Αλγόριθμοι και Προχωρημένες Δομές Δεδομένων

Αλγοριθμικές τεχνικές

Διαίρει και Βασίλευε

Γκόγκος Χρήστος - 22/12/2023

Διάσπαση προβλήματος

- Πλεονεκτήματα διάσπασης προβλήματος σε μικρότερα προβλήματα:
  - Απλούστευση προβλήματος
  - Εστίαση σε λεπτομέρειες που δεν είναι ορατές στο πλήρες πρόβλημα
  - Ανάδυση αναδρομικού αλγορίθμου
  - Πιθανότητα χρήσης πολυπύρηνων υπολογιστικών συστημάτων
- Αλγοριθμικές τεχνικές που χρησιμοποιούν τη διάσπαση προβλήματος:
  - Διαίρει και βασίλευε (διάσπαση του προβλήματος σε «δύο μισά»)
  - Δυναμικός προγραμματισμός (διάσπαση του προβλήματος σε επικαλυπτόμενα υποπροβλήματα)

Διάιρει και Βασίλευε

- Διαίρεση του προβλήματος σε μικρότερα υποπροβλήματα, αναδρομική επίλυση των προβλημάτων, συγχώνευση των μερικών λύσεων σε μια λύση για το πλήρες πρόβλημα
  - Αν η συγχώνευση χρειάζεται λιγότερο χρόνο από την επίλυση των δύο υποπροβλημάτων τότε ο αλγόριθμος είναι αποδοτικός!
  - Παράδειγμα: Η ταξινόμηση με συγχώνευση (mergesort) χρειάζεται O(n) χρόνο για να συγχωνεύσει δύο ταξινομημένες λίστες με n/2 στοιχεία η κάθε μια, και κάθε μια ταξινομημένη λίστα χρειάζεται O(nlogn) χρόνο

Παραδείγματα προβλημάτων που αντιμετωπίζονται με «Διαίρει και Βασίλευε»

- Δυαδική αναζήτηση – $O(\log n)$
- Ταξινόμηση με συγχώνευση – $O(n \log n)$
- Αλγόριθμος του Karatsuba για πολλαπλασιασμό ακεραίων – $O(n^{1.585})$
- Αλγόριθμος του Strassen για πολλαπλασιασμό πινάκων – $O(n^{2.71})$
- Μέγιστη υποακολουθία – $O(n \log n)$
- Πλησιέστερο ζεύγος σημείων στο δισδιάστατο χώρο – $O(n \log n)$

Δυαδική αναζήτηση

- Η «μητέρα» όλων των αλγορίθμων διαίρει και βασίλευε είναι η δυαδική αναζήτηση
- Παράδειγμα εκτίμησης ταχύτητας δυαδικής αναζήτησης:
  - Παιχνίδι εντοπισμού λέξης σε λεξικό με 200000 λέξεις, ο ένας παίκτης επιλέγει μια «κρυφή» λέξη και ο δεύτερος παίκτης προσπαθεί να τη μαντέψει, ενώ για κάθε λάθος επιλογή του δεύτερου παίκτη, ο πρώτος απαντά αν η «κρυφή» λέξη είναι μικρότερη ή μεγαλύτερη (λεξικογραφικά) από τη λέξη του δεύτερου παίκτη.
  - Είναι το παιχνίδι δίκαιο;

```
def binary_search(s, key, low, high):
    if low > high:
        return -1
    middle = (low + high) // 2
    if s[middle] == key:
        return middle
    if s[middle] > key:
        return binary_search(s, key, low, middle - 1)
    else:
        return binary_search(s, key, middle + 1, high)
```

Παραλλαγές δυαδικής αναζήτησης: Καταμέτρηση εμφανίσεων τιμής

- Καταμέτρηση εμφανίσεων τιμής (counting occurrences): Εύρεση πλήθους εμφανίσεων ενός κλειδιού k σε μια ταξινομημένη ακολουθία
- Εφαρμογή δυαδικής αναζήτησης για εντοπισμό ενός στοιχείου της λίστας με τιμή ίση με k και διάσχιση αριστερά και δεξιά για την εύρεση των ορίων (n=μέγεθος λίστας, s=πλήθος εμφανίσεων του k)
  I. Πολυπλοκότητα: O(log n + s)
  II. Ποια είναι η πολυπλοκότητα χειρότερης περίπτωσης;
  III. Μπορούμε να κάνουμε κάτι καλύτερο;

Καταμέτρηση εμφανίσεων τιμής σε ταξινομημένη λίστα – καλύτερη λύση 1

Αρχικός κώδικας

```
def binary_search(s, key, low, high):
    if low > high:
        return -1
    middle = (low + high) // 2
    if s[middle] == key:
        return middle
    if s[middle] > key:
        return binary_search(s, key, low, middle - 1)
    else:
        return binary_search(s, key, middle + 1, high)
```

Εύρεση ορίου προς τις υψηλότερες τιμές

```
def binary_search_high(s, key, low, high):
    if low > high:
        return high
    middle = (low + high) // 2
    if s[middle] > key:
        return binary_search_high(s, key, low, middle - 1)
    else:
        return binary_search_high(s, key, middle + 1, high)
```

