Project DS - Part 1 2021 - 2022

Ονοματεπώνυμο : Χρήστος Ψαθογιαννάκης
Αριθμός Μητρώου (ΑΜ) : 1090052
Email : up1090052@upnet.gr

Ονοματεπώνυμο : Μάριος Ζάγκλας
Αριθμός Μητρώου (ΑΜ) : 1084686
Email : up1084686@upnet.gr

Ονοματεπώνυμο : Μελιτσόπουλος Φώτιος
Αριθμός Μητρώου (ΑΜ) : 1084600
Email : up1084600@upnet.gr

Γλώσσα προγραμματισμού : C++

Μέρος Α:

Για να ξεκινήσουμε το πρώτο μέρος θα χρειαστούμε να καλέσουμε το csv αρχείο κάπως, οπότε δοκιμάσαμε διάφορους τρόπους όπως δισδιάστατους vectors αλλά στο τέλος το κάναμε με 8 μονοδιάστατους vectors ξεχωρίζοντας την κάθε στήλη μέσω του regex.

Regex Expression: (?:,|\n|^)([^,\n]*|(?:\n|$))
(?:,|\n|^) # all values must start at the beginning of the file, # the end of the previous line, or at a comma (# single capture group for ease of use; CSV can be... [^,\n] # anything excluding commas (,), or newlines (\n) * # in any order and any number of times | # ...or (B) a single newline or end-of-file character, # used to capture empty values at the end of (?:\n|$) # the file or at the ends of lines)

Διαβάζουμε το αρχείο χρησιμοποιώντας την συνάρτηση **readFile()**.

readFile.cpp
Δημιουργούμε ένα struct data για τα δεδομένα μας. Διαβάζουμε το αρχείο .csv και τα αποθηκεύουμε στις μεταβλητές τύπου data, ώστε να μπορούμε να τις επιστρέψουμε μαζί στο τέλος της μεθόδου.

Αφού διαβάσαμε το αρχείο ξεκινήσαμε τα πρώτα κομμάτια του πρώτου μέρους.
Όπως θα παρατηρήσετε η άσκηση έχει υλοποιηθεί με συναρτήσεις οι οποίες καλούνται από τις αντίστοιχες συναρτήσεις menu.

menuSorting.cpp
Εκτυπώνουμε τις επιλογές του χρήστη. Και καλούμε τις ανάλογες μεθόδους ταξινόμησης.

Η σκεπτική είναι να κρατήσουμε καθαρή την main για να έχουμε μεγαλύτερη ευκολία στο debugging και στον έλεγχο, καλώντας την κάθε συνάρτηση ξεχωριστά από το υπόλοιπο πρόγραμμα.
Ωστόσο, διαπιστώσαμε στην συνέχεια πως αυτό είναι δύσκολο να γίνει, λόγο του **circular include problem**, του οποίου δημιουργείται όταν μια συνάρτηση καλεί μια άλλη και καλείται ταυτόχρονα και από αυτή μέσω των **#include**.
Περιορίσαμε το πρόβλημα με **include guards** και **forward declaration**, αλλά καταλήξαμε στο να κρατούμε την ανεξαρτησία των συναρτήσεων στο ελάχιστο. Για την καθαρή εκτέλεση του προγράμματος θα πρέπει να τρέξει πρώτα το menu.cpp, το οποίο διαδοχικά θα καλέσει και τις ανάλογες συναρτήσεις.

