CANDIDATE NUMBER:

[TO FILL]

Version: IB/2

EGT3
ENGINEERING TRIPOS PART IIB

Thursday 27 April 2023 2 to 4.30

Module 4M26

ALGORITHMS AND DATA STRUCTURES

Answer not more than ***three*** *questions.*

All questions carry the same number of marks.

The ***approximate*** *percentage of marks allocated to each part of a question is indicated in the right margin.*

Write your candidate number ***not*** *your name on the cover sheet.*

The runtime of a successful test case should be no longer than 10 seconds when run on a standard DPO machine.

No access to internet or other resources is permitted.

Solutions should not use and import any python libraries. Only default Python data structures are to be used.

STATIONERY REQUIREMENTS
Single-sided script paper

SPECIAL REQUIREMENTS TO BE SUPPLIED FOR THIS EXAM
CUED approved calculator allowed
Engineering Data Book
DPO computer

10 minutes reading time is allowed for this paper at the start of the test.

You may not start to read the questions printed on the subsequent pages of this question paper until instructed to do so.

You may not remove any stationery from the Examination Room.

1.

(a) Let $L = [x_0, x_1, \cdots, x_{n-1}]$ be a list of 1D point coordinates on a line. Write the function, **pair**(L), which:

(i) finds the pair of distinct points, i and j, with the smallest distance, $D(i,j) = |x_i - x_j|$;

(ii) and outputs the indexes of the points of the pair found. The indexes should be outputed in an increasing order and separated with a comma (e.g. " `0,3` ").

Your solution should have its run time complexity strictly smaller than $O(n^2)$.

[30%]

Examples:

Input: `[0,1,2,0.1]`
Output: `0,3`

Input: `[0,3,2,1.9,8]`
Output: `2,3`

Constraints:

- $2 \leq n \leq 100$.
- $x_i \in \mathbb{R}$ and $|x_i| \leq 10^6$.
- The pair of points with smallest distance is guaranteed to be unique.

Code:

In [1]:

```python
def merge(left,right):
    n_l = len(left)
    n_r = len(right)
    n=n_l+n_r
    L_res=[{'val':0,'ind':0} for i in range(n)]
    i=0
    j=0
    for c in range(n):
        if i<n_l and (j>=n_r or left[i]['val']<right[j]['val']):
            L_res[c]=left[i]
            i+=1
        else:
            L_res[c]=right[j]
            j+=1
    return L_res

def merge_sort(A):

    if len(A)==1:
        return A

    left=merge_sort(A[:len(A)//2])
    right=merge_sort(A[len(A)//2:])
    A=merge(left,right)

    return A

def pair(L):
    # Insert code here

    A = [{'val':L[i],'ind':i} for i in range(len(L))]
    A=merge_sort(A)

    min_pair = [0,1]

    for i in range(1,len(L)):
        if abs(A[i]['val']-A[i-1]['val'])<abs(A[min_pair[1]]['val']-A[min_pair[0]]['v
            min_pair=[i-1,i]

    lid = min(A[min_pair[0]]['ind'],A[min_pair[1]]['ind'])
    rid = max(A[min_pair[0]]['ind'],A[min_pair[1]]['ind'])
    return str(lid)+','+str(rid)
```

Tests:

Run example test case 1:

In [2]:
```
input_value = [0,1,2,0.1]
print(pair(input_value))
```

```
0,3
```

Run example test case 2:

In [3]:
```
input_value = [0,3,2,1.9,8]
print(pair(input_value))
```

```
2,3
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

In [4]:
```
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution
evaluate_solution(question_id=1,question_part_id='a',function=pair,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 1, part a
Running question 1, part a, test id: 1
Input:
[0, 1, 2, 0.1]
Obtained output:
0,3
<<CORRECT>>
Running question 1, part a, test id: 2
Input:
[0, 3, 2, 1.9, 8]
Obtained output:
2,3
<<CORRECT>>
Running question 1, part a, test id: 3
Input:
[1, 2]
Obtained output:
0,1
<<CORRECT>>
Running question 1, part a, test id: 4
Input:
[100000.01, 2.01, 3.01, 4.01, 5.01, 6.01, 7.01, 8.01, 9.01, 10.01, 11.01, 12.01, 13.0
1, 14.01, 15.01, 16.01, 17.01, 18.01, 19.01, 20.01, 21.01, 22.01, 23.01, 24.01, 25.0
1, 26.01, 27.01, 28.01, 29.01, 30.01, 31.01, 32.01, 33.01, 34.01, 35.01, 19.4]
Obtained output:
18,35
```

