Κ23α - Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2020 – 2021

Καθηγητής Ι. Ιωαννίδης

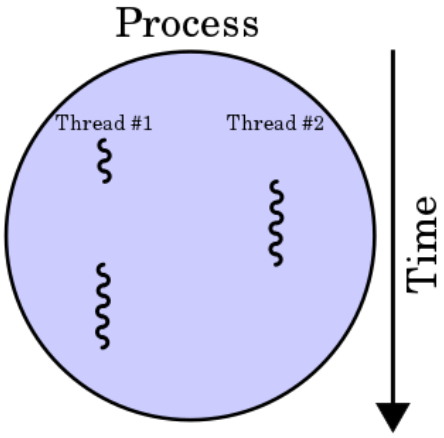
Άσκηση 1 – Παράδοση: Κυριακή 24 Ιανουαρίου 2021

Περιγραφή Λειτουργικότητας και Δομών

Στο 3ο μέρος της εργασίας, θα εισαχθεί παραλληλία με στόχο τη μείωση του συνολικού χρόνου εκτέλεσης. Η παραλληλία θα εφαρμοστεί μέσω πολυνηματισμού κατά τη διαδικασία εκπαίδευσης του μοντέλου (training), αλλά και κατά τη διάρκεια ελέγχου (testing). Στην επόμενη ενότητα θα γίνει μια σύντομη αναφορά στις βασικές τεχνικές πολυνηματισμού.

Πολυνηματισμός

Ένα νήμα είναι μια ελαφριά διεργασία. Η υλοποίηση των νημάτων και των διεργασιών διαφέρει από το ένα λειτουργικό σύστημα στο άλλο. Σε κάθε διεργασία υπάρχει τουλάχιστον ένα νήμα. Αν μέσα σε μια διεργασία υπάρχουν πολλαπλά νήματα, τότε αυτά μοιράζονται την ίδια μνήμη και πόρους αρχείων.



Σχήμα 1. Παράδειγμα εκτέλεσης πολυνηματικής εργασίας

Ένα νήμα διαφέρει από μια διεργασία ενός πολυεπεξεργαστικού λειτουργικού συστήματος στα εξής:

- οι διεργασίες είναι τυπικώς ανεξάρτητες, ενώ τα νήματα αποτελούν υποσύνολα μιας διεργασίας.

- οι διεργασίες περιέχουν σημαντικά περισσότερες πληροφορίες κατάστασης από τα νήματα, ενώ πολλαπλά νήματα μιας διεργασίας μοιράζονται την κατάσταση της διεργασίας, όπως επίσης μνήμη και άλλους πόρους.
- οι διεργασίες έχουν ξεχωριστούς χώρους διευθυνσιοδότησης, ενώ τα νήματα μοιράζονται το σύνολο του χώρου διευθύνσεων που τους παραχωρείται
- η εναλλαγή ανάμεσα στα νήματα μιας διεργασίας είναι πολύ γρηγορότερη από την εναλλαγή ανάμεσα σε διαφορετικές διεργασίες.

Η πολυνημάτωση αποτελεί ένα ευρέως διαδεδομένο μοντέλο προγραμματισμού και εκτέλεσης διεργασιών το οποίο επιτρέπει την ύπαρξη πολλών νημάτων μέσα στα πλαίσια μιας και μόνο διεργασίας. Τα νήματα αυτά μοιράζονται τους πόρους της διεργασίας και μπορούν να εκτελούνται ανεξάρτητα. Το γεγονός ότι επιτρέπει μια διεργασία να εκτελείται παράλληλα σε ένα σύστημα με πολλαπλούς πυρήνες αποτελεί ίσως την πιο ενδιαφέρουσα εφαρμογή της συγκεκριμένης τεχνολογίας.

Το πλεονέκτημα ενός πολυνηματικού προγράμματος είναι ότι του επιτρέπει να εκτελείται γρηγορότερα σε υπολογιστικά συστήματα που έχουν πολλούς επεξεργαστές, επεξεργαστές με πολλούς πυρήνες, ή κατά μήκος μιας συστοιχίας υπολογιστών. Για να μπορούν να χειραγωγηθούν σωστά τα δεδομένα, τα νήματα θα πρέπει ορισμένες φορές να συγκεντρωθούν σε ένα ορισμένο χρονικό διάστημα έτσι ώστε να επεξεργαστούν τα δεδομένα στη σωστή σειρά.

