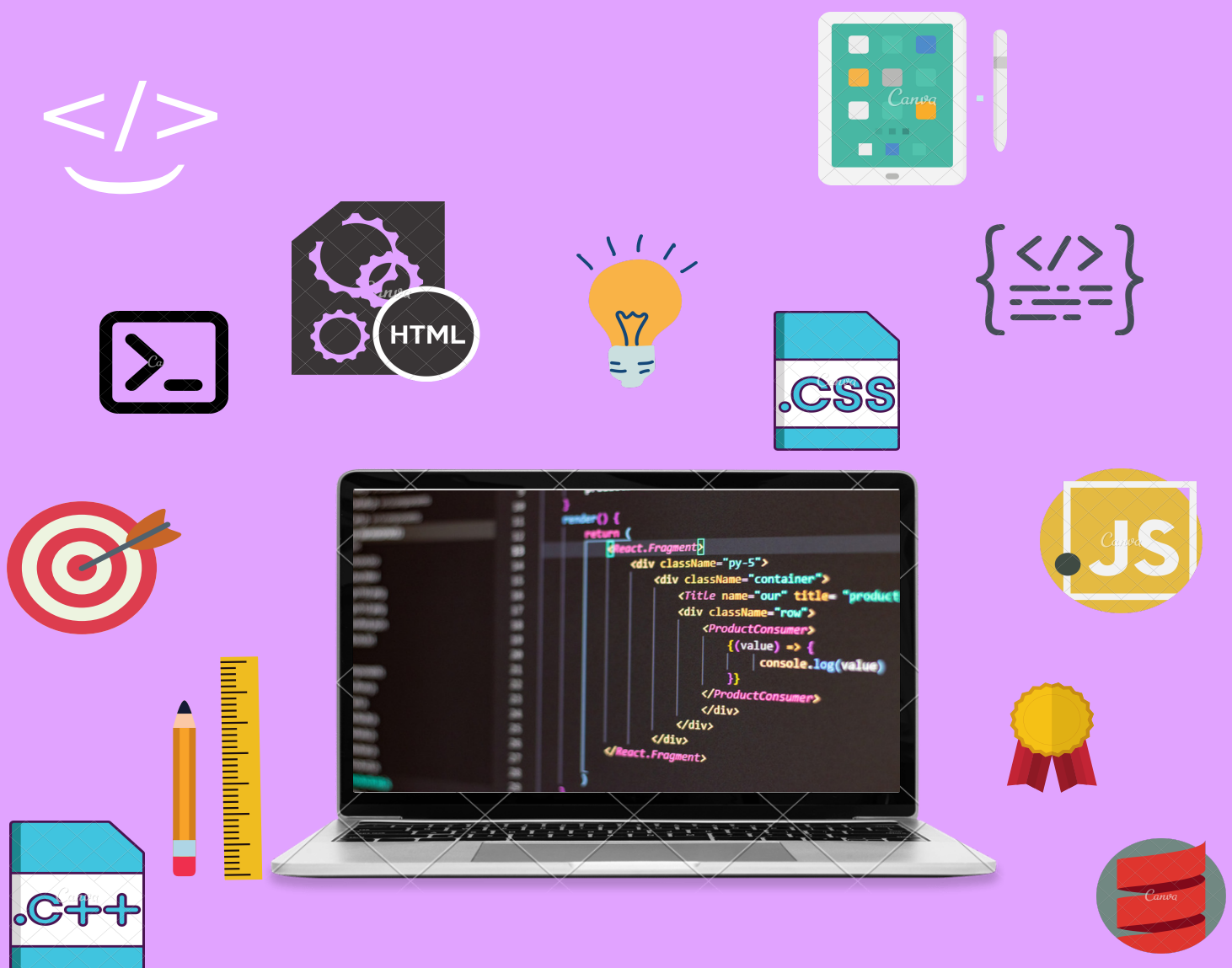


PROJECT: "DATA STRUCTURES 2022"



ΚΑΖΑΣ ΣΤΑΜΑΘΗΣ

(1084636)

ΤΣΙΜΗ ΣΤΥΛΙΑΝΗ

(1084572)

ΣΑΜΑΡΑ ΧΡΙΣΤΙΑ-ΕΛΕΑΝΝΑ

(1084622)

ΤΟΠΑΛΗΣ ΛΑΖΑΡΟΣ

(1088101)

ΠΕΡΙΕΧΟΜΕΝΑ

Πρόλογος.....	4
Part I.....	5
Άνοιγμα αρχείου.....	6
Insertion Sort.....	7
QuickSort.....	9
Ταξινόμηση Σωρού (Heap Sort).....	11
Ταξινόμηση με μέτρηση(Counting Sort).....	14
Δυαδική Αναζήτηση (Binary Search).....	17
Διαδικασία Εκτέλεσης.....	17
Παράδειγμα Υλοποίησης.....	17
Χρονική Πολυπλοκότητα.....	19
Καλύτερη Περίπτωση(Best Case Time Complexity).....	19
Μέση Περίπτωση (Average Case Time Complexity).....	19
Χειρότερη Περίπτωση (Worst Case Time Complexity).....	19
Για το ocean.csv.....	19
Παρατηρήσεις.....	20
Αναζήτηση Παρεμβολής (Interpolation Search).....	21
Διαδικασία Εκτέλεσης.....	21
Παράδειγμα Υλοποίησης.....	21
Χρονική Πολυπλοκότητα.....	22
Καλύτερη Περίπτωση (Best Case Time Complexity).....	22
Μέση Περίπτωση (Average Case Time Complexity).....	22
Χειρότερη Περίπτωση (Worst Case Time Complexity).....	22
Για το ocean.csv.....	23
Παρατηρήσεις.....	23
Δυαδική Αναζήτηση vs Αναζήτηση Παρεμβολής.....	24
Δυική Αναζήτηση Παρεμβολής (BIS).....	25
Βελτιστοποιημένη Δυική Αναζήτηση Παρεμβολής (optimized BIS).....	28
PART II.....	29
AVL by date.....	30

AVL by temperature.....	33
Hashing.....	35
Πηγές.....	38

Πρόλογος

Το Project υλοποιήθηκε κατά το εαρινό εξάμηνο 2021-2022 και είναι διαθέσιμος και διαδικτυακά μέσω της πλατφόρμας του [github](#). Στόχος και σκοπός του ήταν η εκμάθηση και η εξάσκηση των φοιτητών σε βασικούς αλγορίθμους ταξινόμησης και αναζήτησης καθώς και σε δομές δέντρων (AVL tree) και σε δομές δεδομένων (Hashing). Το Project εκτείνεται σε δύο βασικά μέρη, όπου στο πρώτο γίνεται μια προσπάθεια παρουσίασης των αλγορίθμων ταξινόμησης Insertion Sort, Quick Sort, Counting Sort, Heap Sort και των αλγορίθμων αναζήτησης Binary Search, Interpolation Search, Binary Interpolation Search καθώς επίσης και τον optimized Binary Interpolation Search. Σημαντικό είναι να ειπωθεί, πως οι αλγόριθμοι έχουν υλοποιηθεί ξεχωριστά και όχι ενιαία σε ένα κοινό πρόγραμμα, με συνέπεια να υπάρχουν διαφορετικά εκτελέσιμα αρχεία για κάθε αλγόριθμο ταξινόμησης, ενώ οι αλγόριθμοι αναζήτησης υλοποιήθηκαν ανά δύο ξεχωριστά.



Σε αντίθεση με το πρώτο μέρος, το δεύτερο έχει υλοποιηθεί ως ένα ενιαίο πρόγραμμα εφοδιασμένο με μια πολύ βασική αλλά καθόλα εύχρηστη γραφική διεπαφή. Ο χρήστης έχει την επιλογή να ταξινομήσει τα δεδομένα του αρχείου είτε σε μορφή AVL δέντρου είτε σε μορφή Hashing. Στην πρώτη περίπτωση μάλιστα δίνεται η δυνατότητα ταξινόμησης είτε με βάση την ημερομηνία είτε με βάση την θερμοκρασία.

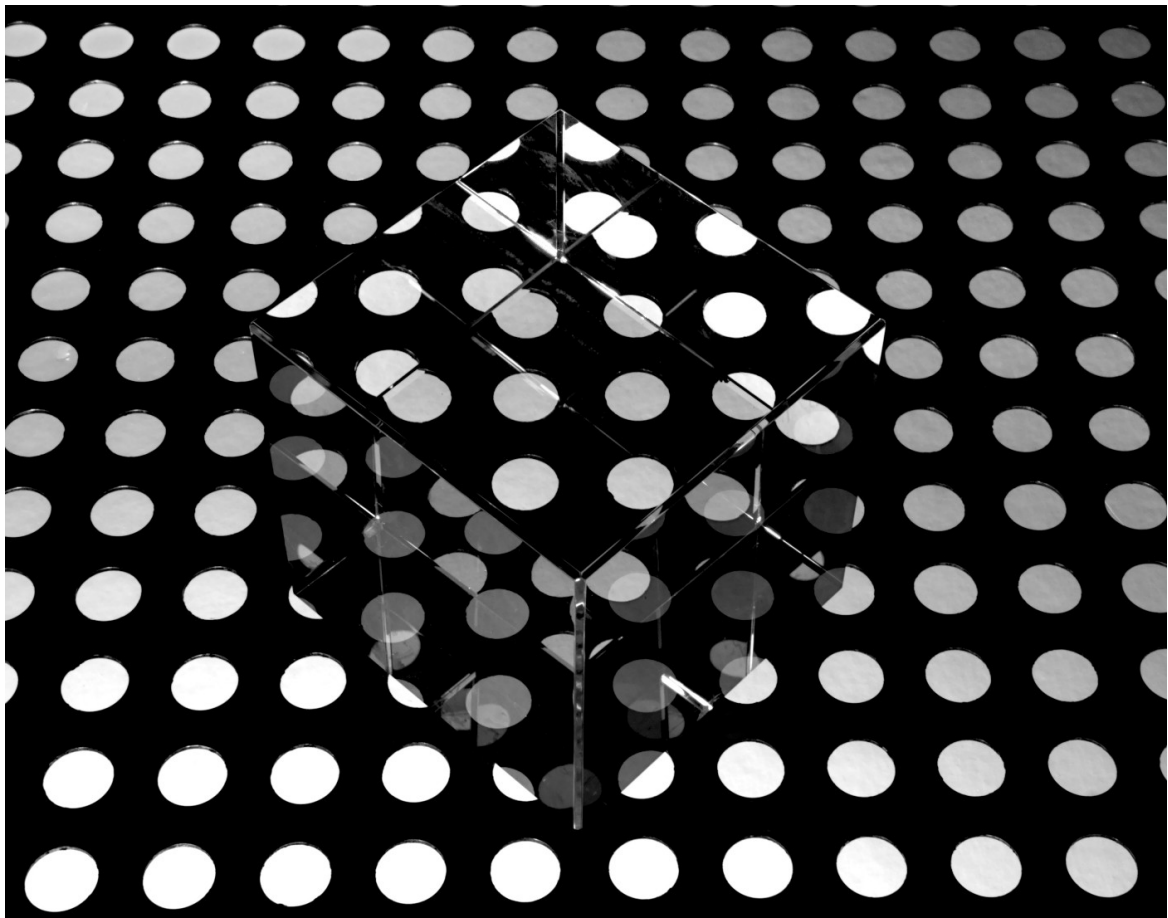
Ακόμα, θα θέλαμε να επισημάνουμε ότι και τα δύο μέρη υλοποιήθηκαν, έχοντας κατασκευαστεί αρχικά βιβλιοθήκες συναρτήσεων, ώστε το πρόγραμμα να είναι αφενός πιο διαχειρίσιμο και αφετέρου πιο ευανάγνωστο. Καθώς αυτή η τακτική οδήγησε στην ύπαρξη πολλών διαφορετικών αρχείων κάτι που κάνει την μεταγλώττιση του προγράμματος περίπλοκη. Για τον λόγο αυτόν και τα δύο μέρη έχουν εφοδιαστεί με μια Makefile η οποία αυτοματοποιεί αυτήν την διαδικασία.