```
<<CORRECT>>
Running question 1, part a, test id: 5
Input:
[-1.1, -3.1, -6.3, -3.005, 9, 12, 14.5, 18.8, 100000]
Obtained output:
1,3
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
----------------------------------
```

(b) Sketch a detailed proof of correctness of your algorithm described in Part (a) and derive its worst-case time complexity.

[15%]

The algorithm consists of two key steps: (i) sorting of the point coordinates in an increasing order using MergeSort and (ii) checking all neighbouring pairs of sorted point coordinates for the shortest distance pair.
Note that the smallest distance pair, (a, b), is bound to be one of the neighbouring element pairs in the sorted list. This can be proven by contradiction - if a pair of closest points was separated by at least one point, c, in the list then then $D(a,b) > min(D(a,c), D(b,c))$. It takes $\Theta(n)$ time to look through the sorted list of points.
The correctness of MergeSort can be proven by considering mathematical induction on the length of the array. It is straight forward that MergeSort gives a correct result for $n = 1$ as its base case. Lets assume that it is correct for any array of lenght, $k > 1$. Since $k > 1$ then two subsequent recursive $MergeSort$ steps on $L[: k//2]$ and $L[k//2 :]$ will obtain the correctly sorted arrays. Furthermore $merge$ step maintains the loop invariant that after each step of the loop, L_res, is a strictly sorted array. Hence the whole procedure obtains a strictly sorted array of length $k + 1$.
The computation time of the MergeSort can be described using the following recursive relationship

$$T(n) = \begin{cases} 2 * T(\lfloor n/2 \rfloor) + \Theta(n), & \text{if } n > 1 \\ \Theta(1), & \text{otherwise} \end{cases}$$

The substitution method can be used. Lets guess that $T(n) = O(n \log_2 n)$. In such case $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor log_2(\lfloor n/2 \rfloor)$.
Hence
$T(n) \leq 2(c\lfloor n/2 \rfloor \log_2(\lfloor n/2 \rfloor)) + cn \leq cn \log_2(n/2) + cn = cn \log_2 n - cn + cn = cn \log_2 n$.
Similarly, one can prove that $T(n) = \Omega(n \log_2 n)$. Hence $T(n) = \Theta(n \log_2 n)$.

(c) Give brief answers to the following questions.

(i) Explain what aspects of computing costs are *not* captured when you specify costs using Big-O notation and why it is still often found useful?

[10%]

Big-O misses out constant factors, it is only an upper bound and there is no guarantee that it is sharp. It may not apply at all in small cases. The key advantage is that Big-O notation allows to analyse algorithms independent to technological advances (e.g. increase in processing simple operations) and simplifies the analysis of algorithms by only looking at the assymptotic behaviour.

(ii) Explain what is meant by "amortized" cost analysis, giving an example from the realm of data structures and algorithms where it is helpful.

[10%]

In amortized analysis, the time required to perform a sequence of data-structure operations over all the operations performed. Amortized analysis guarantees the *average performance* of each operation in the worst case. A good example includes $Multipop$ operation on a stack. The worst case of

$Multipop$ operation for a stack of n elements is $O(n)$, however if we consider a sequence of n operations $Push$, Pop and $Multipop$ operations, their amortised cost per operation is $O(1)$, with total amortized cost being $O(n)$.

(iii) Is it the case that every function of the form $f(n) = An^k$ is asymptotically bounded by $O(2^n)$, given that A and k are constants? Justify your answer.

[10%]

For this statement to be true, we need to prove that $\exists c, n_0 \in \mathbb{R}^+$ s.t. $An^k \leq c2^n$ for all $n >= n_0$. This is equivalent to $\frac{An^k}{2^n} \leq c$. Let $g(n) = \frac{An^k}{2^n}$. We have $\frac{g(n+1)}{g(n)} = \frac{(1+\frac{1}{n})^k}{2}$. For any k we can choose n_0 such that $\frac{(1+\frac{1}{n_0})^k}{2} \leq 1$, hence $g(n+1) \leq g(n)$ for all $n \geq n_0$. Hence $g(n) \leq c = g(n_0)$ for all $n \geq n_0$.

(d) You are given an integer list, $L = [x_0, x_1, \cdots, x_{n-1}]$. Write the function, **subsequence**(L), to find the length of the longest strictly increasing subsequence of the list, L. You can assume that the elements of the list, L, are unique.

Note, a subsequence is a list that can be derived from another list by deleting some or no elements without changing the order of the remaining elements. Your algorithm should have run time complexity $\Theta(n^2)$.

[10%]

Examples:

Input: `[5,2,1,1.6,3.5,8,4.5]`
Output: 4
Explanation: An example of one of the longest increasing subsequences is `[1,1.6,3.5,4.5]` .

Input: `[-5,1,8.5,3,2,12,7]`
Output: 4
Explanation: An example of one of the longest increasing subsequences is `[-5,1,3,7]` .

Constraints:

- $1 \leq n \leq 20.$
- $x_i \in \mathbb{R}$ and $|x_i| \leq 10^6$.

Code:

In [5]:
```python
def subsequence(L):

    # Insert code here

    S = [0 for i in range(len(L)+1)]

    for i in range(len(L)):
        S[i+1]=1
        for j in range(0,i):
            if L[i]>L[j]:
                S[i+1] = max(S[i+1],S[j+1]+1)
    return S[len(L)]
```

Tests:

Run example test case 1:

```
In [6]: input_value = [5,2,1,1.6,3.5,8,4.5]
        print (subsequence(input_value))
```

```
4
```

Run example test case 2:

```
In [7]: input_value = [-5,1,8.5,3,2,12,7]
        print (subsequence(input_value))
```

```
4
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

```
In [8]: #DO NOT EDIT THIS CODE!
        from evaluation_script import evaluate_solution
        evaluate_solution(question_id=1,question_part_id='d',function=subsequence,
                          test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 1, part d
Running question 1, part d, test id: 1
Input:
[5, 2, 1, 1.6, 3.5, 8, 4.5]
Obtained output:
4
<<CORRECT>>
Running question 1, part d, test id: 2
Input:
[-5, 1, 8.5, 3, 2, 12, 7]
Obtained output:
4
<<CORRECT>>
Running question 1, part d, test id: 3
Input:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Obtained output:
15
<<CORRECT>>
Running question 1, part d, test id: 4
Input:
[1]
Obtained output:
1
<<CORRECT>>
Running question 1, part d, test id: 5
Input:
[7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8]
Obtained output:
1
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
----------------------------------
```

(e) Provide key steps of an algorithm which could solve the task described in Part (d) in run time complexity $O(n \log n)$.

[15%]

When computing the largest increasing subsequence for sequence $L[:i]$, the $\Theta(n^2)$ solution requires to check whether the longest increasing subsequence can be formed with all possible $i-1$ potential subsequences. However, this full search is not necessary if we use the following procedure:

(1) Sort the sequence elements in an increasing order using MergeSort. Worst case time complexity $\Theta(n \log_2 n)$.
(2) Build a perfectly balanced binary search tree of the sorted array. This can be done by utilising binary search. For each node in this binary tree store the attribute $v.lis$ which tells what would be the longest increasing subsequence formulated by inserting a node equal or larger than $v.key$.
(3) Initially for all v, we have $v.lis = 0$. Then we start computing the longest increasing subsequence for the first item $L[0]$, second item (ie. $L[:1]$) and so on. We use function **max_value**$(root, L[i])$ to obtain the lenght, lis, of the longest possible increasing sequence in the list so far(ie. $L[:i]$ when item i is considered). Once it is obtained, we find the node with value $L[i]$ in the tree, and update its lis attribute. We then **propagate** this value up the tree. Note that this value should be propagated only to the parents of the left child (i.e. $v.parent.key > v.key$) in the path towards the root.

In [9]:
```
def max_value(v,key):
    if v is None:
        return 0
    if key >v.key:
        return max(v.lis,max_value(v.right_child,key))
    elif key == v.key:
        return v.lis
    return max_value(v.left_child,key)
def propagate(v,lis):
    if v.parent is None:
        return
    if v.parent.left_child == v:
        if lis > v.parent.lis:
            v.parent.lis = lis
    propagate(v.parent,lis)
```

2.

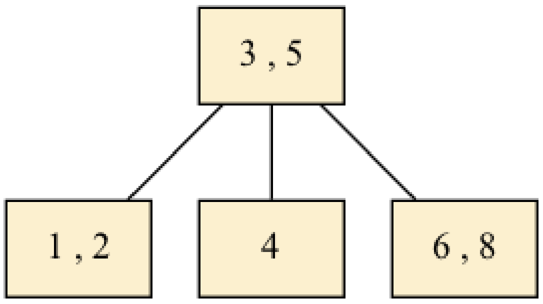
A B-tree of minimum degree t is a data structure with the property that every node may contain at most $2t - 1$ keys. For this question, we will assume that all keys are distinct, that all nodes contain keys but not values, and that all nodes of the B-tree fit into memory (no disk read/writes are required).

