ES215 - COA

Assignment 4

Siddhesh Kanawade, 20110199

Github repository

Question 1:

Q1> 5 stage pipeline with each stage taking one cycle. We have to assume 5 different stages, hence we assume:

→

IF	ID	EX	MEM	WB
200ps	100ps	200ps	400ps	100ps

source: Assignment 3

→ Baseline is Single cycle interaction.

* Basic calculation:

$$T_{net,single\ cycle} = IF + ID + EX + EM + WB$$
$$= 200 + 100 + 200 + 400 + 100$$
$$= \underline{1000ps}$$

For N interactions =>

$$\boxed{T_{net} = 1000N}$$

* Time taken for pipeline:

a) → 30% Raw & 20% branch dependency:

$$T_{clock\ cycle,net} = \max\{\text{Time by stages}\} = 400ps$$

For dependencies:

$$CPI = \Sigma CPI_{per\ inst.} \times fraction$$
$$= (1-0.2-0.3) \times 1 + 0.5 \times 4 + 0.2 \times 3$$
$$= 0.5 \times 1 + 0.3 \times 4 + 0.2 \times 3$$
$$= 0.5 + 1.2 + 0.6$$
$$= \underline{2.3}$$

$$T_{exec} = IC \times CPI \times ClockCycle$$

$$= N \times 2.3 \times 400$$

$$T_{exec} = \underline{920\ N\ ps}$$

$$\therefore Speedup = \frac{T_{baseline}}{T_{pipeline}} = \frac{1000}{920} = \underline{1.087.}$$

b) → 40% branch dependency:

$$CPI = (1-0.4)\times 1 + 0.4\,(1 + stall_{branch})$$

$$= 0.6 \times 1 + 0.4 \times (1+2)$$

$$CPI = 0.6 + 1.2 = 1.8$$

$$\therefore T_{exec} = IC \times CPI \times ClockCycleTime$$

$$= N \times 1.8 \times 400ps$$

$$= 720\ N\ ps.$$

$$\therefore Speedup = \frac{1000N}{720N} = \underline{1.3889}$$

⇒ Branch predictor with 80% accuracy.

a) 30% RAW, 20% branch

On calculation,

4% branch, 30% RAW, 66% Normal.

$$CPI = 0.66 \times 1 + 0.3\,(1 + stall_{RAW}) + 0.04\,(1 + stall_{branch})$$

$$= 0.66 \times 1 + 0.3\,(1+3) + 0.04\,(1+2)$$

$$= 0.66 + 1.2 + 0.12$$

$$= 1.98$$

$T_{exec} = N \times 1.98 \times 400 = 792N$ ps.

$\therefore$ Speedup $= \frac{1000}{792} = \underline{\underline{1.2626}}$

$\Rightarrow$ Speedup increased from the previous value (without branch predictor).

$\Rightarrow$ 40% branch dependency:

$\Rightarrow$ 8% branch, 92% Normal.

$\therefore$ CPI $= 0.92 \times 1 + 0.08 \times (1 + \text{stall}_{branch})$

$= 0.92 + 0.08(1+2)$

$= 0.92 + 0.24 = \underline{\underline{1.16}}$

$\therefore T_{exec} = N \times 1.16 \times 400 = 464N$ ps

$\therefore$ Speedup $= \frac{1000}{464} = \underline{\underline{2.156}}$

$\Downarrow$

increased from previous value, where there isn't branch predictor.

Question 2:

20% branch instruction.

Delayed branching with one delay slot.

Base CPI = 1.5 ... (No delay slots)

$$CPI_{base} = CPI_{nonbranch} \times f_{nonbranch} + CPI_{branch} \times f_{branch}$$

Thus,

$$1.5 = CPI_{nonbranch} \times 0.8 + CPI_{branch} \times 0.2 \quad \text{(1)}$$

* 85% delayed slots are filled:

∴ Fraction of branch instruction where delayed slot is filled ⇒ $0.2 \times 0.85 = 0.17$.

