Programming in C

Anubhav
2021-06-12
Version 2014

Preface

This is an old book that I am rewriting.

ToC

Sorting

Bubble Sort

Bubble Sort is one of the simplest sorting algorithms. Another simple one is selection sort, which makes more sense after finishing learning about bubble and selection sort, but till then, we will look into bubble sort and it's intricacies.

Sorting in bubble sort is done by comparing two adjacent elements and swapping them if need be, also this is done for each element; this way, the largest -or- smallest element bubbles up or down, thus the name.

Let's see the code in C.

```
// Code listing #1 : IDEONE: Ut7nZ0
// ----------------------------------------
void bubble_sort(int A[], int n) {
   int i, j, temp;

   for (i = 0; i < n; ++i) {
       for (j = 0; j < n - 1; ++j) {
           if (A[j] > A[j + 1]) {
               temp = A[j];
               A[j] = A[j + 1];
               A[j + 1] = temp;
           }
       }
   }
}
```

This code is very simple, but still for the completeness' sake, I am going to provide a commented version that will make this code... well, better!

```
// Code listing #2 : IDEONE : HnqSsm
// https : //gist.github.com/anonymous/0e30ee568342c6f25d189f4db43495a0
// ------------------------------------------------
/* Bubble Sort code listing #2,
* bubble_sort function
* this function receives an Array and count of elements in it.
* if count of elements is provided less than actual number of elements
* then, the array will be sorted up until n.
*/
void bubble_sort(int A[], int n) {
   int i, j, temp;

   /* outer loop for each element in the array
* this statement runs once, but it's content are run n times
*/
   for (i = 0; i < n; ++i) {
       /* inner loop for comparing current element with each other element
* this loop statement will run n times, but it's content will run n*n times
*/
       for (j = 0; j < n - 1; ++j) {
           /* comparison logic. If next element is smaller than current one, swap
them
* here we are not taking the user's choice of ordering into consideration, and
* we have decided to go for an ascending order.
*/
           if (A[j] > A[j + 1]) {
               /* standard swapping procedure,
* we could have opted for a function call here, but that would have been an
overkill,
*/
               temp = A[j];
               A[j] = A[j + 1];
               A[j + 1] = temp;
           }
       }
   }
}
```

See, the difference, the code above looks beautiful, is readable and unambiguous.

This code does same thing as code listed in listing #1; it just is a bit bloated and well readable too.

Now, since I mentioned earlier, let's take care of the order first of all.

```
// Code listing #3 : IDEONE : Qn3qhU
// ------------------------------------------
/* Bubble Sort code listing #3,
* bubble_sort function
* this function receive an Array, count of elements in it and order of sort
* if count of elements is provided less than actual number of elements
* then, the array will be sorted up until n.
*/
void bubble_sort(int A[], int n, int order) {
   int i, j, temp;

   for (i = 0; i < n; ++i) {
       for (j = 0; j < n - 1; ++j) {
           /* comparison logic based upon order.
           * if order is 0, it means ascending order.
           * else, it means descending order.
           */
           if (order == 0) {
               if (A[j] > A[j + 1]) {
                   temp = A[j];
                   A[j] = A[j + 1];
                   A[j + 1] = temp;
               }
           } else {
               if (A[j] < A[j + 1]) {
                   temp = A[j];
                   A[j] = A[j + 1];
                   A[j + 1] = temp;
               }
           }
       }
   }
}
```

Now, this code will sort based upon the order value we gave it.
If order is equal to zero, i.e. order == 0, it will sort in increasing order, otherwise, if that's not the case, it will sort in decreasing order.

Though, this is a good code, the purpose of this book is far from it. What do you think happens to the if(order == 0) { … } else { … } code?

I mean, how many times do you think it runs?

Well, to approximate, it will be run n*n times, that is n raised to the power 2. Now, if n is small like, 100, 1000 or 10000, it is okay, but if n is large, say a billion or a trillion, n*n would be a very

huge number.

One thing that I want to point out is that, nobody in their right frame of mind would set out to sort a billion numbered array. This kind of activity happens once.

Till, then let’s improve our code listing #3 and make that order condition-branch execute only once.

```
// Code listing #4 : IDEONE: Ai79Ul
// ---------------------------------------------
/* Bubble Sort code listing #4,
* bubble_sort function
* this function receive an Array, count of elements in it and order of sort
* if count of elements is provided less than actual number of elements
* then, the array will be sorted up until n.
*/
void bubble_sort(int A[], int n, int order) {
   int i, j, temp;

   /* since, I set up order conditional branching here, either if branch will
work;
* or else branch will work
*/
   if (order == 0) {
       /* when order is zero, it will do an ascending sort. */
       for (i = 0; i < n; ++i) {
           for (j = 0; j < n - 1; ++j) {
               if (A[j] > A[j + 1]) {
                   temp = A[j];
                   A[j] = A[j + 1];
                   A[j + 1] = temp;
               }
           }
       }
   } else {
       /* with order != 0, it will do a descending sort. */
       for (i = 0; i < n; ++i) {
           for (j = 0; j < n - 1; ++j) {
               if (A[j] < A[j + 1]) {
                   temp = A[j];
                   A[j] = A[j + 1];
                   A[j + 1] = temp;
               }
           }
       }
   }
}
```

All right, we did it. We have managed to save branch selection time during runtime by putting order branching outside the loop. But, this is no surprise, we have actually bloated our code.