Εύρεση ορίου προς τις χαμηλότερες τιμές

```
def binary_search_low(s, key, low, high):
    if low > high:
        return low
    middle = (low + high) // 2
    if s[middle] >= key:
        return binary_search_low(s, key, low, middle - 1)
    else:
        return binary_search_low(s, key, middle + 1, high)
```

Καταμέτρηση εμφανίσεων τιμής σε ταξινομημένη λίστα – καλύτερη λύση 2

Αρχικός κώδικας

```
def binary_search(s, key, low, high):
    if low > high:
        return -1
    middle = (low + high) // 2
    if s[middle] == key:
        return middle
    if s[middle] > key:
        return binary_search(s, key, low, middle - 1)
    else:
        return binary_search(s, key, middle + 1, high)
```

Εύρεση θέσης για το key +- epsilon

```
def binary_search_eps(s, key, low, high):
    if low > high:
        return (low + high) / 2
    middle = (low + high) // 2
    if s[middle] == key:
        return middle
    if s[middle] > key:
        return binary_search_eps(s, key, low, middle - 1)
    else:
        return binary_search_eps(s, key, middle + 1, high)
```

Δυαδική αναζήτηση μιας κατεύθυνσης (one-sided binary search)

- Έστω μια ακολουθία από μηδενικά που ακολουθείται από μια ακολουθία από μονάδες:
  π.χ. 0000…**01**11111111…
- Δεν γνωρίζουμε το πλήθος των ψηφίων
- Ζητείται ο εντοπισμός της θέσης που συμβαίνει η αλλαγή από 0 σε 1, δηλαδή ο εντοπισμός του σημείου μετάβασης (transition point)

Δυαδική αναζήτηση μιας κατεύθυνσης – μια αποδοτική λύση

- Έλεγχος σε συνεχώς μεγαλύτερα διαστήματα: A[1], A[2], A[4], A[8], A[16], A[32], A[64] μέχρι να βρεθεί η τιμή 1
- Όταν βρεθεί η τιμή 1, συνέχεια με δυαδική αναζήτηση μεταξύ της τελευταίας θέσης στην οποία βρέθηκε τιμή 0 και της θέσης όπου βρέθηκε η τιμή 1
- Για τον εντοπισμό του σημείο μετάβασης αποδεικνύεται ότι αν βρίσκεται στη θέση p, τότε χρειάζονται το πολύ 2*ceil(log(p)) συγκρίσεις

Τετραγωνική ρίζα

- Η τετραγωνική ρίζα ενός αριθμού **n** είναι ένα θετικός αριθμός **r** τέτοιος ώστε $\mathbf{r^2 = n}$
- Γράψτε μια παραλλαγή της δυαδικής αναζήτησης που να εντοπίζει την τετραγωνική ρίζα ενός αριθμού **n**

Υπολογισμός τετραγωνικής ρίζας ενός αριθμού **n**

- Η τετραγωνική ρίζα ενός αριθμού **n >= 1** είναι τουλάχιστον **1** και το πολύ **n**
- Θέτουμε **low = 1**, **high = n** και εφαρμόζουμε δυαδική αναζήτηση
- Το μέσο είναι **middle = (low + high) / 2**
- Σύγκριση του **middle^2** με το **n**:
  - Αν **middle² > n** τότε το **high** γίνεται **middle**
  - Αλλιώς, το **low** γίνεται **middle**

```python
def square_root(n, epsilon = 0.0000001):
    if n < 0:
        raise ValueError("n must be non-negative")
    low = 0
    high = n
    while low < high:
        middle = (low + high) / 2
        if abs(middle * middle - n) < epsilon:
            return middle
        if middle * middle > n:
            high = middle
        else:
            low = middle
```

Αν κληθεί:
square_root(0.04)
Τι θα συμβεί;

