fixed steps of code get executed agian and again for new values.

binary search using recursion. We use an ordered list of items and design a recursive function to take in the list along with starting and ending index as input. Then binary search function calls itself till find teh searched item or concludes about its absence in the list.

```
In [3]: def binary_search(items, minimum, maximum, value):
            if minimum > maximum:
                return None
            else:
                midpoint = minimum + (maximum-minimum)//2
                if items[midpoint] > value:
                    return binary_search(items,minimum,midpoint-1,value)
                elif items[midpoint] < value:
                    return binary_search(items,midpoint+1,maximum,value)
                else:
                    return midpoint
```

```
In [4]: items = [8,11,24,56,88,131]
        print(binary_search(items,0,5,24))
```

```
2
```

```
In [5]: print(binary_search(items,0,5,51))
```

```
None
```

5 Backtracking

Backtracking is a form of recursion. But it involves choosing only option out of any possibilities. We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specific option does not give the required solution. We repeat these steps by going across each available option until we get the desired solution.

Below is an example of finding all possible order of arrangements of a given set of letters. When we choose a pair we apply backtracking to verify if that exact pair has already been created or not. If not already created, the pair is added to the answer list else it is ignored.

```
In [6]: def permute(items,s):
            if items == 1:
                return s
            else:
                return [y+x for y in permute(1,s) for x in permute(items-1,s)]

In [7]: print(permute(1, ["a","b","c"]))

['a', 'b', 'c']


In [8]: print(permute(2, ["a","b","c"]))

['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']


In [9]: print(permute(3, ["a","b","c"]))

['aaa', 'aab', 'aac', 'aba', 'abb', 'abc', 'aca', 'acb', 'acc', 'baa', 'bab', 'bac', 'bba', 'bbb
```

6 Tree traversal algorithms

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomlly access a node in a tree. There are three ways which we use to traverse a tree * In-order Traversal * Pre-order Traversal * Post-order Traversal

6.0.1 In-order Traversal

In this traversal method, the left subtee is visited first, then the root and later the right sub-tree. we should always remember that every node may represent a subreee itself.

```
In [10]: class Tree:
             def __init__(self,node):
                 self.node = node
                 self.right = None
                 self.left = None

             def insert(self,node):
                 if self.node:
                     if node < self.node:
                         if self.left is None:
```

```
                    self.left = Tree(node)
                else:
                    self.left.insert(node)
            elif node > self.node:
                if self.right is None:
                    self.right = Tree(node)
                else:
                    self.right.insert(node)
        else:
            self.node = node
    def inorder_traversal(self,root):
        res = []
        if root:
            res = self.inorder_traversal(root.left)
            res.append(root.node)
            res = res + self.inorder_traversal(root.right)
        return res

    def __repr__(self):
        return "Node: {}, (Right: {}, Left: {})".format(self.node,self.right,self.left)
```

In [11]:
```
root = Tree(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
```

In [12]:
```
root
```

Out[12]:
```
Node: 27, (Right: Node: 35, (Right: Node: 42, (Right: None, Left: None), Left: Node: 31
```

In [13]:
```
root.inorder_traversal(root)
```

Out[13]:
```
[10, 14, 19, 27, 31, 35, 42]
```

6.0.2 Pre-order traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

In [14]:
```
class Tree:
    def __init__(self,node):
        self.node = node
        self.right = None
        self.left = None

    def insert(self,node):
```

```
            if self.node:
                if node < self.node:
                    if self.left is None:
                        self.left = Tree(node)
                    else:
                        self.left.insert(node)
                elif node > self.node:
                    if self.right is None:
                        self.right = Tree(node)
                    else:
                        self.right.insert(node)
            else:
                self.node = node
        def inorder_traversal(self,root):
            res = []
            if root:
                res = self.inorder_traversal(root.left)
                res.append(root.node)
                res = res + self.inorder_traversal(root.right)
            return res

        def preorder_traversal(self, root):
            res = []
            if root:
                res.append(root.node)
                res = res + self.preorder_traversal(root.left)
                res = res + self.preorder_traversal(root.right)
            return res

        def __repr__(self):
            return "Node: {}, (Right: {}, Left: {})".format(self.node,self.right,self.left)
```

In [15]:
```
root = Tree(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.preorder_traversal(root))
```

```
[27, 14, 10, 19, 35, 31, 42]
```

In [16]:
```
root
```

Out[16]:
```
Node: 27, (Right: Node: 35, (Right: Node: 42, (Right: None, Left: None), Left: Node: 31
```

