Grunnatriði

Bergur Snorrason

16. janúar 2023

Grunntög og takmarkanir þeirra

- Í grunninn snýst forritun um gögn.

Grunntög og takmarkanir þeirra

- Í grunninn snýst forritun um gögn.
- Þegar við forritum flokkum við gögnin okkar með *tögum*.

Grunntög og takmarkanir þeirra

- Í grunninn snýst forritun um gögn.
- Þegar við forritum flokkum við gögnin okkar með *tögum*.
- Dæmi um tög í `C/C++` eru `int` og `double`.

Grunntög og takmarkanir þeirra

- Í grunninn snýst forritun um gögn.
- Þegar við forritum flokkum við gögnin okkar með *tögum*.
- Dæmi um tög í `C/C++` eru `int` og `double`.
- Helstu tögin í `C/C++` eru (yfirleitt):

Grunntög og takmarkanir þeirra

- Í grunninn snýst forritun um gögn.
- Þegar við forritum flokkum við gögnin okkar með *tögum*.
- Dæmi um tög í `C/C++` eru `int` og `double`.
- Helstu tögin í `C/C++` eru (yfirleitt):

Heiti	Lýsing	Skorður
`int`	Heiltala	Á bilinu $[-2^{31}, 2^{31} - 1]$
`unsigned int`	Heiltala	Á bilinu $[0, 2^{32} - 1]$
`long long`	Heiltala	Á bilinu $[-2^{63}, 2^{63} - 1]$
`unsigned long long`	Heiltala	Á bilinu $[0, 2^{64} - 1]$
`double`	Fleytitala	Takmörkuð nákvæmni
`char`	Heiltala	Á bilinu $[-128, 127]$

Hvað með tölur utan þessa bila?

- Einn helsti kostur `Python` í keppnisforritun er að heiltölur geta verið eins stórar (eða litlar) og vera skal.

Hvað með tölur utan þessa bila?

- Einn helsti kostur `Python` í keppnisforritun er að heiltölur geta verið eins stórar (eða litlar) og vera skal.

```
from math import factorial
print(factorial(100))
```

Hvað með tölur utan þessa bila?

- Einn helsti kostur `Python` í keppnisforritun er að heiltölur geta verið eins stórar (eða litlar) og vera skal.

```
from math import factorial
print(factorial(100))
```

```
109332621544394415268169923885626670049071596826438162
146859296389521759999322991560894146397615651828625369
792082722375825118521091686400000000000000000000000000
```

Hvað með tölur utan þessa bila?

- Einn helsti kostur `Python` í keppnisforritun er að heiltölur geta verið eins stórar (eða litlar) og vera skal.

```
from math import factorial
print(factorial(100))
```

```
109332621544394415268169923885626670049071596826438162
146859296389521759999322991560894146397615651828625369
792082722375825118521091686400000000000000000000000000
```

- Það er einnig hægt að nota `fractions` pakkann í `Python` til að vinna með fleytitölur án þess að tapa nákvæmni.

Hvað með tölur utan þessa bila?

- Sumir `C/C++` þýðendur bjóða upp á gagnatagið `__int128` (til dæmis `gcc`).

Hvað með tölur utan þessa bila?

- Sumir `C/C++` þýðendur bjóða upp á gagnatagið `__int128` (til dæmis `gcc`).
- Þetta tag býður upp á að nota tölur á bilinu $[-2^{127}, 2^{127} - 1]$.

Hvað með tölur utan þessa bila?

- Sumir `C/C++` þýðendur bjóða upp á gagnatagið `__int128` (til dæmis `gcc`).
- Þetta tag býður upp á að nota tölur á bilinu $[-2^{127}, 2^{127} - 1]$.
- Þetta þarf ekki að nota oft.

Röðun

- Við munum reglulega þurfa að raða gögnum í einhverja röð.

Röðun

- Við munum reglulega þurfa að raða gögnum í einhverja röð.

Forritunarmál	Röðun
`C`	`qsort(...)`
`C++`	`sort(...)`
`Python`	`this.sort()` eða `sorted(...)`

Röðun

- Við munum reglulega þurfa að raða gögnum í einhverja röð.

Forritunarmál	Röðun
`C`	`qsort(...)`
`C++`	`sort(...)`
`Python`	`this.sort()` eða `sorted(...)`

- Skoðum nú hvert forritunarmál til að sjá nánar hvernig föllin eru notuð.

Röðun í `C++`

- Í grunninn tekur `sort(...)` við tveimur gildum.

Röðun í `C++`

- Í grunninn tekur `sort(...)` við tveimur gildum.
- Fyrra gildið svarar til fyrsta staks þess sem við viljum raða og seinna gildið vísar á enda þess sem við viljum raða (ekki síðasta stakið)

Röðun í `C++`

- Í grunninn tekur `sort(...)` við tveimur gildum.
- Fyrra gildið svarar til fyrsta staks þess sem við viljum raða og seinna gildið vísar á enda þess sem við viljum raða (ekki síðasta stakið)
- Ef við erum með *n* staka fylki `a` þá röðum við því með `sort(a, a + n)`.

Röðun í `C++`

- Í grunninn tekur `sort(...)` við tveimur gildum.
- Fyrra gildið svarar til fyrsta staks þess sem við viljum raða og seinna gildið vísar á enda þess sem við viljum raða (ekki síðasta stakið)
- Ef við erum með n staka fylki `a` þá röðum við því með `sort(a, a + n)`.
- Við getum raðað flest öllum ílátum með `sort`.

