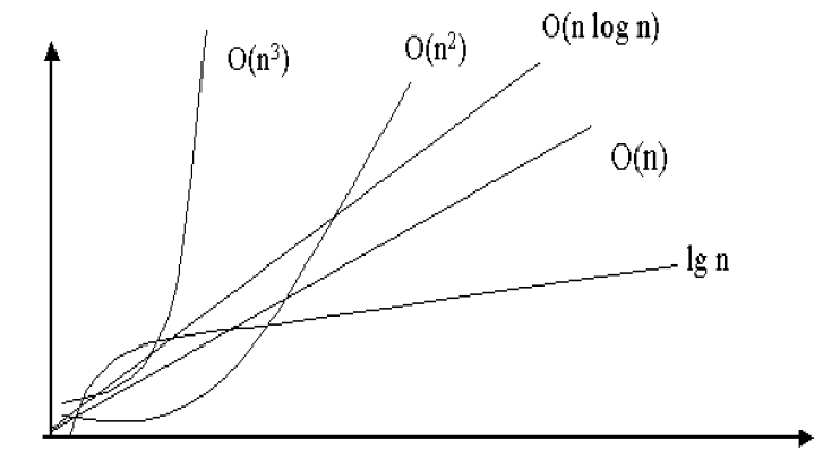
Estudi de la complexitat temporal i eficiència en algorismes destinats a càlcul de nombres triangulars



Joel Bueno Calvache

**ÍNDEX**

1. INTRODUCCIÓ
   1. ELS NOMBRES TRIANGULARS
   2. EL COST COMPUTACIONAL
2. EL CAS PRÀCTIC
   1. ALGORISMES
      1. FORÇA BRUTA
      2. ESTRATÈGIA ÒPTIMA
      3. ESTRATÈGIA DE TAULA
   2. DISSENY
      1. FORÇA BRUTA
      2. ESTRATÈGIA ÒPTIMA
      3. ESTRATÈGIA DE TAULA
   3. IMPLEMENTACIÓ
      1. FORÇA BRUTA
      2. ESTRATÈGIA ÒPTIMA
      3. ESTRATÈGIA DE TAULA
3. RESULTATS
   1. ENTORN DE PROVES
      1. HARDWARE
      2. SOFTWARE
   2. DISSENY DEL *BENCHMARK*
   3. DISCREPANÇES ENTRE ENTORNS
   4. ANÀLISI D’EFICIÈNCIA
      1. FORÇA BRUTA
      2. ESTRATÈGIA ÒPTIMA
      3. ESTRATÈGIA DE TAULA
   5. GRÀFIQUES
   6. ENGINYERIA INVERSA
      1. OPTIMITZACIONS DEL COMPILADOR
4. CONCLUSIONS
5. MOTIVACIONS

**INTRODUCCIÓ**

Quan es desenvolupa un programa per a realitzar una determinada tasca de l’usuari, sempre s’ha de fer un petit estudi sobre com es vol implementar la solució; no totes són igual de eficients i molts cops, acostumen a implementar algorismes poc eficients.

Es per això, que en aquest document detallarem i discutirem diversos algorismes per a calcular (I comprovar) si un determinat nombre es triangular o no.

**ELS NOMBRES TRIANGULARS**

Un nombre es triangular si i només si el sumatori de tots els nombres (Des de 0) previs al mateix equival al anteriorment mencionat i forma un equilàter perfecte.

Exemple:

