ADS:lab session #2

24, August 2015

Kamil Salakhiev

Time estimating in machine

Machine measures time in 2 ways:

- For itself, by counting ticks
- For humans, by converting ticks to date/time with taking into account leap years, leap seconds, coordination shifts (Kazan +3hrs) and network protocol for auto correlation

What about Java

- Each tick is ~10^{-9}s long (for usual CPU frequency ~1-3GHz)

In CPU it converts to elapsed nanoseconds from some moment (first CPU launching, last CPU launching…)

- In Java to get access to it *System.nanoTime()* method exists:

```
long startTime = System.nanoTime();
 // ... the code being measured ...
long estimatedTime = System.nanoTime() - startTime;
```

Another method

Another way to calculate elapsed time is *System.currentTimeMillis()* method:

```
long startTime = System.currentTimeMillis();
// ... do something ...
long estimatedTime = System.currentTimeMillis() – startTime;
```

Why long?

Storage estimating

- **Storage** refers to the data storage consumed in performing a given task, whether primary (e.g., in RAM) or secondary (e.g., on a hard disk drive)
- In Java to estimate consumed memory there is a **`Runtime.getRuntime().totalMemory()`** method, that returns the total amount of memory currently occupied for current objects measured in bytes:

```
long start = Runtime.getRuntime().totalMemory();
System.out.println("start =  " + start); // prints 64487424
int arr[] = new int[100000000];
long finish = Runtime.getRuntime().totalMemory();
System.out.println("finish = " + finish); // prints 464519168
```

The RAM model of computation

The RAM model of computation estimate algorithm according the following rules:

- Each simple operation (+, *, –, =, if, call) takes exactly one time step.
- Loops and procedures are not considered as simple operations.
- Each memory access takes exactly one time step

Example:

```
for (int i = 0; i < n; i++) {
    x++;
}
```

Takes n steps

Big O notation

- In Big O notation we are interested in the determining the order of magnitude of time complexity of an algorithm

Calculate n-th Fibonacci number (n = 0)

```
//print n-th fibonacci number                        step    n = 0
    public static void fibonacci(int n){             //  1       1
        if (n < 0)                                   //  2       1
            System.out.println("Error!");            //  3       0
        else if(n == 0)                              //  4       1
            System.out.println(0);                   //  5       1
        else if(n == 1)                              //  6       0
            System.out.println(1);                   //  7       0
        else{                                        //  8       0
            int fnm2 = 1;                            //  9       0
            int fnm1 = 0;                            //  10      0
            int fn = 0;                              //  11      0
            for (int i = 0; i < n; i++) {            //  12      0
                fn = fnm1 + fnm2;                    //  13      0
                fnm2 = fnm1;                         //  14      0
                fnm1 = fn;                           //  15      0
            }                                        //  16      0
            System.out.println(fn);                  //  17      1
        }
```

Number of steps: 5

Calculate n-th Fibonacci number (n = 1)

```java
//print n-th fibonacci number                    step     n = 1
    public static void fibonacci(int n){         //  1        1
        if (n < 0)                               //  2        1
            System.out.println("Error!");        //  3        0
        else if(n == 0)                          //  4        1
            System.out.println(0);               //  5        0
        else if(n == 1)                          //  6        1
            System.out.println(1);               //  7        1
        else{                                    //  8        0
            int fnm2 = 1;                        //  9        0
            int fnm1 = 0;                        //  10       0
            int fn = 0;                          //  11       0
            for (int i = 0; i < n; i++) {        //  12       0
                fn = fnm1 + fnm2;                //  13       0
                fnm2 = fnm1;                     //  14       0
                fnm1 = fn;                       //  15       0
            }                                    //  16       0
            System.out.println(fn);              //  17       1
        }
```

Number of steps: 6

Calculate n-th Fibonacci number (n > 1)

```
//print n-th fibonacci number                   step    n > 1
    public static void fibonacci(int n){        //  1       1
        if (n < 0)                              //  2       1
            System.out.println("Error!");       //  3       0
        else if(n == 0)                         //  4       1
            System.out.println(0);              //  5       0
        else if(n == 1)                         //  6       1
            System.out.println(1);              //  7       0
        else{                                   //  8       1
            int fnm2 = 1;                       //  9       1
            int fnm1 = 0;                       //  10      1
            int fn = 0;                         //  11      1
            for (int i = 0; i < n; i++) {       //  12      n
                fn = fnm1 + fnm2;               //  13      n-1
                fnm2 = fnm1;                    //  14      n-1
                fnm1 = fn;                      //  15      n-1
            }                                   //  16      0
            System.out.println(fn);             //  17      1
        }
```

Number of steps: $9 + n + 3(n-1) = 4n + 6$

Fibonacci number

- For n = 0 Number of steps: 5
- For n = 1 Number of steps: 6
- For n > 1 Number of steps: 4n – 6

In Big O notation we take the highest complexity in terms of order, remove constants and variables with order lower than the highest one.

Thus:

$$O\big(g(n)\big) = O(4n - 6) = n$$

Time complexities

$O(1)$	Constant (computing time)
$O(n)$	Linear (computing time)
$O(n^2)$	Quadratic (computing time)
$O(n^3)$	Cubic (computing time)
$O(2^n)$	Exponential (computing time)
$O(\log n)$	is faster than $O(n)$ for sufficiently large n
$O(n \log n)$	is faster than $O(n^2)$ for sufficiently large n

More examples

- $O(2^{n+1}) = 2^n$
- $O(2n^2 + 4n + 10) = n^2$
- $O(n \log n + n) = n \log n$
- $O(10n^3 - 5n^2 + 3n - 1) = ??$
- $O(n\sqrt{n} + n^2) = ??$
- $O(n\sqrt{n} + n^2) = ??$
- $O(n! + 2^n) = ??$

Counting sort

For array A with size n, where upper possible element equals K algorithm is the following:

```
SimpleCountingSort
    for i = 0 to k - 1
        C[i] = 0;
    for i = 0 to n - 1
        C[A[i]] = C[A[i]] + 1;
    b = 0;
    for j = 0 to k - 1
        for i = 0 to C[j] - 1
            A[b] = j;
            b = b + 1;
```

Sample output:

n = 20

k = 25

A = 12 2 22 24 22 14 6 18 10 6 3 13 17 5 8 13 24 12 22 19

C = 0 0 1 1 0 1 2 0 1 0 1 0 2 2 1 0 0 1 1 1 0 0 3 0 2

A = 2 3 5 6 6 8 10 12 12 13 13 14 17 18 19 22 22 22 24 24

Task #1

- Implement “counting sort” that sorts an array of integers
- Use `Math.Random() or r.nextInt(k)` to fill array where K is data value upper limit. **Let K = 10000**
- Implement time measurement for the algorithm. Measure time using **System.nanoTime()** for array size of 100, 1000, 10000, 100000, 1000000 elements in array.
- Vary **K** from **10000** to **100000** find the dependency of how it affects time consumption
- *__Extra task.__ Implement **counting** part of counting sort **in parallel.** Compare results

Optional homework

Make a report of done work in LaTex:

- Function graph of time/size(K) (memory)
- Your code (use package: listings)
- Your computer configuration: Processor, number of cores, frequency
- Comparison with parallel sorting - vary number of threads.
- Discuss the performance, your ideas