

Δομές Δεδομένων και Αλγόριθμοι - Εργαστήριο 7

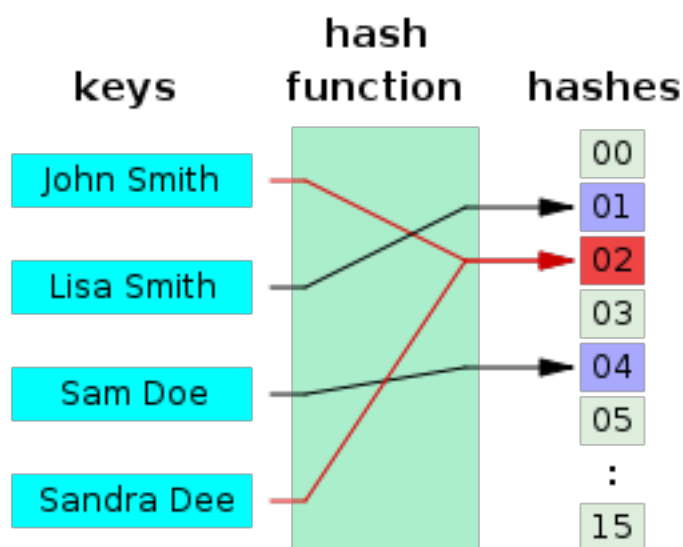
Κατακερματισμός

Τ.Ε.Ι. Ηπείρου, Τμήμα Μηχανικών Πληροφορικής Τ.Ε.
Χρήστος Γκόγκος - Αναπληρωτής Καθηγητής

1 Εισαγωγή

2 Τι είναι ο κατακερματισμός;

Ο κατακερματισμός (hashing) είναι μια μέθοδος που επιτυγχάνει ταχύτατη αποθήκευση και αναζήτηση δεδομένων. Σε ένα σύστημα κατακερματισμού τα δεδομένα αποθηκεύονται σε έναν πίνακα που ονομάζεται πίνακας κατακερματισμού (hash table). Εφαρμόζοντας στο κλειδί κάθε εγγραφής που πρόκειται να αποθηκευτεί ή να αναζητηθεί τη συνάρτηση κατακερματισμού (hash function) προσδιορίζεται μονοσήμαντα η θέση του πίνακα στην οποία τοποθετούνται τα δεδομένα της εγγραφής. Μια καλή συνάρτηση κατακερματισμού θα πρέπει να κατανέμει τα κλειδιά στα κελιά του πίνακα κατακερματισμού όσο πιο ομοιόμορφα γίνεται και να είναι εύκολο να υπολογιστεί.



Σχήμα 1:

Είναι επιθυμητό το παραγόμενο αποτέλεσμα από τη συνάρτηση κατακερματισμού να εξαρτάται από το κλειδί στο σύνολό του.

Οι πίνακες κατακερματισμού είναι ιδιαίτερα κατάλληλοι για εφαρμογές στις οποίες πραγματοποιούνται συχνές αναζητήσεις εγγραφών με δεδομένες τιμές κλειδιών. Ωστόσο, οι πίνακες κατακερματισμού έχουν και μειονεκτήματα καθώς είναι δύσκολο να επεκταθούν από τη στιγμή που έχουν δημιουργηθεί και μετά. Επίσης, η απόδοσή των πινάκων κατακερματισμού υποβαθμίζεται καθώς οι θέσεις τους γεμίζουν με στοιχεία. Συνεπώς, εφόσον ο προγραμματιστής προχωρήσει στη δική του υλοποίηση ενός πίνακα κατακερματισμού είτε θα πρέπει

να γνωρίζει εκ των προτέρων το πλήθος των στοιχείων που πρόκειται να αποθηκευτούν είτε όταν αυτό απαιτηθεί να υπάρχει πρόβλεψη έτσι ώστε τα δεδομένα να μεταφέρονται σε μεγαλύτερο πίνακα κατακερματισμού.