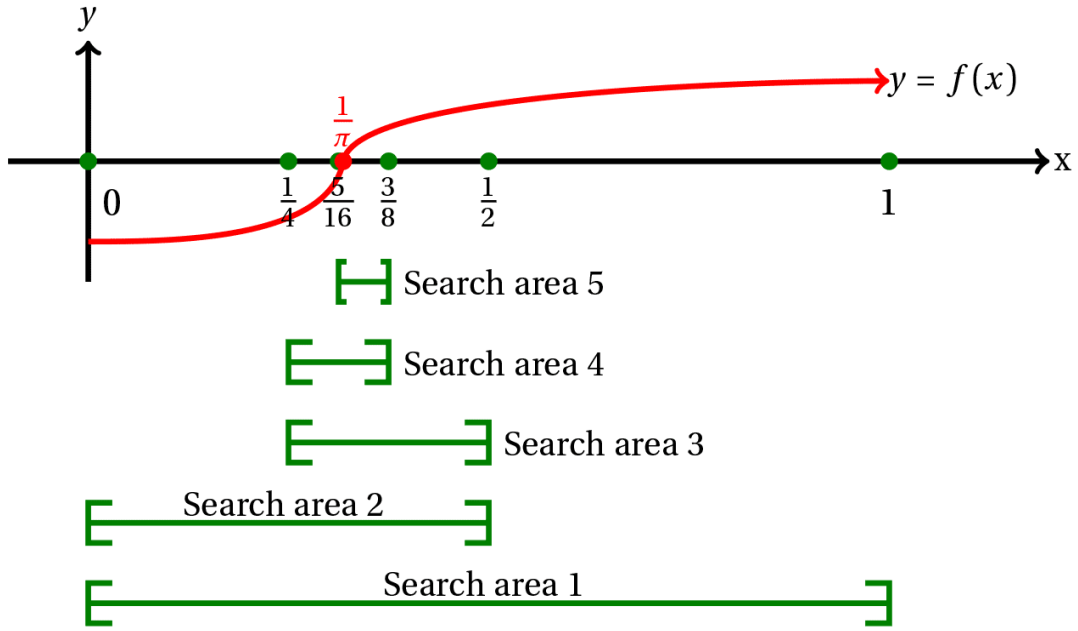
Υπολογισμός τετραγωνικής ρίζας ενός αριθμού **n**

- Ορθή λειτουργία και για αριθμούς > 0 και < 1

```
def square_root(n, epsilon = 0.0000001):
    if n < 0:
        raise ValueError("n must be non-negative")
    if n < 1:
        low = n
        high = 1
    else:
        low = 0
        high = n
    while low < high:
        middle = (low + high) / 2
        if abs(middle * middle - n) < epsilon:
            return middle
        if middle * middle > n:
            high = middle
        else:
            low = middle
```

Μέθοδος διχοτόμησης για εύρεση ριζών εξίσωσης

- Αριθμητική ανάλυση (bisection method)
- Η ρίζα μια συνάρτησης f, είναι η τιμή x για την οποία f(x) = 0
- Αν υπάρχουν low, high τέτοια ώστε f(low) < 0 και f(high) > 0 και η f είναι συνεχής τότε πρέπει να υπάρχει μια ρίζα ανάμεσα στα low και high
- Με βάση το πρόσημο του f(m) με m = (low + high)/2 επιλέγεται η περιοχή τιμών στην οποία θα αναζητηθεί η λύση.



https://amsi.org.au/ESA_Senior_Years/SeniorTopic3/3j/3j_2content_1.html

Η δυαδική αναζήτηση σε βιβλιοθήκες γλωσσών προγραμματισμού και αλλού

C, C++, Python και το git-bisect του GIT

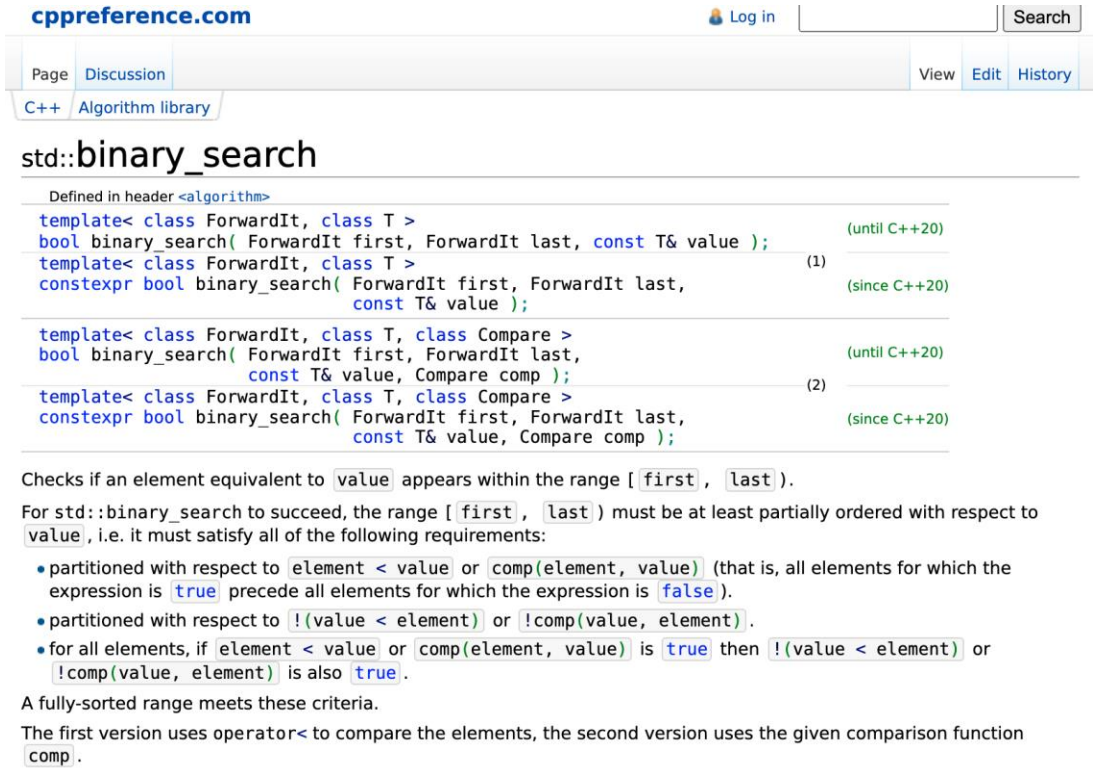
Η δυαδική αναζήτηση ως συνάρτηση της τυπικής βιβλιοθήκης της γλώσσας προγραμματισμού C

