Assignment 3

**2) Key** **Idea**: Sort the list in O(nlgn) time then pair the i-th smallest with the i-th largest

**Algorithm**: minMax-sum

**Input:** a list L[1…n] of integers, such that n is even

**Output:** a list P[1… ] of tuples

**Begin:**

1. mergeSort(L)
2. **for** i = 1 to **do**
3. P[i] = (L[i], L[n+1-i])
4. **endfor**
5. **return** P

**end**

**Time Complexity:**

Line 1 takes O(nlgn) time to sort the list,

The for loop preforms two copies per iteration so each iteration takes O(2) time, and the for loop has n iterations thus the for loop executes in O(n) time.

Therefore, the time complexity of algorithm minMax-sum is O(nlgn)

**Correctness:**

**Lemma 2.1:** The for loop on lines 2-4 sets P[i] = (L[a], L[b]) such that L[a] and L[b] are the i-th largest and i-th smallest elements in L respectively

Proof by induction

(*Induction Basis*) When control reaches line 2 for the 1st time, the list contains no pairs and i = 1. Then control moves to line 3, where P[1] is set to (L[1],L[n]). Since the L is sorted, L[1] is the minimum and L[n] is the maximum, thus the 1st largest is paired with the 1st smallest

(*Induction hypothesis*) Suppose the lemma holds for k=m(m<n)

(Inductive Step)When control reaches line 2 for the (m+1)th time, as the execution of the for loop did not terminate during the mth iteration, m = i ≠ n/2. Control moves to line 3 where P[m+1] is set to (L[m+1], L[n+1-(m+1)]). Since the list is Sorted, L[m] is the ith smallest and L[n+1-(m+1)] is the ith largest. Combine this with the inductive hypothesis, and we have P[1…m+1] = (L[a], L[b]) such that L[a] and L[b] are the i-th largest and i-th smallest elements in L respectively.

**Lemma 2.2:** Given a list of integers L[1…n] such that n is even, a pair-partition P[1… ] with the minimal maximum pair-sum is P[k] = (L[a], L[b]) such that L[a] and L[b] are the k-th largest and k-th smallest elements in L respectively.

Direct proof

Suppose we have P[k] = (L[a], L[b]) such that L[a] and L[b] are the i-th largest and i-th smallest elements in L respectively**.**

Now, pick P[x] = (L[y],L[z]) such that P[x] is the pair with the largest pair-sum. Suppose we swap

**Theorem:** Algorithm minMax-sum correctly finds a pair-partition of L with the smallest max-sum

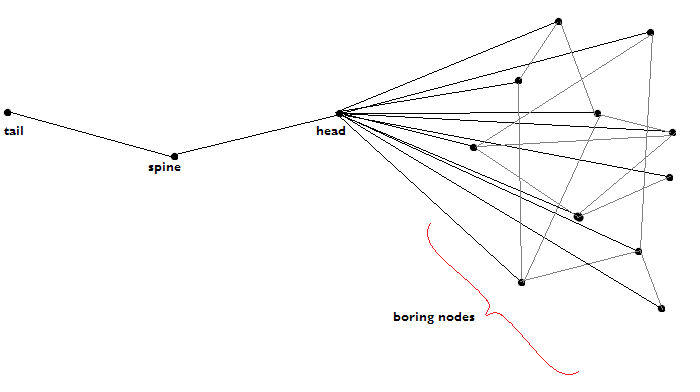
In line 1, the input List is sorted in ascending order using merge sort. Next, control moves to line 2 and enters the for loop. By Lemma 2.1 and 2.2 the for loop correctly populates P with pairs, such that the maximum pair-sum is minimal. Finally, control moves to line 5 where P is returned thus Algorithm minMax-sum returns a pair partition of L with the smallest max-sum.

■

**Greedy**

**3) Key Idea:** if we pick a random vertex, v, in A that has degree 0, n-1, or n, then we know A is not a rake graph. If v has degree n-2, 2, or 1, then we know it is a special node and we simply have to examine the 1 or 2 nodes v is connected to or the 1 node v is disconnected to verify the graph is a rake graph. Finally, if v has degree 3 to n-3, then we know v is a boring node, and that one of the nodes v is connected to the head, and one of the nodes v is disconnected to is the tail. We can find the tail in the nodes disconnected to v by determining which one is also disconnected to all of the nodes that are connected to v since the only node that is not connected to the head is the tail, and the head is one of the nodes connected to v.

Legend:



**Algorithm**: isRake

**Input:** a list A[1…n,1…n] of 0’s and 1’s, representing an undirected graph

**Output:** true if A represents a rake graph, false otherwise

**Begin:**

1. c = connected(1)
2. d = disconnected(1)
3. if(|c| = 0 or |c| = n-1 or |c| = n)
4. return false
5. else if(|c| = 1){  *//1 is boring or tail*
6. return verifyTail(1)
7. else if(|c| = 2) *//1 is boring or spine*
8. x = c[1]
9. y = c[2]
10. c1 = connected(x)
11. c2 = connected(y)
12. if(|c1| = n-2 and |c2| ≠ n-2 )
13. return = verifyHead(x)
14. else if(|c1| ≠ n-2 and |c2| = n-2 )
15. return = verifyHead(y)
16. else
17. return false
18. else if(|c| = n-2) *//1 is head*
19. return verifyHead(1)
20. else *//1 is boring*
21. con = |c|
22. dis = |c|
23. while(con ≠ 0 and dis ≠ 0)
24. if(A[ c[con], d[dis] ] = 1)
25. dis--
26. else
27. con--
28. if(dis = 0)
29. return false
30. else
31. verifyTail(d[dis])

**end**

**Algorithm**: VerifySpine

**Input:** an integer s, representing a vertex in a A

**Output:** true if A represents a rake graph, false otherwise

**Begin:**

1. c = connected(s)
2. x = c[1]
3. c = connected(x)
4. d = disconnected(x)
5. if(|c| = 2) *//s is tail*
6. if(1 = c[1])
7. c = connected(c[2])
8. else
9. c = connected(c[1])
10. else if(|c| = n-2)
11. return true
12. else
13. return false
14. else if(|c| = n-2) *//s is boring*
15. return verifyHead(x)

**end**

**Algorithm**: verifyHead

**Input:** an integer h, representing a vertex in a A

**Output:** true if A represents a rake graph, false otherwise

**Begin:**

1. d = disconnected(x)
2. c = connected(d[1])
3. if(|c| ≠ 1)
4. return false
5. else
6. c = connected(c[1])
7. if(|c| = 2)
8. return true
9. else
10. return false

**end**

**Time Complexity:**

**Correctness:**

■