Στις περισσότερες εφαρμογές υπάρχουν πολύ περισσότερα πιθανά κλειδιά εγγραφών από ότι θέσεις στο πίνακα κατακερματισμού. Αν για δύο ή περισσότερα κλειδιά η εφαρμογή της συνάρτησης κατακερματισμού δίνει το ίδιο αποτέλεσμα τότε λέμε ότι συμβαίνει σύγκρουση (collision) η οποία θα πρέπει να διευθετηθεί με κάποιο τρόπο. Ειδικότερα, η εύρεση μιας εγγραφής με κλειδί k είναι μια διαδικασία δύο βημάτων:

- Εφαρμογή της συνάρτησης κατακερματισμού στο κλειδί της εγγραφής.
- Ξεκινώντας από την θέση που υποδεικνύει η συνάρτηση κατακερματισμού στον πίνακα κατακερματισμού, εντοπισμός της εγγραφής που περιέχει το ζητούμενο κλειδί (ενδεχόμενα θα χρειαστεί να εφαρμοστεί κάποιος μηχανισμός διευθέτησης συγκρούσεων).

2.1 Ανοικτή διευθυνσιοδότηση

2.2 Κατακερματισμός με αλυσίδες

3 Κατακερματισμός με την STL

Η STL διαθέτει την κλάση `std::hash` που μπορεί να χρησιμοποιηθεί για την επιστροφή hash τιμών για διάφορους τύπους δεδομένων. Στον ακόλουθο κώδικα παρουσιάζεται η χρήση της `std::hash`.

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main() {
6     double d1 = 1000.1;
7     double d2 = 1000.2;
8     hash<double> d_hash;
9     cout << "The hash value for: (" << d1 << ") is " << d_hash(d1) << endl;
10    cout << "The hash value for: (" << d2 << ") is " << d_hash(d2) << endl;
11
12    char c1[15] = "This is a test";
13    char c2[16] = "This is a test.";
14    hash<char*> c_strhash;
15    cout << "The hash value for: (" << c1 << ") is " << c_strhash(c1) << endl;
16    cout << "The hash value for: (" << c2 << ") is " << c_strhash(c2) << endl;
17
18    string s1 = "This is a test";
19    string s2 = "This is a test.";
20    hash<string> strhash;
21    cout << "The hash value for: (" << s1 << ") is " << strhash(s1) << endl;
22    cout << "The hash value for: (" << s2 << ") is " << strhash(s2) << endl;
23 }
```

Κώδικας 1: Παράδειγμα χρήσης της `std::hash` (`stl_hash.cpp`)

```

1 The hash value for: (1000.1) is 18248755989755706217
2 The hash value for: (1000.2) is 2007414553616229599
3 The hash value for: (This is a test) is 7536144
4 The hash value for: (This is a test.) is 7536128
5 The hash value for: (This is a test) is 5122661464562453635
6 The hash value for: (This is a test.) is 10912006877877170250
```

Επιπλέον, η STL υποστηρίζει δύο βασικές δομές κατακερματισμού το `std::unordered_set` και το `std::unordered_map`. Το `std::unordered_set` υλοποιείται ως ένας πίνακας κατακερματισμού και μπορεί να περιέχει τιμές (κλειδιά) οποιουδήποτε τύπου οι οποίες γίνονται hash σε διάφορες θέσεις του πίνακα κατακερματισμού. Κατά μέσο όρο,

οι λειτουργίες σε ένα `std::unordered_set` (εύρεση, εισαγωγή και διαγραφή κλειδιού) πραγματοποιούνται σε σταθερό χρόνο $O(1)$. Ένα `std::unordered_set` δεν περιέχει διπλότυπα, ενώ αν υπάρχει αυτή η ανάγκη τότε μπορεί να χρησιμοποιηθεί το `std::unordered_multiset`.

Στον κώδικα που ακολουθεί οι χαρακτήρες ενός λεκτικού εισάγονται ένας προς ένας σε ένα `std::unordered_set` έτσι ώστε να υπολογιστεί το πλήθος των διακριτών χαρακτήρων του λεκτικού.

```

1 #include <cctype> // tolower
2 #include <iostream>
3 #include <unordered_set>
4
5 using namespace std;
6
7 int main() {
8     string text = "You can do anything but not everything";
9     unordered_set<char> uset;
10    for (char c : text)
11        if (c != ' ')
12            uset.insert(tolower(c));
13    cout << "Number of discrete characters=" << uset.size() << endl;
14    for (unordered_set<char>::iterator itr = uset.begin(); itr != uset.end();
15         itr++)
16        cout << *itr << " ";
17    cout << endl;
18 }
```

Κώδικας 2: Παράδειγμα χρήσης του `std::unordered_set` (stl_unordered_set.cpp)

```

1 Number of discrete characters=15
2 r v e g c b y n u o d a t i h
```

Το `std::unordered_map` αποθηκεύει ζεύγη (κλειδί-τιμή). Το κλειδί αναγνωρίζει με μοναδικό τρόπο το κάθε ζεύγος και γίνεται hash σε συγκεκριμένη θέση του πίνακα κατακερματισμού. Όπως και στο `std::unordered_set`, κατά μέσο όρο, οι λειτουργίες σε ένα `std::unordered_map` πραγματοποιούνται σε σταθερό χρόνο $O(1)$. Η ανάθεση τιμής σε κλειδί μπορεί να γίνει με τους τελεστές `=` και `[]` ενώ το πέρασμα από τις τιμές ενός `std::unordered_map` μπορεί να γίνει με `iterator` ή με `range for`.

