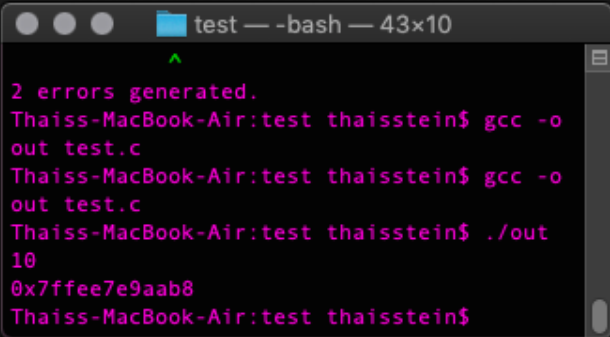


Student: Thais Farias

1. Pointers in C are essentially a variable that contains memory addresses. That is pretty clear in the example below, when we have an integer variable **a** and a pointer **p\_a**. When printing these two variables, we see that **a** stores a variable while **p\_a** stores its address.

```
4
5  int main() {
6      int a = 10;
7      int * p_a = &a;
8      printf("%d\n", a);
9      printf("%p\n", p_a);
10
11      return 0;
12  }
```



```
test — -bash — 43x10
2 errors generated.
Thaiss-MacBook-Air:test thaissstein$ gcc -o
out test.c
Thaiss-MacBook-Air:test thaissstein$ gcc -o
out test.c
Thaiss-MacBook-Air:test thaissstein$ ./out
10
0x7ffee7e9aab8
Thaiss-MacBook-Air:test thaissstein$
```

Thus, we say that the pointer **p\_a** *points* to **a**. The operators used can also be seen in the example above: **\*** and **&**. Since the pointer stores a memory address, **&** is the operator used to access this address. On the other hand, **\*** returns the value of the specific address. So if we printed **\*p\_a** in the example above, it would give us the value of the variable the pointer is indicating (10). Also the operator **\*** is used when declaring a pointer in the code to specify it is a pointer and not a standard variable.

They can be really useful when we have to access the same variable in different parts of the code. Instead of copying this variable every time we can just have pointers that contain its address, saving memory space, time and -possibly- headaches once you can easily alter the variable's value without having to worry about what will happen in every part of the code it is used. They can also be used to pass an argument for reference, so the function can change the value of that argument passed by reference (example below).

```
/* partitioning with the middle element as pivot */
int partition(int* array, int smaller, int biggest)
{
    int pivot = array[((biggest + smaller)/2)];
    int i = (smaller - 1);
```

2. (Every piece of code that is used in this question has been taken from my personal Github repository: <https://github.com/thaisstein/cprogramminglabs/>)

The **slow** sorting algorithms are the Bubble Sort, Insertion Sort and Selection Sort. The **fast** sorting algorithms are Quicksort and Heap Sort.

**Bubble Sort** works by comparing the elements two by two when going through an array (lines 47 and 48 of the code), and if element #2 is greater in value than element #1 (line 49), then they swap places in the array (line 50). Doing this until the end of the array will have as an output a sorted array.

The algorithm is pretty simple and follows that logic, as we can see below:

```
45 void bubble_sort(int* array, int n) {
46     int temp, i, j;
47     for (i = 0; i < n - 1; i++) {
48         for (j = 0; j < n - i - 1; j++) {
49             if (array[j] > array[j + 1]) {
50                 change(&array[j], &array[j+1]);
51             }
52         }
53     }
54 }
55
```

**Insertion Sort** works by dividing the array into a sorted and an unsorted piece. Then, the elements in the unsorted part are taken and we find where they should be in the sorted part and insert it there. This process is repeated until there's no elements left to sort, giving us a sorted array. The algorithm can be seen below:

```
57 void insertion_sort(int* array, int n) {
58     int temp, j;
59     for (int i = 1; i < n; i++) {
60         temp = array[i];
61         j = i - 1;
62         while (j >= 0 && array[j] > temp) {
63             array[j + 1] = array[j];
64             j = j - 1;
65         }
66         array[j + 1] = temp;
67     }
68 }
```

**Selection Sort** works by using the minimum element in the array (min). We divide the array into sorted and unsorted parts and then pick **min** from the unsorted part of the array at each interaction and move it to the front of the sorted part, giving us a sorted array at the end.

The algorithm made can be seen as follows:

```

69
70 void selection_sort(int* array, int n) {
71     int min, temp;
72     for (int i = 0; i < n-1; i++) {
73         min= i;
74         for (int j = i+1; j < n; j++)
75             if (array[j] < array[temp])
76                 min = j;
77         if(min != i)
78             change(&array[min], &array[i]);
79     }
80 }

```

**Quicksort** works by selecting a pivot element with the partition function (can be the first element of the array, the last, the middle, there are many studies on the efficiency of this algorithm depending on the choice) and partitioning the array. Then, we put this element in the right place, the smaller values before the pivot and the greater values after it, until it is fully sorted, in a recursive way. We can see an example below (I used the middle element as the pivot):

```

60 void quick_sort(int* array, int smaller, int biggest)
61 {
62     if (smaller < biggest) {
63         /* pi is partitioning index, arr[p] is now
64         at right place */
65         int divide = partition(array, smaller, biggest);
66         quick_sort(array, smaller, divide - 1);
67         quick_sort(array, divide + 1, biggest);
68     }
69 }

