The most discussed and researched algorithm undoubtedly is quicksort.

We will examine a few variations of quicksort in this installment.

Examine the files in the directory version 1.

We introduce a preprocessor variable DEBUG through the command line option -D.

If the variable has a value 1, we do some extra counting. Otherwise we make the variable 0 and turn off any extra processing.

The array is initialized with numbers from 1 to n and then the elements are shuffled. If DEBUG is on, after sorting, statistics like # of comparisons and # of moves of data are displayed.

Correctness of the algorithm is checked by the function is\_sorted.

**version1/qsort.h**

#pragma once

void myswap(int \*p, int \*q);

void fill(int a[], int n);

int is\_sorted(int a[], int n);

void disp(int\*, int);

void myqsort(int\* a, int l, int r);

**version1/client.c**

#include <stdio.h>

#include "qsort.h"

#if DEBUG

extern int move\_count;

extern int comp\_count;

#endif

#define SIZE 100

int main()

{

int n;

n = SIZE + 1;

int a[n + 1];

fill(a, n);

myqsort(a, 1, n);

printf("is sorted : %d\n", is\_sorted(a, n));

#if DEBUG

printf("# of moves : %d\n# of comp : %d\n",

move\_count, comp\_count);

#endif

}

/\*

// DEBUG is 1

$ gcc -DDEBUG=1 -c client.c

$ gcc -DDEBUG=1 -c qsort.c

$ gcc client.o qsort.o

$ ./a.out

is sorted : 1

# of moves : 849

# of comp : 831

\*/

/\*

// DEBUG is 0

$ gcc -DDEBUG=0 -c client.c

$ gcc -DDEBUG=0 -c qsort.c

$ gcc client.o qsort.o

$ ./a.out

is sorted : 1

\*/

**version1/qsort.c**

#include <stdio.h>

#include <stdlib.h>

#include "qsort.h"

#if DEBUG

int move\_count = 0;

int comp\_count = 0;

#endif

void shuffle(int a[], int n)

{

int pos;

// may want to introduce a seed for random # generation

for(int i = 1; i <= n / 2; ++i)

{

pos = rand() % n + 1;

myswap(&a[i], &a[pos]);

}

}

void fill(int a[], int n)

{

// printf("fill called\n");

for(int i = 1; i <= n; ++i)

{

a[i] = i;

}

shuffle(a, n);

}

void disp(int\* a, int n)

{

for(int i = 0; i < n; ++i)

{

printf("%d ", a[i]);

}

printf("\n");

}

void myswap(int \*p, int \*q)

{

int t = \*p; \*p = \*q; \*q = t;

}

int partition(int a[], int l, int r)

{

int i = l; int j = r + 1;

int p = a[l];

do

{

do

{

++i;

}

while(i <= r && a[i] < p);

do

{

--j;

}

while(j >= l && a[j] > p);

myswap(&a[i], &a[j]);

#if DEBUG

move\_count += 3;

#endif

} while(i < j);

myswap(&a[i], &a[j]);

myswap(&a[l], &a[j]);

#if DEBUG

comp\_count += r - l + 1;

move\_count += 6;

#endif

return j;

}

This qsort routine is based on Hoare’s partition. This is a naive implementation based on the algorithm in your algorithm book.

void myqsort(int a[], int l, int r)

{

if(l < r) // section has more than one element

{

int s = partition(a, l, r);

myqsort(a, l, s - 1);

myqsort(a, s + 1, r);

}

}

int is\_sorted(int a[], int n)

{

for(int i = 1; i < n; ++i)

{

if(a[i] > a[i + 1])

return 0;

}

return 1;

}

**version 2:**

This is same as the first version but for measurement of timings.

The client has changed to implement the measurement.

double time\_elapsed(struct timespec start, struct timespec end)

{

double t;

t = (end.tv\_sec - start.tv\_sec);

t += (end.tv\_nsec - start.tv\_nsec) \* 0.000000001;

return t;

}

#if DEBUG

extern int move\_count;

extern int comp\_count;

#endif

#define SIZE 100000

int main()

{

int n;

n = SIZE + 1;

int a[n + 1];

fill(a, n);

struct timespec start;

struct timespec end;

clock\_gettime(CLOCK\_REALTIME, &start);

myqsort(a, 1, n);

clock\_gettime(CLOCK\_REALTIME, &end);

printf("time %lf \n",

time\_elapsed(start, end));

printf("is sorted : %d\n", is\_sorted(a, n));

