# Expected Number of Comparisons in quick-sort

## Analysis

Let ***A*** be a random array, in which every element is distinct.

The distance between two elements is defined as:

If ***a*** is the ***i***-th smallest element in ***A***, ***b*** is the ***j***-th smallest element in ***A***, then the distance between ***a*** and ***b*** is ***|i-j|***.

We will show that when running quick-sort on an array with N distinct items, the probability of comparing any two elements whose distance are ***n*** is ***2/n***.

Assume that the two elements are the i-th smallest and j-th smallest in the array. If an element is randomly selected, there are three cases :

1. the i-th or th j-th smallest is selected as the partition element. Probability ***2/N***. The two elements will be compared. After the partition, they will never be compared again.

2. the element larger than the i-th and smaller than the j-th is selected as the partition element. Probability ***(n-1) / N***. the two elements won't be compared, and after the partition is done, they belong to different partitions and they will never be compared.

3. the element smaller than the i-th or larger than the j-th is selected. Probability ***(N – n – 1)/N***. The two will not be compared, but since they belong to the same partitions, they may be compared in the following partitions.

Let ***P(N, i, j)*** denote the possibility of comparing the ***i***-th smallest and ***j***-th smallest of an array of length ***N***, where ***i < j*** and ***N >=2*** . Since the array is random, from the previous reasoning we have:

Let ***T(N)*** be the proposition :

***P(N, i, i + n)***  ***= 2 / (n+1)***, where ***0<* i *,0 < n and i + n <= N***

Basic step : when ***N = 2***, ***P(2, 1, 2) = 2 / (1+1) = 1***. the only possible case is ***i = 1*** and n ***= 1***

Inductive step: assume that ***P(K)*** is true for ***2 <= K <= N***. And assume that ***0 < i, 0 < n, i + n <= N + 1***.

For , since ***2 <= N, 1 <= k < i*** we have ***2 <= N + 1 – k < N , i – k > 0, i + n – k <= N + 1 – k*** , by inductive hypothesis ***P(N -k, i-k,i+n-k)=2/(n+1)***.

For , since ***i + n + 1 <= k <= N + 1***, we have ***2 <= k – 1 <= N***, and ***i > 0, i + n <= k-1***, by inductive hypothesis ***P(k-1, i, i+n) = 2 /(n+1)***.

, that is

We have shown that ***T(N + 1)*** follows from inductive hypothesis.

Thus ***T(N)*** is true for all ***N >= 2****.*

So the Expected number of comparisons when quick-sort a random array with distinct elements of length ***N*** is :

## Codes

/\*

\* QuickSortTest.cpp

\*

\* Created on: Jan 4, 2014

\* Author: Stiff Liu

\*/

**#include** <GL/glut.h>

**#include** <vector>

**#include** <thread>

**#include** <mutex>

**#include** <algorithm>

**using** **namespace** std;

**class** QuickSortTest{

**public**:

/\*\*

\* This algorithm is the C++ version of the algorithm 2.5 from the book "Algorithms 4th Edition"

\* This function partitions the input array by the first element in it.

\* Say the first element in the array is {@var a}.

\* This function returns an partition index {@var i}, such that :

\* 1. the elements in the range [0, i] is not greater than than {@var a}.

\* 2. the i-th element equals {@var a}.

\* 3. the elements in the range [i+1, n) is not less than {@var a}.

\* @param a The input array, which will be partitioned.

\* @param n Number of elements in the input array.

\* @param comparator The "less than" relation between two elements in the array.

\* @return The partition index.

\*/

**template**<**class** **T**, **class** **Comparator**>

**static** **unsigned** **int** **partition**(**T** a, **unsigned** **int** n, **Comparator** comparator) {

**if** (n == 0)

**return** 0;

**unsigned** **int** i = 0, j = n;

**while** (**true**) {

**while** (comparator(a[++i], a[(**unsigned** **int**) 0]))

**if** (i == n - 1)

**break**;

**while** (comparator(a[(**unsigned** **int**) 0], a[--j]))

**if** (j == 0)

**break**;

**if** (i >= j)

**break**;

std::swap(a[i], a[j]);

}

std::swap(a[(**unsigned** **int**) 0], a[j]);

**return** j;

}

/\*\*

\* This algorithm is the C++ version of the algorithm 2.5 from the book "Algorithms 4th Edition"

\* The very famous quick sort.

