CMPT 125 D200, Spring 2023

Final Exam - Solutions
April 15, 2023

Name___________________________

SFU ID: |__|__|__|__|__|__|__|__|__|

Problem 1	
Problem 2	
Problem 3	
TOTAL	

Instructions:

1. Duration of the exam is 180 minutes.
2. Write your full name and SFU ID **clearly**.
3. This is a closed book exam, no calculators, cell phones, or any other material.
4. The exam consists of three (3) problems.
5. Write your answers in the provided space.
6. There is an extra page at the end of the exam. You may use it if needed.
7. Explain all your answers.
8. Really, explain all your answers.

Good luck!

Problem 1 [30 points]

a) [6 points] Rewrite the function what() without using recursion. Explain the functionality of the function before writing code.

```
#include <stdio.h>
int what(int n) {
  if (n<=0)                    if n<=0, the function returns 0;
    return 0;
  else {
    printf("%d ", n);         if n>=1 the function prints "n" and then calls recursion.
    return what(n-1)+1;       overall it prints "n n-1 n-2… 1" and return n.
    printf("%d ", n);         ← this printf is never executed because it is after return.
  }
}
int what(int n) {
  if (n<=0)
    return 0;
  for (int i = n; i > 0; i--)
    printf("%d ", i);
  return n;
}
```

b) [7 points] Will the code below compile?
If yes, what will be the result of the execution? If not, explain errors/warnings/potential issues.

```
#include <stdio.h>
#include <string.h>
void foo() {
  static char* str = "hello";    static variables remember their states
  printf("%s\n", str);           from previous executions of the function.
  if (strcmp(str,"hello")==0)
    str = "hi";
  else
    str = "hello";
}
int main() {
  foo();
  foo();
  foo();
  foo();
  return 0;
}
```

Answer: The function will print:

Hello

hi

hello

hi

c) [7 points] Will the code below compile?
If yes, what will be the result of the execution? If not, explain errors/warnings/potential issues.

```
#include <stdio.h>
int* foo(int start) {
  int arr[3];
  arr[0] = start;
  arr[1] = start*2;
  int* ret = arr;
  return ret;
}

int main() {
  int* a1 = foo(0);
  printf("a1 = [%d, %d]\n", a1[0], a1[1]);
  int* a2 = foo(3);
  printf("a1 = [%d, %d]\n", a1[0], a1[1]);
  printf("a2 = [%d, %d]\n", a2[0], a2[1]);
  return 0;
}
```

Answer: The code will compile, but the behavior is undefined because foo() returns a pointer to a local variable. ***Run this code on your laptop, and see what it prints***

d) [10 points] Consider the function asterisks().

```
char* asterisks(int n) {
  char* ret = malloc(n+1);
  ret[0] = 0;
  for (int i = 0; i < n; i++)
    strcat(ret, "*");
  return ret;
}
```

[5 points] Explain the functionality of the function.

Answer: The function returns a string consisting of n asterisks.

[5 points] What is the running time of the function? Use big-O notation to express your answer.
** *Hint: what is the running time of strcat?*

Answer: Note that in the i'th iteration the running time of strcat(ret, “*”) is O(i). This is because strcat needs to reach the i'th position of ret. Therefore, the total running time is O(1+2+3+4+...(n-1)) = O(n^2).

Problem 2 [35 points]

a) [10 points] Consider the *MergeSort* algorithm with linear time merging we saw in class.
1. List all recursive calls the algorithm will perform on the array A = [9,1,8,7,3,6,4,0].
2. List all ***comparisons*** the algorithm will perform on the array A = [9,1,8,7,3,6,4,0].

Answer: 1. Recursive calls are:

```
          [9,1,8,7,3,6,4,0]  ← main call
           /                \
     [9,1,8,7]         [3,6,4,0]
      /       \            /        \
 [9,1]     [8,7]      [3,6]      [4,0]
 /     \      /    \      /     \      /     \
[9]  [1]   [8] [7]  [3]   [6]  [4]    [0]
```

2. Comparisons are:

merge([9], [1]) → [1,9] - compare (9,1)
merge([8], [7]) → [7,8] - compare (8,7)
merge([3], [6]) → [3,6] - compare (3,6)
merge([4], [0]) → [0,4] - compare (4,0)

merge([1,9], [7,8]) → [1,7,8,9] - compare (1,7) (9,7) (9,8). Then move 9 without comparisons
merge([3,6], [0,4]) → [0,3,4,6] - compare (3,0) (3,4) (6,4). Then move 6 without comparisons

merge([1,7,8,9],[0,3,4,6] - compare (1,0) (1,3),(7,3),(7,4),(7,6). Then move 7,8,9 without comparisons

b) [5 points] Give an example of an array of length 5 on which *InsertionSort* makes exactly one swap. Explain your answer.

