Δομές Δεδομένων και Αλγόριθμοι - Εργαστήριο 3
Αλγόριθμοι αναζήτησης και ταξινόμησης, ασυμπτωτική πολυπλοκότητα, χρονομέτρηση κώδικα

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

1 Εισαγωγή

Στο εργαστήριο αυτό παρουσιάζονται ορισμένοι αλγόριθμοι ταξινόμησης και ορισμένοι αλγόριθμοι αναζήτησης. Πρόκειται για μερικούς από τους σημαντικότερους αλγορίθμους στην επιστήμη των υπολογιστών. Σε πρακτικό επίπεδο ένα σημαντικό ποσοστό της επεξεργαστικής ισχύος των υπολογιστών δαπανάται στην ταξινόμηση δεδομένων η οποία διευκολύνει τις αναζητήσεις που ακολουθούν. Επιπλέον, γίνεται αναφορά στη θεωρητική εκτίμηση της απόδοσης ενός αλγορίθμου και στη μέτρηση χρόνου εκτέλεσης κώδικα.

2 Θεωρητική και εμπειρική εκτίμηση της απόδοσης αλγορίθμων

Συχνά χρειάζεται να εκτιμηθεί η καταλληλότητα ενός αλγορίθμου για την επίλυση ενός προβλήματος. Ποιοτικά χαρακτηριστικά όπως ο χρόνος εκτέλεσης και ο χώρος που απαιτεί στη μνήμη και στο δίσκο μπορεί να τον καθιστούν υποδεέστερο άλλων αλγορίθμων ή ακόμα και ακατάλληλο για επίλυση του προβλήματος. Γενικά, η απόδοση ενός αλγορίθμου μπορεί να εκτιμηθεί θεωρητικά και εμπειρικά.

2.1 Θεωρητική μελέτη αλγορίθμων

Η θεωρητική μελέτη προσδιορίζει την ασυμπτωτική συμπεριφορά του αλγορίθμου, δηλαδή πως θα συμπεριφέρεται ο αλγόριθμος καθώς τα δεδομένα εισόδου αυξάνονται σε μέγεθος προσεγγίζοντας μεγάλες τιμές. Μελετώντας θεωρητικά διαφορετικούς αλγορίθμους που επιτελούν το ίδιο έργο μπορεί να πραγματοποιηθεί σύγκριση μεταξύ τους ακόμα και χωρίς να γραφεί κώδικας που να τους υλοποιεί σε μια συγκεκριμένη γλώσσα προγραμματισμού. Η μέθοδος που έχει επικρατήσει για τη θεωρητική μελέτη αλγορίθμων είναι η ασυμπτωτική ανάλυση και ιδιαίτερα ο συμβολισμός του μεγάλου Ο (Big O notation). Ο συμβολισμός του μεγάλου Ο περιγράφει τη χειρότερη περίπτωση εκτέλεσης και συνήθως αφορά το χρόνο εκτέλεσης ή σπανιότερα το χώρο που απαιτείται από τον αλγόριθμο. Στη συνέχεια θα επιχειρηθεί μια πρακτική παρουσίαση του εν λόγω συμβολισμού μέσω παραδειγμάτων.

2.1.1 $O(1)$

Ο συμβολισμός $O(1)$ περιγράφει αλγορίθμους που πάντα εκτελούνται απαιτώντας τον ίδιο χρόνο (ή χώρο), άσχετα με το μέγεθος των δεδομένων με τα οποία τροφοδοτούνται. Ο ακόλουθος κώδικας που είναι ένα παράδειγμα αλγορίθμου με ασυμπτωτική πολυπλοκότητα $O(1)$ επιστρέφει true αν το πρώτο στοιχείο ενός πίνακα ακεραίων είναι άρτιο, αλλιώς επιστρέφει false.

```
1 bool is_first_element_even(int a[]) {
2     return a[0] % 2 == 0;
3 }
```

2.1.2 $O(n)$

Ο συμβολισμός $O(n)$ περιγράφει αλγορίθμους που ο χρόνος εκτέλεσής τους μεγαλώνει γραμμικά και σε ευθεία αναλογία με το μέγεθος της εισόδου. Ο κώδικας που ακολουθεί επιστρέφει true αν το στοιχείο key υπάρχει στον πίνακα a, N θέσεων. Η ασυμπτωτική του πολυπλοκότητα είναι $O(n)$.

```
bool exists(int a[], int N, int key) {
  for (int i = 0; i < N; i++)
    if (a[i] == key)
      return true;
  return false;
}
```