1 es triangular? 0 + 1 = 1 -> Triangular

3 es triangular? 0 + 1 + 2 = 3 -> Triangular

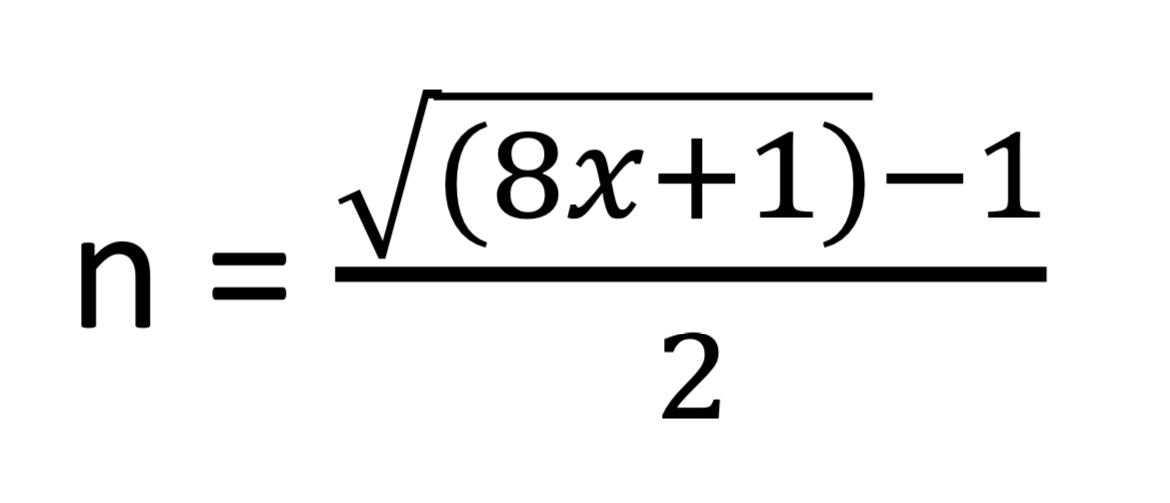
6 es triangular? 0 + 1 + 2 + 3 = -> Triangular

5 es triangular? 0 + 1 + 2 + **3** -> No Triangular!

La qüestió es resumeix en si es pot representar un nombre amb un triangle completament equilàter amb el sumatori dels nombres previs que formen el mateix.

Matemàticament parlant, existeix una expressió que en funció del resultat ens permet discernir si un nombre és o no triangular.

L'expressió és la següent:



On **x** = el nombre a comprovar.

Si el resultat es un nombre enter, el nombre es triangular, en cas contrari, no.

**EL COST COMPUTACIONAL**

Seguint amb la temàtica, un pot definir el cost computacional com una funció respecte la complexitat de temps necessària per a executar un algorisme informàtic.

La complexitat es un factor variable en funció de la implementació del mateix.

Hi han diversos tipus de costos computacionals, quadràtic, lineal, exponencial… i com bé s’ha mencionat abans, recau en la implementació que cada programador efectuï.

**EL CAS PRÀCTIC**

Se’ns presenta un cas perfecte per realitzar un estudi envers el cost computacional dels algorismes.

Càlcul i comparacions de nombres triangulars.

Els requeriments són senzills, suplir nombres enters aleatoris a 3 algorismes diferents que comproven si son o no triangulars i comprovar l'eficiència dels mateixos.

**ALGORISMES**

Oferim un total de 3 algorismes diferents per enfocar el projecte amb prou informació com per donar una sèrie de conclusions recolzades per les dades trobades gràcies a ells.

A continuació es detallaran els algorismes implementats.

**FORÇA BRUTA**

L’algorisme de força bruta genera nombres triangulars (Tot enters positius des de 0) i els va comparant amb el nombre que se li supleix al algorisme.

La estructura del algorisme segueix el següent esquema:

constants; // *O(1)* <- *Declaració de constants*

bucle { // *O(N)* <- *Fer N cops (Variable)*

sumatori; // *O(1)* <- *Increment i assignació.*

comparació } // *O(1)* <- *Comparació de igualtat.*

comparació; // *O(1)* <- *Comparació de superioritat.*

retorn; // *O(1)* <- *Retorn.*

**O(1) + O(N) + O(1) + O(1) + O(1) + O(1) = O(N)**

Els bucles acostumen a tenir una funció lineal respecte el temps, doble de execucions, doble de temps; és O(N) degut a que la resta del temps d’execució es constant O(1) i per tant es descarta a favor del temps superior O(N).

El temps invertit dins de O(N) depèn de N degut a que el nombre pot variar.

Es un mètode que funciona be sempre i quan el nombre no sigui molt gran, en el moment que el nombre és molt gran; perd la poca eficiència que te ja que ha de fer sumatoris molt més extensos cada iteració.

**ESTRATÈGIA ÒPTIMA**

L'algorisme d’estratègia òptima és de lluny, la estratègia més indicada a seguir.

Segueix la següent estructura:

constants; // *O(1)* <- *Declaració*.

comparació; // *O(1)* <- *Comparació*.

retorn; // *O(1)* <- *Retorn*.

**O(1) + O(1) + O(1) = O(1)**

El cost computacional de l’algorisme més òptim es un cost constant O(1)!