Ένα ακόμα πλεονέκτημα της πολυδιεργασίας (και κατ' επέκταση της πολυνημάτωσης) ακόμα και στους απλούς επεξεργαστές (με έναν πυρήνα επεξεργασίας) είναι η δυνατότητα για μια εφαρμογή να ανταποκρίνεται άμεσα. Έχοντας ένα πρόγραμμα που εκτελείται μόνο σε ένα νήμα, αυτό θα φαίνεται ότι κολλάει ή ότι παγώνει, στις περιπτώσεις που εκτελείται κάποια μεγάλη διεργασία που απαιτεί πολύ χρόνο. Αντίθετα σε ένα πολυνηματικό σύστημα, οι χρονοβόρες διεργασίες μπορούν να εκτελούνται παράλληλα με άλλα νήματα που φέρουν άλλες εντολές για το ίδιο πρόγραμμα, καθιστώντας το άμεσα ανταποκρίσιμο.

Για την υλοποίηση των πολυνηματικών λειτουργιών θα χρησιμοποιήσετε τη Pthreads που είναι μια POSIX API C βιβλιοθήκη. Για τη χρήση της πρέπει να συμπεριλαμβάνεται στα αρχεία του κώδικα σας το `<pthread.h>` και να χρησιμοποιήσετε κατά το compilation το "- pthread" flag.

Οι βασικές ρουτίνες των PThreads είναι οι εξής:

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(*start_routine) (void), void *arg)`
- `void pthread_exit(void *value_ptr)`
- `int pthread_join(pthread_t thread, void **value_ptr)`

Ο στόχος της ρουτίνας `pthread_create()` είναι να δημιουργήσει ένα νέο νήμα και αφού αρχικοποιήσει τα χαρακτηριστικά αυτού, το κάνει διαθέσιμο για χρήση. Στη μεταβλητή `thread` επιστρέφετε το αναγνωριστικό του νήματος που μόλις δημιουργήθηκε. Με βάση την τιμή στο

πεδίο `attr` θα αρχικοποιηθούν τα χαρακτηριστικά του νέου νήματος. Στο όρισμα `start_routine` ορίζεται η ρουτίνα που θα εκτελέσει το νέο νήμα όταν δημιουργηθεί και στο πεδίο `arg` θα οριστούν οι παράμετροι αυτής.

Η ρουτίνα `pthread_exit()` θα τερματίσει ένα ήδη υπάρχον νήμα και θα αποθηκεύσει την κατάσταση τερματισμού για όσα άλλα νήματα θα προσπαθήσουν να συνενωθούν με αυτό. Επιπρόσθετα ελευθερώνει όλα τα δεδομένα του νήματος, συμπεριλαμβανομένων και της στοίβας του νήματος. Είναι σημαντικό τα αντικείμενα συγχρονισμού νημάτων ,όπως τα mutexes και οι μεταβλητές κατάστασης (condition variables), που κατανέμονται στη stack του νήματος, να καταστραφούν πριν την κλήση της ρουτίνας `pthread_exit()`.

Η ρουτίνα `pthread_join()` μπλοκάρει το τρέχον νήμα μέχρι να τερματίσει το νήμα που προσδιορίζεται από το πεδίο `thread`. Η κατάσταση τερματισμού του νήματος αυτού επιστρέφεται στο πεδίο `value_ptr`. Αν το συγκεκριμένο νήμα έχει ήδη τερματίσει (και δεν είχε προηγουμένως αποσπαστεί), το τρέχον νήμα δεν θα μπλοκαριστεί.

Συγχρονισμός

Όταν δυο threads χρησιμοποιούν τις ίδιες δομές θα πρέπει να συγχρονίζεται η πρόσβαση σε αυτές, ώστε να εξασφαλιστεί η συνέπεια στο αποτέλεσμα των ενεργειών που εκτελούν τα threads. Η POSIX παρέχει τους mutexes.
Οι mutexes έχουν δύο καταστάσεις locked, unlocked. Χρησιμοποιώντας ένα mutex ανά κοινή δομή, μπορούμε να περιορίσουμε την πρόσβαση σε αυτή σε ένα μόνο thread κάθε στιγμή.