2.1.3 $O(n^2)$

Ο συμβολισμός $O(n^2)$ περιγράφει αλγορίθμους που ο χρόνος εκτέλεσής τους μεγαλώνει ανάλογα με το τετράγωνο του μεγέθους της εισόδου. Αυτό τυπικά συμβαίνει όταν ο κώδικας περιέχει δύο εντολές επανάληψης την μια μέσα στην άλλη. Ο ακόλουθος κώδικας εξετάζει αν ένας πίνακας έχει διπλότυπα και έχει ασυμπτωτική πολυπλοκότητα $O(n^2)$.

```
bool has_duplicates(int a[], int N) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
      if (i == j)
        continue;
      if (a[i] == a[j])
        return true;
    }
  return false;
}
```

2.1.4 $O(2^n)$

Το $O(2^n)$ αφορά αλγορίθμους που ο χρόνος εκτέλεσής τους διπλασιάζεται για κάθε μονάδα αύξησης των δεδομένων εισόδου. Η αύξηση είναι εξαιρετικά απότομη καθιστώντας τον αλγόριθμο μη χρησιμοποιήσιμο παρά μόνο για μικρές τιμές του N. Ένα παράδειγμα αλγορίθμου με ασυμπτωτική πολυπλοκότητα $O(2^n)$ είναι ο αναδρομικός υπολογισμός των αριθμών Fibonacci.

```
int fibo(int n) {
  if (n <= 1)
    return n;
  else
    return fibo(n − 2) + fibo(n − 1);
}
```

2.1.5 $O(log(n))$

Το $O(log(n))$ περιγράφει αλγορίθμους στους οποίους σε κάθε βήμα τους το μέγεθος των δεδομένων που μένει να εξετάσει ο αλγόριθμος μειώνεται στο μισό. Ένα παράδειγμα αλγορίθμου με ασυμπτωτική πολυπλοκότητα $O(log(n))$ είναι ο ακόλουθος κώδικας που επιστρέφει λογική τιμή σχετικά με το εάν υπάρχει το στοιχείο key στον ταξινομημένο πίνακα a, N θέσεων.

```
bool exists_in_sorted(int a[], int N, int key) {
  int left = 0, right = N − 1, m;
  while (left <= right) {
    m = (left + right) / 2;
```

```
    if (a[m] == key)
      return true;
    else if (a[m] > key)
      right = m − 1;
    else
      left = m + 1;
  }
  return false;
}
```

Στη συνέχεια παρατίθενται ασυμπτωτικές πολυπλοκότητες αλγορίθμων από τις ταχύτερες προς τις βραδύτερες: $O(1)$, $O(log(n))$, $O(n)$,$O(nlog(n))$, $O(n^2)$, $O(n^3)$, $O(n^4)$, $O(2^n)$, $O(n!)$. Με εξαιρέσεις, επιθυμητές πολυπλοκότητες αλγορίθμων είναι μέχρι και $O(n^2)$.

2.2 Εμπειρική μελέτη αλγορίθμων

Η εμπειρική εκτίμηση της απόδοσης ενός προγράμματος έχει να κάνει με τη χρονομέτρησή του για διάφορες περιπτώσεις δεδομένων εισόδου και τη σύγκρισή του με εναλλακτικές υλοποιήσεις προγραμμάτων. Στη συνέχεια θα παρουσιαστούν δύο τρόποι μέτρησης χρόνου εκτέλεσης κώδικα που μπορούν να εφαρμοστούν στη C++.

2.2.1 Μέτρηση χρόνου εκτέλεσης κώδικα με τη συνάρτηση clock()

Ο ακόλουθος κώδικας μετράει το χρόνο που απαιτεί ο υπολογισμός του αθροίσματος των τετραγωνικών ριζών 10.000.000 τυχαίων ακέραιων αριθμών με τιμές στο διάστημα από 0 έως 10.000. Η μέτρηση του χρόνου πραγματοποιείται με τη συνάρτηση clock() η οποία επιστρέφει τον αριθμό από clock ticks που έχουν περάσει από τη στιγμή που το πρόγραμμα ξεκίνησε την εκτέλεση του. Ο αριθμός των δευτερολέπτων που έχουν περάσει προκύπτει διαιρώντας τον αριθμό των clock ticks με τη σταθερά CLOCKS_PER_SEC. Αυτός ο τρόπος υπολογισμού του χρόνου εκτέλεσης έχει “κληρονομηθεί” στη C++ από τη C.

