Algoritmi care lucreaza cu tablouri unidimensionale
27.02.2021

1. Deplasarea elementelor nule dintr-un tablou la începutul tabloului

Fie X un tablou unidimensional de numere întregi. Scrieți un program care mută toate valorile nule din tablou în partea stângă a acestuia, menținând ordinea celorlalte elemente. Tabloul se va modifica *in-place,* fără folosirea altor structuri de date adiționale.

De exemplu, pentru tabloul de mai jos

2	7	0	8	1	0	3

, după mutarea elementelor nule în partea stângă se va obține tabloul:

0	0	2	7	8	1	3

Vom lucra cu 2 iteratori: un iterator pentru citire (R) și un iterator pentru scriere (W).

Un iterator este un element care ne permite să traversăm o structură de date de tip *container* - ce conține mai multe element; exemple de asemenea structuri sunt tablourile sau listele înlănțuite. În cazul nostru, când spunem iterator ne referim la un simplu index în tablou.

Ideea de rezolvare este simplă: se va împărți tabloul în două zone: zona ne-explorată (din stânga iteratorului de citire) și zona în care există doar elemente ne-nule (din dreapta iteratorului de scriere, excluzând elementul pe care e poziționat acesta).

La începutul algoritmului ambii iteratori sunt poziționați pe ultima poziție din tablou: nu am „vizitat” nici un element din tablou prin intermediul iteratorului R și nici nu am adăugat vreun element în zona cu elemente nenule.

Apoi, prin intermediul iteratorului de citire vom începe „explorarea”: analizăm fiecare element din tablou, începând cu ultimul element. Dacă elementul este nenul, îl vom muta în zona cu elemente nenule din tablou.

Ce presupune această operație? Am definit zona cu elemente nenule ca fiind zona din dreapta iteratorului de scriere (W), deci tot ce trebuie să facem este „să-i facem loc” elementului în această zonă. În enunțul problemei se specifică faptul că ordinea elementelor nenule nu trebuie să se schimbe. Având în vedere că explorarea tabloului se face de la dreapta la stânga, rezultă că, pentru a respecta această constrângere, elementul nenul trebuie adăugat la începutul zonei cu elemente nenule. Deci, vom copia elementul nenul pe poziția iteratorului de scriere și apoi vom decrementa iteratorul de scriere.

Cu alte cuvinte, iteratorul pentru citire (R) se deplasează către începutul tabloului și se analizează elementul pe care este poziționat:

- Dacă elementul pe care poziționat iteratorul de citire R este nul, nu se schimbă nimic
- Dacă elementul pe care este poziționat iteratorul de citire R este nenul, această valoare este copiată pe poziția iteratorului de scriere, și valoarea acestuia este decrementată.

În final, nu ne rămâne decât să setăm pe valoarea 0 toate elementele până la iteratorul de scriere (inclusiv acesta).

Cu siguranță totul va deveni mai clar dacă vom trasa algoritmul:

2	7	0	8	1	0	3	La început atât iteratorul de scriere cât și cel de citire sunt poziționate pe ultimul element.
						R W	

2	7	0	8	1	0		3	Iteratorul de citire este poziționat pe un element nenul. Se copiază elementul pe poziția iteratorului de scriere W și se decrementează valoarea lui W. Iteratorul R trece la elementul din stânga (este decrementat)
					R	W		

2	7	0	8	1	0	3	Iteratorul de citire este poziționat pe un element nul. Iteratorul R trece la elementul din stânga (este decrementat)
				R	W		

2	7	0	8	1	1	3	Iteratorul de citire este poziționat pe un element nenul. Se copiază elementul pe poziția iteratorului de scriere W și se decrementează valoarea lui W. Iteratorul R trece la elementul din stânga (este decrementat)
			R	W			

2	7	0	8	8	1	3	Iteratorul de citire este poziționat pe un element nenul. Se copiază elementul pe poziția iteratorului de scriere W și se decrementează valoarea lui W. Iteratorul R trece la elementul din stânga (este decrementat)
		R	W				

2	7	0	8	8	1	3	Iteratorul de citire este poziționat pe un element nul. Iteratorul R trece la elementul din stânga (este decrementat)
	R		W				

