UTEP

COSC 2301

Lab1

Alcantar,Luis

TR 9:00-10:20

The Problem for this lab was to create 18 different methods with different complexities. Half of the methods are going to be created using recursion and the other half using iteration. The different complexities for the methods are constant, linear, quadratic, cubic, logarithmic, NLogN, N^2LogN, N^3LogN, and 2^n. Each method has to be tested with different amount of data.

There are several ways to address this problem and to perform the experiments. The first thing is to create 2 methods for each one of the complexities; one recursive and one iterative. After creating the methods, the next thing to do is perform the experiment. Just for simplicity the amounts to be used for the experiment will be 10, 100, and 1000.

Explanation of algorisms:

The constant algorithms are too simple; the iterative will just return the value of n after performing a simple formula. The recursive algorithms just go from a positive number below ten to ten, were if the number is bigger than ten it throws an exception. The linear algorithms just go from n to 0 adding n/1 +n to a variable and that is the same story for both. What distinguish the quadratic algorithms is that they are based in the idea of traversing two-dimensional array and both algorithms works for any type of matrix. The cubic method is more complex, it is based in matrix multiplication; the iterative uses 3 loops, the first and second loops are used to transverse the two matrixes and the third loop is used to perform the multiplication and store the value in the new array. The logarithmic algorithm is similar to the constant but this time, instead of incrementing by one (i=i+1) it increments by the same number (i=i+i), with that been said, N cannot be zero.

As we start with the complexities of NLogN, N^2LogN, N^3LogN the methods start to increase their complexities. The three complexities mentioned above, have very similar methods, they just add one more inner loop. For Example, the NLogN starts with a loop of linear complexity and a inner loop, to make this NLogN the inner loop must be LogN . The same way for N^2LogN three nested loops and the inner loop must be LogN. The most complicated method out of the three is the 2^N , were the simplest way to create an iterative method is having a loop from 0 to 2^n, we can make this number with Math.pow(2,n) .The recursive method is the Fibonacci science with recursion that gives a complexity of 2^n.Each method is coded to handle exceptions except the overflow because numbers above 1000 are not being used.

Experiment.

There is a method call TimeAverage() that was used for the experiments. TimeAverage contains a loop that simply adds the time that any method takes and return the sum divided by ten. The loop iterates ten times, that way the return divides by 10. The experiments were performed with each one of the methods with 10 element, 100 elements, 1000 elements. All the result were recorded and examined.

For each one of the complexities there is a graph that contains the recursive method and the iterative method.

Linear

|  |  |  |
| --- | --- | --- |
| BigO ()-Notation |  | Recursive Equation |
| BigO (1) |  | T(N)=1 |
| BigO (N) |  | T(N)=T(N-1)+1 |
| BigO (N^2) |  | T(N)=2T(N+1)+1 |
| BigO (N^3) |  | T(N)=3T(N+1)+1 |
| BigO (Log(N)) |  | T(N)=T(N/2)+1 |
| BigO (NLog(N)) |  | T(N)=2T(N/2)+1 |
| BigO (N^2Log(N)) |  | T(N)=4T(N/2)+N^2 |
| BigO (N^3Log(N)) |  | T(N)=8T(N/2)+N^3 |
| BigO (2^N) |  | T(N)=2T(N-1)+1 |

In conclusion, based on the results, the recursive method took more time that the iterative methods performed during the entire experiment. There are too many reasons for that to happen. One of the easiest one to explain or obvious reason could be that java is pure iterative so it handles better the iterative method than the recursive method. Another reason is that recursive methods need to allocate a new stack each time is call it self, and need copies of the variables for each recursive call. The only reason to use recursion could be that it is more elegant and easier to understand, and in rare cases is more practical than iterative.

Other part of the conclusion is that for methods above N^3 it can be said that it is useless to implement with large amounts of data. For the exponential methods above 24 elements it takes more than one second, for 34 we can expect to take a day at least. There is a lot to take in consideration like other programs running at the same time but I think I got a solid experiment with the result.

I certify that this project is entirely my own work. I wrote, debugged and tested the code being presented, performed the experiments and wrote the report. I also certify that I did not share my code or report or provide inappropriate assistance to any student in the class.

Code:

**import** java.util.Random;

**public** **class** Complexities {

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"1"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// constant iterative "BigO(1)"

**public** **static** **double** Iconstant() {

**return** 1 / 4 + 444;

}

// recursive"BigO(1)"

**public** **static** **double** Rconstant(**int** n) {

**if** (n < 0 || n >=10)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**if** (n == 10)

**return** 0;

**else**

**return** n + *Rconstant*(n + 1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*N\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Linear Iterative

**public** **static** **double** Ilinear(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**double** sum = 0;

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

sum += i / 2 + i;

}

**return** sum;

}

// Linear recursive T(N)=T(N-1)+1

**public** **static** **double** Rlinear(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**if** (n == 1) {

**return** 1;

} **else** {

**return** (n / 2 + n) + *Rlinear*(n - 1);

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*N^2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// N^2 Iterative