Οι βασικές ρουτίνες των POSIX mutexes είναι οι εξής:

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)`
- `int pthread_mutex_lock(pthread_mutex_t *mutex)`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex)`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex)`

Όσο κάποιο νήμα έχει κλειδωμένο ένα mutex τότε αν κάποιο άλλο νήμα προσπαθήσει να κλειδώσει τον ίδιο mutex, το δεύτερο νήμα θα “κολλήσει” μέχρι να ξεκλειδωθεί. Άρα αν η πρόσβαση στις κοινές δομές γίνεται ανάμεσα σε κλείδωμα και ξεκλείδωμα του αντίστοιχου mutex αποφεύγουμε τις ταυτόχρονες αλλαγές και τα λάθη που μπορεί να προκαλέσουν.

Υλοποίηση

Η χρήση των νημάτων μπορεί να γίνει με δύο τρόπους είτε δημιουργώντας καινούργια νήματα για κάθε παράλληλο κομμάτι, είτε με την υλοποίηση ενός job scheduler. Αν και η δεύτερη επιλογή είναι πιο σύνθετη σε επίπεδο υλοποίησης, είναι συνήθως προτιμότερη ειδικά σε εφαρμογές, όπου η παράλληλη επεξεργασία γίνεται ασύγχρονα κατά τη διάρκεια εκτέλεσης του προγράμματος και φυσικά αποτρέπει την πολλαπλή δημιουργία των δομών των νημάτων. Για τους παραπάνω λόγους θα υλοποιήσετε ένα job scheduler για την παράλληλη επεξεργασία.

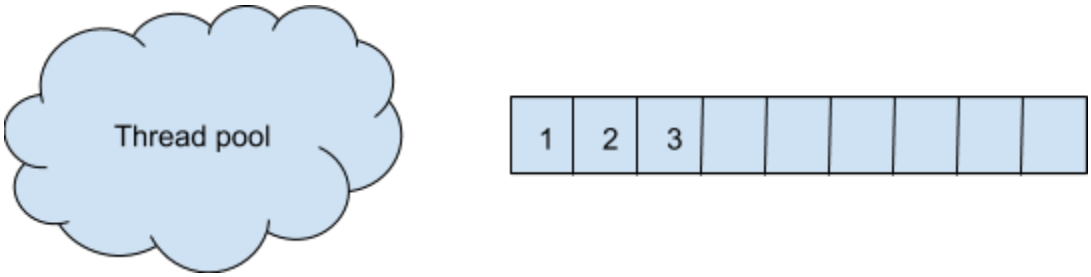
Job Scheduler

Εργασίες (Jobs)

Εργασία (Job) είναι μια ρουτίνα κώδικα η οποία θέλουμε να εκτελεστεί από κάποιο νήμα πιθανότατα παράλληλα με κάποια άλλη. Μπορούμε να ορίσουμε οτιδήποτε θέλουμε ως job και να το αναθέσουμε στον χρονοπρογραμματιστή.

Χρονοπρογραμματιστής Εργασιών (Scheduler) και Δεξαμενή Νημάτων (Thread Pool)

Ο χρονοπρογραμματιστής ουσιαστικά δέχεται δουλειές και αναλαμβάνει την ανάθεση τους σε νήματα, για προσωρινή αποθήκευση των εργασιών χρησιμοποιεί μια ουρά προτεραιότητας. Έστω ότι έχουμε μια Δεξαμενή νημάτων (thread pool) και μια συνεχόμενη ροή από ανεξάρτητες εργασίες (jobs). Όταν δημιουργείται μια εργασία, μπαίνει στην ουρά προτεραιότητας του χρονοπρογραμματιστή και περιμένει να εκτελεστεί. Οι εργασίες εκτελούνται με την σειρά που δημιουργήθηκαν (first-in-first-out - FIFO). Κάθε νήμα περιμένει στην ουρά μέχρι να του ανατεθεί μια εργασία και, αφού την εκτελέσει, επιστρέφει στην ουρά για να αναλάβει νέα εργασία. Για την ορθή λειτουργία ενός χρονοπρογραμματιστή είναι απαραίτητη η χρήση σημαφόρων στην ουρά, ώστε να μπλοκάρουν εκεί τα νήματα, και την κρίσιμη περιοχή (critical section), ώστε να γίνεται σωστά εισαγωγή και εξαγωγή εργασιών από την ουρά.