Röðun í `C++`

- Í grunninn tekur `sort(...)` við tveimur gildum.
- Fyrra gildið svarar til fyrsta staks þess sem við viljum raða og seinna gildið vísar á enda þess sem við viljum raða (ekki síðasta stakið)
- Ef við erum með *n* staka fylki `a` þá röðum við því með `sort(a, a + n)`.
- Við getum raðað flest öllum ílátum með `sort`.
- Ef við erum með eitthvað ílát (til dæmis `vector`) `a` má raða með `sort(a.begin(), a.end())`.

Röðun í `C++`

- Í grunninn tekur `sort(...)` við tveimur gildum.
- Fyrra gildið svarar til fyrsta staks þess sem við viljum raða og seinna gildið vísar á enda þess sem við viljum raða (ekki síðasta stakið)
- Ef við erum með *n* staka fylki `a` þá röðum við því með `sort(a, a + n)`.
- Við getum raðað flest öllum ílátum með `sort`.
- Ef við erum með eitthvað ílát (til dæmis `vector`) `a` má raða með `sort(a.begin(), a.end())`.
- Við getum líka bætt við okkar eigin samanburðarfalli sem þriðja inntak.

Röðun í `C++`

- Í grunninn tekur `sort(...)` við tveimur gildum.
- Fyrra gildið svarar til fyrsta staks þess sem við viljum raða og seinna gildið vísar á enda þess sem við viljum raða (ekki síðasta stakið)
- Ef við erum með *n* staka fylki `a` þá röðum við því með `sort(a, a + n)`.
- Við getum raðað flest öllum ílátum með `sort`.
- Ef við erum með eitthvað ílát (til dæmis `vector`) `a` má raða með `sort(a.begin(), a.end())`.
- Við getum líka bætt við okkar eigin samanburðarfalli sem þriðja inntak.
- Það kemur þá í stað “minna eða samasem” samanburðarins sem er sjálfgefinn.

Röðun í `Python`

- Til að raða lista í `Python` þá má nota annað hvort `this.sort()` eða `sorted(...)`.

Röðun í `Python`

- Til að raða lista í `Python` þá má nota annað hvort `this.sort()` eða `sorted(...)`.
- Gerum ráð fyrir að listinn okkar heiti `a`.

Röðun í `Python`

- Til að raða lista í `Python` þá má nota annað hvort `this.sort()` eða `sorted(...)`.
- Gerum ráð fyrir að listinn okkar heiti `a`.
- Þá nægir að kalla á `a.sort()` og eftir það er `a` raðað.

Röðun í `Python`

- Til að raða lista í `Python` þá má nota annað hvort `this.sort()` eða `sorted(...)`.
- Gerum ráð fyrir að listinn okkar heiti `a`.
- Þá nægir að kalla á `a.sort()` og eftir það er `a` raðað.
- Hinsvegar skilar `sorted(a)` afriti af `a` sem hefur verið raðað.

Röðun í `Python`

- Til að raða lista í `Python` þá má nota annað hvort `this.sort()` eða `sorted(...)`.
- Gerum ráð fyrir að listinn okkar heiti `a`.
- Þá nægir að kalla á `a.sort()` og eftir það er `a` raðað.
- Hinsvegar skilar `sorted(a)` afriti af `a` sem hefur verið raðað.
- Til að raða `a` á þennan hátt þarf `a = sorted(a)`.

Röðun í `Python`

- Til að raða lista í `Python` þá má nota annað hvort `this.sort()` eða `sorted(...)`.
- Gerum ráð fyrir að listinn okkar heiti `a`.
- Þá nægir að kalla á `a.sort()` og eftir það er `a` raðað.
- Hinsvegar skilar `sorted(a)` afriti af `a` sem hefur verið raðað.
- Til að raða `a` á þennan hátt þarf `a = sorted(a)`.
- Nota má inntakið `key` til að raða eftir öðrum samanburðum.

Röðun í `Python`

- Til að raða lista í `Python` þá má nota annað hvort `this.sort()` eða `sorted(...)`.
- Gerum ráð fyrir að listinn okkar heiti `a`.
- Þá nægir að kalla á `a.sort()` og eftir það er `a` raðað.
- Hinsvegar skilar `sorted(a)` afriti af `a` sem hefur verið raðað.
- Til að raða `a` á þennan hátt þarf `a = sorted(a)`.
- Nota má inntakið `key` til að raða eftir öðrum samanburðum.
- Það er einnig inntak sem heitir `reverse` sem er Boole gildi sem leyfir auðveldlega að raða öfugt.

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.
- Fallið tekur fjögur viðföng:

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.
- Fallið tekur fjögur viðföng:
  - `void* a`. Þetta er fylkið sem við viljum raða.

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.
- Fallið tekur fjögur viðföng:
  - `void* a`. Þetta er fylkið sem við viljum raða.
  - `size_t n`. Þetta er fjöldi staka í fylkinu sem `a` svarar til.

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.
- Fallið tekur fjögur viðföng:
  - `void* a`. Þetta er fylkið sem við viljum raða.
  - `size_t n`. Þetta er fjöldi staka í fylkinu sem `a` svarar til.
  - `size_t s`. Þetta er stærð hvers staks í fylkinu okkar (í bætum).

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.
- Fallið tekur fjögur viðföng:
  - `void* a`. Þetta er fylkið sem við viljum raða.
  - `size_t n`. Þetta er fjöldi staka í fylkinu sem `a` svarar til.
  - `size_t s`. Þetta er stærð hvers staks í fylkinu okkar (í bætum).
  - `int (*cmp)(const void*, const void*)`. Þetta er samanburðarfallið okkar.

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.
- Fallið tekur fjögur viðföng:
  - `void* a`. Þetta er fylkið sem við viljum raða.
  - `size_t n`. Þetta er fjöldi staka í fylkinu sem `a` svarar til.
  - `size_t s`. Þetta er stærð hvers staks í fylkinu okkar (í bætum).
  - `int (*cmp)(const void*, const void*)`. Þetta er samanburðarfallið okkar.
- Síðasta inntakið er kannski flókið við fyrstu sýn en er einfalt fyrir okkur að nota.