We can still optimize our bubble sort. But first, Let's look into the theory behind bubble sort.

Suppose, you have 5 elements in an array.

5, 4, 3, 2, 1

And we want to sort it in ascending order. I have chosen the worst case scenario.

What will happen for the outer loop, let's call it loop-i?

Loop-i will run for 5 times, from i = 0 to i < 5, i.e. 0, 1, 2, 3 and 4.

What will happen to loop-j?

It will iterate over j 4 times each for each iteration of loop-i.

		5, 4, 3, 2, 1	
i = 0;	j = 0; 5 > 4, swap.	4, 5, 3, 2, 1	
	j = 1; 5 > 3, swap.	4, 3, 5, 2, 1	
	j = 2; 5 > 2, swap.	4, 3, 2, 5, 1	
	j = 3; 5 > 1, swap.	4, 3, 2, 1, 5	
i = 1;	j = 0; 4 > 3, swap.	3, 4, 2, 1, 5	
	j = 1; 4 > 2, swap.	3, 2, 4, 1, 5	
	j = 2; 4 > 1, swap.	3, 2, 1, 4, 5	
	j = 3; 4 < 5, no swap.	3, 2, 1, 4, 5	*
i = 2;	j = 0; 3 > 2, swap.	2, 3, 1, 4, 5	
	j = 1; 3 > 1, swap.	2, 1, 3, 4, 5	
	j = 2; 3 < 4, no swap.	2, 1, 3, 4, 5	*
	j = 3; 3 < 5, no swap.	2, 1, 3, 4, 5	*
i = 3;	j = 0; 2 > 1, swap.	1, 2, 3, 4, 5	
	j = 1; 1 < 3, no swap.	1, 2, 3, 4, 5	*
	j = 2; 1 < 4, no swap.	1, 2, 3, 4, 5	*
	j = 3; 1 < 5, no swap.	1, 2, 3, 4, 5	*
i = 4;	j = 0; 1 < 2, no swap.	1, 2, 3, 4, 5	*
	j = 1; 1 < 3, no swap.	1, 2, 3, 4, 5	*
	j = 2; 1 < 4, no swap.	1, 2, 3, 4, 5	*
	j = 3; 1 < 5, no swap.	1, 2, 3, 4, 5	*

This is actually what is happening in our bubble sort. The lines that are marked with an asterisk are the ones that are being run without any reason.

Yes, the * is called asterisk after a greek root aster which means star. Where do you think asteroids and astronomy and astrology came from? *TODO: make a footnote.*

Do you believe it? I didn't write all those lines, the following code did. I might have brushed up on the details a little.

Here, run this code.

Code listing #5 : IDEONE: Fv3L2d

```
/* Bubble Sort code listing #5
 * bubble_sort function
 * this function receive an Array, count of elements in it and order of sort
 * if count of elements is provided less than actual number of elements
 * then, the array will be sorted up until n.
 * also prints the iteration information
 */
void bubble_sort(int A[], int n, int order)
{
        int i, j, temp;

/* since, I set up order conditional branching here, either if branch will work;
 * or else branch will work
 */
if( order == 0 ){
/* when order is zero, it will do an ascending sort. */
                for(i = 0; i < n; ++i){
                        printf("i = %d,\n", i);
                        for( j = 0; j < n-1; ++j){
                                printf("\t\tj = %d, ", j);
                                if( A[j] > A[j+1] ){
                                        printf(" swap\t");
                                        temp = A[j];
                                        A[j] = A[j+1];
                                        A[j+1] = temp;
                                } else {
                                        printf(" no swap.\t");
                                }
                                printf("%d, %d, %d, %d, %d\n", A[0], A[1], A[2], A[3], A[4]);
                        }
                        printf("\n");
                }
        } else {
/* with order != 0, it will do a descending sort. */
                for(i = 0; i < n; ++i){
                        printf("i = %d,\n", i);
                        for( j = 0; j < n-1; ++j){
                                printf("\t\tj = %d, ", j);
                                if( A[j] < A[j+1] ){
                                        printf(" swap\t");
```

```
                                        temp = A[j];
                                        A[j] = A[j+1];
                                        A[j+1] = temp;
                                } else {
                                        printf(" no swap.\t");
                                }
                                printf("%d, %d, %d, %d, %d\n", A[0], A[1], A[2], A[3], A[4]);
                        }
                        printf("\n");
                }
        }
}
```

This tells us that we need to a) remove extra runs of the code, and b) get rid of code duplicacy.

