

Beadandó feladat — Legkisebb dátumok

Feladat beadása

Határidő ápr 30 23:59 -ig **Pont** 0 **Beküldés...** egy fájlfeltöltés **Fájltípusok** zip, tar.gz és tgz
Elérhető márc 14, 00:00 - ápr 30, 23:59 körülbelül 2 hónap

Motiváció

Számos fizikai jelenséget jól lehet vizsgálni számítógépes szimulációk segítségével. Ilyen pl. a [Brown-mozgás](http://en.wikipedia.org/wiki/Brownian_motion) [\[http://en.wikipedia.org/wiki/Brownian_motion\]](http://en.wikipedia.org/wiki/Brownian_motion), a [hőmérséklet kiegyenlítődése](http://en.wikipedia.org/wiki/Heat_equation) [\[http://en.wikipedia.org/wiki/Heat_equation\]](http://en.wikipedia.org/wiki/Heat_equation), vagy az [ozmózis](http://en.wikipedia.org/wiki/Osmosis) [\[http://en.wikipedia.org/wiki/Osmosis\]](http://en.wikipedia.org/wiki/Osmosis). Az első esetben legtöbbször kétdimenziós modellek segítségével különböző méretű golyókkal szimuláljuk a részecskék haladását és ütközéseit (ld. [videó](http://www.youtube.com/watch?v=PDnPr3FiYnQ) [\[http://www.youtube.com/watch?v=PDnPr3FiYnQ\]](http://www.youtube.com/watch?v=PDnPr3FiYnQ)!). Minden molekula minden másikkal ütközhet, amely ütközés bekövetkezésének ideje és eredménye kiszámítható az ütközésben résztvevő részecskék sebességvektoraiból (mérték és irány). (Feltételezzük, hogy nem történik olyan tökéletes ütközés, amelyben egyszerre 2-nél több részecske vesz részt.) Azonban van egy probléma: ha ki is számoltuk a szimulációnk összes résztvevőjére az ütközéseiket, egy ütközés megváltoztatja a két résztvevő sebességét és irányát, amelynek eredménye az, hogy ezt követően a már kiszámolt ütközésekben nem, vagy nem úgy vesznek részt. Azaz a részecskék teljes populációjából időrendben haladva mindig az első ütközést kell figyelembe vennünk, ennek eredményével kiszámolni további ütközéseket, és invalidálni azokat, amelyek már nem következhetnek be, mert legalább az egyik résztvevő időben korábban trajektóriát változtatott.

Bevezetés és tudnivalók

Az egyszerűség kedvéért a feladat során nem kell a teljes szimulációt megvalósítani (bár kétségkívül izgalmas lenne...), hanem a **legkorábbi dátumok** kiválasztására szorítkozunk. Szükségünk lesz egy N elemű adatszerkezetre, amelyben megkeressük a legrégebbi dátumot, majd ezt "feldolgozás" után lecseréljük egy véletlenszerűen generált új dátumra (ez lehet akár korábbi is, mint a most éppen feldolgozott!). Ezt az iterációt $N+K$ -szor végezzük el, ahol K **nagyságrendben** nagyobb, mint N .

Pl. $N = 1'000'000$ és $K = 500'000'000$.

A végrehajtás egyes jól körülhatárolt részeinek végrehajtási idejét **meg kell mérni**, és a beadandónak szerves része az, hogy ötleteljetelek, optimalizáljatek, gondolkozzatek, és ügyeskedjetelek, hogy minél gyorsabb legyen a végrehajtás!

Arra kérünk mindenkit, hogy álljon neki a feladatnak amikor csak tud. Nem szükséges vele "egy szuszra" sokat foglalkozni, inkább rendszeresen érdemes foglalkozni vele egy kicsit. Egy beadandót elkészíteni nem egy éjszakás kaland és nem egy triviális feladat. Ez a feladat nem olyan, hogy a határidő előtti délutánon össze lehessen dobni!

✓ **Bármilyen nyelvi és standard-könyvtári elem**, amely egy hosszú stabilitású platformon — Debian 11, Ubuntu 20.04, 22.04, stb. — vagy egy kellően elterjedt fordítóprogram verzióval — pl. GNU GCC 9, 11, illetve LLVM Clang 14, 16 — **elérhető, szabadon felhasználható**. Technikai okokból szeretnénk, ha nem kapnánk olyan beadandókat, amik Microsoft MSVC vagy a Windows-os STL implementáció *"sajátosságaira"* hagyatkoznak! (A kiértékelést x86_64 Linux környezetben fogjuk végezni!) Elsődlegesen **C++17** standard szerinti megoldásokat várunk, a C++20 standard LTS platformokon elérhető elemei is használhatók, de nem kötelező ezt igénybe venni.