Σχήμα 2. Αναπαράσταση δομής job scheduler

Condition Variables

Ένα ακόμα εργαλείο συγχρονισμού που μπορεί να σας φανεί χρήσιμο είναι τα condition variables. Ένα condition variable είναι ένα εργαλείο που επιτρέπει στα POSIX threads να αναβάλουν την εκτέλεση τους μέχρι μια έκφραση να γίνει αληθής. Δύο είναι οι βασικές πράξεις πάνω σε ένα condition variable, `wait()` που αδρανοποιεί το νήμα που την κάλεσε και η `signal()` που ξυπνά ένα από τα νήματα που είναι απενεργοποιημένα πάνω στο ίδιο condition variable. Ένα condition variable χρησιμοποιείται σε συνδυασμό με ένα mutex.

Οι βασικές ρουτίνες των POSIX condition variables είναι οι εξής:

- `int pthread_cond_init(pthread cond)t *cond, pthread_condattr_t *cond attr)`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `int pthread_cond_signal(pthread_cond_t *cond)`
- `int pthread_cond_broadcast(pthread_cond_t *cond)`
- `int pthread_cond_destroy(pthread_cond_t *cond)`

Οι condition variables χρησιμοποιούνται μέσα σε while loops:

```
mutex_lock(mtx1)
while (canRun) {
      cond_var_wait(condvar1, mtx1)
}
//do some work
cond_var_signal(condvar1)
mutex_unlock(mtx1)
```

Πριν τη χρήση του `condvar1` κλειδώνουμε το `mtx1`. Μετά ελέγχουμε σε μια while μια συνθήκη που αν αποτιμηθεί ως αληθής σημαίνει ότι το νήμα δεν μπορεί να κάνει τη δουλειά του και πρέπει να αδρανοποιηθεί με τη χρήση της `cond_var_wait()`. Αλλιώς το νήμα δεν εισέρχεται στο εσωτερικό της while και εκτελεί τη δουλειά του. Χρησιμοποιεί την `cond_var_signal()` για να ξυπνήσει ένα ακόμα νήμα, αν υπάρχει, που είναι αδρανοποιημένο στο ίδιο condition variable και ξεκλειδώνει το `mtx1`.

Ενσωμάτωση δομής Job Scheduler

Στην αρχή του προγράμματος θα δημιουργείται η δομή του JobScheduler, η οποία αποθηκεύει όλα τα jobs, διαχειρίζεται την εκτέλεσή τους θα διατηρείται σε όλη τη διάρκεια εκτέλεσης του προγράμματος και θα καταστρέφεται στο τέλος.

```
struct JobScheduler{
      int execution_threads; // number of execution threads
      Queue* q; // a queue that holds submitted jobs / tasks
      p_thread_t* tids; // execution threads
      …
      // mutex, condition variable, ...
};
```

Εκτός της δομής του scheduler απαιτείται και μια δομή Job, η οποία καθορίζει ουσιαστικά τον κώδικα της εργασίας που εκτελείται παράλληλα. Η δομή αυτή πρέπει να μπορεί να υποστηρίζει διαφορετικούς τύπους εργασιών, τις οποίες εκτελεί ο scheduler παράλληλα, χωρίς να γνωρίζει τη λειτουργικότητά τους.

Οι βασικές λειτουργίες που πρέπει να υλοποιεί η δομή του scheduler είναι οι εξής
`JobScheduler* initialize_scheduler(int execution_threads);`

```
int submit_job(JobScheduler* sch, Job* j);
int execute_all_jobs(JobScheduler* sch);
int wait_all_tasks_finish(JobScheduler* sch);
int destroy_scheduler(JobScheduler* sch)
```

