Solutions for Week 3

Bao Jinge

1

Here i performed experiments on five algorithms, which are QuickSort, Random QuickSort, Dual-Pivot QuickSort, Random Dual-Pivot QuickSort and standard Quicksort provided by C++ STL. Results are like the following figure.

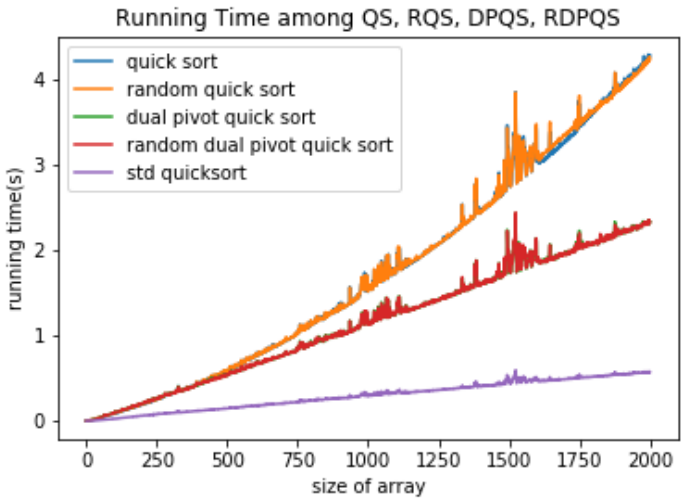


Figure 1: Running Time among QS, RQS, DPQS, RDPQS

The configuration I use in this experiment is as follows:

```
// configuration
bool FLAG_QS = true;
bool FLAG_RQS = true;
bool FLAG_DPQS = true;
bool FLAG_RDPQS = true;
bool FLAG_stdQS = true;
int times_per_data = 10; // get average running time of sorting time
int data_num = 2000; // the number of experiments
int stride = 10; // array size increasing number
int data_len_array[5000]; // array to sort
int data_repeat = 20; // numbers of test to get average running time for ecch array size
```

stride means the increment of array size each experiment. *data_num* is the number of experiments I do for different array size. *times_per_data* is to get average running time over several experiments in case of some bad performance, especially for randomized algorithm. *data_repeat* denotes the times to do repeatly for different array in same array length in case of some odd array.

We can find some interesting things.
1.QuickSort and Random QuickSort have no significant difference in my experiment. So do Dual-Pivot QuickSort and Random Dual-Pivot QuickSort.
2.As growth of array size, Dual-Pivot QuickSort will do better and better than QuickSort no matter what is randomized or not.
3.As for same array to sort, curve of Random algorithms will have same spikes. In other words, they are sensitive to some same kind of array, which make them do badly.
4.From running time, the dual-pivot quicksort algorithm is nearly twice more faster than quicksort.
5.Sort function provided by C++ STL is the fastest.

Here is some patch of my code (mainly based on Yaroslavskiy's paper [2])

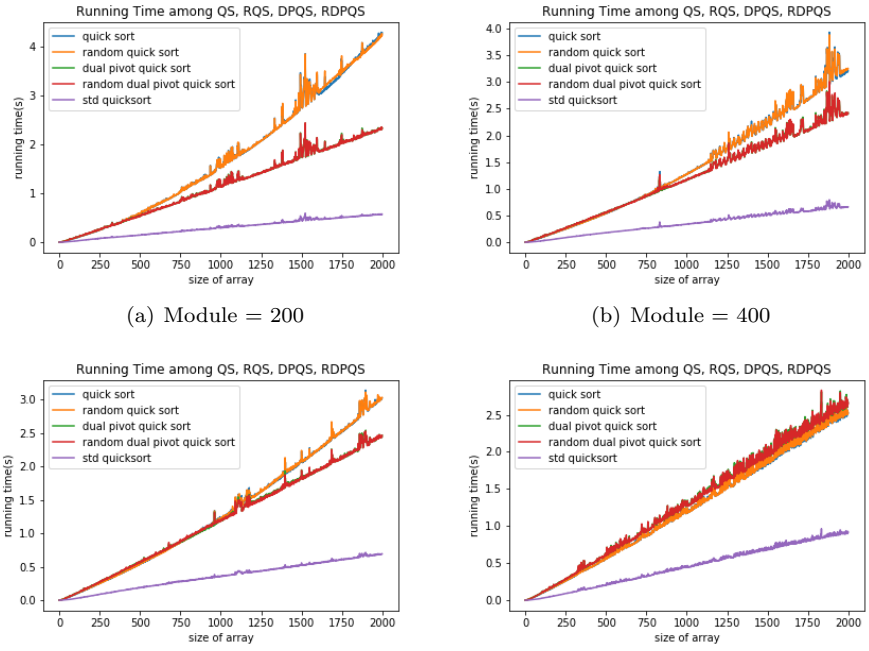
```
// Random Dual Pivot Quick Sort
pair<int,int> RDPQSPartition(int a[], int l, int r) {
    int rand_x = round(1.0*rand()/RAND_MAX*(r-l)+l);
    int rand_y = round(1.0*rand()/RAND_MAX*(r-l)+l);
    while(rand_y == rand_x) {
        rand_y = round(1.0*rand()/RAND_MAX*(r-l)+l);
    }
    swap(a[l],a[rand_x]);
    swap(a[r],a[rand_y]);
    if (a[l] > a[r])
        swap(a[l], a[r]);
    int j = l + 1;
    int g = r - 1, k = l + 1, p = a[l], q = a[r];
    while (k <= g) {
        if (a[k] < p) {
            swap(a[k], a[j]);
            j++;
        }
        else if (a[k] >= q) {
            while (a[g] > q && k < g)
                g--;
            swap(a[k], a[g]);
            g--;
            if (a[k] < p) {
                swap(a[k], a[j]);
                j++;
            }
        }
        k++;
    }
    j--;
    g++;

    swap(a[l], a[j]);
    swap(a[r], a[g]);
```

```
    return make_pair(j, g);
}
void RandomDualPivotQuickSort(int a[], int l, int r) {
    if (l < r) {
        int lp, rp;
        pair<int,int> p = DPQSPartition(a, l, r);
        lp = p.first;
        rp = p.second;
        DualPivotQuickSort(a, l, lp - 1);
        DualPivotQuickSort(a, lp + 1, rp - 1);
        DualPivotQuickSort(a, rp + 1, r);
    }
}
```