∴ fraction of branch instruction (with delay slot = empty) $= 0.2 \times 0.15 = 0.03$

$$CPI_{new} = 0.8 \times CPI_{non\text{-}branch} + 0.2 \times CPI_{branch} - 0.17 \quad \text{(2)}$$

from (1) & (2), we get

$$CPI_{new} = 1.5 - 0.17 = 1.33$$

Question 3:
The code for the problem is present in the attached github repository. We could note that we have 3 nested for loops and we can interchange them maintaining the precision of the solution. Following are the combinations of the loop and their performance

Note: *Time in Nanoseconds.*

I, J, K

N	128	256	512
Iteration 1	11238176	87735141	738401854
Iteration 2	10315913	95228120	724739624
Iteration 3	10026547	88626850	716162352
Iteration 4	10444267	86697339	712669493
Iteration 5	10741363	114325474	734300589
Median	**10444267**	**88626850**	**724739624**

I, K, J

N	128	256	512
Iteration 1	9573563	59786247	420068011
Iteration 2	9467350	59430480	425174818
Iteration 3	11856721	58671657	419498810
Iteration 4	9306361	58621853	386511881
Iteration 5	9381577	59718417	420470088
Median	**9467350**	**59430480**	**420068011**

K, I, J

N	128	256	512
Iteration 1	9981769	58333633	422652882
Iteration 2	9305010	61117587	431034842

Iteration 3	8876656	59704946	426423398
Iteration 4	10132701	59349419	423859587
Iteration 5	8090361	62369342	422590950
Median	**9305010**	**59704946**	**423859587**

K, J, I

N	128	256	512
Iteration 1	13794999	102065793	814084227
Iteration 2	9774119	97619202	798926941
Iteration 3	13413033	103121638	852852350
Iteration 4	12716718	98763877	817270906
Iteration 5	10712025	108070460	811043671
Median	**12716718**	**102065793**	**814084227**

J, K, I

N	128	256	512
Iteration 1	12023385	103437377	812519930
Iteration 2	11741502	107421633	823129128
Iteration 3	13631846	101075464	828516363
Iteration 4	13316720	100282157	817735952
Iteration 5	12827583	103077558	797951845
Median	**12827583**	**103077558**	**817735952**

J, I, K

N	128	256	512
Iteration 1	9882507	89970305	750988468
Iteration 2	10625240	87557448	728962395

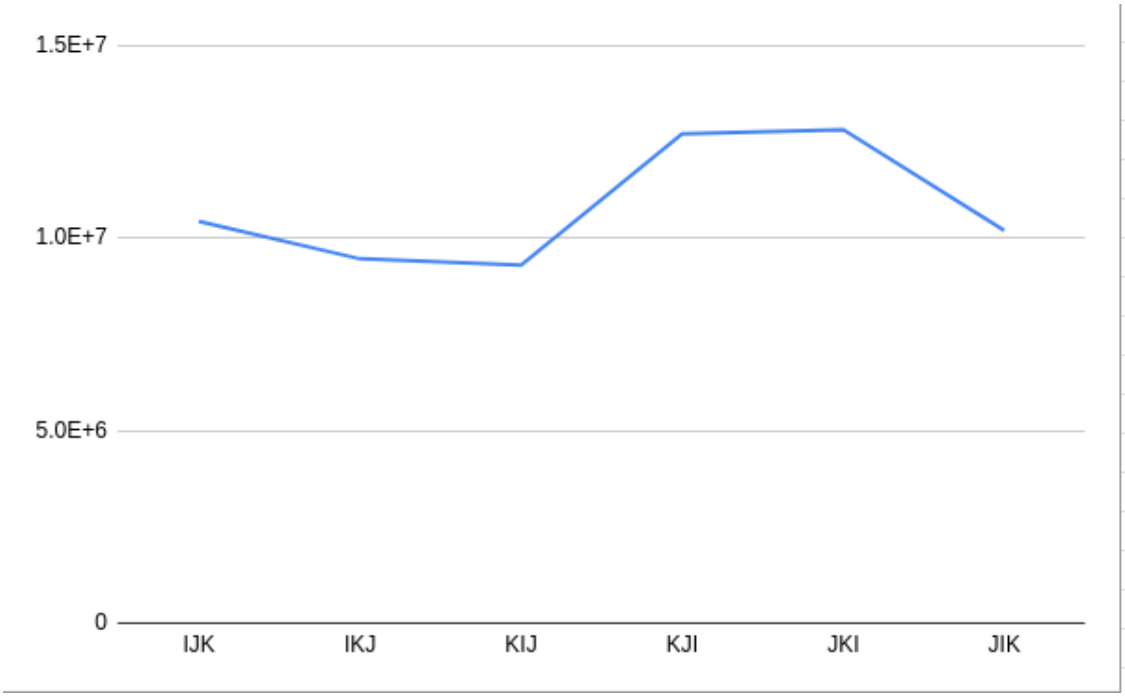
Iteration 3	10200303	90462679	743842431
Iteration 4	9880655	92903623	782598922
Iteration 5	12014748	87495620	724580132
Median	**10200303**	**89970305**	**743842431**

