Homework3 for EECS 340

Yu Mi,yxm319

February 27, 2018

1 Sorting Leftover Elements

Question: Suppose that you are given an array consisting of n sorted elements followed by $f(n)$ elements in an arbitrary order, where $f(n) \in O(n^{1-\epsilon})$ for some $\epsilon \in (0, 1)$. Describe a method to sort the array in $O(n)$ time.
Answer: This question describes a scenario similar to the intermediate steps of a merge sort, so that we need to solve this problem similar to the approach of a merge sort. First, we need to sort the leftover elements with merge sort, which takes $O(n^{1-\epsilon} \cdot \log n^{1-\epsilon})$ time. After that, we will need to merge the two sequence (original sorted one and the left over part) into a whole sorted sequence, which takes $O(n)$ time. To show that $O(n^{1-\epsilon} \cdot \log n^{1-\epsilon})$ takes less time than $O(n)$, we assign $g(n) = n^{1-\epsilon} \cdot \log n^{1-\epsilon}$, $h(n) = n$. Thus we have:

$$\lim_{n\to\infty} \frac{g(n)}{h(n)} = \lim_{n\to\infty} \frac{(1-\epsilon) \cdot n^{1-\epsilon} \cdot \ln n}{n \cdot \ln 2} = \lim_{n\to\infty} \frac{(1-\epsilon)^2}{\ln 2} \cdot (n^{-\epsilon} \cdot \ln n + n^{-\epsilon}) = \lim_{n\to\infty} \frac{(1-\epsilon)^2}{\ln 2} \cdot \frac{\ln n + 1}{n^{\epsilon}} = \lim_{n\to\infty} \frac{(1-\epsilon)^2}{\ln 2} \cdot \frac{1}{\epsilon n^{\epsilon}}$$

L'Hospital's rule is used in the second and forth step. The equation above will approach 0 when $n \to \infty$, so that $O(n^{1-\epsilon} \cdot \log n^{1-\epsilon})$ takes less time than $O(n)$, we can conclude that $O(n^{1-\epsilon} \cdot \log n^{1-\epsilon}) + O(n)$ is $O(n)$.

2 Theory: Sorting Algorithm Run-times

2.1 Give a tight asymptotic bound on $f(n)$

Answer: Since the total amount of permutation od n elements is $n!$, and binary encoding will use $\log(n!)$ bits to encode such permutations, our tight asymptotic bound should be

$$\log(n!) = \sum_{i=1}^{n} \log i$$

Such time bound make sense because it is always smaller than $n \log n$, which is the time of fastest sorting algorithm (at least I know).

2.2 Why doesn't the derived lower bound, from the previous part hold for non-comparison-based sorting algorithms like radix sort.

Answer: Since non-comparison-based is not based on comparisons to make a sort, they cannot be viewed as a comparison-based approach where each comparison can be treated as searching for a bit in the permutation. The lower bound derived above only stands for the comparison based sorting algorithm where we use each comparison to compose a 'bit' for the ultimate answer.

3 Post Office Placement

Question: Suppose you are the postmaster in charge of putting a new post office in a small town, where all the houses are along one street, where the new post office should go as well. Let us view this street as a line and the houses on it as a set of real numbers, $\{x_1, x_2, ..., x_n\}$, corresponding to points on this line. To make everyone in town as happy as possible, the location, p, for the new post office should minimize the sum,

$$\sum_{i=1}^{n} |p - x_i|.$$

Describe an efficient algorithm for finding the optimal location for the new post office, show that your algorithm is correct, and analyze its running time.
Answer: Suppose we have a set S which contains n elements, $s_1 \leq s_2 \leq s_3 \leq ... \leq s_n$, then we are examining:

$$\arg\min_p \sum_{s \in S} |s - p|$$

We should notice that $\frac{d|p|}{dp} = sign(x)$, when the $\arg\min_p \sum_{s \in S} |s - p|$ equals to zero, we have $p = median\{s_1, s_2, ...s_n\}$. As is similar in this question, we need to figure out the median of all the x. Thus we have the algorithm as follows, using the *QuickSort* as is described in class:

Algorithm FindOffice(S, n)
Data: A list of all addresses S, and its length n
Result: The optimal position of the office.
$S' \leftarrow QuickSort(S, n)$ ▷ Using quick sort, might cost $O(n \log n)$ time.
if $n\%2 = 0$ **then**
  return $(S'[n/2 - 1] + S'[n/2]/2)$
else
  return $S'[(n - 1)/2]$
end if