Το Insertion Sort το χρησιμοποιούμαι για την ταξινόμηση όλων των vector ανά ημερομηνία ή οποιοδήποτε από τα στοιχεία που διαβάσαμε από το .csv αρχείο για την υλοποίηση των ερωτημάτων 3 κ 4.

insertionSort.cpp
Δεχόμαστε ως παραμέτρους τα στοιχεία που διαβάσαμε από το .csv αρχείο Έχουμε υλοποίηση την συνάρτηση 2 φορές, έτσι ώστε να έχουμε την δυνατότητα να ταξινομήσουμε όλα τα στοιχεία με βάση είτε της vector<string> ημερομηνίας ή μέσω τον υπόλοιπων vector<float> στοιχείων από το αρχείο που διαβάσαμε. Η συγκεκριμένη συνάρτηση έχει γραφτεί με τρόπο ο οποίος μας επιτρέπει να ταξινομήσουμε πολλά στοιχεία βάση ενός συγκεκριμένου, οπότε δεν ήταν δυνατές οι λύσεις οι οποίες χρησιμοποιούσαν μια εξτρά μεταβλητή για την ανταλλαγή στοιχείων λόγο του όγκου ο κώδικάς αυτός θα χρειαζόταν για την υλοποίηση του, καταλήξαμε στην υλοποίηση της insertion μέσω της μεθόδου swap().

Η Quicksort.cpp αποτελείται από 2 συναρτήσεις, την partition() και την Quicksort().

Quicksort.cpp
Η partition φτιάχνει το pivot να μετακινείται και να γίνονται οι συγκρίσεις ενώ παράλληλα κουβαλάει και τα dates μαζί με τις θερμοκρασίες που συγκρίνονται. Ενώ στην συνάρτηση quicksort απλά βάζουμε τα μέρη που κάνουμε quicksort σαν αλγόριθμο για να γίνει σωστά η σύγκριση και στην συνέχεια την καλούμε στην main.

Για να τα συγκρίνουμε κάνουμε την συνάρτηση από την βιβλιοθήκη chrono και μετράμε το χρόνο αν και κάποιες φορές δημιουργεί θέμα οπότε καλό είναι να γίνουν αρκετές φορές run για να βγάζει κανονικούς χρόνους.
Αυτό που παρατηρούμε είναι ότι η insertion sort έχει τρομερά μεγαλύτερο χρόνο από την quicksort που είναι λογικό γιατί η ύπαρξη της quicksort είναι να γίνεται ο κώδικας πιο γρήγορα και για αυτό αρκετοί προγραμματιστές την προτιμούν.

Για το ερώτημα 2 για αρχή φτιάχνουμε το heap sort που δημιουργούμε 3 συναρτήσεις

heapSort.cpp
Το max_heapify που βρίσκει το μέγιστο και κουβαλάει και το string του dates όπως κάναμε και προηγουμένως. Η build_max_heap που δείχνει απλά την θέση και την heapsort που οργανώνει όλα ώστε να γίνει η ταξινόμηση. Και τέλος την heapSort η οποία με την βοήθεια των 2 παραπάνω συναρτήσεων υλοποιεί την ταξινόμηση των δεδομένων.

Με το counting sort φτιάχνουμε μια συνάρτηση που είναι ίδιας λογικής με τους προηγούμενους αλγόριθμους που υλοποιήσαμε.

countSort.cpp
Γνωρίζουμε ότι για να γίνει counting sort πρέπει να είναι ακέραιος ο πίνακας οπότε φτιάξαμε ένα array που να κάνει τις θερμοκρασίες * 100 και να είναι int οπότε το ταξινομούμε αυτό και στην συνέχεια βάζουμε μια ισότητα ότι η θερμοκρασία είναι το array / 100 οπότε παραμένουν οι δεκαδικοί ενώ είναι ταξινομημένη η στήλη.

Συγκρίνοντας τους 2 χρόνους παρατηρούμε ότι η counting sort είναι πιο γρήγορη από την heap sort γιατί η heap sort έχει μεγαλύτερη κλίμακα αριθμών ενώ η counting περιορισμένη.

Για τα ερωτήματα 3 και 4 ομοίως δημιουργήσαμε μια συνάρτηση menuSearching:

menuSearching.cpp
Εκτυπώνουμε τις επιλογές του χρήστη. Και καλούμε τις ανάλογες μεθόδους αναζήτησης.

