Quicksort (C)

From LiteratePrograms

Other implementations: AWK | **C** | C++ | Eiffel | Erlang | Forth | Haskell | Java | JavaScript | Mathematica | Mercury | Oz | Python | Python, arrays | Scala | Sed | Standard ML | Visual Basic .NET | XProc

Quicksort (http://en.wikipedia.org/wiki/Quicksort) is a simple sorting algorithm that works by choosing a certain element, called the *pivot*, and then dividing the list into two parts, those less than the pivot and those greater than or equal to the pivot. Each list is then recursively sorted. For arrays on typical modern architectures, it is one of the fastest sorting algorithms available.

Contents

Main algorithm

In outline, the sort looks like this:

```
<<public declarations>>=
void quicksort(void * base, size_t num_elements, size_t element_size,
               int (*comparer)(const void *, const void *));
<<quicksort>>=
void quicksort(void * base, size_t num_elements, size_t element_size,
               int (*comparer)(const void *, const void *))
{
    quicksort declarations
    check termination condition
    select a pivot
    partition the array
    recursively sort each subpartition
}
```

We need stddef.h for `size_t`:

```
<<type definition headers>>=
#include <stddef.h>
```

The interface is the same as that used by the C library's `qsort()` (http://www.cplusplus.com/ref/cstdlib/qsort.html) function, which allows the sort to operate on any type of data given a suitable comparison function (this is possible because quicksort is a comparison sort). However, this representation can also make it cumbersome to work with array elements, so we add two helper functions that compare and exchange two array elements, respectively:

<<array element accessors>>=
```
static int compare_elements_helper(void *base, size_t element_size, int idx1, int idx2,
                                   int(*comparer)(const void *, const void*))
{
    char* base_bytes = 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) < 0

static void exchange_elements_helper(void *base, size_t element_size, int idx1, int idx2)
{
    char* base_bytes = 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)))
```

One simple termination condition is that lists of size 1 or smaller are already sorted:

<<simple check termination condition>>=
```
if (num_elements <= 1) {
    return;
}
```

However, quicksort is not the most efficient sort for small lists. Instead, when we reach a small enough sublist we switch to a sort that is efficient for small lists, such as insertion sort. The best point to switch over varies from platform to platform and is also influenced by factors such as element size and cache contents, but for simplicity we use a fixed value here:

<<constants>>=
```
#define MIN_QUICKSORT_LIST_SIZE    32
```

<<check termination condition>>=
```
if (num_elements < MIN_QUICKSORT_LIST_SIZE) {
    insertion_sort(base, num_elements, element_size, comparer);
    return;
}
```

<<insertion sort definition>>=
```
void insertion_sort(void * base, size_t num_elements, size_t element_size,
                    int (*comparer)(const void *, const void *))
{
   int i;
   for (i=0; i < num_elements; i++)
   {
       int j;
       for (j = i - 1; j >= 0; j--)
       {
           if (element_less_than(j, j + 1)) break;
```

```
            exchange_elements(j, j + 1);
        }
    }
}
```

A more efficient implementation of insertion sort would avoid extra assignments by saving out the element that is being inserted in auxilary storage instead of continually exchanging it. This storage would be allocated once by quicksort and reused for each insertion sort. For simplicity we neglect this optimization here.

Partitioning

The key to an efficient quicksort implementation is an efficient partition algorithm. Our goal is to rearrange the array such that all values less than the pivot are found at the beginning of the array and all those greater or equal are found afterwards.

The argument list is similar to that of quicksort itself, but with an added parameter specifying the index of the pivot element. We also return the final index of the pivot, which enables us to identify the position and extent of the two sublists:

<<partition declaration>>=
```
int partition(void * base, size_t num_elements, size_t element_size,
              int (*comparer)(const void *, const void *), int pivotIndex)
```

We invoke the call like this:

<<partition the array>>=
```
pivotIndex = partition(base, num_elements, element_size, comparer, pivotIndex);
```

One straightforward in-place algorithm for implementing this function is to create two indexes pointing into the array, where one starts at the beginning moving forwards and skips over elements less than the pivot and the other starts at the end moving backwards and skips over elements greater than or equal to the pivot. When the two indexes find two elements out of order, we exchange the elements. The place where the pointers meet is where the pivot belongs.