(a) Implement a function to search for a given query key, key, starting from a given node, u, in a B-tree by completing the **search**$(self, u, key)$ function below. If the query key exists in the B-tree, your function should return the node that contains the query key together with the index of the query key among its keys. Otherwise, your function should return `None`. For a B-tree of n keys, your solution should run in $O(t \log_t n)$ time.

[35%]

Examples:

Your searches will be executed on a B-tree of minimum degree $t = 2$ into which the following sequence of keys has been inserted prior to your search: [3, 5, 6, 2, 4, 1, 8]. The resulting B-tree is visualised below.



Input: 6
Output: `[{'is_leaf': True, 'keys': [6, 8], 'children': []}, 0]`
Explanation: The key `6` appears in the B-tree as the 0th key of a leaf node.

Input: 9
Output: `None`
Explanation: The key `9` does not appear in the B-tree.

Constraints:

- Every key is an integer.
- $0 \leq n \leq 10^2$.
- $t = 2$.

Code:

In [10]:
```
from btree_utils import insert

class Btree:

    def __init__(self, t):
        self.t = t
        self.root = {"is_leaf": True, "keys": [], "children": []}

    def search(self, u, key):

        # Insert code here

        i = 0
        while i < len(u["keys"]) and key > u["keys"][i]:
            i += 1

        if i < len(u["keys"]) and key == u["keys"][i]:
            return [u, i]

        if u["is_leaf"]:
            return None

        return self.search(u["children"][i], key)

def find_key(key):

    btree = Btree(t=2)

    for k in [3, 5, 6, 2, 4, 1, 8]:
        insert(btree, k)

    return btree.search(btree.root, key)
```

Tests:

Run example test case 1:

```
In [11]:  input_value = 6
          print(find_key(input_value))
```

```
[{'is_leaf': True, 'keys': [6, 8], 'children': []}, 0]
```

Run example test case 2:

```
In [12]:  input_value = 9
          print(find_key(input_value))
```

```
None
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

```
In [13]:  #DO NOT EDIT THIS CODE!
          from evaluation_script import evaluate_solution
          evaluate_solution(question_id=2,question_part_id='a',function=find_key,
                            test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 2, part a
Running question 2, part a, test id: 1
Input:
6
Obtained output:
[{'is_leaf': True, 'keys': [6, 8], 'children': []}, 0]
<<CORRECT>>
Running question 2, part a, test id: 2
Input:
9
Obtained output:
None
<<CORRECT>>
Running question 2, part a, test id: 3
Input:
4
Obtained output:
[{'is_leaf': True, 'keys': [4], 'children': []}, 0]
<<CORRECT>>
Running question 2, part a, test id: 4
Input:
8
Obtained output:
[{'is_leaf': True, 'keys': [6, 8], 'children': []}, 1]
<<CORRECT>>
Running question 2, part a, test id: 5
Input:
5
Obtained output:
[{'is_leaf': False, 'keys': [3, 5], 'children': [{'is_leaf': True, 'keys': [1, 2], 'c
hildren': []}, {'is_leaf': True, 'keys': [4], 'children': []}, {'is_leaf': True, 'key
s': [6, 8], 'children': []}]}, 1]
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
----------------------------------
```

(b) Derive a mathematical expression for the minimum height of a B-tree containing n keys in terms of its minimum degree t and the number of keys n.

[15%]

In a minimum height B-tree, all internal nodes have $2t$ children, so there are $1 + 2t + (2t)^2 + (2t)^3 + \ldots$ nodes in total. Equivalently, the number of keys is equal to:

$$n = (2t-1)(1 + 2t + (2t)^2 + (2t)^3 + \dots) \tag{1}$$
$$= (2t-1)\Big(\frac{1-(2t)^{h+1}}{1-2t}\Big) \tag{2}$$
$$= (2t)^{h+1} - 1 \tag{3}$$

Rearranging and taking logs (base 2t):

$$h + 1 = log_{2t}(n+1) \tag{4}$$
$$\implies h = log_{2t}(n+1) - 1 \tag{5}$$

This only applies for values of n for which all layers of the B-tree are rammed full. For other values of n, we won't achieve this perfect structure, so we have that:

$$h = \lceil log_{2t}(n+1) - 1 \rceil \tag{6}$$

(c) In addition to searching for a key according to the value of the key, it is often useful to search for keys by their rank. The rank of a key is its position in the list of all keys stored in the B-tree, arranged in ascending order. For example, if a B-tree contains the keys $2, 4, 7$ and 8, the rank of key 2 is 0, the rank of key 4 is 1 and so on.

(i) Describe an algorithm that returns the key with the desired rank. Your algorithm should exhibit $O(n)$ runtime complexity where n is the total number of keys in the tree.

[15%]

This can be achieved by executing an inorder traversal of the B-tree. In detail: - we build a list, $inorder\_keys$, of keys arranged in ascending order by conducting an inorder traversal
- at each node we visit, append the keys of the node to $visited\_keys$, interleaving the appending with visiting each child node to ensure an *inorder* ordering.
- once the traversal is complete, we simply return index n of $visited\_keys$.
Reference code to implement this algorithm is provided below.

In [14]:
```python
def inorder_traversal(node, inorder_keys):

    for ii, key in enumerate(node["keys"]):
        if not node["is_leaf"]:
            keys = inorder_traversal(node["children"][ii], inorder_keys)
        inorder_keys.append(key)

    if not node["is_leaf"]:
        # handle the last child
        inorder_keys = inorder_traversal(node["children"][-1], inorder_keys)

    return inorder_keys

def find_key_with_rank(root, rank):

    inorder_keys = inorder_traversal(root, inorder_keys=[])

    return inorder_keys[rank]