Υλοποίηση μοντέλου μηχανικής μάθησης

Για το 3ο μέρος της εργασίας θα υλοποιηθεί ως εξής το μοντέλο μηχανικής μάθησης. Αφού έχουν αναλυθεί και διανυσματοποιηθεί τα αρχεία json, θα γίνει η μείωση των διαστάσεων στις 1000 σημαντικότερες. Θα διατηρηθούν, δηλαδή, οι λέξεις/διαστάσεις με το υψηλότερο μέσο tf-idf.
Το dataset W θα αναλυθεί με τη σειρά του και θα παραχθούν οι αντίστοιχες κλίκες καθώς και οι αρνητικές συσχετίσεις κλικών. Το τελικό αποτέλεσμα που θα παραχθεί είναι το σύνολο των σχέσεων μεταξύ αντικειμένων και τιμή 1 ή 0 (είναι ή δεν είναι όμοια προϊόντα).
Από αυτές τις γραμμές θα επιλεγεί με τυχαίο τρόπο το 60% ώστε να αποτελέσει το training set. Ένα άλλο 20% θα αποτελεί το testing set και το τελευταίο 20% το validation set.
Για κάθε γραμμή θα λαμβάνεται η συνένωση (concatenation) των διανυσμάτων που αντιστοιχούν σε κάθε ένα από τα αντικείμενα της σχέσης. Ουσιαστικά το διάνυσμα εισόδου στο μοντέλο θα έχει 2000 διαστάσεις.
Η αρχική εκπαίδευση του μοντέλου (2ο παραδοτέο) πραγματοποιείται σύμφωνα με το stochastic gradient descent, δηλαδή την ενημέρωση των συντελεστών βαρύτητας του μοντέλου μετά από την επεξεργασία κάθε μίας γραμμής εισόδου.

Μετά την εκπαίδευση του μοντέλου, μπορούμε να εκτιμήσουμε την απόδοσή του, ελέγχοντας την ακρίβεια (accuracy), εισάγοντας προς εκτίμηση στο μοντέλο μας το training set. Μπορούμε να αυξομειώνουμε διάφορες παραμέτρους για καλύτερη απόκριση, όπως για παράδειγμα το learning rate ή τον αριθμό των διαστάσεων που διατηρούμε, εώς ότου καταλήξουμε στις τελικές τιμές των συντελεστών βαρύτητας του μοντέλου μας.

Στο τέλος θα υλοποιήσουμε το μοντέλο επαναληπτικής μάθησης ως εξής:

```
threshold = initial_value
training_set = initial_training_set
while (threshold<0.5):
        b = train_model(training_set)
        for all x: // all pairs
                if (p(x,b) < threshold) or (p(x,b) > 1 - threshold):
                        add(x, p(x) new_training_set)
        training_set = resolve_transitivity_issues(training_set, new_training_set)
        threshold += step_value
print(training_set) // only positive matches
```

Εισαγωγή πολυνηματισμού στην εκπαίδευση του μοντέλου (training)

Η διαδικασία που ακολουθείται, όπου ο επανυπολογισμός των βαρών των συντελεστών πραγματοποιείται μετά την αξιολόγηση κάθε μίας παρατήρησης από το training set ονομάζεται στοχαστική ανάλυση καθόδου παραγώγου (stochastic gradient descent), επειδή επιλέγεται μία τυχαία παρατήρηση κάθε φορά. Με τον τρόπο αυτό μπορεί να δημιουργηθούν αρκετά απότομες κινήσεις, που δεν βοηθούν στην ομαλή προσέγγιση. Συνήθως υπολογίζεται η παράγωγος σε δέσμες (batches) από παρατηρήσεις αντί για μεμονωμένα στιγμιότυπα. Μία ακραία περίπτωση είναι ο υπολογισμός της παραγώγου συνυπολογίζοντας ολόκληρο το training set (full batch training). Η μέθοδος αυτή προσφέρει την καλύτερη δυνατή εκτίμηση της σωστής κατεύθυνσης μετακίνησης των βαρών. Το μειονέκτημά της είναι η επιβάρυνση σε μνήμη και επεξεργαστική ισχύ που απαιτεί, γεγονός που την καθιστά απαγορευτική για αρκούντως μεγάλα training sets.