El motiu per el qual es el que menys triga en desenvolupar la tasca es perquè no requereix cap bucle (Lineal, O(N)) ni cap bucle imbricat (Quadràtic, O(N\*N)).

**ESTRATÈGIA DE TAULA**

L’algorisme d’estratègia amb taula es una forma interessant d’enfrontar el problema.

En la seva base, es molt semblant a l’algorisme de força bruta; la diferència més gran es que la taula de nombres triangulars està pre-generada, per tant, l’algorisme només s’encarrega de realitzar una cerca lineal fins que comprovi si es un triangular el nombre aportat.

Les taules pre-generades acostumen a ser un mètode utilitzat per estalviar temps d’execució oferint una sèrie de solucions calculades abans per abaratir el cost d’execució.

L’algorisme està dividit en 2 parts; generació de taula i la consegüent comprovació.

Procedirem a analitzar el procediment de la comprovació, al cap i a la fi, una taula pot ser llegida des d’un arxiu per exemple; estalviant la pre-computació de la mateixa.

Segueix la següent estructura:

constants; // *O(1)* <- *Declaració*.

bucle { // *O(N)* <- *Bucle*.

comparació } // *O(1)* <- *Comparació*.

comparació; // *O(1)* <- *Comparació*.

retorn; // *O(1)* <- *Retorn*.

**O(1) + O(N) + O(1) + O(1) + O(1) = O(N)**

La complexitat de temps algorísmica es de O(N) ja que depèn del nombre de *inputs* que rebi, per tant es lineal.

**DISSENY**

**FORÇA BRUTA**

La idea es senzilla, un bucle on s’aplica un petit teorema de sumatori per comprovar si el nombre es triangular o no (Si coincideixen).

Idea:

3 variables (Índex de bucle, Sumatori actual i resultat de triangular)

1 paràmetre, conté el nombre a processar (Enter positiu sens signe)

1 únic bucle

2 condicionals (Comprovació d’igualtat, Triangular superior més pròxim al nombre proveït)

**ESTRATÈGIA ÒPTIMA**

2 variables (Resultat equació i resultat de triangular)

1 paràmetre, conté el nombre a processar (Enter positiu sens signe)

Assignació i comprovació del resultat de la equació.

Si el resultat no es enter, trobar una alternativa.

**ESTRATÈGIA DE TAULA**

3 variables (Índex de bucle, Sumatori actual i resultat de triangular)

2 únics bucles (Un per la funció de popular taula i per la restant)

2 condicionals (Comprovació d’igualtat, Triangular superior més pròxim al nombre proveït)

**IMPLEMENTACIÓ**

**FORÇA BRUTA**

La implementació del algorisme de força bruta es prou senzilla, es crida la funció amb un nombre enter positiu i retorna si es triangular o no.

Utilitza un bucle el qual s’executa mentre el sumatori actual (Que ens serveix per discernir si el nombre proveït es triangular o no) es igual o inferior al nombre a processar.

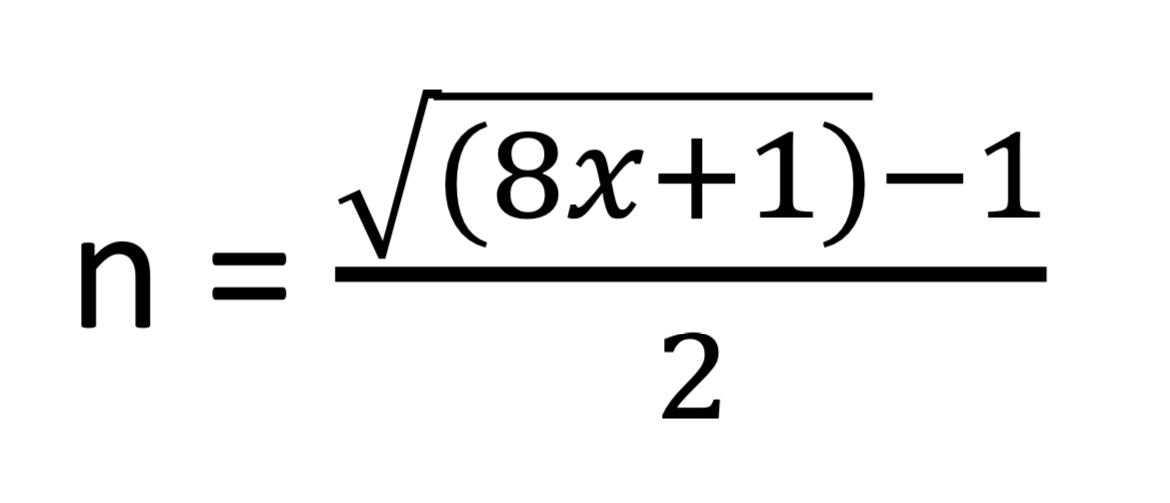
Dins del bucle es compara el sumatori amb el nombre per esbrinar si es triangular o no (Seguint les normes matemàtiques dels nombres triangulars); en cas positiu, es retorna el sumatori, en cas negatiu, es segueix al bucle.