4 Sorting Sequences

Question: Suppose we are given a sequence S of n elements, each of which is an integer in the range $[0, n^2 - 1]$. Describe a simple method for sorting S in $O(n)$ time.
Hint: Think of alternate ways of viewing the elements.
Answer: The approach is based on Radix sort:

Algorithm Radix-sort(S, n)
Data: A unsorted sequence S, and its length n
Result: The sorted sequence S'
buckets $\leftarrow n$ empty lists
for $i \leftarrow 0$ to n **do**
  digit $\leftarrow$**int**$((S[n])\%n)$
  buckets[*digit*].append($S[n]$)
end for
result $\leftarrow$ empty list
for $i \leftarrow 0$ to n **do**
  for *number* in *bucket*[i] **do**
    result.append(*number*)
  end for
end for
buckets $\leftarrow n$ empty lists

for $i \leftarrow 0$ to n **do**
    $digit \leftarrow$**int**$((result[n])/n)$
    $buckets[digit]$.append$(S[n])$
end for
$result \leftarrow$ empty list
for $i \leftarrow 0$ to n **do**
    for *number* in $bucket[i]$ **do**
        $result.append(number)$
    end for
end for

5 Median From Two Lists

Question: Suppose you are given two sorted lists, A and B, of n elements each all of which are distinct. Describe a method that runs in $O(\log n)$ time for finding the median in the set defined by the union of A and B. Note that merging or concatenating the arrays would take $O(n)$ time.
Answer: The approach is based on recursive and dividing:

Algorithm GetMedian(A, B, n)
Data: Two sorted list A and B, and their length are both n
Result: The median of the union of A and B.
if $n == 1$ **then**
    return $(A[0] + B[0])/2$
end if
if $n\%2 == 0$ **then** ▷ Get the median of A and B
    $m1 \leftarrow (A[n/2] + A[n/2-1])/2$
    $m2 \leftarrow (B[n/2] + B[n/2-1])/2$
else
    $m1 \leftarrow A[n/2]$
    $m2 \leftarrow B[n/2]$
end if
if $m1 == m2$ **then** ▷ If the median are equal, return either $m1$ or $m2$
    return m1
end if
if (**then**$m1 < m2$)
    if $n\%2 == 0$ **then**
        return $GetMedian(A + n/2 - 1, B, n - n/2 + 1)$ ▷ The add operation to a list name is same as C language which means to shift the pointer to the next $n/2 - 1$ address.
    else
        return $GetMedian(A + n/2, B, n - n/2)$
    end if
else
    if $n\%2 == 0$ **then**
        return $GetMedian(B + n/2 - 1, A, n - n/2 + 1)$
    else
        return $GetMedian(B + n/2, A, n - n/2)$
    end if
end if

To achieve the $O(\log n)$ time complexity, we need to try divide and recursively solve the problem. First we need to calculate the medians of A and B, and then compare them. If $m1$ and $m2$ are equal, we are simply done. Otherwise, if $m1 < m2$, we can conclude that the median is in the two list, one is from $m1$

to the last element of A and the other one is from the first element of B to $m2$. It is the similar process when $m1 > m2$. When the two arrays have a size of 1, we can simply calculate their average as the result.

6 Warm-up: Graphs

6.1 R-7.1

Question: Suppose we have a social network with members A, B, C, D, E, F, and G, and the set of friendship ties,

$$\{(A, B), (B, C), (C, A), (D, E), (F, G)\}.$$

Where are the connected components?
Answer: The connection relationship can be represented by the Fig.1.

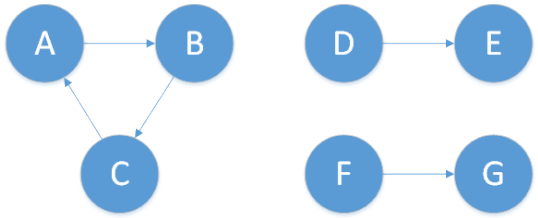

Figure 1: Relation graph

From the figure we can conclude that the connect components are (A, B, C), (D, E) and (F, G).

6.2 R-13.2

Question: Let G be a simple connected graph with n vertices and m edges. Explain why $O(\log m)$ is $O(\log n)$.
Answer: When we have m vertices, the maximum number of edges could be $m < n(n-1) < n^2$, which means $O(\log m) < O(\log n^2) = O(2 \log n) = O(\log n)$. To sum up, $O(\log m)$ is $O(\log n)$.

6.3 Given the graph

6.3.1 Draw the graph where all nodes are annotated with the order that they are visited in a depth-first search from A.

Answer: The result is shown in Fig. 2.