```

1 #include <iostream>
2 #include <unordered_map>
3
4 using namespace std;
5
6 int main() {
7     unordered_map<string, double> atomic_mass{{"H", 1.008}, // Hydrogen
8                                                {"C", 12.011}}; // Carbon
9     atomic_mass["O"] = 15.999; // Oxygen
10    atomic_mass["Fe"] = 55.845; // Iron
11    atomic_mass.insert(make_pair("Al", 26.982)); // Aluminium
12
13    for (unordered_map<string, double>::iterator itr = atomic_mass.begin();
14         itr != atomic_mass.end(); itr++)
15        cout << itr->first << " " << itr->second << " ";
16    cout << endl;
17
18    for (const std::pair<string, double> &kv : atomic_mass)
19        cout << kv.first << " " << kv.second << " ";
20    cout << endl;
21
22    string element = "Fe";
23    // string element = "Ti"; // Titanium
24    if (atomic_mass.find(element) == atomic_mass.end())
```

```

25     cout << "Element " << element << " is not in the map" << endl;
26 else
27     cout << "Element " << element << " has atomic mass " << atomic_mass[element]
28     << "" << endl;
29 }

```

Κώδικας 3: Παράδειγμα χρήσης του std::unordered_map (stl_unordered_map.cpp)

```

1 Al:26.982 H:1.008 C:12.011 O:15.999 Fe:55.845
2 Al:26.982 H:1.008 C:12.011 O:15.999 Fe:55.845
3 Element Fe has atomic mass 55.845

```

4 Κατακερματισμός και κρυπτογράφηση

5 Bloom filters

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a "counting" filter); the more elements that are added to the set, the larger the probability of false positives.

6 Παραδείγματα

6.1 Παράδειγμα 1

Έστω μια επιχείρηση η οποία επιθυμεί να αποθηκεύσει τα στοιχεία των υπαλλήλων της (όνομα, διεύθυνση) σε μια δομή έτσι ώστε με βάση το όνομα του υπαλλήλου να επιτυγχάνει τη γρήγορη ανάκληση των υπόλοιπων στοιχείων των υπαλλήλων. Στη συνέχεια παρουσιάζεται η υλοποίηση ενός πίνακα κατακερματισμού στον οποίο κλειδί θεωρείται το όνομα του υπαλλήλου και η επίλυση των συγκρούσεων πραγματοποιείται με ανοικτή διευθυνσιοδότηση (open addressing) και γραμμική αναζήτηση (linear probing). Ο πίνακας κατακερματισμού μπορεί να δεχθεί το πολύ 10.000 εγγραφές υπαλλήλων. Στο παράδειγμα χρονομετρείται η εκτέλεση για 2.000, 3.000 και 8.000 υπαλλήλους. Παρατηρείται ότι λόγω των συγκρούσεων καθώς ο συντελεστής φόρτωσης του πίνακα κατακερματισμού αυξάνεται η απόδοση της δομής υποβαθμίζεται.

```

1 #include <random>
2 using namespace std;
3
4 mt19937 mt(1821);
5 uniform_int_distribution<int> uni(0, 25);
6
7 struct employee {
8     string name;
9     string address;
10 };
11
12 string generate_random_string(int k) {
13     string s{};
14     const string letters_en = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
15     for (int i = 0; i < k; i++)
16         s += letters_en[uni(mt)];
17     return s;
18 }

```

Κώδικας 4: Ορισμός δομής employee και δημιουργία τυχαίων λεκτικών (employees.cpp)