Röðun í `C`

- Í `C` er enginn sjálfgefinn samanburður, svo við þurfum alltaf að skrifa okkar eigið samanburðarfall.
- Til röðunar notum við fallið `qsort(...)`.
- Fallið tekur fjögur viðföng:
  - `void* a`. Þetta er fylkið sem við viljum raða.
  - `size_t n`. Þetta er fjöldi staka í fylkinu sem `a` svarar til.
  - `size_t s`. Þetta er stærð hvers staks í fylkinu okkar (í bætum).
  - `int (*cmp)(const void*, const void*)`. Þetta er samanburðarfallið okkar.
- Síðasta inntakið er kannski flókið við fyrstu sýn en er einfalt fyrir okkur að nota.
- Þetta er *fallabendir* (e. *function pointer*) ef þið viljið kynna ykkur það frekar.

Röðun í `C`

```c
#include <stdio.h>
#include <stdlib.h>

int cmp(const void* p1, const void* p2)
{
    int x = *(int*)p1, y = *(int*)p2;
    return (x <= y) - (y <= x);
}

int rcmp(const void* p1, const void* p2)
{
    int x = *(int*)p1, y = *(int*)p2;
    return (x >= y) - (y >= x);
}

int main()
{
    int n, i;
    scanf("%d", &n);
    int a[n];
    for (i = 0; i < n; i++) scanf("%d", &a[i]);
    qsort(a, n, sizeof *a, cmp);
    for (i = 0; i < n; i++) printf("%d ", a[i]); printf("\n");
    qsort(a, n, sizeof *a, rcmp);
    for (i = 0; i < n; i++) printf("%d ", a[i]); printf("\n");
    return 0;
}
```

Uppsetning dæma

- Dæmin sem við sjáum á Kattis eru (oftast) af stöðluðu sniði.

Uppsetning dæma

- Dæmin sem við sjáum á Kattis eru (oftast) af stöðluðu sniði.
  - Saga.

Uppsetning dæma

- Dæmin sem við sjáum á Kattis eru (oftast) af stöðluðu sniði.
  - Saga.
  - Dæmið.

Uppsetning dæma

- Dæmin sem við sjáum á Kattis eru (oftast) af stöðluðu sniði.
  - Saga.
  - Dæmið.
  - Inntaks -og úttakslýsingar.

Uppsetning dæma

- Dæmin sem við sjáum á Kattis eru (oftast) af stöðluðu sniði.
  - Saga.
  - Dæmið.
  - Inntaks -og úttakslýsingar.
  - Sýnidæmi.

Uppsetning dæma

- Dæmin sem við sjáum á Kattis eru (oftast) af stöðluðu sniði.
  - Saga.
  - Dæmið.
  - Inntaks -og úttakslýsingar.
  - Sýnidæmi.
- Fyrstu tveir punktarnir geta verið blandaðir saman.

Uppsetning dæma

- Dæmin sem við sjáum á Kattis eru (oftast) af stöðluðu sniði.
  - Saga.
  - Dæmið.
  - Inntaks -og úttakslýsingar.
  - Sýnidæmi.
- Fyrstu tveir punktarnir geta verið blandaðir saman.
- Þeir eru líka lengsti hluti dæmisins.

A Different Problem

Write a program that computes the difference between non-negative integers.

Input

Each line of the input consists of a pair of integers. Each integer is between 0 and 10^{15} (inclusive). The input is terminated by end of file.

Output

For each pair of integers in the input, output one line, containing the absolute value of their difference.

Sample Input 1

```
10 12
71293781758123 72784
1 12345677654321
```

Sample Output 1

```
2
71293781685339
12345677654320
```

Röng lausn. Hver er villan?

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int a, b;
    while (cin >> a >> b)
    {
        cout << abs(a - b) << endl;
    }
    return 0;
}
```

Rétt lausn

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    long long a, b;
    while (cin >> a >> b)
    {
        cout << abs(a - b) << endl;
    }
    return 0;
}
```

`typedef`

- Við getum notað `typedef` til að spara okkur skriftir.

`typedef`

- Við getum notað `typedef` til að spara okkur skriftir.
- Við bætum við `typedef <gamla> <nýja>;` ofarlega í skrána.

`typedef`

- Við getum notað `typedef` til að spara okkur skriftir.
- Við bætum við `typedef <gamla> <nýja>;` ofarlega í skrána.
- Venjan í keppnisforritun er að nota `typedef long long ll;`.

`typedef`

- Við getum notað `typedef` til að spara okkur skriftir.
- Við bætum við `typedef <gamla> <nýja>;` ofarlega í skrána.
- Venjan í keppnisforritun er að nota `typedef long long ll;`.
- Við munum nota `typedef` aftur í námskeiðinu.