**public** **static** **int** Isquare(**int**[][] x) {

**if** (x.length == 0)

**throw** **new** IllegalArgumentException("The matrix is empty ");

**int** sum = 0;

**for** (**int** i = 0; i < x.length; i++) {

**for** (**int** j = 0; j < x[i].length; j++) {

sum += x[i][j];

}

}

**return** sum;

}

// N^2 Recursive T(N)=2T(N+1)+1

**public** **static** **int** Rsquare(**int**[][] x, **int** i, **int** j) {

**if** (x.length == 0)

**throw** **new** IllegalArgumentException("The matrix is empty ");

**if** (i == x.length)

**return** 0;

**if** (j == x[i].length)

**return** *Rsquare*(x, i + 1, j = 0);

**else**

**return** x[i][j] + *Rsquare*(x, i, j + 1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*N^3\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// N^3 Iterative

**public** **static** **int**[][] IMatrixmultiplication(**int**[][] x, **int**[][] y) {

**if** (x.length != y[0].length)

**throw** **new** IllegalArgumentException(

"The matrixes do not have the correct dimentcions to multiply ");

**int**[][] c = **new** **int**[x.length][y[0].length];

**for** (**int** i = 0; i < x.length; i++) {

**for** (**int** j = 0; j < y[0].length; j++) {

**for** (**int** k = 0; k < x[0].length; k++) {

c[i][j] += x[i][k] \* y[k][j];

}

}

}

**return** c;

}

// N^3 REcursive "T(N) = 3T(n+1) + 1

**public** **static** **int**[][] RMatrixmultiplication(**int**[][] x, **int**[][] y,

**int**[][] c, **int** i, **int** j, **int** k) {

**if** (i == x.length)

**return** c;

**if** (j == y[0].length)

**return** *RMatrixmultiplication*(x, y, c, i + 1, j = 0, k);

**if** (k == x[0].length)

**return** *RMatrixmultiplication*(x, y, c, i, j + 1, k = 0);

**else** {

c[i][j] += x[i][k] \* y[k][j];

**return** *RMatrixmultiplication*(x, y, c, i, j, k + 1);

}

}

// Helper

**public** **static** **int**[][] RMatrixmultiplication(**int**[][] x, **int**[][] y) {

**if** (x.length != y[0].length)

**throw** **new** IllegalArgumentException("The matrix is empty ");

**int**[][] c = **new** **int**[x.length][y[0].length];

**return** *RMatrixmultiplication*(x, y, c, 0, 0, 0);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*logN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// logN iterative

**public** **static** **int** ILog(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**int** sum = 0;

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

sum += i;

}

**return** sum;

}

// logN Recursive "T(N) = T(N/2) + 1

**public** **static** **int** RLog(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**if** (n == 0)

**return** 0;

**else**

**return** 0 + *RLog*(n / 2);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*NLogN\*\*\*

// NLogN Iterative

**public** **static** **void** INlogn(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

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

**for** (**int** j = 1; j < n; j += j) {

// System.out.println("");

}

}

}

// NLogN Recursive "T(N) = 2T(n/2) + n"

**public** **static** **void** RNlogn(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**if** (n == 0) {

// System.out.println("l");

} **else** {

*RNlogn*(n / 2);

// System.out.println("l");

*RNlogn*(n / 2);

}

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

// System.out.printl("");

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*N^2LogN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// N^2LogN Iterative

**public** **static** **void** IN2logn(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

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

**for** (**int** j = 0; j < n; j++) {

**for** (**int** k = 1; k < n; k = k + k) {

// System.out.println();

}

}

}

}

// N^2LogN Recursive "T(N) = 4T(n/2) + n^2"

**public** **static** **void** RN2logn(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**if** (n > 0) {

**for** (**int** i = 0; i < 4; i++) {

*RN2logn*(n / 2);

// System.out.println(n);

}

}

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

**for** (**int** j = 0; j < n;) {

j++;

}

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*N^3LogN\*\*\*\*\*

// N^3LogN Iterative

**public** **static** **int** IN3logn(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**int** sum = 1;

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

**for** (**int** j = 0; j < n; j++) {

**for** (**int** k = 0; k < n; k++) {

**for** (**int** c = n; c > 0; c = c / 2) {

// sum+=1;

}

}

}

}

**return** sum;

}

// N^3LogN Recursive "T(N) = 8T(n/2) + n^3"

**public** **static** **void** RN3logn(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**if** (n > 0)

**for** (**int** i = 0; i < 8; i++) {

*RN2logn*(n / 2);

// System.out.println(n);

}

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

**for** (**int** j = 0; j < n; j++) {

**for** (**int** k = 0; k < n;) {

k++;

}

}

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*2^n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 2^n iterative

// because of time issues i crate a simple method

**public** **static** **void** I2n(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**for** (**int** i = 0; i < Math.*pow*(2, n); i++) {

// System.out.println("2^n");

}

}

// 2^n recursive

// fibonachi recursive

**public** **static** **int** R2n(**int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException(

"Please enter a positive nuber x > 0");

**if** (n == 1 || n == 2) {

**return** 1;

} **else** {

**return** *R2n*(n - 2) + *R2n*(n - 1);

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

\* help to print the 2 dimensional arrays , to proof they are correct.