6.0.3 Post-order traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

```
In [17]: class Tree:
             def __init__(self,node):
                 self.node = node
                 self.right = None
                 self.left = None

             def insert(self,node):
                 if self.node:
                     if node < self.node:
                         if self.left is None:
                             self.left = Tree(node)
                         else:
                             self.left.insert(node)
                     elif node > self.node:
                         if self.right is None:
                             self.right = Tree(node)
                         else:
                             self.right.insert(node)
                 else:
                     self.node = node
             def inorder_traversal(self,root):
                 res = []
                 if root:
                     res = self.inorder_traversal(root.left)
                     res.append(root.node)
                     res = res + self.inorder_traversal(root.right)
                 return res

             def preorder_traversal(self, root):
                 res = []
                 if root:
                     res.append(root.node)
                     res = res + self.preorder_traversal(root.left)
                     res = res + self.preorder_traversal(root.right)
                 return res

             def postorder_traversal(self, root):
                 res = []
                 if root:
                     res = self.postorder_traversal(root.left)
                     res = res + self.postorder_traversal(root.right)
                     res.append(root.node)
                 return res
```

```
    def __repr__(self):
        return "Node: {}, (Right: {}, Left: {})".format(self.node,self.right,self.left)
```

In [18]:
```
root = Tree(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.postorder_traversal(root))
```

```
[10, 19, 14, 31, 42, 35, 27]
```

7 Sorting algorithms

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

- Bubble sort
- Merge sort
- Insertion Sort
- Shell sort
- Selection sort

7.0.1 Bubble sort

It is a comparision based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

In [19]:
```
def bubble_sort(items):
    for j in range(len(items)-1,0,-1):
        for i in range(j):
            if items[i] > items[i+1]:
                temp = items[i]
                items[i] = items[i+1]
                items[i+1] = temp
    return items
```

In [20]:
```
items = [19,2,31,45,6,11,121,27]
bubble_sort(items)
```

Out[20]:
```
[2, 6, 11, 19, 27, 31, 45, 121]
```

7.0.2 Merge sort

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

```
In [21]: def merge_sort(items):
             if len(items) <= 1:
                 return items
             middle = len(items)//2
             left_list = items[:middle]
             right_list = items[middle:]
             left_list = merge_sort(left_list)
             right_list = merge_sort(right_list)

             return list(merge(left_list,right_list))

         def merge(left_list,right_list):
             res = []
             while len(left_list) !=0 and len(right_list) !=0:
                 if left_list[0] < right_list[0]:
                     res.append(left_list[0])
                     left_list.remove(left_list[0])
                 else:
                     res.append(right_list[0])
                     right_list.remove(right_list[0])
             if len(left_list) == 0:
                 res = res + right_list
             else:
                 res = res + left_list

             return res
```

```
In [22]: merge_sort([64, 34, 25, 12, 22, 11, 90])
```

```
Out[22]: [11, 12, 22, 25, 34, 64, 90]
```

7.0.3 Insertion sort

Insertion sort involves finding the right place for a given element is a sorted list. So in the beginning we compare the first two element and sort them by comparing them. Then we pick the third elemetn and firnd its proper position among the prevesous two sorted elements. Thsi way we gradually go on adding more elements to already sorted lsit putting them in theeir proper position.

```
In [23]: def insertion_sort(items):
             for i in range(1, len(items)):
                 c = i-1
                 next_element = items[i]
                 while items[c] > next_element  and c >=0:
                     items[c+1] = items[c]
                     c -= 1
                     items[c+1] = next_element
             return items
```

```
In [24]: insertion_sort([19,2,31,45,30,11,121,27])

Out[24]: [2, 11, 19, 27, 30, 31, 45, 121]
```

7.0.4 Shell sort

Shell sort involves sorting elements which are away from each other. We sort a large sublist of a given list and go on reducing the size of the list until all elements are sorted. The below program finds the gap by equating it to half of the length of the list size and then starts sorting all elements in it. Then we keep resetting the gap until the entire list is sorted.

```
In [30]: def shell_sort(items):
             gap = len(items)//2

             while gap > 0:
                 for i in range(gap,len(items)):
                     temp = items[i]
                     j = i
                     while j >= gap and items[j-gap] > temp:
                         items[j] = items[j-gap]
                         j = j-gap
                     items[j] = temp
                 gap = gap//2
             return items

In [31]: shell_sort([19,2,31,45,30,11,121,27])

Out[31]: [2, 11, 19, 27, 30, 31, 45, 121]
```

