

CS 141 Homework Assignment #2

Set Jan. 28, 2008

Due Feb. 11, 2008

(before the midterm)

Problem 1. [15+5 pts] [Levitin, pp. 106-107] Question 10. (The same exercise is also stated in Question 9 on page 106 of the 1st edition, although the specification is a bit brief.)

Instead of writing a real computer program, for this exercise you need only write an informal algorithm in pseudocode. Let us assume that the input is an square matrix $M(n, n)$ of letters. Also assume that you can check if a string x of letters is a legal English word using a Boolean function $Word(x)$, which returns true if x is legal and false otherwise in constant (i.e. $\Theta(1)$) time.

Analyze the time complexity of your algorithm using the Theta notation.

Answer:

Algorithm WORDFIND($M(n, n)$)

```

1: var  $x$ : string
2: for  $i \leftarrow 1$  to  $n$  do
3:   for  $j \leftarrow 1$  to  $n$  do
4:     for  $k \leftarrow 1$  to  $n$  do
5:        $x \leftarrow M[i][j \uparrow (j + k - 1) \bmod n]$  // read "→"
6:       if  $Word(x) = \text{true}$  then
7:         print  $x$ 
8:        $x \leftarrow M[i][j \downarrow (j - k + 1 + n) \bmod n]$  // read "←"
9:       if  $Word(x) = \text{true}$  then
10:        print  $x$ 
11:       $x \leftarrow M[i \uparrow (i + k - 1) \bmod n][j]$  // read "↓"
12:      if  $Word(x) = \text{true}$  then
13:        print  $x$ 
14:       $x \leftarrow M[i \downarrow (i - k + 1 + n) \bmod n][j]$  // read "↑"
15:      if  $Word(x) = \text{true}$  then
16:        print  $x$ 
17:       $x \leftarrow M[i \downarrow (i - k + 1 + n) \bmod n][j \uparrow (j + k - 1) \bmod n]$  // read "↗"
18:      if  $Word(x) = \text{true}$  then
19:        print  $x$ 
20:       $x \leftarrow M[i \uparrow (i + k - 1) \bmod n][j \uparrow (j + k - 1) \bmod n]$  // read "↘"
21:      if  $Word(x) = \text{true}$  then
22:        print  $x$ 
23:       $x \leftarrow M[i \downarrow (i - k + 1 + n) \bmod n][j \downarrow (j - k + 1 + n) \bmod n]$  // read "↖"

```

```

24:         if Word(x) = true then
25:             print x
26:         x ← M[i ↑ (i + k - 1) mod n][j ↓ (j - k + 1 + n) mod n] // read “↖”
27:         if Word(x) = true then
28:             print x

```

[15pts]

For each square in the table, we may find words starting with the letter in the square in eight different directions. For each direction, there are up to n words to check. There are totally n^2 squares in the table, and the Boolean function $Word(x)$ can check whether a word is a legal English word or not in $\Theta(1)$ time, so the time complexity of the algorithm above is $n^2 \cdot 8n \cdot \Theta(1) = \Theta(n^3)$. [5pts]

Problem 2. [15 pts] [Levitin, pp. 119] Question 6. (This is the Question 7 on p. 118 in the 1st edition of Levitin.)

Let us assume that the input set A has n integers and is given to you as an array $A[1 \dots n]$. Recall the bit vector representation of a set that you may have learned in CS 14 (also reviewed on p. 37 of Levitin). You may represent each subset of A as a bit vector of length n . You may also enumerate all the bit vectors by treating a bit vector as a binary number and repeatedly adding 1 to it.

What is the time complexity of your algorithm, assuming that incrementing a bit vector by 1 takes $\Theta(n)$ time in the worst case?

Answer:

Algorithm PARTITION($A[1 \dots n]$)

```

1: var sum, subsum: integer
2: var bitVec: Vector(n)
3: sum ← 0
4: for i ← 1 to n do
5:     sum ← sum + A[i] // Calculate the sum of all elements
6: for i ← 0 to  $2^{n-1}$  do
7:     subsum ← 0 // Enumerate  $2^{n-1}$  subsets
8:     for j ← 1 to n do
9:         bitVec[j] ← extract the jth bit from i // Represent the subset using a bit vector
10:        if bitVec[j] = 1 then
11:            subsum ← subsum + A[j] // Calculate the sum of the subset
12:        if subsum = sum/2 then
13:            break // A partition is found

```

```

14: if subsum = sum/2 then
15:     for j ← 1 to n do
16:         if bitVec[j] = 1 then
17:             print A[j]                // Print out one subset of the partition
18:         return true
19: else
20:     return false