# example usage
btree = Btree(t=2)
for k in [3, 5, 6, 2, 4, 1, 8]:
    insert(btree, k)
find_key_with_rank(btree.root, rank=1)
```

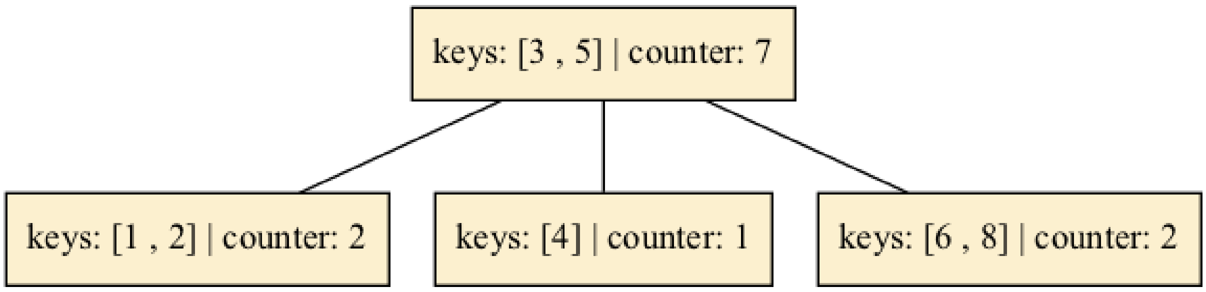
Out[14]:
```
2
```

(ii) Now suppose that we can store an additional non-negative integer counter at each internal node of the B-tree that stores the count of all keys in its subtree (this count includes its own keys). Implement a function, **find_key_with_rank_efficient**$(rank)$, that makes use of counters to retrieve the key with a rank of $rank$ in a B-tree in $O(t \log_t n)$ time.

[15%]

Examples:

As in Part (a), your searches will be executed on a B-tree of minimum degree, $t = 2$, that has had the following sequence of keys already inserted prior to your search: [3, 5, 6, 2, 4, 1, 8]. All counters have been set and stored as a key-value pair of the form `"counter": <counter_value>` in the dictionary representing each node (visualised below).


keys: [3 , 5] | counter: 7
keys: [1 , 2] | counter: 2
keys: [4] | counter: 1
keys: [6 , 8] | counter: 2

Input: `0`
Output: 1
Explanation: The key 1 appears in the B-tree with rank 0.

Input: 3
Output: 4
Explanation: The key 4 appears in the B-tree with rank 3.

In [15]:

```python
from btree_utils import set_subtree_counters

def find_key_with_rank_efficient(rank):

    btree = Btree(t=2)

    for key in [3, 5, 6, 2, 4, 1, 8]:
        insert(btree, key)

    set_subtree_counters(btree.root) #Sets the counters to correct values for the who

    # Insert code here

    return find_key_with_rank_recursive(btree.root, rank, total=0)

def find_key_with_rank_recursive(node, rank, total):

    if node["is_leaf"]:

        return node["keys"][rank - total]

    else:

        ii = 0

        while rank >= (total + node["children"][ii]["counter"]):

            total += node["children"][ii]["counter"] # skip over the number of keys i

            if total == rank:
                return node["keys"][ii]
            else:
                total += 1 # skip current key

            ii += 1

        return find_key_with_rank_recursive(node["children"][ii], rank, total)
```

Tests:

Run example test case 1:

In [16]:

```
input_value = 0
print(find_key_with_rank_efficient(input_value))
```

```
1
```

Run example test case 2:

In [17]:

```
input_value = 3
print(find_key_with_rank_efficient(input_value))
```

```
4
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

In [18]:

```
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution
evaluate_solution(question_id=2,question_part_id='c',function=find_key_with_rank_effi
                  test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 2, part c
Running question 2, part c, test id: 1
Input:
0
Obtained output:
1
<<CORRECT>>
Running question 2, part c, test id: 2
Input:
3
Obtained output:
4
<<CORRECT>>
Running question 2, part c, test id: 3
Input:
6
Obtained output:
8
<<CORRECT>>
Running question 2, part c, test id: 4
Input:
5
Obtained output:
6
<<CORRECT>>
Running question 2, part c, test id: 5
Input:
2
Obtained output:
3
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
----------------------------------
```

(iii) Describe a modified algorithm that improves the runtime complexity of returning the key with a given rank to $O(\log n)$ irrespective of how t is chosen as a function of n. Note, $\log_t n \cdot \log_2 t = \log_2 n$ for $t > 0$. Your algorithm may redefine the meaning of the counter at each node under the constraints that: (i) the counter remains a non-negative integer; (ii) the counter values can be set in a single traversal of the whole tree.

[20%]

One algorithm to solve this problem is as follows: 1. Redefine the counters to keep track of the cumulative key count prior to the current node (for an inorder traversal). 2. When executing the search down the tree for the key with a given rank: - if the current node is a leaf node, return the approapriate key by indexing in $O(1)$ time - otherwise, conduct a binary search over the cumulative counters of the children in $O(\log_2 t)$ time. If the query rank exactly matches the rank of one of the keys, return it in $O(1)$ time. Otherwise, recurse into the appropriate child. Reference code to implement this algorithm is provided below.

In [19]:
```
def set_cumulative_counters(node, cumulative_count=0):
    # use another inorder traversal to keep track of cumulative key count prior to cu
    node["counter"] = cumulative_count
    for ii, key in enumerate(node["keys"]):
        if not node["is_leaf"]:
            cumulative_count = set_cumulative_counters(node["children"][ii], cumulati
        cumulative_count += 1 # account for current key
    if not node["is_leaf"]:
        # handle the last child
        cumulative_count = set_cumulative_counters(node["children"][-1], cumulative_c
    return cumulative_count

def binary_search(counters, query, head=0, tail=None):
    # find largest index that is less than or equal to query
    # assumes query >= counters[0]

    if tail is None:
        tail = len(counters)

    # handle out-of-range at endpoint
    if query > counters[-1]:
        return len(counters)-1

    ptr = (head + tail) // 2
    if query == counters[ptr]:
        return ptr
    elif tail - head <= 1:
        return ptr
    elif query < counters[ptr]:
        return binary_search(counters, query=query, head=head, tail=ptr)
    else:
        return binary_search(counters, query=query, head=ptr, tail=tail)

def find_key_with_rank_more_efficient(node, rank):
    if node["is_leaf"]:
        return node["keys"][rank - node["counter"]]
    else:
        # do a single pass down the tree with binary search at each node over the chl
        counters = [child["counter"] for child in node["children"]]
        ii = binary_search(counters, rank)
        if ii < len(counters) - 1 and rank + 1 == counters[ii+1]:
            return node["keys"][ii]
        else:
            return find_key_with_rank_more_efficient(node["children"][ii], rank)

def run_rank_computation(rank):
    btree = Btree(t=2)
    for key in [3, 5, 6, 2, 4, 1, 8]:
        insert(btree, key)
    set_cumulative_counters(btree.root)
    return find_key_with_rank_more_efficient(btree.root, rank)
```

In [20]:
```
# re-use the tests from the previous part of the question, since the task is the same
from evaluation_script import evaluate_solution
evaluate_solution(question_id=2,question_part_id='c',function=run_rank_computation,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 2, part c
Running question 2, part c, test id: 1
Input:
0
Obtained output:
1
<<CORRECT>>
Running question 2, part c, test id: 2
Input:
3
Obtained output:
4
<<CORRECT>>
Running question 2, part c, test id: 3
Input:
6
Obtained output:
8
<<CORRECT>>
Running question 2, part c, test id: 4
Input:
5
Obtained output:
6
<<CORRECT>>
Running question 2, part c, test id: 5
Input:
2
Obtained output:
3
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
----------------------------------
```

3.

