CMPT 125, Fall 2021 - SOLUTIONS

Midterm Exam
October 25, 2021

Name_____

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

Problem 1	
Problem 2	
Problem 3	
Problem 4	
TOTAL	

Instructions:

1. Duration of the exam is 90 minutes.
2. Write your 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 four (4) problems. Each problem is worth 25 points.
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 [25 points]

a) [6 points] What will be the output of the following program? Explain your answer.

```
#include <stdio.h>
enum numbers {ZERO, ONE, TWO, THREE};

void foo(int* x, int y) {
     int z = 5;
     *x = z;
     y = z;
     x = &y;
}

int main() {
     int a = ONE, b = TWO;
     foo(&a, b);
     printf("a = %d, b = %d", a, b);
     return 0;
}
```

ANSWER: a = 5 b = 2

Explanation:

- In the beginning a = 1, b = 2
- *x gets the value 5, hence the variable a in main becomes 5.
- The line y=z changes y to 5.
- The line x=&y makes x point to y, but this has no effect on a,b in main

b) [6 points] Will the code below compile?
If yes, what will be the output? If not, explain why.

```
#include <stdio.h>

void str_manipulate(char* s) {
    char c = '\0';
    *(s+2) = c;
}

int main() {
    char str[13] = "CMPT125";
    str_manipulate(str+3);
    printf("%s\n", str+1);
    return 0;
}
```

ANSWER: The code will compile, the output will be “MPT1”

Explanation:

- str+3 points to the letter T, so str_manipulate() gets “T125” as argument
- *(s+2) = ‘\0’ changes the char ‘2’ to ‘\0’, thus changing str to “CMPT1”
- printf() prints str starting from M

c) [7 points] Will the code below compile?
If yes, what will be the output? If not, explain why.

```
#include <stdio.h>

int* get_arr3() {
  int arr[5] = {1,2,3};

  int* ret = arr;
  return ret;
}

int main() {
  int* a1 = get_arr3();
  int* a2 = get_arr3();
  a1[0] = 7;
  a2[1] = 8;
  printf("a1 = [%d, %d, %d]\n", a1[0], a1[1], a1[2]);
  printf("a2 = [%d, %d, %d]\n", a2[0], a2[1], a2[2]);
  return 0;
}
```

ANSWER: The code will compile, but the output is undefined because get_arr3 returns a pointer to a local array. This means a1 and a2 are not guaranteed to have the values {1,2,3} when the function returns.

*Remark: arr[5] = {1,2,3} means that the first 3 elements of arr are set to 1,2,3 but the remaining two are not initialized. That's not an error, but in practice we should initialize variables before using them

d) [6 points] What is the running time of the function below. Use Big-O notation to express your answer. Explain your solution.

```
int foo(int n) {
  if (n>0) {
    int i, sum = 0;
    for(i=0; sum<n ; i++) // note the condition is sum < n
      sum = sum+i;
    return i+foo(n-1);
  }
  return 0;
}
```

ANSWER: In the i'th iteration the sum is equal to $1+2+3+4+...i = i(i+1)/2 > i^2/2$.
Therefore, the for loop will not have more than $O(\sqrt{n})$ iterations.
This gives us the recursive formula for running time: $T(n) <= T(n-1) + C*n^{1/2}$, and $T(0) = 1$

If we open this formula using $T(n-1) <= T(n-2) + C*(n-1)^{1/2}$, we'll get
$T(n) <= T(n-2) + C*(n-1)^{1/2} + C*n^{1/2} <= T(n-3) + C*(n-2)^{1/2} + C*(n-1)^{1/2} + C*n^{1/2}$ …
By repeating it n times we get: $T(n) <= C*n^{1/2} + C*(n-1)^{1/2} + C*(n-2)^{1/2} + C*(n-3)^{1/2} + …$
There are n terms in the sum, each less than $C*n^{1/2}$. Therefore, the total time is $O(n^{1.5})$

Problem 2 [25 points]

a) [5 points] Consider the **Binary Search** algorithm. How many comparisons will it make on the input A = [2, 4, 6, 8, 10, 12, 14, 15, 18, 90, 100] when searching for 7. Explain your answer

ANSWER: Two possible solutions will be accepted:

A) Total 3 comparisons.
The algorithm will make the following comparisons:

1. Compare 7 to 12 → search in [2,4,6,8,10]
2. Compare 7 to 6 → search in [8,10]
3. Compare 7 to 8 → search in []
4. Return "NOT FOUND"

B) Total 4 comparisons.
The algorithm will make the following comparisons:

1. Compare 7 to 12 → search in [2,4,6,8,10]
2. Compare 7 to 6 → search in [8,10]
3. Compare 7 to 10 → search in [8]
4. Compare 7 to 8 → search in []
5. Return "NOT FOUND"

b) [8 points] Give an example of an array of length n, on which **InsertionSort** makes exactly one swap in each iteration of the outer loop.

ANSWER: A = [n,1,2,3,4,5,...n-1] That is, the array is almost sorted except for the first element.

Insertion sort will work as follows:

- Insert 1: swap 1 with n → [1,n,2,3,4,5,6,7, …, n-1]
- Insert 2: swap 2 with n → [1,2,n,3,4,5,6,7, …, n-1]
- Insert 3: swap 3 with n → [1,2,3,n,4,5,6,7, …, n-1]
- And so on

In the i'th iteration we'll swap i with n

c) [6 points] Recall the **MergeSort** algorithm.

```
void merge_sort(int* A, int n) {
  if (n >= 2) {
    int mid = n/2;
    merge_sort(A, mid);          // sort left half
    merge_sort(A+mid, n-mid);    // sort right half
    merge(A,n,mid); // merge() we saw in class with running time O(n)
  }
}
```

What is the running time of this algorithm when applied on a sorted array of length n?
Write the tightest possible upper bound on the running time. Explain your answer.