⚠ A feladat oktatási, tanulási jellege miatt arra kérünk, hogy a megoldást te magad gondold végig, és ne függj harmadik féltől származó – bármennyire híres vagy nemzetközileg elismert – library-tól.

⚠ **TILOS** a fordítóprogramok által kedvességből alapértelmezésben, vagy platform-specifikusan csak feltételeesen engedélyezett, nem C++ szabványos nyelvi konstrukciókra — úgy, mint dinamikus méretű tömbök (*variable length array (VLA)*), méret-nélküli farok-tömbök (*flexible array member (FAM)*), a C11 standard "safe" kiterjesztése (*Annex K*) — vagy egyéb non-portable dolgokra — pl. *inline assembly* — hagyatkozás!

⚠ A beadandó készítése során előfordulhat, hogy tervezési zsákutcába viszitek magatokat, ahonnan kijönni *biztonsági mentés* nélkül nagyon nehéz. Arra kérünk mindenkit, hogy gyakran **mentse el** a beadandója állapotát. Beadásként majd egy tömörített archívumot (ZIP vagy TAR.GZ) kell beküldeni!

💬 Úgy általában kommentezd a kódod bátran, megnézéskor nem csak a megoldást szeretnénk látni, hanem a benned felmerülő kérdéseket, és az arra kigondolt válaszokat! Ez a feladat most nem a tankönyvben szereplő hieroglifa-szekvenciának való tökéletes megfelelésről, hanem a kalandról, a kérdésekről, és a gondolkodás módjáról szól.

😊 Előfordulhat, hogy egyes, a feladatokban várt ismeret még nem hangzott el előadáson, vagy nem rendelkeztek vele. Ez esetben javasolt utánaolvasni néhány dolognak: az előadás slide-ok korábbi féléves változatai a honlapon elérhetők — ezek a félév előrehaladtával aktívan frissülnek! — vagy [CppReference](http://en.cppreference.com/w) [\[http://en.cppreference.com/w\]](http://en.cppreference.com/w)-en.

Changelog

- 2024. március 14.: 📦 Nulladik verzió összeáll és kikerül.

Részletes feladatkiírás

Az implementáció során nyugodtan használható több fordítási egység és több header fájl. Ne duplikálj kódot, tedd a megvalósításod újrahazsnosíthatóvá. Ügyelj rá, hogy amennyiben egy függvény-definíciót egy *inline*-olási szempontból előnyösebb helyen írsz meg, akkor az jól viselkedjen ha ugyanabba a binárisba többször kerül szerkesztésre.



A kód megírása során törekedj az igényes, esztétikus munkára. A programelemeket — ahol a feladatkiírás nem köti meg expliciten — kifejezően nevezd el. Törekedj arra, hogy a lokális változók és paraméterek, ha nem változnak a függvény végrehajtása során, akkor megfelelően **const**-ok legyenek, hogy még véletlenül se írhasd felül az értéküket.

Timed task

Első lépésben az idő-mérés megkönnyítése érdekében készíts egy függvénytemplate-et, amely paraméterül kapja a végrehajtott feladat szöveges azonosítóját ("nevét") és a végrehajtott feladatot implementáló lambda függvényt. Ez a függvény írja ki, hogy az adott nevű taszk végrehajtása elkezdődött, befejeződött, időbélyeggel. A kettő között hajtódjon végre a lambdaként átadott taszk, és **kizárólag a kapott taszk** végrehajtásának idejét mérje meg. A befejeződéssel egy időben írja ki a végrehajtásra eltöltött időt valamilyen értelmezhető felbontásban, pl. ezredmásodperc (*millisec.*) vagy századmásodperc (*microsec.*). Ezek a kiírások a *standard error* kimenetre kerüljenek!

Date

Készítsétek el az előadáson bemutatotthoz nagyon hasonló Date osztályt. Ez az osztály feleljen meg az alábbi **minimális** interfész és viselkedési követelményeknek. További segédfüggvények tetszőleges láthatósággal tetszőlegesen létrehozhatók, és az objektumok belső reprezentációja is rátok van bízva. Feltehető — külön ellenőrizni **nem** kell, de akár egy optimalizációnak fontos lehet, — hogy a dátumok az 1800. és 2200. évek közé (zárt intervallum) fognak esni.