We will look into code-duplicacy in the end of the book. Till then we are going to remove extra runs.

What do you observe when you read array values given above with respect to the variable i or loop-i?

Did you find it?

Try to. Look at **i** and then at the last iteration of loop-j's output.

If you still couldn't find it, let me tell you that, for every iteration of loop-i, the biggest number (in our case above), gets into its correct position.

So, we do not need to run loop-j for all numbers. We can run loop j, from 0 to n - 1 - i.

This will make sense.

Suppose you have 3, 2, 1.
After the first iteration of loop-i, we will have 2, 1, 3. I.E. biggest number in it's correct place.
Now, in the second iteration of loop-i, the termination condition for loop-j would be: j < 3 - 1 - 1; which is j < 1.

So, now, in the second iteration of loop-i, loop-j will iterate only once and will set the array up as , 1, 2, 3.

Now, in the third iteration of loop-i, loop-j termination condition will be j < 3 -1 -2; since i == 2, thus j < 0, which will get us out of the inner loop right then and there.

Let’s see what this little change of code does to our program and it’s output.

I am not going to print all of this code, because change is too minimal with respect to the whole code.

Code listing #6 : IDEONE: xxQTAA

What we need changed from code listing #5 is :

for(j = 0; j < n -1 -i; ++j)

and this will be enough. Let's see the code listing #6 output.

OUTPUT:

```
                              5, 4, 3, 2, 1
i = 0,          j = 0,  swap  4, 5, 3, 2, 1
                j = 1,  swap  4, 3, 5, 2, 1
                j = 2,  swap  4, 3, 2, 5, 1
                j = 3,  swap  4, 3, 2, 1, 5
i = 1,          j = 0,  swap  3, 4, 2, 1, 5
                j = 1,  swap  3, 2, 4, 1, 5
                j = 2,  swap  3, 2, 1, 4, 5
i = 2,          j = 0,  swap  2, 3, 1, 4, 5
                j = 1,  swap  2, 1, 3, 4, 5
i = 3,          j = 0,  swap  1, 2, 3, 4, 5

i = 4,
```

See that, the fifth iteration of loop-i is completely rendered inert.

We have successfully optimized our code. There are no empty, meaningless iterations left.

But, there are meaningless variables left. If someone decides to read our code, what will an "i" tell him/her? What meaning will "j" convey?

Nothing.

So what we are going to do is name our variables a little bit appropriately. We are also ditching order for now, we will come back to that later.

So, let's fork our code from code listing #1, and apply our little improvement from code listing #6

```
// Code listing #7 : IDEONE: kVQhdC
// ------------------------------------------
void bubble_sort(int jumbled[], int count) {
   int pass, this, tmp;

   for (pass = 0; pass < count; ++pass) {
       for (this = 0; this < count - 1 - pass; ++this) {
           if (jumbled[this] > jumbled[this + 1]) {
               tmp = jumbled[this];
               jumbled[this] = jumbled[this + 1];
               jumbled[this + 1] = tmp;
           }
       }
   }
}
```

Till now, we have been looking at the worst possible cases, i.e. jumbled numbers in descending order when we need to sort them in ascending order and jumbled numbers in ascending order when we need to sort them in descending order.

For N numbers in the worst case to be sorted, we need N-1 passes. But what if numbers are already sorted or almost already sorted.

Let’s check output of code listing #7 for {1, 2, 3, 4, 5} and { 2, 1, 3, 4, 5 }

```
// Code listing #8 : IDEONE: H8GNjY
// ------------------------------------------
void bubble_sort(int jumbled[], int count) {
   int pass, this, tmp;

   for (pass = 0; pass < count; ++pass) {
       printf("pass #%d\n", pass);
       for (this = 0; this < count - 1 - pass; ++this) {
           printf("currently at :%d, ", this);
           if (jumbled[this] > jumbled[this + 1]) {
               printf("swap.");
               tmp = jumbled[this];
               jumbled[this] = jumbled[this + 1];
               jumbled[this + 1] = tmp;
           } else {
               printf("no swap.");
           }
           printf("\t%d, %d, %d, %d, %d\n", jumbled[0], jumbled[1], jumbled[2],
jumbled[3], jumbled[4]);
       }
   }
}
```

OUTPUT

```
pass #0
currently at :0, no swap.       1, 2, 3, 4, 5
currently at :1, no swap.       1, 2, 3, 4, 5
currently at :2, no swap.       1, 2, 3, 4, 5
currently at :3, no swap.       1, 2, 3, 4, 5
pass #1
currently at :0, no swap.       1, 2, 3, 4, 5
currently at :1, no swap.       1, 2, 3, 4, 5
currently at :2, no swap.       1, 2, 3, 4, 5
pass #2
currently at :0, no swap.       1, 2, 3, 4, 5
currently at :1, no swap.       1, 2, 3, 4, 5
pass #3
currently at :0, no swap.       1, 2, 3, 4, 5
pass #4
```

So as we can see, though there is no swapping occurring, loops are still going about it. Though we have prevented carnage due to this < count -1 -pass; but, still it is bleeding.

