Selection sort (C)

From LiteratePrograms

Other implementations: C | C++ | C# | Erlang | Haskell | Java | Python, arrays

This is an implementation of selection sort (http://en.wikipedia.org/wiki/Selection_sort) in the C language. Selection sort is a simple but inefficient sort which operates by repeatedly extracting the minimum from the list of remaining unsorted elements until none are left. We show an implementation for integer arrays and a generic implementation using function pointers.

Contents

Integer arrays

We start with an implementation specifically for integer arrays:

```
<<function prototypes>>=
void selection_sort_int_array(int a[], int num_elements);
<<integer array sort>>=
void selection_sort_int_array(int a[], int num_elements)
{
    int sorted_size;
    for(sorted_size=0; sorted_size < num_elements; sorted_size++)
    {
        find minimum of remaining unsorted list
        swap a[sorted_size] with minimum
    }
}
```

At any instant the list is storing two sublists: a sorted list of the first `sorted_size` elements, followed by all remaining unsorted elements. To find the minimum of the remaining elements, we just loop over them keeping track of the smallest one we see:

```
<<find minimum of remaining unsorted list>>=
int min_element_index = sorted_size;
int min_element_value= a[sorted_size];
int i;
for(i=sorted_size+1; i<num_elements; i++)
{
    if (a[i] < min_element_value)
    {
        min_element_index = i;
        min_element_value = a[i];
    }
}
```

We then perform the swap, taking advantage of the existing variable `min_element_value`:

```
<<swap a[sorted_size] with minimum>>=
a[min_element_index] = a[sorted_size];
a[sorted_size] = min_element_value;
```

This completes the integer array sort.

Function pointers

Because selection sort is a comparison sort, we can generalize this sort to work with any kind of data that can be compared. Our implementation will use exactly the same interface used by the C library function `qsort() (http://www.cplusplus.com/ref/cstdlib/qsort.html)` :

- `base` is a void pointer to the first element of the array.
- `num_elements` is the number of elements in the array.
- `element_size` is the uniform number of bytes that each element occupies.
- `compare` is a function pointer than takes pointers to two elements and returns a negative, zero, or positive value, depending on whether the first value is less than, equal to, or greater than the second value, respectively.

```
<<function prototypes>>=
void selection_sort(void * base, size_t num_elements, size_t element_size,
                    int (*comparer)(const void *, const void *));

<<type definition headers>>=
#include <stddef.h>   /* for size_t */
```

Because exchanges and comparisons are relatively complex in this representation, we add helper functions to facilitate these (originally from Quicksort (C)):

```
<<generic element compare and exchange helper functions>>=
static int compare_elements_helper(void *base, size_t element_size, size_t idx1, size_t idx2,
                                   int(*comparer)(const void *, const void*))
{
    char* base_bytes = (char*)base;
    return comparer(&base_bytes[idx1*element_size], &base_bytes[idx2*element_size]);
}

#define element_less_than(i,j)  (compare_elements_helper(base, element_size, (i), (j), comparer) <

static void exchange_elements_helper(void *base, size_t element_size, size_t idx1, size_t idx2)
{
    char* base_bytes = (char*)base;
    int i;
    for (i=0; i<element_size; i++)
    {
        char temp = base_bytes[idx1*element_size + i];
        base_bytes[idx1*element_size + i] = base_bytes[idx2*element_size + i];
        base_bytes[idx2*element_size + i] = temp;
    }
}

#define exchange_elements(i,j)  (exchange_elements_helper(base, element_size, (i), (j)))
```

The casts to `char*` allow this code to compile as C++.

We're now prepared to write our implementation, which is very similar to what we had before, except substituting the appropriate macros to perform comparisons and exchanges:

<<function pointer array sort>>=
```
void selection_sort(void * base, size_t num_elements, size_t element_size,
                    int (*comparer)(const void *, const void *))
{
    size_t sorted_size;
    for(sorted_size=0; sorted_size < num_elements; sorted_size++)
    {
        size_t min_element_index = sorted_size;
        size_t i;
        for(i=sorted_size+1; i<num_elements; i++)
        {
            if (element_less_than(i,min_element_index))
            {
                min_element_index = i;
            }
        }
        exchange_elements(min_element_index, sorted_size);
    }
}
```

For simplicity, we no longer keep track of the minimum element's value in a variable, instead comparing to the original array entry in place.

Macros

The function pointer generic C implementation above, like the standard library's `qsort()`, suffers from overhead due to calls through the function pointer, which cannot be inlined. In C++ we can solve this using templates (see Selection sort (C Plus Plus)), in C the only effective way to approach this kind of speed in a generic implementation is using the preprocessor. The implementation is almost identical to our first implementation, except that the element type *int* is replaced by a macro parameter *T*, and the entire implementation is placed inside a single large macro definition (this is why there are trailing backslashes).

