Activity 1. Session0

Factor 1 and 2: Meditions of PythonA1

|  |  |  |
| --- | --- | --- |
| Executions | A1 | A1 |
| 10000 | 2608 | 2545 |
| 20000 | 10866 | 9277 |
| 40000 | 44849 | 31475 |
| 80000 | 168405(OOT) | (OOT)127519 |
| 16000 | OOT | OOT |
| 320000 | OOT | OOT |
| 640000 | OOT | OOT |
| 128000 | OOT | OOT |

|  |  |  |
| --- | --- | --- |
| Memory | Computer A | Computer B |
| CPU | 3.6 Hz | 1,90 HZ |
| RAM | 8 GB | 8 GB |

Computer A is the lab computer and Computer B my personal one.

Data in green represents that the measurements were taken on computer A, data in orange represents those taken on computer B.

Factor 3: Comparison between PythonA1 and JavaA1

|  |  |  |  |
| --- | --- | --- | --- |
| Executions | A1 | A1(Java OP) | A1(Java NO OP) |
| 10000 | 2545 | 121 | 717 |
| 20000 | 9277 | 621 | 3461 |
| 40000 | 31475 | 2483 | 12497 |
| 80000 | (OOT)127519 | 9453 | 43676 |
| 16000 | OOT | 31797 | (OOT)168053 |
| 320000 | OOT | (OOT)121329 | OOT |
| 640000 | OOT | OOT | OOT |
| 128000 | OOT | OOT | OOT |

Farctor 4: Table reflecting the execution times of the modules PythonA1.py, PythonA2.py and PythonA3.py

|  |  |  |  |
| --- | --- | --- | --- |
| Executions | A1 | A2 | A3 |
| 10000 | 2545 | 300 | 136 |
| 20000 | 9277 | 1101 | 541 |
| 40000 | 31475 | 3514 | 2060 |
| 80000 | (OOT)127519 | 12378 | 6943 |
| 16000 | OOT | 44967 | 21624 |
| 320000 | OOT | (OOT)168734 | (OOT)82281 |
| 640000 | OOT | OOT | OOT |
| 128000 | OOT | OOT | OOT |

Execution times of the classes JavaA1.java, JavaA2.java and JavaA3.java, WITHOUT OPTIMIZATION

|  |  |  |  |
| --- | --- | --- | --- |
| Executions | A1(Java NO OP) | A2(Java NO OP) | A3(Java NO OP) |
| 10000 | 717 | 96 | 60 |
| 20000 | 3461 | 420 | 296 |
| 40000 | 12497 | 1537 | 951 |
| 80000 | 43676 | 5361 | 3389 |
| 16000 | (OOT)168053 | 17147 | 11567 |
| 320000 | OOT | 57761 | 37217 |
| 640000 | OOT | (OOT)210287 | (OOT)133582 |
| 128000 | OOT | OOT | OOT |

Execution times of the classes JavaA1.java, JavaA2.java and JavaA3.java, WITH OPTIMIZATION

|  |  |  |  |
| --- | --- | --- | --- |
| Executions | A1(Java OP) | A2(Java OP) | A3(Java OP) |
| 10000 | 121 | 17 | 9 |
| 20000 | 621 | 102 | 68 |
| 40000 | 2483 | 288 | 164 |
| 80000 | 9453 | 981 | 511 |
| 16000 | 31797 | 3586 | 1822 |
| 320000 | (OOT)121329 | 12107 | 6779 |
| 640000 | OOT | 40205 | 21252 |
| 128000 | OOT | (OOT)147357 | OOT)74628 |

Conclusions

It is clear that Python is the slowest of the three ways, followed by Java without optimization, being optimized java the fastest way by far.

Activity 2. Session1.1

How many more years we can continue using this way of counting?

The maximum value of a float is: 9223372036854775807.

The starting value of currentTimeMillis() is Jan 1, 1970

The provided date would be 292271023.05 years in advance to 1970.

The max year would be: 9223370329439789075 ms from now.

In years it would be 292278994. Exactly Sun Aug 17 08:12:55 CET 292,278,994.

What does it mean that the time measured is 0?

It means that n is too small to get an accurate measurement.

From what size of problem (n) do we start to get reliable times?

From n= 10,000,000 vector 2 is reliable

What happens with the time if the problem size is multiplied by 2?

The complexity of the problem would escalate by a factor of 2, f.e O(n) it would result on O(2n) -> 2t. If it was O(n^2) it would escalate to 4t, if it was O(n^3) it would escalate to 8t, and so on…

What happens with the time if the problem size is multiplied by a value k other than 2? (try it, for example, for k=3 and k=4 and check the times obtained)

For 3 and 4 it would result on 3t and 4t respectively, if it was linear.

TABLE1 (times in milliseconds WITHOUT OPTIMIZATION):

|  |  |  |
| --- | --- | --- |
| n | Tsum | Tmaximum |
| 100000 | 151 | 154 |
| 200000 | 297 | 310 |
| 400000 | 695 | 592 |
| 800000 | 1276 | 1264 |
| 1600000 | 2115 | 2425 |
| 3200000 | 3767 | 4648 |
| 6400000 | 7393 | 9177 |
| 1280000 | 14062 | 17509 |
| 2560000 | 27054 | 32829 |
| 5120000 | 50658 | (OOT)62313 |
| 10240000 | (OOT)97622 | OOT |
| 20480000 | OOT | OOT |
| 40960000 | OOT | OOT |
| 81920000 | OOT | OOT |

TABLE2 (times in milliseconds WITHOUT OPTIMIZATION):

|  |  |  |
| --- | --- | --- |
| n | Tmatches1 | TMatches2 |
| 100000 | (OOT)710000 | 238 |
| 200000 | (OOT)2951000 | 382 |
| 400000 | (OOT)11933000 | 818 |
| 800000 | (OOT)47397000 | 1623 |
| 1600000 | OOT | 3101 |
| 3200000 | OOT | 6548 |
| 6400000 | OOT | 13033 |
| 1280000 | OOT | 25296 |
| 2560000 | OOT | 53244 |
| 5120000 | OOT | (OOT)109634 |
| 10240000 | OOT | OOT |
| 20480000 | OOT | OOT |
| 40960000 | OOT | OOT |
| 81920000 | OOT | OOT |

All of them were measured using n=1000 except matches1 that was measured

with n=1 and then conversed accordingly.

