INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR		
		Stamp / Signature of the Invigilator

EXAMINATION (Mid Semester)									**SEMESTER (Spring)**				
Roll Number									**Section**		**Name**		
Subject Number	C	S	2	1	0	0	3	**Subject Name**	*Algorithms – I*				
Department / Center of the Student										**Additional sheets**			

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items or any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging information with others or any such attempt will be treated as **'unfair means'**. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Marks Obtained											

Marks obtained (in words)	**Signature of the Examiner**	**Signature of the Scrutineer**

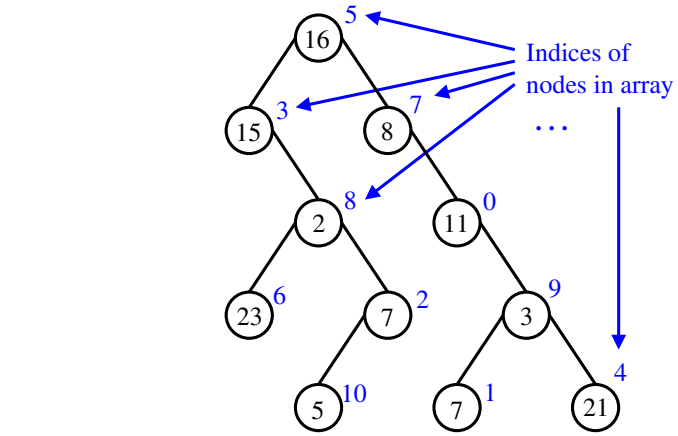
CS21003 Algorithms–I, Spring 2017–2018

Mid-Semester Test

22–February–2018 V4, NR121/122/221/322/422, 02:00pm–04:00pm Maximum marks: 40

Write your answers in the question paper itself. Be brief and precise. Answer all questions.
If you use any algorithm/result/formula covered in the class, just mention it, do not elaborate.

1. Complete binary trees (heaps) have contiguous representations where the indices of the child nodes of a node at a given index can be obtained by easily computable formulas (the children of the node at index i are at indices $2i$ and $2i+1$ under one-based indexing). The parent representations of arbitrary rooted trees are also contiguous. General binary trees can also have contiguous representations. Now, the indices of the child nodes cannot be computed by well-specified formulas, so we need to store the two indices against every node. The adjacent figure demonstrates this idea. The tree is specified by an array of triples $(key, lidx, ridx)$, where key is the key value stored in a node, and $lidx$ and $ridx$ are the indices of the array cells storing its left and right children. Do not assume any specific storage ordering (like level-by-level) of the node triples in the array. Externally remember the array index storing the root.



0	1	2	3	4	**5**	6	7	8	9	10
11	7	7	15	21	16	23	8	2	3	5
–	–	10	–	–	3	–	–	6	1	–
9	–	–	8	–	7	–	0	2	4	–

In the above example, the root is stored at index 5. Its key is 16, and its two children are at indices 3 and 7. The node at index 0 stores key 11, has no left child, and has a right child stored at index 9. The node stored at index 1 is a leaf. The non-existence of a child is specified by a special marker (shown as – in the figure, and can be the invalid index -1). In this representation, we do not store the index of the parent of a node.

(a) Propose an algorithm that, given a binary tree represented as an array of triples and the index of the root, computes and returns the height of the tree. Mention the running time of your algorithm. **(4+1)**

Solution The following recursive algorithm computes the height. The outermost call should pass the index of the root as `nodeidx`.

```
int height ( triple T[], int nodeidx )
{
    int lht, rht;

    if (nodeidx == INVALID) return -1;

    lht = height(T, T[nodeidx].lidx);
    rht = height(T, T[nodeidx].ridx);

    return 1 + max(lht,rht);
}
```

The running time is $O(n)$, where n is the number of nodes in the tree.