```

[10pts]

For a set of n elements, there are 2^n subsets. We only need to calculate the sum of 2^{n-1} subsets to check whether there exists a partition of two subsets with equal sum or not. First, we calculate the sum of all the elements in the set in $\Theta(n)$ time. Then we enumerate all the 2^{n-1} subsets. In the inner *for* loop, we use $\Theta(n)$ time to represent a subset as a bit vector and calculate the sum of that subset. If we found there is a partition, then we print out one of the subsets in $\Theta(n)$ time. So the time complexity of the algorithm above is $\Theta(n) + 2^{n-1} \cdot \Theta(n) + \Theta(n) = \Theta(n \cdot 2^n)$. [5pts]

Problem 3. [20 pts] Let $A[1 \dots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion in A .

- a. List the five inversions in the array $\langle 2, 3, 8, 6, 1 \rangle$.
- b. What array with elements from the set $\{1, 2, \dots, n\}$ has the most inversions? How many does it have?
- c. Write an algorithm that determines the number of inversions in any permutation of n elements in $O(n \log n)$ time in the worst case. (Hint: Modify Mergesort.)

Answer:

- a. Five inversions are as follows: [5pts]

(1, 5) (2, 5) (3, 4) (3, 5) (4, 5)

- b. The array has the most inversions is as follows:

$\langle n, n - 1, n - 2, \dots, 3, 2, 1 \rangle$

There is an inversion between any two elements in the array above. So the total number of inversions is $\binom{n}{2} = n(n - 1)/2$. [5pts]

- c.

Algorithm INVERSIONS($A[0 \dots n - 1]$)

```

1: if  $n = 1$  then
2:   return 0
3: else if  $n > 1$  then
4:   copy  $A[0 \dots \lfloor n/2 \rfloor - 1]$  to  $B[0 \dots \lfloor n/2 \rfloor - 1]$ 
5:   copy  $A[\lfloor n/2 \rfloor \dots n - 1]$  to  $C[0 \dots \lfloor n/2 \rfloor - 1]$ 
6:   return  $\text{INVERSIONS}(B[0 \dots \lfloor n/2 \rfloor - 1]) + \text{INVERSIONS}(C[0 \dots \lfloor n/2 \rfloor - 1]) + \text{MERGE}(B, C, A)$ 

```

Algorithm $\text{MERGE}(B[0 \dots p - 1], C[0 \dots q - 1], A[0 \dots p + q - 1])$

```

1:  $i \leftarrow 0, j \leftarrow 0, k \leftarrow 0$ 
2:  $\text{numOfInv} \leftarrow 0$ 
3: while  $i < p$  and  $j < q$  do
4:   if  $B[i] \leq C[j]$  then
5:      $A[k] \leftarrow B[i], i \leftarrow i + 1$ 
6:   else
7:      $A[k] \leftarrow C[j], j \leftarrow j + 1$ 
8:      $\text{numOfInv} \leftarrow \text{numOfInv} + p - i$ 
9:     // Element in  $C$  is smaller, and it fixes  $(p - i)$  inversions
10:     $k \leftarrow k + 1$ 
11: if  $i = p$  then
12:   copy  $C[j \dots q - 1]$  to  $A[k \dots p + q - 1]$ 
13: else
14:   copy  $B[i \dots p - 1]$  to  $A[k \dots p + q - 1]$ 
15: return  $\text{numOfInv}$ 

```

The function `INVERSIONS` divides the array into two halves, and the total number of inversions is equal to the sum of the inversions of the two sub arrays plus the inversions when merging them. [8pts]

Since the above algorithm just did little modification to the original *MergeSort* algorithm, it is not hard to get the recurrence relation for the time complexity: $T(n) = 2T(n/2) + \Theta(n)$. So $T(n) \in \Theta(n \log n)$. [2pts]

Problem 4. [10 pts] [Levitin, pp. 171-172] Question 8(b). (Question 8(b) on p. 169 in the 1st edition.)

How should the input graph be represented and what is the time complexity of your algorithm?

