Shiv Nadar University

CSD101: Introduction to Computing and Programming

Endsem Exam

Max marks: 135 9-12-2021

Time: 3 hours for exam + 30mins (for upload, internet/power problems etc.).
Submit by 12.30pm.

1. *Answer all 8 questions.*
2. *For all program fragments assume that they are embedded in programs that compile and link without errors.*
3. *Please do not collaborate. In cases where the overlap between two answers is unreasonably high both will be given zero marks.*
4. *Answer each question directly on Blackboard. PDF/image files are not acceptable.*
5. *Any submission after 12.30pm will not be graded. This is a hard deadline.*

Questions 1 to 6 are based on the description below:

The instructor of a lab course conducts 10 labs during the semester and decides to use the best 8 labs for grading. Assume the course has `N` students and the data is available in two files:

`students.txt` - contains the roll number followed by the name of a student in the course - one student per line. Roll number and name is separated by white space. The name can have a maximum length of 25 characters and the roll number is an integer. The first 5 lines of the file `students.txt` are shown below (note they are not in roll number order):

```
20191234 Anurag Gupta
20202134 R T T Archna
20211342 Sania Sheikh
20181243 Remo Gonsalves
20193412 Surekha R Padmanabhan
```

`performance.txt` - marks for each student in each of the 10 labs. Each line in the file contains the roll number of the student followed by 10 non-negative integers giving the marks in each lab out of 100. All data items in the line are integers and are separated by white space. The first 5 lines of the file `performance.txt` are shown below (note that they are not in roll number order):

```
20181243 27 56 45 78 67 63 37 46 67 34
20191234 40 45 60 90 87 64 56 75 66 80
20202134 75 47 39 89 76 10 97 78  64  48
20211342 76 56 45 73 82 29 63 81 38 70
20193412 66 55 44 87 68 50 90 57 78 76
```

The instructor writes out results into a file called `results.txt` where each line in the file has the roll number, name, total marks of a student separated by a white space and the students are arranged in descending order of total marks.

The instructor defines the following two structures to store the data for each student and his/her performance. A pointer to an array for each structure is used to store the data for the whole class.

```
struct Student {
   unsigned int rollno;
   char name[26];//name can be at most 25 chars long
};

struct Marks {
   unsigned int rollno;
   unsigned int labmarks[10];//marks for the 10 labs
   unsigned int labtotal;//total of best 8 labs
};
```

The instructor's design for the program is as follows:

1. Read in the data from file `students.txt` into an array pointed to by `stud` of type `Student`. Read in data from file `performance.txt` into an array pointed to by `perf` of type `Marks`. This is in function `readData`.

2. For each student sort the `marks` field obtained in the 10 labs in descending order. Calculate the total and store it in the field `labtotal`. This is defined in function `calcLabtotal`. This uses a comparison function which is passed as an argument to the `qsort` function defined in the standard library `stdlib`. `qsort` uses the Quicksort algorithm.

3. Sort the array `perf` in descending order based on the `labtotal` field in each record. Sorting is again done using the `qsort` function from the standard library `stdlib` and it uses a function, passed as an argument to `qsort` for comparison. `qsort` uses the Quicksort algorithm for sorting. This is implemented as part of the `main` function.

4. Write out results into the file `results.txt` where each line in the file contains the roll number, name and total lab marks of the student separated by white spaces. The lines are in descending order of marks. This is implemented in the function `writeResults`.

5. The `main` function puts everything together.

The prototype of the `qsort` function and its description which is adapted from the documentation of `stdlib.h` is given below:

```
void qsort(void *arr, size_t n, size_t len, int (*cmp)(const void *, const void *))
```

`qsort` sorts array arr[0]...arr[n-1] of objects of size `len` in ascending order using the **Quicksort** algorithm. The `cmp` comparison function returns a negative value if its first argument is less than the second, zero if the first and second arguments are equal and a positive value if the first argument is greater than the second.

Note that `void *` is type compatible with any pointer type. The `const` keyword means the program cannot change the values pointed to and `size_t` is an unsigned integer type guaranteed to be at least 16-bits and is the type of the value returned by the `sizeof` function.