2	7	0	7	8	1	3	Iteratorul de citire este poziționat pe un element nenul. Se copiază elementul pe poziția iteratorului de scriere W și se decrementează valoarea lui W. Iteratorul R trece la elementul din stânga (este decrementat)
	R	W					

2	7	2	7	8	1	3	Iteratorul de citire este poziționat pe un element nenul. Se copiază elementul pe poziția iteratorului de scriere W și se decrementează valoarea lui W. Iteratorul R trece la elementul din stânga (este decrementat)
R	W						

0	0	2	7	8	1	3	Am explorat tot tabloul. Setăm elementele din stânga iteratorului de scriere, inclusiv acesta, pe valoarea 0.
R	W						

Complexitatea algoritmului propus este $O(N)$, unde N este dimensiunea tabloului de intrare.

Implementare în limbajul C

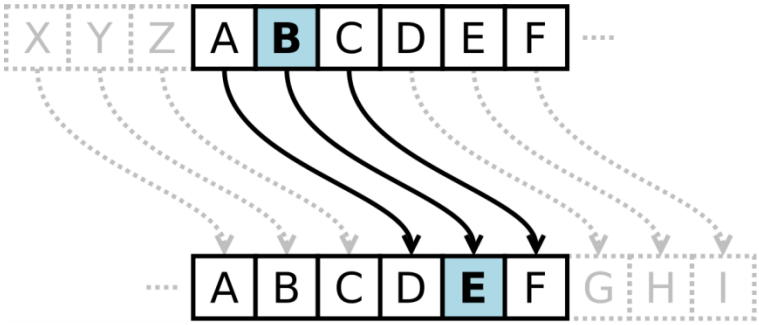
```
#include <iostream>
using namespace std;
```

```
void afisare(int arr[], int sz);
void deplasare_zerouri_stanga(int arr[], int dim);

void afisare(int arr[], int sz){
    cout<<"[";
    for(int i = 0; i < sz; i++){
        cout<<arr[i];
        if(i < sz - 1)
            cout<<", ";
    }
    cout<<"]"<<endl;
}

void deplasare_zerouri_stanga(int arr[], int dim) {

    int it_scriere = dim - 1;
    int it_citire = dim - 1;

    while(it_citire >= 0) {
        if(arr[it_citire]) {
            arr[it_scriere] = arr[it_citire];
            it_scriere--;
        }
        it_citire--;
    }
    while(it_scriere >= 0) {
        arr[it_scriere] = 0;
        it_scriere--;
    }
}

int main() {
    int v[] = {2, 7, 0, 8, 1, 0, 3};
    int n = sizeof(v) / sizeof(v[0]);

    cout <<"Tabloul initial este: " <<endl;
    afisare(v, n);
    deplasare_zerouri_stanga(v, n);
    cout << "Tabloul dupa deplasarea elementelor nule spre stanga: "<<
endl;
    afisare(v, n);

    return 0;
}
```

2. Criptarea folosind cifrul lui Cezar. Spargerea cifrului folosind analiza de frecvență

Este cunoscut faptul că împăratul roman Iulius Cezar obișnuia să trimită mesajele militare importante folosind următoarea regulă: fiecare literă din textul original era înlocuită cu o altă literă situată la o distanță fixă de aceasta.

Dacă avea ceva confidențial de comunicat, scria încifrat, adică schimba ordinea literelor din alfabet, astfel încât nu se putea înțelege nici un cuvânt. Dacă cineva dorește să descifreze și să înțeleagă, trebuie să înlocuiască a patra literă din alfabet, adică D, cu A, și așa mai departe pentru celelalte. — Suetoniu, Viața lui Iulius Cezar 56



Rezolvare.

În zilele noastre, cifrul lui Cezar este una din cele mai simple metode de criptare.
Pentru început vom scrie o funcție pentru criptarea și decriptarea unor mesaje codificate folosind cifrul lui Cezar. Semnele de punctuație și cifrele nu se vor modifica.