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  clock_t t1, t2;
  t1 = clock();
  srand(1821);
  double sum = 0.0;
  for (int i = 1; i <= 10000000; i++) {
    int x = rand() % 10000 + 1;
    sum += sqrt(x);
  }
  cout << ”The sum is: ” << sum << endl;

  t2 = clock();
  double elapsed_time = (double)(t2 − t1) / CLOCKS_PER_SEC;
  cout << ”Elapsed time ” << elapsed_time << ” seconds” << endl;
}
```

Κώδικας 1: Μέτρηση χρόνου εκτέλεσης κώδικα(timing1.cpp)

```
The sum is: 6.39952e+008
Elapsed time 0.213
```

2.2.2 Μέτρηση χρόνου εκτέλεσης κώδικα με τη χρήση του high_resolution_clock::time_point

Η C++ έχει προσθέσει νέους τρόπους μέτρησης του χρόνου εκτέλεσης προγραμμάτων. Στον ακόλουθο κώδικα παρουσιάζεται ένα παράδειγμα με χρήση time_points.

```
#include <chrono>
#include <cmath>
#include <iostream>
#include <random>

using namespace std;
using namespace std::chrono;

int main() {
  high_resolution_clock::time_point t1 = high_resolution_clock::now();
  mt19937 mt(1821);
  uniform_int_distribution<int> dist(0, 10000);
  double sum = 0.0;
  for (int i = 1; i <= 10000000; i++) {
    int x = dist(mt);
    sum += sqrt(x);
  }
  cout << "The sum is: " << sum << endl;
  high_resolution_clock::time_point t2 = high_resolution_clock::now();
  auto duration = duration_cast<microseconds>(t2 - t1).count();
  cout << "Time elapsed: " << duration << " microseconds "
       << duration / 1000000.0 << " seconds" << endl;
}
```

Κώδικας 2: Μέτρηση χρόνου εκτέλεσης κώδικα (timing2.cpp)

```
The sum is: 6.6666e+008
Time elapsed: 537030 microseconds 0.53703 seconds
```

Θα πρέπει να σημειωθεί ότι κατά τη μεταγλώττιση είναι δυνατό να δοθεί οδηγία προς το μεταγλωττιστή έτσι ώστε να προχωρήσει σε βελτιστοποιήσεις του κώδικα που παράγει που θα οδηγήσουν σε ταχύτερη εκτέλεση. Το flag που χρησιμοποιείται είναι το -O και οι πιθανές τιμές που μπορεί να λάβει είναι: -O0, -O1, -O2, -O3. Καθώς αυξάνεται ο αριθμός δεξιά του -O που χρησιμοποιείται στη μεταγλώττιση ενεργοποιούνται περισσότερες βελτιστοποιήσεις σε βάρος του χρόνου μεταγλώττισης. Στη συνέχεια παρουσιάζονται οι εντολές μεταγλώττισης και ο χρόνος εκτέλεσης για κάθε μια από τις 4 περιπτώσεις του κώδικα .

```
g++ timing2.cpp -o timing2a -O0 -std=c++11
./timing2a
The sum is: 6.6666e+008
Time elapsed: 537030 microseconds 0.53703 seconds

g++ timing2.cpp -o timing2b -O1 -std=c++11
./timing2b
The sum is: 6.6666e+008
Time elapsed: 125007 microseconds 0.125007 seconds

g++ timing2.cpp -o timing2c -O2 -std=c++11
./timing2c
The sum is: 6.6666e+008
Time elapsed: 127007 microseconds 0.127007 seconds

g++ timing2.cpp -o timing2d -O3 -std=c++11
./timing2d
The sum is: 6.6666e+008
Time elapsed: 114006 microseconds 0.114006 seconds
```

3 Αλγόριθμοι ταξινόμησης

3.1 Ταξινόμηση με εισαγωγή

Η ταξινόμηση με εισαγωγή (insertion-sort) λειτουργεί δημιουργώντας μια ταξινομημένη λίστα στο αριστερό άκρο των δεδομένων και επαναληπτικά τοποθετεί το στοιχείο το οποίο βρίσκεται δεξιά της ταξινομημένης λίστας στη σωστή θέση σε σχέση με τα ήδη ταξινομημένα στοιχεία. Ο αλγόριθμος ταξινόμησης με εισαγωγή καθώς και η κλήση του από κύριο πρόγραμμα για την αύξουσα ταξινόμηση ενός πίνακα 10 θέσεων παρουσιάζεται στον κώδικα που ακολουθεί.

```
template <class T> void insertion_sort(T a[], int n) {
  for (int i = 1; i < n; i++) {
    T key = a[i];
    int j = i - 1;
    while ((j >= 0) && (key < a[j])) {
      a[j + 1] = a[j];
      j--;
    }
    a[j + 1] = key;
  }
}
```

Κώδικας 3: Ο αλγόριθμος ταξινόμησης με εισαγωγή (insertion_sort.cpp)

```
#include "insertion_sort.cpp"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
  cout << "Sort using insertion sort" << endl;
  insertion_sort(a, 10);
  for (int i = 0; i < 10; i++)
    cout << a[i] << " ";
}
```

Κώδικας 4: sort1.cpp

```
Sort using insertion sort
7 11 11 15 16 16 18 21 32 45
```

3.2 Ταξινόμηση με συγχώνευση

Η ταξινόμηση με συγχώνευση (merge-sort) είναι αναδρομικός αλγόριθμος και στηρίζεται στη συγχώνευση ταξινομημένων υποακολουθιών έτσι ώστε να δημιουργούνται νέες ταξινομημένες υποακολουθίες. Μια υλοποίηση του κώδικα ταξινόμησης με συγχώνευση παρουσιάζεται στη συνέχεια.