- Készítsünk egy **három paraméteres konstruktort**, amely *"Év"*, *"Hónap"*, *"Nap"* *int*-ekből konstruálja az objektumot. Ez a konstruktor ne ellenőrizze, hogy a dátum helyes-e.
- Legyen három *getter* a dátum konstruálására használt három komponens külön-külön lekérdezésére. Ezek a függvények a belső reprezentációtól függetlenül *int*-et adjanak vissza! Neveik, értelemszerűen: *getYear*, *getMonth*, *getDay*.
- Legyen lehetőség egy dátum *helyességét* lekérdezni akkor is, ha létrejött az objektum. Készíts ehhez megfelelő nevű *member függvényt*, és azonosan viselkedő *bool*-ra konvertáló *operátort* is. (Legyen egy dátum *bool*-ként akkor, és csak akkor *igaz*, ha helyes.)
- Készítsünk egy **három paraméteres factory függvényt**, amely *"Év"*, *"Hónap"*, *"Nap"* *int*-ekből konstruál objektumot. Ez ellenőrizze, hogy az ezen értékekből megkonstruált dátum (a Gergely–naptár szerint) helyes-e (szökőnap csak szökőévben, nincs 14. hónap, nincs május 35-e, stb.).
 - Amennyiben nem helyes, az objektummal ne térjen vissza. Helyette dobjon *bad_date* exceptiót, amelyet szintén te implementálj le.
 - bad_date* legyen [std::runtime_error](http://en.cppreference.com/w/cpp/error/runtime_error)  (http://en.cppreference.com/w/cpp/error/runtime_error).leszármazott osztálya, és a kivételt magyarázó indokolás szövege tartalmazza tetszőleges, de érthető módon a hiba okát.
- Készítsünk egy **factory függvényt**, amely paraméterül kap egy *"Minimum"* és egy *"Maximum"* dátum objektumot, és legyárt **egy** véletlen generált *helyes* dátumot a paraméter *min.* és *max.* között (egyenlőség is megengedett), amellyel visszatér.
 - Ha a véletlenszám-generátor rossz dátumot állított volna elő, akkor a keletkező *bad_date* kivételt kapjuk el, és anélkül, hogy a *factory* függvényből visszatérnénk, kíséreljük meg egy új dátum előállítását.
 - Amennyiben P próbálkozás után (P egyénileg választott határérték, de a programban egy fordítási idejű konstansként, megfelelő névvel és jól láthatóan szerepeljen) se sikerül ennek a függvénynek jó dátumot előállítani, dobjunk *terrible_random* kivételt. *terrible_random* is legyen [std::runtime_error](http://en.cppreference.com/w/cpp/error/runtime_error)  (http://en.cppreference.com/w/cpp/error/runtime_error).leszármazott osztálya.

Random-generator

Az absztrakt feladatnak része a feldolgozott elem *véletlen* elemre cserélése. Azonban **nem** szeretnénk a főprogram *közben* egyesével a randomszám-generátor futásteljesítményét mérni, ezért **egy főprogramban** készítsétek el azt, hogy kigeneráltok N+K darab véletlen dátumot egy szöveges fájlba.

A program kapja meg N és K értékét parancssori argumentumként. A kimeneti fájl az opcionális harmadik paraméterben kerüljön átadásra, ha ez nem lett megadva, vagy az értéke kötőjel ('-'), akkor a standard *output*-ra íródjon ki az eredmény. (A standard Unix rendszerprogramok nagy részben hasonló logikát követnek, amikor tudnak fájlba és standard *streamre* dolgozni.)

Kezeljük le megfelelő hibaüzenettel és parancssori visszatérő értékkel (programstátusszal), ha helytelen számosságú (2-nél kevesebb, vagy 3-nál több) parancssori argumentum került a felhasználó által megadásra.

Ebben a programban használjuk fel a korábban elkészített *factory* függvényt. A generáláskor használt minimum dátum legyen 1970. január 1-je, a maximum pedig 2038. január 19-e. Ha nem sikerül kellő számú helyes random dátumot előállítani, a *terrible_random* kivételt hagyjuk kiszökni és a programot hagyjuk elszállni.

A szöveges fájl formátuma az alábbi legyen. A sorokat '\n' newline-karakter zárja, az egyes dátumok komponensei év-hónap-nap sorrendben, szőkőzőkkel elválasztva, soronként egy dátumot írjunk.

```
N
K
Y_1 M_1 D_1
Y_2 M_2 D_2
```

...
Y_(N+K) M_(N+K) D_(N+K)

Consumer

A feladatot implementáló fejlesztés felé haladván először készítsük el a dátumok "feldolgozását" szimuláló függvényt. **A függvény implementációja önálló fordítási egységbe kerüljön!** A függvény neve tetszőlegesen megválasztható. Ez a függvény pontosan **egy** dátumot kapjon!

Adjon vissza `int`-t **-1, 0, vagy 1 értéket** [↗] (http://en.wikipedia.org/wiki/Sign_function) annak függvényében, hogy a paraméterül kapott dátum régebbi (kisebb), egyenlő, vagy újabb (nagyobb), mint az a dátum, amellyel a függvényt előzőleg meghívták.

Az első meghíváskor ez a kérdés értelmetlen, így csak ekkor a függvény válasza **42** [↗]

(http://en.wikipedia.org/wiki/Phrases_from_The_Hitchhiker%27s_Guide_to_the_Galaxy#The_Answer_to_the_Ultimate_Question_of_Life,_the_Universe,_and_I legyen).