Από την οποία τρέχουν οι υπόλοιπες συναρτήσεις.

binSearch.cpp
Η μέθοδος binary search, παίρνει ως παραμέτρους μία μεταβλητή τύπου string στην οποία περνάμε την ημερομηνία που θέλουμε να αναζητήσουμε μέσω του κυρίως προγράμματος και την μετατρέπουμε σε integer μέσω της μεθόδου vecstoi που φτιάξαμε, μια vector<int> μεταβλητή, η οποία περιέχει τις ημερομηνίες του αρχείου ocean.csv, αλλά σε μορφή ακεραίου, έτσι ώστε να μην χρειάζεται να κάνουμε συγκρίσεις με αλφαριθμητικούς ή περαιτέρω μετατροπές μέσα στο πρόγραμμα. Τέλος, 2 μεταβλητές τύπου int για τα περιθώρια στα οποία θα γίνει η αναζήτηση. Η υλοποίηση έγινε με την recursive μέθοδο, στην οποία υπολογίζουμε την μέση θέση, και αναλόγως καλούμε τον αλγόριθμο recursively.

interpolationSearch.cpp
Η μέθοδος interpolationSearch, παίρνει ως παραμέτρους μεταβλητή string και vector<int> όπως στην binSearch.cpp. Είναι παρόμοια με την Binary Search, αλλά αντί να υπολογίζει την μέση θέση της λίστας, υπολογίζει την επόμενη πιθανή θέση της ημερομηνίας που αναζητάμε. Οπότε αντιθέτως με την Binary Search που κάθε φορά θα μειώσει τον όγκο αναζήτησης κατά μισό. Η Interpolation Search μπορεί να μας μειώσει τον όγκο κατά μόνο 1 στην χειρότερη περίπτωση. Δίνοντας μας χειρότερη περίπτωση O(n) για n αντικείμενα.

BIS.cpp
Η μέθοδος Binary Interpolation Search (BIS), παίρνει τους ίδιους παραμέτρους όπως η Interpolation Search. Αλλά σε αντίθεση με την Interpolation Search όπου έκανε βήματα +1,-1. Η BIS κάνει βήματα χρησιμοποιώντας ένα μετρητή **i=0**, $\mathrm{i} * \sqrt{n}$, με τον μετρητή **i** να αυξάνεται κατά +1 κάθε βήμα.

BIS2.cpp
Η μέθοδος της Exponential Binary Interpolation Search, ομοίως με τους πρόγονους της, παίρνει τους ίδιους παραμέτρους. Αλλά σε αντίθεση με την BIS, χρησιμοποιούμε τον μετρητή **i** ως εκθέτη για το 2 υλοποιώντας την εξίσωση 2^i η οποία είναι ο καινούργιος μας πολλαπλασιαστής, ώστε να κάνουμε εκθετικά μεγάλα βήματα $2^i\sqrt{n}$
menuPart1.cpp
Κάνουμε include τις μεθόδους για τα κομμάτια Sorting και Searching του πρότζεκτ Και υλοποιούμε την μέθοδο exportCSV ώστε να δημιουργήσουμε καινούργιο αρχείο .csv με τα περιεχόμενα ταξινομημένα ανά ημερομηνία.