(a) Write the function, **scheduler**(L), which: (i) takes a list, L, as an input. This list in turn contains two lists, the first of which, T, stores tasks represented as strings (e.g. `"homework"`). The second list, C, stores pairwise constraints (e.g. `"eat-homework"`) for the order of the tasks (e.g. `"homework"` should be performed only after `"eat"`); (ii) finds a schedule which respects all the pairwise constraints.

The schedule is represented as a sequence of tasks separated by `"-"` . If no valid schedule is possible, the function outputs a string, `"impossible"` . The worst case runtime of your solution should be not worse than $O(n+m)$, where n is the number of tasks and m is the number of constraints.

Note that a function, **to_adjacency_representation**(T, C), is provided to convert the task list, T, and the constraint list, C, into an adjacency list representation. You can ignore the costs of this operation in your reasoning.

[35%]

Examples:

Input: `[["homework","sleep","drive","eat","learn"],["sleep-drive","eat-drive","eat-homework","learn-drive"]]`
Output: `sleep-eat-learn-drive-homework`

Input: `[["homework","sleep","drive","eat","learn"],["sleep-drive","eat-drive","homework-drive","learn-drive","drive-eat"]]`
Output: `impossible`

Constraints:

- $1 \leq n \leq 50.$
- $0 \leq m \leq 50.$

Code:

In [21]:

```python
def to_adjacency_representation(T,C):
    V={}
    Adj=[]
    for i in range(len(T)):
        V[T[i]]=i
        Adj.append([])
    for i in range(len(C)):
        v0 = C[i].split('-')[0]
        v1 = C[i].split('-')[1]
        Adj[V[v0]].append(V[v1])
    return V,Adj

def BFS(bfs_tree, time, v,v_list):
    v['start_time']=time
    assert v['color']=='white'
    if v['color']=='white':
        v['color']='gray'
        for i in range(len(v['neighbours'])):
            v_n = bfs_tree[v['neighbours'][i]]
            if v_n['color'] == 'white':
                time = time+1
                v_n['parent']=v['id']
                time,v_list=BFS(bfs_tree,time,v_n,v_list)

            elif v_n['color'] == 'gray':
                v['loop_detected']=True
        v['color']='black'
        time+=1
        v['stop_time']=time
        v_list= [v]+v_list
    return time,v_list

def scheduler(L):

    T, C = L[0], L[1]
    V, Adj = to_adjacency_representation(T,C)

    # Insert code here

    bfs_tree = [{'name': T[i],'id':i,'parent':-1,'neighbours':Adj[i], 'loop_detected'

    time=0
    v_list=[]
    for v in bfs_tree:
        if v['color']=='white':
            time,v_list = BFS(bfs_tree,time,v,v_list)
            time +=1
    result=v_list[0]['name']
    if v_list[0]['loop_detected']==True:
        return "impossible"
    for i in range(1,len(v_list)):
        if v_list[i]['loop_detected']==True:
            return "impossible"
        result+="-"+v_list[i]['name']
    return result
```

Tests:

Run example test case 1:

In [22]:
```
input_value = [["homework","sleep","drive","eat","learn"],
               ["sleep-drive","eat-drive","eat-homework","learn-drive"]]
print (scheduler(input_value))
```

```
learn-eat-sleep-drive-homework
```

Run example test case 2:

In [23]:
```
input_value = [["homework","sleep","drive","eat","learn"],
               ["sleep-drive","eat-drive","homework-drive","learn-drive","drive-eat"]
print (scheduler(input_value))
```

```
impossible
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

In [24]:
```
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution
from evaluation_script import valid_solution_scheduler
evaluate_solution(question_id=3,question_part_id='a',function=scheduler,
                  comparison_function=valid_solution_scheduler,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 3, part a
Running question 3, part a, test id: 1
Input:
[['homework', 'sleep', 'drive', 'eat', 'learn'], ['sleep-drive', 'eat-drive', 'eat-ho
mework', 'learn-drive']]
Obtained output:
learn-eat-sleep-drive-homework
<<CORRECT>>
Running question 3, part a, test id: 2
Input:
[['homework', 'sleep', 'drive', 'eat', 'learn'], ['sleep-drive', 'eat-drive', 'homewo
rk-drive', 'learn-drive', 'drive-eat']]
Obtained output:
impossible
<<CORRECT>>
Running question 3, part a, test id: 3
Input:
[['homework'], []]
Obtained output:
homework
<<CORRECT>>
Running question 3, part a, test id: 4
Input:
[['1', '2', '3', '4', '5', '6'], ['1-2', '3-2', '3-1', '4-3', '5-4', '6-4']]
Obtained output:
6-5-4-3-1-2
<<CORRECT>>
Running question 3, part a, test id: 5
```