Si es surt del bucle, s’assignarà el sumatori (Nombre triangular) més proper al nombre proveït.

**ESTRATÈGIA ÒPTIMA**

De lluny, l’estratègia més adient.

Computa si un nombre es triangular en funció de si una equació dona resultat enter o decimal.



Per obtenir el següent (Més alt) triangular a partir de x, es tant senzill com obtenir-lo alterant la fórmula fins obtenir:

Aquest algorisme ens proporciona el cost més baix dels tres [O(1)], constant.

Obté nombres enters positius com a paràmetres de la funció.

**ESTRATÈGIA DE TAULA**

Gairebé igual al de força bruta; la diferència es que es pre-calcula una taula de P nombres triangulars per agilitzar el procés (Evita la el còmput constant de sumatoris) i procedir directament al bucle de comparacions.

També comprova que el nombre no excedeix el triangular de la posició P, en cas d’excedir-lo, no processa el nombre.

Tot dues funcions (Popular taula i processar taula) accepten un *array* de P-elements amb nombres enters positius sense signe en forma de referència (Per evitar ‘clonar’ la variable en temps d’execució, així s’altera la taula original gràcies a treballar amb punters).

**RESULTATS**

Analitzarem els resultats obtinguts seguint un model senzill de comparacions visuals de temps d’execució per determinar la eficiència i el rendiment de cada algorisme.

**ENTORN DE PROVES**

***HARDWARE***

Per dur a terme aquesta pràctica, he utilitzat 2 ordinadors personals diferents amb finalitats diferents (Portàtil i ordinador de sobretaula) per poder aconseguir una idea de com (O no) de bé s’orienta cada algorisme a cada *hardware* i *software*.

El meu entorn consta del següent:

**Ordinador de Sobretaula:**

**MOBO**: GIGABYTE B550M DS3H VER.1

**CPU**: AMD Ryzen 5 5600G, 6C/12T; 3.9-4.4GHz

**RAM**: Corsair Venegance 16 GB DDR4 Dual-Channel 3200MHz (1600MT/s)

**GPU**: NVIDIA GEFORCE GTX 1650 SUPER

**M.2 SSD**: HYNIX 240GB M.2

**Ordinador Portàtil:**

**MOBO**: Propietary Intel Chipset (HP)

**CPU**: Intel i5 1035G1 (4C / 8T; 1.0-3.6GHz)

**RAM**: KINGSTON HYPERX SO-DIMM DDR4 2667MHz (1333MT/s)

**iGPU**: Intel Iris Plus Graphics G1

**dGPU**: NVIDIA GEFORCE GTX 1050 Max-Q

**M.2 SSD**: Propietary HP 240GB M.2

Son dos ordinadors amb finalitats completament diferents i ens ajudarà a treure unes conclusions més encertades degut a la naturalesa de cadascun dels dos.

***SOFTWARE***

L’ordinador de sobretaula (Des d’ara, anomenat *AMD*) executarà el *benchmark* en 3 entorns diferents:

* Windows 11 Pro (Insider Preview, Fast Channel)
  + Build 22572.ni\_release.220304-1536
* WSL 2 (Windows Subsystem for Windows)
  + Amb la distro Ubuntu 22.04 (Development branch)
  + Linux Ryzen5 5.10.60.1-microsoft-standard-WSL2 #1 SMP Wed Aug 25 23:20:18 UTC 2021 x86\_64 x86\_64 x86\_64 GNU/Linux
* Ubuntu 22.04 (Development branch) Natiu
  + Linux ubuntu 5.15.0-22-generic #22-Ubuntu SMP Tue Feb 8 10:16:30 UTC 2022 x86\_64 x86\_64 x86\_64 GNU/Linux