Για την εκτέλεση του Makefile απαιτείται η εκτέλεση της εντολής `make` εφόσον βρισκόμαστε στον ίδιο φάκελο. Σε σύστημα με λειτουργικό Windows είναι απαραίτητη και “η εγκατάσταση του πακέτου [GNU make](#) χρησιμοποιώντας το [chocolatey](#) ώστε να προστεθεί η εντολή `make` στο global path και έπειτα η εκτέλεση της εντολής `choco install make`”, όπως αναφέρεται και στο [stackoverflow](#), ενώ σε σύστημα με Linux ή MacOS δεν απαιτείται καμία ενέργεια. ;)

Τέλος, θα θέλαμε να δώσουμε ιδιαίτερη μνεία στους καθηγητές Χ. Μακρή, Σ. Σιούτα, στον υποψήφιο διδάκτορα Γ. Βονιτσάνο, για την ανάθεση και διόρθωση της παρούσας εργασίας και στον επίτιμο καθηγητή Α. Τσακαλίδη, το βιβλίο του οποίου αποτέλεσε σημαντικό αρωγό στην εκπόνηση της.

Πάτρα,
Ιούλιος 2022

Part I



Στο πρώτο μέρος του project πραγματοποιήθηκε η υλοποίηση και ο έλεγχος μεμονωμένων απλών αλγορίθμων ταξινόμησης και αναζήτησης. Συγκεκριμένα, υλοποιήθηκαν οι Insertion Sort, Quick Sort, Heap Sort και Counting Sort, ενώ στην περίπτωση των αναζητήσεων οι Binary Search, Interpolation Search, Binary Interpolation Search (εφεξής “BIS”) καθώς και ο Optimized Binary Interpolation Search, ο οποίος μειώνει τον χρόνο χειρότερης περίπτωσης του BIS.

Άνοιγμα αρχείου

Για το άνοιγμα του αρχείου φτιάξαμε μια συνάρτηση με όνομα `open_file`, η οποία παίρνει ως όρισμα το αρχείο `ocean.csv`, έναν πίνακα `data array` τύπου `table_data *` και μια συμβολοσειρά `contents` τύπου `char*`. Στον πίνακα `data array` αποθηκεύουμε όλα τα στοιχεία του αρχείου και στην συμβολοσειρά `contents` αποθηκεύουμε την πρώτη γραμμή του αρχείου `ocean.csv`, δηλαδή τις κατηγορίες δεδομένων που διαθέτουμε. Για να χωρίσουμε τα δεδομένα του αρχείου χρησιμοποιήσαμε τη συνάρτηση `strtok` η οποία μας χώρισε τα `string` με βάση τον οριοθέτη `“,”` με αποτέλεσμα να χωριστούν ανάλογα με τις κατηγορίες τους. Τέλος αξίζει να σημειωθεί ότι οι ημερομηνίες δεν αντιμετωπίζονται ως `string`, αλλά ως `tm struct` (της βιβλιοθήκης `time`), για δική μας ευχέρεια, ώστε να συγκρίνουμε ευκολότερα ημερομηνίες, όπως και θα χρειαστεί στην συνέχεια του προγράμματος.

Insertion Sort

Ο insertion sort είναι ένας από τους πιο απλούς αλγορίθμους ταξινόμησης στοιχείων. Είναι λιγότερο αποδοτικός σε σχέση με τους υπόλοιπους αλγορίθμους που υλοποιήσαμε, για αυτό και συνήθως δεν χρησιμοποιείται σε πίνακες ή λίστες με πολλά στοιχεία. Με τον αλγόριθμο αυτό ταξινομούμε τα στοιχεία κατά αύξουσα σειρά.

Ο τρόπος με τον οποίο λειτουργεί ο αλγόριθμος, έστω σε έναν πίνακα A όπως και στην συγκεκριμένη περίπτωση είναι ο εξής:

Ο αλγόριθμος ξεκινάει παίρνοντας τα δύο πρώτα στοιχεία του πίνακα A_1 και A_2 . Συγκρίνοντας τα στοιχεία αυτά για να βρεθεί το μεγαλύτερο, εκτελεί δύο συγκρίσεις:

1. Αν το πρώτο στοιχείο του πίνακα είναι μικρότερο από το δεύτερο στοιχείο $A_1 < A_2$, ο πίνακας παραμένει ίδιος.
2. Ενώ αν το πρώτο στοιχείο του πίνακα είναι μεγαλύτερο από το δεύτερο στοιχείο $A_1 > A_2$, εκτελείται μια εντολή swap που αλλάζει τις θέσεις των δύο στοιχείων στον πίνακα.

Ως αποτέλεσμα των συγκρίσεων αυτών έχουμε την ταξινόμηση των δύο πρώτων στοιχείων του πίνακα A από το μικρότερο στο μεγαλύτερο, χωρίς να σημαίνει ότι αυτές είναι και οι τελικές θέσεις τους στον πίνακα A μέχρι το τέλος του αλγορίθμου. Συνεχίζοντας με παρόμοιο τρόπο, παίρνουμε τα στοιχεία A_2 και A_3 με σκοπό να εκτελέσουμε πάλι τις συγκρίσεις (1) και (2) για να βρούμε την μαθηματική σχέση που συνδέει τα στοιχεία αυτά. Αν το δεύτερο στοιχείο του πίνακα είναι μικρότερο από το τρίτο στοιχείο $A_2 < A_3$, τότε ο πίνακας παραμένει ο ίδιος. Αν το δεύτερο στοιχείο του πίνακα είναι μεγαλύτερο από το τρίτο στοιχείο $A_2 > A_3$, εκτελείται μια εντολή swap που αλλάζει τις θέσεις των δύο στοιχείων. Ωστόσο, επειδή πριν τα στοιχεία A_2 και A_3 προηγούνται και άλλα στοιχεία (το A_1 στην προκειμένη περίπτωση), θα πρέπει να εκτελέσουμε τις συγκρίσεις (1) και (2) μεταξύ του στοιχείου A_3 (το οποίο πήρε την θέση του στοιχείου A_2 στον πίνακα με την χρήση της swap) και των προηγούμενων στοιχείων, σε περίπτωση που είναι μικρότερο και του A_1 και χρειαστεί πάλι να εκτελεστεί η εντολή swap. Με άλλα λόγια, κάθε φορά που εκτελείται η σύγκριση (2) με επιτυχία για ένα στοιχείο A_i του πίνακα A θα εκτελείται η εντολή swap μεταξύ των A_i και A_{i+1} και μετά θα εκτελείται αναδρομικά για το A_{i+1} η ίδια διαδικασία με τα προηγούμενα στοιχεία του πίνακα (δηλαδή σύγκριση του A_{i+1} που πήρε την θέση του A_i , με τον A_{i-1}).

Με την διαδικασία αυτή, καταφέρνουμε για κάθε στοιχείο που θέλουμε να ταξινομήσουμε, να κάνουμε τις απαραίτητες συγκρίσεις σαρώνοντας τα προηγούμενα στοιχεία του πίνακα και βρίσκοντας την κατάλληλη θέση του στον πίνακα, μετακινώντας ουσιαστικά όλα τα υπόλοιπα στοιχεία μια θέση δεξιά.

Έχοντας ως κατεύθυνση τα παραπάνω, φτιάξαμε την συνάρτηση `insertion_sort`, η οποία παίρνει ως όρισμα ένα `struct data_array`, τύπου `"table_data *"` το οποίο σε κάθε γραμμή περιέχει ένα `struct` τύπου `"table_data"`, δηλαδή όλα τα δεδομένα της κάθε γραμμής του αρχείου `"ocean.csv"`, και έναν ακέραιο αριθμό, τον οποίο και αρχικοποιούμε ύστερα στην `main` της `insertion_sort` με την τιμή `"DATA"` που είναι ίση με τον αριθμό στοιχείων μας, δηλαδή 1405. Παίρνοντας μια μια τις θερμοκρασίες του `data_array`, κάνουμε τις γνωστές συγκρίσεις (1) και (2) και αν εκτελεστεί η σύγκριση (2) με επιτυχία, εκτελούνται οι απαραίτητες ενέργειες, δηλαδή σύγκριση του στοιχείου μικρότερου στοιχείου με τα προηγούμενα στοιχεία του πίνακα.

Όσον αφορά την πολυπλοκότητα του συγκεκριμένου αλγορίθμου, ισχύουν τα εξής:

- Στην χειρότερη περίπτωση, όπου ο πίνακάς μας είναι αντίστροφα ταξινομημένος και αποτελείται από n στοιχεία, χρειαζόμαστε n^2 πράξεις για να τον ταξινομήσουμε, άρα η πολυπλοκότητά μας είναι $T(n)=O(n^2)$.
- Στην καλύτερη περίπτωση όπου ο πίνακάς μας είναι ήδη ταξινομημένος και αποτελείται από n στοιχεία, η πολυπλοκότητα του αλγορίθμου είναι $T(n)=O(n)$.
- Σε μια μέση περίπτωση όπου ο πίνακας αποτελείται από n στοιχεία, η πολυπλοκότητα του αλγορίθμου μας είναι και πάλι $T(n)=O(n^2)$.

QuickSort

Ο quicksort είναι ένας από τους βασικότερους αλγορίθμους ταξινόμησης, η λογική του οποίου στηρίζεται στην αλγοριθμική τεχνική Διαίρει και Βασίλευε (Divide and Conquer). Για την ταξινόμηση των στοιχείων, ο αλγόριθμος αυτός επιλέγει τυχαία ένα στοιχείο του πίνακα το οποίο ορίζει ως άξονα ή αλλιώς τιμή σύγκρισης ή αλλιώς ρινότ, και τοποθετεί τα υπόλοιπα στοιχεία του πίνακα αριστερά αν είναι μικρότερα του ή δεξιά αν είναι μεγαλύτερα του. Έπειτα στους δύο υποπίνακες που δημιουργήθηκαν, εκτελείται η ίδια διαδικασία (δηλαδή βρίσκουμε ένα άξονα στον κάθε υποπίνακα και τοποθετούμε αναλόγως τα υπόλοιπα στοιχεία), και συνεχίζοντας αναδρομικά με ανάλογο τρόπο ταξινομούνται και τα υπόλοιπα στοιχεία του πίνακα.

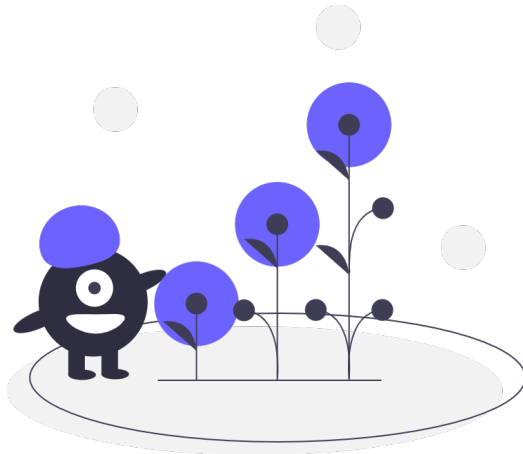
Έχοντας ως κατεύθυνση τα παραπάνω, φτιάξαμε την συνάρτηση quicksort η οποία παίρνει ως όρισμα ένα struct data_array, τύπου "table_data *" το οποίο σε κάθε γραμμή περιέχει ένα struct τύπου "table_data", δηλαδή όλα τα δεδομένα της κάθε γραμμής του αρχείου "ocean.csv", το κάτω όριο του διαστήματος του πίνακα (ή υποπίνακα) των οποίων τα στοιχεία θα συγκριθούν και το πάνω όριο του διαστήματος του πίνακα (ή υποπίνακα) των οποίων τα στοιχεία θα συγκριθούν (το κάτω όπως και το πάνω όριο ορίζονται στην main με την τιμή 0, DATA-1, δηλαδή 1405-1=1404, αντιστοίχως). Τα όρια αυτά εισήχθησαν ως ορίσματα καθώς, αφού, ο αλγόριθμος εκτελείται αναδρομικά σε διαφορετικά υπό-διαστήματα του αρχικού πίνακα (ο οποίος αποτελείται από τα 1405 στοιχεία), οφείλουμε σε κάθε αναδρομή να ορίζουμε την αρχή και το τέλος του πίνακα (και των υποπινάκων στον οποίο θα εφαρμοστεί ο παραπάνω αλγόριθμος. Επίσης, αξίζει να τονιστεί ότι ο quicksort εκτελείται μόνο στους πίνακες, όπου το κάτω όριο παραμένει μικρότερο από το πάνω όριο. Αν δεν ισχύει η σχέση αυτή σημαίνει ότι δεν υπάρχουν άλλα στοιχεία για ταξινόμηση στον πίνακα και συνεπώς φτάσαμε στο επιθυμητό αποτέλεσμα. Το εσωτερικό της συνάρτησης, χτίστηκε με βάση τον ορισμό του αλγορίθμου που δόθηκε παραπάνω, όπου εμείς ως άξονα ή αλλιώς τιμή σύγκρισης ή αλλιώς ρινότ, επιλέγουμε κάθε φορά το πάνω όριο του πίνακα δηλαδή το τελευταίο του στοιχείο του πίνακα σε κάθε περίπτωση.

