Activity 1. Some Iterative Models

Classes **Loop1.java, Loop2.java, Loop3.java and Loop4.java** are provided.

**Loop1.java**

long cont = 0;

long n1 = 1;

while (n1 <= n \* n) { // log3(n^2)

for (long i = 1; i <= 2 \* n; i += 3) // 2n/3

cont++;

n1 = 3 \* n1;

}

// total complexity -> log3(n^2)\*2n/3 -> O(nlog(n))

return cont;

**Loop2.java**

long cont = 0;

long n1 = n;

do {

for (long i = 1; i <= n; i++) // n

for (long j = n; j >= 0; j -= 2) // n/2

cont++;

n1 = n1 / 3;

} while (n1 >= 1); // log3(n)

// total complexity -> log3(n)\*n\*n/2 -> O(n^2log(n))

return cont;

**Loop3.java**

long cont = 0;

long i = 1;

while (i <= 2 \* n) { // 2n

for (long j = i; j >= 0; j -= 2) // 2n/2

for (long k = 1; k <= n; k \*= 2) // log2(n)

cont++;

i++;

}

// total complexity -> 2n\*2n/2\*log2(n) -> O(n^2log(n))

return cont;

**Loop4.java**

long cont = 0;

for (int i = 1; i <= n; i++) // n

for (int j = 1; j <= i; j++) // n

for (int k = 1; k <= j; k++) // n

cont++;

// total complexity -> n\*n\*n -> O(n^3)

return cont;

After the execution of the different algorithms, we can measure their times and observe that, in order, from loop 1 to loop 4 the time increases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **tLoop1** | **tLoop2** | **tLoop3** | **tLoop4** |
| **100** | 0.0099 | 0.367 | 1.27 | 1.24 |
| **200** | 0.0207 | 1.37 | 5.44 | 9.73 |
| **400** | 0.0452 | 6.39 | 26.85 | 78.51 |
| **800** | 0.123 | 28.3 | 113 | 638 |
| **1600** | 0.359 | 115 | 503 | 5244 |
| **3200** | 0.478 | 521 | 2168 | 37901 |
| **6400** | 1.02 | 2059 | 9142 | Oot |
| **12800** | 2.37 | 9216 | 36909 | Oot |
| **25600** | 4.83 | 41427 | Oot | Oot |
| **51200** | 10.13 | Oot | Oot | Oot |

Knowing that:

**loop1** has O(n log(n)) complexity

**loop2** has O(n2 log(n)complexity

**loop3** has O(n2 log(n)) complexity

**loop4** has O(n3) complexity

We can observe that the increase of these algorithm theoretical complexities directly reflects into our time measurements

Activity 2. Creation of iterative models of a given time complexity

We're asked to create new **Loop5.java, Loop6.java and Loop7.java** classes implementing new loops that satisfies the following theoretical time complexities:

**loop5** has O(n2 log2(n)) complexity

**loop6** has O(n3 log(n)) complexity

**loop7** has O(n4) complexity

The following implementations are my approach for the problem:

**Loop5.java**

long cont = 0;

long n1 = n;

do {

for (long i = 1; i <= n; i++) // n

for (long j = n; j >= 0; j -= 2) // n/2

for(long k = 1; k <= n; k \*= 2) // log2(n)

cont++;

n1 = n1 / 2;

} while (n1 >= 1); // log2(n)

// total complexity -> log2(n)\*log2(n)\*n\*n/2 -> O(n^2log^2(n))

return cont;

**Loop6.java**

long cont = 0;

long n1 = n;

do {

for (long i = 1; i <= n; i++) // n

for (long j = n; j >= 0; j -= 2) // n/2

for(long k = 0; k <= n; k++) // n

cont++;

n1 = n1 / 2;

} while (n1 >= 1); // log2(n)

// total complexity -> log2(n)\*n\*n\*n/2 -> O(n^3log(n))

return cont;

**Loop7.java**

long cont = 0;

for (int i = 1; i <= n; i++) // n

for (int j = 1; j <= i; j++) // n

for (int k = 1; k <= j; k++) // n

for (int m = 1; m <= k; m++) // n

cont++;

// total complexity -> n\*n\*n\*n -> O(n^4)

return cont;

Then, the table of times (in milliseconds without Optimization) that we obtain from the execution of this algorithms without optimizations is:

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **tLoop5** | **tLoop6** | **tLoop7** |
| **100** | 5.6 | 51.1 | 40.2 |
| **200** | 30 | 482 | 552 |
| **400** | 109 | 3897 | 8323 |
| **800** | 513 | 34985 | Oot |
| **1600** | 3069 | Oot | Oot |
| **3200** | 12940 | Oot | Oot |
| **6400** | 57776 | Oot | Oot |

In this case, as we increase the theoretical complexities of our algorithms, the time increases again as O(n2 log2(n)) < O(n3 log(n)) < O(n4)

Activity 3. Comparison of two algorithms (A)

To compare two algorithms we can calculate the **division ratio** of the execution time for the same size of the problem.

