Homework 3

Pham Tien Hung, Zhang Xu

Exercise 1. Use binary heap with priority queue for implementation.

Algorithm 1: Merge k sorted lists ($L[1][:], ..., L[k][:]$)

Let BH be an empty heap;
$result = []$; `// output list`
for $iin[1, ..., k]$**:**
  Add(i, $L[i][1]$); `// Initial heap`
while BH *is not empty*:
  j = Extract-min(BH) ; `// j-th sorted list`
  x = pop($L[j][1]$) ;
  Add x to $result$;
  if $L[j][:]$ is not *empty*; `// insert only if not end of the list`
  Add(j, $L[j][1]$);
return $result$;

Proof. Correctness:

The correctness follows that very time the minimum among all uninserted numbers is added to the output list of $result$.

Time Complexity:

It takes $O(k)$ to build the initial heap; for every element, it takes $O(lgk)$ for Extract-min and $O(lgk)$ to insert the next one from the same list. In total it takes $O(k + nlgk) = O(nlgk)$.

□

Exercise 2. Make use of matrix multiplication.

Algorithm 2: $Fib(n)$

if $n = 0$**:**
  return 0
elif $n = 1$**:**
  return 1
else:
  $F(0) = 0$;
  $F(1) = 1$;
  $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$;
return $A^n[F(1), F(0)]^T[0]$; `// return first element,` $F(n)$

Proof. Correctness:

With the matrix multiplication hint provided and definition of Fibonacci number, we can have the following relationship

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix} \cdot \begin{bmatrix}F(n-1)\\ F(n-2)\end{bmatrix} = \begin{bmatrix}F(n-1)+F(n-2)\\ F(n-1)\end{bmatrix} = \begin{bmatrix}F(n)\\ F(n-1)\end{bmatrix} \tag{1}$$

Let A denote $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$, then it's not difficult to obtain

$$A^n \cdot \begin{bmatrix}F(1)\\ F(0)\end{bmatrix} = \begin{bmatrix}F(n)\\ F(n-1)\end{bmatrix} \tag{2}$$

Time Complexity:

$O(\log n)$, which is the time compexity for computing matrix multiplication for 2-by-2 size matrix. □

Exercise 3. **a)** Let $S(i, j)$ denote sum of the array from i-th element to the j-th element.

$$S(i,j) = \begin{cases} 0 \text{ if } i < j \\ \sum_{k=i}^{j} L[k] \text{ otherwise} \end{cases}$$

Algorithm 3: Naive Sum(L[1..n])

```
Let S be a n-by-n zero matrix;
max = S[1, 1];    // initialize max sum to be at location [1, 1]
for i in [1..n]:
    for j in [1, ..., n]:
        if j = i:
            S[i, j] = L[j];
        elif j > i:
            S[i, j] = S[i, j - 1] + L[j];
            if S[i, j] > max:
                max = S[i, j]
return max
```

Proof. Correctness:

Intuitivey, sum is stored in a $n - by - n$ matrix S, where each entry S[i, j] is the summation from i-th element to j-th element in array L for $i \leq j$. The algorithm will return the maximum sum from the matrix and return 0 if all numbers in array L are negative.

Time complexity:

$\Theta(n^2)$. Constant computing costs within 2 nested for-loop of size n.

□

b) Dynamic programming.

Proof. Correctness:

Sum is stored in the number $maxEnding$ and max keeps track of the maximum sum. $maxEnding$ will be reset to 0 every time it goes to negative. The

Algorithm 4: DP Sum($L[1..n]$)

```
max = 0;
maxEnding = 0;
for i in [1..n]:
    maxEnding = maxEnding + L[i];
    if maxEnding < 0:
        maxEnding = 0
    elif max < maxEnding:
        max = maxEnding
return max
```

algorithm will return the maximum sum and return 0 if all numbers in array L are negative.

Time complexity:

$\Theta(n)$. Constant computing costs within one for-loop of size n.

□

Exercise 4. The problem can be interpreted as a graph problem with each currency to be a node and each exchange rate to be te weight of edge between two currencies. By doing the following transforms, we can solve the problem by applying Floyd-Warshall Algorithm for determining the existence of negative cycle.

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_k, i_1] > 1$$
$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_k, i_1]} < 1$$
$$\log(\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_k, i_1]}) < \log(1)$$
$$\log(\frac{1}{R[i_1, i_2]}) + \log(\frac{1}{R[i_2, i_3]}) + ... + \log(\frac{1}{R[i_k, i_1]}) < 0$$

Thus by replacing edge weight of $R[i, j]$ to $\log(\frac{1}{R[i,j]})$, the problem reduces to finding a negative cycle.

Time Complexity:

$O(n^3)$. The Floyd-Warshall Algorithm uses 3 nested for-loop over n nodes.

Exercise 5. The Bipartite Matching problem can be transformed to Max-Flow Formulation.

To check whether a given matching M in bipartite graph G is maximum matching, we have to construct its residual graph and check whether there is still augmenting path.

Given bipartite graph $G = \{L \cup R, E\}$ and its matching M, a directed graph $G' = \{L \cup R \cup \{s, t\}, E'\}$ is created as specified in notes. Then residual graph G_f is then constructed as following:

All edges from L are directed to R and assigned infinite capacity while those edges in M also have reverse edges with unity capacity. Source s is added and connected with each vertex in L with unit capacity edges. For those vertices in L but not in M, the edges point out from s, and for those in L and M, edges point to s. Similarly, sink t is added with unit capacity edges. For vertices in R but not in M, edges point to t, otherwise edges come out from t.

Algorithm 5: Check Max Matching($G(L \cup R, E), M$)

Construct residual graph $G_f(L \cup R \cup \{s, t\}, E')$;
if *there exists an augmenting path* P *in* G_f**:**
  return $False$
else:
  return $True$

Proof. Correctness:

According to the theorem that max cardinality of a matching in G equals to the value of max flow in G', and the Augmenting Path Theorem, we will construct the residual graph and search whether there is augmenting path. If there is, the flow is not max-flow, which indicates that the matching is not a maximum matching, the algorithm returns $False$. Otherwise returns $True$.

Time Complexity:

$O(m + n)$. BFS or DFS can be performed to check whether there is augmenting path. □