Halmstad University

School of Information Technology

October 2015

Introduction to Algorithms, Data Structures, and Problem Solving (DA4002)

ALL QUESTIONS must be answered

Number of questions: 6

Total mark: 100

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

Course convener: Amin Farjudian

Office: F321
Office ext.: 7846
Mobile: 072 917 7191

Question 1 (15)

Assume that H is an array-backed binary *min* heap, i. e. the elements on every path from the root to a leaf are in non-decreasing order, and the root always holds a smallest element. The following is the contents of the array representing the heap H:

1	10	5	11	17	20	30

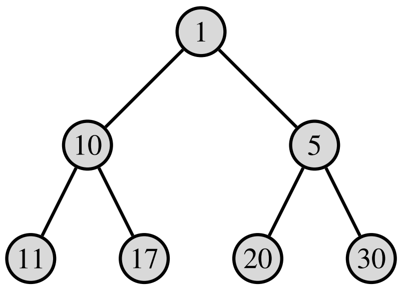
(a) Draw the binary tree representation of the heap H. (5)

(b) Next, we extract (remove) the root of the heap. As a result, we need to reheapify after the removal of the root. Go through the step-by-step procedure of reheapifying, explain each step briefly, and draw the binary tree representation of the intermediate results. (5)

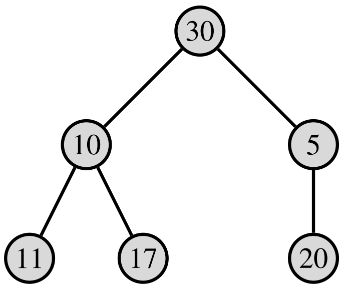
(c) Now we take the heap that we obtained in part (b), and add a new element with value `11` to the heap. Once again, go through the step-by-step procedure of insertion and reheapifying. Explain each step briefly, and draw the binary tree representation of the intermediate results. (5)

Solution

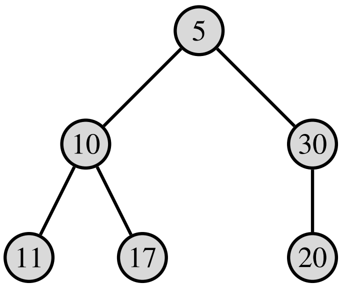
(a) Here is the binary tree representation of H:



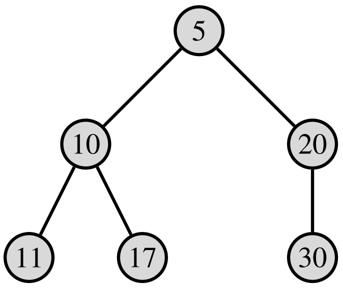
(b) We first put the right-most element of the last level in the root:



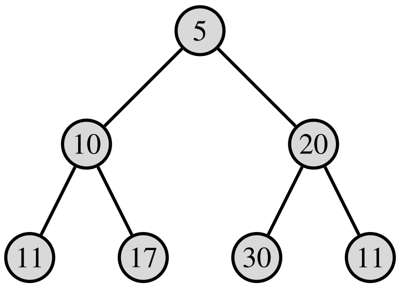
Then we choose 5 *(not* 10*) to swap with* 30 *as* $30 - 5 > 30 - 10$*:*



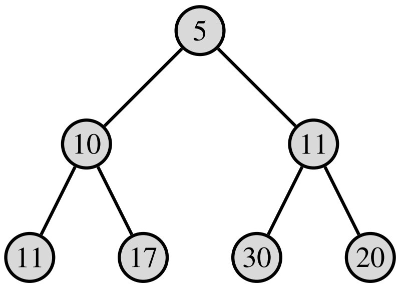
Finally as 30 *has only one child, we swap and then the reheapification is complete:*



(c) We begin by adding 11 *to the first empty cell of the last row:*



Then we let 11 *bubble up. As it is smaller than* 20*, then we need to swap:*



Now we compare 11 *and* 5*, but as* $11 > 5$*, then heap is in order and the computation stops.*

Question 2 **(20)**

Consider a graph $G = (V, E)$ in which $V = \{v_1, v_2, v_3, v_4\}$, i. e. G has four vertices. The adjacency matrix of G is as follows:

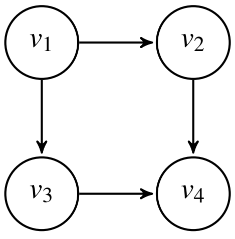
$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

(a) Is G directed or undirected? Is it possible to answer this question by only inspecting A? (4)

(b) Draw the graph G based on the given adjacency matrix. (4)

(c) Write the *pseudo-code* of an algorithm which takes the adjacency matrix $A = [a_{ij}]_{1 \le i,j \le n}$ of a graph $G = (V, E)$ with n vertices $\{v_1, v_2, \ldots, v_n\}$, together with two indices $1 \le i, j \le n$ given as integers, and return `true` if there exists a path from v_i to v_j, and return `false` otherwise. (12)

For instance, given the following graph, and indices 1 and 4, the algorithm must return `true` as there is a path from v_1 to v_4, i. e. $v_1 \to v_2 \to v_4$, whereas given the indices 4 and 1, the algorithm must return `false` as there are no paths going from v_4 to v_1.