```

1 #include "employees.cpp"
2 #include <chrono>
3 #include <iomanip>
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
8 using namespace std::chrono;
9
10 const int N = 10000; // HashTable size
11
12 int simple_string_hash(string &s) {
13     int h = 0;
14     for (char c : s)
15         h += c;
16     return h % N;
17 }
18
19 void insert(employee hash_table[], employee &ypa) {
20     int pos = simple_string_hash(ypa.name);
21     while (hash_table[pos].name != "") {
22         pos++;
23         pos %= N;
24     }
25     hash_table[pos] = ypa;
26 }
27
28 bool search(employee hash_table[], string &name, employee &ypa) {
29     int pos = simple_string_hash(name);
30     int c = 0;
31     while (hash_table[pos].name != name) {
32         if (hash_table[pos].name == "")
33             return false;
34         pos++;
35         pos %= N;
36         c++;
37         if (c > N)
38             return false;
39     }
40     ypa = hash_table[pos];
41     return true;
42 }
43
44 int main() {
45     vector<int> SIZES{2000, 3000, 8000};
46     for (int x : SIZES) {
47         struct employee hash_table[N];
48         // generate x random employees, insert them at the hashtable
49         vector<string> names;
50         for (int i = 0; i < x; i++) {
51             employee ypa;
52             ypa.name = generate_random_string(3);
53             ypa.address = generate_random_string(20);
54             insert(hash_table, ypa);
55             names.push_back(ypa.name);
56         }
57         // generate x more names
58         for (int i = 0; i < x; i++)
59             names.push_back(generate_random_string(3));
60         // time execution of 2*x searches in the HashTable

```

```

61 auto t1 = high_resolution_clock::now();
62 employee ypa;
63 int c = 0;
64 for (string name : names)
65     if (search(hash_table, name, ypa)) {
66         // cout << "Employee " << ypa.name << " " << ypa.address << endl;
67         c++;
68     }
69 auto t2 = high_resolution_clock::now();
70 std:
71 chrono::duration<double, std::micro> duration = t2 - t1;
72 cout << "Load factor: " << setprecision(2) << (double)x / (double)N
73     << " employees found: " << c << " employees not found: " << 2 * x - c
74     << " time elapsed: " << std::fixed << duration.count()
75     << " microseconds" << endl;
76 }
77 }

```

Κώδικας 5: Υλοποίηση πίνακα κατακερματισμού για γρήγορη αποθήκευση και αναζήτηση εγγραφών (lab07_ex1.cpp)

```

1 Load factor: 0.2 employees found: 2000 employees not found: 2000 time elapsed: 141452.50 microseconds
2 Load factor: 0.30 employees found: 3000 employees not found: 3000 time elapsed: 316067.20 microseconds
3 Load factor: 0.80 employees found: 8000 employees not found: 8000 time elapsed: 2270363.00 microseconds

```

6.2 Παράδειγμα 2

Στο παράδειγμα αυτό παρουσιάζεται η λύση του ίδιου προβλήματος με το παράδειγμα 1 με τη διαφορά ότι πλέον χρησιμοποιείται η δομή `std::unordered_map` της STL.

```

1 #include "employees.cpp"
2 #include <chrono>
3 #include <iomanip>
4 #include <iostream>
5 #include <string>
6 #include <unordered_map>
7 #include <vector>
8
9 using namespace std::chrono;
10
11 int main() {
12     vector<int> SIZES{2000, 3000, 8000};
13     for (int x : SIZES) {
14         unordered_map<string, employee> umap;
15         // generate x random employees, insert them at the hashtable
16         vector<string> names;
17         for (int i = 0; i < x; i++) {
18             employee ypa;
19             ypa.name = generate_random_string(3);
20             ypa.address = generate_random_string(3);
21             umap[ypa.name] = ypa;
22             names.push_back(ypa.name);
23         }
24         // generate x more names
25         for (int i = 0; i < x; i++)
26             names.push_back(generate_random_string(10));
27
28         // time execution of 2 * E searches in the HashTable
29         auto t1 = high_resolution_clock::now();
30         int c = 0;

```

```

31 for (string name : names)
32     if (umap.find(name) != umap.end()) {
33         // cout << "Employee " << name << " " << umap[name].address << endl;
34         c++;
35     }
36     auto t2 = high_resolution_clock::now();
37     std:
38     chrono::duration<double, std::micro> duration = t2 - t1;
39     cout << "Load factor: " << setprecision(2) << umap.load_factor()
40         << " employees found: " << c << " employees not found: " << 2 * x - c
41         << " time elapsed: " << std::fixed << duration.count()
42         << " microseconds" << endl;
43 }
44 }

```

Κώδικας 6: Γρήγορη αποθήκευση και αναζήτηση εγγράφων με τη χρήση της `std::unordered_map` (lab07_ex2.cpp)