Όσον αφορά την πολυπλοκότητα του συγκεκριμένου αλγορίθμου, ισχύουν τα εξής:

- Η χειρότερη περίπτωση προκύπτει όταν ως άξονας ή τιμή σύγκρισης ή ρινότ, ορίζεται κάποια ακραία τιμή του πίνακα, δηλαδή ή το πρώτο ή το τελευταίο στοιχείο σε κάθε αναδρομή. Στην περίπτωση αυτή, η πολυπλοκότητα του αλγορίθμου για πίνακα με n στοιχεία είναι $T(n)=O(n^2)$.
- Η καλύτερη περίπτωση προκύπτει όταν ως άξονας ή τιμή σύγκρισης ή ρινότ ορίζεται το μεσαίο στοιχείο σε κάθε αναδρομή. Στην περίπτωση αυτή, η πολυπλοκότητα του αλγορίθμου για πίνακα με n στοιχεία είναι $T(n)=O(n\log n)$.
- Σε μια περίπτωση όπου ο πίνακας αποτελείται από n στοιχεία, η πολυπλοκότητα του αλγορίθμου είναι $T(n)=O(n\log n)$.

Συγκρίνοντας πειραματικά τους δύο αλγορίθμους όσον αφορά την ταχύτητά τους, διαπιστώνουμε και αποδεικνύουμε μιας και γνωρίζαμε ήδη τα αποτελέσματα, ότι ο αλγόριθμος quicksort είναι αισθητά πιο γρήγορος από τον αλγόριθμο insertion sort. Αναλυτικότερα, μια μέση τιμή που απεικονίζει τον χρόνο εκτέλεσης του quicksort είναι η 0,0005813705s ενώ μια μέση τιμή που απεικονίζει τον χρόνο εκτέλεσης της insertion sort είναι η 0,0221361811s.

Ταξινόμηση Σωρού (Heap Sort)



Η λογική της μεθόδου είναι η εξής:

Βήμα 1: Από το μη ταξινομημένο σύνολο S , επιλέγουμε το μεγαλύτερο στοιχείο, έστω m . Το αφαιρούμε από το S και το τοποθετούμε ως το μικρότερο στοιχείο ταξινομημένου συνόλου O .

Βήμα 2: Οργανώνουμε το $S'=S-\{m\}$, για να βρούμε το μεγαλύτερο στοιχείο του S' .

Βήμα 3: Θέτουμε $S'=S$ και επιστρέφουμε στο Βήμα 1.

Η έξοδος μας λαμβάνει την τελική της μορφή, όταν $S=0$. Πώς θα μπορούσαμε να εκτελέσουμε πιο αποδοτικά το Βήμα 1;

Αυτό επιτυγχάνεται με την βοήθεια του ισοζυγισμένου δυαδικού σωρού. Ο ισοζυγισμένος δυαδικός σωρός είναι ένα δυαδικό δέντρο με τις εξής ιδιότητες:

- Κάθε κόμβος u περιέχει μια τιμή από το σύνολο S
- Ισχύει η ιδιότητα του σωρού ($\text{πατέρας}(u) \geq \text{τιμή}(u)$)
- Ως ισοζυγισμένο, κάθε κόμβος του έχει 0 ή 2 παιδιά με πιθανή εξαίρεση ενός κόμβου
- Τα φύλλα έχουν βάθος k ή $k+1$
- Όσα φύλλα είναι στοιχισμένα αριστερά έχουν βάθος $k+1$

Επίσης, σημαντική του ιδιότητα είναι ότι μπορεί να αναπαρασταθεί με έναν πίνακα, για τον οποίο ισχύει ότι $\text{αριθμός}(\text{πατέρας}(u)) = \lfloor \text{αριθμός}(u)/2 \rfloor$. Με άλλα λόγια, ένας πίνακας $S[1..n]$, ονομάζεται σωρός από την θέση i (σύμφωνα με το βιβλίο του κ. Τσακαλίδη), όταν :

Για κάθε j : $i \leq \lfloor j/2 \rfloor \leq j \leq n$ ισχύει: $S[\lfloor j/2 \rfloor] \geq S[j]$

Άρα, συμπεραίνουμε ότι πριν ξεκινήσουμε τον Heap Sort, πρέπει να φτιάξουμε μια συνάρτηση `Heapify()`, που θα μετατρέπει έναν πίνακα $S[1..n]$ με μη διατεταγμένα στοιχεία σε σωρό.

Επεξήγηση του κώδικα της συνάρτησης `Heapify()`:

Αρχικά, ως ορίσματα παίρνει έναν πίνακα τύπου `table_data` (struct που περιέχει κάθε είδος πληροφορίας που θέλουμε να αποθηκεύσουμε), έναν κόμβο και το μέγεθος του σωρού. Δημιουργούμε μια μεταβλητή `bigger` τύπου `int` και την αρχικοποιούμε με τον κόμβο που δώσαμε ως όρισμα. Θεωρούμε, δηλαδή, ότι είναι το `root`. Στη συνέχεια, θέτουμε : `int right = 2 * node + 2;`

`int left = 2 * node + 1;`

Οι κόμβοι αποτελούν τους δείκτες του πίνακα μας. Ελέγχουμε αν το αριστερό παιδί είναι μεγαλύτερο από την μεταβλητή `bigger` (όσο αναφορά τις τιμές του φωσφόρου του καθενός) και αν ναι, θέτουμε ως `bigger` το αριστερό παιδί. Όμοια, ελέγχουμε και για το δεξί. Στη συνέχεια, ελέγχουμε αν η μεταβλητή μας ισούται με τον κόμβο, δηλαδή αν είναι η ρίζα και αν όχι τις κάνουμε `swap`. Τέλος, ξανακαλούμε την συνάρτηση μόνο που τώρα αντί για το `node` στο όρισμα βάζουμε την μεταβλητή `bigger`. Σκοπός μας να κάνουμε σωρό και το υπόλοιπο υπο-δέντρο.

Αφού ολοκληρώσαμε την φάση της δόμησης, πρέπει να υλοποιήσουμε την φάση της διαλογής. Διαλέγουμε και απομακρύνουμε το μεγαλύτερο στοιχείο, που θα βρίσκεται στη ρίζα του σωρού. Εφαρμόζουμε ξανά την φάση της Δόμησης για τα υπόλοιπα στοιχεία και επαναλαμβάνουμε τα βήματα, μέχρι να ταξινομηθεί ο πίνακας.

Επεξήγηση του κώδικα της συνάρτησης `HeapSort()`:

Αρχικά, παίρνει ως ορίσματα έναν πίνακα τύπου `table_data` και το μέγεθος της σωρού. Χτίζουμε την σωρό, καλώντας την συνάρτηση `Heapify()` μέσα σε μία `for`. Η συνάρτηση `Heapify()` παίρνει ως δεύτερο όρισμα το εξής: `heap_size / 2 - 1(=i)`. Από εκείνη τη θέση ξεκινάμε την δημιουργία του σωρού και μετά σε κάθε επανάληψη μειώνουμε το `i` κατά 1 μέχρι να ισχύει `i=0`. Έχουμε φτιάξει το σωρό και τώρα μέσω μιας `for` και καθώς το `array[i]` “βυθίζεται” προς τα κάτω στην σωρό, εναλλάσσεται επαναλαμβανόμενα με το μεγαλύτερο παιδί του(`array[0]`). Έτσι, απομακρύνουμε το μεγαλύτερο στοιχείο κάθε φορά. Τέλος, καλούμε ξανά την `Heapify()` για να “φτιάξει” την σωρό.

Όσο αναφορά, την πολυπλοκότητα της `HeapSort` ισχύει ότι:

$$H(k) \leq n/2^k$$

Όπου n =συνολικός αριθμός κόμβων του σωρού και k =ύψος κόμβων u

Ας δούμε τώρα το κόστος των βημάτων της ταξινόμησης.

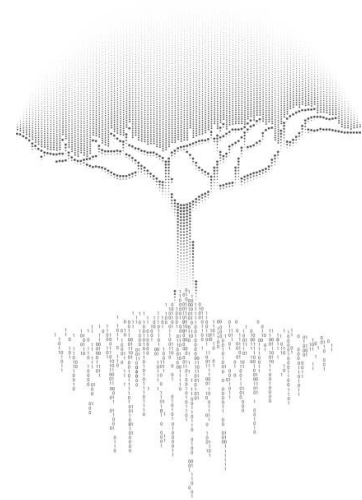
Φάση δόμησης: Ένα στοιχείο $S[i]$ πριν προστεθεί στην σωρό, έστω ότι έχει ύψος ίσο με $\Upsilon\psi\omicron\varsigma(i)$. Μέχρι να γίνει φύλλο, θα χρειαστεί να βυθιστεί το πολύ κατά $\Upsilon\psi\omicron\varsigma(i)$ επίπεδα. Έχουμε, δηλαδή, $O(\Upsilon\psi\omicron\varsigma(i))$ συγκρίσεις και κόστος:

$$\sum_{i=1}^n (\text{κόστος για πρόσθεση του } S[i]) = \sum_{i=1}^n O(\Upsilon\psi\omicron\varsigma(i)) = O\left(n + \sum_{k=0}^{\infty} (k \times H(K))\right) = O(n)$$

Φάση διαλογής: Το μέγιστο στοιχείο θα απομακρυνθεί από τη ρίζα του σωρού το πολύ n φορές. Μετά την απομάκρυνση, χρειάζεται επανόρθωση του σωρού. Στην χειρότερη περίπτωση, θα εκταθεί από την ρίζα ως τα φύλλα. Το μονοπάτι αυτό έχει μήκος $O(\log n)$ με $O(1)$ συγκρίσεις σε κάθε κόμβο. Άρα, έχουμε κόστος $O(\log n)$.

Συμπεραίνουμε ότι το συνολικό κόστος είναι $O(n \log n)$ για ταξινόμηση συνολικού μεγέθους n .

Χρησιμοποιώντας την `clock_t` από την βιβλιοθήκη `time.h`, υπολογίσαμε το χρόνο εκτέλεσης του αλγορίθμου => 0.000934.



Ταξινόμηση με μέτρηση(Counting Sort)

Ο αλγόριθμος Counting Sort είναι ένας απλός αλγόριθμος για την ταξινόμηση ακεραίων. Η ιδέα του είναι ότι μετρά πόσοι αριθμοί είναι μικρότεροι από έναν συγκεκριμένο αριθμό, με σκοπό να βρει την θέση του σε έναν ταξινομημένο πίνακα.