L’ordinador portàtil (Des d’ara, anomenat *HP*) executarà el benchmark en 3 entorns diferents:

* Windows 11 Pro (Insider Preview, Fast Channel)
  + Build 22572.ni\_release.220304-1536
* Ubuntu 22.04 (Development branch) Natiu
  + Linux CS3003NS 5.16.9-051609-generic #202202110934 SMP PREEMPT Fri Feb 11 09:59:18 UTC 2022 x86\_64 x86\_64 x86\_64 GNU/Linux
* macOS Monterey 12.2.1 (Hackintosh)
  + Darwin MacBook-Air-de-Joel.local 21.3.0 Darwin Kernel Version 21.3.0: Wed Jan 5 21:37:58 PST 2022; root:xnu-8019.80.24~20/RELEASE\_X86\_64 x86\_64

La finalitat principal d’executar en 2 entorns de *hardware* diferents i 3 de *software* també diferents, es comprovar les diferencies arquitecturals i de disseny de cadascun dels mencionats prèviament.

He trobat adient realitzar això ja que m’assegura un codi 100% portable, funcional, i em dona unes dades molt valuoses a l’hora de treure conclusions.

Si vol comprovar la resta del entorn de *software*, pot accedir a uns petits documents de text on es documenta la versió de cada programa que intervé en la compilació de la pràctica.

He decidit afegir la versió de cada utilitat degut a que poden haver diferències subtils entre versions i canvis de programari d’un S.O. a un altre (Per exemple, *time(1)* a *Linux* i a *macOS* funcionen diferent, perquè per exemple, *macOS* utilitza *zsh* com a *shell* i *Ubuntu*, *bash*).

**DISSENY DEL *BENCHMARK***

M’ha semblat una idea genial dividir les proves en 2; el primer *set* de proves, utilitza el programa tal i qual està entregat (Calcula el *delta* entre l’inici i el final dels bucles mitjançant estructures *time\_t*); i l’altre *subset* de proves ofereix una vista més personalitzada (Divideix els 3 algorismes en 3 programes diferents i utilitza la utilitat *time(1)* per a trobar el temps d’execució de manera més detallada).

Cada prova s’executa 10 milions de cops a cada lloc.

S’ha d’afegir que cada *set* de proves, es feia per 2 degut a que el primer *subset* l’executava amb el programa *as-is* i el segon *subset* l’executava amb un nivell de optimització per compilació elevat (*-O3* en comparació amb *-O0*).

Es a dir, en total tinc ((2 \* 2) \* 3) \* 2) = 24 *sets* de proves dividits per nivell de optimització, sistema operatiu i *hardware*.

Esquema dels jocs de proves:

1 Ordinador

3 Sistemes Operatius per Ordinador

2 Mètodes de *benchmark* (Temps intern i extern)

2 Modes de compilació diferents a cada *benchmark* (Diferents optimitzacions)

**DISCREPANÇES ENTRE ENTORNS**

Durant la totalitat del desenvolupament de la pràctica, m’he trobat amb un fet molt intrigant...

Windows (Tant a HP com a AMD) triga 5000 cops menys en executar els algorismes, sent l’únic lloc que hauré de descartar les dades a l’hora d’extreure les conclusions.

Tot i així mai deixo les coses a mitges i he dedicat un apartat a la meva recerca i hipòtesi del perquè podria estar passant això.

**ANÀLISI D’EFICIÈNCIA**

Dins del arxiu de la pràctica es poden trobar captures de pantalla amb cada execució.

Per entendre que significa cada arxiu, vaig decidir donar’ls-hi un petit i senzill format:

MÀQUINA-TEMPS\_INTERN\_O\_EXTERN-NIVELL\_OPTIMITZACIO-SO

Exemples:

AMD-INT\_TIME-O0-WIN

A l’ordinador de sobretaula, mostra el temps mesurat per la pràctica mateixa, compilada sense optimitzacions i amb Windows com a *target*.

HP-EXT\_TIME-O3-MAC

Al portàtil, mostra el temps mesurat amb la utilitat externa *time(1)*, compilat amb nivell d’optimització *O3* (Alt) i amb *target* cap a *macOS*.

Dins de les dos carpetes es pot trobar tot el que s’ha anat treballant.

Començarem comparant el rendiment i eficiència global dels 3 algorismes.

