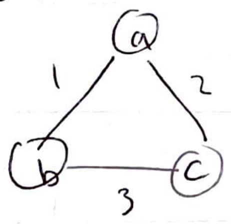
Homework # 3

(Due: 11/16/22 **)**

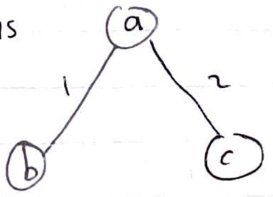
Group Number: 6

Group Members		
Name	SBU ID	% Contribution
Michael Lee	112424954	30
Christopher Lobato	114661869	70

1. a) Every MBST of G is <u>not</u> a MST.
for example give graph G:

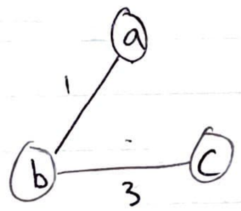


The MST is

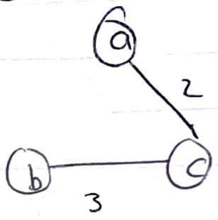


Total weight: 3

and the MBST are:



total weight: 4



total weight: 5

(The bottle neck edge is "3")

Here we can see that the MBST has a total weight greater than 3

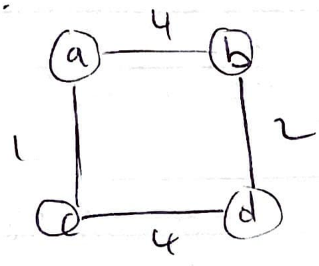
b) Every MST is a MBST.
Given an edge E of a MST ~~[illegible]~~ in which the edge holds the greatest weight of the spanning tree T, and E' ~~[illegible]~~ an edge of MBST with the bottleneck edge from spanning tree T',
1, If the edge E and E' are same value, T must be a MBST as it contains the bottleneck edge

2 ~~[illegible]~~ the edge E' cannot be greater than E as T' is a MBST

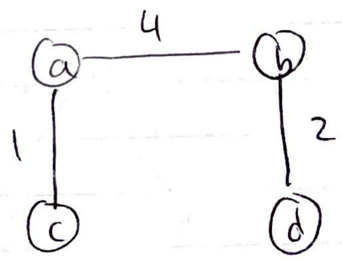
3. If the ~~[illegible]~~ edge E is of vertices (X, Y) ~~[illegible]~~ which has a greater weight than E',
if we cut the graph in two parts X, Y such that Y can be reached from y without going through x and X can't be reached from x without going through y. Then we have MBST T' with lesser weight than w(x, y).
This will cause a contradiction and only possibity is that E and E' are the same.

c) even when edge costs aren't disinct, every MBST of G is not a MST

Given graph G:

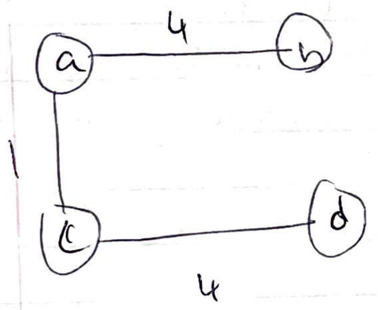


The MST is



With total weight of 7

The MBST is



with total weight of 9.

Thus not all MBST is MST.

d) For the same reasons as b), A MBST must exist within a MST that share the bottleneck edge

2a.

SortElementsKDistance(A[1…n],k)

```
1.  for i = 1 to n
2.      j = 1
3.      currentMin = A[i]
4.      currentMinIndex = i
5.      while(j+i  <= A.length and j <= k)
6.          if(akArray[i+j] < currentMin):
7.              currentMin = A[i+j]
8.              currentMinIndex = i + j
9.          j = j + 1
10.     Temp = A[i]
11.     A[i] = A[currentMinIndex]
12.     A[currentMinIndex] = temp
```

In this case the algorithm needs to be average time complexity Θ (nk). Lines 1-12 traverse and visit each of the n numbers in A therefore the outer loop is Θ(n). At each iteration in lines 5-9 we check of k subsequent elements to find the minimum value within those elements. This while loop takes at most k iterations to find the minimum. Lines 10-12 are constant time as we are only swapping the values at the indices to the appropriate position. So, because if the nested loops the algorithms average time complexity is Θ(n) * Θ(k) or Θ(nk).

b.

nlogKSort(A[1…n],k)

```
1.  i = 1
2.  while i <= n
3.      S = [] //create an Empty minimum heap/Priority Queue
4.      j = 0
5.      While( i < A.length and j <=k)
6.          insert(S.,A[i])
7.          i = i +1
8.          J = j + 1
9.      While(j >= 0)
10.         A[i-j] = Extract-Min(S)
11.         j-=1
12.     i = i + 1
```

Similarly to the previous problem we want to traverse the input array and ensure that the next k numbers after index i are sorted. In this case we can use a minimum heap and add the next k values to our heap and then sort the array by extracting the minimum till the heap is empty again. Lines 2- 12 is done in O(n) time as we traverse each element only once. The nested while loop continues moving our i pointer along and inserts the current element to the Minimum Heap till k numbers have been added to the heap(Lines 5-8). After this k numbers have been added, the minimum is extracted k times and inserted at the appropriate index. In this case the each of the n elements gets added and deleted from the heap once and we know that the worst-case time complexity for insertion and deletion is O(log(n)) where n is the number of elements. In this case we are adding k elements so for each of the n numbers the operations cost O(log(k)) + O(log(k)) = O(log(k)) time. As mentioned before this is nested within our array traversal so worst case time complexity is O(nlog(k)).

