ANLY-550 Assignment 2

Yilun Zhu

Collaborate with: Siyao (Logan) Peng

1 Heap sort

1.1 Give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list.

Input are the k sorted lists are $A_1, A_2, A_3, ..., A_k$ and $A_1 + A_2 + A_3 + ... + A_k = n$
Desired output is a sorted list with length n

1. Create an output array with length n $S[1...n]$
2. Remove the first element of each sorted list $(A_1[1], A_2[1], A_3[1], ..., A_k[1])$ and build a min-heap H with length k. Building a heap takes $O(k)$ time.
3. Repeat following steps n times.
   (a) Extract the minimal element in the heap, which takes time complexity $O(\log k)$ (Because it is a min-heap, the smallest element is always the root of the heap). Store the element into the output array.
   (b) Remove the next element $A_i[j+1]$ from the corresponding sorted list and replace the heap root by it. If the corresponding list is empty, then insert an infinite to the root. The removing and insertion operation take $O(\log k)$ time.
   (c) Run min-heapify on H. The operation of complexity is $O(\log k)$.

The total running time is $O(k + n \cdot \log k) = O(n \log k)$.

1.2 Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.

Input is a k-close list with n numbers
Desired output is a sorted list with length n

1. Construct a min-heap H of length $k+1$ with the first $k+1$ element in the k-sorted list, which takes running time $O(k)$.
2. Repeat the following step $(n-k)$ times.
   (a) Remove the smallest element in the min-heap (i.e. the root).
   (b) Insert the next element (e.g. $k+2$) into the root of min-heap. The removing and inserting takes $O(\log k)$ time.
   (c) Run min-heapify H.

The total running time is $O(k + (n-k) \cdot \log k) = O(n \log k)$.

2 Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $n/3$ digits.

2.1 Integer multiplication algorithm

Input are two integers x and y
Desired output is the multiplication of x and y

1. Because each number is split into three parts,
   $x = 10^{2n/3} \cdot a + 10^{n/3} \cdot b + c$
   $y = 10^{2n/3} \cdot d + 10^{n/3} \cdot e + f$
2. $x \cdot y = 10^{4n/3} \cdot (ad) + 10^{n} \cdot (ae + bd) + 10^{2n/3} \cdot (af + be + cd) + 10^{n/3} \cdot (bf + ce) + cf$
   so that we need $ad, (ae + bd), (af + be + cd), (bf + ce)$ and cf.
3. temp1 $= (a + b) \cdot (d + e) = (ae + bd) + ad + be$
   temp2 $= (b + c) \cdot (e + f) = (bf + ce) + be + cf$
   temp3 $= (a + c) \cdot (d + f) = (af + cd) + ad + cf$
   For now, we can generate $(ae + bd), (bf + ce)$, and we still have three other parts to get $ad, (af + be + cd)$ and cf.
4. temp4 $= ad$
   temp5 $= be$
   temp6 $= cf$

ad = temp4
$ae + bd$ = temp1 - temp4 - temp5
$af + be + cd$ = temp3 - temp1 - temp6 + temp5
$bf + ce$ = temp2 - temp5 - temp6
cf = temp6

2.2 Running time

$T(n) \leq 6 \cdot T(n/3) + O(n)$
$\therefore T(n) = O(n^{\log_3 6}) = O(n^{1.63})$

2.3 Five multiplications

Two parts: $T(n) \leq 5 \cdot T(n/2) + O(n)$
$\therefore T(n) = O(n^{\log_2 5}) = O(n^{2.32})$

Three parts: $T(n) \leq 5 \cdot T(n/3) + O(n)$
$\therefore T(n) = O(n^{\log_3 5}) = O(n^{1.46})$

Therefore, three parts is more efficient.

2.4 Challenge

TODO

3 Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time.

Algorithm 1: Count inversions

```
MergeCount(L, R)
Input: Two sorted lists L and R
Output: An output sorted list S and count number of inversions
i, j ← 0;
count ← 0;
while i < length L and j < length R do
    if R[j] < L[i] then
        count ++;
        j++;
        Add R[j] to S;
    else
        i++;
        Add L[i] to S;
    end if
end while
Add rest of L and R to S;
return count, S

CountInversions(A)
Input: An array A[1...n]
Output: The sum of inversions in the n-element array
if the array A has one element then
    return 0, A
else
    Divide A into two halves L and R;
    L_c, L ← CountInversions(L);
    R_c, R ← CountInversions(R);
end if
Merge_c, S ← MergeCount(L, R);
return count = L_c + R_c + Merge_c, S
```

The pseudocode is shown in Algorithm 1. The only difference between the algorithm and merge sort is counting inversions during the procedure. Every time the element in R is smaller than the element in L, we count the inversions. When the length of a divided array is 1, there is no inversions. If the length of divided array ≥ 1, the algorithm will count inversions within the two sub-arrays and inversions between the two sub-arrays. When the algorithm stops, it counts all the inversions in the list.

The running time of the algorithm is the same as merge sort which is $O(n \log n)$, because counting runs in constant time.

4 Prove that if a graph G is undirected, then any depth first search of G will never encounter a cross edge.

Solution: Suppose there is a cross edge (v, u) in G and the DFS algorithm has already visited the vertex u but does not see the node v, thus $u < v$. According to the algorithm of DFS, after the node u is visited, all the nodes $v_1 ... v_i$ outgoing from u will be visited in the next step. Then, the edge (u, v) is part of the DFS search tree and v will become the descendant of u. When v is visited, (v, u) is a back edge. Because cross edge does not have any ancestral relation between two nodes, the definition contradicts the fact. Therefore, any DFS search tree will never encounter a cross edge in an undirected graph.