```
template <class T> void merge(T a[], int l, int m, int r) {
  T la[m - l + 1];
  T ra[r - m];
  for (int i = 0; i < m - l + 1; i++)
    la[i] = a[l + i];
  for (int i = 0; i < r - m; i++)
    ra[i] = a[m + 1 + i];
  int i = 0, j = 0, k = l;
  while ((i < m - l + 1) && (j < r - m)) {
    if (la[i] < ra[j]) {
      a[k] = la[i];
```

```
12          i++;
13        } else {
14          a[k] = ra[j];
15          j++;
16        }
17        k++;
18      }
19      if (i == m - l + 1) {
20        while (j < r - m) {
21          a[k] = ra[j];
22          j++;
23          k++;
24        }
25      } else {
26        while (i < m - l + 1) {
27          a[k] = la[i];
28          i++;
29          k++;
30        }
31      }
32    }
33
34    template <class T> void merge_sort(T a[], int l, int r) {
35      if (l < r) {
36        int m = (l + r) / 2;
37        merge_sort(a, l, m);
38        merge_sort(a, m + 1, r);
39        merge(a, l, m, r);
40      }
41    }
42
43    template <class T> void merge_sort(T a[], int N) { merge_sort(a, 0, N - 1); }
```

Κώδικας 5: Ο αλγόριθμος ταξινόμησης με συγχώνευση (merge_sort.cpp)

```
1  #include "merge_sort.cpp"
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char **argv) {
7    int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
8    cout << "Sort using merge sort" << endl;
9    merge_sort(a, 10);
10   for (int i = 0; i < 10; i++)
11     cout << a[i] << " ";
12 }
```

Κώδικας 6: sort2.cpp

```
1  Sort using merge sort
2  7 11 11 15 16 16 18 21 32 45
```

3.3 Γρήγορη ταξινόμηση

Ο κώδικας της γρήγορης ταξινόμησης παρουσιάζεται στη συνέχεια. Πρόκειται για κώδικα ο οποίος καλείται αναδρομικά σε υποακολουθίες των δεδομένων και σε κάθε κλήση επιλέγει ένα στοιχείο (pivot) και διαχωρίζει τα υπόλοιπα στοιχεία έτσι ώστε αριστερά να είναι τα στοιχεία που είναι μικρότερα του pivot και δεξιά αυτά τα οποία είναι μεγαλύτερα.

```
#include <utility> // std::swap

template <class T> int partition(T a[], int l, int r) {
  int p = l;
  int i = l + 1;
  for (int j = l + 1; j <= r; j++) {
    if (a[j] < a[p]) {
      std::swap(a[j], a[i]);
      i++;
    }
  }
  std::swap(a[p], a[i − 1]);
  return i − 1;
}

template <class T> void quick_sort(T a[], int l, int r) {
  if (l >= r)
    return;
  else {
    int p = partition(a, l, r);
    quick_sort(a, l, p − 1);
    quick_sort(a, p + 1, r);
  }
}

template <class T> void quick_sort(T a[], int N) { quick_sort(a, 0, N − 1); }
```

Κώδικας 7: Ο αλγόριθμος γρήγορης ταξινόμησης (quick_sort.cpp)

```
#include "quick_sort.cpp"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
  cout << "Sort using quick sort" << endl;
  quick_sort(a, 10);
  for (int i = 0; i < 10; i++)
    cout << a[i] << " ";
}
```

Κώδικας 8: sort3.cpp