$ man bsearch

```
BSEARCH(3)                 Library Functions Manual                 BSEARCH(3)

NAME
     bsearch, bsearch_b – binary search of a sorted table

SYNOPSIS
     #include <stdlib.h>

     void *
     bsearch(const void *key, const void *base, size_t nel, size_t width,
         int (*compar) (const void *, const void *));

     void *
     bsearch_b(const void *key, const void *base, size_t nel, size_t width,
         int (^compar) (const void *, const void *));

DESCRIPTION
     The bsearch() function searches an array of nel objects, the initial
     member of which is pointed to by base, for a member that matches the
     object pointed to by key.  The size (in bytes) of each member of the
     array is specified by width.

     The contents of the array should be in ascending sorted order according
     to the comparison function referenced by compar.  The compar routine is
```

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

// Συνάρτηση σύγκρισης για την qsort και την bsearch
int cmp(const void *a, const void *b) { return (*(int *)a - *(int *)b); }

int main(void) {
  int array[10] = {20, 5, 10, 3, 15, 7, 30, 25, 1, 40};
  int key = 25; // Το στοιχείο που αναζητούμε
  int *found;

  // Ταξινόμηση του πίνακα με την qsort
  qsort(array, 10, sizeof(int), cmp);

  printf("Ταξινομημένος πίνακας: ");
  for (int i = 0; i < 10; i++) {
    printf("%d ", array[i]);
  }
  printf("\n");

  // Δυαδική αναζήτηση με την bsearch
  found = (int *)bsearch(&key, array, 10, sizeof(int), cmp);

  if (found != NULL) {
    printf("Το στοιχείο %d βρέθηκε στη θέση %ld.\n", key,
           (found - array)); // ή found - &array[0]
  } else {
    printf("Το στοιχείο %d δεν βρέθηκε.\n", key);
  }

  return 0;
}
```

Η δυαδική αναζήτηση ως συνάρτηση της τυπικής βιβλιοθήκης της γλώσσας προγραμματισμού C++



cppreference.com

Log in | Search

Page | Discussion | View | Edit | History

C++ | Algorithm library

std::binary_search

Defined in header <algorithm>

```
template< class ForwardIt, class T >
bool binary_search( ForwardIt first, ForwardIt last, const T& value );                    (until C++20)
template< class ForwardIt, class T >                                                      (1)
constexpr bool binary_search( ForwardIt first, ForwardIt last,                            (since C++20)
                              const T& value );

template< class ForwardIt, class T, class Compare >
bool binary_search( ForwardIt first, ForwardIt last,                                      (until C++20)
                    const T& value, Compare comp );                                       (2)
template< class ForwardIt, class T, class Compare >
constexpr bool binary_search( ForwardIt first, ForwardIt last,                            (since C++20)
                              const T& value, Compare comp );
```

Checks if an element equivalent to `value` appears within the range [`first` , `last`).

For `std::binary_search` to succeed, the range [`first` , `last`) must be at least partially ordered with respect to `value` , i.e. it must satisfy all of the following requirements:

- partitioned with respect to `element < value` or `comp(element, value)` (that is, all elements for which the expression is `true` precede all elements for which the expression is `false`).
- partitioned with respect to `!(value < element)` or `!comp(value, element)` .
- for all elements, if `element < value` or `comp(element, value)` is `true` then `!(value < element)` or `!comp(value, element)` is also `true` .

A fully-sorted range meets these criteria.

The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp` .

https://en.cppreference.com/w/cpp/algorithm/binary_search

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> haystack{1, 3, 4, 5, 9};
    std::vector<int> needles{1, 2, 3};

    for (const auto needle : needles)
    {
        std::cout << "Searching for " << needle << '\n';
        if (std::binary_search(haystack.begin(), haystack.end(), needle))
            std::cout << "Found " << needle << '\n';
        else
            std::cout << "No dice!\n";
    }
}
```

Output:

```
Searching for 1
Found 1
Searching for 2
no dice!
Searching for 3
Found 3
```

Επιπλέον, υπάρχουν και οι συναρτήσεις lower_bound και upper_bound

https://en.cppreference.com/w/cpp/algorithm/lower_bound
https://cplusplus.com/reference/algorithm/upper_bound/

Ο αλγόριθμος bisection του module bisect της Python

Python Standard Library

`bisect` — Array bisection algorithm

Source code: Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over linear searches or frequent resorting.