Această funcție va avea ca parametri un text pe care vrem să îl criptăm (un tablou unidimensional de caractere) *str* și deplasamentul pe care vrem să îl folosim pentru criptare *dep*. Funcția va înlocui fiecare literă din text cu litera aflată la o distanță de *dep* poziții la dreapta în alfabetul englez. Observăm că putem folosi această funcție și pentru decriptare: în acest caz deplasamentul pe care îl furnizăm funcției trebuie să fie 26 – *dep*, unde *dep* este deplasamentul folosit pentru a cripta textul.

```
char* codifica_cifru_Cezar(const char* text, int deplasament){
    char* rezultat = new char[strlen(text) + 1];

    for (unsigned int i = 0; i < strlen(text); i++)
    {
        if(isalpha(text[i])){
            if (isupper(text[i]))
                rezultat[i] = (text[i] + deplasament - 'A') % 26 + 'A';
            else
                rezultat[i] = (text[i] + deplasament - 'a') % 26 + 'a';
        }else
            rezultat[i] = text[i];
    }
    rezultat[strlen(text)] = '\0';

    return rezultat;
}
```

De exemplu, dacă criptăm textul

quo deorum ostenta et inimicorum iniquitas uocat. Iacta alea est!

folosind deplasamentul 4, vom obține textul:

uys hisvyq swxirxe ix mrmqmgsvyq mrmuymxew ysgex. Megxe epie iwx!

Spargerea cifrului lui Cezar

În continuare vom folosi analiza de frecvență pentru a sparge cifrul lui Cezar (aceasta este o altă metodă de tip *brute force* pentru spargerea cifrului). Spre deosebire de problema decriptării, în acest caz avem doar un text criptat, dar nu cunoaștem deplasamentul care s-a folosit în criptarea textului.
Analiza de frecvență se bazează pe faptul că anumite litere (sau combinații de litere) apar mai des într-o limbă, indiferent de mărimea textului. De exemplu, în limba engleză literele E şi A sunt cele mai des întâlnite, în timp ce Z şi Q apar cel mai rar. Distribuția tuturor caracterelor din limba engleză este prezentată în figura de mai jos:

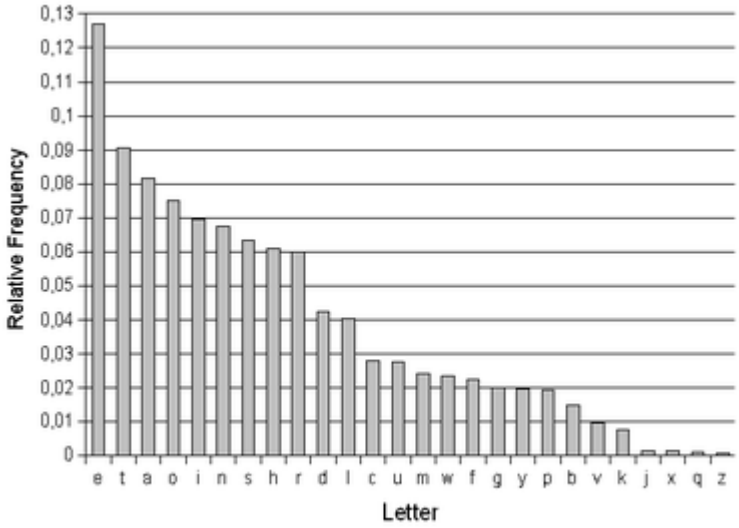


Figura 2. Distribuția literelor din alfabetul englez

Ideea acestei metode este de a calcula frecvența literelor textului criptat și de a compare compara frecvența literelor cu distanta *Chi-Squared*:

$$\chi^2(C,E) = \sum_{i='A'}^{i='Z'} \frac{(C_i - E_i)^2}{E_i}$$

, unde C_i reprezintă numărul real de apariții ale celui de-al *i*-lea caracter, iar E_i este numărul previzionat de apariții ale celui de-al *i*-lea caracter al alfabetului.
Formula pare complicată la prima vedere, dar în realitate ea nu este chiar așa. În principiu, pentru fiecare caracter posibil (intervalul de variație al lui *i* este de la 'A' la 'Z'), măsurăm discrepanța dintre numărul său de apariții în textul criptat text (C_i) şi numărul său previzionat de apariții în textele scrise în limba engleză (E_i); diferența C_i - E_i este ridicată la pătrat (în limba engleză: *squared*) astfel încât să eliminăm valorile negative. Împărțirea la E_i este pur şi simplu un factor de normalizare.
Cu cât $\chi^2(C,E)$ este mai mică, cu atât histogramele C şi E sunt mai asemănătoare.
Întrucât acest algoritm constituie totodată o metodă de *brute force* pentru spargerea cifrului, este necesar să calculați histogramele pentru toate deplasările posibile şi să calculaţi distanţa *Chi Squared*