Answer: For example take [2,1,3,4,5]

The only swap will be (2,1)

c) [5 points] Give an example of an array of length 8 on which *InsertionSort* makes a total of 28 swaps. Explain your answer.

Answer: For example take [8,7,6,5,4,3,2,1]

7 will make one swap, 6 will make two swaps, and so on,,. 1 will make seven swaps

The total number of wasps will be 1+2+3+4..+7=28

d) [10 points] Consider the following variant of the selection sort algorithm.
Given an array A of length n it works as follows:

- Let M[0…n-1] be an array of length n
- For i=0…n-1
    - Let j be the index of the minimal element in A[i…n-1]
    - swap(A[i], A[j])
    - M[i] = j

In the end of the funcion the array A is sorted, and the array M contains the indices of the minimal elements in each iteration.
For example if we start with A = [5,9,6,1,8], then

- after the iteration i=0 we get M[0]=3, we swap 5 and 1, and A becomes [**1**,9,6,5,8]
- after the iteration i=1 we get M[1]=3, we swap 6 and 5, and A becomes [1,**5**,6,9,8]
- after the iteration i=2 we get M[2]=2, we swap 6 and 6, and A becomes [1,5,**6**,9,8]
- after the iteration i=3 we get M[3]=4, we swap 9 and 8, and A becomes [1,5,6,**8**,9]
- after the iteration i=4 we get M[4]=4, we swap 9 and 9, and A becomes [1,5,6,8,**9**]

Write a function that performs "reverse engineering" to the algorithm.
The function gets the sorted array A of length n, and the array M of length n.
The function returns A to its initial state before selection sort was applied.
Explain the idea before writing code.

```
void reverse_selecion_sort(int* A, const int* M, int n) {
```

Answer: The idea is to run the loop of the insertion sort in the reverse order. The loop will go from n-1 to 0, and will swap A[i] with the minimum, which exactly reverses the i'th iteration of the selection sort. Just stare at the code below until you are convinced it works.

```
  for (int i = n-1; i >= 0; i--) { // swap A[i] with A[M[i]]
    int tmp = A[i];
    A[i] = A[M[i]];
    A[M[i]] = tmp;
  }




}
```

[5 points] Use big-O notation to express the running time of your algorithm.

Answer: One loop of length n with O(1) operations per iteration. Total running time is O(n).

Problem 3 [25 points]

In the problem a Linked List of ints is represented as follows.

```
struct LL_node {
  int data;
  struct LL_node* next;
};
typedef struct LL_node LL_node_t;

typedef struct {
  LL_node_t* head;
  LL_node_t* tail;
} LL_t;
```

a) [10 points] Write a function that gets two linked lists, and checks if they are equal.
The running time of the function must be O(length of the shortest list).
For example, if one list has O(1) nodes, and the other has n nodes for some large n, then the running time should be O(1) independent of n.

*** Note: LL_t does not have the field size.*

```
bool are_equal(LL_t* list1, LL_t* list2) {

  LL_node_t* cur1 = list1->head;
  LL_node_t* cur2 = list2->head;

  // iterate through both arrays until reaching the end of one of them
  while (cur1 && cur2) {
    if (cur1->data != cur2->data)
      return false;

    cur1 = cur1->next;
    cur2 = cur2->next;
  }

  if (cur1==NULL && cur2==NULL)
      return true;
  else
      return false;




}
```

b) [15 points: 5 points each] Implement the following standard functions on a Linked List with pointers to head and tail, using the struct above.

```
// adds a node with the given value to the head of the list
// the running time is O(1)
void add_to_head(LL_t* list, int val) {
  LL_node_t* newNode = malloc(sizeof(LL_node_t));
  if (!newNode)
   return;

  newNode->data = val;
  newNode->next = list->head;
  list->head = newNode;
  if (list->tail==NULL)
    list->tail = newNode;

}
// adds a node with the given value to the tail of the list
// the running time is O(1)
void add_to_tail(LL_t* list, int val) {
  LL_node_t* newNode = malloc(sizeof(LL_node_t));
  if (!newNode)
   return;
  newNode->data = val;
  newNode->next = NULL;

  if (list->tail==NULL) {
    list->head = newNode;
    list->tail = newNode;
  }
  else {
    list->tail->next = newNode;
    list->tail = newNode;
  }

}
// sets the i'th node to be the given value (indices start from 0)
// Assumption: the list has at least i+1 nodes.
void set_value(LL_t* list, int i, int value) {
  LL_node_t* cur = list->head;
  for (int ind = 0; ind < i; ind++)
    cur = cur->next;

  cur->data = value;
}
```

Extra page

Empty page