Independentment de la arquitectura, S.O. i potència bruta de tots llocs on s’han executat les proves; es veu una clara diferència entre els 3 algorismes.

Ordenats de més temps d’execució a menys:

Força Bruta > Taula > Òptim

El clar guanyador es el algorisme “òptim” degut al seu cost O(1) comparat amb el cost O(N) dels altres.

Per fer les comparacions de temps d’execució, em basaré en els temps oferts per la utilitat *time(1)* i el seu desglossament de temps de execució.

Al cap i a la fi, el sistema operatiu té altres processos concurrents i ha d’anar saltant de tasca en tasca per oferir les capacitats de multiprocés, per tant, si utilitzo el *delta* de temps que m’ofereix el *time­\_t* estic introduïnt segons extra a cada joc de proves.

Basarem l’estudi envers el temps proporcionat per la subcategoria *user* oferida per *time(1)*.

*time(1)* ofereix 3 subclasses de mesura de temps:

* ***real***
  + Mateix temps que podria obtenir amb el *delta* (Total de temps “real”)
* ***user***(El que usarem)
  + Temps que dedica el ordinador al programa, elimina les variables de temps afegides per el administrador de tasques del *kernel* i el temps invertit en funcions de la llibreria C per exemple, d’aquesta manera obtenim el temps d’execució del propi algorisme.
* ***sys***
  + Temps invertit en “trucades” i processos derivats del algorisme que acostumen a derivar en *syscalls* que executen codi en un mode més privilegiat que l’usuari i per tant, no compten com a execució del algorisme.

**FORÇA BRUTA**

La mitjana de temps d’execució del algorisme (Sense optimització) de força bruta al ordinador AMD es d’uns 5 minuts i mig.

Amb optimitzacions, 1 minut i mig.

La mitjana de temps d’execució del algorisme (Sense optimització) de força bruta al ordinador HP es d’uns 6 minuts.

Amb optimitzacions, 3 minuts i mig.

En un apartat següent veurem la desensamblació dels dos binaris (Optimitzat i no-optimitzat) i compararem per trobar on està la diferència en els temps d’execució.

**ESTRATÈGIA ÒPTIMA**

La mitjana de temps d’execució del algorisme (Sense optimització) de força bruta al ordinador AMD es d’uns 20 segons.

Amb optimitzacions, 12 segons.

La mitjana de temps d’execució del algorisme (Sense optimització) de força bruta al ordinador HP es d’uns 25 segons.

Amb optimitzacions, 20 segons.

**ESTRATÈGIA DE TAULA**

La mitjana de temps d’execució del algorisme (Sense optimització) de força bruta al ordinador AMD es d’uns 4 minuts i mig.

Amb optimitzacions, 4 minuts i mig.

La mitjana de temps d’execució del algorisme (Sense optimització) de força bruta al ordinador HP es d’uns 5 minuts.

Amb optimitzacions, 4 minuts i mig.

**GRÀFIQUES**

O(1)

O(N)

O(N)

**ENGINYERÍA INVERSA**

Abans he mencionat que els binaris de Windows s’executaven molt ràpid, he desencadellat els binaris per trobar alguna mena de pista de perquè pot estar passant això. No he pogut treure cap conclusió clara (I em nego a acceptar que Windows pot ser 5000x cops més ràpid) però si hipòtesis.

He comparat la mesura de cada binari per a cada sistema entre sí per tal d’observar diferències grans en el mateix, fora de les optimitzacions (Que acostumen a estalviar un *byte* per aquí i per allà), els binaris de Windows són 600x cops més pesats.

He pensat que potser el compilador estava veient que s’estava calculant una taula i ell mateix la estava pre-compilant al binari per fer la feina més ràpid (Tot i que ho fes, no justifica la abismal diferència), però ho he descartat al moment degut a que sota una petita inspecció a la secció .data, .rodata i .bss no he trobat evidències d’això (Es una hipòtesi bastant treta de lloc, però he vist pitjors coses fetes pel compilador *mingw*...).

La segona hipòtesi que he plantejat es que simplement *mingw* no està compilant codi correctament (Un company de grup s’ha trobat en les mateixes, que va molt ràpid); havia pensat en utilitzar el *MSVC*, el compilador C/C++ que incorpora Visual Studio; però no he sigut capaç de descarregar només l’entorn de compilació, per tant no puc dir que *mingw* es el culpable si no tinc cap manera de comparar resultats amb altres compiladors.

