CS11001/11002/13002 Programming and Data Structures, Spring 2007–2008

End-semester examination

Maximum marks: 100 | April 23, 2008 (F/N) | Total time: 3 hours

---

Answer ***any five*** questions. Start answering each question in a fresh page.
Try to answer all parts of a question together. If you cannot, supply continuation pointers.

1. (a) Assuming that in a recursive quick sort, each recursive call partitions the input array into two roughly equal halves, give the recurrence relation depicting the time complexity; hence, obtain a close form of the time complexity. **(10)**

(b) Write a *recursive* C function **`findMed`** which returns the median of a one-dimensional array of integers, that is, the element which is larger than half of the elements and smaller than half of the elements. Suppose $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$ is the sorted version of the input array A. If $n = 2k + 1$, then the element a_{k+1} is the median of A. On the other hand, if $n = 2k$, we take a_k as the median of A. Your function should use a partitioning technique as in quick sort. Use the following function prototype:

```
    int findMed ( int A[], int startidx, int endidx, const int medidx );
```

Here the second and the third parameters are the start and end indices of the current partition; the fourth parameter is the index of the median in the sorted version of A and is kept constant across the calls. **(10)**

2. (a) Write a C function which takes a square matrix A and its order n as two parameters and prints each of the n left diagonals of A starting from the top and ending at the bottom. The diagonals are treated in a wrap-around fashion as exemplified by the 4×4 matrix: $\begin{pmatrix} \times & + & - & \bullet \\ \bullet & \times & + & - \\ - & \bullet & \times & + \\ + & - & \bullet & \times \end{pmatrix}$. **(10)**

(b) The *tensor product* $A \otimes B$ of two $n \times n$ matrices is defined as $A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1}B & a_{n2}B & \cdots & a_{nn}B \end{pmatrix}$, where a_{ij} is the (i, j)-th element of A and $a_{ij}B$ stands for the $n \times n$ block obtained by the scalar multiplication of the matrix B by the element a_{ij}. For example, if A and B are 2×2 matrices, then $A \otimes B = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}$. Write a C function which takes two square matrices A and B, each of order n, and a third matrix C as input. The function computes and stores in C the tensor product of A and B. **(10)**

3. (a) Write a recursive C function **`char *lastOccur (char *s, char c)`** which takes a string **`s`** and a character **`c`** as input; it returns a pointer to the last occurrence of **`c`** in **`s`**, or **`NULL`** if the character **`c`** does not occur in **`s`**. *Do not use any string library functions.* **(12)**

(b) Write an iterative C function **`void append (char *s, char *t)`** which appends the string **`t`** immediately after the end of the string **`s`**. For example, if **`s`** and **`t`** are passed respectively as **`"flanelo"`** and **`"humbo fractosus"`**, then your function should change the string **`s`** to **`"flanelohumbo fractosus"`**. *Do not use any string library functions.* **(8)**

4. (a) Write a suitable **`typedef`** for representing ordered pairs of integers of the form $\langle x, y\rangle$. **(3)**

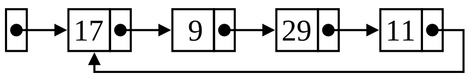
(b) Suppose we would like to represent a real number $x \times 10^y$ as an ordered pair $\langle x, y\rangle$, with the first member satisfying $1000 \leqslant |x| \leqslant 9999$. For example, 6.023×10^{23} is represented as $\langle 6023, 20\rangle$, whereas -1.6×10^{-19} is represented as $\langle -1600, -22\rangle$. Write a C function which takes two real numbers represented as two ordered pairs as given above and returns an ordered pair representing the product of the pairs passed as parameters. *Do not use any math library functions.* **(7)**

(c) Write a C function which takes a real number as input and returns the corresponding ordered pair. For example, if the floating point value 3.14159265 is passed, then it returns $\langle 3141, -3 \rangle$. Similarly, if -43217936.51289 is passed, then it returns $\langle -4321, 4 \rangle$. *Do not use any math library functions.* **(10)**

5. (a) Give a suitable **`typedef`** to represent a linked list of integers. **(3)**

(b) Write the C function which takes a list of integers, an integer i and another integer d. It enters d after the i-th element in the list. If $i = 0$, then it enters at the beginning of the list. Mention clearly whether a dummy header node is used in your function. **(8)**

(c) In a *circular linked list*, the **`next`** pointer of the last node points to the starting node of the list. Write a *recursive* C function that prints the elements of a circular linked list of integers in the *reverse* order (that is, from end to beginning). For example, for the following linked list, your function should print 11 29 9 17.



Use the following function prototype:

```
void printCList ( clist l, const clist h );
```

Here the second parameter points to the beginning of the list and is kept constant across the calls. Assume that no dummy header node is used in the circular linked list. **(9)**

6. (a) Give a suitable **`typedef`** for representing a stack of integers using *either* an array *or* a linked list. **(2)**

(b) Write C functions for implementing the following stack operations. If you have opted for the linked list representation, clearly indicate whether a dummy header node is used.

`init` – which constructs and returns an empty stack, **(1)**
`empty` – which returns a value indicating whether a given stack is empty or not, **(1)**
`push` – which pushes a given integer on to a given stack, **(3)**
`top` – which returns the top element of a given stack, **(2)**
`delete` – which deletes the top element of a given stack. **(3)**

(c) Write an *iterative* C function which takes an unsigned integer and prints its representation to the base 5 *using a stack*; the function should use the data type defined by you for representing a stack and only the functions of Part (b). **(8)**

7. (a) Give a suitable **`typedef`** with a brief explanation for representing a job which comprises a positive integer identification number and another positive integer for job size given as the time needed for completing the job. **(3)**

(b) Give a suitable **`typedef`** with a brief explanation for representing a queue of jobs using *either* an array *or* a linked list. **(3)**

(c) Give prototypes for the following functions on a queue of jobs (*no need* to write the function bodies). **(4)**

`empty` – which returns a value indicating whether a given queue is empty or not,
`enqueue` – which puts a given job in a given queue,
`front` – which returns the job from the front of a given queue if the queue is not empty; otherwise it returns a job with a negative identification number,
`dequeue` – which deletes the front of a given queue.

(d) One of the ways to process a collection of jobs, called *Round-Robin strategy*, is as follows: There is a predetermined period of time, called *time slice*. Each job is taken from the queue; if its time is less than or equal to the *time slice*, then it is processed until completion; otherwise, it is processed for a period equal to the *time slice*, its remaining time requirement is accordingly updated and then the job is put back into the queue for taking up in future after processing other jobs in a similar manner.

Write a C function which takes a queue of jobs and a time slice and processes the jobs using the Round-Robin strategy until the completion of each of them. The function should use the functions of Part (c). **(10)**