However, Dual-Pivot Quicksort doesn't always do better, epecially for some other kinds of distribution of number. For example, here I set different the modulo when code generate a test array, where will be more repeated numbers with smaller modulo. And i got some interesting findings.

```
for (int j = 0; j < data_len_array[i]; j++) {
    arr[j] = rand()%MOD_NUM;
}
```

As follow, here are results when module took 2000, 1000, 500, 400



(a) Module = 200

(b) Module = 400

(c) Module = 500

(d) Module = 1000

Figure 2: Running Time among QS, RQS, DPQS, RDPQS on module 200, 400, 500, 1000

As we can see, as module become bigger, meaning less repeated numbers, dual-pivot quicksort gradually lose his strength. In other words, Dual-Pivot Quicksort are good at coping with conditions

where there are many repeated numbers. For illustrating weird performance of dual-pivot quicksort, Sebastian Wild proposed 'The Memory Wall' and 'Scanned Elements' [1]. The former is based on device development of CPU and RAM, the latter is based on memory access. Because of viewpoint of using comparision numbers denoting time complexoty, we don't distinguish read and write accesses when analysising time complexity. Actually, it improve the performance of alogrithms.

2

Let X_{ij} indicates ball i and ball j collide or not. When collision happens, $X_{ij} = 1$, otherwise $X_{ij} = 0$, which $i, j \in [m]$ So we have

$$E(X_{ij}) = \sum_{i=1}^{n} p_i^2 = \mu_{ij} = \mu_0$$

Let X donates number of collisions, then

$$E(X) = E(\sum_{i=1}^{m}\sum_{j=i+1}^{m} X_{ij}) = \sum_{i=1}^{m}\sum_{j=i+1}^{m} E(X_{ij}) = \sum_{i=1}^{m}\sum_{j=i+1}^{m} \mu_{ij} = \frac{(m-1)m\mu_0}{2} = \mu$$

As what Chernoff bound tells, when $t \leq 0$

$$Pr(X < (1-\delta)\mu) \leq \frac{e^{\mu(e^t-1)}}{e^{t(1-\delta)\mu}}$$

So,let $\delta = 1$, we can get

$$Pr(X = 0) = P(X \leq 0) \leq e^{\mu(e^t-1)}$$

Since $Pr(X = 0) < \frac{1}{2}$ is always true, we must have

$$e^{\mu(e^t-1)} < \frac{1}{2}$$

for all $t \leq 0$. So when $t = 0$, we got the maximun of left side,

$$\begin{aligned} e^{-\mu} &< \frac{1}{2} \\ \mu &> \ln 2 \\ \frac{m(m-1)\mu_0}{2} &> \frac{1}{2} \\ m &> \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{2\ln 2}{\sum_{i=1}^{n} p_i^2}} \end{aligned} \tag{1}$$

So, we get the smallest m is

$$m = \lceil \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{2\ln 2}{\sum_{i=1}^{n} p_i^2}} \rceil$$