#if DEBUG

printf("# of moves : %d\n# of comp : %d\n",

move\_count, comp\_count);

#endif

}

/\*

$ gcc -DDEBUG=0 -c client.c

$ gcc -DDEBUG=0 -c qsort.c

$ gcc client.o qsort.o

$ ./a.out

time 0.020015

is sorted : 1

$ ./a.out

time 0.018569

is sorted : 1

$ ./a.out

time 0.018829

is sorted : 1

$ ./a.out

time 0.018404

is sorted : 1

$ ./a.out

time 0.018706

is sorted : 1

$ ./a.out

time 0.023675

is sorted : 1

\*/

/\*

$ gcc -DDEBUG=1 -c qsort.c

$ gcc -DDEBUG=1 -c client.c

$ gcc client.o qsort.o

$ ./a.out

time 0.020588

is sorted : 1

# of moves : 1332144

# of comp : 2725819

$ ./a.out

time 0.019137

is sorted : 1

\*/

We are sorting an array of 100000 integers.

With no extra measuments, the code is a bit faster.

Version 3:

We will change the partition technique to Lomuto partition instead of Hoare partition.

This algorithm is much simpler to implement than Hoare’s partition.

The model is like this.

The position l is the pivot position.

At any point in time, all elements from l + 1 to m will be less than the pivot value.

All elements between m + 1 and i will be greater than the pivot value.

Those from i + 1 to r are yet to be examined.

// use Lomuto Partition

int partition(int a[], int l, int r)

{

int m = l;

for(int i = l + 1; i <= r; ++i)

{

if(a[i] < a[l])

{

myswap(&a[i], &a[++m]);

#if DEBUG

move\_count += 3;

#endif

}

}

myswap(&a[l], &a[m]);

#if DEBUG

move\_count += 3;

comp\_count += r - l - 1;

#endif

return m;

}

I have created two shell programs a.sh and b.sh to compile with or without DEBUG enabled.

Have a look.

**a.sh**

gcc -DDEBUG=1 -c qsort.c

gcc -DDEBUG=1 -c client.c

gcc client.o qsort.o

./a.out

**b.sh**

gcc -DDEBUG=0 -c qsort.c

gcc -DDEBUG=0 -c client.c

gcc client.o qsort.o

./a.out

/\*

$ ./a.sh

time 0.018717

is sorted : 1

# of moves : 3097173

# of comp : 2196081

\*/

/\*

$ ./b.sh

time 0.017749

is sorted : 1

$ ./b.sh

time 0.016855

is sorted : 1

$ ./b.sh

time 0.017398

is sorted : 1

$ ./b.sh

time 0.017527

is sorted : 1

$ ./b.sh

time 0.017685

is sorted : 1

\*/

Do not worry about the counts of comparison and movement. This program seems to be a bit more efficient compared to Hoare’s partition.

**Version 4:**

To make the progtram more efficient, we may inline the code for swapping.

// use Lomuto Partition

int partition(int a[], int l, int r)

{

int m = l; int p = a[l]; int t;

for(int i = l + 1; i <= r; ++i)

{

if(a[i] < p)

{

//myswap(&a[i], &a[++m]);

++m;

**t = a[i]; a[i] = a[m]; a[m] = t;**

#if DEBUG

move\_count += 3;

#endif

}

}

//myswap(&a[l], &a[m]);

**t = a[l]; a[l] = a[m]; a[m] = t;**

#if DEBUG

move\_count += 3;

comp\_count += r - l - 1;

#endif

return m;

}

/\*

$ ./a.sh

time 0.014929

is sorted : 1

# of moves : 3097173

# of comp : 2196081

\*/

/\*

$ ./b.sh

time 0.014494

is sorted : 1

$ ./b.sh

time 0.014530

is sorted : 1

$ ./b.sh

time 0.014521

is sorted : 1

$ ./b.sh

time 0.014697

is sorted : 1

$ ./b.sh

time 0.014488

is sorted : 1

\*/

You can observe that the timings have improved.

Can we improve these better?

Qsort works fine for large arrays on average case. It is not very good for small arrays. There is no simple way of finding what is small.

We may decide to stop qsort when a particular threshold size is reached and switch over to insertionsort.

This is a common technique used in library routines.