Εάν υποθέσουμε ότι θα ταξινομήσουμε ακεραίους που βρίσκονται στο διάστημα $[1, k]$ και ως είσοδο έχουμε πίνακα $A[1..n]$, τότε θα χρειαστούμε έναν πίνακα $B[1..n]$ και έναν $C[1..k]$ για την παραγωγή της εξόδου και την μέτρηση των τιμών αντίστοιχα.

Ο αλγόριθμος διαιρείται σε 3 φάσεις:

Φάση 1: Αρχικοποιούμε τον C με μηδενικά. Σαρώνουμε τον A από τα αριστερά στα δεξιά. Αυξάνουμε τον μετρητή της θέσης $C[A[i]]$, κάθε φορά που βρίσκουμε την τιμή $A[i]$.

Array:	5	2	5	3	6	1	5	3	9	6
--------	---	---	---	---	---	---	---	---	---	---

Count Array:	0	1	2	3	4	5	6	7	8	9
	0	1	1	2	0	3	2	0	0	1

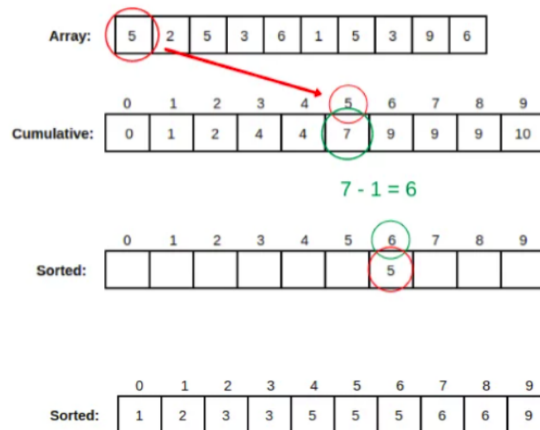
Φάση 2: Από τα αριστερά προς τα δεξιά του πίνακα C προσθέτουμε τα στοιχεία $C[i]$ και $C[i-1]$. Έτσι, σε κάθε θέση j έχουμε το άθροισμα των στοιχείων του A που είναι μικρότερα ή ίσα του j.

Array:	5	2	5	3	6	1	5	3	9	6
--------	---	---	---	---	---	---	---	---	---	---

Count Array:	0	1	1	2	0	3	2	0	0	1
--------------	---	---	---	---	---	---	---	---	---	---

Cumulative:	0	1	2	4	4	7	9	9	9	10
	0	1	2	3	4	5	6	7	8	9

Φάση 3: Σαρώνουμε το A τώρα από τα δεξιά και τοποθετούμε τις τιμές π.χ. i στην κατάλληλη θέση στον πίνακα B, ανάλογα με το $C[i]$. Αν υπάρχουν k στοιχεία μικρότερα από το j, τότε θέτουμε $B[k]=i$ και μειώνουμε κατά ένα το $C[i]$.



Επεξήγηση του κώδικα της ταξινόμησης:

Δημιουργούμε μια συνάρτηση `CountingSort()` με ορίσματα έναν πίνακα(`array[]`) τύπου `table_data` και το μέγεθος του τελικού μας ταξινομημένου πίνακα. Η ταξινόμηση πρέπει να γίνει με βάση την τιμή του φωσφόρου, που δεν αποτελεί πάντα ακέραιο αριθμό. Ο `CountingSort()`, όμως, ταξινομεί μόνο ακραίους και για αυτό πολλαπλασιάζουμε τις τιμές του φωσφόρου των στοιχείων του πίνακα με το 100. Πρέπει να βρούμε την μεγαλύτερη τιμή φωσφόρου. Για να το πετύχουμε, αρχικοποιούμε μια μεταβλητή `max` με την τιμή του φωσφόρου του πρώτου στοιχείου του πίνακα και στην συνέχεια διατρέχουμε με μια `for` τις τιμές φωσφόρου των στοιχείων και τις συγκρίνουμε με την `max`. Όποια είναι μεγαλύτερη γίνεται η νέα τιμή της `max`. Στο τέλος της `for`, έχουμε βρει και τον μέγιστο αριθμό φωσφόρου, αλλά και το μέγεθος-1 του πίνακα(`new_array()`) που θα κρατάει τις τιμές. Αφού τον ορίσουμε, τον αρχικοποιούμε με μηδενικά. Στη συνέχεια, διατρέχουμε τα στοιχεία του `array[]` και αποθηκεύουμε το `count` κάθε στοιχείου \Rightarrow `new_array[(int)(array[i].PO4uM * 100)]`, αυξάνοντας το κάθε φορά που βρίσκουμε την τιμή `(int)(array[i].PO4uM * 100)`.

Διατρέχουμε, τώρα, τον πίνακα `new_array[]` και ακολουθεί η Φάση 2 \Rightarrow `new_array[i] += new_array[i - 1]`. Στην Φάση 3, δημιουργούμε τον τελικό πίνακα (`output[]`) με βάση τον πίνακα `new_array[]` (για όλα τα πεδία που έχουν τα στοιχεία (τύπου `tabledata`) του πίνακα `array[]`) και μειώνουμε κατά 1 το `new_array[(int)(array[i].PO4uM * 100)]`. Τέλος, τοποθετούμε τα στοιχεία του `output[]` στον αρχικό μας πίνακα(`array[]`).

Η χρονική πολυπλοκότητα του αλγορίθμου εξαρτάται από το μέγεθος του αρχικού μας πίνακα και το διάστημα, στο οποίο ανήκουν οι αριθμοί, με τους οποίους δουλεύουμε. Αν χρησιμοποιήσουμε τους πίνακες που αναφέραμε για να εξηγήσουμε τις φάσεις του αλγορίθμου, τότε ισχύει ότι:

1. Η αρχικοποίηση του `C` έχει κόστος k .
2. Η αύξηση του μετρητή της θέσης `C[A[i]]`, κάθε φορά που βρίσκουμε την τιμή `A[i]`, έχει κόστος n . ▫ Η πρόσθεση των στοιχείων `C[i]` και `C[i-1]` από τα αριστερά προς τα δεξιά του πίνακα `C` έχει κόστος $k-1$.
3. Η Φάση 3 έχει κόστος n .

Άρα, το συνολικό κόστος είναι $O(n+k)$.

Χρησιμοποιώντας την `clock_t` από την βιβλιοθήκη `time.h`, υπολογίσαμε το χρόνο εκτέλεσης του αλγορίθμου. Καταλήξαμε ότι εκτελείται κατά μέσο όρο σε 0.000105s. Συμπεραίνουμε, λοιπόν, ότι ο Counting Sort κάνει πιο γρήγορη ταξινόμηση από ότι ο Heap Sort. Αυτό είναι λογικό, αφού ο Heap Sort συμπεριλαμβάνει και το χρόνο που απαιτείται για τη δημιουργία της σωρού.

Δυαδική Αναζήτηση (Binary Search) ¹

Η Δυαδική Αναζήτηση (Binary Search) ανήκει στην οικογένεια των αλγορίθμων Διαίρει και Βασίλευε και εφαρμόζεται σε έναν ταξινομημένο, μονοδιάστατο πίνακα. Είναι αρκετά αποτελεσματικός τρόπος αναζήτησης ενός στοιχείου (key), καθώς η χρονική πολυπλοκότητά του είναι λογαριθμική ως προς το μέγεθος (n) της εισόδου.

Διαδικασία Εκτέλεσης

Η Δυαδική αναζήτηση χρησιμοποιείται σε έναν ταξινομημένο πίνακα. Τα στοιχεία του μπορεί να είναι είτε σε αύξουσα σειρά είτε σε φθίνουσα. Εμείς θα υποθέσουμε ταξινόμηση κατά αύξουσα σειρά.

Ξεκινώντας, η δυαδική αναζήτηση βρίσκει το στοιχείο που βρίσκεται στην μέση του πίνακά μας και το συγκρίνει με το key (δηλ. το στοιχείο το οποίο αναζητάμε).

Στην περίπτωση που είναι ίσα επιστρέφεται η θέση του μεσαίου στοιχείου.

Στην περίπτωση που το μεσαίο στοιχείο είναι μεγαλύτερο από το key, τότε η αναζήτηση θα επαναληφθεί για το αριστερό μισό του πίνακά μας.

Στην περίπτωση που το μεσαίο στοιχείο είναι μικρότερο από το key, τότε η αναζήτηση θα επαναληφθεί για το δεξιό μισό του πίνακά μας.

Με αυτήν την τεχνική, η δυαδική αναζήτηση ελαχιστοποιεί τις πιθανές θέσεις στις οποίες μπορεί να βρίσκεται το στοιχείο που ψάχνουμε, μειώνοντας έτσι τις συνολικές συγκρίσεις που απαιτούνται και κατ' επέκτασιν και τον συνολικό χρόνο αναζήτησης.

Παράδειγμα Υλοποίησης

Παρακάτω έχουμε τον ταξινομημένο πίνακα A, ο οποίος αποτελείται από 10 στοιχεία.

10	14	19	26	27	31	33	35	42	44
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Ας υποθέσουμε ότι εμείς ψάχνουμε τον αριθμό 19, (δηλ. key=19). Αρχικά βρίσκουμε το μέσον (mid) του πίνακα με τον τύπο:

1 Σε όλους τους αλγορίθμους που αφορούν την αναζήτηση στο part I είναι απαραίτητη προϋπόθεση ο πίνακας να είναι ταξινομημένος. Στον συγκεκριμένο κώδικα, ο πίνακας ταξινομείται με χρήση του bubble sort γι' αυτό παρατηρείται καθυστέρηση σε αυτό το στάδιο.

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Όπου low και high οι θέσεις του πρώτου και του τελευταίου στοιχείου της περιοχής του πίνακα στην οποία θέλουμε να κάνουμε αναζήτηση. Εδώ: low = 0 και high = 9, άρα mid = 4 (ο αριθμός mid είναι ακέραιος).

10	14	19	26	27	31	33	35	42	44
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Τώρα συγκρίνουμε το στοιχείο της θέσης που υπολογίσαμε με το key, δηλαδή το 26. Επειδή το στοιχείο που ψάχνουμε είναι μικρότερο του μεσαίου στοιχείου του πίνακα ($19 < 27$) και ο πίνακάς μας είναι ταξινομημένος, για την συνέχεια του αλγορίθμου θα πάρουμε το αριστερό μισό του αρχικού πίνακα.

Το νέο μας high τώρα θα είναι: $\text{high} = \text{mid} - 1 = 3$.

10	14	19	26	27	31	33	35	42	44
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Υπολογίζουμε το νέο mid με τον ίδιο τύπο με πριν και έχουμε: $\text{mid} = 1$ (ακέραιο μέρος του 1,5).

10	14	19	26	27	31	33	35	42	44
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Συγκρίνουμε πάλι το key με το περιεχόμενο της θέσης mid, και βρίσκουμε ότι το στοιχείο που αναζητάμε είναι μεγαλύτερο ($19 > 14$). Άρα, για μια ακόμη φορά υποδιπλασιάζουμε τον πίνακά μας και κρατάμε το δεξιό μισό. Άρα τώρα, το νέο μας low θα είναι: $\text{low} = \text{mid} + 1 = 1$

10	14	19	26	27	31	33	35	42	44
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Για το mid ισχύει: $\text{mid} = 1 + (3 - 1) / 2 = 2$

10	14	19	26	27	31	33	35	42	44
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Αυτήν την φορά, συγκρίνοντας το key με το περιεχόμενο της θέσης mid υπάρχει ταύτιση, άρα καταλαβαίνουμε ότι το στοιχείο μας βρίσκεται στην θέση 5.

Χρονική Πολυπλοκότητα

Έστω N τα στοιχεία του ταξινομημένου πίνακα στον οποίο εφαρμόζουμε την δυαδική αναζήτηση.

Καλύτερη Περίπτωση (Best Case Time Complexity)

Η καλύτερη περίπτωση είναι το στοιχείο που ψάχνουμε να βρίσκεται στο μέσον του πίνακά μας, καθώς αυτό είναι το στοιχείο με το οποίο θα πραγματοποιηθεί η πρώτη σύγκριση.