SUMMARY:
The summary of the above experiment is shown in the following table:

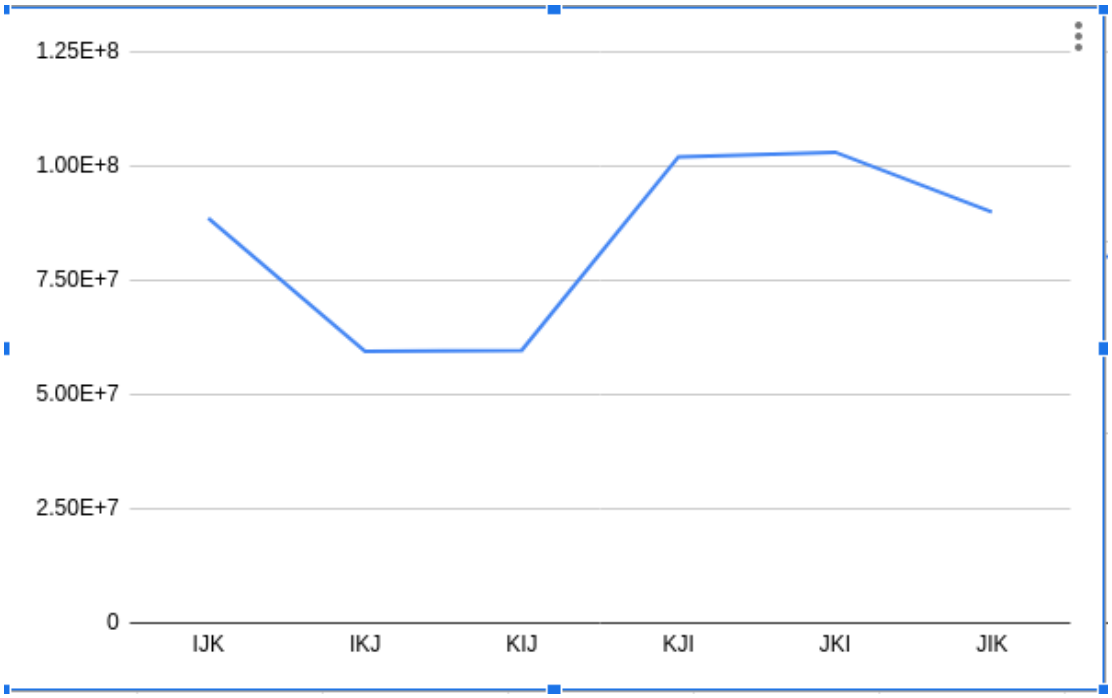
	128	**256**	**512**
IJK	10444267	88626850	724739624
IKJ	9467350	59430480	420068011
KIJ	9305010	59704946	423859587
KJI	12716718	102065793	814084227
JKI	12827583	103077558	817735952
JIK	10200303	89970305	743842431