```
Input:
[['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8', 'h9', 'h10'], ['h1-h2', 'h2-h3', 'h
3-h4', 'h4-h5', 'h5-h6', 'h6-h7', 'h7-h8', 'h8-h9', 'h9-h10', 'h10-h1']]
Obtained output:
impossible
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
-----------------------------------
```

(b) Give brief answers to the following questions.

(i) Derive worst-case time complexity of your proposed algorithm in Part (a).

[10%]

The proposed method is performing Depth First Search (DFS). Since each vertice is marked from white into gray and black exactly once and all edges are explored for each vertex exactly once we have worst case time complexity $\Theta(V+E)$. Maintaining the list v_list of vertices in reverse visited order is $\Theta(V)$. Producing output is $O(V)$. Hence the total worst-case time complexity is $\Theta(V+E)$.

(ii) Explain how depth first search (DFS) can be used to find the strongly connected components in a directed acyclic graph.

[10%]

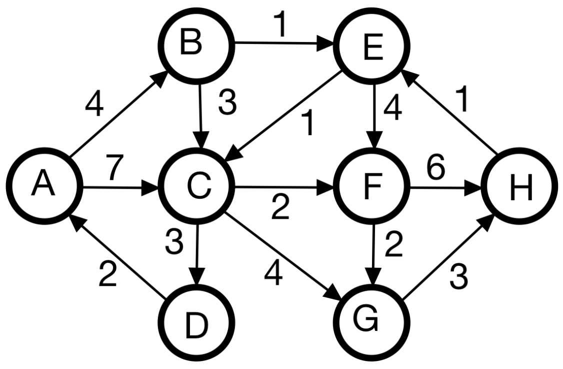
First, Depth First Search should be run on a graph G to compute finishing times $u.f$ for each vertex u. Then DFS should be called again on the transposed graph G^T by considering the vertices in the decreasing finishing time $u.f$. The vertices of each tree in the resulting depth-first forest forms a separate strongly connected component. Run time complexity: $\Theta(V+E)$. The correctness of this algorithm relies on the observation that the component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ is a directed acyclic graph. Note that reversing edges and considering vertices from the largest finish time upwards guarantees that the each strongly connected component considered is the leaf in the transposed graph of strongly connected components not yet considered.

(iii) Explain how the longest cycle can be found efficiently in a directed graph in which each node has at most one outgoing edge.

[10%]

We can perform the detection of strongly connected components in $\Theta(V+E)$. Each component must consist of a single simple cycle involving all the vertices in that component. This statement can be proven by contradiction. Lets say two loops exist which share a set of vertices. Then at least one vertice has to have at least two outgoing edges, otherwise the two loops would coincide. The strongly connected component with most vertices will give the length of the longest cycle.

(c) Answer questions about the graph illustrated below.



(i) List vertices in the order in which they would be explored when running the Dijkstra's algorithm, starting from vertex A. A vertex is considered explored when *relaxation* operation is performed on all edges outgoing from the vertex in question.

[5%]

A-B-E-C-F-D-G-H

(ii) List vertices in the order in which they would be explored when running the breadth first search algorithm starting from vertex A. A vertex is considered explored when its color is set to *black*. In the case that there is ambiguity in the order of exploration any valid ordering can be provided.

[5%]

A-B-C-E-D-G-F-H

(iii) Assume this graph is transformed into an undirected graph by replacing all directed edges with equivalent undirected edges with corresponding weights. Write out the edges of the minimum spanning tree discovered by Primm's algorithm in the same order as they would be added to this tree. The algorithm is run starting from vertex A.

[5%]

A-D-C-E-B-H-F-G MST length: 12

(d) Write the function, **jobs**(L), which:

(i) takes a list, L, as an input. The first element of this list is another list, S, containing names of students. The second element is also a list, V, containing the names of job vacancies. The third element is a list, A, containing job applications represented by the student's name, followed by a dash, " − ", and the vacancy name to which this student wants to apply and;

(ii) finds and outputs the maximum number of job vacancies that can be filled. Each student can apply for multiple jobs but can only take at most one job and each job vacancy can be assigned at most to one student.

[20%]

Examples:

Input: `[["S1","S2","S3"],["Cam","Ox","Lon"],["S1-Cam","S1-Ox","S1-Lon","S2-Cam","S3-Cam"]]`
Output: 2

Input: `[["John","Peter"],["Kings","Trinity","Jesus"],["John-Kings","John-Trinity","John-Jesus","Peter-Jesus"]]`
Output: 2

Constraints:

- $1 \leq \text{len}(S) \leq 20.$
- $1 \leq \text{len}(V) \leq 20.$
- $0 \leq \text{len}(A) \leq 100.$

Code:

In [25]:

```python
def jobs(L):

    # Insert code here
    V, Adj = to_adjacency_representation(L[0]+L[1],L[2])

    n=len(L[0])+len(L[1])

    Adj.append([])
    Adj.append([])

    for i in range(len(L[0])):
        Adj[n].append(i)
    for i in range(len(L[1])):
        Adj[i+len(L[0])].append(n+1)

    while True:
        bfs_tree = [{'neighbours':Adj[i], 'id':i,'parent':-1,'loop_detected':False,'c

        _,_= BFS(bfs_tree,0,bfs_tree[n],[])

        if bfs_tree[n+1]['color']=='white':
            break

        v_path=bfs_tree[n+1]

        while not v_path['parent'] == -1:
            v_path_parent = bfs_tree[v_path['parent']]
            Adj[v_path_parent['id']].remove(v_path['id'])
            Adj[v_path['id']].append(v_path_parent['id'])
            v_path=v_path_parent

    return len(L[0])-len(Adj[n])
```

Tests:

Run example test case 1:

In [26]:
```
input_value = [["S1","S2","S3"],["Cam","Ox","Lon"],["S1-Cam","S1-Ox","S1-Lon",
                                                     "S2-Cam","S3-Cam"]]
print (jobs(input_value))
```

```
2
```

Run example test case 2:

In [27]:
```
input_value = [["John","Peter"],["Kings","Trinity","Jesus"],["John-Kings",
                                "John-Trinity","John-Jesus","Peter-Jesus"]]
print (jobs(input_value))
```

```
2
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

In [28]:
```
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution
evaluate_solution(question_id=3,question_part_id='d',function=jobs,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 3, part d
Running question 3, part d, test id: 1
Input:
[['S1', 'S2', 'S3'], ['Cam', 'Ox', 'Lon'], ['S1-Cam', 'S1-Ox', 'S1-Lon', 'S2-Cam', 'S
3-Cam']]
Obtained output:
2
<<CORRECT>>
```

```
Running question 3, part d, test id: 2
Input:
[['John', 'Peter'], ['Kings', 'Trinity', 'Jesus'], ['John-Kings', 'John-Trinity', 'Jo
hn-Jesus', 'Peter-Jesus']]
Obtained output:
2
<<CORRECT>>
Running question 3, part d, test id: 3
Input:
[['S1', 'S2'], ['D1', 'D2'], ['S1-D1', 'S2-D1']]
Obtained output:
1
<<CORRECT>>
Running question 3, part d, test id: 4
Input:
[['S1', 'S2', 'S3', 'S4', 'S5'], ['D1', 'D2', 'D3', 'D4'], ['S1-D1', 'S1-D2', 'S2-D
3', 'S3-D1', 'S3-D2', 'S4-D2', 'S5-D2']]
Obtained output:
3
<<CORRECT>>
Running question 3, part d, test id: 5
Input:
[['S1', 'S2', 'S3', 'S4', 'S5', 'S6'], ['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D
8'], ['S1-D2', 'S2-D3', 'S3-D4', 'S4-D5', 'S5-D6', 'S6-D1', 'S1-D3', 'S1-D4', 'S1-D
5', 'S1-D6', 'S2-D4', 'S2-D5', 'S2-D6', 'S1-D7', 'S1-D8']]
Obtained output:
6
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
----------------------------------
```

4.

Following a longstanding tradition, Jesus College Boat Club committee hosted the Fairbairn Cup in December 2022. This competition involved rowing crews that raced to cover a stretch of the River Cam in the shortest period of time. For each rowing crew participating in the competition, the committee has gathered the crew name and the time taken to complete the course. In order to publish a table that ranks rowing crews by performance, the committee would like to sort the crews according to the time that they took to complete the course.

(a) Provide an implementation of the heapsort algorithm (function **heapsort**$(crews)$) that takes as input a list of n rowing crews and returns them sorted by course time in ascending order. Your implementation should employ a max binary heap. The input is an unsorted list of dictionaries of the form `[{"crew": <crew_name>, "time": <course_time_in_secs>}, ...]` where `<crew_name>` is a string and `<course_time_in_secs>` is an integer.