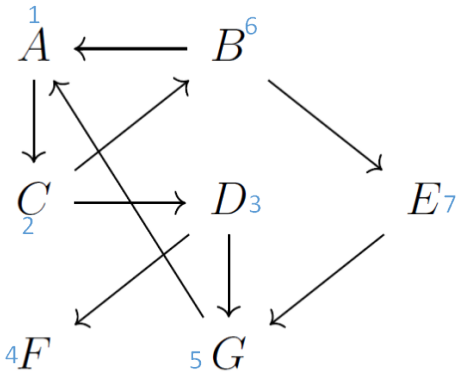

Figure 2: Annotated graph

6.3.2 Draw a representation of the tree generated by the DFS annotated over the previous part.

Answer: The result is shown in Fig. 3.

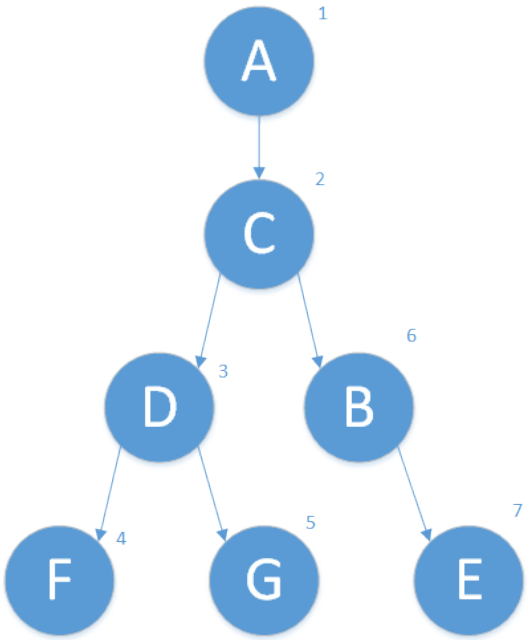

Figure 3: Tree graph

7 Application: Chess

Answer: To solve this problem in $O(n^2)$ time, we need to use the idea of Union-Find set to reduce the union time.

Algorithm Chess($Table, n$)
Data: The chess table $Table$, which is a two dimension table, and its one-dimensional size n.
Result: A list of ordered pairs (i, j) representing the positions of queens in the largest standoff.
$Forest \leftarrow$ empty union find forest
for $i \leftarrow 0$ to n **do**
  for $j \leftarrow 0$ to n **do**
    if $Table[i][j]$ **then**
      $Forest.Makeset(i, j)$ ▷ Make union-find set for every queen.
    end if
  end for
end for
for $i \leftarrow 0$ to n **do**
  $Temp \leftarrow (-1, -1)$
  for $j \leftarrow 0$ to n **do**
    if $Table[i][j]$ **then**
      if $Temp \neq (-1, -1)$ **then**
        $Union((i, j), Temp)$ ▷ Merge the queens in the same row
      end if
      $Temp \leftarrow (i, j)$
    end if

```
    end for
end for
for i ← 0 to n do
    Temp ← (−1, −1)
    for j ← 0 to n do
        if Table[j][i] then
            if Temp ≠ (−1, −1) then
                Union((j, i), Temp)                ▷ Merge the queens in the same column
            end if
            Temp ← (j, i)
        end if
    end for
end for
for i ← 0 to 2n − 1 do
    Temp ← (−1, −1)
    for j ← 0 to n − |n − i| do
        if i ≤ n then
            x ← n − i
        else
            x ← 0
        end if
        y ← j
        if Table[x][y] then
            if Temp ≠ (−1, −1) then
                Union((x, y), Temp)                ▷ Merge the queens in the same diagonal
            end if
            Temp ← (x, y)
        end if
    end for
end for
for i ← 0 to 2n − 1 do
    Temp ← (−1, −1)
    for j ← 0 to n − |n − i| do
        if i ≤ n then
            x ← n − i
        else
            x ← 0
        end if
        y ← j
        if Table[y][x] then
            if Temp ≠ (−1, −1) then
                Union((y, x), Temp)                ▷ Merge the queens in the same (other)diagonal
            end if
            Temp ← (y, x)
        end if
    end for
end for
MaxSize ← 0
MaxPos ← 0
for i ← 0 to size(Forest) do
```

```
    if Forest[i].size() > MaxSize then
        MaxSize ← Forest[i].size()
        MaxPos ← i
    end if
end for
return Forest[MaxPos]
```

Since the union operation only cost $O(1)$ time, we can conclude that the 'For' loops in the above pseudo-code all cost $O(n^2)$ time.

