Sorting

- *Sorting* means arranging a group of data in an order, for ex., ordering some words alphabetically, or some numbers from smallest to largest, or from largest to smallest.

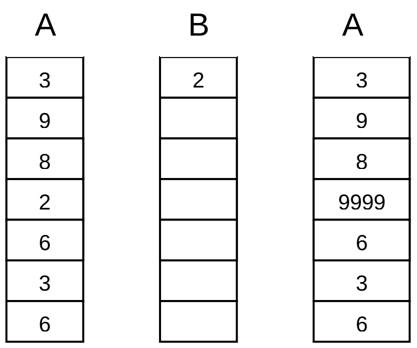
- Arranging numeric data in increasing order is named as sorting in *ascending* order, where arranging them in decreasing order is named as sorting in *descending* order. Arranging strings in alphabetical order is also named as sorting in *ascending* order.

- All sorting algorithms assume that the data to be sorted is already put into an array, and the number of them (thus the array size) is known.

Simple Selection Sort

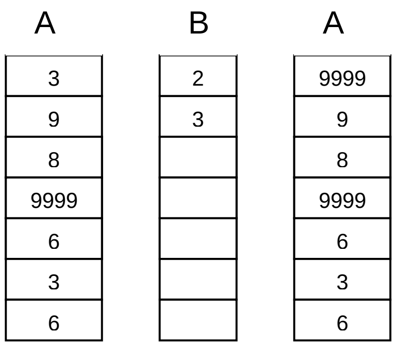
- This algorithm sorts the elements of an array and uses another array to store the sorted elements.

- If the array to be sorted is the A array, and you want to sort it in ascending order, and put the result in the B array, first find the position of the minimum element in A. The minimum is 2 and it is in position 3. Store the minimum in position 0 of the second array B, and put a very large value to the position where the minimum was found in A (position 3), so that this element will never be the minimum again.



First Pass (6 comparisons to find the min.)

- Then, in the second pass, find the position of the minimum element in the new A array (thus the second min. in the initial A). The minimum is 3 and it is in position 0. Store the minimum in position 1 of the second array B, and put a very large value to the position 0 of A.



Second Pass (6 comparisons to find the min.)

- Repeat the same operations until the B array is full, thus **n** times, where **n** is the size of the array (i.e., 7 times)

- We can summarize the steps of Simple Selection Sort algorithm as follows:

  1. Find the position of the minimum element of the given array
  2. Store the minimum value into the first empty position of the new array
  3. Assign a large value to the position where the minimum was found
  4. Repeat the same operation **n** times, where **n** is the size of the array

- Let's define a function for Simple Selection Sort. Notice that, we can define a separate function to find the position of the minimum element of an array. In fact, we defined that function last semester several times.

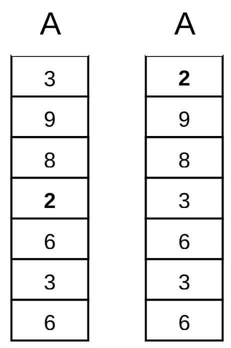
```
int min_pos(int ar[], int size) {
     int k, min, min_k;

     min = ar[0];
     min_k = 0;
     for (k = 1; k < size; k++)
         if (ar[k] < min)
         {
             min = ar[k];
             min_k = k;
         }
     return (min_k);
}

void simple_selection_sort(int A[], int B[], int n) {
     int k, ind_of_min;
     for (k = 0; k < n; k++)
     {
         ind_of_min = min_pos(A, n);
         B[k] = A[ind_of_min];
         A[ind_of_min] = 9999;
     }
}
```

- Notice that, this algorithm is very simple and easy to follow, but it is very inefficient, because it uses two arrays which increases memory allocation a lot, especially if we want to sort a very large array with thousands of elements. In addition, it is very slow, because it makes `n` passes, and makes `n-1` comparisons to find the minimum in each pass. Thus, it makes (6x7=) 42 comparisons to sort our A array.

- Another disadvantage is that the large (or the small) value to be put in A array may not be decided correctly everytime. It is easy to decide if you are sorting some data whose range is well defined, such as exam grades (101 or -1 would be proper values), but what about if you are sorting some data whose range is not known?

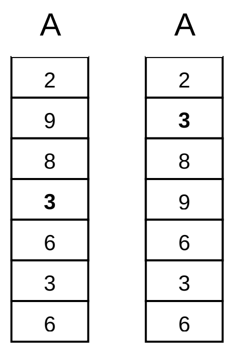
Selection Sort

- This algorithm is very similar to Simple Selection Sort algorithm, except that it uses the same array to store the sorted data. Assume we again want to sort our A array. First find the position of the minimum element in A. The minimum is 2 and it is in position 3. Then, exchange the first element and the minimum element.