```
Sort using quick sort
7 11 11 15 16 16 18 21 32 45
```

3.4 Ταξινόμηση κατάταξης

ο αλγόριθμος ταξινόμησης κατάταξης (rank-sort) λειτουργεί ως εξής: Για κάθε στοιχείο του δεδομένου πίνακα a που επιθυμούμε να ταξινομήσουμε υπολογίζεται μια τιμή κατάταξης (rank). Η τιμή κατάταξης ενός στοιχείου του πίνακα είναι το πλήθος των μικρότερων από αυτό στοιχείων συν το πλήθος των ίσων με αυτό στοιχείων που έχουν μικρότερο δείκτη σε σχέση με αυτό το στοιχείο (δηλαδή βρίσκονται αριστερά του). Δηλαδή ισχύει ότι η τιμή κατάταξης ενός στοιχείου x του πίνακα είναι ίση με το άθροισμα 2 όρων: του πλήθους των μικρότερων στοιχείων του x από όλο τον πίνακα και του πλήθους των ίσων με το x στοιχείων που έχουν μικρότερο δείκτη σε σχέση με το x. Για παράδειγμα στην ακολουθία τιμών a=[44, 21, 78, 16, 56, 21] θα πρέπει να δημιουργηθεί ένας νέος πίνακας r = [3, 1, 5, 0, 4, 2]. Έχοντας υπολογίσει τον πίνακα r θα πρέπει τα στοιχεία

του a να αντιγραφούν σε ένα νέο βοηθητικό πίνακα temp έτσι ώστε κάθε τιμή που υπάρχει στον πίνακα r να λειτουργεί ως δείκτης για το που πρέπει να τοποθετηθεί το αντίστοιχο στοιχείο του a στον πίνακα temp. Τέλος θα πρέπει να αντιγραφεί ο πίνακας temp στον πίνακα a. Στη συνέχεια παρουσιάζεται ο κώδικας του αλγορίθμου rank-sort. Παρουσιάζονται δύο υλοποιήσεις. Η πρώτη υλοποίηση (rank_sort) αφορά τον αλγόριθμο όπως έχει περιγραφεί παραπάνω ενώ η δεύτερη (rank_sort_in_place) είναι από το βιβλίο "Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++ του Sartaj Sahnii, Εκδόσεις Τζιόλα, 2004" στη σελίδα 63 (πρόγραμμα 2.11) και δεν απαιτεί τη χρήση του βοηθητικού πίνακα temp, συνεπώς είναι αποδοτικότερος.

```
#include <iostream>
#include <utility> // swap

using namespace std;

template <class T> void rank_sort(T a[], int n) {
   int r[n] = {0};
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         if (a[j] < a[i] || (a[j] == a[i] && j < i))
            r[i]++;
   int temp[n];
   for (int i = 0; i < n; i++)
      temp[r[i]] = a[i];
   for (int i = 0; i < n; i++)
      a[i] = temp[i];
}

template <class T> void rank_sort_in_place(T a[], int n) {
   int r[n] = {0};
   for (int i = 0; i < n; i++)
      for (int j = 0; j < i; j++)
         if (a[j] <= a[i])
            r[i]++;
         else
            r[j]++;
   for (int i = 0; i < n; i++)
      while (r[i] != i) {
         int t = r[i];
         swap(a[i], a[t]);
         swap(r[i], r[t]);
      }
}

int main(int argc, char **argv) {
   int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
   cout << "Sort using rank sort" << endl;
   rank_sort(a, 10);
   for (int i = 0; i < 10; i++)
      cout << a[i] << " ";
   cout << endl << "Sort using rank sort (in place)" << endl;
   int b[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
   rank_sort_in_place(b, 10);
   for (int i = 0; i < 10; i++)
      cout << b[i] << " ";
}
```

Κώδικας 9: Ο αλγόριθμος ταξινόμησης κατάταξης (sort4.cpp)

```
Sort using rank sort
7 11 11 15 16 16 18 21 32 45
Sort using rank sort (in place)
7 11 11 15 16 16 18 21 32 45
```

4 Αλγόριθμοι αναζήτησης

4.1 Σειριακή αναζήτηση

Η σειριακή αναζήτηση είναι ο απλούστερος αλγόριθμος αναζήτησης. Εξετάζει τα στοιχεία ένα προς ένα στη σειρά μέχρι να βρει το στοιχείο που αναζητείται. Το πλεονέκτημα του αλγορίθμου είναι ότι μπορεί να εφαρμοστεί σε μη ταξινομημένους πίνακες.

```
#include <iostream>

using namespace std;

template <class T> int sequential_search(T a[], int n, T key) {
  for (int i = 0; i < n; i++)
    if (a[i] == key)
      return i;
  return -1;
}

int main(int argc, char **argv) {
  int a[] = {5, 11, 45, 23, 10, 17, 32, 8, 9, 4};
  int key;
  cout << "Search for: ";
  cin >> key;
  int pos = sequential_search(a, 10, key);
  if (pos == -1)
    cout << "Not found" << endl;
  else
    cout << "Found at position " << pos << endl;
}
```

Κώδικας 10: Ο αλγόριθμος σειριακής αναζήτησης (search1.cpp)

```
Search for: 45
Found at position 2
```

4.2 Δυαδική αναζήτηση

Η δυαδική αναζήτηση μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα. Διαιρεί επαναληπτικά την ακολουθία σε 2 υποακολουθίες και απορρίπτει την ακολουθία στην οποία συμπεραίνει ότι δεν μπορεί να βρεθεί το στοιχείο.