Σε αυτήν την περίπτωση ο χρόνος πολυπλοκότητας θα είναι $O(1)$.

Μέση Περίπτωση (Average Case Time Complexity)

Η μέση περίπτωση μέσο αριθμό συγκρίσεων $= N * \log N / (N+1) - N/(N+1) + 1/(N+1)$, και επομένως σε αυτήν την περίπτωση ο χρόνος πολυπλοκότητας θα είναι $O(\log N)$.

Χειρότερη Περίπτωση (Worst Case Time Complexity)

Η χειρότερη περίπτωση είναι το στοιχείο που ψάχνουμε να βρίσκεται στην πρώτη ή την τελευταία θέση του πίνακά μας, δηλαδή θα χρειαστούμε $\log N$ συγκρίσεις για να το βρούμε.

Σε αυτήν την περίπτωση ο χρόνος πολυπλοκότητας θα είναι $O(\log N)$.

Για το ocean.csv

Λαμβάνοντας υπόψιν όλα τα παραπάνω, υλοποιήσαμε την Δυαδική Αναζήτηση για το αρχείο που μας δόθηκε. Αρχικά αποθηκεύσαμε τις μετρήσεις του πειράματος σε δομές (struct), τις οποίες ταξινομήσαμε με βάση την ημερομηνία.

Ο αλγόριθμός μας ζητάει από τον χρήστη μια ημερομηνία και στην συνέχεια εκτελεί την δυαδική αναζήτηση με αυτήν την ημερομηνία ως key. Εάν την βρει, ο χρήστης μπορεί να επιλέξει για διάβασμα τα πεδία

PASS

Amount of tests 65535

ALGORITHM	TIME	FOUND	NOT FOUND	SUCCESS RATE
Insertion Sort	: 0.0221361811			
Quick Sort	: 0.0005813705			
HeapSort	: 0.0009342237			
Counting Sort	: 0.0001047003			
Binary search	: 0.0001217531	9872	55663	0.150637
Interpolation search	: 0.0001329886	9872	55663	0.150637
BIS	: 0.0001360918	9872	55663	0.150637
Optimized BIS	: 0.0000396932	9872	55663	0.150637

Οι στήλες FOUND και NOT FOUND δεν αντιπροσωπεύουν επακριβώς αυτό που περιγράφουν. Στις στήλες αυτές παρουσιάζεται το πλήθος των τυχαίων ημερομηνιών που υπήρχαν στο αρχείο (στήλη FOUND) και των ημερομηνιών που δεν υπήρχαν στο αρχείο (στήλη NOT FOUND). Μία πιο εύστοχη απόδοση πιθανώς να ήταν "pass" και "fail". Αντίστοιχα το SUCCESS RATE δείχνει το ποσοστό των ημερομηνιών που υπήρχαν στο αρχείο σε σχέση με το συνολικό πλήθος των δοκιμών. Σε κάθε περίπτωση, το επιθυμητό είναι και οι 4 αλγόριθμοι να έχουν το ίδιο αποτέλεσμα.

Το παραπάνω στιγμιότυπο αποτελεί την έξοδο ενός προγράμματος-tester που κατασκευάστηκε ακριβώς για αυτόν τον λόγο, για να ελεγχεί κατά πόσο οι αλγόριθμοι αναζήτησης εκτελούνται σωστά, καθώς και τον μέσο χρόνο που χρειάζεται κάθε αλγόριθμος ώστε να εκτελεστεί ύστερα από ένα μεγάλο πλήθος δοκιμών (σε αυτήν την περίπτωση 65535) και δεν περιλαμβάνεται στο παραδοτέο αρχείο.

Παρατηρήσεις

Η Δυαδική Αναζήτηση είναι ένας από τους καλύτερους και πιο αποδοτικούς αλγόριθμους αναζήτησης. Η χρονική και χωρική πολυπλοκότητά του είναι επίσης πολύ χαμηλή· η μοναδική προϋπόθεση που χρειάζεσαι να ισχύει πριν την εφαρμογή του σε πίνακα είναι ο πίνακας αυτός να είναι ταξινομημένος.

Και ενώ η ταξινόμηση είναι απαραίτητη για τον αλγόριθμο, εύκολα συμπεραίνουμε ότι η ομοιομορφία στην κατανομή των τιμών του πίνακα δεν διαφοροποιεί την απόδοσή του. Ας παρατηρήσουμε τον τύπο που χρησιμοποιεί για την εκτίμηση της θέσης του στοιχείου αναζήτησης:

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Ο τύπος αυτός (ο γνωστός σε όλους τύπος για την εύρεση του μέσου ενός διαστήματος) λαμβάνει υπόψιν του αποκλειστικά και μόνο το μήκος του

διαστήματος στο οποίο εφαρμόζουμε τον αλγόριθμο. Επομένως, οι τιμές των στοιχείων δεν παίζουν ρόλο στην εκτίμηση της θέσης, αφού σε κάθε επανάληψη η θέση που ελέγχεται είναι το μέσον.

Αναζήτηση Παρεμβολής (Interpolation Search)

Η Αναζήτηση Παρεμβολής αποτελεί μια βελτιωμένη έκδοση της Δυαδικής που εφαρμόζεται σε ταξινομημένο πίνακα στοιχείων. Περιεγράφηκε για πρώτη φορά από τον W. W. Peterson το 1957 και παρομοιάστηκε με τον τρόπο που ψάχνουμε ένα τηλέφωνο στον τηλεφωνικό κατάλογο.



Διαδικασία Εκτέλεσης

Η Αναζήτηση Παρεμβολής αποτελεί μια βελτιωμένη έκδοση της Δυαδικής που εφαρμόζεται σε ταξινομημένο πίνακα στοιχείων. Η διαφορά μεταξύ τους βρίσκεται στον τρόπο με τον οποίο προσεγγίζουν τον υπολογισμό του στοιχείου αναζήτησης (key). Συγκεκριμένα η Δυαδική Αναζήτηση χρησιμοποιεί τον κλασικό τύπο μέσου $mid = low + (high - low)$ και σε κάθε επανάληψη συγκρίνει την τιμή του μέσου του διαστήματος με το key.

Από την άλλη, η Αναζήτηση Παρεμβολής χρησιμοποιεί τον πιο εξελιγμένο τύπο

$pos = low + \left[\frac{(key - arr(low)) * (high - low)}{arr(high) - arr(low)} \right]$, όπου $arr()$ ταξινομημένος πίνακας, low και $high$ η πρώτη και η τελευταία θέση του διαστήματος στο οποίο εκτελούμε την αναζήτηση και key το στοιχείο αναζήτησης. Με αυτόν τον τύπο, η Αναζήτηση Παρεμβολής κάνει μια εκτίμηση της θέσης στην οποία πιθανόν να βρίσκεται το key, με βάζει τις τιμές αρχής και τέλους.

Παράδειγμα Υλοποίησης

Για την πλήρη κατανόηση της λειτουργίας της Αναζήτησης Παρεμβολής, καθώς και για την ευκολότερη σύγκριση με την Δυαδική Αναζήτηση, θα χρησιμοποιήσουμε το ίδιο παράδειγμα με προηγουμένως.

Παρακάτω έχουμε τον ταξινομημένο πίνακα A, ο οποίος αποτελείται από 10 στοιχεία.

10	14	19	26	27	31	33	35	42	44
----	----	----	----	----	----	----	----	----	----

Ας υποθέσουμε ότι εμείς ψάχνουμε τον αριθμό 19, (δηλ. $key=19$). Αρχικά βρίσκουμε την εκτίμηση θέσης (pos) του πίνακα με τον τύπο:

$$pos = low + [(key - arr(low)) * (high - low)] / (arr(high) - arr(low))$$

Όπου low και $high$ οι θέσεις του πρώτου και του τελευταίου στοιχείου της περιοχής του πίνακα στην οποία θέλουμε να κάνουμε αναζήτηση. Εδώ: $low = 0$ και $high = 9$, άρα $pos=2$ (ο αριθμός pos είναι ακέραιος).

10	14	19	26	27	31	33	35	42	44
----	----	----	----	----	----	----	----	----	----

Τώρα συγκρίνουμε το στοιχείο της θέσης που υπολογίσαμε με το key , δηλαδή το 19. Δηλαδή βρήκαμε το στοιχείο που ψάχναμε στην πρώτη επανάληψη του αλγορίθμου. Στην Δυαδική αναζήτηση χρειάστηκαν 3 επαναλήψεις.

Χρονική Πολυπλοκότητα

Έστω N τα στοιχεία του ταξινομημένου πίνακα στον οποίο εφαρμόζουμε την δυαδική αναζήτηση.

Καλύτερη Περίπτωση (Best Case Time Complexity)

Η καλύτερη περίπτωση είναι το στοιχείο που ψάχνουμε να βρίσκεται στο μέσον του πίνακά μας, καθώς αυτό είναι το στοιχείο με το οποίο θα πραγματοποιηθεί η πρώτη σύγκριση.

Σε αυτήν την περίπτωση ο χρόνος πολυπλοκότητας θα είναι $O(1)$.

Μέση Περίπτωση (Average Case Time Complexity)

Στην μέση περίπτωση ο χρόνος πολυπλοκότητας θα είναι $O(\log(\log N))$.

Χειρότερη Περίπτωση (Worst Case Time Complexity)

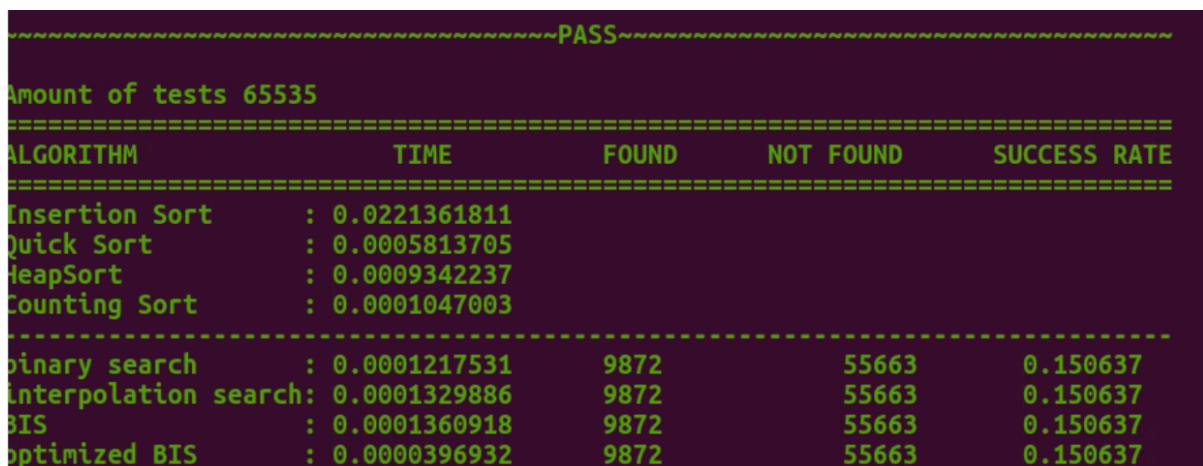
Η χειρότερη περίπτωση είναι ο αλγόριθμος να χρειαστεί να κάνει N συγκρίσεις μέχρι να βρεθεί το στοιχείο αναζήτησης. Αυτό συμβαίνει σε περιπτώσεις που οι αριθμητικές τιμές των στοιχείων του πίνακα αυξάνονται γεωμετρικά.

Στην χειρότερη περίπτωση ο χρόνος πολυπλοκότητας είναι $O(N)$.

Για το ocean.csv

Λαμβάνοντας υπόψιν όλα τα παραπάνω, υλοποιήσαμε την Αναζήτηση Παρεμβολής για το αρχείο που μας δόθηκε. Αρχικά αποθηκεύσαμε τις μετρήσεις του πειράματος σε δομές (struct), τις οποίες ταξινομήσαμε με βάση την ημερομηνία.