<<partition definition>>=
```
partition declaration
{
    int low = 0, high = num_elements - 1;
    save pivot in element num_elements-1 of array
    while (1) {
        advance indexes
        if (low > high) break;
        exchange elements pointed to by high and low indexes
    }
    move pivot into final position and return final position
}
```

We increment the low index until it finds an element greater than or equal to the pivot, and decrement the high index until it finds an element less than the pivot:

<<advance indexes>>=

```
while (element_less_than(low, num_elements-1)) {
    low++;
}
while (!element_less_than(high, num_elements-1)) {
    high--;
}
```

Now the two indexes point to two elements in the wrong order, so we exchange them:

```
<<exchange elements pointed to by high and low indexes>>=
exchange_elements(low, high);
```

We move the pivot "out of the way" to ensure that it isn't moved around during the process - this would complicate things. At the end we store it where *low* points, which will be one past the end of the lesser elements, its final position:

```
<<save pivot in element num_elements-1 of array>>=
exchange_elements(num_elements - 1, pivotIndex);

<<move pivot into final position and return final position>>=
exchange_elements(low, num_elements - 1);
return low;
```

Because the pivot was saved at the end of the array, it's necessary to exchange it with the first element in the sublist of greater-or-equal elements rather than the last element of the lesser elements.

Using the index returned by `partition()`, we can now recursively sort each subpartition. Because the right partition will have a different base pointer (it starts at index *pivot index* + 1), we treat the old base pointer as a byte array and increment it by the appropriate number of bytes:

```
<<recursively sort each subpartition>>=
quicksort(base, pivotIndex, element_size, comparer);
quicksort(((char*)base) + element_size*(pivotIndex+1),
          num_elements - (pivotIndex + 1), element_size, comparer);
```

Pivot selection

We use a simple but effective pivot selection based on C's built-in random number generator `rand()`:

```
<<quicksort declarations>>=
int pivotIndex;

<<select a pivot>>=
pivotIndex = rand() % num_elements;
```

We need stdlib.h for rand:

```
<<quicksort implementation headers>>=
#include <stdlib.h>
```

There are other techniques available, but this one performs well in practice with lists of many sizes and is less prone to unexpected worst-case behaviour than some more deterministic choices.

To ensure that certain common inputs do not by sheer chance have bad performance in every run, we choose a random seed based on the current system time:

```
<<header files>>=
#include <time.h>
#include <stdlib.h>

<<initialization>>=
srand(time(NULL));
```

Because resisting intentional worst-case input is a non-goal in this implementation, we won't discuss secure selection of random seeds, but this would just be done instead of seeding with the time. If the seed is hidden then a worst-case behaviour cannot be triggered, since we don't reveal intermediate state.

Files

We provide a header file exposing only the main quicksort method, along with a source file containing the implementation:

```
<<quicksort.h>>=
#ifndef _QUICKSORT_H_
#define _QUICKSORT_H_

type definition headers

public declarations

#endif

<<quicksort.c>>=
quicksort implementation headers
#include "quicksort.h"

constants
array element accessors
insertion sort definition
partition definition
quicksort
```

Test driver

Here's a very simple test driver that runs quicksort on a large array of randomly chosen integers and then verifies that the result is in order:

```
<<quicksort_test.c>>=
#include <stdio.h>
header files
#include "quicksort.h"

int compare_int(const void* left, const void* right) {
    return *((int*)left) - *((int*)right);
}

int main(int argc, char* argv[]) {
    int size = atoi(argv[1]);
    int* a = malloc(sizeof(int)*size);
    int i;
    initialization
    for(i=0; i<size; i++) {
        a[i] = rand() % size;
    }
    quicksort(a, size, sizeof(int), compare_int);
    for(i=1; i<size; i++) {
        if (!(a[i-1] <= a[i])) {
            puts("ERROR");
            return -1;
        }
    }
    puts("SUCCESS");
    return 0;
}
```

I compiled this using "gcc -O3" for Linux and ran it on a Pentium 4 2.8 GHz with 1GB of RAM. I got the following total runtime measurements for lists of various sizes:

List size	10,000	100,000	1,000,000	10,000,000	100,000,000
Total time (s)	0.007	0.053	0.614	6.928	85.483
Microseconds per element	0.7	0.53	0.614	0.693	0.855
Comparisons	225,362	2,777,440	34,706,044	409,614,366	4,696,992,890
Comparisons per element	22.5	27.8	34.7	41.0	47.0
glib qsort time (s)	0.006	0.045	0.500	5.681	57.414

We can see that the algorithm is quick even for very large lists, and that the time per element rises slowly, never exceeding a microsecond in these examples. A large proportion of this time is spent invoking the user-supplied comparison function; for example, in one run on a list of 10 million elements, there were over 400 million comparisons, and the comparison function took up 20% of the runtime. C++ is able to eliminate much of this overhead using templates, which enable inlining of the comparison function; see Quicksort (C Plus Plus).

As predicted by theory, the number of comparisons per element rises proportionally to log *n*; as the list size increases geometrically, it increases roughly linearly (by about 6 for each factor of 10). Finally, the GNU implementation of the standard library's `qsort()` function consistently outperforms our own by 20%-30%, due to its extensive optimizations.

Download code (http://en.literateprograms.org/index.php?title=Special:Downloadcode/Quicksort_(C)&oldid=10011)

Retrieved from "http://en.literateprograms.org/index.php?title=Quicksort_(C)&oldid=10011"
Categories: Programming language:C | Environment:Portable | Quicksort