What if we look for the actions taken in the last iteration of the loop-i, or last pass of loop-i, frankly, they are the same things, just different names.

Let's create a state named swapped. A state in programming is just a variable with a value. As a variable can have any value, those values can be called states that your variable and by extension program is in.

Before going to code, let's think over this state variable called swapped first. It can be integer for our purpose, an integer is 4 Bytes on most modern (circa 2014) machines. It could have been a character which is 1 Byte (for an ASCII character). That way, we can save 3 Bytes.

And there's a reason why I used a capital B with Bytes above: B means Bytes, b means bits. Most people confuse this or don't know the difference to begin with, which creates innocent confusion; confusing confusion nevertheless.

int swapped = 1; it will be for our code.

If you are thinking about those 3 Bytes, go change it to a char yourself and save, by all means, those 3 Bytes.

you can use char swapped = ‘y’;

Ok. Now, this much is clear that every time we swap two elements because our condition met, we have to turn on the swapped state. A turned on swapped state would mean for our condition testing that our code swapped elements.

Now, Where do we check it?

Should we check it inside the loop-j or inner loop?

Well, the inner loop just compares adjacent elements one at a time, per iteration. There may be a case, when our compare condition doesn’t match and we quit our inner loop because it did not turn on the swapped state.

What about the outer loop?

Well we can put it there. Then turn the state off. Run the inner loop, hoping at least one element will get swapped. If it does, we will throw a switch and the swapped state variable will be turned on, and if it doesn't then the swapped state variable will stay off.
If it was turned on, we move to the next pass, there we throw the state off again and so on.
If it stays turned off, we get out of the outer loop and know for sure that our array is actually sorted.

Let’s see the code and output for verification.

Code listing #9 : IDEONE: itMWcV

```
void bubble_sort(int jumbled[], int count) {
   int pass, this, tmp, swapped = 1;

   for (pass = 0; pass < count && swapped; ++pass) {
       swapped = 0;
       printf("pass #%d\n", pass);
       for (this = 0; this < count - 1 - pass; ++this) {
           printf("currently at :%d, ", this);
           if (jumbled[this] > jumbled[this + 1]) {
               swapped = 1;
               printf("swap.");
               tmp = jumbled[this];
               jumbled[this] = jumbled[this + 1];
               jumbled[this + 1] = tmp;
           } else {
               printf("no swap.");
           }
           printf("\t%d, %d, %d, %d, %d\n", jumbled[0], jumbled[1], jumbled[2],
jumbled[3], jumbled[4]);
       }
   }
}
```

Notice the bold code, this is all that changed from code listing #8 and now check output.

Output

pass #0
currently at :0, no swap. 1, 2, 3, 4, 5
currently at :1, no swap. 1, 2, 3, 4, 5
currently at :2, no swap. 1, 2, 3, 4, 5
currently at :3, no swap. 1, 2, 3, 4, 5

Notice how only the first pass is run. Next pass is not required as there are no changes to make.

Now, let's check for semi-sorted input.

for int jumbled [5] = { 2, 1, 4, 3, 5 }; Output would be:

pass #0
currently at :0, swap. 1, 2, 4, 3, 5
currently at :1, no swap. 1, 2, 4, 3, 5

currently at :2, swap. 1, 2, 3, 4, 5
currently at :3, no swap. 1, 2, 3, 4, 5
pass #1
currently at :0, no swap. 1, 2, 3, 4, 5
currently at :1, no swap. 1, 2, 3, 4, 5
currently at :2, no swap. 1, 2, 3, 4, 5

Note how in the first pass, the array is completely sorted till end, but the next pass had to run anyway, to meet the condition where there was no swapping in the entire pass and then no further pass ran.

Now we can say that we have optimized our bubble sort to a good extent.

Here's the gist of things we have done till now.

1. Bubble sort program
    Very crude but accurate.
2. Well commented bubble sort program
    Comments tell the reader what the program is about. And if a programmer is coming to his own code after months or years, good comments help him or instruct him too.
3. Order of sorting
    We got side tracked, and went to take users' choice of sort-order into consideration.
4. Order of sorting code optimized
5. Printed the iteration information as bubble sort progresses with the input
6. Optimized bubble sort by reducing the amount of inner iterations with respect to passes.
7. Self explanatory code
    For the first time after 2 we chose to make code better by altering the presentation. Code is self-explanatory and optimized.
8. Evidence that our code though works well for the worst case, is actually pretty bad for the best case.
    Irony is that, what's best case is supposed to be for our code, is actually the worst case till now.
9. Enter swapped state variable
    How we stop our code from running unnecessarily by introducing a state variable and keeping track of it's on-ness.

Phew.