First Pass (6 comparisons to find the min.)

- Then, in the second pass, find the position of the next minimum element, starting from the second element of the new A array. The minimum is 3 and it is in position 3. Exchange the second element and that minimum element.



Second Pass (5 comparisons to find the min.)

- Repeat the same operations until the minimum of last two elements are found, thus **n-1** times, where **n** is the size of the array (i.e., 6 times)

- We can summarize the steps of Selection Sort algorithm as follows:

  1. Find the position of the minimum element of the given array, starting from the top element
  2. If the top element is not the minimum element exchange the top element and the minimum element
  3. Move the top downwards one element
  4. Repeat the same operation **n - 1** times, where **n** is the size of the array

- Let's define a function for Selection Sort algorithm. We can again define a separate function to find the position of the minimum element of an array. We can not use the same function (`min_pos`), because this time our function should not always start to search for the minimum from the first element of the array, but from a different position in each pass. Thus, it needs one more parameter that shows where the operation should start from.

```
int min_pos_range(int ar[], int first, int last) {
     int k, min, min_k;
     min = ar[first];
     min_k = first;
     for (k = first + 1; k <= last; k++)
         if (ar[k] < min) {
             min = ar[k];
             min_k = k;
         }
     return (min_k);
}

void selection_sort(int list[], // array being sorted
                    int n)      // no. of elements to sort
{
         int top,// first element in unsorted sub array
         temp,  // temporary variable
         ind_of_min; // subscript of the smallest element

     for (top = 0; top < n - 1; top++)
     {
         ind_of_min = min_pos_range(list, top, n - 1);
         // Exchange elements at top and ind_of_min
         if (top != ind_of_min) {
             temp = list[ind_of_min];
             list[ind_of_min] = list[top];
             list[top] = temp;
         }
     }
 }
```

- We can also define another separate function to exchange two elements of an array and use it in our function.

```
void swap(
     int ar[],        // (input/output) given array
     int k, int j)    // indices of the elms to be exchanged
{
     int temp;        // temporary variable
     temp = arr[k];
     arr[k] = arr[j];
     arr[j] = temp;
}
...
         if (top != ind_of_min)
             swap(list, top, ind_of_min);
```

- Remember that we declared another `swap` function in our first lecture, which was exchanging values of two integer variables using pointers. Can we use it here? Yes, but when we are calling that function, we have to send the addresses of the array elements to be exchanged:

```
void swap(int *x, int *y)
{
     int temp;
     temp = *x;
     *x = *y;
     *y = temp;
}
...
         if (top != ind_of_min)
             swap(&list[top], &list[ind_of_min]);
...
```

- Notice that, this algorithm is also very simple, but it is more efficient than the Simple Selection Sort. First of all, it uses less memory space, because it does not need an extra array. In addition, it is faster, because it makes **n-1** passes, and makes one less comparison to find the minimum in each pass. Thus, it makes (6+5+4+3+2+1=) 21 (+ 5 for the `if` stmt before the `swap` call) comparisons to sort our A array, which is exactly half of the number of comparisons the previous algorithm did.

- Another advantage is that we don't need to decide any large (or small) value in this algorithm.

- However, it is still not a very good algorithm, because it is possible to sort the same array faster using some other sort algorithms.

Bubble Sort

- Compare the first two elements, and exchange them if they are out of order, thus move the smallest one to top. Then compare the new second element with the third one, and exchange them if they are out of order. Go on this operation, thus compare two adjacent elements and order them, until you reach to the end of the array.

A	A	A	A	A	A	A
3	3	3	3	3	3	3
9	9	8	8	8	8	8
8	8	9	2	2	2	2
2	2	2	9	6	6	6
6	6	6	6	9	3	3
3	3	3	3	3	9	6
6	6	6	6	6	6	9

First Pass (6)

- When the first pass is completed, notice that the largest element moves to the bottom of the array.

- Now, repeat the same operations, but decrease the number of comparisons, because you don't need to compare the last element with the element before it, since you are sure that it is the largest element.

A	A	A	A	A	A
3	3	3	3	3	3
8	8	2	2	2	2
2	2	8	6	6	6
6	6	6	8	3	3
3	3	3	3	8	6
6	6	6	6	6	8
9	9	9	9	9	9

Second Pass (5)

- When the second pass is completed, notice that the second largest element moves to the bottom of the array. Thus, the number of necessary comparisons in the third pass will decrease to 4.

- Therefore, we can say that for an array of **n** elements, the number of pairs of elements compared in a particular pass is **n - pass** where **pass** is the number of current pass, starting with 1 for the first pass.