Rétt lausn með `typedef`

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

int main()
{
    ll a, b;
    while (cin >> a >> b)
    {
        cout << abs(a - b) << endl;
    }
    return 0;
}
```

`Time Limit Exceeded`

- Hvernig vitum að lausnin okkar sé of hæg?

`Time Limit Exceeded`

- Hvernig vitum að lausnin okkar sé of hæg?
- Ein leið er að útfæra lausnina, senda hana inn og gá hvað Kattis segir.

Time Limit Exceeded

- Hvernig vitum að lausnin okkar sé of hæg?
- Ein leið er að útfæra lausnina, senda hana inn og gá hvað Kattis segir.
- Það myndi þó spara mikla vinnu ef við gætum svarað spurningunni án þess að útfæra.

`Time Limit Exceeded`

- Hvernig vitum að lausnin okkar sé of hæg?
- Ein leið er að útfæra lausnina, senda hana inn og gá hvað Kattis segir.
- Það myndi þó spara mikla vinnu ef við gætum svarað spurningunni án þess að útfæra.
- Einnig gæti leynst önnur villa í útfærslunni okkar sem gefur okkur `Time Limit Exceeded` (`TLE`).

`Time Limit Exceeded`

- Hvernig vitum að lausnin okkar sé of hæg?
- Ein leið er að útfæra lausnina, senda hana inn og gá hvað Kattis segir.
- Það myndi þó spara mikla vinnu ef við gætum svarað spurningunni án þess að útfæra.
- Einnig gæti leynst önnur villa í útfærslunni okkar sem gefur okkur `Time Limit Exceeded` (`TLE`).
- Til að ákvarða hvort lausn sé nógu hröð þá notum við *tímaflækjur*.

`Time Limit Exceeded`

- Hvernig vitum að lausnin okkar sé of hæg?
- Ein leið er að útfæra lausnina, senda hana inn og gá hvað Kattis segir.
- Það myndi þó spara mikla vinnu ef við gætum svarað spurningunni án þess að útfæra.
- Einnig gæti leynst önnur villa í útfærslunni okkar sem gefur okkur `Time Limit Exceeded` (`TLE`).
- Til að ákvarða hvort lausn sé nógu hröð þá notum við *tímaflækjur*.
- Sum ykkar þekkja tímaflækjur en önnur kannski ekki.

`Time Limit Exceeded`

- Hvernig vitum að lausnin okkar sé of hæg?
- Ein leið er að útfæra lausnina, senda hana inn og gá hvað Kattis segir.
- Það myndi þó spara mikla vinnu ef við gætum svarað spurningunni án þess að útfæra.
- Einnig gæti leynst önnur villa í útfærslunni okkar sem gefur okkur `Time Limit Exceeded` (`TLE`).
- Til að ákvarða hvort lausn sé nógu hröð þá notum við *tímaflækjur*.
- Sum ykkar þekkja tímaflækjur en önnur kannski ekki.
- Skoðum fyrst hvað tímaflækjur eru í grófum dráttum.

Tímaflækjur í grófum dráttum

- Keyrslutími forrits er háður stærðinni á inntakinu.

Tímaflækjur í grófum dráttum

- Keyrslutími forrits er háður stærðinni á inntakinu.
- Tímaflækjan lýsir hvernig keyrslutími forritsins eykst þegar inntakið stækkar (í versta falli).

Tímaflækjur í grófum dráttum

- Keyrslutími forrits er háður stærðinni á inntakinu.
- Tímaflækjan lýsir hvernig keyrslutími forritsins eykst þegar inntakið stækkar (í versta falli).
- Ef forritið er með tímaflækju $\mathcal{O}(f(n))$ þýðir það að keyrslutíminn vex eins og f þegar n vex.

Tímaflækjur í grófum dráttum

- Keyrslutími forrits er háður stærðinni á inntakinu.
- Tímaflækjan lýsir hvernig keyrslutími forritsins eykst þegar inntakið stækkar (í versta falli).
- Ef forritið er með tímaflækju $\mathcal{O}(f(n))$ þýðir það að keyrslutíminn vex eins og f þegar n vex.
- Til dæmis ef forritið hefur tímaflækju $\mathcal{O}(n)$ þá tvöfaldast keyrslutími þegar inntakið tvöfaldast.

Tímaflækjur í grófum dráttum

- Keyrslutími forrits er háður stærðinni á inntakinu.
- Tímaflækjan lýsir hvernig keyrslutími forritsins eykst þegar inntakið stækkar (í versta falli).
- Ef forritið er með tímaflækju $\mathcal{O}(f(n))$ þýðir það að keyrslutíminn vex eins og f þegar n vex.
- Til dæmis ef forritið hefur tímaflækju $\mathcal{O}(n)$ þá tvöfaldast keyrslutími þegar inntakið tvöfaldast.
- Til annars dæmis ef forritið hefur tímaflækju $\mathcal{O}(n^2)$ þá faldast keyrslutími þegar inntakið tvöfaldast.

Tímaflækjur í grófum dráttum

- Keyrslutími forrits er háður stærðinni á inntakinu.
- Tímaflækjan lýsir hvernig keyrslutími forritsins eykst þegar inntakið stækkar (í versta falli).
- Ef forritið er með tímaflækju $\mathcal{O}(f(n))$ þýðir það að keyrslutíminn vex eins og f þegar n vex.
- Til dæmis ef forritið hefur tímaflækju $\mathcal{O}(n)$ þá tvöfaldast keyrslutími þegar inntakið tvöfaldast.
- Til annars dæmis ef forritið hefur tímaflækju $\mathcal{O}(n^2)$ þá fjórfaldast keyrslutími þegar inntakið tvöfaldast.

Tímaflækjur í grófum dráttum

- Keyrslutími forrits er háður stærðinni á inntakinu.
- Tímaflækjan lýsir hvernig keyrslutími forritsins eykst þegar inntakið stækkar (í versta falli).
- Ef forritið er með tímaflækju $\mathcal{O}(f(n))$ þýðir það að keyrslutíminn vex eins og f þegar n vex.
- Til dæmis ef forritið hefur tímaflækju $\mathcal{O}(n)$ þá tvöfaldast keyrslutími þegar inntakið tvöfaldast.