```

**Heapsort** works in a similar way to the Selection Sort, previously explained, but it is more efficient. We find the minimum element (min) and place it at the beginning of the array, repeating the process until the array is sorted. The difference is that heapsort uses a heap data structure as a way to keep the unsorted part of the array, and finding elements in a heap is way faster than in a linear model, as seen below:

```

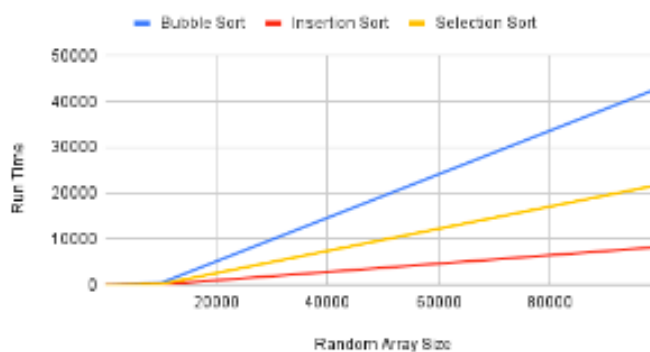
84 void heap_sort(int array[], int index) {
85     /* max heap */
86     for(int i = (index/2)-1; i <= 0; i--) {
87         heapify(array, index, i);
88     }
89     /* heap sort */
90     for (int i = index - 1; i >=0; i--) {
91         change(&array[0], &array[i]);
92         heapify(array, i, 0);
93     }
94 }
95 }

```

The related complexity for the average case of a random array of these algorithms has been studied during the Lab02 and Lab01 and can be seen through the graphs below. As we can see the fast algorithms perform way better than the slow algorithms, the slope difference from the slowest one (bubble sort) is very notable. That happens because there's a lot of operations needed to finish the sorting when compared to the others, so its slope in the graph is the steepest.

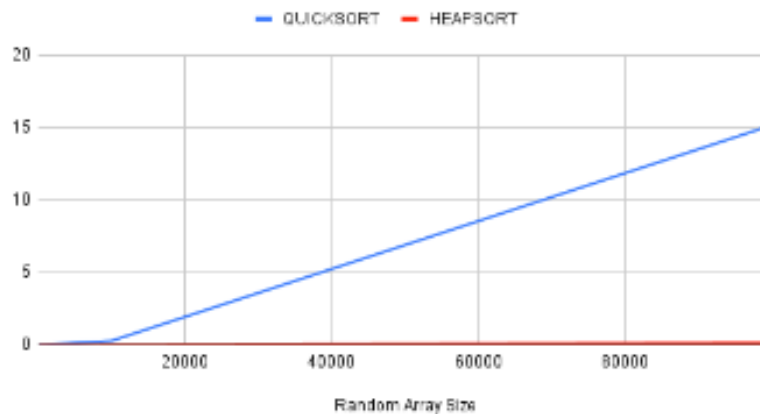
Comparing the insertion and the selection sort it's clear in the graph that Insertion sort has a better performance. That means that the insertion sort algorithm makes fewer comparisons than the other ones in the average case of a random generated list.

Bubble Sort, Insertion Sort and Selection Sort - Case 1



Taking a closer look at the difference between the fast algorithms, the heap sort performs, on average, better than the quick sort. An interesting thing to note is that, for smaller arrays, the time performance of both algorithms is nearly the same, as seen in the first quadrant of the graph 2.

QuickSort and HeapSort for a Random Sorted Array



3.

### Array and linked list.

(Every piece of code that is used in this question has been taken from my personal Github repository: <https://github.com/thaisstein/cprogramminglabs/>)

An **array** is a linear structure that contains elements stored in a continuous way (one after the other). That means that to insert the last element of the array of size  $n$  would take  $O(n)$  time. The array can be statically or dynamically allocated in the memory. Below is an example of how we fill an array that is dynamically stored in the memory, note that it follows a linear order since we go from the element 0 to the element  $\text{length}-1$  in the array.

```

7 void rand_fill(int** array, unsigned int lenght) {
8     int* temp = (int*)malloc(lenght * sizeof(unsigned int));
9     if (temp == NULL) {
10         exit(0);
11     }
12     time_t time_seed;
13     srand((unsigned) time(&time_seed));
14     for (int j = 0; j < lenght ;j++) {
15         temp[j] = rand()%100;
16     }
17     *array = temp;
18 }

```

A **linked list** is a structure that contains essentially "blocks" of elements, and each of these blocks contains a pointer that points to the next element stored. These blocks can contain anything, normally being a struct. For example, in the Lab 03 each list node (picture below) contains a word, the counting of this word and, obviously, the pointer to the next node. The first node is always a pointer and the last node is always NULL.

```

13  /* Defines the node */
14  typedef struct Node {
15      char *word;
16      int count;
17      struct Node *next; // Pointer to struct Node
18  } Node;
19  typedef Node * p_node;
20
21  /* Creates an empty list */
22  p_node create () {
23      return NULL;
24  }

```

They are similar since they store the data in a similar way since they're both linear. The difference is that, as we've seen, an array is continuously stored, meaning that it takes a "chunk" of memory space going from its first to its last element. An element on a linked list on the other hand, can be stored anywhere in the memory. Each of them have different advantages: to access a specific element in an array is much faster than in a linked list, whereas inserting and deleting elements on a linked list is way faster than in an array.

4. **Which non-linear data structures do you know? What are they useful for and how can they be represented ? Please, provide an example (or several of them) of a problem where a corresponding data structure is relevant.**

#### **Trees.**

Trees are a data structure that is organized non-linearly and contains nodes, such as the linked list, that store a reference to the other nodes, called "children", since they obey a hierarchical organization where there's a root node and every node that comes after it is a child node.

So, the parent node comes before a given node, the child node is its successor and the root node is the node that does not have a parent node.

It is really useful to store data in an efficient way once we can search, put and delete elements in the tree way faster than in the array or linked list due to its hierarchical organization. An example of using a tree that is really big is the Linux Kernel, that is organized in a Left Leaning Red Black Tree (<https://github.com/torvalds/linux>).