[35%]

Examples:

Input: `[{"crew_name": "Peterhouse W1", "course_time": 1038}]`
Output: `[{"crew_name": "Peterhouse W1", "course_time": 1038}]`
Explanation: The input contains only one crew, and thus it is automatically sorted.

Input: `[{"crew": "LMBC W1", "time": 1009}, {"crew": "Cantabs Women's VIII", "time": 990}]`
Output: `[{"crew": "Cantabs Women's VIII", "time": 990}, {"crew": "LMBC W1", "time": 1009}]`
Explanation: The course time of `Cantabs Women's VIII` is less than `LMBC W1`, and so they are ordered first in the output.

Constraints:

- $0 \leq n \leq 100$.

Code:

In [29]:
```
def left_child(i):
    return 2 * i + 1

def right_child(i):
    return 2 * i + 2

def parent(i):
    return math.floor( (i - 1) / 2)

def max_heapify(A, heap_size, i):
    left = left_child(i)
    right = right_child(i)
    max_i = i
    if left < heap_size and A[max_i]["time"] < A[left]["time"]:
        max_i = left
    if right < heap_size and A[max_i]["time"] < A[right]["time"]:
        max_i = right
    if max_i != i:
        A[max_i], A[i] = A[i], A[max_i]
        max_heapify(A, heap_size, max_i)

def build_max_heap(A):
    heap_size = len(A)
    for i in range(heap_size // 2 - 1, -1, -1):
        max_heapify(A, heap_size, i)

def heapsort(crews):

    # Insert code here

    build_max_heap(crews)
    heap_size = len(crews)

    while heap_size > 1:
        crews[0], crews[heap_size-1] = crews[heap_size - 1], crews[0]
        heap_size = heap_size - 1
        max_heapify(crews, heap_size, 0)

    return crews
```

Tests:

Run example test case 1:

In [30]:
```
input_value = [{"crew_name": "Peterhouse W1", "course_time": 1038}]
print(heapsort(input_value))
```

```
[{'crew_name': 'Peterhouse W1', 'course_time': 1038}]
```

Run example test case 2:

In [31]:
```
input_value = [{"crew": "LMBC W1", "time": 1009},
                {"crew": "Cantabs Women's VIII", "time": 990}]
print(heapsort(input_value))
```

```
[{'crew': "Cantabs Women's VIII", 'time': 990}, {'crew': 'LMBC W1', 'time': 1009}]
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

In [32]:
```
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution
evaluate_solution(question_id=4,question_part_id='a',function=heapsort,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

```
-----------------------------------
Evaluating question 4, part a
Running question 4, part a, test id: 1
Input:
[{'crew_name': 'Peterhouse W1', 'course_time': 1038}]
Obtained output:
[{'crew_name': 'Peterhouse W1', 'course_time': 1038}]
<<CORRECT>>
Running question 4, part a, test id: 2
Input:
[{'crew': 'LMBC W1', 'time': 1009}, {'crew': "Cantabs Women's VIII", 'time': 990}]
Obtained output:
[{'crew': "Cantabs Women's VIII", 'time': 990}, {'crew': 'LMBC W1', 'time': 1009}]
<<CORRECT>>
Running question 4, part a, test id: 3
Input:
[]
Obtained output:
[]
<<CORRECT>>
Running question 4, part a, test id: 4
Input:
[{'crew': 'Caius W1', 'time': 1016}, {'crew': 'Emma W1', 'time': 1043}, {'crew': "Can
tabs Women's VIII", 'time': 990}]
Obtained output:
[{'crew': "Cantabs Women's VIII", 'time': 990}, {'crew': 'Caius W1', 'time': 1016},
{'crew': 'Emma W1', 'time': 1043}]
<<CORRECT>>
Running question 4, part a, test id: 5
Input:
[{'crew': 'Caius W1', 'time': 1016}, {'crew': 'Emma W1', 'time': 1043}, {'crew': "Can
tabs Women's VIII", 'time': 990}, {'crew': 'Fitz W1', 'time': 1085}, {'crew': 'LMBC W
1', 'time': 1009}, {'crew': 'Peterhouse W1', 'time': 1038}, {'crew': 'Pembroke W1',
'time': 1012}, {'crew': 'Jesus W1', 'time': 1017}, {'crew': 'Newnham W1', 'time': 103
0}]
Obtained output:
[{'crew': "Cantabs Women's VIII", 'time': 990}, {'crew': 'LMBC W1', 'time': 1009},
{'crew': 'Pembroke W1', 'time': 1012}, {'crew': 'Caius W1', 'time': 1016}, {'crew':
'Jesus W1', 'time': 1017}, {'crew': 'Newnham W1', 'time': 1030}, {'crew': 'Peterhouse
W1', 'time': 1038}, {'crew': 'Emma W1', 'time': 1043}, {'crew': 'Fitz W1', 'time': 10
85}]
<<CORRECT>>
-----------------
Total correct outputs:
5 out of 5
-----------------------------------
```

(b) Derive a mathematical expression for the height of a binary heap as a function of the number of keys, n.

[15%]

The most number of nodes in a tree of height h is $2^{h+1} - 1$.
The least number of nodes in a tree of height h is $1 + 2 + \cdots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h$.
So $2^h \leq n \leq 2^{h+1} - 1$. This implies that $h \leq \log n$ and $\log(n+1) \leq h + 1$.
Since $\log n < \log(n+1)$, we have that $h \leq \log n < h + 1$. This implies that $h = \lfloor \log(n) \rfloor$

(c) Prove that construction of a max binary heap has a complexity of $\Theta(n)$, where n denotes the number of keys.

[15%]

In our **build_map_heap** function, we perform a for-loop over half of the n elements of the list, hence the complexity is $\Omega(n)$. To prove that complexity is $O(n)$, we first observe that we can decompose the cost of the **build_max_heap** function into components:

$$\sum_{h \in \text{node-heights}} \text{max. num. nodes at height } h \times \text{cost of max\_heapify at height } h \tag{7}$$

The maximum number of nodes at height h is given by: $\lceil \frac{n}{2^{h+1}} \rceil$
The cost of **max_heapify** is $O(h)$ at height h, which we can write as $C \cdot h$ for some constant C.
The height of a heap is $\lfloor \log n \rfloor$. We can thus write the total cost as:

$$\begin{align}
&\sum_{h=1}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot Ch \tag{8}\\
&\leq \sum_{h=1}^{\lfloor \log n \rfloor} \frac{n}{2^h} \cdot Ch \quad \text{since } \lceil u \rceil \leq 2u \text{ for } u \geq 1/2 \tag{9}\\
&\leq Cn \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h} \tag{10}\\
&\leq Cn \sum_{h=1}^{\infty} \frac{h}{2^h} \quad \text{since all terms in the sum are positive} \tag{11}\\
&= Cn \cdot K \quad \text{for some constant K, since the sum converges} \tag{12}
\end{align}$$

Hence **build_max_heap** is $O(n)$ in addition to $\Omega(n)$, thus it is $\Theta(n)$.