Till now, we have been bubbling one number towards one side. We bubbled the greatest number to the right while sorting in ascending order. Smaller numbers in that case stayed almost where they were. So, in effect, the number we wanted to bubble went fast through an array of elements towards its rightful place, and the small numbers swayed a little in each pass. What we can do now to increase the speed of bubbling of small numbers is put another loop for each pass.

```
loop-i {
    loop-j  {}
    loop-k {}
}
```

loop-j will take care of the biggest number per pass, and loop-k will take care of the smallest number per pass.

Let's fork our code listing #9.

Code listing #10 : IDEONE: PQxBEI

```
#include<stdio.h>

#define SIZE 20

void bubble_sort(int A[], int n);

void Print(int A[]);

int main() {
   int A[SIZE] = {3, 2, 5, 2, 7, 9, 7, 0, 8, 0, 5, 2, 1, 3, 4, 5, 6, 7, 8, 0};
   int i;
   printf("Printing original array: \n");
   Print(A);
   bubble_sort(A, SIZE);
   printf("Printing sorted array: \n");
   Print(A);
   return 0;
}

void bubble_sort(int jumbled[], int count) {
   int pass, this, tmp, swapped = 1;

   for (pass = 0; pass < count && swapped; ++pass) {
       Print(jumbled);
       swapped = 0;
       for (this = 0; this < count - 1 - pass; ++this) {
           if (jumbled[this] > jumbled[this + 1]) {
               swapped = 1;
               tmp = jumbled[this];
               jumbled[this] = jumbled[this + 1];
               jumbled[this + 1] = tmp;
           }
       }
       for (this = count - 1 - pass; this > 1; --this) {
           if (jumbled[this] < jumbled[this - 1]) {
               swapped = 1;
               tmp = jumbled[this];
               jumbled[this] = jumbled[this - 1];
               jumbled[this - 1] = tmp;
           }
       }
   }
   Print(jumbled);
}

void Print(int A[]) {
   int i;
```

```
    for (i = 0; i < SIZE; ++i) {
        printf(" %d ", A[i]);
    }
    printf("\n");
}
```

Notice that again we are meeting code duplication which is a really bad thing in programming, but here, we have cut the number of total iterations that are required to get sorting done.

Performance Data.
For performance related discussion move to appendix. There will be something there.

Generic sorting.
This till now was the integer swap. If we want string and other objects swapped, we need to have a generic approach.

I think that you need to read another book that I am writing. You can of course read online or hack away yourself in the meantime.

Selection Sort

Selection Sort is another simple sorting mechanism. It sorts things almost like bubble sort, but without the penalty of moving things unnecessarily. Like bubble sort, the objective is to find the greatest -or- smallest number and put it at the rightful place.

But while bubble sort moves elements every time the comparison condition is true, selection sort does not.

This makes selection sort marginally better; and the algorithm kind of looks a bit mature, it's not entirely a brute force that bubble sort algorithm is. But other things are equal, like you need two loops, one nested in another, and comparison condition.

In the end it boils down to the fact that bubble sort is too much movement and selection sort is selection. That's it.

Let's see an example that people usually in their college end up writing. This is an example of bad coding as there will be no self-explanation, other than that code works.

```
// Code listing # 11: IDEONE: dIf1ye
// ------------------------------------------------
void selection_sort(int A[]) {
   int i, j, min, temp;

   for (i = 0; i < SIZE - 1; ++i) {
       min = i;
       for (j = i + 1; j < SIZE; ++j)
           if (A[min] > A[j])
               min = j;

       temp = A[min];
       A[min] = A[i];
       A[i] = temp;
   }
}
```

This is as I said above: bad coding. Although i, j, k make good names for trivial loops, these loops are not that trivial. For instance, try to start the inner loop with j = 1, I dare you. Things won't make sense there. A trivial loop is an iterative construct N times, that's it. But when algorithms depend upon some particular values of an index, it is not anymore trivial. Non-trivial loops should have a meaningful index.
So, I leave it up to you to rename i as pass, and j as this or current, and so on.

A little bit of trivia.

Question: ***write a piece of code in C++, which is valid C code but not valid C++ code.***
Answer: Any piece of code that contains added keywords in C++ is valid C code but invalid C++; given that code has keywords in non-C++-meaningful ways.

For example, use of this. We have been doing that in our code for a while. this is a special pointer in C++, this is enough to break C++ code compilation process. Others could be virtual, class etc.

Why this question? Because it is usually remarked that C++ is backward compatible. It's not.

IDEONE: 8rQuxd

Let's move on to our assignment at hand. We have to understand selection sort. It goes like this:

1. Go through all elements picking one index at a time but last.
    loop-i
2. Name this index min_index.
    min = i
3. Again, go over all the elements that are right to the current mid_index element.
    loop-j
4. Compare them. If the element is smaller, update min_index.
    if A[min] > A[j]
5. When out of the inner loop, swap the elements.
    element swapping code

This might not be the best algorithm or best description of it for selection sort. But it is enough to get us going for now.

So, if you read point 5, you will find that the swapping of elements is done regardless of the fact that there was even a change in the index. min_index is set up every time control enters a pass. If the inner loop finds the comparing condition to be true, it updates the min_index. But what if the smallest number was at the index of current pass, then it would make no sense to swap, because in effect we will be doing nothing.

