List of Figures

Contents

I. Exercise 1

1. Problem modelling:

- Input:
  - Integer N, number of bases/sites
  - `cost[][]`: 2D array size `NxN`, where `cost[i][j]` is the cost to build base `i` on at site `j`.
- Output (Form of solution)
  - Integer minCost: the minimum energy needed to build N bases
  - N integers: `matching[i]` = the site where base `i` is built on
- Constraint:
  - Since this is intended to be a recursive research problem with worst-case complexity O(N!), N must be small. Such as `N <= 30`
  - `cost[][]` can be float or integers. In my code, I use double.
- Optimality:
  - This problem can be solved using exhaustive search with complexity O(N!). But in practice, it is much faster than O(N!) because we use branch and bound.

2. Branch and bound strategy:

- During recursive search, if the current cost is already larger than the best cost that we found, we can stop searching deeper.

```
currentMatch[baseId] = siteId;
currentCost += sortedCost[baseId][i].first; // cost[baseId][siteId];
siteBuilded[siteId] = true;
if (currentCost < bestCost) bestFirstSearch(baseId + 1); // branch and bound
currentCost -= sortedCost[baseId][i].first;
siteBuilded[siteId] = false;
```

3. Implementing depth-first-search and best-first-search:

- Both methods have been implemented in my code.
- For best-first-search, before the actual exhaustive search, we create an array `sortedCost[][]` where `sortedCost[i][j]` is an array of `pair<double,int>`. `sortedCost[i][j].first` is the cost to build base i at site `sortedCost[i][j].second`

- By doing the above step, finding the next best-node during the recursive search is trivial, we just need to loop over `sortedCost[][]`.
- The solution my code produced is:
  - Min cost = 9
  - Best matching = base1-site3, base2-site1, base3-site2