- Til annars dæmis ef forritið hefur tímaflækju $\mathcal{O}(n^2)$ þá fjórfaldast keyrslutími þegar inntakið tvöfaldast.
- Við gerum ráð fyrir að grunnaðgerðirnar okkar taki fastann tíma, eða séu með tímaflækju $\mathcal{O}(1)$.

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.
- Hún segir okkur til dæmis að tvöföld `for`-lykkja, þar sem hver `for`-lykkja er n löng, er $\mathcal{O}(\quad)$.

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.
- Hún segir okkur til dæmis að tvöföld `for`-lykkja, þar sem hver `for`-lykkja er n löng, er $\mathcal{O}(n^2)$.

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.
- Hún segir okkur til dæmis að tvöföld `for`-lykkja, þar sem hver `for`-lykkja er n löng, er $\mathcal{O}(n^2)$.
- Ef við erum með tvær einfaldar `for`-lykkjur, báðar af lengd n, þá er forritið $\mathcal{O}(n + n) = \mathcal{O}(\quad)$

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.
- Hún segir okkur til dæmis að tvöföld `for`-lykkja, þar sem hver `for`-lykkja er n löng, er $\mathcal{O}(n^2)$.
- Ef við erum með tvær einfaldar `for`-lykkjur, báðar af lengd n, þá er forritið $\mathcal{O}(n + n) = \mathcal{O}(n)$

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.
- Hún segir okkur til dæmis að tvöföld `for`-lykkja, þar sem hver `for`-lykkja er n löng, er $\mathcal{O}(n^2)$.
- Ef við erum með tvær einfaldar `for`-lykkjur, báðar af lengd n, þá er forritið $\mathcal{O}(n + n) = \mathcal{O}(n)$
- Einnig gildir að tímaflækja forritsins okkar takmarkast af hægasta hluta forritsins.

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.
- Hún segir okkur til dæmis að tvöföld `for`-lykkja, þar sem hver `for`-lykkja er n löng, er $\mathcal{O}(n^2)$.
- Ef við erum með tvær einfaldar `for`-lykkjur, báðar af lengd n, þá er forritið $\mathcal{O}(n+n) = \mathcal{O}(n)$
- Einnig gildir að tímaflækja forritsins okkar takmarkast af hægasta hluta forritsins.
- Til dæmis er $\mathcal{O}(n+n+n+n+n^2) = \mathcal{O}(\quad)$.

- Ef forritið okkar þarf að framkvæma $\mathcal{O}(f(n))$ aðgerð m sinnum þá er tímaflækjan $\mathcal{O}(m \cdot f(n))$.
- Þetta er reglan sem við notum oftast í keppnisforritun.
- Hún segir okkur til dæmis að tvöföld `for`-lykkja, þar sem hver `for`-lykkja er n löng, er $\mathcal{O}(n^2)$.
- Ef við erum með tvær einfaldar `for`-lykkjur, báðar af lengd n, þá er forritið $\mathcal{O}(n + n) = \mathcal{O}(n)$
- Einnig gildir að tímaflækja forritsins okkar takmarkast af hægasta hluta forritsins.
- Til dæmis er $\mathcal{O}(n + n + n + n + n^2) = \mathcal{O}(n^2)$.

Stærðfræði

- Við segjum að fall $g(x)$ sé í menginu $\mathcal{O}(f(x))$ ef til eru rauntölur c og x_0 þannig að

$$|g(x)| \leq c \cdot f(x)$$

fyrir öll $x > x_0$.

Stærðfræði

- Við segjum að fall $g(x)$ sé í menginu $\mathcal{O}(f(x))$ ef til eru rauntölur c og x_0 þannig að
$$|g(x)| \leq c \cdot f(x)$$
fyrir öll $x > x_0$.
- Þetta þýðir í raun að fallið $|g(x)|$ verður á endanum minna en $c \cdot f(x)$.

Stærðfræði

- Við segjum að fall $g(x)$ sé í menginu $\mathcal{O}(f(x))$ ef til eru rauntölur c og x_0 þannig að

  $$|g(x)| \leq c \cdot f(x)$$

  fyrir öll $x > x_0$.
- Þetta þýðir í raun að fallið $|g(x)|$ verður á endanum minna en $c \cdot f(x)$.
- Þessi lýsing undirstrikar betur að $f(x)$ er efra mat á $g(x)$, það er að segja $g(x)$ hagar sér ekki verr en $f(x)$.

Þekktar tímaflækjur

- Tímaflækjur algengra aðgerða eru:

Aðgerð	Lýsing	Tímaflækja
Línulega leit	Almenn leit í fylki	$\mathcal{O}(n)$
Helmingunarleit	Leit í röðuðu fylki	$\mathcal{O}(\log n)$
Röðun á heiltölum	Röðun á heiltalna fylki	$\mathcal{O}(n \log n)$
Strengjasamanburður	Bera saman tvo strengi af lengd n	$\mathcal{O}(n)$
Almenn röðun	Röðun með $\mathcal{O}(T(m))$ samanburð	$\mathcal{O}(T(m) \cdot n \log n)$

Þekktar tímaflækjur

- Tímaflækjur algengra aðgerða eru:

Aðgerð	Lýsing	Tímaflækja
Línulega leit	Almenn leit í fylki	$\mathcal{O}(n)$
Helmingunarleit	Leit í röðuðu fylki	$\mathcal{O}(\log n)$
Röðun á heiltölum	Röðun á heiltalna fylki	$\mathcal{O}(n \log n)$
Strengjasamanburður	Bera saman tvo strengi af lengd n	$\mathcal{O}(n)$
Almenn röðun	Röðun með $\mathcal{O}(T(m))$ samanburð	$\mathcal{O}(T(m) \cdot n \log n)$

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími" ekki endilega rétt orðið.

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími” ekki endilega rétt orðið.
- Við erum frekar að lýsa fjölda aðgerða sem forritið framkvæmir.

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími" ekki endilega rétt orðið.
- Við erum frekar að lýsa fjölda aðgerða sem forritið framkvæmir.
- Í keppnisforritun notum við 10^8 *regluna*:

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími" ekki endilega rétt orðið.
- Við erum frekar að lýsa fjölda aðgerða sem forritið framkvæmir.
- Í keppnisforritun notum við 10^8 *regluna*:
  - Tökum verstu tilfellin sem koma fyrir í inntakslýsingunni á dæminu, stingum því inn í tímaflækjuna okkar og deilum með 10^8.

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími" ekki endilega rétt orðið.
- Við erum frekar að lýsa fjölda aðgerða sem forritið framkvæmir.
- Í keppnisforritun notum við 10^8 *regluna*:
  - Tökum verstu tilfellin sem koma fyrir í inntakslýsingunni á dæminu, stingum því inn í tímaflækjuna okkar og deilum með 10^8.
  - Ef útkoman er minni en fjöldi sekúnda í tímamörkum dæmisins þá er lausnin okkar nógu hröð, annars er hún of hæg.

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími" ekki endilega rétt orðið.
- Við erum frekar að lýsa fjölda aðgerða sem forritið framkvæmir.
- Í keppnisforritun notum við 10^8 *regluna*:
  - Tökum verstu tilfellin sem koma fyrir í inntakslýsingunni á dæminu, stingum því inn í tímaflækjuna okkar og deilum með 10^8.
  - Ef útkoman er minni en fjöldi sekúnda í tímamörkum dæmisins þá er lausnin okkar nógu hröð, annars er hún of hæg.
- Þessa reglu mætti um orða sem: "Við gerum ráð fyrir að forritið geti framkvæmt 10^8 aðgerðir á sekúndu".

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími" ekki endilega rétt orðið.
- Við erum frekar að lýsa fjölda aðgerða sem forritið framkvæmir.
- Í keppnisforritun notum við 10^8 *regluna*:
  - Tökum verstu tilfellin sem koma fyrir í inntakslýsingunni á dæminu, stingum því inn í tímaflækjuna okkar og deilum með 10^8.
  - Ef útkoman er minni en fjöldi sekúnda í tímamörkum dæmisins þá er lausnin okkar nógu hröð, annars er hún of hæg.
- Þessa reglu mætti um orða sem: "Við gerum ráð fyrir að forritið geti framkvæmt 10^8 aðgerðir á sekúndu".
- Þessi regla er gróf nálgun, en virkar mjög vel því þetta er það sem dæmahöfundar hafa í huga þegar þeir semja dæmi.

10^8 reglan

- Þegar við ræðum tímaflækjur er „tími" ekki endilega rétt orðið.
- Við erum frekar að lýsa fjölda aðgerða sem forritið framkvæmir.
- Í keppnisforritun notum við 10^8 *regluna*:
  - Tökum verstu tilfellin sem koma fyrir í inntakslýsingunni á dæminu, stingum því inn í tímaflækjuna okkar og deilum með 10^8.
  - Ef útkoman er minni en fjöldi sekúnda í tímamörkum dæmisins þá er lausnin okkar nógu hröð, annars er hún of hæg.
- Þessa reglu mætti um orða sem: "Við gerum ráð fyrir að forritið geti framkvæmt 10^8 aðgerðir á sekúndu".
- Þessi regla er gróf nálgun, en virkar mjög vel því þetta er það sem dæmahöfundar hafa í huga þegar þeir semja dæmi.
- Með þetta í huga fáum við eftirfarandi töflu.

Stærð n	Versta tímaflækja	Dæmi
≤ 10	$\mathcal{O}((n+1)!)$	TSP með tæmandi leit
≤ 15	$\mathcal{O}(n^2 2^n)$	TSP með kvikri bestun
≤ 20	$\mathcal{O}(n 2^n)$	Kvik bestun yfir hlutmengi
≤ 100	$\mathcal{O}(n^4)$	Almenn spyrðing
≤ 400	$\mathcal{O}(n^3)$	Floyd-Warshall
$\leq 10^4$	$\mathcal{O}(n^2)$	Lengsti sameiginlegi hlutstrengur
$\leq 10^5$	$\mathcal{O}(n\sqrt{n})$	Reiknirit sem byggja á rótarþáttun
$\leq 10^6$	$\mathcal{O}(n \log n)$	Röðun (og margt fleira)
$\leq 10^8$	$\mathcal{O}(n)$	Línulega leit
$\leq 2^{10^8}$	$\mathcal{O}(\log n)$	Helmingunarleit
$> 2^{10^8}$	$\mathcal{O}(1)$	Ad hoc

`TLE` trikk

- Stundum fær maður `TLE` þótt maður sé viss um að lausnin sé nógu hröð.

`TLE` trikk

- Stundum fær maður `TLE` þótt maður sé viss um að lausnin sé nógu hröð.
- Ef forritið þarf að lesa eða skrifa mikið gæti það verið að hægja nóg á forritun til að gefa `TLE`.

`TLE` trikk

- Stundum fær maður `TLE` þótt maður sé viss um að lausnin sé nógu hröð.
- Ef forritið þarf að lesa eða skrifa mikið gæti það verið að hægja nóg á forritun til að gefa `TLE`.
- Þegar við lesum af staðalinntaki eða skrifum á staðalúttak þarf forritið að tala við stýrikerfið.

`TLE` trikk

- Stundum fær maður `TLE` þótt maður sé viss um að lausnin sé nógu hröð.
- Ef forritið þarf að lesa eða skrifa mikið gæti það verið að hægja nóg á forritun til að gefa `TLE`.
- Þegar við lesum af staðalinntaki eða skrifum á staðalúttak þarf forritið að tala við stýrikerfið.
- Slíkar að gerðir eru mjög hægar.

`TLE` trikk

- Stundum fær maður `TLE` þótt maður sé viss um að lausnin sé nógu hröð.
- Ef forritið þarf að lesa eða skrifa mikið gæti það verið að hægja nóg á forritun til að gefa `TLE`.
- Þegar við lesum af staðalinntaki eða skrifum á staðalúttak þarf forritið að tala við stýrikerfið.
- Slíkar að gerðir eru mjög hægar.
- Til að leysa þetta skrifa föll oft í *biðminni* (e. *buffer*) og prenta bara þegar það fyllist.

`TLE` trikk

- Stundum fær maður `TLE` þótt maður sé viss um að lausnin sé nógu hröð.