Ο αλγόριθμός μας ζητάει από τον χρήστη μια ημερομηνία και στην συνέχεια εκτελεί την δυαδική αναζήτηση με αυτήν την ημερομηνία ως key. Εάν την βρει, ο χρήστης μπορεί να επιλέξει για διάβασμα τα πεδία «Θερμοκρασία» και «Φώσφορος» .



```

PASS

Amount of tests 65535
=====
ALGORITHM                TIME                FOUND              NOT FOUND          SUCCESS RATE
=====
Insertion Sort           : 0.0221361811
Quick Sort               : 0.0005813705
HeapSort                 : 0.0009342237
Counting Sort            : 0.0001047003
-----
Binary search            : 0.0001217531      9872               55663              0.150637
Interpolation search     : 0.0001329886      9872               55663              0.150637
BIS                      : 0.0001360918      9872               55663              0.150637
optimized BIS            : 0.0000396932      9872               55663              0.150637

```

Παρατηρήσεις

Ρίχνοντας μια ματιά στον τύπο που χρησιμοποιεί η Αναζήτηση Παρεμβολής

$$\text{pos} = \text{low} + [(\text{key} - \text{arr}(\text{low})) * (\text{high} - \text{low})] / (\text{arr}(\text{high}) - \text{arr}(\text{low}))$$

συμπεραίνουμε πολύ εύκολα την σημασία που παίζει η κατανομή των στοιχείων στον πίνακα. Ο αλγόριθμος της Αναζήτησης Παρεμβολής βασίζεται στην εξής σκέψη: Αν η τιμή του στοιχείου αναζήτησης βρίσκεται αριθμητικά πιο κοντά στο πρώτο στοιχείο του πίνακα, τότε θα ελεγχθούν τα στοιχεία που είναι πιο κοντά στην αρχή του πίνακα. Αντίστοιχα, στην περίπτωση που το key είναι πιο κοντά στο τελευταίο στοιχείο του πίνακα, οι έλεγχοι θα αρχίσουν από τα κατάλληλα σημεία.

Για τον λόγο αυτό, προϋπόθεση της καλής λειτουργίας της Αναζήτησης Παρεμβολής είναι η ομοιόμορφη κατανομή των στοιχείων του πίνακα στον οποίο εφαρμόζουμε την αναζήτηση, έτσι ώστε να αξιοποιηθούν οι δυνατότητες του αλγορίθμου στο έπακρο.

Δυαδική Αναζήτηση vs Αναζήτηση Παρεμβολής

Οι αλγόριθμοι των δύο αναζητήσεων έχουν ίδια δομή και η μόνη τους διαφορά βρίσκεται στον τρόπο με τον οποίο επιλέγουν στοιχεία για σύγκριση. Αναλύσαμε παραπάνω τους δύο τύπους, και τώρα θα ασχοληθούμε με τις διαφορές τις οποίες προκαλούν στην χρονική πολυπλοκότητα.

Οι μέσοι χρόνοι των Binary και Interpolation, αν N είναι το πλήθος των στοιχείων, είναι: $\log N$ και $\log(\log N)$ αντιστοίχως.



Παρατηρούμε στο παραπάνω διάγραμμα ότι η Αναζήτηση Παρεμβολής, με δεδομένο ότι τα στοιχεία είναι ομοιόμορφα κατανεμημένα, είναι σημαντικά πιο αποτελεσματική από την Δυαδική Αναζήτηση, και αυτός είναι ο λόγος που προτιμάται.

Δυική Αναζήτηση Παρεμβολής (BIS)

Ο BIS αποτελεί μια βελτιωμένη έκδοση του Interpolation Search και προτάθηκε από τους Perl και Reingold. Η κύρια διαφορά του είναι ότι στον BIS μειώνουμε το μέγεθος του μεσοδιαστήματος στο οποίο γίνεται η αναζήτηση από n (που έχει ο Interpolation Search) σε \sqrt{n} (στον BIS).

Και στον BIS, όπως και στον interpolation search, οι πίνακες στους οποίους θα γίνει η αναζήτηση πρέπει να είναι ταξινομημένοι και τα στοιχεία του όσο το δυνατόν πιο ομοιόμορφη κατανομή. Με άλλα λόγια, αν σε έναν πίνακα τα στοιχεία του είναι διαδοχικοί ακέραιοι αριθμοί (π.χ. 5 6 7 8 9), η BIS, όπως και η interpolation search, θα βρει πολύ γρήγορα το στοιχείο που ψάχνουμε. Αυτό, ωστόσο, δεν σημαίνει ότι σε έναν πίνακα (ασφαλώς ταξινομημένο) με όχι και τόσο ομοιόμορφη κατανομή (π.χ. 10 12 15 16 18), ο BIS δεν θα δώσει γρήγορα αποτέλεσμα, αλλά θα χρειαστεί περισσότερες επαναλήψεις σε σχέση με έναν πίνακα με πλήρως ομοιόμορφη κατανομή.

Όπως και στον Interpolation Search, η αναζήτηση ξεκινάει, με τον αλγόριθμο να “μαντεύει” σε ποια θέση βρίσκεται το στοιχείο που αναζητείται. Την θέση την βρίσκει εκτελώντας την πράξη $size \cdot \frac{e \cdot x - S(left)}{S(right) - S(left)} + 1$, όπου size το μέγεθος του

πίνακα, x το στοιχείο που αναζητούμε, $S[left]$ το αριστερότερο στοιχείο (δηλαδή το πρώτο-μικρότερο στοιχείο του πίνακα) και $S[right]$ το δεξιότερο (δηλαδή το τελευταίο-μεγαλύτερο) στοιχείο. Διαισθητικά, ο αλγόριθμος υπολογίζει την απόσταση (με την μαθηματική έννοια) του μικρότερου αριθμού του πίνακα μας σε σχέση με τον μεγαλύτερο καθώς και την (μαθηματική) απόσταση του αριθμού που ζητάμε σε σχέση με τον μεγαλύτερο αριθμό του πίνακά μας και το πηλίκο αυτών των δύο το πολλαπλασιάζει με το μέγεθος του πίνακα.

Η “μαντεψιά” αυτή όπως μπορούμε να αποδείξουμε είναι αρκετά σωστή, καθώς αν έχουμε έναν πλήρως ομοιόμορφα κατανομημένο πίνακα τότε η (απόσταση) του μεγαλύτερου από το μικρότερο αριθμό θα είναι όσο και το μέγεθος του πίνακα μας, και συνεπώς από την παραπάνω σχέση θα έχουμε $x - S(left) + 1$, η οποία πράγματι μας δίνει την ακριβή θέση του στοιχείου.

4	5	6	7	8	9
---	---	---	---	---	---



$$(6-1)*(7-4)/(9-4)+1 =$$

$$5*(3/5)+1 = 4$$

ΕΠΙΤΥΧΙΑ

Αντίθετα, σε έναν ταξινομημένο πίνακα με στοιχεία όχι και τόσο ομοιόμορφα κατανεμημένα, ο αλγόριθμος ναί μεν θα εκτελεστεί σωστά, αλλά θα απαιτηθούν περισσότερες από μια “μαντεψιές” ώστε να βρεθεί η τιμή που ζητείται.

8	12	17	25	29	30	32
---	----	----	----	----	----	----



$$2\eta \text{ Μαντεψιά: } (6-1)*(30-8)/(30-8)+1 =$$

$$5*(22/22)+1 = 6$$

ΕΠΙΤΥΧΙΑ

$$1\eta \text{ μαντεψιά: } (7-1)*(30-8)/(32-8)+1 =$$

$$6*(22/24)+1 = 7$$

ΑΠΟΤΥΧΙΑ

Στην παρούσα εργασία τα δεδομένα του αρχείου είναι οργανωμένα σε μεμονωμένες δομές τύπου struct table_data στα οποία αποθηκεύονται η ημερομηνία και οι τιμές των διάφορων χημικών στοιχείων που παρατηρούνται στον επί εξεταζόμενο ωκεανό. Ο BIS εκτελείται με τον ίδιο τρόπο όπως παρουσιάστηκε προηγουμένως λαμβάνοντας όμως υπόψιν μας την ημερομηνία. Στο σημείο αυτό, κρίνεται χρήσιμο να τονιστεί, ότι στην περίπτωση διπλότυπων ημερομηνιών, ο αλγόριθμος εντοπίζει - και κατ' επέκταση τυπώνει - μόνο ένα αποτέλεσμα.

Ο μέσος χρόνος που απαιτείται για την εύρεση του στοιχείου που ψάχνουμε δίνεται από την αναδρομική σχέση

$$T(n) = C + T(\sqrt{n}),$$

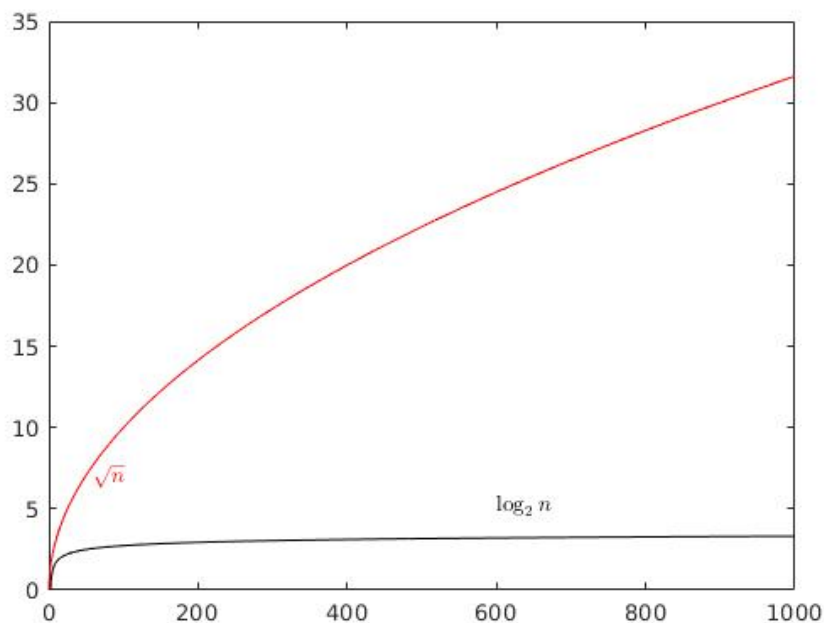
όπου C είναι ο χρόνος που απαιτείται ώστε το μέγεθος ενός πίνακα να μειωθεί από n σε \sqrt{n} και $T(n)$, ο χρόνος που απαιτείται για την αναζήτηση ενός στοιχείου σε έναν πίνακα μεγέθους n . Αναλύοντας την παραπάνω σχέση μπορούμε να δούμε ότι ισχύει

$$T(n) = C \log_2 (\log_2 (n)) \implies O(\log_2 (\log_2 (n))) .$$

Ωστόσο, όταν το στοιχείο που αναζητούμε βρίσκεται κοντά στην τελευταία θέση του του πίνακα, δηλαδή είναι κάποιο από τα μεγαλύτερα στοιχεία του πίνακα μας, τότε ο BIS θα κάνει συνεχόμενα άλματα μεγέθους \sqrt{size} με αποτέλεσμα να καταλήγουμε σε χρόνο χειρότερης περίπτωσης

$$T(n) = \sum_i \frac{n^1}{i} = O(\sqrt{n})$$

που σημαίνει ότι δεν υπάρχει καμία βελτίωση στον χρόνο σε σχέση με τον interpolation search. Για τον λόγο αυτό έχει προταθεί ο optimized Binary Interpolation Search, ο οποίος θα παρουσιαστεί στο κεφάλαιο [Βελτιστοποιημένη Δυική Αναζήτηση Παρεμβολής](#).



Στην συγκεκριμένη υλοποίηση τα δεδομένα που αναλύονται αριθμούν 1405 και συνεπώς αναμένουμε ο μέσος χρόνος να είναι άνω φραγμένος στην τιμή $\log_2 (\log_2 (1405)) = 3.38$ ενώ ο χρόνος χειρότερης περίπτωσης αναμένεται να είναι φραγμένος από την τιμή $\sqrt{1405} = 37.48$. Εκτελώντας τον αλγόριθμο αρκετές φορές φτάνουμε στο συμπέρασμα ότι ο μέσος χρόνος εκτέλεσης του είναι 0.0000745547 δευτερόλεπτα.

