SORTING LARGE DATA FILES

BY ERICA TU LUAN FEBRUARY 11, 2016
## THE RESULT

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Bitsortgen</th>
<th>Sort (A)</th>
<th>Sort (B)</th>
<th>Sort (C)</th>
<th>Bitsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K</td>
<td>0.328</td>
<td>0.060000</td>
<td>0.040000</td>
<td>0.430909</td>
<td>10.6990909</td>
</tr>
<tr>
<td>300K</td>
<td>0.388</td>
<td>0.228182</td>
<td>0.125385</td>
<td>1.352727</td>
<td>0.7627273</td>
</tr>
<tr>
<td>500K</td>
<td>0.445</td>
<td>0.408182</td>
<td>0.207500</td>
<td>2.355455</td>
<td>0.8227273</td>
</tr>
<tr>
<td>1M</td>
<td>0.558</td>
<td>0.913636</td>
<td>0.425455</td>
<td>4.904545</td>
<td>1.003636</td>
</tr>
<tr>
<td>2M</td>
<td>0.828</td>
<td>2.099091</td>
<td>0.869091</td>
<td>9.94091</td>
<td>1.350000</td>
</tr>
<tr>
<td>3M</td>
<td>1.058</td>
<td>3.334545</td>
<td>1.324545</td>
<td>15.44273</td>
<td>1.734545</td>
</tr>
<tr>
<td>4M</td>
<td>1.328</td>
<td>4.639091</td>
<td>1.768182</td>
<td>22.78182</td>
<td>2.057273</td>
</tr>
<tr>
<td>5M</td>
<td>1.673</td>
<td>6.089091</td>
<td>2.185455</td>
<td>27.80545</td>
<td>2.630909</td>
</tr>
<tr>
<td>10M</td>
<td>2.816</td>
<td>13.66455</td>
<td>4.468182</td>
<td>62.02818</td>
<td>4.366364</td>
</tr>
</tbody>
</table>
Orange line: bitsortgen
MY DATA ACCURACY

• Each data in the form for the sorts part is calculated by the mean of 11 times of repeat experiments.
• Each data in the form for the file generating part is calculated by the mean of 5 times of repeat experiments.
• Every execution is run on the same computer.
BITSORTGEN.C
An equivalent version of Fisher–Yates shuffle/ Knuth shuffle

The algorithm:

```
-- To shuffle an array a of n elements (indices 0..n-1):
for i from 0 to n-2 do
    j ← random integer such that 0 ≤ j < n-i
    exchange a[i] and a[i+j]
```

Via Wikipedia "Fisher–Yates shuffle"

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAXN 100000000 /* Max 100 million numbers */
int x[MAXN];

int randint(int a, int b)
{
    /* return a + (RAND_MAX * rand() + rand()) % (b + 1 - a); */
    return a + (rand() % (b - a + 1));
}
int main(int argc, char *argv[]) {
    int i, k, n, t, p;
    srand((unsigned) time(NULL));
    k = atoi(argv[1]);
    n = atoi(argv[2]);
    for (i = 0; i < n; i++)
        x[i] = i;
    for (i = 0; i < k; i++) {
        p = randint(i, n);
        t = x[p]; x[p] = x[i]; x[i] = t;
        printf("%d\n", x[i]);
    }
    return 0;
}
```c
int main(int argc, char *argv[]) {
    int i, k, n, t, p;

    srand((unsigned) time(NULL));

    k = atoi(argv[1]);

    n = atoi(argv[2]);

    for (i = 0; i < n; i++)
        x[i] = i;

    for (i = 0; i < k; i++) {
        p = randint(i, n);
        t = x[p]; x[p] = x[i]; x[i] = t;
        printf("%d\n", x[i]);
    }

    return 0;
}
```
SORT(A)
## THE LINUX SYSTEM SORT

<table>
<thead>
<tr>
<th>Sort(A)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.060000</td>
<td></td>
</tr>
<tr>
<td>0.2281818</td>
<td></td>
</tr>
<tr>
<td>0.4081818</td>
<td></td>
</tr>
<tr>
<td>0.9136364</td>
<td></td>
</tr>
<tr>
<td>2.099091</td>
<td></td>
</tr>
<tr>
<td>3.334545</td>
<td></td>
</tr>
<tr>
<td>4.639091</td>
<td></td>
</tr>
<tr>
<td>6.089091</td>
<td></td>
</tr>
<tr>
<td>13.66455</td>
<td></td>
</tr>
</tbody>
</table>
The implementation in GNU Core Utilities, used on Linux, employs the merge sort algorithm.

