Insertion sort (C)

From LiteratePrograms

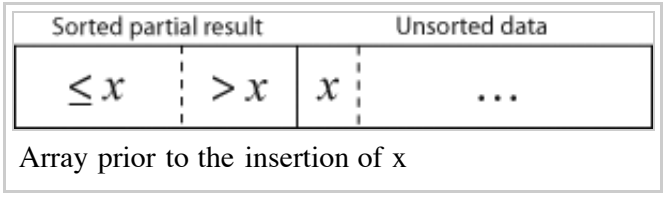
> **Other implementations**: ACL2 | **C** | C, simple | Eiffel | Erlang | Forth | Haskell | Io | Java | Lisp | OCaml | Python | Python, arrays | Ruby | Scala, list | Smalltalk | Standard ML | Visual Basic .NET

This is a simple example of the insertion sort or "bin sort" sorting algorithm, written in C. Since this implementation is intended to be as simple and easy to understand as possible, rather than particularly useful in practice, we will focus on sorting a simple array of *int*s of a given length.

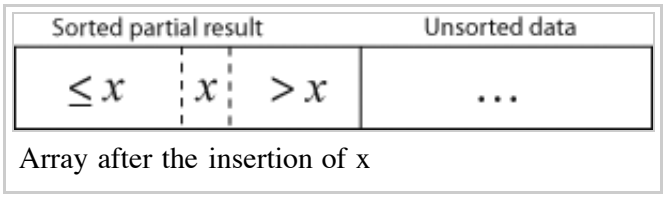
Main algorithm

When sorting an array with insertion sort, we conceptually separate it into two parts:

- The list of elements already inserted, which is always in sorted order and is found at the beginning of the array;
- The list of elements we have yet to insert, following.



Array prior to the insertion of x



Array after the insertion of x

In outline, our primary function looks like this:

```
<<insertion_sort>>=
/* Sort an array of integers */
void insertion_sort(int a[], int length)
{
  int i;
  for (i=0; i < length; i++)
  {
      insert a[i] into sorted sublist
  }
}
```

Here, the index i represents both the index of the next element to insert and the length of the sorted sublist constructed so far.

To insert each element, we need to create a hole in the array at the place where the element belongs, then place the element in that hole. We can combine the creation of the hole with the searching for the place by starting at the end and shifting each element up by one until we find the place where the element belongs. This overwrites the element we're inserting, so we have to save it in a variable first:

```
<<insert a[i] into sorted sublist>>=
/* Insert a[i] into the sorted sublist */
int j, v = a[i];

for (j = i - 1; j >= 0; j--)
{
    if (a[j] <= v) break;
    a[j + 1] = a[j];
```

```
}
a[j + 1] = v;
```

On average, we move about half the elements of the already-sorted sublist to insert each element. Consequently, the total number of assignments on average to sort *n* elements is:

$$\sum_{i=0}^{n-1} i/2 = \frac{(n-1)n}{4} = O(n^2)$$

The summation starts from zero because the maximum number of assignments is one less than the serial number of the element in the array.

In the worst case, the list is in reverse sorted order, so that each element must push up all other elements encountered so far. The total number of assignments in this case is twice the average case:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$$

Note

Insertion sort can be O(n log n) if small gaps are left in the array. See http://citeseer.ist.psu.edu/bender04insertion.html

Test driver

To try out the sort, we can write a simple test driver file with the following structure:

```
<<insertion_sort.c>>=
header files

insertion_sort
helper functions

test main
```

Our test main will create a small array, sort it, then print out the result. We use the traditional "sizeof division" trick (http://c-faq.com/aryptr/arraynels.html) to get the array length at compile time:

```
<<test main>>=
int main(void)
{
    test main variable declarations
    int a[] = {5, 1, 9, 3, 2};
    insertion_sort(a, sizeof(a)/sizeof(*a));
    print out a
    check that the array is sorted
    return 0;
}
```

Checking that the array is sorted is a pretty simple loop but we have to make sure that we handle the cases where there are zero or one elements in the array, hence we start at the second element and always compare

an element to the one before it.

```
<<helper functions>>=
void checkThatArrayIsSorted (int array[], int length)
{
    int i;
    for (i = 1; i < length; i+=1)
    {
       if (array[i-1] > array[i])
       {
          printf("The array is not in sorted order at position %d\n", i-1);
       }
    }
}

<<check that the array is sorted>>=
checkThatArrayIsSorted(a, sizeof(a)/sizeof(*a));
```

A better test main might ask for numbers from the user or generate an array of random numbers. For sanity's sake, we will print out the array so that we can visually verify it is sorted. To print the array, we print out each element of the array using printf.

```
<<test main variable declarations>>=
unsigned int i;
<<print out a>>=
/* Print out a */
for(i=0; i<sizeof(a)/sizeof(*a); i++)
{
    printf("%d\n", a[i]);
}
```

We use unsigned because the sizeof division result is unsigned. We'll need to include stdio.h for printf:

```
<<header files>>=
#include <stdio.h>
```

This completes our simple exposition of iterative insertion sort in C. On UNIX, you can build and run this sample using the following script:

```
<<build_and_run.sh>>=
#!/bin/bash
gcc -Wall -O2 insertion_sort.c -o insertion_sort
./insertion_sort
```

Download code (http://en.literateprograms.org/index.php?title=Special:Downloadcode/Insertion_sort_(C)&oldid=18489)

Retrieved from "http://en.literateprograms.org/index.php?title=Insertion_sort_(C)&oldid=18489"
Categories: Programming language:C | Environment:Portable | Insertion sort