Βελτιστοποιημένη Δυική Αναζήτηση Παρεμβολής (optimized BIS)

Όπως αναφέρθηκε και προηγουμένως, αν και ο BIS μειώνει αρκετά τον χρόνο μέσης περίπτωσης, στην περίπτωση χειρότερης περίπτωσης απαιτείται χρόνος όμοιος με αυτόν του interpolation search. Για την αντιμετώπιση αυτού του προβλήματος, προτάθηκε μια εναλλακτική υλοποίηση του BIS. Σύμφωνα με αυτή, αντί τα άλματα να είναι κάθε φορά μεγέθους \sqrt{size} , επιλέγουμε να είναι σε κάθε επανάληψη $2^i \cdot \sqrt{size}$ όπου το i παίρνει τιμές από το 0, με αποτέλεσμα να φτάνει σε υποδιάστημα μεγέθους \sqrt{size} πιο γρήγορα και συνεπώς φτάνει συντομότερα σε λύση.

PART II



Για την απάντηση των ερωτημάτων του Part II δημιουργήθηκε ένα κοινό αρχείο AVL.c, με το οποίο προσαρμόζουμε τα δεδομένα που παίρνουμε από το ocean.csv σε μορφή δέντρου AVL. Στο αρχείο αυτό υλοποιείται η εισαγωγή, μαζί με τις περιστροφές που αυτή χρειάζεται, η διαγραφή κόμβου, καθώς και άλλες χρήσιμες συναρτήσεις, όπως είναι η υψοζύγιση και η εκτύπωση του δέντρου. Οι συναρτήσεις καλούνται από την menu.c.

AVL by date

Για την υλοποίηση του AVL κατασκευάστηκε αρχικά μια δομή η οποία έχει ως σκοπό την αποθήκευση όλων των δεδομένων που απαιτούνται. Αυτά είναι τα δεδομένα που παίρνουμε από το αρχείο, δηλαδή η ημερομηνία (σε μορφή struct tm) και την θερμοκρασία (σε μορφή double). Επίσης, είναι απαραίτητη η δημιουργία του αλγορίθμου εισαγωγής ενός στοιχείου. Αυτός με την σειρά του, ελέγχει αν έχει οριστεί ρίζα ή με άλλα λόγια αν το δέντρο έχει έστω και ένα στοιχείο. Στην περίπτωση που το δέντρο είναι άδειο, δημιουργεί έναν κόμβο και τον θέτει σαν ρίζα μιας και είναι ο μοναδικός κόμβος που υπάρχει στο δέντρο και συνεπώς θα αποτελεί και ρίζα του.

Κάθε κόμβος του δέντρου AVL έχει κατασκευαστεί έτσι ώστε να περιέχει πρωτίστως τα στοιχεία που θέλουμε από το αρχείο (ημερομηνία και θερμοκρασία), έναν δείκτη προς το αριστερό και έναν προς το δεξί παιδί του εκάστοτε κόμβου και δευτερευόντως το ύψος που έχει ο κόμβος. Η δομή που φιλοξενεί την συγκεκριμένη υλοποίηση και σκέψη ονομάζεται Node στον κώδικα.

Στην εισαγωγή μιας δεύτερης τιμής - και γενικότερα στην εισαγωγή ενός στοιχείου σε δέντρο που ήδη περιέχει στοιχεία - αναζητά την θέση στην οποία θα τοποθετηθεί ο καινούριος κόμβος σύμφωνα με τις ιδιότητες των δυαδικών δέντρων² και στην συνέχεια εισάγει σε εκείνο το σημείο έναν κόμβο με τις τιμές που δόθηκαν. Στο σημείο αυτό να τονιστεί ότι δεδομένα με την ίδια ημερομηνία αγνοήθηκαν, καθώς σε ένα AVL δεν υπάρχουν διπλότυπα και, καθώς η ιδιότητα αυτή δεν ζητήθηκε ρητώς, δεν υλοποιήθηκε. Ωστόσο, αναπτύχθηκε και παρουσιάζεται αναλυτικώς στο κεφάλαιο [AVL by temperature](#).

Μέχρι στιγμής ακολουθήθηκε η εισαγωγή ενός κόμβου όπως ακριβώς θα γινόταν και σε ένα binary tree. Η διαφορά του Binary tree με το AVL είναι, ότι το δεύτερο ανήκει στην κατηγορία των ισοζυγισμένων δέντρων. Αυτό οδηγεί στο συμπέρασμα ότι πρέπει να γίνονται οι απαραίτητοι έλεγχοι αν $2 \leq$ Η εισαγωγή ενός νέου κόμβου σε ένα δυαδικό δέντρο ακολουθεί την ιδιότητα πως τα στοιχεία με τιμή μικρότερα ενός κόμβου τοποθετούνται αριστερά του κόμβου, ενώ τα στοιχεία με τιμή μεγαλύτερα του συγκεκριμένου κόμβου δεξιά του. Για παράδειγμα, αν έχουμε έναν κόμβο με τιμή 5 και έπειτα εισάγουμε έναν καινούριο κόμβο με τιμή 2, αφού το 2 είναι μικρότερο από το 5, θα τοποθετηθεί αριστερά του και θα έχουμε (2 - 5). Έπειτα, αν θέλουμε να εισάγουμε κόμβο με τιμή 8, τότε το 8 είναι μεγαλύτερο από το 5 και συνεπώς θα τοποθετηθεί δεξιά του 5 και θα έχουμε (2 - 5 - 8). Στο συγκεκριμένο πρόγραμμα εκτελείται ακριβώς το ίδιο σκεπτικό με την μόνη διαφορά που σαν μέτρο σύγκρισης έχουμε την διαφορά των δευτερολέπτων των δύο κόμβων και έτσι αποφασίζουμε σε ποιο σημείο θα τοποθετηθεί ο νέος κόμβος.

και εφόσον τηρούνται τα κριτήρια ώστε το δέντρο να αποτελεί ένα AVL tree. Για τον σκοπό αυτό, υπολογίζονται τα ύψη των κόμβων από το στοιχείο που προστέθηκε μέχρι την ρίζα, ελέγχοντας τους προγόνους του. Ο υπολογισμός των υψών ακολουθεί έναν πολύ απλό κανόνα: ύψος ενός κόμβου είναι η μέγιστη τιμή των υψών των παιδιών του συγκεκριμένου κόμβου αυξημένη κατά 1. Στην συνέχεια ελέγχουμε την υποζύγηση, δηλαδή την διαφορά του ύψους του αριστερού παιδιού με το ύψος του δεξιού παιδιού. Για την υλοποίηση ενός AVL αποδεκτές τιμές είναι μόνο οι τιμές του συνόλου $\{-1, 0, +1\}$. Οποιαδήποτε άλλη τιμή είναι μη αποδεκτή και πρέπει να ληφθούν τα κατάλληλα μέτρα ώστε να εξασφαλιστεί ότι το δέντρο που κατασκευάζεται ανήκει στην κατηγορία των AVL.

Σύμφωνα με τα την υπόθεση που κάναμε προηγουμένως, ότι η υποζύγηση υπολογίζεται ως η διαφορά των υψών του αριστερού μείον του δεξιού παιδιού, καταλαβαίνουμε ότι όταν προκύψει υποζύγηση -2 τότε το δέντρο “γέρνει” προς τα δεξιά, δηλαδή υπάρχει παραβίαση των κανόνων του AVL στο δεξί υποδέντρο, ενώ αν προκύψει υποζύγηση +2 τότε το δέντρο “γέρνει” προς τα αριστερά, δηλαδή υπάρχει παραβίαση των κανόνων του AVL στο αριστερό υποδέντρο. Οι τεχνικές διόρθωσης τέτοιων προβλημάτων γίνονται με χρήση των αλγορίθμων left rotate (leftRotate function), right rotate (rightRotate function), left-right rotate (LR_rotate function) και right-left rotate (RL_rotate function). Τέλος, αφού εκτελεστούν και αυτές οι περιστροφές και το δέντρο ικανοποιεί τις απαιτήσεις ενός AVL δέντρου, επιστρέφεται και η διεύθυνση της ρίζας ώστε να μπορούμε να έχουμε πρόσβαση σε ολόκληρο το δέντρο.

Όλα όσα αναφέρθηκαν προς το παρόν αφορούν την εισαγωγή ενός στοιχείου το περιεχόμενο του οποίου δίνεται με κάποιον τρόπο (είτε από τον χρήστη είτε “καρφωτά” από το πρόγραμμα). Ωστόσο, εμείς αρχικά επιθυμούμε η αρχική αρχικοποίηση του δέντρου να γίνεται από ένα αρχείο και γι’ αυτό ήταν απαραίτητη η δημιουργία της συνάρτησης insert_from_file, η οποία διαβάζει τα περιεχόμενα ενός αρχείου, στην συγκεκριμένη περίπτωση ενός csv, και περνάει τα δεδομένα σε ένα struct το οποίο μπαίνει σαν όρισμα στην συνάρτηση ώστε να δημιουργηθεί το Node και αυτή με την σειρά της καλεί την insert που παρουσιάστηκε προηγουμένως.

Μία ακόμα συνάρτηση που υλοποιήθηκε είναι η printAVL, η οποία τυπώνει τα περιεχόμενα του δέντρου σε ενδοδιατεταγμένη (συμμετρική) μορφή. Αρχικά επισκεπτόμαστε το αριστερό παιδί, έπειτα την ρίζα και τέλος το δεξί παιδί. Αυτό που αναμένεται στην έξοδο είναι να δούμε τις ημερομηνίες σε αύξουσα σειρά.

Έχει υλοποιηθεί επίσης η αναζήτηση ενός κόμβους, δεδομένου μιας συγκεκριμένης ημερομηνίας την οποία την δίνει ο χρήστης. Για την υλοποίηση του συγκεκριμένου αλγορίθμου εκμεταλλευτήκαμε την

ιδιότητα που παρουσιάστηκε παραπάνω σχετικά με τα δυαδικά δέντρα. Αριστερά από έναν κόμβο βρίσκεται ένα στοιχείο με τιμή μικρότερη από την τιμή του κόμβου, ενώ δεξιά κόμβος με τιμή μεγαλύτερη της τιμής του πατέρα.

Με χρήση του παραπάνω αλγορίθμου δημιουργήθηκε η συνάρτηση (και δυνατότητα) αλλαγής μιας θερμοκρασίας δεδομένης μιας συγκεκριμένης ημερομηνίας. Ο αλγόριθμος εντοπίζει την προς τροποποίηση ημερομηνία και ζητάει από τον χρήστη να δώσει την νέα θερμοκρασία. Έπειτα, βγαίνει ένα προτρεπτικό μήνυμα για να επιβεβαιώσει ότι επιθυμεί να πραγματοποιηθεί η συγκεκριμένη αλλαγή.

Φυσικά, δεν μπορούσε να λείπει η δυνατότητα διαγραφής ενός κόμβου. Και σε αυτήν την περίπτωση αναζητούμε τον κόμβο που θέλουμε να διαγράψουμε. Εφόσον εντοπιστεί διακρίνονται σε 3 βασικές περιπτώσεις για την διαγραφή ενός κόμβου.