dintre aceste histograme şi distribuţia medie a caracterelor în limba engleză. Deplasarea care prezintă cea mai mică distanţa *Chi Squared* reprezintă soluția.

Ca să recapitulăm, pentru a rezolva sparge cifrul lui Cezar trebuie să realizăm următorii pași:

1. Să scriem o funcție care calculează frecvenţa apariţiei fiecărui caracter (o histogramă) într-un anumit text.
2. Să scriem o funcţie care calculează distanţa *Chi Squared* dintre două histograme
3. Să scriem o funcţie care sparge cifrul lui Cezar folosind analiza de frecvenţă: deplasează în mod iterativ codul criptat prin toate permutările posibile, calculează distanţa *Chi Squared* dintre fiecare permutare şi distribuţia aproximativă a literelor în limba engleză, iar în final returnează permutarea cu distanţa *Chi Squared* cea mai mică drept soluție.

Încercați să decriptați mesajul:

Bksco iyeb aekvsdi cdkxnkbnc kc rsqr kc iye mkx vsfo gsdr, kfysn gkcdsxq iyeb dswo yx byedsxo zbylvowc, kxn kvgkic dbi dy gybu kc mvycovi kc zyccslvo kd dro lyexnkbi yp iyeb klsvsdsoc. Ny drsc, lomkeco sd sc dro yxvi gki yp nscmyfobsxq ryg drkd lyexnkbi cryevn lo wyfon pybgkbn

Cswzvsmsdi sc k qbokd fsbdeo led sd boaesboc rkbn gybu dy kmrsofo sd kxn onemkdsyx dy kzzbomskdo sd. Kxn dy wkuo wkddobc gybco: mywzvohsdi covvc loddob

Vom adresa fiecare din aceste probleme pe rând.

<u>Calcularea frecvenței de apariție a unui caracter într-un text</u>

Această funcție va primi ca intrare un tablou unidimensional de caractere și va returna un tablou de numere întregi care stochează frecvența de apariție a fiecărui caracter în text: o histogramă.

Dimensiunea tabloului de ieșire este 26 (numărul de caractere din alfabetul englez). Funcția nu va face diferența între literele mari și mici ale alfabetului.

```
float* calculeaza_histograma_normalizata(const char* str){
    unsigned int num_litere = 'z' - 'a' + 1;
    float* hist = new float[num_litere]();

    for(size_t i = 0; i < strlen(str); i++){
        if(isalpha(str[i])){
            int ch = tolower(str[i]);
            hist[ch - 'a']++;
        }
    }
    for(size_t i = 0; i < num_litere; i++){
        hist[i] /= strlen(str);
    }

    return hist;
}
```

<u>Calcularea distanței Chi squared între două histograme</u>

Această funcție va primi ca două histograme, precum și lungimea acestora, și va returna o valoare de tip real care reprezintă distanța Chi-Squared dintre aceste două histograme.

```
float calculeaza_distanta_chi_squared(float *h1, float *h2, int sz){
    float dist = 0;
```

```
    for(int i = 0; i < sz; i++)
        dist += (h1[i] - h2[i])*(h1[i] - h2[i])/(h2[i] +
std::numeric_limits<float>::epsilon());

    return dist;
}
```