ANSWER: The running time is O(n log(n)) even on a sorted array.
Indeed ,even if the array is sorted, the running time is T(n) = 2T(n/2) + O(n), which behaves like O(n log(n)).

d) [6 points] Consider the following variant of **MergeSort**:

```
bool is_sorted(int* A, int n); //checks if A is sorted in O(n) time

void merge_sort(int* A, int n) {
  if (n <= 1 || is_sorted(A, n)) // stopping condition
    return;

  int mid = n/2;
  merge_sort(A, mid);          // sort left half
  merge_sort(A+mid, n-mid);    // sort right half
  merge(A,n,mid); // merge() we saw in class with running time O(n)
}
```

What is the running time of this algorithm when applied on A=[n, n-1, n-2, n-3, n-4,… 3, 2, 1]?
Write the tightest possible upper bound on the running time. Explain your answer.

ANSWER: The running time is O(n log(n)) on this array..
This is because the subarrays we are sending in recursive calls are never sorted, so the running time is still T(n) = 2T(n/2) + O(n), which behaves like O(n log(n))

Problem 3 [25 points]

a) [8 points] Write a function that gets two strings and computes the length of their longest common prefix. For example,

- longest_common_prefix("abcd", "ab12") is 2 - the prefix is "ab"
- longest_common_prefix("abcd", "cd") is 0 - the prefix is empty
- longest_common_prefix("abcd", "abcdefg") is 4 - the prefix is "abcd"

Explain your idea before writing code.

```
int longest_common_prefix(const char* str1, const char* str2) {
```

ANSWER: have a counter.
We'll iterate over both arras, and as long as str1[i]==str2[i] we will increase the counter.
We will stop when either str1[i]!=str2[i] or when we reach the end of one of the strings

```
  int count = 0; // this will be the return value
  int i = 0;
  // increase ret as long as str1[i]==str2[i]
  //and we haven't reached the end of the strings
  while (str1[i]==str2[i] && str1[i] != 0 && str2[0] != 0) {
      count++;
      i++;
  }
  return count;
}
```

**Remark: In the while condition it is ok to check

```
  while (str1[i]==str2[i] && str1[i] != 0)
```

**or

```
  while (*(str1+i)==*(str2+i) && *(str1+i) != 0)
```

**or

```
  while (*(str1+i)==*(str2+i) && *(str1+i))
```

```
}
```

[4 points] What is the running time of your function? Use big-O notation to state your answer. Give the tightest possible answer.

ANSWER: The running time is O(n), where n is the length of the longest common prefix.
Note that n can be potentially much smaller than the input length

*Answers O(input length) will also be accepted

b) [10 points] Implement the function strcat. The function gets *dest* and *src*, and appends the string pointed to by *src* to the end of the string pointed to by *dest*.
This function returns a pointer to the resulting string dest.

For example, if dest = “ABC” and src = “DEF”, then after the function returns we have dest = “ABCDEF”.

You are not allowed to use any library functions to solve this.

```
// assumption: dest has enough memory allocated
// to store the concatenation of the two strings
char* strcat(char* dest, const char* src) {

  char* ptr1 = dest;
  const char* ptr2 = src;

// find the end of dest
  while (*ptr1 != 0) // same as *ptr1 != '\0'
      ptr1++;


// copy ptr2 to ptr1
  while (*ptr2 != 0) {
      *ptr1=*ptr2;
      ptr1++;
      ptr2++;
  }
  *ptr1 = '\0'; // don't forget to end with NULL terminator


  return dest;




}
```

[3 points] What is the running time of your function in terms of the length of the strings in the worst case? Use big-O notation to state your answer. Give the tightest possible answer.

ANSWER:
The first loop runs for O(length(dest)) time
The second loop runs for O(length(src)) time

The total running time is O(length(dest) + length(src)).

Problem 4 [25 points]

a) [8 points] Write a function that gets a parameter n. It reads n ints from the user (using scanf), and prints the numbers in the reverse order.

```
void read_and_print_reverse(int n) {

  // make sure to use malloc
  // never use variable length arrays: int ar[n] is wrong!
  int* ar = (int*) malloc(n*sizeof(int));

  int i;
  for (i=0;i<n;i++)
      scanf("%d", &ar[i]); // same as scanf("%d", ar+i);

  for (i=n-1;i>=0;i--)
      printf("%d ", ar[i]); // printf("%d ", *(ar+i));

 free(ar);









}
```

b) [15 points] Write a function that gets an array of ints of length n, and returns an array of length n, such that the output[i] is equal to the maximal element in the input subarray [0,..., i]. For example,

- input [1, 4, 3, 8, 2, 9]
- output [1, 4, 4, 8, 8, 9].

You may write helper functions if that makes the solution more readable.

- A correct answer with linear running time, will give you 15 points
- A correct answer with quadratic running time, will give you 10 points

```
int* max_prefixes(int* ar, int n) {

  if (n==0)
      return NULL;

  int* output = (int*) malloc(n*sizeof(int));
  if (output==NULL)
      return NULL;

  output[0] = ar[0];
  int i;
  for (i=1;i<n;i++) {
      if (output[i-1] > ar[i])
        output[i] = output[i-1];
      else
        output[i] = ar[i];
  }

  return output;

}
```

[3 points] Explain the running time of your function.

ANSWER: O(n).
We have one for loop of length n with O(1) operations in each iteration

****
Alternatively can define a helper function that computers max of the first i elements

```
  int max(int* ar, int i) {}
```

Then, we can populate output using this max function

```
  for (i=1;i<n;i++) {
    output[i] = max(ar, i);
```

This runs in $O(n^2)$ time

Extra page