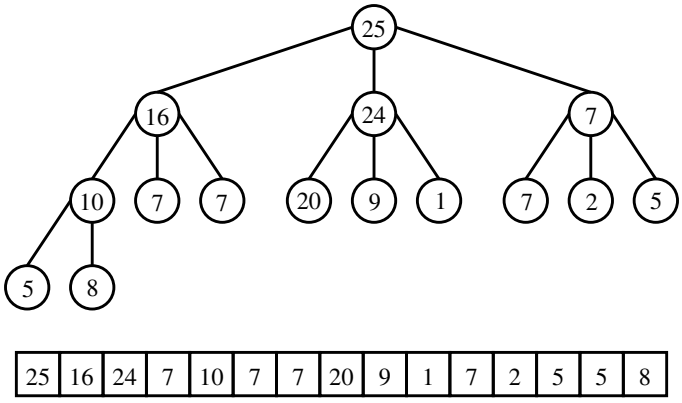
(b) The ancestors of a node v in a binary tree are defined as: $\text{Ancestor}^{(0)}(v) = v$, and $\text{Ancestor}^{(k)}(v) = \text{Parent}(\text{Ancestor}^{(k-1)}(v))$ for $k \geqslant 1$. The parent of the root is assumed to be the root itself. Write a function that, given a binary tree represented as an array of triples, the index of the root, and the index of an arbitrary node v in the array of triples, prints the keys of all the ancestors of v in the tree. For example, in the tree given on the earlier page, the ancestors of the node at index 2 have the key values $7, 2, 15, 16$. Mention the running time of your algorithm. **(4+1)**

Solution We make a traversal of the tree. Only the ancestors of v print their key values. The return value (Boolean) indicates whether the current node is an ancestor of v. The outermost call should pass the index of the root as **`nodeidx`**.

```
int printancestors ( triple T[], int vidx, int nodeidx )
{
    int lresp, rresp;

    if (nodeidx == INVALID) return 0;

    if (nodeidx == vidx) {
        print(T[nodeidx].key);
        return 1;
    }

    lresp = printancestors(T, vidx, T[nodeidx].lidx);
    rresp = printancestors(T, vidx, T[nodeidx].ridx);

    if ((lresp == 0) && (rresp == 0)) return 0;

    print(T[nodeidx].key);
    return 1;
}
```

The running time is again $O(n)$ (in the worst case), where n is the number of nodes in the tree.

2. Heaps introduced in the class are called binary heaps. *Ternary heaps* are realized by complete ternary trees where each node can have three children: left, mid, and right. All levels except perhaps the last must be full. Leaves appear only in the last two levels. Also, the leaves at the last level must be to the left of the empty positions. Finally, the key stored at any node must not be smaller than the keys stored in its children. The adjacent figure illustrates a ternary heap and its contiguous representation.



Propose an algorithm to convert an array H of integers of size n to a ternary heap in the contiguous representation. More precisely, write the **`heapify`** and **`makeheap`** functions for a ternary heap. **(7+3)**

Solution Let us use zero-based indexing. The three children of the node at index i are at indices $3i+1$, $3i+2$ and $3i+3$. The parent of the node at index i is at index $\lfloor (i-1)/3 \rfloor$.

```
void heapify ( int H[], int n, int i )
{
   int lidx, midx, ridx, maxidx, t;

   while (1) {
      lidx = 3*i+1; midx = 3*i+2; ridx = 3*i+3;
      if (lidx >= n) break;
      if (midx >= n) maxidx = lidx;
      else if (ridx >= n) maxidx = (H[lidx] >= H[midx]) ? lidx : midx;
      else {
         maxidx = lidx;
         if (H[midx] > H[maxidx]) maxidx = midx;
         if (H[ridx] > H[maxidx]) maxidx = ridx;
      }
      if (H[i] >= H[maxidx]) break;
      t = H[i]; H[i] = H[maxidx]; H[maxidx] = t;
      i = maxidx;
   }
}


void makeheap ( int H[], int n)
{
   int i;

   for (i=(n-2)/3; i>=0; --i)
      heapify(H,n,i);
}
```

3. The Fibonacci numbers are defined as $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geqslant 2$. Let $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$.