```

1 Load factor: 0.53 employees found: 2000 employees not found: 2000 time elapsed: 0.00 microseconds
2 Load factor: 0.80 employees found: 3000 employees not found: 3000 time elapsed: 0.00 microseconds
3 Load factor: 0.53 employees found: 8000 employees not found: 8000 time elapsed: 0.00 microseconds

```

6.3 Παράδειγμα 3

Στο παράδειγμα αυτό εξετάζονται πέντε διαφορετικοί τρόποι με τους οποίους ελέγχεται για ένα μεγάλο πλήθος τιμών (5.000.000) πόσες από αυτές δεν περιέχονται σε ένα δεδομένο σύνολο 1.000 τιμών. Οι τιμές είναι ακέραιες και επιλέγονται με τυχαίο τρόπο στο διάστημα [0,100.000]. Ο χρόνος που απαιτεί η κάθε προσέγγιση χρονομετρείται.

- Η πρώτη προσέγγιση (scenario1) χρησιμοποιεί ένα vector για να αποθηκεύσει το σύνολο των 1.000 τυχαίων ακεραίων τιμών και αναζητά σειριακά κάθε τιμή στο vector.
- Η δεύτερη προσέγγιση (scenario2) χρησιμοποιεί επίσης ένα vector για να αποθηκεύσει το σύνολο των 1.000 τυχαίων ακεραίων τιμών, τις ταξινομεί και αναζητά κάθε τιμή στο ταξινομημένο vector.
- Η τρίτη προσέγγιση (scenario3) αποθηκεύει τις 1.000 τυχαίες ακεραίες τιμές σε ένα `std::set` (υλοποιείται στην STL ως δυαδικό δένδρο αναζήτησης) και αναζητά κάθε τιμή σε αυτό.
- Η τέταρτη προσέγγιση (scenario4) αποθηκεύει τις 1.000 τυχαίες ακεραίες τιμές σε ένα `std::unordered_set` (υλοποιείται στην STL ως πίνακας κατακερματισμού) και αναζητά κάθε τιμή σε αυτό.
- Η πέμπτη προσέγγιση (scenario5) υλοποιεί ένα Bloom filter που χρησιμοποιεί ως βασική δομή ένα σύνολο δυαδικών ψηφίων με μέγεθος 10.001.

```

1 #include <algorithm>
2 #include <bitset>
3 #include <chrono>
4 #include <iostream>
5 #include <random>
6 #include <set>
7 #include <unordered_set>
8 #include <vector>
9
10 using namespace std;
11 using namespace std::chrono;
12
13 // number of items in the set
14 constexpr int N = 1000;
15 // number of values checked whether they exist in the set
16 constexpr int M = 5E6;
17 // number of bits of the bitmap

```

```

18 constexpr int BS_SIZE = 10001;
19 uniform_int_distribution<uint32_t> dist(0, 1E5);
20
21 void scenario1(vector<uint32_t> &avector) {
22     long seed = 1940;
23     mt19937 mt(seed);
24     int c = 0;
25     for (int i = 0; i < M; i++)
26         if (find(avector.begin(), avector.end(), dist(mt)) == avector.end())
27             c++;
28     cout << "Values not in the set (using unsorted vector): " << c << " ";
29 }
30
31 void scenario2(vector<uint32_t> &avector) {
32     sort(avector.begin(), avector.end());
33     long seed = 1940;
34     mt19937 mt(seed);
35     int c = 0;
36     for (int i = 0; i < M; i++)
37         if (!binary_search(avector.begin(), avector.end(), dist(mt)))
38             c++;
39     cout << "Values not in the set (using sorted vector): " << c << " ";
40 }
41
42 void scenario3(set<uint32_t> &aset) {
43     long seed = 1940;
44     mt19937 mt(seed);
45     int c = 0;
46     for (int i = 0; i < M; i++)
47         if (aset.find(dist(mt)) == aset.end())
48             c++;
49     cout << "Values not in the set (using std::set): " << c << " ";
50 }
51
52 void scenario4(unordered_set<uint32_t> &auset) {
53     long seed = 1940;
54     mt19937 mt(seed);
55     int c = 0;
56     for (int i = 0; i < M; i++)
57         if (auset.find(dist(mt)) == auset.end())
58             c++;
59     cout << "Values not in the set (using std::unordered_set): " << c << " ";
60 }
61
62 inline uint32_t hash1(uint32_t x) { return x; }
63 inline uint32_t hash2(uint32_t x) { return 3 * x; }
64 inline uint32_t hash3(uint32_t x) { return 5 * x + 7; }
65
66 void scenario5(bitset<BS_SIZE> &bs, unordered_set<uint32_t> &aset) {
67     long seed = 1940;
68     mt19937 mt(seed);
69     int c = 0;
70     for (int i = 0; i < M; i++) {
71         uint32_t x = dist(mt), h1 = hash1(x), h2 = hash2(x), h3 = hash3(x);
72         if (!bs.test(h1 % BS_SIZE) || !bs.test(h2 % BS_SIZE) ||
73             !bs.test(h3 % BS_SIZE) || aset.find(x) == aset.end())
74             c++;
75     }
76     cout << "Values not in the set (using bloom filter + std::unordered_set): "
77         << c << " ";
78 }

```



