FIT1045 Very Brief Summary Notes S1 2021

Introduction to Algorithm and Python (Monash University)

Revision Notes

Intro:

Algorithm/Framework	Problems	Source
Selection Sort	Sorting	Lecture 8
Insertion Sort	Sorting	Lecture 8
Prim's Algorihm***	Minimum Spanning Tree	Lecture 10
Greedy*	Knapsack, Coin Exchange	Lecture 10
Brute Force Bitlists*	Knapsack	Lecture 11
*Brute Force Permutations**	Traveling Salesperson, N-Queens	Lecture 11, 15
Binary Search	Search (in sorted list)	Lecture 12
Merge Sort	Sorting	Lecture 12
Quick Sort	Sorting	Workshops
Backtracking*	N-queens, Hamiltonian Cycles, Knapsack	Lecture 15
Depth-first-search (iter, recur)	Reachability, Height of Binary Tree**	Lectures 16, 18
Breadth-first-search	Reachability, Shortest Path (unit weights	Lecture 16
Dijkstra's Algorithm	Shortest Path (weighted)	Lecture 17
Binary Search Tree Search	Search (in BST)	Lecture 18
Binary Search Tree Insertion	Insert (into BST)	Lecture 18, Workshop
Heap Extract Min	Find and Remove Minimum (in heap)	Lecture 19, Workshop
Heap Insertion	Insert (into heap)	Lecture 19
Heap Sort	Sorting	Lecture 20

Selection Sort and Insertion Sort:

```
def selection_sort(lst):
    for k in range(len(lst)-1):
        m = k
        for i in range(k+1, len(lst)):
            if lst[i] < lst[m]:
                m = i
        lst[m], lst[k] = lst[k], lst[m]
```

```
def insertion_sort(lst):
    for k in range(1, len(lst)):
        j = k
        while j > 0 and lst[j-1] > lst[j]:
            lst[j-1], lst[j] = lst[j], lst[j-1]
            j -= 1
```

Prims Algorithm/Minimum Spanning Tree

```
def min_extension(con, g):
    """I: vertices con connected in edge-weighted graph g
       O: minimum weight edge (i,j) of g such that
          i in con and j not in con"""
    min_weight = inf
    for i in con:
        for j in range(len(g)):
            if j not in con and 0 < g[i][j] < min_weight:
                v, w = i, j
                min_weight = g[i][j]
    return v, w
```

infinity; has to be imported from math

chained comparison; same as:
0 < g[i][j] and g[i][j] < min_weight

```
def minimum_spanning_tree(graph):
    """I: edge-weighted graph (adjacency matrix)
       O: minimum spanning tree of graph"""
    ...
```

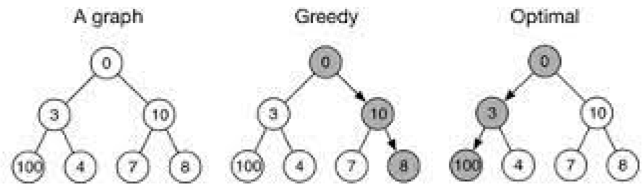
```
def empty_graph(n):
    c1 = 0
    c2 = 0
    rows = []
    res = []
    while c1 < n:
        while c2 < n:
            rows.append(0)
            c2 += n
        res.append(rows)
        c1 += 1
        c2 = 0
    return res
```

```
def min_extension(con, g):
    """I: vertices con connected in edge-weighted graph g
       O: minimum weight edge (i,j) of g such that
          i in con and j not in con"""
    ...
```

```
def minimum_spanning_tree(graph):
    """I: edge-weighted graph (adjacency matrix)
       O: minimum spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = [0]
    while len(con) < len(graph):
        i, j = min_extension(con, graph)
        tree[i][j], tree[j][i] = graph[i][j], graph[j][i]
        con += [j]
        #IV: tree contains MST of con
    return tree
```

Greedy Algorithm

So basically a greedy algorithm picks the locally optimal choice hoping to get the globally optimal solution.



A greedy algorithm fails to maximise the sum of nodes along a path from the top to the bottom because it lacks the foresight to choose suboptimal solutions in the current iteration that will allow for better solutions later

Binary Search

```
def bin_search(lst, x):
    """
    >>> bin_search([1,2,4,5,78,8,5,2,4,6,7,886,123456789,69, 3,46,100000, 2, 1000], 69)
    True
    """
    n = len(lst)
    lst_2 = sorted(lst)
    while n != 1:
        n = n//2
        if lst_2[n] == x:
            return True
        elif lst_2[n] > x:
            lst_2 = lst_2[:n]
        else:
            lst_2 = lst_2[n:]
    return False
```

```
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    while a <= b:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
    return None
```

Binary Search is an example of decrease and conquer as it checks within a sorted list if the value that is being searched is bigger or smaller than the middle value and it will cut off the lower half of the list if it's larger and vice-versa.

Merge Sort

```
def merge(lst1, lst2):
    """
    >>> merge([1, 2, 4, 6], [3, 5, 7, 8])
    [1, 2, 3, 4, 5, 6, 7, 8]
    """
    res = []
    n1, n2 = len(lst1), len(lst2)
    i, j = 0, 0
    while i < n1 and j < n2:
        if lst1[i] <= lst2[j]:
            res += [lst1[i]]
            i += 1
        else:
            res += [lst2[j]]
            j += 1
    return res + lst1[i:] + lst2[j:]
```

`def mergesort(ls):`		
`k, n = 1, len(ls)`	$O(1)$	$O(1)$
`while k < n:`	$O(\log n)$	$O(\log n)$
`nxt = []`	$O(\log n)$	$O(\log n)$
`for a in range(0, n, 2*k):`	$O(n)$	$O(n)$
`b, c = a + k, a + 2*k`	$O(n)$	$O(n)$
`nxt += merge(ls[a:b], ls[b:c])`	$O(n\log n)$	$O(n\log n)$
`ls = nxt`	$O(\log n)$	$O(\log n)$
`k = 2 * k`	$O(\log n)$	$O(\log n)$
`return ls`	$O(1)$	$O(1)$
	$O(n\log n)$	$O(n\log n)$

- Let k_i be chunk size after i iterations of loop
- Merging cost in i-th iteration:

$$\frac{n}{2k_i} T_{\text{merge}}(2k_i) = O\left(n\frac{2k_i}{2k_i}\right) = O(n)$$

- Resulting total merge cost: $O(n\log n)$

Merge Sort

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```

```
def mergesort(ls):
    n = len(ls)
    if n <= 1:
        return ls
    else:
        sub1 = mergesort(ls[:n//2])
        sub2 = mergesort(ls[n//2:])
        return merge(sub1, sub2)
```

Mergesort as a recursive function

Merge Sort is an example of divide and conquer as the function keeps being halved until its pair and then it is sorted as it is joined together at the end.

DFS and BFS Traversal

```
def dfs_traversal(graph, s):
    visited = []
    boundary = [s]
    while len(boundary) > 0:
        v = boundary.pop()
        visited += [v]
        for w in neighbours(v, graph):
            if w not in visited and w not in boundary:
                boundary.append(w)
    return visited
```

```
def bfs_traversal(graph, s):
    visited = []
    boundary = [s]
    while len(boundary) > 0:
        v = boundary.pop(0)
        visited += [v]
        for w in neighbours(v, graph):
            if w not in visited and w not in boundary:
                boundary.append(w)
    return visited
```

DFS: Remember the queuing method and how when pop is used it will take the last item from the queue

BFS: Remember the stack methods and how pop(0) will take the first number from the stack.