- Ef forritið þarf að lesa eða skrifa mikið gæti það verið að hægja nóg á forritun til að gefa `TLE`.
- Þegar við lesum af staðalinntaki eða skrifum á staðalúttak þarf forritið að tala við stýrikerfið.
- Slíkar að gerðir eru mjög hægar.
- Til að leysa þetta skrifa föll oft í *biðminni* (e. *buffer*) og prenta bara þegar það fyllist.
- Svona er þetta gert í `C`.

`TLE` trikk

- Í `C++` er biðminnið tæmt þegar `std::endl` er prentað.

`TLE` trikk

- Í `C++` er biðminnið tæmt þegar `std::endl` er prentað.
- Til að koma í veg fyrir þetta er hægt að prenta `\n` í staðinn.

`TLE` trikk

- Í `C++` er biðminnið tæmt þegar `std::endl` er prentað.
- Til að koma í veg fyrir þetta er hægt að prenta `\n` í staðinn.
- Til dæmis `cout « x « '\n'`.

`TLE` trikk

- Í `C++` er biðminnið tæmt þegar `std::endl` er prentað.
- Til að koma í veg fyrir þetta er hægt að prenta `\n` í staðinn.
- Til dæmis `cout « x « '\n'`.
- Það borgar sig einnig að setja `ios::sync_with_stdio(false)` fremst í `main()`.

`TLE` trikk

- Í `C++` er biðminnið tæmt þegar `std::endl` er prentað.
- Til að koma í veg fyrir þetta er hægt að prenta `\n` í staðinn.
- Til dæmis `cout « x « '\n'`.
- Það borgar sig einnig að setja `ios::sync_with_stdio(false)` fremst í `main()`.
- Ef þið eruð í `Java` mæli ég með `Kattio`.

`TLE` trikk

- Í `C++` er biðminnið tæmt þegar `std::endl` er prentað.
- Til að koma í veg fyrir þetta er hægt að prenta `\n` í staðinn.
- Til dæmis `cout « x « '\n'`.
- Það borgar sig einnig að setja `ios::sync_with_stdio(false)` fremst í `main()`.
- Ef þið eruð í `Java` mæli ég með `Kattio`.
- Það má finna á GitHub.

Innbyggðar gagnagrindur í `C++`

- Grunnur `C++` býr yfir mörgum sterkum gagnagrindum.

Innbyggðar gagnagrindur í `C++`

- Grunnur `C++` býr yfir mörgum sterkum gagnagrindum.
- Skoðum helstu slíku gagnagrindur og tímaflækjur mikilvægust aðgerða þeirra.

Innbyggðar gagnagrindur í `C++`

- Grunnur `C++` býr yfir mörgum sterkum gagnagrindum.
- Skoðum helstu slíku gagnagrindur og tímaflækjur mikilvægust aðgerða þeirra.
- Við munum bara fjalla um gagnagrindurnar í grófum dráttum.

Innbyggðar gagnagrindur í `C++`

- Grunnur `C++` býr yfir mörgum sterkum gagnagrindum.
- Skoðum helstu slíku gagnagrindur og tímaflækjur mikilvægust aðgerða þeirra.
- Við munum bara fjalla um gagnagrindurnar í grófum dráttum.
- Það er hægt að finna ítarlegra efni og dæmi um notkun á netinu.

Fylki

- Lýkt og í mörgum öðrum forritunarmálum eru fylki í `C++`.

Fylki

- Lýkt og í mörgum öðrum forritunarmálum eru fylki í `C++`.
- Fylki geyma gögn og eru af fastri stærð.

Fylki

- Lýkt og í mörgum öðrum forritunarmálum eru fylki í `C++`.
- Fylki geyma gögn og eru af fastri stærð.
- Þar sem þau eru af fastri stærð má gefa þeim tileinkað, aðliggjandi svæði í minni.

Fylki

- Lýkt og í mörgum öðrum forritunarmálum eru fylki í `C++`.
- Fylki geyma gögn og eru af fastri stærð.
- Þar sem þau eru af fastri stærð má gefa þeim tileinkað, aðliggjandi svæði í minni.
- Þetta leyfir manni að vísa í fylkið í $\mathcal{O}(1)$.

Fylki

- Lýkt og í mörgum öðrum forritunarmálum eru fylki í `C++`.
- Fylki geyma gögn og eru af fastri stærð.
- Þar sem þau eru af fastri stærð má gefa þeim tileinkað, aðliggjandi svæði í minni.
- Þetta leyfir manni að vísa í fylkið í $\mathcal{O}(1)$.

Aðgerð	Tímaflækja
Lesa eða skrifa ótiltekið stak	$\mathcal{O}(1)$
Bæta staki aftast	$\mathcal{O}(n)$
Skeyta saman tveimur	$\mathcal{O}(n)$

`vector`

- Gagnagrindin `vector` er að mestu leiti eins og fylki.

`vector`

- Gagnagrindin `vector` er að mestu leiti eins og fylki.
- Það má þó bæta stökum aftan á `vector` í $\mathcal{O}(1)$.

`vector`

- Gagnagrindin `vector` er að mestu leiti eins og fylki.
- Það má þó bæta stökum aftan á `vector` í $\mathcal{O}(1)$.
- Margir nota bara `vector` og aldrei fylki sem slík.

`vector`

- Gagnagrindin `vector` er að mestu leiti eins og fylki.
- Það má þó bæta stökum aftan á `vector` í $\mathcal{O}(1)$.
- Margir nota bara `vector` og aldrei fylki sem slík.

Aðgerð	Tímaflækja
Lesa eða skrifa ótiltekið stak	$\mathcal{O}(1)$
Bæta staki aftast	$\mathcal{O}(1)$
Skeyta saman tveimur	$\mathcal{O}(n)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    srand(time(NULL));
    int i, n = 10;
    vector<int> a;
    for (i = 0; i < n; i++) a.push_back(rand()%100);
    for (i = 0; i < n; i++) printf("%2d ", a[i]);
    printf("\n");
    sort(a.begin(), a.end());
    for (i = 0; i < n; i++) printf("%2d ", a[i]);
    printf("\n");
    sort(a.rbegin(), a.rend());
    for (i = 0; i < n; i++) printf("%2d ", a[i]);
    printf("\n");
    return 0;
}
```

```
>> ./a.out
83 23 26 24 52 74  0 39  0 90
 0  0 23 24 26 39 52 74 83 90
90 83 74 52 39 26 24 23  0  0
```