Μία ενδιάμεση λύση είναι το batch training (ή mini-batch training). Το μοντέλο εκπαιδεύεται σε μία δέσμη παρατηρήσεων (π.χ. 512 ή 1024, αριθμός μικρότερος του συνολικού). Με τον τρόπο αυτό μπορούμε να κάνουμε υπολογιστικά αποδοτική την εκπαίδευση του μοντέλου μας. Επιπλέον, η επεξεργασία/υπολογισμός κάθε μίας δέσμης μπορεί να παραλληλοποιηθεί και στη συνέχεια να αθροιστεί το σφάλμα (loss).

Θα πρέπει να τροποποιήσουμε τον ορισμό για τη συνάρτηση σφάλματος cross-entropy loss, αλλά και την αντίστοιχη παράγραφο. Η επέκταση της συνάρτησης σφάλματος για δέσμες διάστασης m θα είναι:

$$Cost(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(\sigma\left(\mathbf{w}^{\mathrm{T}}\mathbf{x}+\mathrm{b}\right)\right)\right] + \left(1-y^{(i)}\right)\log\left(1-\sigma\left(\mathbf{w}^{\mathrm{T}}\mathbf{x}+\mathrm{b}\right)\right)]$$

και αντίστοιχα, η παράγωγος της δέσμης θα είναι το μέσο των μεμονωμένων παραγώγων:

$$\frac{\partial}{\partial w_j}Cost(\mathbf{w}, b) = \sum_{i=1}^{m}\left(\sigma\left(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(\mathrm{i})} + b\right) - y^{(i)}\right)x_j^{(i)}$$

Εισαγωγή πολυνηματισμού στην εκτέλεση του μοντέλου (testing)

Για την εισαγωγή πολυνηματισμού στην εκτέλεση του μοντέλου, θα εξετάζετε σύνολα υποψηφίων ζευγών σε δέσμες. Σε αυτή την περίπτωση, οι απαιτήσεις συγχρονισμού είναι μικρότερες, μιας και τα νήματα θα χρειάζεται να συγχρονίζονται μόνο κάθε φορά που ολοκληρώνεται η επεξεργασία ενός ολόκληρου testing set.

Τελική Αναφορά

Στη τελική αναφορά θα παρουσιάσετε μια σύνοψη ολόκληρη της εφαρμογής που υλοποιήθηκε. Μπορείτε να αναφέρετε πράγματα που παρατηρήσατε κατά την μοντελοποίηση/υλοποίηση της

εφαρμογής σας, με αποτέλεσμα να σας οδηγήσουν σε συγκεκριμένες σχεδιαστικές επιλογές που βελτίωσαν την εφαρμογή σας σε επίπεδο χρόνου, μνήμης, κτλ.
Στην αναφορά πρέπει ακόμη να παρουσιάσετε ένα σύνολο από πειράματα, τα οποία θα δείχνουν το χρόνο εκτέλεσης για όλες τις επιλογές των δομών που αναπτύξατε στα τρία παραδοτέα. Για παράδειγμα μπορείτε να αναφέρετε ή/και να παρουσιάσετε με διαγράμματα τον χρόνο εκτέλεσης, κατανάλωση μνήμης, πολυνηματισμό κ.α. . Τέλος, είναι απαραίτητο να παρουσιάσετε τις δομές που σας έδωσαν τους καλύτερους χρόνους εκτέλεσης για τα datasets και τον τελικό χρόνο/μνήμη, μαζί με τις προδιαγραφές του μηχανήματος που εκτελέσατε τα πειράματα. Η αναφορά δεν πρέπει να ξεπερνά τις 30 σελίδες.

Παράδοση εργασίας

Προθεσμία παράδοσης: 24/01/2021
Γλώσσα υλοποίησης: C / C++ χωρίς χρήση stl.
Περιβάλλον υλοποίησης: Linux (gcc > 5.4+).

Παραδοτέα: Η παράδοση της εργασίας θα γίνει με βάση το τελευταιο commit πριν την προθεσμία υποβολής στο git repository σας. **Η χρήση git είναι υποχρεωτική**.
Επιπλέον, εκτός από τον πηγαίο κώδικα, θα παραδώσετε μια σύντομη αναφορά, με τις σχεδιαστικές σας επιλογές καθώς και να εφαρμόσετε ελέγχους ως προς την ορθότητα του λογισμικού με τη χρήση ανάλογων βιβλιοθηκών (Software testing).