The module is called `bisect` because it uses a basic bisection algorithm to do its work. Unlike other bisection tools that search for a specific value, the functions in this module are designed to locate an insertion point. Accordingly, the functions never call an `__eq__()` method to determine whether a value has been found. Instead, the functions only call the `__lt__()` method and will return an insertion point between values in an array.

https://docs.python.org/3/library/bisect.html

```
import bisect
sorted_list = [1, 3, 4, 7, 9, 12]
new_element = 5
insert_position = bisect.bisect(sorted_list, new_element)
sorted_list.insert(insert_position, new_element)
print(sorted_list)
```

[1, 3, 4, 5, 7, 9, 12]

Performance Notes

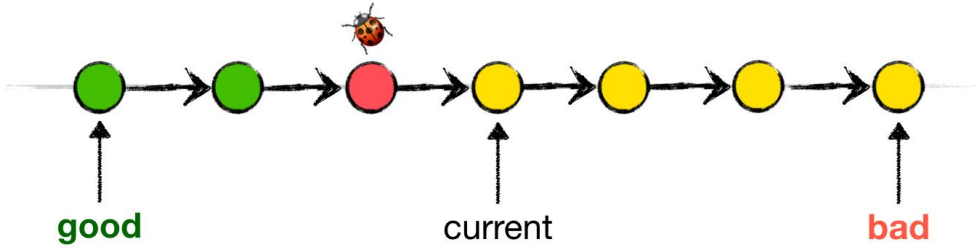
When writing time sensitive code using *bisect()* and *insort()*, keep these thoughts in mind:

- Bisection is effective for searching ranges of values. For locating specific values, dictionaries are more performant.
- The *insort()* functions are `O(n)` because the logarithmic search step is dominated by the linear time insertion step.
- The search functions are stateless and discard key function results after they are used. Consequently, if the search functions are used in a loop, the key function may be called again and again on the same array elements. If the key function isn't fast, consider wrapping it with `functools.cache()` to avoid duplicate computations. Alternatively, consider searching an array of precomputed keys to locate the insertion point (as shown in the examples section below).

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])       # Or use operator.itemgetter(1).
>>> keys = [r[1] for r in data]         # Precompute a list of keys.
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

Μια εφαρμογή της δυαδικής αναζήτησης: Git-bisect

- Η εντολή git-bisect είναι μια ισχυρή εντολή του συστήματος ελέγχου εκδόσεων Git που βοηθά τον χρήστη να εντοπίσει το συγκεκριμένο commit που εισήγαγε κάποιο σφάλμα στον κώδικα
- Αρχικά ορίζονται τα commits που αντιστοιχούν σε «κακό» και σε «καλό» κώδικα με εντολές:
  - git bisect bad <commit-hash>
  - git bisect good <commit-hash>
- Μετά ακολουθείται μια διαδικασία δυαδικής αναζήτησης, όπου εντοπίζεται το commit που βρίσκεται στη μέση για το οποίο ο χρήστης θα πρέπει να απαντήσει αν είναι καλό ή κακό
- Η διαδικασία συνεχίζεται μέχρι να εντοπιστεί το commit που εισήγαγε το πρόβλημα



https://www.metaltoad.com/blog/beginners-guide-git-bisect-process-elimination

Αναδρομικές σχέσεις

- Παραδείγματα αναδρομικών σχέσεων:
  - Ακολουθία Fibonacci: 0,1,1,2,3,5,8,13,… $F_0=0$, $F_1=1$, $F_n = F_{n-1} + F_{n-2}$
  - Εκθετική συνάρτηση: $a_n=2*a_{n-1}$, $a_1=1$ ➔ $a_n=2^n$
  - Παραγοντικό: $a_n=n*a_{n-1}$, $a_1=1$ ➔ $a_n = n!$

- Πολλοί αναδρομικοί αλγόριθμοι έχουν πολυπλοκότητες χρόνου που μπορούν να περιγραφούν με φυσικό τρόπο χρησιμοποιώντας αναδρομικές σχέσεις