Assume all the code is in a single file `labcourse.c`. The preamble of the file `labcourse.c` is shown below:

```
#include <stdio.h>
#include <stdlib.h>
#define N 375 //This is the number of students

struct Student {
   unsigned int rollno;
   char name[26];//name can be at most 25 chars long
};

struct Marks {
   unsigned int rollno;
   unsigned int labmarks[10];//marks for the 10 labs
   unsigned int labtotal;//total of best 8 labs
};

/* All other function definitions and the main program start after this. */
```

Answer question 1 to question 6 based on the description above. In each question you will be told exactly what to do. In several questions you are given a program fragment with missing parts indicated by the comment `/* MISSING CODE ... */`, where `...` is a short description telling you what the code is doing. You should write out the complete program fragment by filling in code at the places indicated so that it works correctly. Your answer should have the full fragment and not just the missing code. The comments in the code fragments are there to help you write the code. You can leave them out when you write the answer.

Each `MISSING CODE ...` is typically one or two lines of code, rarely three. If you are writing more than that then you are most probably on the wrong track.

Note that there is no partial credit for the `MISSING CODE ...` questions. If your answer is logically correct you get full credit otherwise zero.

1. This question has a code fragment for the first step in the Instructor's program - namely reading the input. Fill in code for the parts indicated by `MISSING CODE ...` where `...` describes what the code does

```
1  void readLine(char l[], unsigned int n, FILE *in) {
2     /*Reads from current point in the input stream 'in' till it encounters
3     a new line or EOF. Puts chars in array l. Adds '\0' at the end.
4     */
5     char c;
6     int i=0;
7     c=fgetc(in);
8     while (/*MISSING CODE - Condition for the loop*/) {
9        /*MISSING CODE - to read chars into array l*/
10    }
11    l[i]='\0';
12    return;
13 }
14 void readData(struct Student *stud, struct Marks *perf) {
15    /*Reads student data into array pointed to by stud and marks data into
16    array pointed to by perf for N students*/
17    FILE *in1,*in2;
18    /*MISSING CODE - links input files to the file pointers*/
19    for(int i=0; i<N; i++) {
20       fscanf(in1,/*MISSING CODE - reads roll number from student.txt*/);
21       readLine(/*MISSING CODE - reads name from student.txt*/);
22       fscanf(in2, /*MISSING CODE - reads roll number from performance.txt*/);
23       /*MISSING CODE - loop to read the 10 lab marks from performance.txt*/
24    }
25    /*MISSING CODE - delink the file pointers from the files*/
26    return;
27 }
```

Solution:

```
void readLine(char l[], unsigned int n, FILE *in) {
   /*Reads from current point in the input stream in till it encounters
   a new line or EOF. Puts chars in array l. Adds '\0' at the end.
   */
   char c;
   int i=0;
   c=fgetc(in);
   while (c!='\n' && c!=EOF && i<=n /*c!='\'n' && c!=EOF also acceptable*/) {
```

```
      l[i++]=c;
      c=fgetc(in);
   }
   l[i]='\0';
   return;
}
void readData(struct Student *stud, struct Marks *perf) {
   /*Reads student data into array pointed to by stud and marks data into
   array pointed to by perf for N students*/
   FILE *in1,*in2;
   in1=fopen("students.txt","r");
   in2=fopen("performance.txt", "r");
   for(int i=0; i<N; i++) {
      fscanf(in1,"%d",&((stud+i)->rollno));
      readLine((stud+i)->name,25,in1);
      fscanf(in2, "%d", &((perf+i)->rollno));
      for(int j=0; j<10; j++)
         fscanf(in2, "%u", &(perf+i)->labmarks[j]);
   }
   fclose(in1);
   fclose(in2);
   return;
}
```

[(2,3),(2,2,2,2,(2,3),2)=20]

2. This code fragment is for step 2 in the Instructor's program. Here the lab marks are sorted in descending order and the lab total is calculated by choosing the best eight marks.

```
1  int mcmpfn(const unsigned int *n1, const unsigned int *n2)
2     /*This is a comparison function to sort labmarks in descending
3     order.*/
4     int retval=1;
5     /*MISSING CODE - if statement to set correct value for retval*/
6     return retval;
7  }
8
9  void calcLabtotal(struct Marks *perf) {
10    /*Calculates the field labtotal for record pointed to by perf.
11    First sorts the marks field in descending order and adds the first 8
12    values in marks to get the total that is stored in the field labtotal
13    Sorting is done by the qsort function in the standard library stdlib.
14    A comparison function must be passed to the qsort function as an argument.
15    */
16    /*MISSING CODE - call the qsort function to sort labmarks
17    in descending  order*/
18    perf->labtotal=0;
19    /*MISSING CODE - loop to add the best 8 marks to give labtotal*/
20    return;
21 }
```