3a.

For the data structure D we would have the following operations as described in the homework document, INSERT(x), CHANGE-VALUE(p,v), DELETE(p), REPORT-SIGMA(). In addition to this D would also have an operation named REPORT-MU() which would return the median of the current set S. The data members of the data structure would be a Max Heap and Min Heap named Left and Right respectively to store and manage the numbers in S. Left will contain the numbers less than or equal to the median and Right will contain the numbers greater than or equal to the median. The data structure D also would contain an integer named Sumxisqr initialized to 0 to store the current value of $\sum_{i=1}^{n} x_i^2$, an integer named Sumxi initialized to 0 to store the current value of $\sum_{i=1}^{n} x_i$, and an integer named size to store the number of elements in the set or n.

REPORT-MU()

```
1. if (this.size % 2 != 0)
2.     if ( this.Right.size > this.Left.Size)
3.         return minimum(this.Right)
4.     else
5.         return maximum(this.Left)
6. else
7.     return (minimum(this.Right) + maximum(this.Left))/2
```

As we covered in lecture obtaining the minimum and maximum from a heap is Θ(n) time. Therefore, as the data structure grows we only need to compare the sizes of our left and right to see where the median is and these comparisons also take constant time. So, obtaining the median is a constant time operation.

b.

INSERT(x)

```
1. Pointer = &x //assign address of x to pointer
2. this.Sumxisqr = This.Sumxisqr + x^2
3. this.Sumxi = This.Sumxi + x
4. this.size = this.size + 1
5. if(Left.heapsize == 0)
6.     insert(this.Left,x)
```

```
7.  if(this.size % 2 == 0)
8.      Med = this.REPORT-MU()
9.      if ( x > med)
10.         insert(this.Right,x)
11.     else
12.         insert(this.Left, x)
13. if(this.size % 2 != 0)
14.     Med = this.REPORT-MU()
15.     if ( x > med)
16.         insert(this.Right,x)
17.     else
18.         insert(this.Left, x)
19. if( this.Left.heapsize – this.Right.heapsize > 1)
20.      insert(this.Right,Extract-Max(this.Left)
21. else if( this.Right.heapsize – this.Left.heapsize > 1)
22.      insert(this.Left,Extract-Max(this.Right)
23. return Pointer
```

Here the only operation that are not constant time is the insert operation for our right and left heaps because inserting into a heap takes O(logn) time. For this operation we only need to decide where to add our new element depending on our median and the size of our left and right. However, we need to maintain a size balance between our left and right so that our median can be easily retrieved. For this reason, if either left or right becomes too large we extract the minimum or maximum from the larger heap to the smaller one. As a result, at most for this INSERT we do O(logn) + O(logn) = O(logn) operations. It is also important to note that as we insert a new value we also need to update the value of Sumxisqr and Sumxi.

CHANGE-VALUE(p ,v)

```
1. valueToRemove = *p //get value pointed to by p
2. this.DELETE(valueToRemove)
3. this.INSERT(v)
```

Since our INSERT and DELETE operation both take O(logn) time the worst case time complexity for this operation is O(logn) + O(logn) = O(logn)

DELETE(p)

```
1. Med = this.REPORT-MU()
2. this.size = this.size - 1
3. valueToRemove = *p//get value pointed to by p
4. this.Sumxisqr = This.Sumxisqr - valueToRemove^2
5. this.Sumxi = This.Sumxi - valueToRemove
6. if (valueToRemove < Med)
7.     delete(this.Left,valueToRemove)
8. if (valueToRemove > Med)
```

```
9.      delete(this.Right,valueToRemove)
10. if( this.Left.heapsize – this.Right.heapsize > 1)
11.      insert(this.Right,Extract-Max(this.Left)
12. else if( this.Right.heapsize – this.Left.heapsize > 1)
13.      insert(this.Left,Extract-Max(this.Right)
```

Similarly, to inserting we first remove the value by finding which heap contains the value according to our median. After this we use can delete the value from the corresponding heap which is an O(logn) operation. If after deletion the difference in sizes between the heaps is too great then we can extract the value where necessary to rebalance our heap(as stated in INSERT this is also a O(logn) operation). Overall, the DELETE operation is a O(2logn) or O(logn) operation. It is also important to note that as we insert a new value we also need to update the value of Sumxisqr and Sumxi.

REPORT_SIGMA()

```
1.  Med = this.REPORT-MU()
2.  Return sqrt($\frac{(\text{this.Sumxisqr} + 2*\text{Med}*\text{this.Sumxi} + \text{this.size}*(\text{Med}\hat{\ }2))}{\text{this.size}}$)
```

As stated previously the REPORT-MU operation returns the median in constant time. Additionally, from the given equation for standard deviation we can also see the resulting equivalent expression $\sqrt{\frac{\sum_{i=1}^{n} x_i^2 - 2\cdot\mu_{med}\sum_{i=1}^{n} x_i + n(\mu_{med})^2}{n}}$. During our insertions and deletions, we have been able to keep track of $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i^2$ through Sumxi and Sumxisqr so retrieval of these values only takes constant time. As a result the calculation to find the standard deviation is O(1) time.