EE2703 Applied Programming Lab
Programming Contest Report

Team name: kalba

Logesh K T EE21B079
Niranjan A Kartha EE21B095
Sidarth S Kumar EE21B0130

March 24, 2023

Contents

We have added documentation alongside the code that we used during the competition. We have not modified any of the actual code that we used to generate our submissions, instead our documentation is inserted in the comments.

1 Problem 1

1.1 Description

We were given a set of numbers. The end goal is to separate these numbers into two different sets such that the difference between the sum of the elements of the two lists is minimised.

Listing 1: Program used for solving problem statement 1

```
import sys

# read command line args to know which file to parse
lines = open(sys.argv[1], "r").readlines()

arr = []

# remember the order in which the numbers were in the file when we append
# the values to our list
i = 0
for line in lines[1:]:
    arr.append([int(line.strip()), i])
    i += 1

# reverse-sort the array based on the numeric value
arr.sort()
arr.reverse()

sum0 = 0
sum1 = 0

# outputs -- each element will be a 2-element array where the first element
# will be the original position of that number in the input file
outs = []

# add each number we encounter to the array with the lower sum
for val in arr:
    if sum0 <= sum1:
        sum0 += val[0]
        outs.append([val[1], 0])
    else:
        sum1 += val[0]
        outs.append([val[1], 1])

print(sum0)
print(sum1)

```

```python
diff = abs(sum0 - sum1)

# we should have a good solution at this point, but we tried to improve it
# by swapping elements from the larger to smaller array if it was
# beneficial

# we happened to make mistakes in the implementation of this algorithm, so
# it didn't result in any improvements

smaller = 0
if sum1 < sum0:
    smaller = 1

old_diff = diff

while True:
    for i in range(len(outs)):
        a = outs[i][1]
        val = arr[i][0]

        # if an element is in the larger array, and swapping it to the
        # smaller one will decrease the difference, then swap it
        if a != smaller and val < diff:
            print(f"{val} (in array {a}) is less than {diff}")

            # we had made a mistake here -- these two lines are wrong
            sum1 -= val
            sum0 += val
            # we ideally should add val to the smaller sum, but this code
            # always adds it to sum0

            # now recompute the difference
            diff = abs(sum0 - sum1)
            smaller = 0
            if sum1 < sum0:
                smaller = 1

            # this line is also wrong -- it should swap it to the smaller
            # list, instead it always swaps to list 0
            outs[i][1] = 0

            # the above code would work only if sum0 happens to be smaller
            # initially

    # if we haven't got better, stop
    if diff == old_diff:
        break

    old_diff = diff

```

```
# print the values after the improvement
print(sum0)
print(sum1)

# now put the outputs in the same order in which they appeared in the input
outs.sort()

outfile = open(sys.argv[2], "w")
outfile.write(str(len(outs)) + "\n")

for val in outs:
    outfile.write(str(val[1]))
    outfile.write("\n")
```

2 Problem 2

2.1 Description

We have a input list which has nodes and edges. We are given information about all the nodes present and edges it connects to. Each of this edges have a weight associated with them. Our aim here is to cut this list into two such that the summation of all the weights that brigde elements between these two lists is maximised

Listing 2: Program used for solving problem statment 2

```
import sys


# read command line args to know which file to parse
lines = open(sys.argv[1], "r").readlines()

# dictionary of vertices -- this keeps track of which grouping each vertex
# goes to
vertices = {}

# dictionary of vertices to neighbours
neighbours = {}

# array of edges
edges = [] # weight, node1, node2

# unnecessary variable that we forgot to remove
i = 0

for line in lines[1:]:
    split = line.strip().split()

    # initially leave the grouping of each vertex as None
    vertices[split[0]] = None
    vertices[split[1]] = None

    for i in 0, 1:
        # add each vertex to the neighbours dictionary
        if split[i] not in neighbours:
            neighbours[split[i]] = []

        # add the neighbour and the weight of the edge
        # this adds [neighbour, weight] to the array
        neighbours[split[i]].append([split[1 - i], int(split[2])])

    # add [weight, vertex1, vertex2] to the edge array
    edges.append([int(split[2]), split[0], split[1]])

print(len(edges))

```

```
# our first idea was to go down a reverse-sorted list of edges, and keep
# splitting edges down this list until we can't split anymore

# reverse sort the edges
edges.sort()
edges.reverse()

# while cutting an edge, if both vertices happen to have a grouping of
# None, this function decides where to put the vertex on the left -- we
# wanted to eventually randomize this process so we left it as a function.
# we ended up coming up with a better strategy so we didn't implement this.
def where_to_put_new_ones():
    return 0

# sum of edges that we cut (variable name is a misnomer)
sum_vert = 0

for edge in edges:
    n1 = vertices[edge[1]]
    n2 = vertices[edge[2]]

    # we look at each edge in descending order of weight, and consider the
    # vertices that this edge connects

    # if both vertices have no grouping assigned, put them in different
    # groups based on where_to_put_new_ones

    # if one of the vertices has no group and the other does, add the one
    # that doesn't have a group in the group opposite to the one that does

    # if both vertices are already given groups, don't do anything

    if n1 is None:
        if n2 is None:
            vertices[edge[1]] = where_to_put_new_ones()
            vertices[edge[2]] = 1 - vertices[edge[1]]
        else:
            vertices[edge[1]] = 1 - vertices[edge[2]]
    elif n2 is None:
        vertices[edge[2]] = 1 - vertices[edge[1]]

    # if the vertices are in different groups, add the weight of this edge
    # to our sum
    if vertices[edge[1]] != vertices[edge[2]]:
        sum_vert += edge[0]

print(sum_vert)

# this was our first round of submissions

```

```python
# we then figured out a strategy to optimize our solution
# we loop through the vertices, and judge whether swapping it to the other
# group is advantageous or not
# if we move a vertex from one group to another, the edges that are
# currently cut (we call these "active edges") will be uncut, and the ones
# that aren't cut (we call these "inactive edges") will be cut
# so we calculate the sums of active and inactive edges and compare them
# we swap groups if the inactive sum is bigger than the active sum

prev_sum = sum_vert

# we keep running this optimization until we stop getting better results
while True:
    for vert in vertices.keys():
        # sum of edges that are being cut
        active_sum = 0

        # sum of edges that aren't being cut
        inactive_sum = 0

        # calculate the sums
        for vert2 in neighbours[vert]:
            if vertices[vert2[0]] == vertices[vert]:
                inactive_sum += vert2[1]
            else:
                active_sum += vert2[1]

        # swap if it's advantageous
        if active_sum < inactive_sum:
            vertices[vert] = 1 - vertices[vert]

    sum_vert = 0

    # calculate the new cut weight
    for edge in edges:
        if vertices[edge[1]] != vertices[edge[2]]:
            sum_vert += edge[0]

    print(sum_vert)
    if sum_vert == prev_sum: # break if we've reached a saturation point
        break

    prev_sum = sum_vert

# we need to sort the vertices in ascending order
keys = list(vertices.keys())

for i in range(len(keys)):
    keys[i] = int(keys[i])

```

```python
keys.sort()

o = open(sys.argv[2], "w")

o.write(str(len(keys)) + "\n")

# calculate the number of vertices that are in group 1 for a quick sanity
# check
sum_a = 0

for key in keys:
    sum_a += vertices[str(key)]
    o.write(str(vertices[str(key)]) + "\n")

print(sum_a)
```