```
template <class T> int binary_search(T a[], int l, int r, T key) {
  int m = (l + r) / 2;
  if (l > r) {
    return -1;
  } else if (a[m] == key) {
    return m;
  } else if (key < a[m]) {
    return binary_search(a, l, m - 1, key);
  } else {
    return binary_search(a, m + 1, r, key);
  }
}

template <class T> int binary_search(T a[], int n, T key) {
```

```
15   return binary_search(a, 0, n − 1, key);
16 }
```

Κώδικας 11: Ο αλγόριθμος δυαδικής αναζήτησης (binary_search.cpp)

```
1  #include ”binary_search.cpp”
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char **argv) {
7    int a[] = {11, 45, 53, 60, 67, 72, 88, 91, 94, 98};
8    int key;
9    cout << ”Search for: ”;
10   cin >> key;
11   int pos = binary_search(a, 10, key);
12   if (pos == −1)
13     cout << ”Not found” << endl;
14   else
15     cout << ”Found at position ” << pos << endl;
16 }
```

Κώδικας 12: search2.cpp

```
1  Search for: 60
2  Found at position 3
```

4.3 Αναζήτηση με παρεμβολή

Η αναζήτηση με παρεμβολή (interpolation-search) είναι μια παραλλαγή της δυαδικής αναζήτησης και μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα. Αντί να χρησιμοποιηθεί η τιμή 50% για να διαχωριστούν τα δεδομένα σε 2 ισομεγέθεις λίστες (όπως συμβαίνει στη δυαδική αναζήτηση) υπολογίζεται μια τιμή η οποία εκτιμάται ότι θα οδηγήσει πλησιέστερα στο στοιχείο που αναζητείται. Αν l είναι ο δείκτης του αριστερότερου στοιχείου της ακολουθίας και r ο δείκτης του δεξιότερου στοιχείου της ακολουθίας τότε υπολογίζεται ο συντελεστής c = (key−a[l])/(a[r]−a[l]) όπου key είναι το στοιχείο προς αναζήτηση και a είναι η ακολουθία τιμών στην οποία αναζητείται το key. Η ακολουθία των δεδομένων διαχωρίζεται με βάση τον συντελεστή c σε δύο υποακολουθίες. Η διαδικασία επαναλαμβάνεται ανάλογα με τη δυαδική αναζήτηση. Στη συνέχεια παρουσιάζεται ο κώδικας της αναζήτησης με παρεμβολή.

```
1  template <class T> int interpolation_search(T a[], int l, int r, T key) {
2    int m;
3    if (l > r) {
4      return −1;
5    } else if (l == r) {
6      m = l;
7    } else {
8      double c = (double)(key − a[l]) / (double)(a[r] − a[l]);
9      if ((c < 0) || (c > 1))
10       return −1;
11     m = (int)(l + (r − l) * c);
12   }
13   if (a[m] == key) {
14     return m;
15   } else if (key < a[m]) {
16     return interpolation_search(a, l, m − 1, key);
17   } else {
18     return interpolation_search(a, m + 1, r, key);
19   }
```

```
}

template <class T> int interpolation_search(T a[], int n, T key) {
  return interpolation_search(a, 0, n − 1, key);
}
```

Κώδικας 13: Ο αλγόριθμος αναζήτησης με παρεμβολή (interpolation_search.cpp)

```
#include ”interpolation_search.cpp”
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  int a[] = {11, 45, 53, 60, 67, 72, 88, 91, 94, 98};
  int key;
  cout << ”Search for: ”;
  cin >> key;
  int pos = interpolation_search(a, 10, key);
  if (pos == −1)
    cout << ”Not found” << endl;
  else
    cout << ”Found at position ” << pos << endl;
}
```

Κώδικας 14: search3.cpp

```
Search for: 60
Found at position 3
```

5 Παραδείγματα

5.1 Παράδειγμα 1

Γράψτε πρόγραμμα που να συγκρίνει τους χρόνους εκτέλεσης των αλγορίθμων ταξινόμησης insertion-sort, merge-sort, quick-sort καθώς και του αλγορίθμου ταξινόμησης της βιβλιοθήκης STL (συνάρτηση sort). Η σύγκριση να αφορά τυχαία δεδομένα τύπου float με τιμές στο διάστημα από -1.000 έως 1.000. Τα μεγέθη των πινάκων που θα ταξινομηθούν να είναι 5.000, 10.000, 20.0000, 40.000, 80.000, 160.0000 και 320.000 αριθμών.