Solution:

```
int mcmpfn(const unsigned int *n1, const unsigned int *n2){
   /*This is a comparison function to sort labmarks in descending
   order.*/
   int retval=1;
   if (*n1>*n2) retval=-1;
   else if (*n1==*n2) retval=0;
   return retval;
}
void calcLabtotal(struct Marks *perf) {
   /*Calculates the field labtotal for each record pointed to by perf.
   First sorts the marks field in descending order and adds the first 8
   values in marks to get the total that is stored in the field labtotal
   Sorting is done by the qsort function in the standard library stdlib.
```

```
   A comparison function must be passed to the qsort function as an argument.
   */
   qsort(perf->labmarks,10,sizeof(unsigned int),mcmpfn);
   perf->labtotal=0;
   for(int i=0; i<8; i++)
      perf->labtotal+=perf->labmarks[i];
   return;
}
```

[3,4,3=10]

3. This question gives the code fragment for step 4 in the Instructors program. Here the data is written out in file `results.txt`. Each line has a student's roll number, name and the labtotal. The data has already been arranged in descending order of labtotal before this step.

```
1  char *findName(unsigned int rollno, struct Student *stu) {
2     /*Searches for student's name given rollno. Returns NULL
3     if rollno does not exist in stu.*/
4
5     /*MISSING CODE - loop to find if rollno exists in stu and return
6       the corresponding name otherwise return NULL.*/
7  }
8
9  void writeResults(struct Marks *perf, struct Student *stud) {
10    /*Writes results in results.txt. Each line has:
11      rollno name labtotal
12      for a student.*/
13    FILE *out;
14    /*MISSING CODE - links output file to the file pointer*/
15    out=fopen("results.txt", "w");
16    /*MISSING CODE - loop to print the required data in results.txt*/
17    fclose(out);
18    return;
19 }
```

Solution:

```
char *findName(unsigned int rollno, struct Student *stu) {
   /*Searches for student's name given rollno. Returns NULL
   if rollno does not exist in stu.*/
   for (int i=0; i<N; i++)
      if (rollno==(stu+i)->rollno) return (stu+i)->name;
   return NULL;
}

void writeResults(struct Marks *perf, struct Student *stud) {
   /*Writes results in results.txt. Each line has:
     rollno name labtotal
     for a student.*/
```

```
   FILE *out;
   out=fopen("results.txt", "w");
   for(int i=0; i<N; i++)
      fprintf(out,"%d %s %d\n", (perf+i)->rollno,
         findName((perf+i)->rollno, stud), (perf+i)->labtotal);
   fclose(out);
   return;
}
```

[6,(1,(1,2,3,2))=15]

4. This question covers steps 3 and 5 of the program design. It shows a compare function for sorting and the main function of the program that executes the required steps sequentially.

```
1  int pcmpfn(const struct Marks *p1, const struct Marks *p2) {
2     /*This is a comparison function to sort performance records
3      in descending order of labmarks.*/
4     int retval=1;
5     /*MISSING CODE - if statement to set appropriate value for retval*/
6     return retval;
7  }
8
9  int main(void) {
10    struct Student *stud=calloc(/*MISSING CODE - calloc arguments*/);
11    struct Marks *perf=calloc(/*MISSING CODE - calloc arguments*/);
12    readData(stud, perf);
13    /*MISSING CODE - loop to sort labmarks in a perf record in
14      descending order and calculate labtotal using calcLabtotal.*/
15
16    /*MISSING CODE - use qsort to sort perf records in descending
17        order of labtotal*/
18    writeResults(perf, stud);
19    exit(0);
20 }
```

Solution:

```
int pcmpfn(const struct Marks *p1, const struct Marks *p2) {
   /*This is a comparison function to sort performance records
    in descending order of labmarks.*/
   int retval=1;
   if(p1->labtotal>p2->labtotal) retval=-1;
   else if (p1->labtotal==p2->labtotal) retval=0;
   return retval;
}

int main(void) {
   struct Student *stud=calloc(N, sizeof(struct Student));
   struct Marks *perf=calloc(N, sizeof(struct Marks));
   readData(stud, perf);
   for(int i=0; i<N; i++)
```

```
      calcLabtotal(perf+i);
   qsort(perf, N, sizeof(struct Marks), pcmpfn);
   writeResults(perf, stud);
   exit(0);
}
```

[3,(2,2,(2,3),4=15]

5. Based on the instructor's program design and the code in questions 1 to 4 what is the average case time complexity for the whole task in big-O notation? Justify your answer by calculating the big-O time complexity for each of the 5 major steps of the program design given in the description and the code in questions 1 to 4 and then infer the big-O complexity of the task. Remember there are N students in the course.
   No credit if proper justification not given.