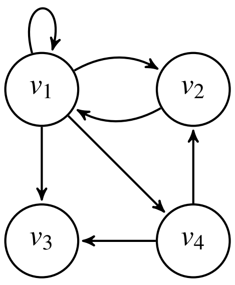


Note: You are required to write a pseudo-code, not a C code. So you do not need to take the particular way in which C handles memory into account. Nonetheless, in all other respects, your pseudo-code must be clear and complete so that a skilled C programmer can translate your instructions into a C code in a straightforward way.

Solution

(a) G is directed. This can be deduced by realizing that A is not symmetric.

(b) Here is one way of drawing G:



(c) A simple algorithm can be devised by matrix multiplication. For every $k \in \mathbb{N}$, if $[c_{i,j}]_{n\times n} = A^k$, then we have:

$$c_{i,j} = \begin{cases} 1 & \text{if there exists a path of length } k \text{ from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Thus, to see whether there exists a path between v_i and v_j, we must inspect the (i, j) element of A, A^2, $\ldots$, A^M, $\ldots$. But at some point we need to stop, otherwise we could face non-termination. Luckily, it suffices to continue until $A^{|E|}$, i. e. up to the number of edges, at most.

Algorithm 1 existsPathBetween(A, i, j)

Input: An adjacency matrix A of size $n \times n$ and two integers $1 \le i, j \le n$
Output: `true` if there is a path between v_i and v_j, `false` otherwise

```
let ne ← numOfEdges(A)
let As ← $I_{n\times n}$ {As initialized to the identity matrix, will hold powers of A}

for $k = 1$ to $ne$ do
   let As ← As*A {$k$-th power of A}
   if As[i,j]>0 then
      return true {It is crucial to return as soon as a path is found}
   end if
end for
return false {No path exists}
```

There is no need for explicit formulation of an algorithm for `numOfEdges`, as it is just going through all the elements of `A` and counting the `1`s. But here is one anyway:

Algorithm 2 numOfEdges(A)

Input: An adjacency matrix A of size $n \times n$
Output: Number of edges in the corresponding graph.

```
let n ← number of rows of A
let sum ← 0
for $i = 1$ to $n$ do
   for $j = 1$ to $n$ do
      let sum ← sum + A[i,j]
   end for
end for
return sum
```

Question 3 **(15)**

Let us first have a reminder of the master theorem, which states that for any solution of the recurrence relation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \quad (a \geq 1, b > 1)$$

we have:

$$\begin{cases} T(n) \in \Theta(n^{\log_b a}) & \text{if } f(n) \in O(n^c) & c < \log_b a \\ T(n) \in \Theta(n^c \log^{k+1} n) & \text{if } f(n) \in \Theta(n^c \log^k n) & c = \log_b a, k \geq 0 \\ T(n) \in \Theta(f(n)) & \text{if } f(n) \in \Theta(n^c) \text{ and reg. cond.} & c > \log_b a \end{cases}$$

in which the *regularity condition* of the last clause is formulated as: there exists a $k \in (0, 1)$ such that for sufficiently large n we have:

$$af(n/b) \leq kf(n)$$

(a) Let $T_M(n)$ be the number of unit steps taken by a merge sort algorithm on input data of size n. Write down a recurrence relation for $T_M(n)$. (4)

(b) Is it possible to work out the Big-Oh class of T_M using the master theorem? If yes, then you should present a calculation. If not, then you should first argue why the master theorem does not apply to this recurrence relation, and then work out the Big-Oh class by directly manipulating the recurrence relation. (4)

(c) Assume that the time complexity $T(n)$ of an algorithm satisfies:

$$\begin{cases} T(0) = 1 \\ T(n) = 3T(n-1) + 2^{n+1} \qquad (n \geq 1) \end{cases}$$

Is it possible to use the master theorem to obtain the Big-Oh class of T? Again, if yes, then you should present a calculation. If not, then you should first argue why the master theorem does not apply to this recurrence relation, and then work out the Big-Oh class by directly manipulating the recurrence relation. (7)

Solution

(a) $T_M(1) = 1, T_M(n) = 2T_M(n/2) + \alpha n$, *where* α *is a constant.*

(b) Yes we can use the master theorem. The second clause would work by taking:

$$a = 2, b = 2, c = 1 = \log_a b, k = 0$$

Therefore $T(n) \in \Theta(n \log n)$.