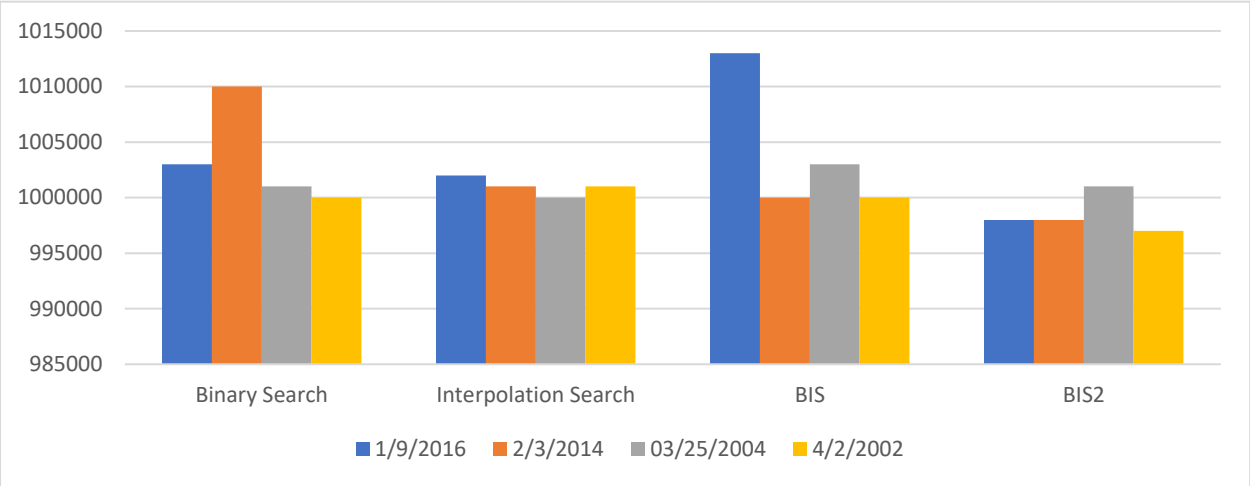
Vecstoi.cpp
Μετατρέπουμε δεδομένα τύπου vector<string> σε δεδομένα τύπου vector<int>. Επίσης μετατρέπουμε δεδομένα τύπου string σε δεδομένα τύπου int. Και τα επιστρέφουμε.

Οι μέθοδοι χρησιμοποιούνται για τις μετατροπές των ημερομηνιών, για την υλοποίηση των ανάλογων μεθόδων που δεν δουλεύουν με strings.

printData.cpp
Η μέθοδος δέχεται τις μεταβλητές που δημιουργήθηκαν από το διάβασμα του αρχείου και τις εκτυπώνει στην οθόνη.

Σχετικά με τον χρόνο υλοποίησης στους αλγόριθμους αναζήτησης, συγκρίνοντας τους χρόνους, μπορούμε να δούμε ότι ο κάθε αλγόριθμος διαδοχικά φέρνει πιο γρήγορα αποτελέσματα.

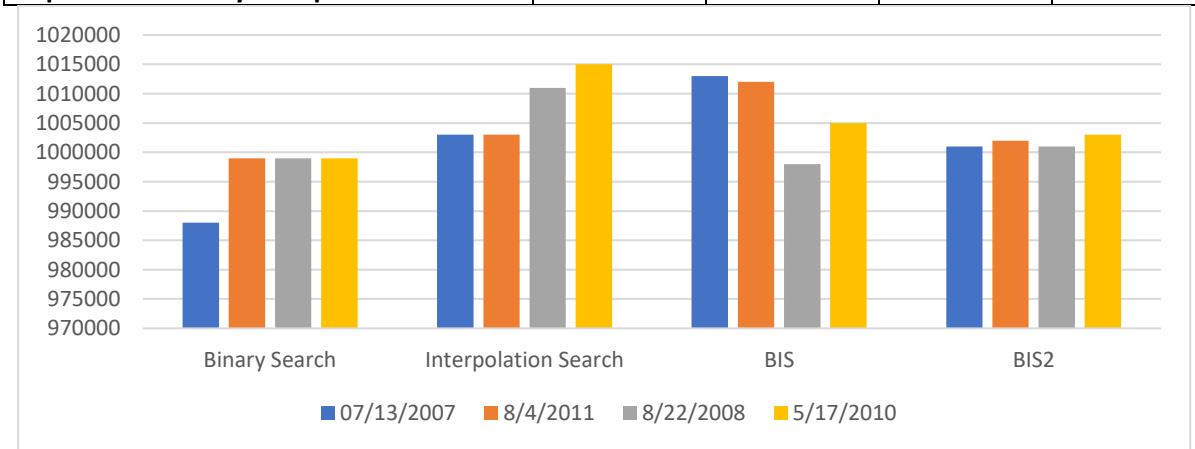
Searching Algorithms	Random Dates			
	01/09/2016	2/3/2014	03/25/2004	04/02/2002
Binary Search	1003000ns	1010000ns	1001000ns	1000000ns
Interpolation Search	1002000ns	1001000ns	1000000ns	1001000ns
Binary Interpolation Search	1013000ns	1000000ns	1003000ns	1000000ns
Exponential Binary Interpolation Search	998000ns	998000ns	1001000ns	997000ns



