Batteryfix Future Developments

I never wrote a theoretical proof before, so I would appreciate some feedback. I am aware that my algorithm is really slow, and I can't even get to finish in time for $n \geq 100$, but I still want to show my observations that might be key to solve the problem. This theory is written in English (not my first language), as most resources I found are in English. I am going to try to be as formal as possible with my limited knowledge and I would be happy to explain my thoughts in a call, if something is not understandable.

First of all, I want to start with one of the more obvious observations. As given by the task, the number of poles is equal to the number of connections n times 2, which makes sense because every pole has to be connected to another pole, so the there are n connections at the end. If the number of poles was odd, one pole would always remain unconnected.

The most important part to my solution is that when you connect two poles, the one part and the other part on the left and right of the connection can be seen as two completely new riddles. This is the main observation used in the algorithm I developed. I implemented this recursively. I will come back to this later.

The final thing that stood out to me was the maximum sum possible with the given number n. As the numbers on the circle are a permutation of the list {0, 1, 2 … 2n-1}, the maximum potential achievable would be n^2 in an ideal situation, in which the first element is connected to the last element, the second to the second last and so on. This can be summarized in the following formula:

$$\sum_{i=0}^{n-1}(2n-1-2i) = n^2$$

Here the case n = 3:

$$\big(6-1-(2\times 0)\big)+\big(6-1-(2\times 1)\big)+\big(6-1-(2\times 2)\big) = n^2 = 3^2$$

My algorithm is extremely inefficient, as it looks at all the possible states, until it either finds a solution with a total potential of n^2 or until there are no more possible combinations, in which case it chooses the one with the biggest potential.

At the beginning I tried using a greedy method of always taking the local best, so the pole that can be connected to with the biggest polarity, but that wasn't able to produce a viable result, as because it just took the best option at that time, it blocked other options that produced a better result.

I don't really know how to write my algorithm in pseudocode, so I'm going to paste in the C++-code and explain it line by line:

```cpp
vector<pair<int, int> > batteryfix (vector<int> vect){
    vector<pair<int, int> > curmaxcombs;
    int curmaxpot = 0;
```

The function batteryfix takes a vector of integers as an input. This vector contains the numbers on the perimeter of the circle. It also declares the current maximum combinations to be empty and the current maximum potential to be 0.

```
    for(int i = 1; i < vect.size(); i+=2){
        vector<int> newvect1, newvect2;
        vector<pair<int, int> > newpairs1, newpairs2;
        int pot = abs(vect[0] - vect[i]);
```

It then goes into a for-loop. The current integer i is the pole the first pole is going to be connected to. It jumps in intervals of 2, and then declares the maximum combinations (newpairs1, newpairs2) and the sections (newvect1, newvect2) on the left and right of the connection.

```
       for(int j = 1; j < i; j++) newvect1.push_back(vect[j]);
       if(newvect1.size()>=2) {
            newpairs1 = batteryfix(newvect1);
            pot += newpairs1[0].first;
            newpairs1.erase(newpairs1.begin());
        }

        for(int j = i+1; j < vect.size(); j++) newvect2.push_back(vect[j]);
        if(newvect2.size()>=2) {
            newpairs2= batteryfix(newvect2);
            pot +=  newpairs2[0].first;
            newpairs2.erase(newpairs2.begin());
        }
```

Now the two vectors that we declare before are filled with the values on their respective side. If they have numbers, the function batteryfix is called recursively. It gets the best possible combinations in this segment, as well as the best possible potential back. It then adds potential to the current maximum potential declared above.

```
        if(pot > curmaxpot) {
            curmaxpot = pot;
            curmaxcombs.clear();
            curmaxcombs.push_back(make_pair(pot, 0));
            curmaxcombs.push_back(make_pair(vect[0], vect[i]));
            for(int j = 0; j < newpairs1.size(); j++) curmaxcombs.push_back(newpairs1[j]);
            for(int j = 0; j < newpairs2.size(); j++) curmaxcombs.push_back(newpairs2[j]);
            if(curmaxpot == n*n) return curmaxcombs;
        }
```

Now it knows the current potential if the pole at index 0 is connected to the pole at index i. It checks if that potential is higher than than the max potential of before and if so, it sets the current maximum potential to the current potential. It clears the list of combinations to make a new one. The first element in the list is just the potential itself for the recursive function, so it must not be considered at output. From there on the current maximum combination vector is filled with the current combination of pole 0 and i, as well as the combinations of the segments on the left and right of the connection. If the potential is equal to n^2 it means that the algorithm found one optimal solution and it is returned.

```
    }
    return curmaxcombs;
}
```

The vector of current maximum combinations is returned to be further processed.

The reason why this algorithm is correct is because it looks at all the possible combinations. Even though in the first step it only connects the first pole to another pole and not some pole to another, it produces all the possibilities, as the other poles can be connected in the recursive call of the function. It will always find the best, as it calculates the potential of all of the possibilities and takes the best/the first optimal solution.

I am not really secure when it comes to runtime and memory usage analysis. My guess goes as follows:

1. The outer loop in line 4 is executed n (= the size of the input vector divided by 2) times.
2. The inner loops on lines 8 and 15 are together executed 2n-2 times
3. The recursive function is called twice.
4. In the recursive function, the outer loop is executed times
5. This is repeated until $n \leq 2$

My guess is that the running time of my algorithm is O(n*log(n)), even though it should technically be able to run at n = 1000, but it doesn't finish in time.

As for the memory usage, I believe it's at O(n*log(n)) as it has to calculate every possible combination and save that.

I have the vague feeling that maybe this problem is solvable using a greedy algorithm, but that is just a theory.

I am really keen to hear back from you, even though, at the time of writing, it looks like I will not be invited to camp. I did my best and it was really fun. You will hear more from me in the future, as I have discovered a new hobby with competitive programming. I hope that next year I will be able to compete with the others. Thanks for the great competition!