7.0.5 Selection sort

In selection sort we start by findig the minimum value in a given list and move it to a sorted list. Then we repeat the process for each of the remaining elements in the unsorted lsit. The next element entering the sorted list is compared with the existing elements and placed at its correct position. So at the end all the elements from the unsorted list are sorted.

```
In [34]: def selection_sort(items):
             for i in range(len(items)):
                 minimum = i
                 for j in range(i+1,len(items)):
                     if items[minimum] > items[j]:
                         minimum = j
                 items[i],items[minimum] = items[minimum],items[i]
             return items

In [35]: selection_sort([19,2,31,45,30,11,121,27])

Out[35]: [2, 11, 19, 27, 30, 31, 45, 121]
```

8 Searching algorithms

8.0.1 Linear search

Every item is checked and if a match is found then that particular item is returned, otherwise the search contimutes till the end of the data structue.

```
In [7]: def linear_search(items, value):
            result = False
            c = 0
            while c<len(items) and result == False:
                if items[c] == value:
                    result = True
                    print("found in index: ",c)
                    break
                else:
                    c += 1
        #             print("did not found in index: ",c)

            return result
```

```
In [8]: l = [64, 34, 25, 12, 22, 11, 90]
        print(linear_search(l, 12))
        print(linear_search(l, 91))
```

```
found in index:  3
True
False
```

8.0.2 Interpolation search

This search algorithm works on the probing position of the required value. for this algorithm to work properly, the data collection should be in a sorted form and equally distributed. Initially, the probe position is the poisition of the middle most itme of the collection.If a match occurs, then the index of the item is returned. If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

```
In [14]: def interpolation_search(items,value):
             minimum = 0
             maximum = len(items)-1
             while minimum <= maximum and value >= items[minimum]  and value <= items[maximum]:
                 midpoint = int(minimum + (maximum - minimum)/(items[maximum] - items[minimum])*

                 if items[midpoint] == value:
                     return "Found "+str(value)+" at index "+str(midpoint)
                 if items[midpoint] < value:
                     minimum = midpoint + 1
             return "Value not in items"
```

```
In [15]: l = [2, 6, 11, 19, 27, 31, 45, 121]
         print(interpolation_search(l, 2))
```

```
Found 2 at index 0
```

9 Graph algorithms

9.0.1 Depth first traversal

Also called depth first search (DFS), this algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration. We implement DFS for a graph in python using the set data types as they provide teh required functinalities to keep track of visited and unvisited nodes.

```
In [30]: class graph:

             def __init__(self,gdict = {}):
                 self.gdict = gdict

         def dfs(graph, start, visited = None):
             if visited is None:
                 visited = set()
             visited.add(start)
             print(start)
             for next in graph[start] - visited:
                 dfs(graph, next, visited)
             return visited

         gdict = { "a" : set(["b","c"]),
                         "b" : set(["a", "d"]),
                         "c" : set(["a", "d"]),
                         "d" : set(["e"]),
                         "e" : set(["a"])
                         }


         dfs(gdict, 'a')
```

```
a
b
d
e
c
```

```
Out[30]: {'a', 'b', 'c', 'd', 'e'}
```

```
In [31]: a = {}
         print(type(a))
```

```
<class 'dict'>
```

```
In [32]: a = {'a','b','c'}
         b = set()
         b.add("c")
         b.add("d")
```

```
In [33]: a-b
```

```
Out[33]: {'a', 'b'}
```

```
In [34]: c = {'c','d'}
```

```
In [35]: a-c
```

```
Out[35]: {'a', 'b'}
```

9.0.2 Breadth first traversal

Also called breadth first search (BFS), this algorithm traverses a graph breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

We implement BFS for a graph in python using queue data struture. When we keep visiting the adjacent unvisited nodes and keep adding it to the quue. Then we start dequeqe only the node which is left with no unvisted nodes. We stop the program when there is no next adjacen tnode to be visited.

```
In [1]: import collections
        class graph:
            def __init__(self,gdict=None):
                if gdict is None:
                    gdict = {}
                self.gdict = gdict

        def bfs(graph, startnode):
        # Track the visited and unvisited nodes using queue
                seen, queue = set([startnode]), collections.deque([startnode])
                while queue:
                    vertex = queue.popleft()
                    marked(vertex)
                    for node in graph[vertex]:
                        if node not in seen:
                            seen.add(node)
                            queue.append(node)

        def marked(n):
            print(n)
```

```python
# The graph dictionary
gdict = { "a" : set(["b","c"]),
                "b" : set(["a", "d"]),
                "c" : set(["a", "d"]),
                "d" : set(["e"]),
                "e" : set(["a"])
                }

bfs(gdict, "a")
```

```
a
c
b
d
e
```