

CS5008 Practice Midterm

1. Which of the following is a valid declaration of a pointer in C?

- (a) `int *ptr;`
- (b) `ptr int*;`
- (c) `*ptr int;`

2. Consider the following code:

```
int sum = 0;
for (int i = 1; i < n; i = i * 2) {
    sum++;
}
```

What is the run-time of the code (in Big-O notation?)

- (a) $O(n^3)$
- (b) $O(n)$
- (c) $O(n^2)$
- (d) $O(2^n)$
- (e) $O(\log(n))$

3. Which of the following statements about Merge Sort and Quick Sort is true?

- (a) Merge Sort has a worst-case time complexity of $O(n \log n)$, while Quick Sort has a worst-case time complexity of $O(n^2)$.
- (b) Merge Sort uses divide and conquer strategy, while Quick Sort uses dynamic programming.
- (c) Merge Sort has a worst-case space complexity of $O(n)$, while Quick Sort has a worst-case space complexity of $O(\log n)$.

4. Given an array [34, 7, 23, 32, 5, 62], Write down the split and merge process of the merge sort algorithm applied to this array.

| | | | |
|----------------------|----|-----------|----|
| 34, 7, 23, 32, 5, 62 | | | |
| 34, 7, 23 | | 32, 5, 62 | |
| 34, 7 | 23 | 32, 5 | 62 |
| 7, 34 | 23 | 5, 32 | 62 |
| 7, 23, 34 | | 5, 32, 62 | |
| 5, 7, 23, 32, 34, 62 | | | |

5. Complete the missing part of the Bubble Sort function in the provided C code.

```
#include <stdio.h>
```

```
void bubbleSort(int arr[], int n) {  
    int i, j;  
  
    for (i = 0; i < n - 1; i++) {  
        //complete the code  
    }  
}
```

Please see [bubblesort.c](#) file

6. Determine the type of sort given the following code: (Bubble, Selection, Insertion)

```
void swap(int *xp, int *yp)  
{  
    int temp = *xp;  
    *xp = *yp;  
    *yp = temp;  
}
```

Selection Sort

```
void Sort(int arr[], int n)  
{  
    int i, j, min_idx;  
    for (i = 0; i < n-1; i++)  
    {  
        min_idx = i;  
        for (j = i+1; j < n; j++)  
            if (arr[j] < arr[min_idx])  
                min_idx = j;  
  
        if(min_idx != i)  
            swap(&arr[min_idx], &arr[i]);  
    }  
}
```

7. Explain why the time complexity of merge sort is $O(n \log n)$, while that of quicksort can be $O(n^2)$ in the worst case. Discuss how this difference affects the choice of sorting algorithm.

The array is divided into two halves recursively until each subarray contains a single element. This will have $\log n$ levels. At each level, there are n elements. So, merging the subarrays back together in a sorted order takes $O(n)$. Therefore, the overall time complexity is $O(n \log n)$.

Each time we select a bad pivot, always the smallest² or largest element in the array.

Merge sort is more reliable because the time complexity will always be $O(n \log n)$, so when stability is required, we can choose merge sort. Also, the space complexity will be $O(n)$ because we require extra space to store the temporary arrays. Therefore, with limited space, we had better not to choose merge sort.

Quicksort: the average time complexity is $O(n)$, and we can do swaps in place and the time complexity is $O(1)$ (if we recursively call the partition method recursively, we will need $O(\log n)$ space. So, if we can take the risks of $O(n^2)$ time complexity in the worst case. Quicksort is a good way to do sorting.

8. (Coding) Find the K -th Smallest Element Using QuickSort Partition: Implement the Quick-Select algorithm to find the k -th smallest element in an unsorted array using the QuickSort partitioning method. You may create helper functions, such as a partition function, to assist in your implementation (the function should operate in-place)

```
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function for QuickSelect
int partition(int arr[], int low, int high) {

// QuickSelect function to find K-th smallest element
int quickSelect(int arr[], int low, int high, int k) {
//TODO
}

// Example input and output:
// Input: arr[] = {7, 10, 4, 3, 20, 15}, k = 3
// Output: 7

int main() {
    int arr[] = {7, 10, 4, 3, 20, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;

    printf("The %d-th smallest element is: %d\n", k, quickSelect(arr, 0, n - 1, k));
    return 0;
}
```

[Please see findksmallest.c file](#)

9. Complete the following function to remove all occurrences of a specified element `x` from an array `arr` of size `n`, and return the new size of the array. Fill in the blanks to make the code functional.

```
#include <stdio.h>
// Function to remove all occurrences of x from arr and return new size
int removeElement(int arr[], int n, int x) {
    int newSize = 0;
    for (int i = 0; i < n ; i++) {
        if (arr[i] != x ) {
            arr[newSize] = arr[i];
            newSize++;
        }
    }
    return newSize;}

```

10. (Coding) Implement the below “insertionSortDescending()” function in C. The function should sort the array in descending order, and calculate the number of swaps and comparisons required.

```
int main() {
    int arr[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int comparisons, swaps;

    insertionSortDescending(arr, n, &comparisons, &swaps);

    printf("Sorted array in descending order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    printf("\n");
    printf("Total comparisons: %d\n", comparisons);
    printf("Total swaps: %d\n", swaps);

    return 0;
}

```

[please see insertionSortDescending.c](#)

11. Analyze the following C program by commenting on the specified lines and answering the questions. After commenting, also write the expected output of the program.

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_PEOPLE 3
int i;
typedef struct { // Comment in this line: What is the purpose of this structure?
    char name[20]; // This structure defines a person with a name(String) and age(int).
    int age;
} Person;
Person* people[NUM_PEOPLE];

// what does this function do? // This function copies a string from source to destination character
// by character.
void copyString(char *destination, const char *source) {
    int j = 0;
    while (source[j] != '\0') {
        destination[j] = source[j]; //Comment in this line: What does this line do?
        j++; // This copies each character from source to destination.
    }
    destination[j] = '\0'; // Comment in this line: What does this line do?
} // Adds a NULL terminator to mark the end of destination string.

int main() {
    // Comment in this line: what is the purpose of this loop? // This loop dynamically allocates
    for (i = 0; i < NUM_PEOPLE; i++) { // memory for each Person structure
        people[i] = (Person*)malloc(sizeof(Person)); // in the people array.
    }
    copyString(people[0]->name, "John"); // Comment in this line: What does this line do?
    people[0]->age = 25; // Comment in this line: What does this line do?
    // This copies the String "John" into people[0]-> name.
    copyString(people[1]->name, "Alice"); // Assigns the age 25 to the first person.
    people[1]->age = 30;

    copyString(people[2]->name, "Bob");
    people[2]->age = 22; // This will print the name and age of each person stored
    // in the people array.

    // Comment in this line: what is the purpose of this loop?
    for (i = 0; i < NUM_PEOPLE; i++) {
        printf("Name: %s, Age: %d\n", people[i]->name, people[i]->age); //Comment
    }

    // Comment in this line: what is the purpose of this loop?
    for (i = 0; i < NUM_PEOPLE; i++) {
        free(people[i]);
    } // This frees the dynamically allocated memory to prevent
    return 0; // memory leaks.
}
```

Expected Output:

Name: John, Age: 25

Name: Alice, Age: 30

Name: Bob, Age: 22