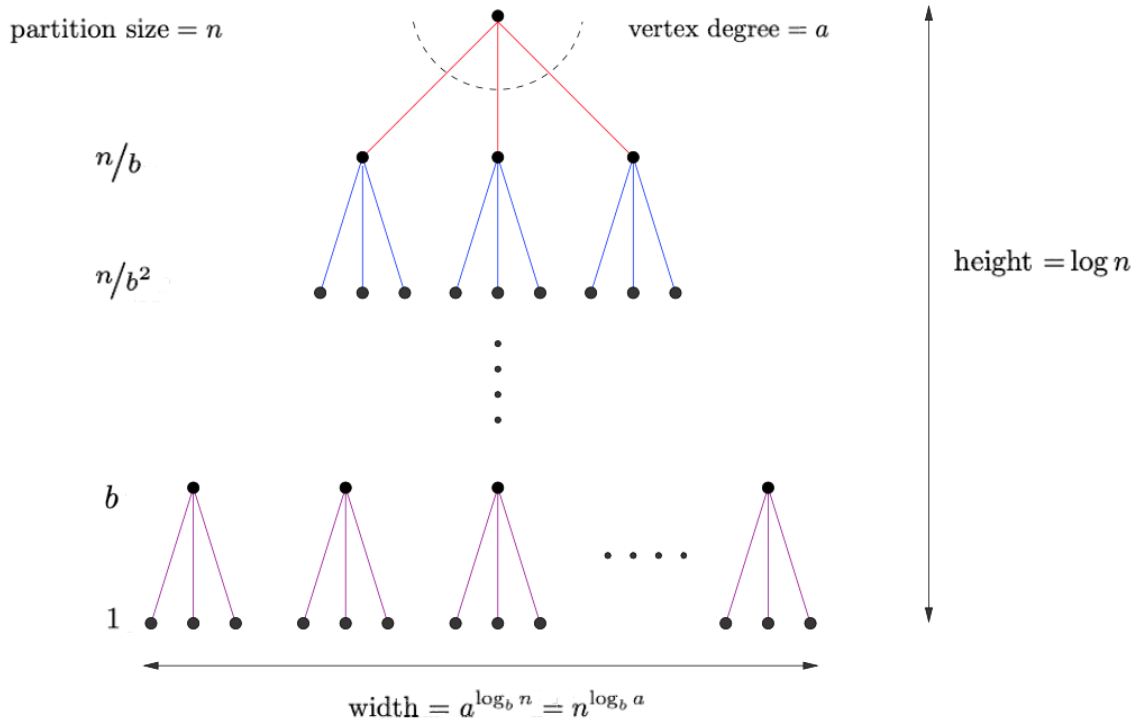
Αναδρομές διαίρει και βασίλευε

- Ένας τυπικός αλγόριθμος διαίρει και βασίλευε διασπά ένα πρόβλημα σε a μικρότερα προβλήματα καθένα από τα οποία είναι μεγέθους n/b και μετά χρειάζεται χρόνο f(n) για να συνδυάσει τις λύσεις σε ένα πλήρες αποτέλεσμα:
  $T(n) = a*T(n/b) + f(n)$

- Παραδείγματα:
  - Ταξινόμηση με συγχώνευση
    $T(n)=2T(n/2)+O(n)$ ➔ $O(n \log n)$
  - Δυαδική αναζήτηση
    $T(n)=T(n/2)+O(1)$ ➔ $O(\log n)$
  - Κατασκευή σωρού με τη μέθοδο bubble_down
    $T(n)=2T(n/2)+O(\log n)$ ➔ $O(n)$

Master θεώρημα

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$.



Περίπτωση 1: Υπάρχουν πολλά φύλλα - αν το πλήθος των φύλλων υπερισχύει του κόστους αποτίμησης εσωτερικά στο δένδρο τότε ο συνολικός χρόνος εκτέλεσης είναι $O(n^{\log_b a})$

Περίπτωση 2: Ισορροπία εργασίας ανά επίπεδο του δένδρου – καθώς κινούμαστε προς τα κάτω στο δένδρο, κάθε πρόβλημα γίνεται μικρότερο, αλλά υπάρχουν περισσότερα προβλήματα, ο συνολικός χρόνος εκτέλεσης είναι $O(n^{\log_b a} \log n)$

Περίπτωση 3: Το κόστος επεξεργασίας της ρίζας υπερισχύει – η εργασία για την κατασκευή των εσωτερικών κόμβων υπερισχύει, ο συνολικός χρόνος είναι $O(f(n))$

https://www.nayuki.io/page/master-theorem-solver-javascript

Πολλαπλασιασμός ακεραίων

- Υποθέτουμε ότι ο πολλαπλασιασμός δύο μονοψήφιων ακεραίων γίνεται σε σταθερό χρόνο (πιθανά χρησιμοποιώντας έναν lookup πίνακα)
- Ο πολλαπλασιασμός δύο ακεραίων με n ψηφία ο καθένας χρειάζεται $O(n^2)$ χρόνο με το γνωστό αλγόριθμο πολλαπλασιασμού ακεραίων

- Εναλλακτικός τρόπος πολλαπλασιασμού ακεραίων με διαίρει και βασίλευε:
  - Υποθέτουμε ότι κάθε αριθμός έχει n=2m ψηφία
  - Κάθε αριθμός διασπάται σε δύο ομάδες m ψηφίων η καθεμία, a_0a_1 και b_0b_1 αντίστοιχα.
  - Αν $w = 10^{m+1}$, τότε $A = a_0 + a_1w$ και $B = b_0 + b_1w$

$$A \times B = (a_0 + a_1w) \times (b_0 + b_1w) = a_0b_0 + a_0b_1w + a_1b_0w + a_1b_1w^2$$

  - $T(n) = 4T(n/2) + O(n) \xrightarrow{MT} O(n^2)$

Αλγόριθμος του Karatsuba

$$A \times B = (a_0 + a_1 w) \times (b_0 + b_1 w) = a_0 b_0 + a_0 b_1 w + a_1 b_0 w + a_1 b_1 w^2$$
$$= q_0 + (q_1 - q_0 - q_2) w + q_2 w^2$$

$$q_0 = a_0 b_0$$
$$q_1 = (a_0 + a_1)(b_0 + b_1)$$
$$q_2 = a_1 b_1$$

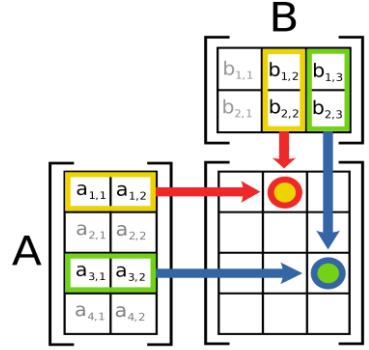
Χρειάζονται 3 πολλαπλασιασμοί (q0, q1, q2) και ένας σταθερός αριθμός αθροίσεων!