8 Application: A World of Voxels

8.1 Describe a method to preprocess the array in $O(n\log(n))$ time.

Answer: To preprocess the array and make such data structure in $O(n)$ size with the preprocess costing $O(n\log(b))$ time, we can use a tree structure, where a father node can have 6 leaves, representing the up, down, front, behind, left and right block of a giving block. To create a tree, we can use both **DFS** and **BFS** to build the tree, they can both reveal the linked relationship but using **BFS** would better simulate the real metal block neighboring relationship, since in the real world, the longer the link is, the higher electrical resistance would be.

```
Algorithm BuildTree(Matrix, m, n, p)
Data: A matrix representing the metal blocks, and its three dimensional size m, n, p
Result: The tree represent of the metal blocks.
Root ← (−1, −1, −1)
Roots ← empty list                                          ▷ the list of roots
for i ← 0 to m do
    for j ← 0 to n do
        for k ← 0 to p do
            Visited[i][j][k] ← Matrix[i][j][k]              ▷ Copy to make a 'visited' matrix.
            if Matrix[i][j][k] then
                Root ← (i, j, k)                            ▷ Find the root node
                Roots.append(Root)
                Root.father ← NULL
                RecursiveBuildTree(Martix, Visit, Root[0], Root[1], Root[2], m, n, p)
            end if
        end for
    end for
end for

def RecursiveBuildTree(Matrix, Visit, i, j, k, m, n, p)
Root ← (i, j, k)
if i + 1 < m then
    if Matrix[i + 1][j][k] and Visit[i + 1][j][k] then      ▷ first dimension plus one
        Root.link[0] ← (i + 1, j, k)
        Visit[i + 1][j][k] ← 0
        Node[i + 1][j][k].father ← Root
    end if
end if
if j + 1 < n then
    if Matrix[i][j + 1][k] and Visit[i][j + 1][k] then      ▷ second dimension plus one
        Root.link[1] ← (i, j + 1, k)
```

```
        Visit[i][j+1][k] ← 0
        Node[i][j+1][k].father ← Root
    end if
end if
if k + 1 < p then
    if Matrix[i][j][k+1] and Visit[i][j][k+1] then          ▷ third dimension plus one
        Root.link[2] ← (i, j, k+1)
        Visit[i][j][k+1] ← 0
        Node[i][j][k+1].father ← Root
    end if
end if
if i − 1 ≥ 0 then
    if Matrix[i−1][j][k] and Visit[i−1][j][k] then          ▷ first dimension minus one
        Root.link[3] ← (i−1, j, k)
        Visit[i−1][j][k] ← 0
        Node[i−1][j][k].father ← Root
    end if
end if
if j − 1 ≥ 0 then
    if Matrix[i][j−1][k] and Visit[i][j−1][k] then          ▷ second dimension minus one
        Root.link[4] ← (i, j−1, k)
        Visit[i][j−1][k] ← 0
        Node[i][j−1][k].father ← Root
    end if
end if
if k − 1 ≥ 0 then
    if Matrix[i][j][k−1] and Visit[i][j][k−1] then          ▷ third dimension minus one
        Root.link[5] ← (i, j, k−1)
        Visit[i][j][k−1] ← 0
        Node[i][j][k−1].father ← Root
    end if
end if
for i ← 0 to 5 do
    if Root.link[i] ≠ NULL then
        RecursiveBuildTree(Matrix, Visit, Root.link[i][0], Root.link[i][1], Root.link[i][2], m, n, p)
    end if
end for
```

Since in this algorithm, we are using a tree to represent the linking relationship. $Node[i][j][k]$ here represents the node with position (i, j, k). To find whether two given block is connected, we can run the algorithm which can find the root node:

```
Algorithm IsConnected(i1, j1, k1, i2, j2, k2)
Data: The position of two blocks (i1, j1, k1) and (i2, j2, k2)
Result: Whether the two blocks are connected.
return FindRoot(i1, j1, k1) = FindRoot(i2, j2, k2)      ▷ Return whether the root of this two blocks are same

def FindRoot(i, j, k)
if Node[i][j][k].father ≠ NULL then
    return FindRoot(i, j, k)
else
```

```
    return (i, j, k)
  end if
```

Since the depth of the tree is at most $O(\log n)$, this query function can at most reach the $O(\log n)$ time complexity.