(d) Suppose that all keys in the input list are identical. What is the Big-O runtime complexity of heapsort with respect to the number of keys, n, for this case? Explain your answer with reference to your implementation of heapsort.

[10%]

The complexity is $O(n)$.
The first part of **heapsort** builds a max heap. As noted above, the construction of a max heap has complexity $O(n)$.
The second part of **heapsort** involves a for loop that calls **max_heapify** on elements of the unsorted array.
However, since all elements are equal, each call to **max_heapify** will complete in $O(1)$ time (there will be no recursion since there are no heap property violations).
Therefore, the overall complexity is $O(n)$.

(e) A *stable* sorting algorithm ensures that items that are equal maintain their input ordering in the sorted ordering. By default, heapsort is not a stable sorting algorithm. In the Fairbairns Cup, some crews completed the course in exactly the same time. Jesus College Boat Club has decided to rank crews with equal times according to the order that they appeared in the original unsorted list.

By using $O(n)$ additional space, or otherwise, provide an implementation for the **stable_heapsort**($crews$) function below that will perform a stable sort of the rowing crews by course time.

[25%]

Examples:

Input: `[{"crew": "LMBC W1", "time": 1009}, {"crew": "Cantabs Women's VIII", "time": 990}]`
Output: `[{"crew": "Cantabs Women's VIII", "time": 990}, {"crew": "LMBC W1", "time": 1009}]`
Explanation: As for the original heapsort implementation, since the course time of `Cantabs Women's VIII` is less than `LMBC W1`, they are ordered first in the output.

Input: `[{"crew": "Fitz W1", "time": 1085}, {"crew": "Christs", "time": 1085}]`
Output: `[{"crew": "Fitz W1", "time": 1085}, {"crew": "Christs", "time": 1085}]`
Explanation: Since `Fitz W1` and `Christs W1` recorded the same time, they appear in the output in the same order as the input.

Constraints:

- $0 \leq n \leq 20.$

Code:

In [33]:

```python
def is_less_than(crew1, crew2):
    if crew1["time"] == crew2["time"]:
        return crew1["initial_index"] < crew2["initial_index"]
    return crew1["time"] < crew2["time"]

def max_heapify(A, heap_size, i):
    left = left_child(i)
    right = right_child(i)
    max_i = i
    if left < heap_size and is_less_than(A[max_i], A[left]):
        max_i = left
    if right < heap_size and is_less_than(A[max_i], A[right]):
        max_i = right
    if max_i != i:
        A[max_i], A[i] = A[i], A[max_i]
        max_heapify(A, heap_size, max_i)

def stable_heapsort(crews):

    # Insert code here

    for ii, _ in enumerate(crews):
        crews[ii]["initial_index"] = ii
    build_max_heap(crews)
    heap_size = len(crews)
    while heap_size > 1:
        crews[0], crews[heap_size-1] = crews[heap_size - 1], crews[0]
        heap_size = heap_size - 1
        max_heapify(crews, heap_size, 0)
    for elem in crews:
        del elem["initial_index"]
    return crews
```

Examples:

Run example test case 1:

```
In [34]:
crews = [{"crew": "LMBC W1", "time": 1009},
         {"crew": "Cantabs Women's VIII", "time": 990}]
print(stable_heapsort(crews))
```

```
[{'crew': "Cantabs Women's VIII", 'time': 990}, {'crew': 'LMBC W1', 'time': 1009}]
```

Run example test case 2:

```
In [35]:
crews = [{"crew": "Fitz W1", "time": 1085}, {"crew": "Christs", "time": 1085}]
print(stable_heapsort(crews))
```

```
[{'crew': 'Fitz W1', 'time': 1085}, {'crew': 'Christs', 'time': 1085}]
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

```
In [36]:
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution
evaluate_solution(question_id=4,question_part_id='e',function=stable_heapsort,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

```
----------------------------------
Evaluating question 4, part e
Running question 4, part e, test id: 1
Input:
[{'crew': 'LMBC W1', 'time': 1009}, {'crew': "Cantabs Women's VIII", 'time': 990}]
Obtained output:
[{'crew': "Cantabs Women's VIII", 'time': 990}, {'crew': 'LMBC W1', 'time': 1009}]
<<CORRECT>>
Running question 4, part e, test id: 2
Input:
[{'crew': 'Fitz W1', 'time': 1085}, {'crew': 'Christs', 'time': 1085}]
Obtained output:
[{'crew': 'Fitz W1', 'time': 1085}, {'crew': 'Christs', 'time': 1085}]
<<CORRECT>>
Running question 4, part e, test id: 3
Input:
[]
Obtained output:
[]
<<CORRECT>>
Running question 4, part e, test id: 4
Input:
[{'crew': 'Christs', 'time': 1085}, {'crew': 'Fitz W1', 'time': 1085}, {'crew': 'Caiu
s W1', 'time': 1016}]
Obtained output:
[{'crew': 'Caius W1', 'time': 1016}, {'crew': 'Christs', 'time': 1085}, {'crew': 'Fit
z W1', 'time': 1085}]
<<CORRECT>>
Running question 4, part e, test id: 5
Input:
[{'crew': 'Caius W1', 'time': 1016}, {'crew': 'Emma W1', 'time': 1043}, {'crew': "Can
tabs Women's VIII", 'time': 990}, {'crew': 'Fitz W1', 'time': 1085}, {'crew': 'LMBC W
1', 'time': 1009}, {'crew': 'Peterhouse W1', 'time': 1038}, {'crew': 'Christs', 'tim
e': 1085}, {'crew': 'Pembroke W1', 'time': 1012}, {'crew': 'Jesus W1', 'time': 1017},
{'crew': 'Newnham W1', 'time': 1030}]
Obtained output:
[{'crew': "Cantabs Women's VIII", 'time': 990}, {'crew': 'LMBC W1', 'time': 1009},
{'crew': 'Pembroke W1', 'time': 1012}, {'crew': 'Caius W1', 'time': 1016}, {'crew':
'Jesus W1', 'time': 1017}, {'crew': 'Newnham W1', 'time': 1030}, {'crew': 'Peterhouse
W1', 'time': 1038}, {'crew': 'Emma W1', 'time': 1043}, {'crew': 'Fitz W1', 'time': 10
85}, {'crew': 'Christs', 'time': 1085}]
<<CORRECT>>
-----------------
Total correct outputs:
```

```
5 out of 5
----------------------------------
```

END OF PAPER