```
#include ”insertion_sort.cpp”
#include ”merge_sort.cpp”
#include ”quick_sort.cpp”
#include <algorithm>
#include <chrono>
#include <iomanip>
#include <iostream>
#include <random>

using namespace std;
using namespace std::chrono;

void benchmark_sort_algorithm(string alg) {
  mt19937 mt(1821);
  uniform_real_distribution<float> dist(−1000, 1000);

  int sizes[] = {5000, 10000, 20000, 40000, 80000, 160000, 320000};
  for (int i = 0; i < 7; i++) {
```

```
19      int N = sizes[i];
20      float *a = new float[N];
21      for (int i = 0; i < N; i++)
22        a[i] = dist(mt);
23
24      auto t1 = high_resolution_clock::now();
25      if (alg.compare("merge-sort") == 0)
26        merge_sort(a, N);
27      else if (alg.compare("quick-sort") == 0)
28        quick_sort(a, N);
29      else if (alg.compare("STL-sort") == 0)
30        sort(a, a + N);
31      else if (alg.compare("insertion-sort") == 0)
32        insertion_sort(a, N);
33      auto t2 = high_resolution_clock::now();
34
35      auto elapsed_time = duration_cast<milliseconds>(t2 - t1).count();
36      cout << fixed << setprecision(3);
37      cout << "Elapsed time " << alg << "\t" << sizes[i] << "\t" << elapsed_time
38           << " milliseconds" << endl;
39      delete[] a;
40    }
41  }
42
43  int main(int argc, char **argv) {
44    benchmark_sort_algorithm("insertion-sort");
45    cout << "####################################" << endl;
46    benchmark_sort_algorithm("merge-sort");
47    cout << "####################################" << endl;
48    benchmark_sort_algorithm("quick-sort");
49    cout << "####################################" << endl;
50    benchmark_sort_algorithm("STL-sort");
51    cout << "####################################" << endl;
52  }
```

Κώδικας 15: Σύγκριση χρόνου εκτέλεσης αλγορίθμων ταξινόμησης (lab03_ex1.cpp)

```
1  Elapsed time insertion-sort 5000 46 milliseconds
2  Elapsed time insertion-sort 10000 167 milliseconds
3  Elapsed time insertion-sort 20000 658 milliseconds
4  Elapsed time insertion-sort 40000 2595 milliseconds
5  Elapsed time insertion-sort 80000 10377 milliseconds
6  Elapsed time insertion-sort 160000 41441 milliseconds
7  Elapsed time insertion-sort 320000 167593 milliseconds
8  ####################################
9  Elapsed time merge-sort 5000 1 milliseconds
10 Elapsed time merge-sort 10000 3 milliseconds
11 Elapsed time merge-sort 20000 7 milliseconds
12 Elapsed time merge-sort 40000 14 milliseconds
13 Elapsed time merge-sort 80000 32 milliseconds
14 Elapsed time merge-sort 160000 60 milliseconds
15 Elapsed time merge-sort 320000 125 milliseconds
16 ####################################
17 Elapsed time quick-sort 5000 1 milliseconds
18 Elapsed time quick-sort 10000 3 milliseconds
19 Elapsed time quick-sort 20000 5 milliseconds
20 Elapsed time quick-sort 40000 10 milliseconds
21 Elapsed time quick-sort 80000 24 milliseconds
22 Elapsed time quick-sort 160000 47 milliseconds
23 Elapsed time quick-sort 320000 105 milliseconds
24 ####################################
25 Elapsed time STL-sort 5000 1 milliseconds
26 Elapsed time STL-sort 10000 2 milliseconds
27 Elapsed time STL-sort 20000 4 milliseconds
28 Elapsed time STL-sort 40000 8 milliseconds
```

```
Elapsed time STL−sort 80000 17 milliseconds
Elapsed time STL−sort 160000 37 milliseconds
Elapsed time STL−sort 320000 79 milliseconds
#####################################
```

5.2 Παράδειγμα 2

Γράψτε πρόγραμμα που να συγκρίνει τους χρόνους εκτέλεσης των αλγορίθμων αναζήτησης binary-search, interpolation-search και του αλγορίθμου αναζήτησης της βιβλιοθήκης STL binary_search για ταξινομημένα ακέραια δεδομένα με τιμές στο διάστημα από 0 έως 10.000.000. Η σύγκριση να εξετάζει τα ακόλουθα μεγέθη πινάκων 5.000, 10.000, 20.000, 40.000, 80.000, 160.000 και 320.000 αριθμών. Οι χρόνοι εκτέλεσης να αφορούν τους συνολικούς χρόνους που απαιτούνται έτσι ώστε να αναζητηθούν 100.000 τυχαίες τιμές με καθένα από τους αλγορίθμους.