`list`

- Gagnagrindin `list` geymir gögn líkt og fylki gera, en stökin eru ekki aðliggjandi í minni.

`list`

- Gagnagrindin `list` geymir gögn líkt og fylki gera, en stökin eru ekki aðliggjandi í minni.
- Því er uppfletting ekki hröð.

`list`

- Gagnagrindin `list` geymir gögn líkt og fylki gera, en stökin eru ekki aðliggjandi í minni.
- Því er uppfletting ekki hröð.
- Aftur á móti er hægt að gera smávægilegar breytingar á `list` sem er ekki hægt að gera á fylkjum.

`list`

- Gagnagrindin `list` geymir gögn líkt og fylki gera, en stökin eru ekki aðliggjandi í minni.
- Því er uppfletting ekki hröð.
- Aftur á móti er hægt að gera smávægilegar breytingar á `list` sem er ekki hægt að gera á fylkjum.

Aðgerð	Tímaflækja
Finna stak	$\mathcal{O}(n)$
Bæta staki aftast	$\mathcal{O}(1)$
Bæta staki fremst	$\mathcal{O}(1)$
Bæta staki fyrir aftan tiltekið stak	$\mathcal{O}(1)$
Bæta staki fyrir framan tiltekið stak	$\mathcal{O}(1)$
Skeyta saman tveimur	$\mathcal{O}(1)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int i;
    list<int> a;
    for (i = 1; i <= 5; i++) a.push_back(i);
    for (i = 1; i <= 5; i++) a.push_front(i);
    for (auto p : a) printf("%d ", p);
    printf("\n");
    return 0;
}
```

```
>> ./a.out
5 4 3 2 1 1 2 3 4 5
```

`stack`

- Gagnagrindin `stack` geymir gögn og leyfir aðgang að síðasta staki sem var bætt við.

`stack`

- Gagnagrindin `stack` geymir gögn og leyfir aðgang að síðasta staki sem var bætt við.

Aðgerð	Tímaflækja
Bæta við staki	$\mathcal{O}(1)$
Lesa nýjasta stakið	$\mathcal{O}(1)$
Fjarlægja nýjasta stakið	$\mathcal{O}(1)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int i, n = 10;
    stack<int> a;
    for (i = 0; i < 10; i++) a.push(i);
    while (a.size() > 0)
    {
        printf("%d ", a.top());
        a.pop();
    }
    printf("\n");
    return 0;
}
```

```
>> ./a.out
9 8 7 6 5 4 3 2 1 0
```

`queue`

- Gagnagrindin `queue` geymir gögn og leyfir aðgang að fyrsta stakinu sem var bætt við.

`queue`

- Gagnagrindin `queue` geymir gögn og leyfir aðgang að fyrsta stakinu sem var bætt við.

Aðgerð	Tímaflækja
Bæta við staki	$\mathcal{O}(1)$
Lesa elsta stakið	$\mathcal{O}(1)$
Fjarlægja elsta stakið	$\mathcal{O}(1)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int i, n = 10;
    queue<int> a;
    for (i = 0; i < 10; i++) a.push(i);
    while (a.size() > 0)
    {
        printf("%d ", a.front());
        a.pop();
    }
    printf("\n");
    return 0;
}
```

```
>> ./a.out
0 1 2 3 4 5 6 7 8 9
```

`set`

- Gagnagrindin `set` geymir gögn án endurtekninga og leyfir hraða uppflettingu.

`set`

- Gagnagrindin `set` geymir gögn án endurtekninga og leyfir hraða uppflettingu.

Aðgerð	Tímaflækja
Bæta við staki	$\mathcal{O}(\log n)$
Fjarlægja stak	$\mathcal{O}(\log n)$
Gá hvort staki hafi verið bætt við	$\mathcal{O}(\log n)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    srand(time(NULL));
    int i, n = 10;
    set<int> a;
    while (a.size() < 10) a.insert(rand()%100);
    while (a.size() > 0)
    {
        printf("%2d ", *a.begin());
        a.erase(a.begin());
    }
    printf("\n");
    return 0;
}
```

```
>> ./a.out
 2  9 18 29 34 42 43 46 91 97
```