Graphs(*showing variation of N with permutation of i, j, k***)**

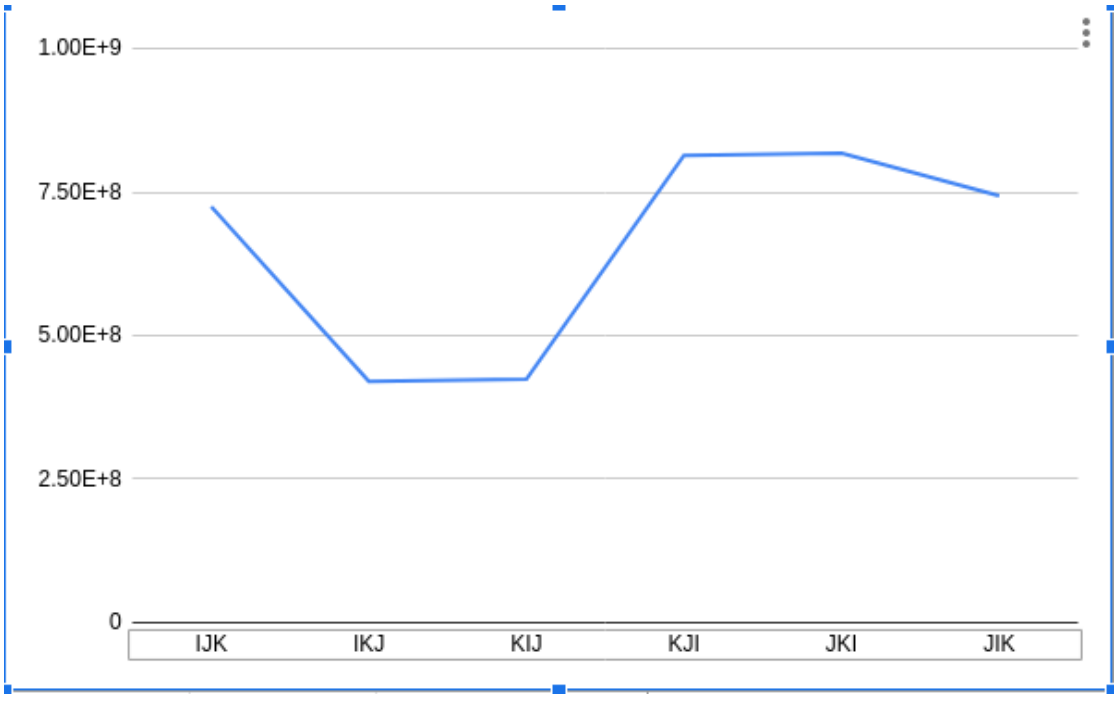
N = 128



N = 256


1.25E+8
1.00E+8
7.50E+7
5.00E+7
2.50E+7
0
IJK
IKJ
KIJ
KJI
JKI
JIK

N = 512


1.00E+9
7.50E+8
5.00E+8
2.50E+8
0
IJK
IKJ
KIJ
KJI
JKI
JIK

SUMMARY

All the observations noted here have above graphs and experimental experience as basis

1. The overall **behavior** of graph for all values of N remain **similar**
2. **Minimum** time(best sequence) = **k, i, j**
3. **Maximum** time(worst sequence) = **j, k, i**
4. The performance of **k, i, j** and **i, k , j** are very close and in most cases the difference isn't significant. Same is true with **j, k, i** and **k, j, i.**
5. Most of the performance is determined by the inner most loop:

$$Performance(as\ inner\ most\ loop):\ j\ >\ k\ >\ i$$

The above graphs are self explanatory and above were the major conclusions.

Question 4:
The code for the problem is present in the attached github repository. We could note that we have 3 nested for loops and we can interchange them maintaining the precision of the solution. Following are the combinations of the loop and their performance

Note: *Time in Seconds*

I, J, K

N	128	256	512
Iteration 1	0.28224969199982297	2.073409698999967	16.996527972000877
Iteration 2	0.26032037300046795	2.0517706720002025	16.78452467199986
Iteration 3	0.26722570800029644	1.99435360499956	17.841820249000193
Median	**0.26722570800029644**	**2.0517706720002025**	**16.996527972000877**

I, K, J

N	128	256	512
Iteration 1	0.25056748799943307	1.8869987930002026	13.756732945000294
Iteration 2	0.2495645169992713	1.8789108379996833	14.168155080999895
Iteration 3	0.2748677259996839	1.9172490860000835	15.733258565000142
Median	**0.25056748799943307**	**1.8869987930002026**	**14.168155080999895**

K, I, J

N	128	256	512
Iteration 1	0.26016236700070294	1.9112885190006637	13.158623727999839
Iteration 2	0.2540488489994459	1.9509402309995494	14.424502025999573
Iteration 3	0.2537215259999357	1.9074726670005475	14.70270794499993
Median	**0.2540488489994459**	**1.9112885190006637**	**14.424502025999573**

K, J, I

N	128	256	512
Iteration 1	0.26719552600025054	1.995343726000101	17.234896037999533
Iteration 2	0.2630918090007981	2.048039387999779	17.18846589900022
Iteration 3	0.26402612299898465	2.1766023209993364	16.882021737000287
Median	**0.26402612299898465**	**2.048039387999779**	**17.18846589900022**

J, K, I

N	128	256	512
Iteration 1	0.28957370899843227	2.091274442998838	19.85446316099842
Iteration 2	0.2797036539996043	2.0989001909983926	19.695707141998355
Iteration 3	0.2809136839987332	2.059013183999923	19.81430137800089
Median	**0.2809136839987332**	**2.091274442998838**	**19.81430137800089**

J, I, K

N	128	256	512
Iteration 1	0.307991289999336	2.116191377999712	17.332238044000405
Iteration 2	0.27211802800047735	1.9963988079998671	15.595265014999313
Iteration 3	0.27281849100108957	2.0243031270001666	15.420025614999759
Median	**0.27281849100108957**	**2.0243031270001666**	**15.595265014999313**

