Tasks

Write two programs that perform in-place case-sensitive 1) ascending selection sort and 2) descending insertion sort on a list of words provided as command-line arguments.

Example Inputs and Outcomes

Upon executing `selectionsort (insertionsort) happy chicken orange cheese chase bank`, your program should output a sorted list of these words: `bank chase cheese chicken happy orange` (`orange happy chicken cheese chase bank`). Preferably each word is printed on its own line.

Goals

Through this project, you will become familiar with:

1. utilizing command-line arguments (`argc, argv`)
2. iterative sorting algorithms (selection and insertion sort)

Background

1 Command-Line Arguments

Whenever a program is executed, all of the strings separated by at least one whitespace within the same command are passed to this program. This includes the name of the command/program/executable itself. For example, if I run `python -c "a=5; print(a)"`, the program '`python`' receives a list of three arguments: '`python`', the name of the program itself, '`-c`', the flag indicating that python code will follow, and '`"a=5; print(a)"`', the code passed to `python` to execute.

The simplest way to access command-line arguments in C++ is to declare the parameters `int argc` and `char **argv` (or `char *argv[]`) in the `main` function definition. In every C++ program, the `main` function is the 'entry-point' where execution begins and these command-line arguments are the only arguments `main` can receive. Consider the short program (`args.cpp`) below. Upon running `./args`, the first argument in `argv` will always be '`./args`'!

```cpp
#include <iostream>

using namespace std;

int main (int argc, char **argv) {
    int i;
    for ( i = 0 ; i < argc ; i++ )
        cout << argv[i] << endl;
}
```

2 Iterative Sorting Preliminaries

Characters and Computers

When sorting a list of numbers, each element is ordered by its numerical value. However, when sorting a list of strings, how is each element compared using only the tools available to a computer? It has no sense of lexicographical ordering... right?

It turns out that there is a hardcoded lexicographical ordering for characters. Your computer (specifically YOURS, no one else's... trust me, I checked) represents characters with 'character codes' that map a literal character to a sequence of bits (often represented as bytes). The most common characters are within the first 128 integers, hence the `char` data type representing one byte or the first 255 Unicode characters, and these are your ASCII characters. Uppercase latin characters (A-Z) are represented by integers 65 to 90; lowercase latin characters (a-z) span the range 97 to 122; digit characters (0-9) span 48 to 57. The characters beyond the just one byte (first 256) have their codes determined by the Unicode standard which uses 4 bytes; the Unicode extension, usually used for emoji, uses 8 bytes. Note that you should write your own routine to compare two strings by iterating through each of their characters; there are utility functions in the `string` library if writing this proves too tedious.

Case-sensitivity is an important aspect of sorting words. Should you care that a file is named `Animal.png` vs. `animal.png`? This depends on your use case; for this project, you will care: sorting will be case-sensitive so that that 'A' is distinct from 'a'. This is easier to implement, since strings will be compared purely on the numerical value of their constituent characters' codes and you will not have to handle the special case of upper- and lower-case characters coinciding.

Algorithmic Complexity

An algorithm's ability to complete a task in a reasonable amount of time is measured by its 'time complexity'. Its ability to do so with efficient usage of memory is measured by its 'space complexity'. Often, computer scientists and mathematicians will use 'Big-O notation' to describe the worst-case performance of an algorithm given the size of the inputs. Consider two algorithms that search through a sorted array of N elements: *linear search*, iteratively passing over each element and comparing it to the desired value; *binary search*, recursively splitting the array into halves and continuing to search in the half that contains the element.

The *linear search*, in the worst case, will require as many as N comparisons to find the desired element. Thus, the big-O time complexity of *linear search* is $O(N)$ – note that we drop any constants or smaller-order terms. If the particular implementation of linear search actually took exactly $2N + 2$ comparisons in the worst case scenario of the desired element being the very last element of the sorted array, we would still say it has worst-case time-complexity $O(N)$.

The *binary search*, in the worst case, will require as many as $\log_2 N$ comparisons to find the desired element since we can only perform $\log_2 N$ splits on an array of N elements before the halves become 0 or 1 elements. Thus, the big-O time complexity of *binary search* is $O(\log N)$ – note that we drop the base of the log because two logs of any base are related by a multiplicative constant (log rules).

Lastly, iterative sorting algorithms work by iterating over an array multiple times to construct a complete solution (fully sorted array). Each algorithm iteratively applies a unique rule to obtain the partial solution of $n + 1$ elements from a partial solution of n elements and the remaining unsorted elements. I will denote the array of sorted elements as S and unsorted elements as U.

Note: these sorting algorithms can be done 'in-place', that is, by modifying the input array and not creating new arrays. You should not create any new arrays for these sorting algorithms.

Note: generally, when searching for algorithms on Wikipedia for more background, they will often have code snippets implementing them – avert your eyes!!

Hint: have a variable track the position that partitions the full array into the sorted and unsorted arrays.

3 Sorting Algorithms

Selection Sort

Start with a partially sorted array S of $|S| = n$ elements. To build S' of $|S'| = n+1$ elements, search the unsorted partition U for the appropriate element to follow the last element of S. Swap this optimal (smallest or largest) element for the element at position $n+1$. You are essentially building the desired sequence of elements by hand-picking which one comes next.

The big-O time complexity of this algorithm is $O(N^2)$ since for each of the N elements it must always search the entire unsorted partition of $O(N)$ elements for the optimal element.

Insertion Sort

Start with a partially sorted array S of $|S| = n$ elements. To build S' of $|S'| = n+1$ elements, search the sorted array S for the appropriate position to insert the first element of the unsorted array U. Insert the first element of U into the desired position and push the elements that follow over by one position. You are essentially inserting new elements one-by-one into an already-sorted array.

The big-O time complexity of this algorithm is $O(N^2)$ since, for each of the N elements it must, in the worst case, search the entire sorted partition of $O(n)$ elements for the correct insertion location.