Solution:

The task complexity is $O(n^2)$. Justification:
Step 1: N records are read so complexity $O(N)$.
Step 2: Sorting 10 marks and adding top 8 takes constant or $O(1)$ time. This is done for all N records so $O(N)$ time.
Step 3: N records are sorted using quicksort whose average case complexity is $O(N\,log_2(N))$.
Step 4: This writes out N records. For each record it has to search the `stud` array to find the name of the student given the roll number - this will take $O(N)$ time since it is linear search. So, this step takes $O(N^2)$ time on average.
Step 5: This simply does steps 1 to 4 in order so it is simply the sum of the above time complexities. But the dominant term in the addition is clearly $O(N^2)$.

So, the task as a whole has an average time complexity of $O(N^2)$.

[5,3,3,3,3,3=20]

6. Can you improve the big-O average case time complexity of the whole task? If yes, say how and if no then justify why not.

   Credit only if either yes/no answer is properly justified.

Solution:

Yes, the average time complexity can be improved.

Clearly, the limiting step is step 4 that has an average time complexity of $O(N^2)$. Since N records have to be written we can try and improve the search algorithm. This can be done by sorting the array of records `stud` and then use binary search instead of using linear search. Sorting will take

$O(N\,log_2(N))$ time; binary search takes $O(log_2(N))$ time on average and for N searches that will mean a time of $O(N\,log_2(N))$. So, now all complexities are either linear or $O(N\,log_2(N))$ so the task time complexity becomes $O(N\,log_2(N))$ which is a significant improvement on $O(N^2)$. Minor improvements can also be made by sorting the marks array as soon as it is read but that will not improve the big-O complexity.

[10]

7. A rooted binary tree is shown in the figure below. Each node (shown by a circle) has a value (an integer in this case) and pointers to at most two child sub-trees which are also binary trees. For example node 1 (the parent which is also the root node) has two child binary sub-trees a left sub-tree rooted at 2 and a right sub-tree rooted at 3. A `leaf` (example nodes 4, 5, 7) does not have any children and a `root` (example node 1) does not have any parent. Node 5 has node 3 as a parent.

   The `height` of a binary tree is the length of the longest path from the root to any leaf. So, the tree in the example figure has a height of 3 from node 1 to node 7. An empty tree is represented by *NULL*. The height of an empty tree is -1 and that of a leaf is 0. Note that a binary tree is a recursive data structure.

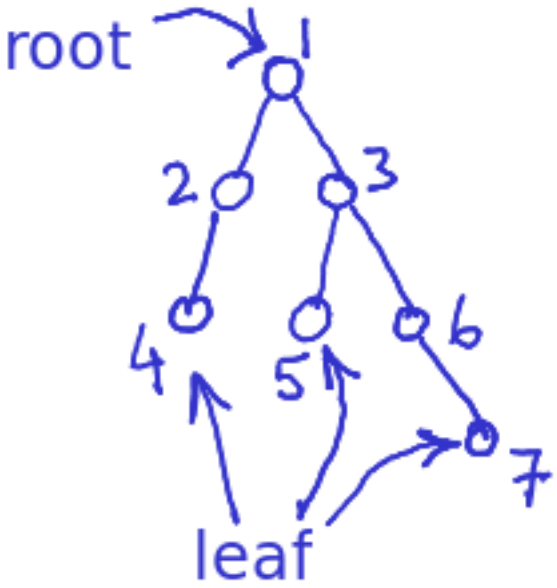

Figure 1: An example of a binary tree.

We can represent a binary tree by the structure below.

```
struct btree {
   int val;
   struct btree *ltp, *rtp;//pointers to left, right subtrees
};
```

Based on the above answer the questions below. For `/*MISSING CODE ...*/` you should fill in the appropriate missing code in the fragment so that it works correctly. In your answer give the whole fragment and not just the missing code. The `...` in the comment is a short description of what the missing code does.