```

79
80 int main() {
81     long seed = 1821;
82     mt19937 mt(seed);
83     high_resolution_clock::time_point t1, t2;
84     duration<double, std::micro> duration_micro;
85     vector<uint32_t> avector(N);
86     // fill vector with random values using std::generate and lambda function
87     std::generate(avector.begin(), avector.end(), [&mt]() { return dist(mt); });
88
89     t1 = high_resolution_clock::now();
90     scenario1(avector);
91     t2 = high_resolution_clock::now();
92     duration_micro = t2 - t1;
93     cout << "elapsed time: " << duration_micro.count() << " microseconds, "
94           << duration_micro.count() / 1E6 << " seconds" << endl;
95
96     t1 = high_resolution_clock::now();
97     scenario2(avector);
98     t2 = high_resolution_clock::now();
99     duration_micro = t2 - t1;
100    cout << "elapsed time: " << duration_micro.count() << " microseconds, "
101          << duration_micro.count() / 1E6 << " seconds" << endl;
102
103    set<uint32_t> aset(avector.begin(), avector.end());
104    t1 = high_resolution_clock::now();
105    scenario3(aset);
106    t2 = high_resolution_clock::now();
107    duration_micro = t2 - t1;
108    cout << "elapsed time: " << duration_micro.count() << " microseconds, "
109          << duration_micro.count() / 1E6 << " seconds" << endl;
110
111    unordered_set<uint32_t> auset(avector.begin(), avector.end());
112    t1 = high_resolution_clock::now();
113    scenario4(aset);
114    t2 = high_resolution_clock::now();
115    duration_micro = t2 - t1;
116    cout << "elapsed time: " << duration_micro.count() << " microseconds, "
117          << duration_micro.count() / 1E6 << " seconds" << endl;
118
119    bitset<BS_SIZE> bs;
120    for (int x : avector) {
121        bs.set(hash1(x) % BS_SIZE, true);
122        bs.set(hash2(x) % BS_SIZE, true);
123        bs.set(hash3(x) % BS_SIZE, true);
124    }
125    t1 = high_resolution_clock::now();
126    scenario5(bs, auset);
127    t2 = high_resolution_clock::now();
128    duration_micro = t2 - t1;
129    cout << "elapsed time: " << duration_micro.count() << " microseconds, "
130          << duration_micro.count() / 1E6 << " seconds" << endl;
131 }

```

Κώδικας 7: Έλεγχος ύπαρξης τιμών σε ένα σύνολο τιμών (lab07_ex3.cpp)

-
- 1 Values not in the set (using unsorted vector): 4994953 elapsed time: 1.9185e+007 microseconds, 19.185 seconds
 - 2 Values not in the set (using sorted vector): 4994953 elapsed time: 1.02876e+006 microseconds, 1.02876 seconds
 - 3 Values not in the set (using std::set): 4994953 elapsed time: 1.09391e+006 microseconds, 1.09391 seconds
 - 4 Values not in the set (using std::unordered_set): 4994953 elapsed time: 598562 microseconds, 0.598562 seconds
 - 5 Values not in the set (using bloom filter + std::unordered_set): 4994953 elapsed time: 297832 microseconds, 0.297832 seconds
-

7 Ασκήσεις

1. Γράψτε μια συνάρτηση που να δέχεται έναν πίνακα ακεραίων και έναν ακέραιο αριθμό sum και να βρίσκει το πλήθος από όλα τα ζεύγη τιμών του πίνακα που το άθροισμά τους είναι ίσο με sum.
2. β

Αναφορές