\*/

**public** **static** **void** print(**int**[][] x) {

**for** (**int** i = 0; i < x.length; i++) {

**for** (**int** j = 0; j < x[i].length; j++) {

System.*out*.print(x[i][j] + "\t");

}

System.*out*.println("");

}

}

/\*

\* TimeAverage is used for testing it computers a method 10 times and add

\* the time for each of the iteration

\*

\* @return sum/10 sum is the addition of the iteration and 10 the ten times

\* , returns the average time .

\*/

**public** **static** **double** TimeAverage() {

**double** sum = 0;

**long** time;

// //int[][] matrix = { { 15, 59, 6, 4, 5, 6, 7, 8 },

// { 9, 32, 43, 3, 87, 65, 32, 23 }, { 15, 59, 6, 4, 5, 6, 7, 8 },

// { 9, 32, 43, 3, 87, 65, 32, 23 }, { 15, 59, 6, 4, 5, 6, 7, 8 },

// { 9, 32, 43, 3, 87, 65, 32, 23 }, { 15, 59, 6, 4, 5, 6, 7, 8 },

// { 9, 32, 43, 3, 87, 65, 32, 23 } };

// int[][] a = new int[6][4];

// int[][] b = new int[6][6];

**int**[] nln = **new** **int**[10000];

nln = *fill*(nln);

**for** (**int** i = 0; i < 10; i++) {

time = System.*nanoTime*();

// method

*IN3logn*(100);

sum += System.*nanoTime*() - time;

}

**return** sum / 10;

}

// helps to test the Array.sort in method N^3LogN filling a array with

// random numbers

**public** **static** **int**[] fill(**int**[] x) {

Random r = **new** Random();

**for** (**int** i = 0; i < x.length; i++) {

x[i] = r.nextInt(100);

}

**return** x;

}

**public** **static** **void** main(String[] args) {

**long** time;

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Constant\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.*out*.println("");

time = System.*nanoTime*();

System.*out*.print(*Iconstant*());// "20"time average 1000.0M-Time"200" time

// average 4300.0M-Time 2000 time

// average 55400.0.0M

System.*out*.println(" -> The time for constant iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

System.*out*.print(*Rconstant*(1

System.*out*.println(" -> The time for constant Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Linear\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

time = System.*nanoTime*();

System.*out*.print(*Ilinear*(10));

System.*out*.println(" -> The time for Linear iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

System.*out*.print(*Rlinear*(10

System.*out*.println(" -> The time for Linear Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*square\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

time = System.*nanoTime*();

**int**[][] matrix = { { 15, 59, 26, 87 }, { 9, 32, 43, 3 } };

System.*out*.print(*Isquare*(matrix

System.*out*.println(" -> The time for Square iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

System.*out*.print(*Rsquare*(matrix, 0, 0

System.*out*.println(" -> The time for Square Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Cubic\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

**int**[][] a = **new** **int**[3][3];

**int**[][] b = **new** **int**[3][3];

a[0][0] = 1;

a[1][0] = 4;

a[2][0] = 6;

a[0][1] = 1;

a[1][1] = 4;

a[2][1] = 6;

a[0][2] = 1;

a[1][2] = 4;

a[2][2] = 6;

b[0][0] = 2;

b[0][1] = 5;

b[0][2] = 7;

b[1][0] = 3;

b[1][1] = 8;

b[1][2] = 9;

b[2][0] = 3;

b[2][1] = 8;

b[2][2] = 9;

time = System.*nanoTime*();

*IMatrixmultiplication*(b, a System.*out*.println(" -> The time for Cubic iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

*RMatrixmultiplication*(b, a System.*out*.println(" -> The time for Cubic Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

// print(l);//testing purposes

// print(k);//testing purposes

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*LogN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

time = System.*nanoTime*();

*ILog*(1000);

System.*out*.println(" -> The time for Log iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

*RLog*(1000);

System.*out*.println(" -> The time for Log Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*NLogN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

time = System.*nanoTime*();

*INlogn*(1000);

System.*out*.println(" -> The time for NLogN iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

*RNlogn*(1000);

System.*out*.println(" -> The time for NLogN Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*N^2LogN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

time = System.*nanoTime*();

*IN2logn*(10);

System.*out*.println(" -> The time for N^2LogN iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

*RN2logn*(10);

System.*out*.println(" -> The time for N^2LogN Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*N^3LogN\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

time = System.*nanoTime*();

*IN3logn*(1000);

System.*out*.println(" -> The time for N^3LogN iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

*RN3logn*(1000);

System.*out*.println(" -> The time for N^3LogN Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*2^N\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

time = System.*nanoTime*();

*I2n*(10);

System.*out*.println(" -> The time for 2^N iterative is "

+ (System.*nanoTime*() - time) + " Miliseconds");

time = System.*nanoTime*();

*R2n*(10);

System.*out*.println(" -> The time for 2^N Recursive is "

+ (System.*nanoTime*() - time) + " Miliseconds");

}

}