```cpp
#include "binary_search.cpp"
#include "interpolation_search.cpp"
#include <algorithm>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <random>
#include <chrono>

using namespace std;
using namespace std::chrono;

void benchmark_search_algorithm(string alg) {
  clock_t t1, t2;
  mt19937 mt(1729);
  uniform_int_distribution<int> dist(0, 1000000);
  int M = 100000;
  int keys[M];
  for (int i = 0; i < M; i++) {
    keys[i] = dist(mt);
  }
  int sizes[] = {5000, 10000, 20000, 40000, 80000, 160000, 320000};
  for (int i = 0; i < 7; i++) {
    int N = sizes[i];
    int *a = new int[N];
    for (int i = 0; i < N; i++)
      a[i] = dist(mt);
    sort(a, a + N);

    auto t1 = high_resolution_clock::now();
    int c = 0;
    if (alg.compare("binary−search") == 0) {
      for (int j = 0; j < M; j++)
        if (binary_search(a, 0, N − 1, keys[j]) != −1)
          c++;
    } else if (alg.compare("interpolation−search") == 0) {
      for (int j = 0; j < M; j++)
        if (interpolation_search(a, 0, N − 1, keys[j]) != −1)
          c++;
    } else if (alg.compare("STL−binary−search") == 0)
      for (int j = 0; j < M; j++)
        if (binary_search(a, a + N, keys[j]))
          c++;
    auto t2 = high_resolution_clock::now();

```

```
46      auto elapsed_time = duration_cast<milliseconds>(t2 − t1).count();
47      cout << fixed << setprecision(3);
48      cout << "Elapsed time " << alg << "\t" << sizes[i] << "\t" << elapsed_time
49              << " milliseconds" << endl;
50      delete[] a;
51    }
52  }
53
54  int main(int argc, char **argv) {
55    benchmark_search_algorithm("binary−search");
56    cout << "####################################" << endl;
57    benchmark_search_algorithm("interpolation−search");
58    cout << "####################################" << endl;
59    benchmark_search_algorithm("STL−binary−search");
60    cout << "####################################" << endl;
61  }
```

Κώδικας 16: Σύγκριση χρόνου εκτέλεσης αλγορίθμων ταξινόμησης (lab03_ex2.cpp)

```
1  Elapsed time binary−search 5000 20 milliseconds
2  Elapsed time binary−search 10000 20 milliseconds
3  Elapsed time binary−search 20000 22 milliseconds
4  Elapsed time binary−search 40000 24 milliseconds
5  Elapsed time binary−search 80000 31 milliseconds
6  Elapsed time binary−search 160000 30 milliseconds
7  Elapsed time binary−search 320000 35 milliseconds
8  ####################################
9  Elapsed time interpolation−search 5000 13 milliseconds
10 Elapsed time interpolation−search 10000 14 milliseconds
11 Elapsed time interpolation−search 20000 14 milliseconds
12 Elapsed time interpolation−search 40000 15 milliseconds
13 Elapsed time interpolation−search 80000 15 milliseconds
14 Elapsed time interpolation−search 160000 17 milliseconds
15 Elapsed time interpolation−search 320000 19 milliseconds
16 ####################################
17 Elapsed time STL−binary−search 5000 30 milliseconds
18 Elapsed time STL−binary−search 10000 33 milliseconds
19 Elapsed time STL−binary−search 20000 35 milliseconds
20 Elapsed time STL−binary−search 40000 39 milliseconds
21 Elapsed time STL−binary−search 80000 41 milliseconds
22 Elapsed time STL−binary−search 160000 43 milliseconds
23 Elapsed time STL−binary−search 320000 51 milliseconds
24 ####################################
```

6 Ασκήσεις

1. Ο αλγόριθμος bogosort αναδιατάσσει τυχαία τις τιμές ενός πίνακα μέχρι να προκύψει μια ταξινομημένη διάταξη. Γράψτε ένα πρόγραμμα που να υλοποιεί τον αλγόριθμο bogosort για την ταξινόμηση ενός πίνακα ακεραίων τιμών. Χρησιμοποιήστε τη συνάρτηση shuffle.

2. Να υλοποιηθεί ο αλγόριθμος ταξινόμησης με επιλογή (selection sort) και να εφαρμοστεί για τη ταξινόμηση ενός πίνακα πραγματικών τιμών, ενός πίνακα ακεραίων και ενός πίνακα με λεκτικά (δηλαδή να γίνουν τρεις κλήσεις του αλγορίθμου). Ο αλγόριθμος ταξινόμησης με επιλογή ξεκινά εντοπίζοντας το μικρότερο στοιχείο και το τοποθετεί στη πρώτη θέση. Συνεχίζει, ακολουθώντας την ίδια διαδικασία χρησιμοποιώντας το τμήμα του πίνακα που δεν έχει ταξινομηθεί ακόμα.

3. Γράψτε μια αναδρομική έκδοση του κώδικα για την ταξινόμηση με επιλογή (selection sort).