Answer:

Algorithm BIPARTITE(G)

```
1: // Implements a breadth-first search traversal of a given graph
2: // Input: Graph  $G = \langle V, E \rangle$ 
3: // Output: Graph  $G$  is a bipartite or not
4: mark each vertex in  $V$  with 0 as a mark of being “unvisited”
5: for all  $v \in V$  do
6:     if  $v$  is marked with 0 then
7:         mark  $v$  with color red
8:         if BFS( $v$ ) = false then
9:             return false
10: return true
```

Algorithm BFS(v)

```
1: // visits all the unvisited vertices connected to vertex  $v$  by a path
2: while the queue is not empty do
3:     for each vertex  $w$  in  $V$  adjacent to the front vertex  $v_0$  do
4:         if  $w$  is marked with 0 then
5:             mark  $w$  with the other color different from the color of  $v_0$ 
6:             add  $w$  to the queue
7:         else if  $w$  has already been marked and has the same color as  $v_0$  then
8:             return false
9:     remove the front vertex  $v_0$  from the queue
10: return true
```

In the algorithm above, we try to mark the vertices in graph G with two different colors by bread-first search traversal. If we can mark all the vertices so that every edge in G has its vertices colored in different colors, then G is a bipartite graph; otherwise, G is not a bipartite graph. [8pts]

Input graph can be represented by adjacency matrix, or adjacency list, with time complexity of $\Theta(|V|^2)$ and $\Theta(|V| + |E|)$ respectively. But the $\Theta(|V| + |E|)$ time solution is preferred. [2pts]

Problem 5. [10 pts] Let $G = (V, E)$ be a DAG, where $|V| = n$ and $|E| = m$. A vertex v is called a sink of G if there is a path from every vertex in G to v . A DAG may or may not have a sink in general. Design an $O(n + m)$ time algorithm to check if G has a sink, assuming that G is represented as adjacency lists.

Hint: A DAG may have at most one sink. Use topological sorting or a simple modification of the source-removal algorithm on page 175. (pp. 172-173 in the 1st edition.)

Answer:

Algorithm SINK(G)

```
1: Find all the vertices with no outgoing edges in  $G$  by checking each of the adjacency
   lists
2: if there is only one such vertex then
3:   return true
4: else
5:   return false
```

Since G is a DAG, there is at least one vertex with no outgoing edges in G . If there are more than one such vertices, then none of them are sinks since there are no edges between any two of them. So there is a sink in G if and only if there is only one such vertex that has no outgoing edges in G . [8pts]

If we use adjacency lists to represent a digraph, then the time complexity of the algorithm above is $O(|V| + |E|)$, which is $O(m + n)$. [2pts]

Problem 6. [Optional, 10 bonus pts] [Levitin, p. 135] Question 11. (Question 10 on p. 133 in the 1st edition.)

You do not have to give a formal analysis (or proof) of the $\Theta(n \log n)$ time complexity of your algorithm, but you need give some reasonable argument. For example, you may compare the steps in your algorithm with those in *QuickSort*. You may also assume that the each given bolt uniquely matches a nut (i.e., its corresponding nut).

Answer:

Basic idea:

Use a pivot nut x to partition the *bolts* into two sets (i.e. set *smallBolts* consisting of bolts that are too small for x and set *largeBolts* consisting of bolts that are too large for x) as in *QuickSort*, as well as to identify the matching bolt y . Then use y to partition the *nuts* into two respective sets, *smallNuts* and *largeNuts*. Similar to *QuickSort*, the original instance can be solved by two recursive calls to $(smallNuts, smallBolts)$ and $(largeNuts, largeBolts)$.

Algorithm NUTBOLTMATCH($nuts, bolts$)

```
1: if  $|nuts| > 0$  and  $|bolts| > 0$  then
2:    $smallNuts, largeNuts, smallBolts, largeBolts \leftarrow \emptyset$ 
3:   select the first element in  $nuts$  as a pivot, call it  $x$ 
4:   // Use  $x$  to partition  $bolts$  into two sets
```

```

5:   for all  $b_i \in bolts$  do
6:       if  $b_i > x$  then
7:           add  $b_i$  to largeBolts
8:       else if  $b_i < x$  then
9:           add  $b_i$  to smallBolts
10:      else
11:          found a matching bolt, define it as  $y$            // Found a match ( $x, y$ )
12:      // Use  $y$  to partition nuts into two sets
13:      for all  $n_i \in nuts$  do
14:          if  $n_i > y$  then
15:              add  $n_i$  to largeNuts
16:          else if  $n_i < y$  then
17:              add  $n_i$  to smallNuts
18:      NUTBOLTMATCH(smallNuts, smallBolts)
19:      NUTBOLTMATCH(largeNuts, largeBolts)
20:      return ( $x, y$ )                                     [8pts]

```

Since the algorithm above simulates two *QuickSort* for *nuts* and *bolts* simultaneously, so the average-case time complexity is $\Theta(n \log n)$. [2pts]