(a) Prove that $\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = M^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$ for all $n \geqslant 0$. (**Remark:** M^0 is the identity matrix $I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.) **(5)**

Solution [By induction on n] The result is obvious for $n = 0$. So assume that the result holds for some $n \geqslant 0$. But then, $M^{n+1} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = M \times M^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n + F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix}$.

(b) Propose an $O(\log n)$-time divide-and-conquer algorithm for computing F_n using the formula of Part (a). Justify that your algorithm runs in $O(\log n)$ time. **(3+2)**

Solution First, notice that two 2×2 matrices can be multiplied in $O(1)$ time. Moreover, a 2×2 matrix and a two-dimensional vector can be multiplied again in $O(1)$ time. Therefore it suffices to compute M^n (or M^{n-1} which also contains F_n if $n > 0$) in $O(\log n)$ time. The following recursive algorithm computes M^n.

1. If $n = 0$, return I_2.
2. If $n = 1$, return M.
3. Let $m = \lfloor n/2 \rfloor$.
4. Recursively compute $N = M^m$, and set $P = N^2$.
5. If $n = 2m + 1$, set $P = PM$.
6. Return P.

We have $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$. Since $\log_2 1 = 0$, the master theorem of divide-and-conquer recurrences implies $T(n) = \Theta(\log n)$.

4. You have n jobs, each requiring one unit of time. At any point of time, you can do only one job. If you start a job, you must finish it before you start another job. For each i, there is a deadline $t_i \in \{0,1,2,\ldots,n-1\}$ associated with the i-th job. If the job is started at a time $\leqslant t_i$, you get a profit of p_i (a positive integer). If the job is started at a time $> t_i$, you get no profit at all. This means that if you cannot schedule a job on or before its start deadline, you do not schedule this job at all. Notice that the deadlines t_i need not be distinct from one another. Your task is to select and schedule a subset of the jobs in such a way that your total profit is maximized. Propose an $O(n \log n)$-time greedy algorithm to solve this problem. Analyze that your algorithm has the given running time. **(8+2)**

Solution We first sort the deadline array in descending order. We use a max-priority queue Q of maximum size n. The priority queue stores (t_i, p_i), and its heap ordering is based on the profits (the second component). The steps of the algorithm are given below.

1. Sort the array of deadlines by an optimal sorting algorithm.
2. Initialize a max-priority queue Q to empty.
3. For $time = n-1, n-2, \ldots, 2, 1, 0$, repeat:
   (a) March through the sorted array of deadlines, and insert in Q all (t_i, p_i) with $t_i = time$.
   (b) If Q is non-empty, schedule the job at the root of Q, and then delete this job (deletemax) from Q.

The correctness of this greedy approach is straightforward to prove.

Step 1 can be done in $O(n \log n)$ time, for instance, if we use merge sort or heap sort. Step 2 takes $O(1)$ time. Since T is sorted in the descending order, the march in Step 3(a) is only a forward march, and n pairs are inserted in Q in a total of $O(n \log n)$ time. Finally, each run of Step 3(b) requires $O(\log n)$ time (time for deletemax). So the overall running time of this algorithm is $O(n \log n)$.

Remarks: The earliest-deadline-first greedy strategy need not work. Consider the following example with $n = 3$. Job 0 has the earliest deadline. If we schedule it at $time = 0$, we can schedule either Job 1 or Job 2 at $time = 1$ (and no job at $time = 2$). This gives a maximum profit of $2 + 6 = 8$. A better strategy is to schedule Jobs 1 and 2 at $time = 0, 1$ (in any order), leading to a profit of $6 + 5 = 11$.

i	0	1	2
t_i	0	1	1
p_i	2	6	5

The maximum-profit-first greedy strategy too may fail to work as illustrated by the following example with $n = 3$. This strategy advocates scheduling Job 0 at $time = 0$. But then, we can schedule neither Job 1 nor Job 2. Thus the profit is only 5. A better strategy is to schedule Job 1 at $time = 0$ and then Job 0 at $time = 1$ or $time = 2$. The profit is now $5 + 3 = 8$,

i	0	1	2
t_i	2	0	0
p_i	5	3	2

FOR LEFTOVER ANSWERS OR ROUGH WORK