So what we can do is that we can put a condition as, if min_index is not equal to current pass index value, we will swap otherwise they are equal and don't need to be swapped.

```
// Code listing # 12: IDEONE: nA4pDy
// ------------------------------------------------
void selection_sort(int jumbled[]) {
   int pass, this, min_index, tmp;

   for (pass = 0; pass < SIZE - 1; ++pass) {
       min_index = pass;
       for (this = pass + 1; this < SIZE; ++this) {
           if (jumbled[min_index] > jumbled[this])
               min_index = this;
       }
       if (min_index != pass) {
           tmp = jumbled[min_index];
           jumbled[min_index] = jumbled[pass];
           jumbled[pass] = tmp;
       }
   }
}
```

For following 3 cases:
1. Good case, where elements are nearly sorted.
2. Random case.
3. Bad case, where elements are in opposite order.

For good cases, our condition will make more conditional checks than it would make swaps. Thus we will end up saving swapping time.

For bad cases, our condition will make almost equal checks as it would swap, there, we will have a performance hit. Instead of just blindly swapping, we will be first checking if we should swap and then we will swap. Double work.

For random cases, it will be fifty-50.

Moving on.

Remember how in bubble sort, how we sorted things out in half the step by introducing another loop per pass? Same thing here, all we have to do is select the maximum of them all too.

So let’s do this.
TODO: later.

Read 15. Greatest of 3 numbers next.

Insertion Sort

Recursion

Tower of Hanoi

Fibonacci series

Current number (N) is the sum of last (N-1) and last to last (N-2) numbers; and the first two numbers are 0 and 1.

Fn = Fn-1 + Fn-2
F0 = 0
F1 = 1

This looks like a recursive function.

F2 would be F1+F0, i.e F2 = 0 + 1 = 1
F3 = 2
F4 = 3
F5 = 5 and so on.

```
// Code listing #19: IDEONE : Tc1EAq
// ------------------------------------------------
int fibonacci_numbers(int nth) {
   int n, n1, n2;
   if (nth <= 1) {
       return 1;
   } else {
       n1 = fibonacci_numbers(nth - 1);
       n2 = fibonacci_numbers(nth - 2);
       n = n1 + n2;
       printf("%dth number (%d + %d =) %d.\n", nth, n1, n2, n);
       return n;
   }
}
```

This code does exactly what we asked it to do. But there's a catch, since it is the theme of this book, it is very redundant and the output is quite perplexing. A little better approach would be to generate the nth number and get it returned, then print it in the calling function.

Code listing #20: IDEONE : faHsrZ

```c
#include <stdio.h>

int fibonacci_number(int nth);

int main(void) {
   int i = 0;
   for (i = 0; i < 10; ++i) {
       printf("Fibonacci #%d = %d.\n", i, fibonacci_number(i));
   }

   return 0;
}

int fibonacci_number(int nth) {
   if (nth <= 1) {
       return 1;
   } else {
       return fibonacci_number(nth - 1) + fibonacci_number(nth - 2);
   }
}
```

But this is all good presentation, still not good use of processor's cycles. If we check how many times our recursive function is called, the answer would be too many, even for a small place like 10th position.

Code listing #20b: IDEONE : jo6WTc

```
int fibonacci_number(int nth) {
   static int called = 0;
   ++called;
   printf("called: %d\n", called);
   if (nth <= 1) {
       return 1;
   } else {
       return fibonacci_number(nth - 1) + fibonacci_number(nth - 2);
   }
}
```

for even the first 10 numbers, the recursive function calls are too many; for 100, it would be unacceptable.

Though we can still optimize away return statements.

Code listing #20c

```
int fibonacci_number(int nth) {
   static int called = 0;
   ++called;
   printf("called: %d\n", called);
   return nth > 1 ? fibonacci_number(nth - 1) + fibonacci_number(nth - 2) : 1;
}
```

So what we can do is create an array and put our numbers there. Then, we want them, just extract from that array.
This algorithm would go like this:

1. Create and Initialize fibonacci number array with 0s and first two elements would be 1.
2. for n-th fibonacci number, check if that entry is not 0, if it is not, return that value, if it is call recursively with n-1.

Code listing #21b: IDEONE : Oc3Vtp

```
int fibnums[100] = {1, 1, 0};

int fibonacci_number(int nth) {
   static int called = 0;
   int n1;

   ++called;
   printf("called: %d\n", called);

   if (nth < 100 && fibnums[nth])
       return fibnums[nth];

   n1 = fibonacci_number(nth - 1);
   fibnums[nth] = n1 + fibnums[nth - 2];
   printf("%d\n", fibnums[nth]);
   return fibnums[nth];
}
```

Call this function with 29 or any number for that matter, and you will find that the number of times this recursive function executes is of the order of N. This recursive function has become iterative from an execution point of view.