Spargerea cifrului lui Cezar folosind analiza frecvenței

```
char* sparge_cifru_Cezar(const char* text){

    float frecventa_alfabet_englez[] = {0.08167f, 0.01492f, 0.02782f,
0.04253f, 0.12702f, 0.02228f, 0.02015f, 0.06094f, 0.06966f, 0.00153f,
0.00772f, 0.04025f, 0.02406f, 0.06749f, 0.07507f, 0.01929f, 0.00095f,
0.05987f, 0.06327f, 0.09056f, 0.02758f, 0.00978f, 0.02360f, 0.00150f,
0.01974f, 0.00074f};
    float min_dist = std::numeric_limits<float>::max();
    int min_idx = -1;
    for(int i = 1; i <= 25; i++){
        char* text_decriptat = codifica_cifru_Cezar(text, 26 - i);

        float* hist = calculeaza_histograma_normalizata(text_decriptat);
        float dist = calculeaza_distanta_chi_squared(hist,
frecventa_alfabet_englez, 26);

        if(min_dist > dist){
            min_dist = dist;
            min_idx = i;
        }
        delete[] hist;
        delete[] text_decriptat;
    }

    return codifica_cifru_Cezar(text, 26 - min_idx);
}
```

3. Găsirea unei perechi cu suma *K* într-un tablou unidimensional.

Fiind dat un tablou de numere întregi *A* de dimensiune *N* și o valoare întreagă *K*, să se determine dacă în tabloul *A* există o pereche de elemente care au suma egală cu *K*.
De exemplu:

- pentru tabloul [2, 7, 0, 8, 1, 0, 3] și *K* = 10 se va afișa DA, deoarece există două perechi cu suma 10: (2, 8) și (7, 3).
- pentru tabloul [2, 100, 23, 40, 19, 2] și *K* = 99, se va afișa NU, deoarece nu există nici o pereche ale cărei elemente însumate să dea valoarea 99.

Rezolvare.
Varianta 1. Metoda directă și simplă de rezolvare este să generăm toate perechile posibile și să verificăm dacă există o pereche cu suma egală cu K.
Implementarea acestei metode o găsim mai jos:

```
int exista_pereche_suma_k(int tab[], int dim, int K){

    for(int i = 0; i < dim - 1; i++){
        for(int j = i + 1; j < dim; j++){
            if(tab[i] + tab[j] == K)
                return 1;
        }
    }
    return 0;
}
```

Funcția primește ca parametrii un tablou unidimensional *tab*, dimensiunea acestuia *dim* și valoarea *K* și returnează valoarea 1 dacă există o pereche cu această proprietate, respectiv 0 în caz contrar. Folosim două **for**-uri imbricate pentru a genera toate perechile de numere, iar când găsim o pereche cu suma egală cu *K* ne oprim.
Complexitatea acestei metode este $O(N^2)$, unde *N* este dimensiunea șirului de intrare; în cazul cel mai defavorabil, numărul de perechi pe care le putem genera este $N * (N - 1) / 2$.
Varianta 2. Să ne gândim acum o la o varianta mai eficientă de implementare. Am putea sorta tabloul de intrare, și pentru fiecare element din tablou *tab[i]*, să verificăm dacă valoarea *K – tab[i]* se află în tablou. Căutarea binară ne permite să căutam un element într-un tablou sortat, iar complexitatea acestei metode de căutare este $O(\log N)$, unde *N* este dimensiunea tabloului de intrare.
Deci, pentru a rezolva problema prin această problemă, va trebui să parcurgem următorii pași:

1. Să sortăm tabloul în ordine crescătoare;
2. Pentru fiecare element din tablou *tab[i]*, să folosim căutarea binară pentru a determina dacă elementul *K - tab[i]* este prezent în tablou. Când am găsit prima pereche cu această proprietate putem să ne oprim și să returnăm valoarea 1 (am găsit o pereche cu această proprietate).
3. Dacă am trecut prin toate elementele din tablou și nu am găsit nici un element cu această proprietate, returnăm valoarea 0.