\*

\* @param a The array to be sorted.

\* @param n Number of elements in the array.

\* @param comparator The "less than" relation between two elements in the array.

\*/

//static unsigned int smallArrayCount;

**template**<**class** **T**, **class** **Comparator**, **class** **PartitionMethod**>

**static** **void** **quickSortBase**(**T** a, **unsigned** **int** n, **Comparator** comparator,

**PartitionMethod** partition) {

**if** (n <= 1)

**return**;

**unsigned** **int** j = partition(a, n, comparator);

//if(j <= 2)

// ++ smallArrayCount;

//if(j + 1 + 2 >= n)

// ++smallArrayCount;

**if** (j > 1)

quickSort(a, j, comparator);

**if** (j + 2 < n)

quickSort(a + (j + 1), n - j - 1, comparator);

}

**template**<**class** **T**, **class** **Comparator**>

**static** **void** **quickSort**(**T** a, **unsigned** **int** n, **Comparator** comparator) {

**unsigned** **int** (\*func)(**T**, **unsigned** **int**, **Comparator**) = partition;

quickSortBase(a, n, comparator, func);

}

**template**<**class** **T**>

**struct** CountedLessThan {

**unsigned** **long** **long** \*counter;

**CountedLessThan**(**unsigned** **long** **long** \* counter = **nullptr**) :

counter(counter) {

}

**bool** **operator()**(**T** val1, **T** val2) {

**if** (counter != **nullptr**)

++\*counter;

**return** val1 < val2;

}

};

**template**<**int** **N**, **class** **T** = **double**>

**struct** TTuple {

**static\_assert**(**N** > 0, "N should be greater than zero");

**T** values[**N**];

**static** **const** **int** *size* = **N**;

**typedef** **T** type;

**template**<**typename** ... **U**>

**TTuple**(**U** ... vals) :

values { vals... } {

}

};

**template**<**class** **T**>

**static** **void** **plotPoints**(**const** std::vector<**T**>& data, **double** maxValue,

size\_t count) {

**double** xStart = -0.95;

**double** yStart = -0.95;

**double** xEnd = 0.95;

**double** yEnd = 0.95;

**double** xLen = xEnd - xStart;

**double** yLen = yEnd - yStart;

**glBegin**(GL\_POINTS);

**for** (size\_t i = 0; i < count; ++i) {

**double** x = xStart + i \* xLen / data.size();

**double** y = yStart + yLen \* data[i] / maxValue;

//assert(data[i] <= maxValue);

**glVertex2f**(x, y);

}

**glEnd**();

}

**static** **void** **openGLInit**() {

**glClearColor**(0.000, 0.110, 0.392, 0.0); // JMU Gold

**glColor3f**(0.314, 0.314, 0.000); // JMU Purple

**glMatrixMode**(GL\_PROJECTION);

**glLoadIdentity**();

**glPointSize**(2.0);

**gluOrtho2D**(-1.0, 1.0, -1.0, 1.0);

}

**template**<**class** **T**, **int** **N** = 2>

**class** RunningTimePlotter {

std::thread workingThread;

std::mutex m;

**bool** isDone = **false**;

**T** val;

vector<vector<**double**>> values;

vector<**unsigned** **int**> counts;

TTuple<3 \* **N**> colors;

**static** **void** **display**() {

*instance*->show();

}

**static** **void** **timer**(**int** value) {

**glutTimerFunc**(*interval*, *timer*, 1000);

**glutPostRedisplay**();

}

**static** **void** **func**(RunningTimePlotter \* **const** i) {

i->work();

}

**void** **work**() {

**while** (!isDone) {

TTuple<**N**> tmp = val();

{

std::lock\_guard<std::mutex> lk(m);

**for** (size\_t i = 0; i < values.size(); ++i) {

vector<**double**>& v = values[i];

**unsigned** **int**& count = counts[i];

**if** (count == v.size()) {

v.erase(v.begin());

v.push\_back(tmp.values[i]);

} **else** {

v[count] = tmp.values[i];

++count;