This technique is called memorization.

Only problem with this code is that it requires you to have a memory or cache or basically a value-store or an array, where you put your values as soon as you calculate them and retrieve them in the future as need be.

What else can we do? Can we still improve our fibonacci generation? The code that we wrote above is best if you want to access the fibonacci numbers randomly and position wise. For example, in the above code, if you want to know what the number is at 13th position, all you would need to do is calculate fibnums[13] or fibnums[12], depending upon how you see your numbers in the array.
But what if you don't want random access to fibonacci numbers, what if all you want is print the damn numbers or feed an outgoing pipe or just generate and collect into an array yourself.

Also note that, in our code listing #21b, we are able to get the order of N, if we call the biggest number or maximum fibonacci number we want, to keep the number of recursive calls small. But if you put #21b in an iterative loop and starting with 0, you will find that the number of calls is still more, not high.

We can save ourselves from that. But this code wouldn’t be recursive.

Code listing #22: IDEONE : WQDC1D

```c
#include <stdio.h>

int next();

int fibnums[100] = {1, 1, 0};

int main(void) {
   int i = 0;
   for (i = 2; i < 10; ++i) {
       fibnums[i] = next();
       printf("Fibonacci #%d = %d.\n", i, fibnums[i]);
   }
   for (i = 10; i < 29; ++i) {
       fibnums[i] = next();
       printf("Fibonacci #%d = %d.\n", i, fibnums[i]);
   }
   return 0;
}

int next() {
   static int n1 = 1, n2 = 1;
   int n;
   n = n1;
   n1 = n2;
   n2 = n = n + n1;
   return n;
}
```

We have used the property of static numbers, that they remember their values across multiple invocations. We just remembered what the last two numbers were and were able to generate the current number. Also, we are memorizing numbers in an array but outside our primary function, so this is strictly not memorization. But what we can do is have a static fibonacci number generating function too which will use memorization properly. So, if you want fibonacci numbers in a row use next() and if you want to say n-th fibonacci number use the recursive one.

Problems

Greatest of 3 numbers.

Suppose you are given two variables A and B, and you are asked to find the bigger of them. Easy-peasy.

```
if(A > B)
        return A;
else
        return B;
```

But if you are given 3 variables A, B and C and then you are asked to find the biggest of them, then the above logic will not work in an elegant manner.

```
if(A > B && A > C)
        return A;
else if(B > A && B > C)
        return B;
else
        return C;
```

This is working code, but it is a bit ugly. What happens if I give you 4 numbers or 5?

Let’s revisit our code.

Code listing #14 : IDEONE: HqaL6i

```
int bigger(int A, int B) {
   if (A > B)
       return A;
   else
       return B;
}
```

Notice how, if two numbers are the same, we are returning the second number. Since we are passing parameters by value we don't much care as the values are the same. But if we pass by reference, then it will matter, as to what we are actually returning, first parameter's reference or second's.

If we call this function as int max = bigger(A, B); where A and B are two variables with integer values, we will get a maximum of two. You probably are thinking that this is trivial code, why does it require a mention here?
Well, this is trivial code, and it is really a lego piece like trivial: small and possesses extensibility. Now, if I ask you to find the bigger of three numbers, this function is good enough for that purpose.

int maxof3 = bigger(A, bigger(B, C));

See. Now four.

int maxof4 = bigger(bigger(A, B), bigger(C, D));

If you are thinking that it looks ugly, or
If you are thinking that it is not scalable, know this: You are right.

This method is not scalable at all. This is a bit like daisy-chain. But then that's all of it. It is not elegant at all.

Code listing #15 : IDEONE: XJB49B

```
int biggest(int A, int B, int C) {
   if (A > B && A > C)
       return A;
   else if (B > A && B > C)
       return B;
   else
       return C;
}
```

But this code is as limited to scalability as code listing #14. Try finding biggest of four or five now.

Well there is a better way.

Code listing #16 : IDEONE: lBArRy

```
int biggest(int A, int B, int C) {
   int max = A;
   if (B > max)
       max = B;
   if (C > max)
       max = C;
   return max;
}
```

This code at least doesn’t look hideous. But is it scalable? No. Not yet. We have to change our approach. Instead of passing integers separately, pass an array of numbers and ta da.

Code listing #17 : IDEONE: SsCIgH

```
int biggest(int A[], int count) {
   int max, i;

   for (max = i = 0; i < count; ++i) {
       if (A[max] < A[i])
           max = i;
   }
   return A[max];
}
```

This code is scalable. You want to find a maximum of two numbers, done. Three numbers? Okay. A hundred numbers? Sure, bring it on. All you have to do is put those numbers in an array, and you are done.

There's another way. And that way will prove to be better than this one.

Code listing #18 : IDEONE: ifXn8E

```
int biggest(int count, int A, ...) {
   int max = A, i = 0, num;
   va_list arglist;
   va_start(arglist, A);
   for (max = num = A, i = 0; i < count; ++i) {
       num = va_arg(arglist,
       int);
       if (num > max) {
           max = num;
       }
   }
   va_end(arglist);
   return max;
}
```

This code is elegant, scalable and accurate. There is nothing better you can do to find the greatest of N numbers.