- Therefore, we can summarize the steps of Bubble Sort algorithm as follows:

  1. Pass over the whole array comparing two adjacent elements
  2. If the second one is less than the first, exchange these two elements
  3. Repeat the same operation **n - 1** times, making **n – pass** comparisons in each pass

- Let's define a function for Bubble Sort:

```
void bubble_sort(int list[], int n)
{
      int pass, k;
      for (pass = 1; pass <= n – 1; pass++)
          for (k = 0; k < n – pass; k++)
              if (list[k] > list[k+1])
                  swap(&list[k], &list[k+1]);
}
```

- Notice that, this algorithm makes **n - 1** passes, and makes one less comparison in each pass. Thus, it makes (6+5+4+3+2+1=) 21 comparisons to sort our A array, which is exactly the same number of comparisons the Selection Sort algorithm did.

- We can modify the Bubble Sort algorithm so that it works faster than before. Notice that our array is sorted at the end of third pass, and no swap operations are done in the following passes. In fact, if no swap operation is done during a pass, it means that all elements are in the correct place, thus the array is sorted, and we can stop the operation in such a case.

```
void bubble_sort(int list[], int n)
{
     int k,
         pass,    // number of current pass
         sorted; /* flag to indicate whether sorting is
                    finished or not */
     pass = 1;
     do
     {
         sorted = 1; // assumes array is sorted
         for (k = 0; k < n - pass; k++)
             if (list[k] > list[k + 1])
             {
                 swap(&list[k], &list[k + 1]);
                 sorted = 0;
             }
         pass++;
     } while (!sorted);
}
```

- Notice that, although the first Bubble Sort algorithm made 21 comparisons to sort our array, this algorithm makes only (6+5+4+3=) 18 comparisons, because it will stop at the end of the fourth pass.

String Sorting

- Let's modify our `bubble_sort` function so that it sorts an array of strings.

- First of all, we need to rewrite the `swap` function to swap two strings of maximum `STR_LEN` characters.

```
void swap_str(char *str1, char *str2) {
     char temp[STR_LEN];
     strcpy(temp, str1);
     strcpy(str1, str2);
     strcpy(str2, temp);
}
```

- Now, we can rewrite the `buble_sort` function to sort a list of `n` strings of maximum `STR_LEN` characters. Thus, we can represent it as a two-dimensional array with `n` rows and `STR_LEN` columns.

```
void bubble_sort_str(char list[][STR_LEN], int n)
{
        int k,
        pass,   // number of current pass
        sorted; /* flag to indicate whether sorting is
                   finished or not */
     pass = 1;
     do
     {
        sorted = 1; // assumes array is sorted
        for (k = 0; k < n-pass; k++)
            if (strcmp(list[k], list[k + 1]) > 0)
            {
                swap_str(list[k], list[k + 1]);
                sorted = 0;
            }
        pass++;
     } while (!sorted);
}
```

Home Exercise:

Write a main that reads 5 words (each with maximum 20 characters) and displays them in alphabetically sorted order.

- An alternative approach for sorting a list of strings may be using an array of pointers. Initially, its elements will be initialized to the starting address of each string in the original array. Then, we will write a bubble sort function to sort that array, so that its elements will point to the sorted list.

```
#include <stdio.h>
#include <string.h>
#define STR_LEN 20

void bubble_sort_str_ptr(char *list[], int n) {
     int k,
        pass,   // number of current pass
        sorted; /* flag to indicate whether sorting is
                   finished or not */
     char *temp; // temporary pointer
     pass = 1;
```

```
        do {
            sorted = 1; // assumes array is sorted
            for (k = 0; k < n-pass; k++)
                if (strcmp(list[k], list[k + 1]) > 0) {
                    temp = list[k];
                    list[k] = list[k + 1];
                    list[k + 1] = temp;
                    sorted = 0;
                }
            pass++;
        } while (!sorted);
   }
   int main (void) {
        char words[5][STR_LEN]; // List of strings
        char *str_ptr[5]; // Pointer array
        int k;
        // Get the words
        for (k = 0; k < 5; k++) {
            printf("Enter Word %d: ", k + 1);
            gets(words[k]);
        }
        // Copy their addresses to pointer array
        for (k = 0; k < 5; k++)
            str_ptr[k] = words[k];
        // Sort the pointer array
        bubble_sort_str_ptr(str_ptr, 5);
        // Display the sorted list
        for (k = 0; k < 5; k++)
            printf("%s\n", str_ptr[k]);
        return (0);
   }
```

- Notice that, in the above program the original list will not change. The advantage of this method is that copying only pointers not complete arrays of characters executes faster.