(c) No, we cannot use the master theorem simply because there is no constant value that can play the role of b. Therefore we have to work out the solution directly:

$$\begin{aligned}
T(n) &= 3T(n-1) + 2^{n+1} \\
&= 3(3T(n-2) + 2^n) + 2^{n+1} = 3^2T(n-2) + 3^1 2^n + 3^0 2^{n+1} \\
&= \ldots \\
&= 3^k T(n-k) + 3^{k-1} 2^{n+1-(k-1)} + 3^{k-2} 2^{n+1-(k-2)} + \cdots + 3^0 2^{n+1} \\
&= \ldots \\
&= 3^n + \Sigma_{i=0}^{n-1} 3^i 2^{n+1-i} \\
&= 3^n + 2\Sigma_{i=0}^{n-1} 3^i 2^{n-i} \\
&\leq 3^n + 2\Sigma_{i=0}^{n-1} 3^n \\
&\leq 3^n + 2n3^n \\
&= (1+2n)3^n \\
&\in O(n3^n)
\end{aligned}$$

Question 4 **(15)**

Write a C implementation of the method

```
int subsetSumZero( int* set, int length);
```

which takes an array of integer values together with its length, and returns the **number of non-empty subsets of this array that sum to** 0. Remember that in a set, there is no order on the elements. For instance, the sets $\{-1, 1\}$ and $\{1, -1\}$ are regarded equal. Thus, if you consider the array

```
int set[] = { -1, 0, 1, 2};
```

then `subsetSumZero( set, 4)` should return `3`. The three subsets that sum to zero are as follows:

$$\{0\}, \{-1, 1\}, \{-1, 0, 1\}$$

whereas for the array

```
int set[] = { -6, -5, -1, 1, 2, 3};
```

the value returned by `subsetSumZero( set, 6)` should be `4` where the corresponding `4` subsets are

$$\{-1, 1\}, \{-5, 2, 3\}, \{-6, 1, 2, 3\}, \{-5, -1, 1, 2, 3\}$$

The marks for this question are distributed as follows:

Explanation of the algorithm: You **must** first explain the *main idea* behind your algorithm *in plain words* before including the C implementation. Your explanation should be succinct and clear, and not be longer than 100 words. (4)

Commenting the C Code: In your C code, you must include clear comments that make your code understandable. (4)

Correctness: Trivially, the correctness of your solution is also a contributing factor. (4)

Elegance: Try to write an elegant code. This includes the design of the algorithm, its efficiency, how it handles memory, and how many statements are included in your code. The more elegant your solution is the higher the mark that you get. (3)

Solution *One way of writing this method is as follows:*

```
int subsetSumZero( int* set, int length){
  if (length < 1) {
    return 0;
  }
  else {
    return subsetSumZero_help( set, 0, length - 1, 0);
  }
}

int subsetSumZero_help( int* set, int l, int r, int pivot){
  int result = 0;
  int adjustment = ( pivot == set[ l]) ? 1 : 0;
  if ( l == r) {
    result = adjustment;
  }
  else {
    result =
      subsetSumZero_help( set, l + 1, r, pivot - set[ l]) +
      subsetSumZero_help( set, l + 1, r, pivot) +
      adjustment;
  }
  return result;
}
```

Marking: Minor typos and syntactic errors should be ignored for marking.

Question 5 **(15)**

Consider the function f defined by the following recursive relation:

$$f(m,n) = \begin{cases} 1 & \text{if } m \leq 0 \text{ or } n \leq 0 \\ f(m-1,n) + f(m,n-1) & \text{otherwise} \end{cases}$$

We implement this function by directly using the recursive definition as follows:

```
int frec( int m, int n){
  return ( m<=0 || n<=0) ? 1 : frec( m-1, n) + frec( m, n-1);
}
```

(a) Write an iterative implementation of the function f with the following signature: (5)

```
int fiter( int m, int n){
   // YOUR TASK
}
```

(b) Let $T_i(m,n)$ be the time complexity of the iterative implementation you have come up with in part (a). What is the Big-Oh complexity of T_i? (3)

(c) Let $T_r(m,n)$ be the number of essential operations required to compute $f(m,n)$ using `frec`. What is the Big-Oh complexity of T_r? (7)

Hint: First write a recurrence relation for T_r. Then work out a few values of $T_r(m,n)$ and compare them with the values of $f(m,n)$. Each $T_r(m,n)$ is related to $f(m,n)$ through a simple relationship. The function f in turn is a famous and commonly used function in mathematics.