Via Wikipedia “Unix sort”
Nonlinear regression model

model: cx4 ~ K * cy * log(cy) + B * cy + C

data: parent.frame()

K          B          C
2.212e-07  -2.205e-06  5.123e-02

residual sum-of-squares: 0.004307

The linear regression model:

Coefficients:

(Intercept)        cy
-4.445e-01  1.368e-06

residual sum-of-squares: 0.8039869
SORT(B)
#include <stdio.h>
#include <stdlib.h>

/* function to compare two integers */
int intcomp(int *x, int *y)
{
    return *x - *y;
}
/* end of intcomp */

int a[100000000]; /* the array to store 100 million integers */

int main()
{
    int i, n=0;
    /* read the data into array, a */
    while (scanf("%d", &a[n]) != EOF)
        n++;
    /* sort a using quick sort library function */
    qsort(a, n, sizeof(int), intcomp);
    /* output the data */
    for (i = 0; i < n; i++)
        printf("%d\n", a[i]);
    return 0;
} /* end of main */

What is the time complexity for a quick sort?

O(n) for looping from 0 to n-1
ANALYSTS OF ALGORITHM OF QUICK SORT

- Best case: O(nlogn)
- Worse case: O(n^2)
- Average case: O(nlogn)

Via Wikipedia
DATA ANALYSIS

<table>
<thead>
<tr>
<th>Sort(B)</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.04000</td>
<td></td>
</tr>
<tr>
<td>0.1253846</td>
<td></td>
</tr>
<tr>
<td>0.2075000</td>
<td></td>
</tr>
<tr>
<td>0.4254545</td>
<td></td>
</tr>
<tr>
<td>0.8690909</td>
<td></td>
</tr>
<tr>
<td>1.324545</td>
<td></td>
</tr>
<tr>
<td>1.768182</td>
<td></td>
</tr>
<tr>
<td>2.185455</td>
<td></td>
</tr>
<tr>
<td>4.468182</td>
<td></td>
</tr>
</tbody>
</table>

Size of file vs. Run time
Quite linear, right?

Black Line: Linear Model (By Linear regression)
Can we get a better fit model?

**Red Line:** Linear Model

**Blue Line:** Non-Linear Model
DATA ANALYSIS WITH MODELS
WHEN ZOOM IN

RED LINE: Linear Model
BLUE LINE: Non-Linear Model
THE FORMULA FOR THE TWO MODELS

**BLUE LINE: Non-Linear function** : run_time =
7.706e-09*size_of_file*log(size_of_file)+3.223e-07*size_of_file;
(The model: y=k*n*logn+b*n)
Residual sum-of-squares: 0.0007289

**RED LINE: Linear function:** run_time=4.469e-07*size_of_file+(-1.794e-02)
(The model: y=kn+b)
Residual sum-of-squares: 0.001589269
Referred to glibc/stdlib/qsort.c:

/* Order size using quicksort. This implementation incorporates four optimizations discussed in Sedgewick:

1. Non-recursive, using an explicit stack of pointer that store the next array partition to sort. To save time, this maximum amount of space required to store an array of SIZE_MAX is allocated on the stack. Assuming a 32-bit (64 bit) integer for size_t, this needs only 32 * sizeof(stack_node) == 256 bytes (for 64 bit: 1024 bytes). Pretty cheap, actually.

2. Chose the pivot element using a median-of-three decision tree. This reduces the probability of selecting a bad pivot value and eliminates certain extraneous comparisons.

3. Only quicksorts TOTAL_ELEMS / MAX_THRESH partitions, leaving insertion sort to order the MAX_THRESH items within each partition. This is a big win, since insertion sort is faster for small, mostly sorted array segments.

4. The larger of the two sub-partitions is always pushed onto the stack first, with the algorithm then concentrating on the smaller partition. This *guarantees* no more than log (total_elems) stack size is needed (actually O(1) in this case)!

*/

/** Copyright (C) 1991-2015 Free Software Foundation, Inc. 
   This file is part of the GNU C Library. 
   Written by Douglas C. Schmidt (schmidt@ics.uci.edu). */
SORT(C)
#include <iostream>
#include <set>
using namespace std;

int main()
{
    set<int> S;
    int i;
    set<int>::iterator j;
    while (cin >> i)
    {
        S.insert(i);
    }
    for (j = S.begin(); j != S.end(); ++j)
    {
        cout << *j << "\n";
    }
    return 0;
}
Sort(C)

0.4309091
1.352727
2.355455
4.904545
9.940909
15.44273
22.78818
27.80545
62.02818

DATA ANALYSIS
Still quite linear, right?