Pentru a sorta tabloul putem folosi metoda *qsort()* din C/C++ care, începând cu standardul c++11, garantează o complexitate de $O(N \log N)$. Sau dacă vreți să implementați de la zero o metodă de sortare, să alegeți o metodă eficientă cum ar fi sortarea prin interclasare.
Implementarea acestei metode o găsim mai jos:

```
int compara(const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}

int cautare_binara(int tab[], int start, int final, int val){

    if (final >= start) {
        int mid = start + (final - start) / 2;
        // daca elementul cautat este cel din mijloc
        if (tab[mid] == val)
            return mid;

        // daca elementul pe care il cautam mai mic decat elementul din
mijlocul intervalului,
        // atunci trebuie sa cautam elementul la stanga elementului
curent
        if (tab[mid] > val)
            return cautare_binara(tab, start, mid - 1, val);

        // altfel trebuie sa cautam elementul la dreapta elementului
curent
        return cautare_binara(tab, mid + 1, final, val);
    }

    // daca ajungem pana aici, atunci elementul nu este prezent in tablou
    return -1;
}

int exista_pereche_suma_k_bs(int tab[], int dim, int K){
    // sortam tabloul
    qsort(tab, dim, sizeof(tab[0]), compara);
    // pentru fiecare element din tablou
    for(int i = 0; i < dim; i++){
        // cautam daca exista un element cu valoarea K - tab[i] folosind
cautarea binara
        int idx = cautare_binara (tab, 0, dim - 1, K - tab[i]);
        // daca am gasit elementul cu aceasta valoare, ne oprim
        if (idx != -1)
            return 1;
    }
// daca ajungem pana aici inseamna ca nu am gasit nici o pereche cu suma
K
    return 0;
}
```

Deși avem de scris puțin mai mult cod, complexitatea aceste metode este $O(N \log N)$: sortarea printr-o metodă eficientă $O(N \log N)$, apoi pentru fiecare element *tab[i]* (avem N elemente) aplicăm căutarea binară (cu complexitate $\log N$) ca să căutam elementul K – *tab[i]*.
Merită acest efort din partea noastră? Să vedem care este diferența din punct de vedere al timpului de execuție al celor două metode pentru tablouri de diferite dimensiuni.
Pentru început vom scrie o funcție care ne generează un tablou uni-dimensional de dimensiune N și îl inițializează cu elemente aleatorii pozitive.

```
int* genereaza_tablou(int dim){
    int * tab = new int[dim];
    for(int  i = 0; i < dim; i++)
        // functia rand() genereaza un numar aleator intre 0 si RAND_MAX
        tab[i] = rand();
    return tab;
}
 void compara_timp_executie(){
     clock_t start, final;
     double durata_v1, durata_v2;
     int K = -1;
     cout<<setw(10)<<"Dimensiume"<<"\t"<<setw(8)<<"v1
 (ms)"<<"\t"<<setw(8)<<"v2 (ms)"<<endl;
     for(int dim = 100; dim < 500000; dim+=50000){
         int* tab = genereaza_tablou(dim);

         start = clock();
         exista_pereche_suma_k(tab, dim, K);
         final = clock();
         durata_v1 = (double)(final - start) / CLOCKS_PER_SEC*1000;

         start = clock();
         exista_pereche_suma_k_bs(tab, dim, K);
         final = clock();
         durata_v2 = (double)(final - start) / CLOCKS_PER_SEC*1000;

 cout<<setw(10)<<dim<<"\t"<<setw(8)<<durata_v1<<"\t"<<setw(8)<<durata_v2
 <<endl;
         delete[] tab;
     }
 }
```

Apoi vom scrie un program care generează mai tablouri inițializate aleatoriu de dimensiuni din ce în ce mai mari și calculează timpul de execuție pentru fiecare din cele două metode pe care le-am scris. Fiindcă ne interesează cazul defavorabil, vom seta K pe valoarea -1 pentru a ne asigura că în tablourile generate aleatoriu nu vor exista două elemente cu suma K.
Am obținut următorii timpi de execuție.

Dimensiume	v1 (ms)	v2 (ms)
100	0	0
50100	781	9
100100	3022	23
150100	6527	41
200100	11698	39
250100	18298	50
300100	27044	70
350100	36350	82
400100	40876	71
450100	50955	82

Ca fapt divers, există o altă metodă mai eficientă de a rezolva această problemă, bazată pe tabele de dispersie, dar acestea nu fac subiectul acestei întâlniri.

În fine, în tabelul de mai jos aveți prezentați timpii de execuție pentru algoritmi de diferite complexități; mai precis el arată cât timp îi ia unui algoritm care utilizează $f(n)$ operații pentru a fi executat pe un calculator în care fiecare operație are durata de 1 ns. Tabelul este preluat din cartea: *The algorithm design manual*, Steven S. Skiena.

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		