8.2 Describe how to support additions, what is the asymptotic runtime of an addition?

Answer: To add a new block in this tree, we need first to find whether there is any block in other tree that is a neighbor of this tree and, may be merge two trees into one. The algorithm is shown as follow:

```
Algorithm AddBlock(i, j, k, m, n, p)
LinkedTree ← empty list
if i + 1 < m then
    if FindRoot(i + 1, j, k) ≠ (i, j, k) then                ▷ Is in a tree
        LinkedTree.append(i, j, k)
        Node[i + 1, j, k].linked[3] ← (i, j, k)
        Node[i, j, k].father ← (i + 1, j, k)
    end if
end if
if j + 1 < n then
    if FindRoot(i, j + 1, k) ≠ (i, j, k) then                ▷ Is in a tree
        LinkedTree.append(i, j, k)
        Node[i, j, k].linked[4] ← (i, j, k)
        Node[i, j, k].father ← (i, j + 1, k)
    end if
end if
if k + 1 < p then
    if FindRoot(i, j, k + 1) ≠ (i, j, k) then                ▷ Is in a tree
        LinkedTree.append(i, j, k)
        Node[i, j, k + 1].linked[5] ← (i, j, k)
        Node[i, j, k].father ← (i, j, k + 1)
    end if
end if
if i − 1 ≥ 0 then
    if FindRoot(i − 1, j, k) ≠ (i, j, k) then                ▷ Is in a tree
        LinkedTree.append(i, j, k)
        Node[i − 1, j, k].linked[0] ← (i, j, k)
        Node[i, j, k].father ← (i − 1, j, k)
    end if
end if
if j − 1 ≥ 0 then
    if FindRoot(i, j − 1, k) ≠ (i, j, k) then                ▷ Is in a tree
        LinkedTree.append(i, j, k)
        Node[i, j − 1, k].linked[1] ← (i, j, k)
        Node[i, j, k].father ← (i, j − 1, k)
    end if
end if
if k − 1 ≥ 0 then
    if FindRoot(i, j, k − 1) ≠ (i, j, k) then                ▷ Is in a tree
        LinkedTree.append(i, j, k)
        Node[i, j, k − 1].linked[2] ← (i, j, k)
```

$Node[i,j,k].father \leftarrow (i,j,k-1)$
end if
end if
if $LinkedTree.size() > 1$ **then**
for $i \leftarrow 1$ to $LinketTree.size()$ **do**
$Node[i,j,k].linked[5+i] = LinkedTree[i]$
$LinkedTree[i].father \leftarrow (i,j,k)$ ▷ NOTE: This might destroy the original relationship of neighboring, but will keep the union-find feature.
end for
end if

As is shown in the algorithm above, the time of an addition should be $O(1)$. But if we want to keep the neighboring relationship in the tree, we may need to transverse the tree and cost more time. However, it is not required in this question to keep such relationship so the time costing can be reduced.

8.3 Describe a way to support deletions of blocks, also describe any practical problems with the runtime of your method.

Answer: To remove a block, we can simply remove the block from the tree and make other adjacent blocks be the root of a new tree if the new tree will never be connected to the original tree. However, as the tree we described above may not represent all the neighboring relationship, we will have to infer such relationship using the index of all the remaining blocks in the forest. Such approach will achieve a time costing of $O(n^2)$ because it may need to compare every pair of blocks to verify if they are adjacent. Such time costing is much larger than the building tree approach as described in question 8.1. Thus my approach of removing a block would be remove the block out of the original matrix $Matrix$ and then rebuild the tree.

9 Graph on a Grid

9.1 Show the adjacency list representation of the following graph.

Answer: The adjacency list is shown as follow:
$(2,2) \rightarrow (1,2), (2,1), NULL$
$(2,1) \rightarrow (1,1), (2,0), NULL$
$(1,2) \rightarrow (1,1), (0,2), NULL$
$(1,1) \rightarrow (1,0), (0,1), NULL$
$(2,0) \rightarrow (1,0), NULL$
$(0,2) \rightarrow (0,1), NULL$
$(0,1) \rightarrow (0,0), NULL$
$(1,0) \rightarrow (0,0), NULL$
$(0,0) \rightarrow NULL$

9.2 Show two different depth-first traversals from $(2,2)$ on the graph in part a.

Answer: First DFS:$(2,2), (1,2), (0,2), (0,1), (0,0), (1,1), (1,0), (2,1), (2,0)$.
Second DFS:$(2,2), (2,1), (2,0), (1,0), (0,0), (1,1), (0,1), (1,2), (0,2)$.

10 EECS 454 only

10.1 C-17.4

Question: Show that every language L in **P** is polynomial-time reducible to the language $M = 5$, that is, the language that simply asks whether the binary encoding of the input is equal to 5.

10.2 C-17.5

Question: Show how to construct a Boolean circuit C such that, if we create variables only for the inputs of C and then try to build a Boolean formula that is equivalent to C, then we will create a formula exponentially larger than an encoding of C.
Hint: Use recursion to repeat subexpressions in a way that doubles their size each time they are used.