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4309091</td>
<td>1.352727</td>
</tr>
<tr>
<td>1.352727</td>
<td>2.355455</td>
</tr>
<tr>
<td>2.355455</td>
<td>4.904545</td>
</tr>
<tr>
<td>4.904545</td>
<td>9.940909</td>
</tr>
<tr>
<td>9.940909</td>
<td>15.44273</td>
</tr>
<tr>
<td>15.44273</td>
<td>22.78818</td>
</tr>
<tr>
<td>22.78818</td>
<td>27.80545</td>
</tr>
<tr>
<td>27.80545</td>
<td>62.02818</td>
</tr>
</tbody>
</table>
How does the `insert()` work?
std::set

- std::set is an associative container that contains a sorted set of unique objects of type Key. Sorting is done using the key comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as red-black trees.

RED-BLACK TREES

Self-balancing binary search tree

via Wikipedia
Theorem: A red-black tree with \( n \) keys has height \( h \leq 2 \lg(n + 1) \).

Red-black trees will never reach the worse case for the binary search tree because of the height guarantee.
Sort(C)

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.430909</td>
</tr>
<tr>
<td>1.352727</td>
</tr>
<tr>
<td>2.355455</td>
</tr>
<tr>
<td>4.904545</td>
</tr>
<tr>
<td>9.940909</td>
</tr>
<tr>
<td>15.44273</td>
</tr>
<tr>
<td>22.78818</td>
</tr>
<tr>
<td>27.80545</td>
</tr>
<tr>
<td>62.02818</td>
</tr>
</tbody>
</table>

BLUE LINE: Non-Linear Model
RED LINE: Linear Model
THE FORMULA FOR THE TWO MODELS

BLUE LINE: Non-Linear function: run_time =
8.287e-07 * file_size * log(file_size) + (-7.198e-06)* file_size + 3.902e-01
(The model: y=k*n*logn+b*n+c)
Residual sum-of-squares: 1.218

RED LINE: Linear function: run_time = 6.187e-06 * size_of_file + (-1.467e+00)
(The model: y=kn+b)
Residual sum-of-squares: 12.43855
BIT SORT
#include <stdio.h>

#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define N 100000000

int a[1 + N/BITSPERWORD];

void set(int i) { a[i>>SHIFT] |= (1<<(i & MASK)); }
void clr(int i) { a[i>>SHIFT] &= ~(1<<(i & MASK)); }
int test(int i){ return a[i>>SHIFT] & (1<<(i & MASK)); }
int main()
{
    int i;
    for (i = 0; i < N; i++)
        clr(i);
/* Replace above 2 lines with below 3 for word-parallel init */
    int top = 1 + N/BITSPERWORD;
    for (i = 0; i < top; i++)
        a[i] = 0;
*/
    while (scanf("%d", &i) != EOF)
        set(i);
    for (i = 0; i < N; i++)
        if (test(i))
            printf("%d\n", i);
    return 0;
}
<table>
<thead>
<tr>
<th>Bitsort</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6990909</td>
<td></td>
</tr>
<tr>
<td>0.7627273</td>
<td></td>
</tr>
<tr>
<td>0.8227273</td>
<td></td>
</tr>
<tr>
<td>1.003636</td>
<td></td>
</tr>
<tr>
<td>1.350000</td>
<td></td>
</tr>
<tr>
<td>1.734545</td>
<td></td>
</tr>
<tr>
<td>2.057273</td>
<td></td>
</tr>
<tr>
<td>2.630909</td>
<td></td>
</tr>
<tr>
<td>4.366364</td>
<td></td>
</tr>
</tbody>
</table>

[Graph showing a straight line with data points]
REFERENCE

sortints.cpp: /* Copyright (C) 1999 Lucent Technologies From 'Programming Pearls' by Jon Bentley */

qsortints.c: /* Copyright (C) 1999 Lucent Technologies. From 'Programming Pearls' by Jon Bentley. Modified by Deepak Kumar, January 2014 */

bitsort.c: /* Copyright (C) 1999 Lucent Technologies. From 'Programming Pearls' by Jon Bentley. Modified by Deepak Kumar, January 2014 */

bitsortgen.c/* Copyright (C) 1999 Lucent Technologies. From 'Programming Pearls' by Jon Bentley. Modified by Deepak Kumar, January 2014 */
THANK YOU