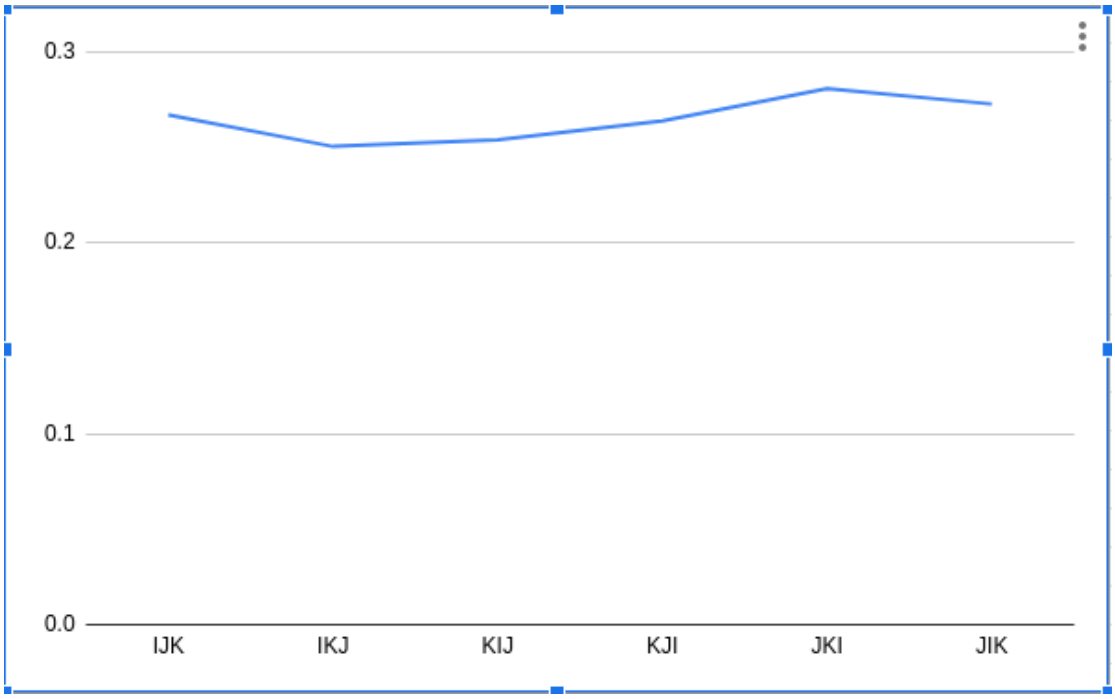
SUMMARY:

The summary of the above experiment is shown in the following table:

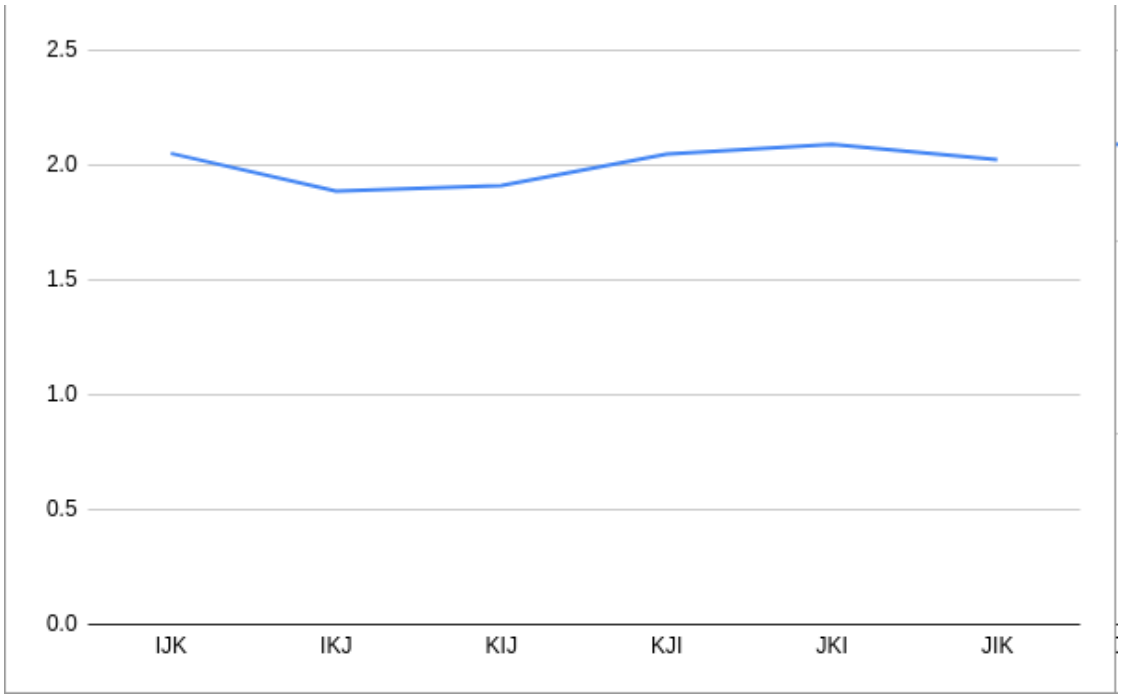
	128	256	512
IJK	0.267225708	2.051770672	16.99652797
IKJ	0.250567488	1.886998793	14.16815508
KIJ	0.254048849	1.911288519	14.42450203
KJI	0.264026123	2.048039388	17.1884659
JKI	0.280913684	2.091274443	19.81430138
JIK	0.272818491	2.024303127	15.59526501