Αλλά τα αποτελέσματα και στους interpolation αλγορίθμους αναζήτησης, εξαρτούνται από την κατανομής της λίστας

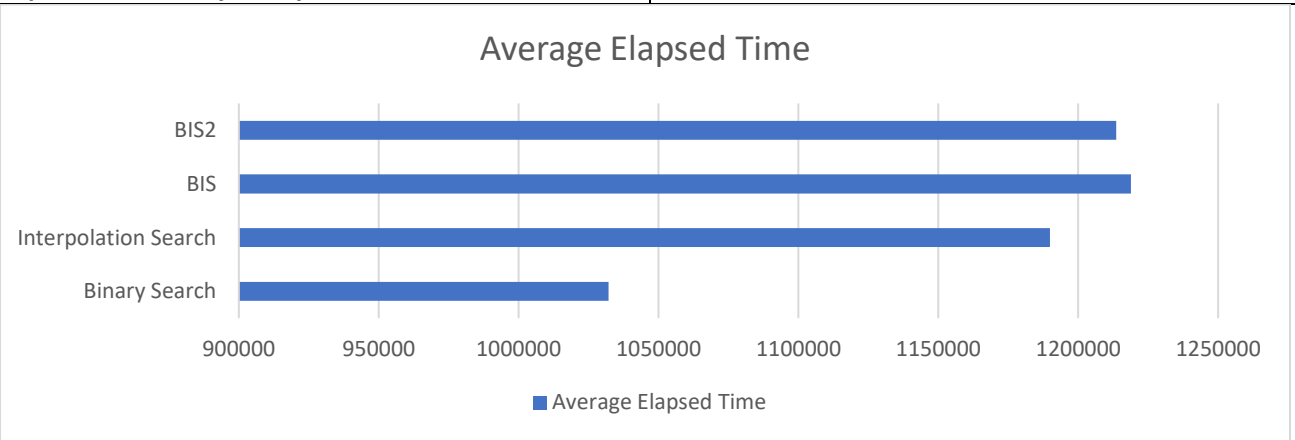
Για παράδειγμα, οπότε έχουμε περιπτώσεις όπου αν πάρουμε μια ημερομηνία κοντά στην μέση της λίστας, η δυαδική αναζήτηση μπορεί να είναι εξίσου γρήγορη ή και πιο γρήγορη.

Searching Algorithms	Random Dates			
	07/13/2007	08/04/2011	08/22/2008	05/17/2010
Binary Search	988000ns	999000ns	999000ns	999000ns
Interpolation Search	1003000ns	1003000ns	1011000ns	1015000ns
Binary Interpolation Search	1013000ns	1012000ns	998000ns	1005000ns
Exponential Binary Interpolation Search	1001000ns	1002000ns	1001000ns	1003000ns



Επίσης, η χρήση των αλγορίθμων Interpolation Search στην καλύτερη περίπτωση μπορεί να μας δίνουν χρόνο **log(log n),** αλλά στην χειρότερη έχουν χρόνο **0(n)** θέτοντας τους σε μειονέκτημα όταν δεν έχουμε ομοιόμορφη κατανομή.

Searching Algorithms	Average Elapsed Time
Binary Search	**1032101ns**
Interpolation Search	**1189842ns**
Binary Interpolation Search	**1218823ns**
Exponential Binary Interpolation Search	**1213524ns**



Μέρος Β:

Thank you Teacher,
But our Part 2 is in another castle!