3

Let X_n denote the number of comparisons needs in sorting n elements. $Pr_n(X = k)$ denotes the probability for $X_n = k$. Use methods of Generating Functions, we suppose

$$G_n(x) = \sum_k Pr_n(X = k)x^k$$

As conditions, we have

$$G_n(1) = \sum_k Pr_n(X = k) = 1$$

So

$$E(X_n) = G'_n(1)$$

If we choose one pivot from arrays, we divided the array into 2 sub-arrays. Thus

$$G_n(x) = \frac{1}{n}x^{n-1}\sum_{j=1}^{n} G_{n-j}(x)G_{j-1}(x)$$

Differentiating both sides of definition of $G_n(x)$ and differentiating again, we got

$$\begin{aligned} G'_n(x) = &\frac{1}{n}(n-1)x^{n-2}\sum_{j=1}^{n} G_{j-1}(x)G_{n-j}(x) \\ &+ \frac{1}{n}x^{n-1}\sum_{j=1}^{n} G'_{j-1}(x)G_{n-j}(x) \\ &+ \frac{1}{n}x^{n-1}\sum_{j=1}^{n} G_{j-1}(x)G'_{n-j}(x) \end{aligned} \tag{2}$$

$$
\begin{aligned}
G_n''(x) = \frac{(n-1)(n-2)}{n} x^{n-3} \sum_{j=1}^{n} G_{j-1} G_{n-j} \\
+ \frac{n-1}{n} x^{n-2} \sum_{j=1}^{n} G_{j-1}'(x) G_{n-j}(x) \\
+ \frac{n-1}{n} x^{n-2} \sum_{j=1}^{n} G_{j-1}(x) G_{n-j}'(x) \\
+ \frac{n-1}{n} x^{n-2} \sum_{j=1}^{n} G_{j-1}'(x) G_{n-j}(x) \\
+ \frac{1}{n} x^{n-1} \sum_{j=1}^{n} G_{j-1}''(x) G_{n-j}(x) \\
+ \frac{1}{n} x^{n-2} \sum_{j=1}^{n} G_{j-1}'(x) G_{n-j}'(x) \\
+ \frac{n-1}{n} x^{n-2} \sum_{j=1}^{n} G_{j-1}(x) G_{n-j}'(x) \\
+ \frac{1}{n} x^{n-2} \sum_{j=1}^{n} G_{j-1}'(x) G_{n-j}'(x) \\
+ \frac{1}{n} x^{n-2} \sum_{j=1}^{n} G_{j-1}(x) G_{n-j}''(x)
\end{aligned} \tag{3}
$$

We know that

$$E(X_n) = G_n'(1) = M_n$$

and

$$G_n''(1) = (n-1)(n-2) + \frac{2}{n}(n-1) \sum_{j=1}^{n} M_{j-1} + \frac{2}{n}(n-1) \sum_{j-1}^{n} M_{n-j} + \frac{1}{n} \sum_{j=1}^{n} (G_{j-1}''(1) + G_{n-j}''(1)) + \frac{2}{n} \sum_{j=1}^{n} M_{j-1} M_{n-j}$$

Accoring to six preportities about Harmonic number,

$$\sum_{i=1}^{n} H_i = (n+1) H_n - n \tag{4}$$

$$\sum_{i=1}^{n} i H_{i-1} = \frac{n(n+1) H_{n+1}}{2} - \frac{n(n+5)}{4} \tag{5}$$

$$\sum_{i=1}^{n} i^2 H_{i-1} = \frac{6n(n+1)(2n+1) H_{n+1} - n(n+1)(4n+23)}{36} \tag{6}$$

$$\sum_{i-1}^{n} i(n-i+1) H_{n-i} = \frac{6n H_{n+1}(n^2+3n+2) - 5n^3 - 27n^2 - 22n}{36} \tag{7}$$

$$\sum_{i=1}^{n} H_i H_{n+1-i} = (n+2)(H_{n+1}^2 - H_{n+1}^{(2)}) - 2(n+1)(H_{n+1} - 1) \tag{8}$$

$$H_{n+1}^2 - H_{n+1}^{(2)} = 2\sum_{i=1}^{n} \frac{H_j}{j+1} \tag{9}$$

which, $H_n^{(2)} = \sum_{i=1}^{n} \frac{1}{i^2}$ With equation(4)-(9)We got

$$g_n''(1) = 4(n+1)^2 H_n^2 - (16n+4)(n+1)H_n + \frac{2n(23n+17)}{2} - 4(n+1)^2 H_n^{(2)}$$

Thus

$$\begin{aligned} Var(X_n) &= g_n''(1) + g_n'(1) - (g_n'(1))^2 \\ &= -4(n+1)^2 H_n^{(2)} + 7n^2 - 2(n+2)H_n + 13n \end{aligned} \tag{10}$$

As we know properity about $H_n^{(2)}$

$$\lim_{n\to\infty} H_n^{(2)} = \frac{\pi^2}{6}$$

So the variance of randomized quick sort is

$$Var(X_n) = O(n^2)$$

References

[1] Sebastian Wild. Why is dual-pivot quicksort fast?, 2015.

[2] Vladimir Yaroslavskiy. Dual-pivot quicksort algorithm. *Retrieved December*, 24:2018, 2009.