Graphs(*showing variation of N with permutation of i, j, k***)**

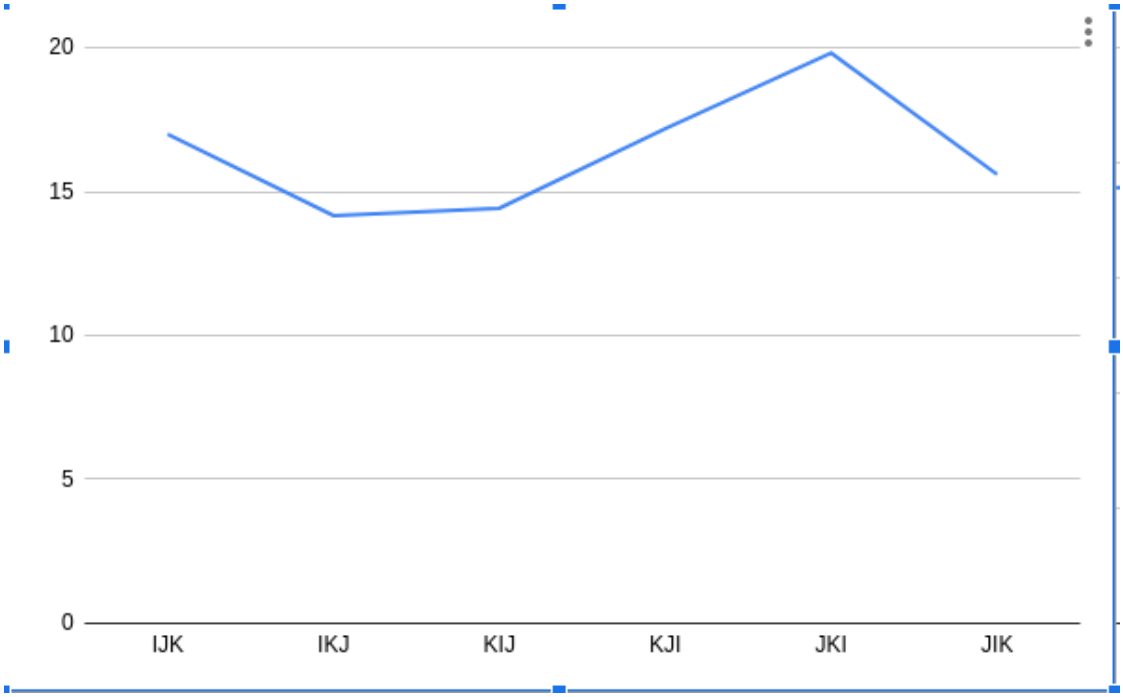
N = 128



N = 256



N = 512



SUMMARY

All the observations noted here have above graphs and experimental experience as basis

1. The overall **behavior** of the graph for all values of N remains **similar,** but we observe significant deviation for N = 512.
2. The code execution in python takes a lot longer than code execution in C++, which was expected.
3. We observe a steep slope for N = 512, which is absent in rest two.
4. **Minimum** time(best sequence) = **i, k, j**
5. **Maximum** time(worst sequence) = **j, k, i**
6. The performance of **k, i, j** and **i, k , j** are very close and in most cases the difference isn't significant. Unlike for C++, the difference of performance between **j, k, i** and **k, j, i,** isn't negligible.
7. Most of the performance is determined by the inner most loop:

$$Performance(as\ inner\ most\ loop): j > i > k$$