- ❖ Αν ο κόμβος δεν έχει κανένα παιδί. Σε αυτήν την περίπτωση απλά διαγράφουμε τον κόμβο, θέτοντας τον δείκτη του πατέρα του σε NULL και αποδεσμεύοντας τον χώρο μνήμης που δέσμευε η συγκεκριμένη εγγραφή. Υπολογίζουμε εκ νέου τα νέα ύψη και ελέγχουμε για παραβιάσεις των κανόνων του AVL με χρήση των υποζυγίσεων. Σε περίπτωση που προκύψουν αραβιάσεις τις λύνουμε με χρήση κατάλληλων περιστροφών.
- ❖ Αν ο προς διαγραφή κόμβος έχει ένα παιδί. Σε αυτήν την περίπτωση αντικαθιστούμε τον κόμβο που διαγράφουμε, με το παιδί του και αποδεσμεύουμε τον χώρο που καταλάμβανε ο αρχικός κόμβος.
- ❖ Αν ο προς διαγραφή κόμβος έχει δύο παιδιά.
 - Αν το δεξί παιδί του προς διαγραφή κόμβου δεν έχει αριστερό παιδί, τότε αντικαθιστούμε το κόμβο με το δεξί του παιδί. Υπολογίζουμε τα ύψη και τις υποζυγίσεις και τυχόν παραβιάσεις τις διορθώνουμε με περιστροφές.
 - Αν το δεξί παιδί του προς διαγραφή κόμβου έχει αριστερό παιδί, τότε αντικαθιστούμε το κόμβο με τον επόμενο του στην ενδοδιατεταγμένη διάταξη, δηλαδή με το πιο αριστερό κόμβο του υποδέντρου αυτού. Υπολογίζουμε τα ύψη και τις υποζυγίσεις και τυχόν παραβιάσεις τις διορθώνουμε με περιστροφές.

Τέλος, χρειάζεται να αναφερθεί ότι έχει υλοποιηθεί συνάρτηση η οποία διαγράφει ολόκληρο το δέντρο. Αυτό θα μας φανεί χρήσιμο στην περίπτωση που ο χρήστης ολοκληρώσει τις ενέργειες στον συγκεκριμένο κομμάτι του προγράμματος και θέλει να στραφεί σε κάποιο άλλο, καθώς αποδεσμεύουμε τον χώρο που καταλαμβάνει το AVL.

AVL by temperature

Κατασκευάζοντας το AVL by temperature αξιοποιήσαμε και τροποποιήσαμε αναλόγως τον κώδικα του ερωτήματος [\(1\)](#). Αναλυτικότερα, η δόμηση του AVL γίνεται με βάση τις θερμοκρασίες, με την διαφορά ότι σε πολλές περιπτώσεις τυγχάνει να έχουμε παρόμοιες θερμοκρασίες με διαφορετικές ημερομηνίες με αποτέλεσμα να δημιουργείτε πρόβλημα μιας και το AVL δέντρο αποτελεί δυαδικό δέντρο αναζήτησης και δεν μπορεί να χειριστεί τέτοιες καταστάσεις. Για την αντιμετώπιση του προβλήματος προσπαθήσαμε σε κάθε κόμβο node να δημιουργήσουμε μια λίστα η οποία θα περιέχει τις ημερομηνίες στοιχείων των οποίων η θερμοκρασία έχει παρόμοια τιμή με τον κόμβο που έχει ήδη εισαχθεί στο δέντρο. Με άλλα λόγια, τα στοιχεία τοποθετούνται με τους γνωστούς κανόνες σειριακά στο AVL δέντρο και όταν συναντάμε μια θερμοκρασία η οποία υπάρχει ήδη σαν κόμβος, προσθέτουμε την επιπλέον ημερομηνία στην λίστα του ήδη υπάρχοντα κόμβου.

Αξιοποιώντας τα παραπάνω δεδομένα, ξεκινήσαμε κατασκευάζοντας τα κατάλληλα structs που θα μας επιτρέψουν να φτιάξουμε κόμβους με λίστες. Δημιουργήσαμε ένα struct με όνομα `table_data_T` το οποίο περιέχει την θερμοκρασία ενός κόμβου (`double T_DegC`) και ένα struct τύπου `struct tm` με όνομα `"d"` όπου αποθηκεύεται η ημερομηνία. Ένα struct με όνομα `dateList` το οποίο αποτελείται από δύο επιπλέον struct, ένα struct τύπου `struct tm` με όνομα `"date"` που περιέχει την ημερομηνία του κόμβου και ένα struct τύπου `struct dateList*` με όνομα `"next"` το οποίο δείχνει στον επόμενο κόμβο. Ένα struct με όνομα `table_data_by_temp` το οποίο αποτελείται επίσης από μια θερμοκρασία και ένα struct τύπου `struct dateList *` και όνομα `"date"`. Και τέλος το struct `Node_T` το οποίο είναι το βασικό struct κάθε κόμβου για αυτό και αποτελείται από δύο παρόμοια struct `"left"` και `"right"` που δείχνουν τους επόμενους κόμβους αριστερά και δεξιά αντιστοίχως, από το ύψος του κόμβου και από ένα struct τύπου `struct table_data_by_temp` και όνομα `key`. Αυτό που καταφέρνουμε με τα struct αυτά είναι να δημιουργήσουμε κόμβους οι οποίοι πέρα από το δεξί παιδί, αριστερό παιδί και το ύψος, περιέχουν συγχρόνως και την θερμοκρασία και την ημερομηνία. Το struct `table_data_T` το χρησιμοποιούμε ως όρισμα στις συναρτήσεις `insertion_temp` (που καλούμε την `newNode`) και `newNode_T` για να αρχικοποιήσουμε τον νέο κόμβο με τις ανάλογες τιμές θερμοκρασίας και ημερομηνίας που του αντιστοιχούν.

Όσον αφορά την διαφοροποίηση της συνάρτησης `insert_temp` στο δέντρο AVL by temperature και της συνάρτησης `insert` του δέντρου AVL by date είναι ότι στην `insert_temp` προστέθηκε ο έλεγχος όπου ο τωρινός κόμβος έχει ίδια ημερομηνία με την ημερομηνία μας. Στην περίπτωση αυτή, το προσωρινό struct `temp` τύπου `DL *` (που ορίσαμε μέσα στην συνάρτηση

`insert_temp`) αρχικοποιείται με τις τιμές του `key` (το οποίο είναι όρισμα της συνάρτησής μας, με τύπο `table_data_T`), δηλαδή στη λίστα που φτιάξαμε με τις ημερομηνίες του τωρινού κόμβου αποθηκεύεται η νέα ημερομηνία του `key` που έχει την ίδια θερμοκρασία με τον τωρινό κόμβο. Οι υπόλοιπες λειτουργίες της `insertion_temp` λειτουργούν ακριβώς όπως αυτές της `insert` στο προηγούμενο ερώτημα.

Τέλος, διαφοροποιήθηκε ελάχιστα και η συνάρτηση `new_node_T` η οποία, επειδή πρέπει να αρχικοποιούμε και την ημερομηνία του κόμβου με μια τιμή (πέρα από την θερμοκρασία) θα πρέπει να πάρει το ανάλογο όρισμα, δηλαδή ένα `struct key` τύπου `table_data_T`.

Hashing

Ο κατακερματισμός είναι μια τεχνική μέσω της οποίας αντιστοιχίζουμε κλειδιά σε συγκεκριμένες θέσεις ενός πίνακα, μέσω μιας συνάρτησης κατακερματισμού. Στην περίπτωση μας, χρησιμοποιούμε κατακερματισμό με αλυσίδες. Σε κάθε θέση του πίνακα αντιστοιχίζεται μια λίστα, ώστε να αποθηκευτούν όσα στοιχεία χρειάζεται να εισαχθούν σε εκείνη τη θέση. Μέσω hashing επιτυγχάνουμε γρηγορότερη πρόσβαση σε στοιχεία. Η συνάρτηση κατακερματισμού θα υπολογίζεται ως το υπόλοιπο (modulo) της διαίρεσης του αθροίσματος των κωδικών ASCII των επιμέρους χαρακτήρων που απαρτίζουν την ημερομηνία με τον αριθμό των buckets. Για το λόγο αυτό δημιουργούμε μια συνάρτηση την `hash_function_hashing()` στην οποία ως όρισμα δίνουμε την ημερομηνία και μας επιστρέφει το αποτέλεσμα της παραπάνω πράξης.

Ένας πίνακας κατακερματισμού έχει κάποιες βασικές κύριες λειτουργίες :

- **Αναζήτηση** – Αναζητά ένα στοιχείο σε έναν πίνακα κατακερματισμού.
- **Εισαγωγή** – εισάγει ένα στοιχείο σε έναν πίνακα κατακερματισμού.
- **διαγραφή** – Διαγράφει ένα στοιχείο από έναν πίνακα κατακερματισμού.

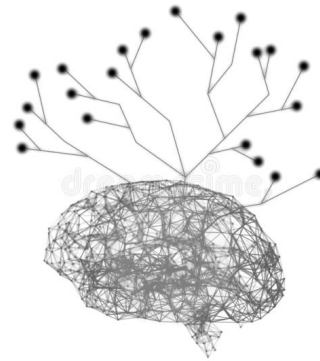
Ας ξεκινήσουμε με την εισαγωγή. Κάθε φορά που πρόκειται να εισαχθεί ένα στοιχείο υπολογίζουμε το hash code του κλειδιού και τοποθετούμε το στοιχείο, μέσω της `hash_function_hashing()`, στο κατάλληλο index του πίνακα. Συγκεκριμένα, στην συνάρτηση `insert_hashing()`, αφού ελέγχουμε ότι η θέση του πίνακα του κάθε κλειδιού είναι κενή, δεσμεύουμε το χώρο αυτό και τοποθετούμε το στοιχείο μας. Αλλιώς, το στοιχείο μας μπαίνει ως το επόμενο στοιχείο σε μία λίστα. Πρέπει, όμως, να εισάγουμε τα στοιχεία μας από το ocean file. Δημιουργούμε την συνάρτηση `insert_from_file_hashing()`. Σε αυτή ανοίγουμε το αρχείο, το διαβάζουμε και μέσω της `hash_function_hashing()` υπολογίζουμε το κλειδί και καλούμε την `insert_hashing()` με ορίσματα τον τύπο των στοιχείων μας(data), το κλειδί της ημερομηνίας που διαβάσαμε και τον πίνακα μας.

Τώρα, στην αναζήτηση υπολογίζουμε την συνάρτηση κατακερματισμού($h(x)$) και στη συνέχεια βρίσκουμε το στοιχείο στη λίστα που υποδεικνύει ο δείκτης $T(h(x))$, όπου T ο πίνακας κατακερματισμού, μέσω της συνάρτησης `search_hashing()`. Ο χρήστης ,μέσω της `user_search_hashing()`, εισάγει την ημερομηνία που θέλει να αναζητήσει και με την βοήθεια της παραπάνω συνάρτησης βρίσκουμε το στοιχείο που ψάχνουμε, άρα και την θερμοκρασία του.

Η διαγραφή επιτυγχάνεται με την βοήθεια της `search`. Αφού, βρούμε το στοιχείο x από την λίστα που υποδεικνύει ο δείκτης $T(h(x))$, το διαγράφουμε από την λίστα. Ο

χρήστης δίνει ημερομηνία, εκτελείται αναζήτηση και αν βρεθεί, τότε ο δείκτης `*next` του προηγούμενου στοιχείου δείχνει στο επόμενο αυτού που βρήκαμε και ο δείκτης `*prev` του επόμενου του `x` δείχνει στο προηγούμενο στοιχείο από το `x`. Έγινε, δηλαδή, η διαγραφή, μέσω της συνάρτησης `delete_node_hashing()`.

Ακόμη, επιθυμούμε να τροποποιήσουμε τα στοιχεία εγγραφής βάσει ημερομηνίας που θα δίνεται από το χρήστη. Η τροποποίηση προφανώς θα αφορά μόνο το πεδίο της θερμοκρασίας. Αφού αποθηκεύσουμε την ημερομηνία, την περνάμε ως όρισμα στην `search_hashing()` και αν βρεθεί, ζητάμε μια νέα τιμή θερμοκρασίας από τον χρήστη. Τέλος, εισάγουμε την νέα τιμή στο αποτέλεσμα της αναζήτησης.



Πηγές

Βιβλιογραφικές

[A robust variation of interpolation search](#) by Warren Burton, Gilbert N. Lewis

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ by Athanasios K. Tsakalidis

Διαδικτυακές

www.geeksforgeeks.org

stackoverflow.com

wikipedia.org