<<macro array sort prototype macro>>=
```
#define DECLARE_MACRO_SELECTION_SORT(name, T, compare_less) \
void name(T a[], int num_elements);
```

<<macro array sort macro>>=
```
#define DEFINE_MACRO_SELECTION_SORT(name, T, compare_less) \
void name(T a[], int num_elements) \
{ \
    int sorted_size; \
    for(sorted_size=0; sorted_size < num_elements; sorted_size++) \
    { \
        int min_element_index = sorted_size; \
        T min_element_value = a[sorted_size]; \
        int i; \
        for(i=sorted_size+1; i<num_elements; i++) \
        { \
            if (compare_less(a[i], min_element_value)) \
            { \
                min_element_index = i; \
                min_element_value = a[i]; \
            } \
        } \
        a[min_element_index] = a[sorted_size]; \
        a[sorted_size] = min_element_value; \
    } \
}
```

Passing the `less_than` comparator as a macro parameter allows us to sort things that aren't built-in types, which using "<" would restrict us to. We can still use the built-in "<" operator by passing in this macro:

<<comparer macros>>=
```
#define less_than(x,y)  ((x) < (y))
```

To use the macro, we first invoke the macros to declare and/or define the desired function, then we can call the function. For example, this reproduces our original algorithm:

```
/* In header file: */
DECLARE_MACRO_SELECTION_SORT(selection_sort_int_array, int, less_than)
/* In source file: */
DEFINE_MACRO_SELECTION_SORT(selection_sort_int_array, int, less_than)
/* At call site: */
int a[] = {5, 2, 4, 1, 3};
selection_sort_int_array(a, 5);
```

Files

We can wrap these three implementations up in source and header files for reuse, keeping in mind that the macros must go entirely in the header files since they only declare macros:

```
<<selection_sort.h>>=
#ifndef _SELECTION_SORT_H_

type definition headers

function prototypes

comparer macros
macro array sort prototype macro
macro array sort macro

#endif

<<selection_sort.c>>=
#include "selection_sort.h"

generic element compare and exchange helper functions

integer array sort

function pointer array sort
```

This completes the implementation.

Test driver

To test our implementation, we create a separate source file invoking all three sorts on a random integer array of a size specified on the command-line:

```
<<selection_sort_test.c>>=
#include <stdio.h>
#include <stdlib.h>
#include "selection_sort.h"

DECLARE_MACRO_SELECTION_SORT(selection_sort_int_array2, int, less_than)
DEFINE_MACRO_SELECTION_SORT(selection_sort_int_array2, int, less_than)

int less_compare(const void * a, const void * b)
{
    return (*(const int*)a) - (*(const int*)b);
}

int main(int argc, char* argv[])
{
    int length = atoi(argv[1]);
    int* a = (int*)malloc(sizeof(int) * length);
    int i;
    for (i=0; i<length; i++)
    {
        a[i] = rand();
    }
    selection_sort_int_array(a, length);
    for (i=1; i<length; i++)
    {
        if (a[i] < a[i-1])
        {
            puts("ERROR");
        }
    }

    for (i=0; i<length; i++)
    {
        a[i] = rand();
    }
    selection_sort(a, length, sizeof(int), less_compare);
    for (i=1; i<length; i++)
    {
        if (a[i] < a[i-1])
        {
            puts("ERROR");
        }
    }

    for (i=0; i<length; i++)
    {
        a[i] = rand();
    }
    selection_sort_int_array2(a, length);
    for (i=1; i<length; i++)
    {
        if (a[i] < a[i-1])
        {
            puts("ERROR");
        }
    }

    return 0;
}
```

It verifies that the resulting arrays are in order. Since we know the sort only performs exchanges, and so the output is a permutation of the input, this establishes correctness for the particular given input. When we run this with various list sizes, it produces no output, as desired.

The cast of malloc's result is not strictly necessary but allows this code to compile as C++.

Performance

We performed a trial using the first implementation on a Gentoo Linux machine with a 2.8 GhZ CPU, compiled using gcc with the "-O3" option. We compared it against three other implementations:

- A simple insertion sort implementation for integers.
- `qsort()`, which suffers from some additional overhead due to its use of function pointers;
- C++'s `std::sort`, a carefully tuned quicksort implementation which does not possess qsort's liability.

Times are all in seconds and include setup and tear down. The results are as follows:

List size	100	1,000	10,000	25,000	50,000	75,000	100,000	250,000
Selection sort	0.001	0.003	0.060	0.350	1.38	3.35	5.92	48.5
Insertion sort	0.001	0.003	0.036	0.193	0.773	1.70	3.08	27.7
qsort	0.001	0.001	0.004	0.010	0.019	0.029	0.037	0.104

std::sort	0.002	0.003	0.003	0.005	0.009	0.012	0.016	0.038

It is evident that selection sort is never useful on random lists, even for very small lists, even against the equally simple insertion sort. At just 100,000 elements, `std::sort` surpasses it by a factor of over 1000 times. Don't use it.

Improvements

Selection sort's strategy for selecting the minimum element is naive; using a much more clever strategy we can find it in O(log n) time, giving heapsort; see Heapsort (C) for an implementation.

Although the macro implementation is fast, it suffers from poor readability and takes up a lot of code space. In C++, we can use templates to produce a much easier to read fast generic sort implementation, and many compilers can combine instantiations of a templated class; see Selection sort (C Plus Plus).

Download code (http://en.literateprograms.org/index.php?title=Special:Downloadcode/Selection_sort_(C)&oldid=13212)

Retrieved from "http://en.literateprograms.org/index.php?title=Selection_sort_(C)&oldid=13212"
Categories: Programming language:C | Environment:Portable | Selection sort

- This page was last modified on 23 April 2008, at 16:45.
- Content is available under the MIT/X11 License.