(a) Fill in the missing code for the function `ctr` below which constructs a binary tree.

```
struct btree *ctr(int v, struct btree *ltrp, struct btree *rtrp) {
   /*Constructs a binary tree given node value v and pointers ltrp and
     rtrp to the left and right sub-trees respectively.*/
   struct btree *trp=malloc(/*MISSING CODE - argument of malloc*/);

   /*MISSING CODE - constructs the binary tree*/

   return trp;
}
```

Solution:

```
struct btree *ctr(int v, struct btree *ltrp, struct btree *rtrp) {
   /*Constructs a binary tree given node value v and pointers to
   the left and right sub-trees*/
   struct btree *trp=malloc(sizeof(struct btree));
   trp->val=v;
   trp->ltp=ltrp;
   trp->rtp=rtrp;
   return trp;
}
```

(b) Using function `ctr` write the expression to construct the binary tree in the figure above.

Solution:

```
ctr(1,ctr(2,ctr(4,NULL,NULL),NULL),
      ctr(3,ctr(5,NULL,NULL),ctr(6,NULL,ctr(7,NULL,NULL)))
   )
```

(c) The function `height` finds the height of a binary tree. Given that the height of a binary tree is the length of the longest path from the root till any leaf and that the height of an empty tree is -1 and that of a leaf is 0 fill in the missing code in the recursive function `height` below. Assume that function `int max(int, int)`, which returns the maximum of two integers, is available.

```
int height(struct btree *trp) {
   /*MISSING CODE - base case for height of empty tree*/

   /*MISSING CODE - base case for height of a leaf*/

   /*MISSING CODE - recursive case for height. It should be one
    more than the max height of the two sub-trees.*/
}
```

Solution:

```
int height(struct btree *trp) {
   if (trp==NULL) return -1;//empty tree has ht of -1
   if (trp->ltp==NULL && trp->rtp==NULL) return 0;//leaf has ht of 0
   return (1+max(height(trp->ltp), height(trp->rtp)));//recursive case
}
```

[(2,3),6,(1,3,5)=20]

8. You have divided your code into the following `.c` files: `baseLib.c`, `lib1.c`, `lib2.c` and `myprog.c`. The file `myprog.c` has the `main` function. Assume that an executable called `myexe` has to be generated and that the `gcc` compiler is being used for compiling and generating the executable. The header/ preamble in each file is shown below:

```
File: baselib.c
#include <string.h>

/* baselib library code */

File: lib1.c
#include "baselib.h"

/* lib1 library code */

File: lib2.c
#include "baselib.h"

/* lib2 library code */

File: myprog.c
#include <stdio.h>
#include <stdlib.h>
#include "lib1.h"
#include "lib2.h"

/* code of myprog */
```

Note that standard libraries are indicated by `#include <...>`.

(a) Which `.h` files do you have to create?

Solution:
`baselib.h, lib1.h, lib2.h` *Marks: 2 marks for each file (6)*

(b) Give the sequence of `gcc` commands on the command-line to generate the executable `myexe`.

Solution:
First create all the `.o` files. (This can be done with one command or multiple commands.)
`gcc -c baselib.c lib1.c lib2.c myprog.c`
Then link them to create the executable.
`gcc -o myexe baselib.o lib1.o lib2.o myprog.o`
Marks: 1 mark for each .o file (4 marks); 3 marks for creating executable

(c) Write the `makefile` that can be used to create the executable `myexe`. **Do not use any implicit rules**. Give all rules explicitly. Include a phony `clean` target to remove the executable and all `.o` files.

Solution:

The makefile is shown below:

```
myexe : myprog.o lib1.o lib2.o
gcc -o myexe myprog.o lib1.o lib2.o

myprog.o : myprog.c lib1.c lib2.c lib1.h lib2.h
gcc -c myprog.c

lib1.o : lib1.c baselib.o baselib.h
gcc -c lib1.c

lib2.o : lib2.c baselib.c baselib.h
   gcc -c lib2.c

baselib.o : baselib.c
   gcc -c baselib.c

.PHONY : clean
clean :
rm myexe myprog.o baselib.o lib1.o lib2.o
```

Marks: 2 marks for each rule, 1 - for dependency, 1 - for command (12 marks). No partial credit for a dependency or a rule.

[6,7,12=25]