If someone asks you to find the greatest of 3 numbers, and you come up with this code, you are sure to win him or her over.

The five new things in this code are :

1. Ellipsis i.e. '. . .'
three dots together without space between them in the argument list is known as ellipsis. It means that more variables than provided in the argument list can be passed. This is for the cases where we cannot apprehend beforehand the number of arguments the programmer is going to use. Usually this kind of code is available in libraries and other programmers that are going to use that library will make the calls to that library. For example you want to add a bunch of numbers, you can provide an array, as a single argument or you can provide an ellipsis.

2. va_list: this helps in declaring that we have a variable-list that will hold the variables.

3. va_start: we have to initialize an argument list. We just created an variable argument list variable, and using the last named variable in the argument list , we can tell our code where to start working from.

4. va_arg: we pass our arglist variable and type of the data structure so that we can get the correct value.

5. va_end: this is where we tell our code that we are not going to use arglist anymore.

Prime Number Generation

Primes are the numbers that are divisible by 1 and self. Any other pair of numbers cannot divide them. 2, 3, 5, 7, 11, 13, 17, 19… are prime numbers.

A naive algorithm for finding if N is prime or not.

1. Divide the number from 2 to N-1, if you find an integer that divides N, the number is not prime.

Code listing #23: IDEONE : mQNmKB

```
int naiveIsPrime(int n) {
    int i, isPrime = 1;
    for (i = 2; i < n && isPrime; ++i) {
        if (n % i == 0) {
            isPrime = 0;
        }
    }
    return isPrime;
}
```

There is nothing wrong with this code listing. This is quite what is the definition of prime number. But from the optimization point of view, this is a weak contender.

We have seen a few tricks till now that we might apply here such as Memorization, or we can use mathematics to our advantage.

Look at the ridiculousness of our code, we are trying to figure out if N is divisible by N-1. What number is N then?

If you really know mathematics, 2 and 0 are two such contenders. 0 is divisible by -1 and 2 is divisible by 1. But is there any other pair?

So, read this carefully.

Suppose you want to know what is the maximum number that divides the number N. It has to be N/2. Reason is simple, 2 is the smallest prime factor a number could have and thus N/2 will be the greatest factor for the number N.

Now, if a number is not divisible by 2, then what can possibly be it's greatest factor? The answer to that question is N/3.
Again, 3 is the smallest prime factor after 2, and thus N/3 becomes the greatest factor.

Let's see some examples.

Suppose we want to know whether 100 is a prime factor. All we need to do is see if it is divisible by 2, it is so, it is not a prime number. But before beginning to search for the divisibility, we should have set the ceiling of the for loop to the 100/2 = 50. If any number uptil 50 can divide 100, then it won't be a prime number.

So, we established that 100 is not a prime.

What should have been the ceiling if the number was 99?
Well we will start with 2 and uptil 99/2, which in computer algebra, 48. So if 99 is divisible by any number uptil 48, it won't be prime.
So the ceiling is 48.
Does 2 divide it? No. So, the next number is 3. Should the ceiling be 48 still? No. Ceiling now should become 99/3 = 33.
Does 3 divide 99? Yes. So 99 is not a prime either.

Let's take 101 for our next case.
Ceiling will be set to 101/2 = 50.
Does 2 divide it? No. Ceiling updated to 101/3 = 33.
Does 3 divide it? No. Ceiling updated to 101/4 = 25.
Does 4 divide it? No. Ceiling updated to 101/5 = 20.
Does 5 divide it? No. Ceiling updated to 101/6 = 16.
Does 6 divide it? No. Ceiling updated to 101/7 = 14.
Does 7 divide it? No. Ceiling updated to 101/8 = 12.
Does 8 divide it? No. Ceiling updated to 101/9 = 11.
Does 9 divide it? No. Ceiling updated to 101/10 = 10.
Does 10 divide it? No. OOPS. Ceiling was 10. This is a prime number.

Did you notice what happened there? Instead of checking naively for divisibility upto 101-1 = 100, we checked up to 10 and got our result.

Isn't that an optimization?

So, how will we go ahead and write code for that?

Code listing #24: IDEONE : jZ56ch

```
int notSoNaiveIsPrime(int n) {
   int i, isPrime = 1;
   for (i = 2; i <= nextCeiling(n, i) && isPrime; ++i) {
       if (n % i == 0) {
           isPrime = 0;
       }
   }
   return isPrime;
}

int nextCeiling(int n, int i) {
   return n / i;
}
```

nextCeiling() is not required, just n/i would have done the job, but a function call is much more dramatic.

All we are doing is decreasing the number of divisions we need to perform to verify whether a number is prime or not.

So far so good. For 25013, only 157 iterations would run instead of 25012.

Let’s see if we can improve it further.