He depurat el programa sota Windows amb l’ajuda del IDA Pro i entra correctament als bucles i realitza la feina correctament.

Es ben estrany.

Tercera i última hipòtesi; veient que el codi compila a qualsevol plataforma de manera correcta i funcional; possiblement *mingw* estigui fent de les seves... trencant el programa suficientment com per a que sigui depurable i que a simple vista no falli res però que alhora estigui menjant-se el temps d’execució... si fos tant obvi no entraria ni tant sols als bucles...

Queda per veure que es el que passa realment.

**OPTIMITZACIONS DEL COMPILADOR**

Dins de la carpeta Disassembly/ hi han exemples comparatius de cada funció compilada amb optimitzacions i sense; els exemples més prominents que he observat son l’ús de registres extres (Agilitzar la alteració de dades), mètodes més òptims per a l’accés a una taula i la substitució de certs for() per: do {} while();

La resta de canvis són característics de cada compilador, S.O. i arquitectura.

**CONCLUSIONS**

Queda més que demostrat que no tot algorisme serveix a l’hora de desenvolupar una solució per a un problema.

Es essencial i un requisit bàsic saber avaluar quin tipus d’algorisme és el més idoni per resoldre un problema.

Es un error utilitzar una solució O(N\*N) quadràtica quan es podria fer-ne una de lineal O(N), per exemple.

Es una molt bona pràctica realitzar proves en altres ordinadors, sistemes operatius i compiladors; gràcies a això es poden descartar segons quins problemes i seguir de manera correcta amb la pràctica a més de donar-nos una idea de la escalabilitat del nostre codi en sistemes més i menys potents.

Per extreure conclusions s’ha de tenir un **N** (*Sample*) prou gran com per poder extrapolar i treure conclusions de manera efectiva (Vaig decidir usar N=10 milions a tots llocs).

Sempre es podrà optimitzar més un algorisme, la meva implementació del mètode “òptim” ha tingut gairebé 5 revisions, va començar calculant només si era o no triangular, es va afegir una manera d’obtenir el següent més proper amb un for() (Abaixant la eficiència en cas que no fos triangular a la primera), després es va substituir amb una fórmula dins d’un while() i l’última revisió re-implementa la fórmula original per obtenir el primer triangular superior al nombre donat amb una funció O(1).

S’ha de entendre que el cost execucional es un problema que molts cops es passa per alt, temps enrere els programadors dissenyaven solucions creatives a problemes complexes amb 1/10000 de potència computacional.

<https://en.wikipedia.org/wiki/Fast_inverse_square_root#Overview_of_the_code>

Tenir més potència no ens ha de tornar més mandrosos ;)

S’haurien de tenir a mà mínim un parell o tres d’eines de *profiling* per poder analitzar els “punts calents” del nostre codi per potser dissenyar una solució molt més potent i eficient.

Els *switches* dels compiladors que ens ofereixen nivells més alts d’optimització estàn be en funció del projecte, però no es bona idea utilitzar-los per a codi de producció ja que sovint utilitzen mètodes massa agressius que poden destapar *bugs* al programari que mai havíem vist si no compilàvem sense optimitzacions i per tant, poden obstruir amb la experiència final de l’usuari; fora d’això, donen una “segona opinió” sobre com resoldre un determinat problema.

Es un error mantenir funcions externes (printf()’s majorment) dins d’un codi preparat per a *benchmarks*, només afegeix temps d’execució i afegeix *soroll* a les dades obtingudes.

Que tinguis la funció *pow()* disponible, no implica que l’hagis d’usar per fer una potència de 2, fent n\*n t’estalvies una crida externa...

Tot suma! Utilitzar funcions de menys precisió (sqrtf a favor de sqrt) o tipus més petits ajuda a reduir el cost d’execució (Tot i que varia en funció del tipus de projecte en el que estiguis treballant...).

Amb temps, ganes i calma; es poden fer meravelles.

**MOTIVACIONS**

El cost execucional sempre havia sigut per a mi un gran desconegut (Matemàticament parlant), analitzar la eficiència i rendiment dels algorismes que implementem es un gran pas dins del desenvolupament de *software* que ens permet formar-nos com a millors programadors. Pot semblar “palla” dins de la assignatura; qui coneix el potencial que aquests anàlisis els hi ofereix, arribarà molt lluny. Un microcontrolador repudiarà un algorisme O(!N)...