If the complexities are different, the ratio can either:

* Tend to 0 if the numerator has the **best** complexity
* Tend to ∞ if the numerator has the **worst** complexity

If the size of the problem grows and the algorithms has the same complexity, it will tend to a constant that will be:

* < 1 if the algorithm in the numerator is better
* > 1 if the algorithm in the denominator is better
* = 1 if the algorithms are exactly equal

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **tLoop1** | **tLoop2** | **t1/t2** |
| **100** | 0.0099 | 0.367 | 0.02698 |
| **200** | 0.0207 | 1.37 | 0.01511 |
| **400** | 0.0452 | 6.39 | 0.00707 |
| **800** | 0.123 | 28.3 | 0.00434 |
| **1600** | 0.359 | 115 | 0.00312 |
| **3200** | 0.478 | 521 | 0.00092 |
| **6400** | 1.02 | 2059 | 0.0005 |
| **12800** | 2.37 | 9216 | 0.00026 |
| **25600** | 4.83 | 41427 | 0.00012 |
| **51200** | 10.13 | Oot | ------------------------ |

As it can be seen, the division ratio tends to zero. According to the previous rules, when the division ratio tends to zero, we can conclude that the theoretical time complexity of the algorithm in the numerator (tLoop1 O(n log(n))) is better than the theoretical time complexity of the algorithm in the denominator (tLoop2 O(n2 log(n)))

Activity 4. Comparison of two algorithms (B)

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **tLoop3** | **tLoop2** | **t3/t2** |
| **100** | 1.27 | 0.367 | 3.46049 |
| **200** | 5.44 | 1.37 | 3.97080 |
| **400** | 26.85 | 6.39 | 4.20188 |
| **800** | 113 | 28.3 | 3.99293 |
| **1600** | 503 | 115 | 4.37391 |
| **3200** | 2168 | 521 | 4.16123 |
| **6400** | 9142 | 2059 | 4.44002 |
| **12800** | 36909 | 9216 | 4.00488 |
| **25600** | Oot | 41427 | ------------------------ |
| **51200** | Oot | Oot | ------------------------ |

As it can be seen, the division rate tends (approx) to the constant 4, that is greater than zero. When the division rate tends to a constant, according to the previous rules, we can conclude that both algorithms have the same theoretical time complexity. Knowing that the constant is greater than 1, we can conclude that the algorithm in the numerator (tLoop3 O(n2 log(n))) is worst than the algorithm in the denominator (tLoop2 O(n2 log(n)))

Activity 5. Comparison of two algorithms (C)

We can also compare two algorithms exactly equal but written in different programming languages. In this case, we can execute the same algorithm in python and the same algorithm in Java with optimization.

Note: As these measurements were taken on my personal computer at home (Lenovo Laptop

with a AMD Ryzen 7 5800H), I'll repeat all of them again.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N** | **tLoop4**  **(Python) -**  **t41** | **tLoop4 (Java**  **without**  **optimization) - t42** | **tLoop4 (Java with**  **optimization) - t43** | **t42/t41** | **t43/t42** |
| **200** | 83 | 26.9 | 0.343 | 0.324096 | 0.012751 |
| **400** | 553 | 198 | 1.08 | 0.358047 | 0.005455 |
| **800** | 4940 | 1506 | 6.19 | 0.304858 | 0.004110 |
| **1600** | 37992 | 12077 | 55 | 0.317883 | 0.004554 |
| **3200** | Oot | Oot | 377 | -------------- | -------------- |
| **6400** | Oot | Oot | 2422 | -------------- | -------------- |

As it can be see in the table, it's not a surprise that both division ratios tend to a constant as we knew that the algorithms had the same complexity. Therefore:

* **t42/t41**:

This division ratio compares the algorithm in Java without optimization (t42) at the numerator and the algorithm in Python (t41) at the denominator. As the constant is lower than one (0.3 < 1), we can state that although they have the same complexity, the algorithm in the numerator (The Java one), has a better performance than the one in Python.

* **t43/t42**:

This division ratio compares the algorithm in Java with optimization (t43) at the numerator and the algorithm in Java without optimization (t42) at the denominator. As the constant is lower than one, we can state that although both algorithms have the same complexity, the one in the numerator (t43) has better performance than the other.