}

}

}

std::chrono::milliseconds dura(50);

std::this\_thread::sleep\_for(dura);

}

}

**double** **getMaxValue**() {

**double** maxValue = 0.0;

**for** (size\_t i = 0; i < values.size(); ++i) {

vector<**double**>& v = values[i];

maxValue = std::max(maxValue,

\*std::max\_element(v.begin(), v.end()));

}

**if** (maxValue == 0.0)

maxValue = 1.0;

**return** maxValue;

}

**public**:

**RunningTimePlotter**(**const** **T**& t, **const** TTuple<3 \* **N**>& colors) :

val(t), colors(colors) {

*instance* = **this**;

values.resize(**N**);

**for** (size\_t i = 0; i < **N**; ++i) {

values[i].resize(1000);

counts.push\_back(0);

}

}

**void** **show**() {

**glClear**(GL\_COLOR\_BUFFER\_BIT);

{

std::lock\_guard<std::mutex> lk(m);

**double** maxValue = getMaxValue();

**glPointSize**(3.0);

**for** (size\_t i = 0; i < values.size(); ++i) {

vector<**double**>& v = values[i];

**glColor3dv**(colors.values + 3 \* i);

*plotPoints*(v, maxValue, counts[i]);

}

}

**glFlush**();

}

**int** **run**(**int** argc, **char** \*argv[]) {

**glutInit**(&argc, argv);

**glutInitDisplayMode**(GLUT\_SINGLE | GLUT\_RGB);

**glutInitWindowSize**(640, 480);

**glutInitWindowPosition**(0, 0);

**glutCreateWindow**("Test");

**glutDisplayFunc**(*display*);

**glutTimerFunc**(*interval*, *timer*, 1000);

*openGLInit*();

**for** (size\_t i = 0; i < counts.size(); ++i)

counts[i] = 0;

isDone = **false**;

workingThread = std::thread(*func*, **this**);

**glutMainLoop**();

workingThread.**join**();

**return** 0;

}

**static** **const** **int** *interval* = 200;

**static** **thread\_local** RunningTimePlotter \**instance*;

};

**template**<**class** **ForwardIterator**>

**static** **void** **randUInts**(**ForwardIterator** begin, **ForwardIterator** end,

**unsigned** **int** maxValue) {

std::random\_device rd;

std::uniform\_int\_distribution<**decltype**(maxValue)> generator(0,

maxValue);

std::generate(begin, end, [&] {**return** generator(rd);});

}

**static** **void** **randUInts**(std::vector<**unsigned** **int**>& values, **unsigned** **int** count,

**unsigned** **int** maxValue) {

values.resize(count);

*randUInts*(values.begin(), values.end(), maxValue);

}

**class** SortingTime {

**unsigned** **int** start = 100;

**public**:

TTuple<2> **operator()**() {

vector<**unsigned** **int**> values;

*randUInts*(values, start, start \* 2);

**unsigned** **int** comp = 0;

**unsigned** **int** count = values.size();

//double expected = count \* 5.0 / 6.0;

//smallArrayCount = 0;

*quickSort*(&values[0], start,

[&comp](**unsigned** **int** v1, **unsigned** **int** v2) {++comp; **return** v1 < v2;});

start \*= 1.01;

**return** {(**double**)comp, 2 \* count \* log(count)};

}

};

**static** **int** **test**(**int** argc, **char** \*argv[]) {

SortingTime tmp;

RunningTimePlotter<SortingTime, 2> test(tmp, { 1.0, 0.0, 0.0, 0.0, 1.0,

0.0, });

**return** test.run(argc, argv);

}

};

**template**<**class** **T**, **int** **N**>

**thread\_local** QuickSortTest::RunningTimePlotter<**T**, **N**> \**QuickSortTest::RunningTimePlotter<T, N>::instance* =

**nullptr**;

**int** **main**(**int** argc, **char** \*argv[]) {

**return** QuickSortTest::*test*(argc, argv);

}

## Screen Shots

The Green points represents the function “2\*N\*ln(N)”,where “N” is the number of elements. The red points represents the number of comparisons used by the quick-sort algorithm, the number of elements grows by 1.01 times from left to right. We could see from the pictures that the number of comparisons is just a little less than the function “2\*N\*ln(N)”