A fő főprogram

A randomszám-generáló résztől független főprogramban készítjük el a feladat tényleges megvalósítását, amely végrehajtja a legkisebb dátum megkeresését, annak "feldolgozását", egy új "véletlen" dátum kiválasztását megfelelő sokszor. A programhoz szükségünk lesz egy legfeljebb N elemet tárolni képes saját **"C"** adatszerkezetre — ennek ügyes implementációja része az optimalizációs feladatnak! — és a maradék K elemet tárolni képes adatszerkezetre, amely viszont **kötelezően** `std::vector<Date>`. A saját adatszerkezet típus, és a két adatszerkezet változóneve tetszőlegesen megválasztható. (A továbbiakban **"C"** jelöli a saját adatszerkezetet.)

Ezt a főprogramot mindig **egy** parancssori argumentummal hívassuk, amely egy fájl elérési útvonala. (Kevesebb, vagy több argumentum esetén adjunk értelmes hibaüzenetet és kilépési státuszt.) Az itt kapott fájl a korábban említett formátumú, már kigenerált véletlen dátumokat tartalmazó fájl, ha helyes!

Nyissuk meg, és olvassuk be a fájlból N -t és K -t, majd olvassunk be N darab dátumot a **"C"** adatszerkezetbe, az ezt követő K dátumot pedig a `std::vector<Date>`-be.

A beolvasási részt **mérjük meg** a korábban implementált függvénytemplate megfelelő felhasználásával. A beolvasás közben kezeljük le a *fájl I/O*-ból származó szélsőséges eseteket, pl. nem létező vagy nem olvasható fájl, üres fájl, negatív N és K , nulla N , és azt is, ha nincs a fájlban legalább annyi dátum sor, amennyit be kellene olvasni.

Nem szükséges ellenőrizni és jelezni, hogy ha több dátum van.

Itt feltehetjük, hogy a fájlból beolvasott dátumok formailag és tartalmilag is helyesek — mind a három mező megfelelően ki van töltve, és az értékek egy *helyes* dátumot adnak.

Bár a fájl perzisztensen létezik a háttértáron, tekintsünk rá **szekvenciális inputfájlként**. Azaz a korábbi beolvasást úgy valósítsd meg, hogy egy-egy értéket (annak jellegétől függetlenül) csak egyszer olvasol be a fájlból. Tehát **NEM** jó megoldás, ha a *"Van-e megfelelő mennyiségű dátum?"* kérdésre előbb külön megnézed hány sor van a fájlban, és ha ez rendben van, akkor "visszatérsz" a dátumok komponenseinek olvasásához! (Erre azért van szükség, mert ha standard Unix rendszerprogramként szeretnénk viselkedni, akkor képesnek kéne lenni a standard *input*-ról (hiányzó vagy '-' parancssori paraméter esetén) olvasni, ahol nincs szép lehetőség visszaugrásra. A standard *input* kezelésétől most eltekintünk!)

A főprogramod fordítási egységében, globális élettartammal és teljes láthatósággal (más fordítási egységből is elérhető) definiálj egy függvény pointer-t. A függvénypointered kezdőértéke a korábban implementált *"Consumer"* függvény, a típusa pedig ennek megfelelő legyen. Írd ki részletesen a függvénypointer típusát — típus-aliaszal vagy anélkül, — **és ne használj auto-t**.

Az adatszerkezetekbe való sikeres beolvasást követően egy újabb **megmért** blokkban legyen implementálva az algoritmus:

1. Válasszuk ki **"C"**-ből a legkisebb dátumot.
2. Ezt adjuk át a fent megadott *"consumer"* függvényre mutató pointernek, a függvényből válaszul kapott értéket azonban most csak eldobjuk.
3. A feldolgozott dátumot töröljük a **"C"** adatszerkezetből
4. Vegyük a `std::vector<Date>`-ből a balról következő dátumot, amelyet a **"C"**-ből törölt dátum helyére szúrjunk be.
5. Ez $(N+K)$ -szor fusson le ciklusban, amíg a beolvasott dátumok teljesen elfogyasztásra kerültek

Figyelj arra, hogy az algoritmus végrehajtásának optimalizálása miatt indokolt lehet, ha az adatszerkezetekből való "kivétel" ("törlés") nem minden esetben von maga után *fizikai* törlést (és pl. a `std::vector<Date>` esetében a sok fennmaradó elem egyel balra tologatását).

Az nyer, akinek a beadandója a leggyorsabb végrehajtást produkálja. A végrehajtás idejébe az algoritmust implementáló ciklus megmért idejét vesszük szigorúan figyelembe, és a beadandók összehasonlítása ugyanazokra az előregenerált random számokra fog történni.