

Part I (Written component):

Due: **Friday, December 1 at 2:15 pm.**

Submission: Please hand in a paper copy of Part I at the beginning of class.

1. [6 points]

Give a divide and conquer algorithm that solves the Maximum Contiguous Subsequence Sum Problem, which is described below, in $\theta(n \log n)$ time. Note that there are many ways to solve this problem, but you will only receive credit for an algorithm that uses a divide and conquer approach.

Given a sequence of integers, a *contiguous subsequence* is a (possibly shorter) sequence in which the elements occur in the same order as in the original sequence, and the elements are contiguous in the original sequence. For example, $\langle -4, 13, -5 \rangle$ is a contiguous subsequence of $\langle -2, 11, -4, 13, -5, 2 \rangle$, but $\langle -2, 13, -5 \rangle$ is not. A *contiguous subsequence sum* is the sum of the values in a contiguous subsequence.

Maximum Contiguous Subsequence Sum Problem:

Given a sequence of (possibly negative) integers A_0, A_1, \dots, A_{N-1} , find (and identify the subsequence corresponding to) the *maximum contiguous subsequence sum* of the given sequence. For example, given the sequence $\langle -2, 11, -4, 13, -5, 2 \rangle$, the maximum contiguous subsequence sum is 20, which is the sum of elements 2 through 4, with values $\langle 11, -4, 13 \rangle$.

By definition, the maximum contiguous subsequence sum is zero if all the integers are negative.

2. [10 points]

Solve Problem 1, page 246 of the text.

Part II (Programming component):

Due: **Friday, December 8 at midnight**

Submission: Please submit the files `MatchMaker.java`, `FlowNetwork.java`, `DirectedGraph.java`, and any other files you may have written or modified that are necessary to build and run your code, by sending them as email attachments to submit301@cs.grinnell.edu.

NOTE: Please be aware that I am not allowed to extend the due date of any assignment beyond December 8. Therefore, you may not use a grace day for Part II. Given that, I have set the due time at midnight. Even so, please plan to submit well before that.

Overview:

In this assignment, you will solve a given Bipartite Matching Problem by finding a maximum flow through a corresponding flow network. The data file I have provided is intended to model “Match Day”, the day each spring when medical students are assigned to the hospitals where they will do their residency year. The real “Match Day” algorithm is similar, but a bit more complex, than the one we will use.

Posted Files:

1. I have posted a modified version of `DirectedGraph.java` that you may use if you wish. This version includes two methods, `BFS` and `shortestUnweightedPath`, that you may find helpful. The addition of these methods is the only difference between this and the previous version.

2. I have also posted a file `FixedSizeQueue.java`, which is needed by `DirectedGraph.BFS()`. If you use my `DirectedGraph`, you should probably use my `FixedSizeQueue` as well.

3. I have not posted a new version of `EdgeList`. You may use the previously posted version. You may also modify it, even though it states (with regard to a previous assignment) that you should not have to.

4. I have posted a data file named “`bipartite.txt`”, which contains input data for this assignment. Your program only needs to work on that data file. The format of the file is as follows.

- The first record contains two integers: `nStudents` and `nCities`.
- The next `nStudents` records have the format: `vertexNumber studentName`.
- The next `nCities` records have the format: `vertexNumber cityName`.
- The remaining records have the format: `studentName cityName`. Each record specifies a (directed) graph edge. The presence of the edge indicates that the student is willing to move to that city, and the hospital in that city is willing to accept the student.

YOUR TASKS:

1. Modify `DirectedGraph` such that it stores `capacity` and `flow` for each edge in the graph. (To make things easier, it is ok if each edge stores `cost` as well. That way your graph will not need to handle more than one kind of edge.)

You may also find it useful to add the following methods to `DirectedGraph`:

- `insert(v1,v2,capacity,flow)`
- `isEdge(v1,v2)`
- `getCapacity(v1,v2), setCapacity(v1,v2,capacity)`
- `getFlow(v1,v2), setFlow(v1,v2,flow)`

You may also find it useful to add a method to `EdgeList`, which returns a reference to a given `EdgeNode` or `EdgeData` object. Doing so is not required, but it may simplify the task of implementing the methods listed above.

2. Write a class named `FlowNetwork`. Your class should include the following method, which implements the Ford-Fulkerson algorithm and returns the input graph modified such that it carries a maximal flow. Of course, you may also write additional methods as needed.

```
public static DirectedGraph fordFulkerson(DirectedGraph G, int source, int target)
```

HINT: Be careful not to insert an edge into the residual graph with zero capacity. Recall that BFS does not consider cost (or capacity), and thus it could well include such an edge in an augmenting s-t path.

3. Write a class named `MatchMaker`, which will be the driver class for the assignment. It should perform the following tasks:

- Read the input data file, which specifies a bipartite graph `G`.
- Create a flow network to model the Bipartite Matching Problem on the input graph.
- Run the Ford-Fulkerson algorithm to find a maximum flow through your flow network.
- Print a nicely formatted list of edges in a maximal matching on graph `G`. You may print your output to the screen or to a data file. If you choose the latter, please print a message to the screen indicating the output file name.