5 Design an efficient algorithm to solve the single source smallest bottleneck problem.

Suppose there is a non-negative weighted directed graph $G = (V, E)$. Finding the longest edge means we need to find out an edge with the largest weight in the shortest path. In this case, we seek the bottleneck path by using the Dijkstra's algorithm with modification. In Dijkstra, the algorithm updates the shortest distance dist[w] by seeking all the outgoing edges from $(v \to w)$. In the bottleneck algorithm as below, the only modification is that bottleneck updates the shortest distance by seeking the longest path.

Algorithm 2: Bottleneck

```
bottleneck(G = (V, E, length); s ∈ V)
v, w: vertices;
dist: array[V] of integer;
prev: array[V] of vertices;
H: priority heap of V;
H ← {s : 0};
for v ∈ V do
    dist[v] = ∞;
    prev[v] = NULL;
end for
dist[s] = 0;
while H != ∅ do
    v ← deletemin(h);
    for each (v, w) ∈ E do
        if dist[w] > max(dist[v] + length(v, w)) then
            dist[w] = max(dist[v] + length(v, w));
            prev[w] ← v;
            insert w, dist[w] to H;
        end if
    end for
end while
```

The operation of complexity is the same as Dijkstra's algorithm by using binary heap, which is $O((|E| + |V|) \cdot \log |V|)$.

6 Describe a modification of Dijkstra's algorithm that works in time $O(|E|+|V|\cdot M)$, where M is the maximum cost of any edge in the graph.

The runtime of Dijkstra's algorithm is $O((|E|+|V|)\cdot \log|V|)$. The relax function (updating the shortest distance) takes $O(|E|)$ time. To get a total runtime in $O(|E|+|V|\cdot M)$, we build a heap with priority queue of size $|V|M$.

The *deletemin* function from H is run with non-decreasing priority, because every time a new value inserted to a heap, the distance from (v,w) is added to dist$[v]$. In this case, insertion is to insert the vertex w to $H[\text{dist}[w]]$. *Deletemin* is to repeatedly delete entry itself, starting from 0, until the pointer is on a non-empty entry $H[v]$. Then the function return the vertex v. Because M is the maximum cost of any edge, the distance cannot bigger than $|V|M$, so that the algorithm will not encounter a memory overflow error.

The insertion runs in constant time. The deletemin function takes $O(|E|+|V|\cdot M)$. There are $(V-1)<M$ edges in H. When we reach to the end of H, the algorithm stops. Therefore, the total runtime is the scanning time for the length of the queue plus the number of edges relaxed (equals to the number of edges), which is $O(|E|+|V|\cdot M)$.

7 Design an efficient algorithm to detect if a risk-free currency exchange exists.

Suppose there is a non-negative directed graph $G=(V,E)$. Because there is an exchange rate $r_{i;j}$ for every two currencies c_i and c_j, $|V|=n$ and $|E|=n(n-1)$. According to the promopt, we observe that

$$r_{i_1;i_2}\cdot r_{i_2;i_3}\cdot ...\cdot r_{i_{k-1};i_k}\cdot r_{i_k;i_1}>1$$

if and only if

$$\frac{1}{r_{i_1;i_2}}\cdot\frac{1}{r_{i_2;i_3}}\cdot ...\cdot\frac{1}{r_{i_{k-1};i_k}}\cdot\frac{1}{r_{i_k;i_1}}<1$$

This equals to

$$\log(\frac{1}{r_{i_1;i_2}}\cdot\frac{1}{r_{i_2;i_3}}\cdot ...\cdot\frac{1}{r_{i_{k-1};i_k}}\cdot\frac{1}{r_{i_k;i_1}})<\log 1$$

From the equation, we get

$$\log\frac{1}{r_{i_1;i_2}}+\log\frac{1}{r_{i_2;i_3}}+...+\log\frac{1}{r_{i_{k-1};i_k}}+\log\frac{1}{r_{i_k;i_1}}<0$$

Therefore, we can define an edge

$$\begin{aligned}(c_i,c_j) &= \log\tfrac{1}{r_{i;j}}\\ &= -\log r_{i;j}\end{aligned}$$

If there is a negative weight circle in the graph G, then a risk-free currency exchange exists. We can use Bellman-Ford algorithm to solve the problem.

Algorithm 3: Find risk-free currency exchange

Input: $G = (V, E,$length$)$
Output: Whether the risk-free currency exchange exists
initialize;
dist$[s] \leftarrow 0$;
dist$[v] \leftarrow 0$ for $v \neq s$;
prev$[v] \leftarrow$ NULL for all $v \in V$;
for *i = 1 to n-1* **do**
  for *every (v, w)* $\in E$ **do**
    relax$((v, w))$
  end for
end for
for *every edge (v, w)* $\in E$ **do**
  if *dist[w] > dist[v] + length(v, w)* **then**
    return *True*
  end if
end for
return *False*

Initializing the algorithm takes $O(V)$ time. Relax takes $O(E(V-1)) = O(VE)$ time. Detecting negative cycles takes $O(V)$ time. Overall, the algorithm for detecting risk-free currency exchange takes $O(VE)$ time, which equals to the running time of Bellman-Ford algorithm.