Solution

(a) Here is one implementation:

```
// iterative version
int fiter( int m, int n){

  if( m <= 0 || n <= 0){
    return 1;
  }

  int table[m+1][n+1];
  int i, j;

  // first fill in the top row and leftmost column
  for( j = 0; j < n+1; j++){
    table[ 0][ j] = 1;
  }
  for( i = 0; i < m+1; i++){
    table[ i][ 0] = 1;
  }

  // now fill in the table
  for( i = 1; i < m+1; i++){
    for( j = 1; j < n+1; j++){
      table[ i][ j] = table[ i-1][ j] + table[ i][ j-1];
    }
  }

  return table[m][n];

}
```

(b) $T_i(m,n) \in O(mn)$

(c) We have

$$\begin{cases} T_r(0,n) = T_r(m,0) = 1 \\ T_r(m,n) = 1 + T_r(m-1,n) + T_r(m,n-1) \end{cases}$$

If you draw the table for a few values you will see that:

$$T_r(m,n) = 2\binom{m+n}{m} - 1$$

Therefore $T_r(m,n) \in O(\frac{(m+n)!}{m!n!})$. *It can be analyzed further using Stirling's formula but it is not needed for the purposes of this exam.*

Question 6 **(20)**

Imagine you have been given a binary search tree with the following node structure and some synonyms defined through `typedef`:

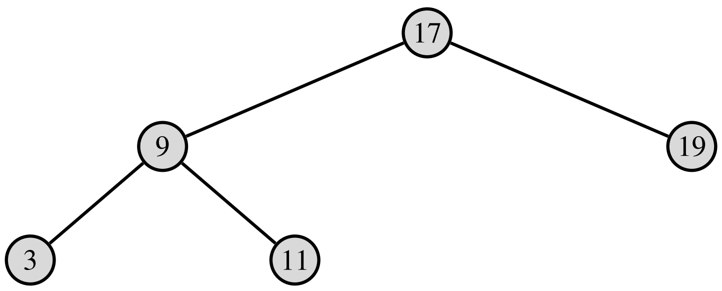
```
struct treeNode {
   struct treeNode *leftPtr; // pointer to left subtree
   int data; // node value
   struct treeNode *rightPtr; // pointer to right subtree
};
typedef struct treeNode TreeNode; // synonym for struct treeNode
typedef TreeNode *TreeNodePtr; // synonym for TreeNode*
```

Write the implementation of a function

```
int *bstToArray( TreeNodePtr rootPtr, unsigned int *sizeA);
```

which, given the pointer `rootPtr` to the root of a binary search tree, returns an array holding all of the data in the tree *in order*. Furthermore, it should write the number of nodes of the tree into the location pointed to by `sizeA`.

For instance, given the following tree:



through a call such as

```
unsigned int sizeA;
int *a = bstToArray( rootPtr, &sizeA);
```

we should get `a = {3,9,11,17,19}` and `sizeA = 5`.

Note: Please take the following into consideration:

- You may need to write a couple of helper functions to achieve this task.

- The whole task can be accomplished in about 30 lines of code (without comments and blank lines of course). If your program is significantly longer than 30 lines, you will lose marks accordingly. For every extra 5 lines, you will lose 5 marks.
- Explain what your code does through succinct and informative comments. ***If your code is not understandable, you will lose all the marks for this question.*** Your comment lines will not be counted towards your 30 line limit.

Solution *This is one way of achieving the task:*

```
int *bstToArray( TreeNodePtr rootPtr, unsigned int *sizeA){
  int *a = malloc( sizeBST( rootPtr) * sizeof( int));
  if( a == NULL){
    fprintf( stderr, "%s\n", "bstToArray: malloc failed");
  }
  else{
    bstToArrayHelper( rootPtr, a, sizeA);
  }
  return a;
}
void bstToArrayHelper( TreeNodePtr rootPtr, int *a,
                       unsigned int *sizeOfA){
  if( rootPtr == NULL){
    *sizeOfA = 0;
    return;
  }
  unsigned int sizeLeft;
  unsigned int sizeRight;
  bstToArrayHelper( rootPtr->leftPtr, a, &sizeLeft);
  a[ sizeLeft] = rootPtr->data;
  bstToArrayHelper( rootPtr->rightPtr,
                    &( a[ sizeLeft + 1]), &sizeRight);
  *sizeOfA = sizeLeft + 1 + sizeRight;
}
unsigned int sizeBST( TreeNodePtr rootPtr){
  if( rootPtr == NULL){
    return 0;
  }
  return 1 + sizeBST( rootPtr->leftPtr) +
    sizeBST( rootPtr->rightPtr);
}
```

Marking: *The following should be taken into account:*

- *How succinct and efficient the code is.*
- *A code that is too long will lose 5 marks for every extra 5 lines.*
- *No memory leak.*
- *Sound use of helper functions.*