Activity 3. Session1.2

For both table1 and 2 taken into reference and converted if necessary to n=1000

TABLE1 (times in milliseconds and WITHOUT OPTIMIZATION):

For Loop1.java, -> n log n

Loop2.java -> n^2 log n

Loop4.java -> n^3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | tLoop1 | tLoop2 | tLoop3 | tLoop4 |
| 100 | 14,1 | 197 | 2014 | 2378 |
| 200 | 32,7 | 1961 | 7999 | 14673 |
| 400 | 68,2 | 9375 | 27867 | (OOT)83004 |
| 800 | 160,5 | 44574 | OOT | OOT |
| 1600 | 342,7 | (OOT)164291 | OOT | OOT |
| 3200 | 628,3 | OOT | OOT | OOT |
| 6400 | 1228,7 | OOT | OOT | OOT |
| 12800 | 2759,4 | OOT | OOT | OOT |
| 25600 | 5822,9 | OOT | OOT | OOT |
| 51200 | OOT | OOT | OOT | OOT |

The results seem correct, as the files were ordered in ascending complexity. Being loop1 the best algorithm, complexity-wise.

TABLE2 (times in milliseconds and WITHOUT OPTIMIZATION):

For Loop5.java, -> n^2 log^2 n

Loop6.java -> n^3 log n

Loop7.java -> n^4

|  |  |  |  |
| --- | --- | --- | --- |
| n | tLoop5 | tLoop6 | tLoop7 |
| 100 | 214 | (OOT)25300 | (OOT)57300 |
| 200 | 765 | (OOT)243300 | (OOT)674700 |
| 400 | 3422 | (OOT)1467900 | (OOT)7706300 |
| 800 | 10416 | OOT | OOT |
| 1600 | 36358 | OOT | OOT |
| 3200 | OOT | OOT | OOT |
| 6400 | OOT | OOT | OOT |
| 12800 | OOT | OOT | OOT |
| 25600 | OOT | OOT | OOT |
| 51200 | OOT | OOT | OOT |

(In this case orange just mean those of which its code was not already provided).

The results seem correct, as the files were ordered in ascending complexity,

n^2 log^2 n < n^3 log n < n^4

Tables 3 and 4 use a factor of conversion for n=1

TABLE3 (times in milliseconds and WITHOUT OPTIMIZATION):

|  |  |  |  |
| --- | --- | --- | --- |
| n | tLoop1 | tLoop2 | t1/t2 |
| 100 | 0,0141 | 0,197 | 0,0715736 |
| 200 | 0,0327 | 1,961 | 0,01667517 |
| 400 | 0,0682 | 9,375 | 0,00727467 |
| 800 | 0,1605 | 44,574 | 0,00360075 |
| 1600 | 3,4257 | 164,291 | 0,02085142 |
| 3200 | 6283 | OOT |  |
| 6400 | 1,2287 | OOT |  |
| 12800 | 2,7594 | OOT |  |
| 25600 | 5,8229 | OOT |  |
| 51200 | OOT | OOT |  |

The result is logical, as loop1 is n log n and loop2 is n^2 log n as the n gets bigger, the time gap between both increases faster.

TABLE4 (times in milliseconds and WITHOUT OPTIMIZATION):

|  |  |  |  |
| --- | --- | --- | --- |
| n | tLoop3 | tLoop2 | t3/t2 |
| 100 | 2,014 | 0,197 | 10,2233503 |
| 200 | 7,999 | 1,961 | 4,07904131 |
| 400 | 27,867 | 9,375 | 2,97248 |
| 800 | OOT | 44,574 |  |
| 1600 | OOT | 164,291 |  |
| 3200 | OOT | OOT |  |
| 6400 | OOT | OOT |  |
| 12800 | OOT | OOT |  |
| 25600 | OOT | OOT |  |
| 51200 | OOT | OOT |  |

Same algorithm in different development environments

In table 5 the approach of n=1 is also applied.

TABLE5 (times in milliseconds):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | tLoop4(PY)-t41 | tLoop4(Java No OP)-t42 | tLoop4(Java OP)-t43 | t42/t41 | t43/t42 |
| 200 | 7,4 | 2,39 | 0,131 | 0,32297297 | 0,05481172 |
| 400 | 56,6 | 17,04 | 0,797 | 0,30106007 | 0,0467723 |
| 800 | 434,5 | 99,01 | 4,986 | 0,22787112 | 0,05035855 |
| 1600 | 2509,2 | 623,54 | 31,987 | 0,24850151 | 0,05129903 |
| 3200 | OOT | OOT | OOT |  |  |
| 6400 | OOT | OOT | OOT |  |  |

As seen in other sessions, we can clearly see that python is the slower option, followed by java without compilation and as always Java making use of the compiler is the fastest one.