$T(n) = 3T(n/2) + O(n) \overset{MT}{\rightarrow} O\left(n^{\log_2 3}\right) = O(n^{1.585})$

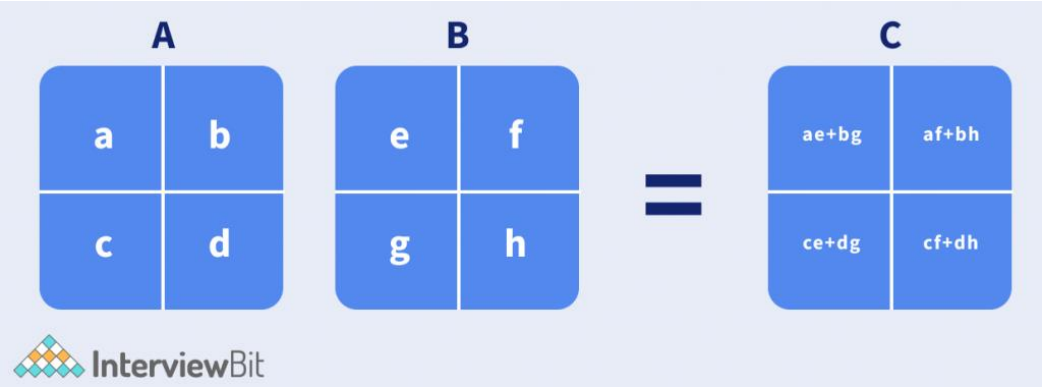
Αν τα ψηφία των αριθμών που πολλαπλασιάζονται είναι πολλά (π.χ. 500) τότε ο αλγόριθμος του Karatsuba (λιγότεροι πολλαπλασιασμοί, περισσότερα αθροίσματα) νικά το γνωστό αλγόριθμο!

https://brilliant.org/wiki/karatsuba-algorithm/

Πολλαπλασιασμός πινάκων



- Πολλαπλασιασμός πίνακα A(n x n) με πίνακα B(n x n)

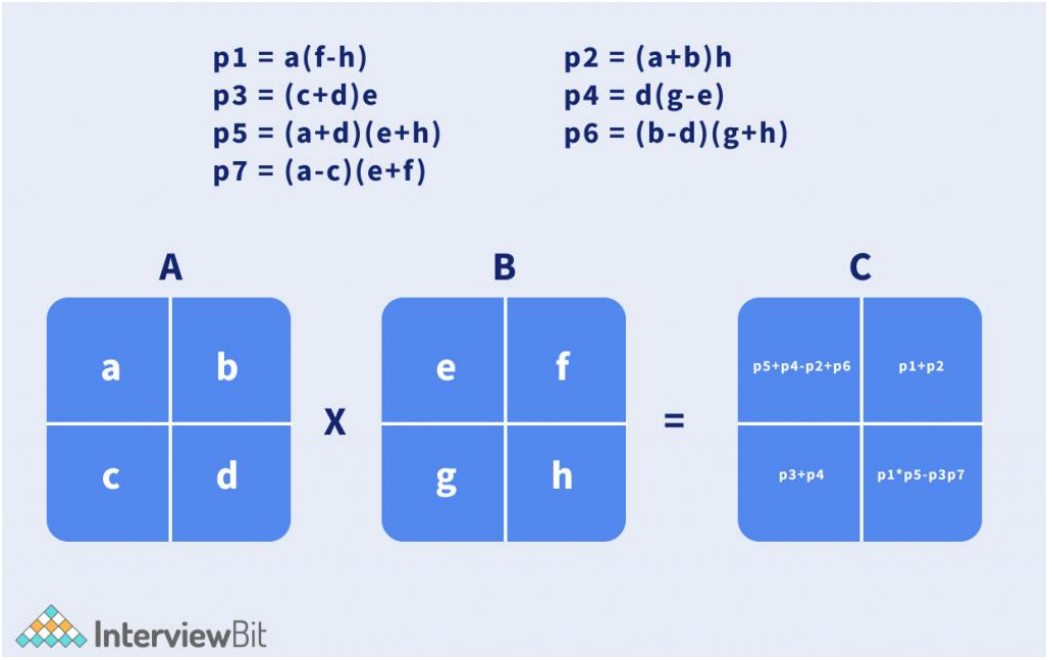


Διαίρει και βασίλευε:

$T(n) = 8T(n/2) + O(n^2)$ ➔ $O(n^3)$

https://www.interviewbit.com/blog/strassens-matrix-multiplication/

Αλγόριθμος του Strassen

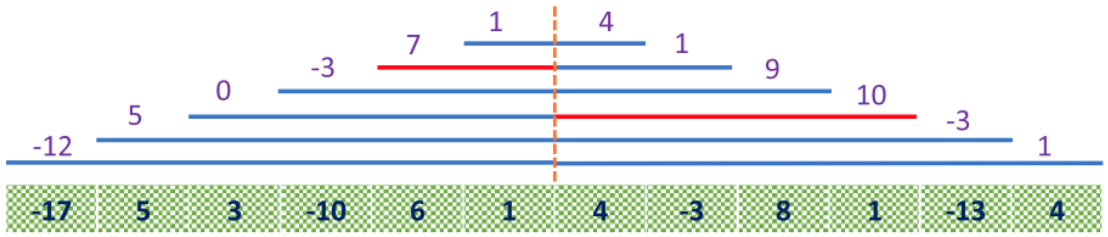


$T(n) = 7T(n/2) + O(n^2)$ ➔ $O(n^{2.8074})$

$O(n^{2.8074})$ --εξέλιξη--> $O(n^{2.3727})$

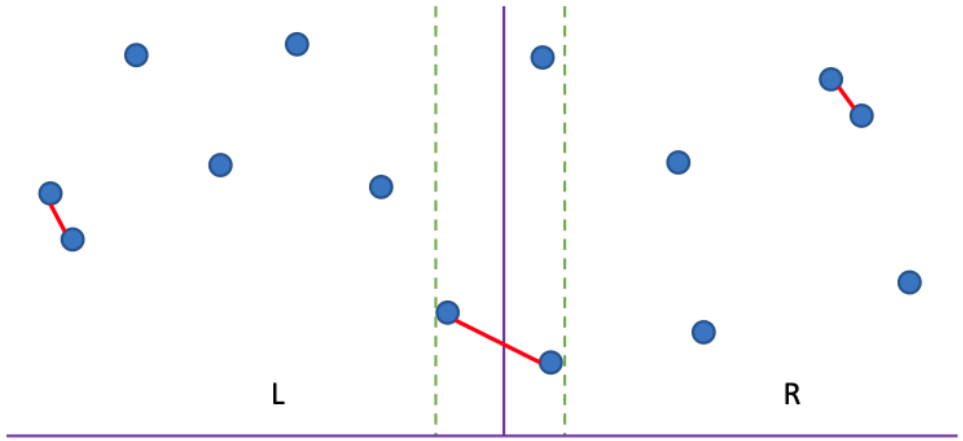
Μέγιστη υποακολουθία

- Στο πρόβλημα της μέγιστης υποακολουθίας (largest subrange ή maximum subarray) δίνεται μια ακολουθία Α από τιμές και ζητείται να βρεθεί η μεγαλύτερη υποακολουθία τιμών
- Μια απλή λύση που δοκιμάζει όλες τις πιθανές θέσεις για αρχή και τέλος της υποακολουθίας χρειάζεται $O(n^3)$ χρόνο



- Υπάρχει λύση διαίρει και βασίλευε που εντοπίζει τη μέγιστη υποακολουθία σε χρόνο O(n logn)
- Η ακολουθία χωρίζεται σε δύο μισά, η μεγαλύτερη υποακολουθία θα βρίσκεται είτε στο αριστερό μισό, είτε στο δεξί μισό, είτε θα ξεκινά στο αριστερό και θα τελειώνει στο δεξί μισό
- Βασική ιδέα:
  - Έστω m το κέντρο, η μεγαλύτερη υποακολουθία που διασχίζει το m θα είναι η μεγαλύτερη υποακολουθία από το αριστερό τμήμα με τελευταίο στοιχείο το m και θα συνεχίζει με τη μεγαλύτερη υποακολουθία του δεξιού τμήματος με αρχή το m+1
  - Και οι δύο υποακολουθίες μπορούν να εντοπιστούν με ένα πέρασμα, άρα απαιτούν χρόνο O(n)
- Ο χρόνος που απαιτείται είναι:
  $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$

Πλησιέστερα σημεία σε 2D



- T(n) = 2T(n/2) + O(n) ➔ O(n log n)

- Ταξινόμηση των σημείων σύμφωνα με τις x-συντεταγμένες τους
- Διαίρεση των σημείων σε σημεία αριστερά και σημεία δεξιά της μεσαίας x-τιμής
- Τα πλησιέστερα σημεία θα είναι είτε στα σημεία δεξιά, είτε στα σημεία αριστερά, είτε σε σημεία που η απόστασή τους «διασχίζει» τη μεσαία τιμή
- Στην τρίτη περίπτωση χρειάζεται να εξεταστεί μια λωρίδα με πλάτος ίσο με τη μικρότερη τιμή απόστασης σημείων που έχει εντοπιστεί στο αριστερό και στο δεξιό τμήμα

Αναφορές

- The Algorithm Design Manual, Steven Skiena, 3rd edition