La relazione dovrà includere una descrizione del problema, ma non necessariamente la descrizione della forma dei dati in ingresso e/o uscita.

L’algoritmo è solo quella cosa che lega, logicamente, istanze e soluzioni: tutto ciò che è tipico esclusivamente del codice C che lo esegue (come le librerie utilizzate) non è parte dell’algoritmo.

Una volta descritto l’algoritmo ha senso iniziare a parlare di quali sono, qualitativamente, i dati in entrata (la configurazione di pezzi su una scacchiera, l’elenco di prodotti in un negozio, ecc..) per poi parlare di quale tipologia di strutture dati sono state necessarie per rappresentare in memoria tali dati.

ES: se ad esempio devo caricare una matrice n\*n devo osservare che la procedura di input (che sarà implementata in due cicli for di lunghezza n) ha una complessità di n2. Idem per la complessità spaziale.  
Se la matrice è di dimensione costante posso parlare di spazio e complessità costante.

OSS: ogni questione puramente implementativa si mette in calce alla relazione, se necessario, sotto a “note implementative”

Ogni macrosezione di codice deve avere una descrizione che comprende la propria complessità asintotica ed, eventualmente, spaziale. (una sezione che opera strettamente su memoria già allocata non consuma memoria)

Criteri valutazione RELAZIONE:

COMPLETEZZA: deve includere la descrizione del problema (riassunto specifiche, sostanzialmente), descrizione del modello impiegato (es: le strade sono un grafo, gli eventi uno heap, ecc..), dettagli sull’implementazione (es: ho usato qualcosasort per quel pezzo lì, ecc..).  
Molti aspetti di questa parte devono prescindere il più possibile dalla programmazione (es: carico da linea di comando, faccio un ciclo for, ecc..).  
Se ci sono cose notevoli dal punto di vista programmativo si può mettere in calce.

CORRETTEZZA: non devi dir vaccate in senso assoluto, ma anche relativamente a quanto fatto nel codice.

STRUTTURA: deve essere organizzata in modo coerente. Conta meno.

Criteri valutazione CODICE:

CORRETTEZZA: Deve essere corretto, girare e risolvere il problema. Nel peggiore dei casi un po’ di debug lo fa il prof. Occhio che sia funzionale su ogni macchina. Più è modulare meglio è. (Il prof usa Microsoft visual studio express 2012 per compilare. In C ’89)

Struttura dati forniti

**Rete**: ogni campo rappresenta una strada nel seguente modo:

Nodo\_Partenza Nodo\_Arrivo Tempo\_percorrenza

**Veicoli**: osserva che la batteria non si consuma se il veicolo è fermo

Numero\_Veicoli Tempo\_Servizio\_secondi Durata\_Batteria\_secondi Tempo\_Ricarica\_secondi

**Chiamate**: il veicolo DEVE arrivare a destinazione prima del termine del tempo

Secondo\_ricezione\_chiamata Nome\_Cliente Punto\_Partenza Punto\_Arrivo Secondo\_partenza\_richiesta Secondo\_Arrivo\_richiesto Bonus\_se\_il\_cliente\_non\_deve\_aspettare\_l’arrivo\_del\_veicolo

Il guadagno è al secondo: il guadagno massimo si ha se il veicolo arriva prima dell’orario partenza (il più vicino possibile) e arriva il più tardi possibile (viaggio tariffato al secondo): Il prof non vuole le truffe, usa solo il cammino minimo.

La ricarica deve essere fatta se a fine viaggio la batteria ha meno del 10% di carica

Si ricarica un solo veicolo per volta, e si forma una coda.

La partenza dei veicoli deve essere successiva alla chiamata corrispondente

“Sistema di simulazione ad eventi discreti”

Quando si verifica un evento so già qual è il tempo totale: metto gli eventi in heap e scelgo il più efficiente (tempo minimo), poi modifico lo stato del sistema, e ricalcolo i possibili eventi successivi, eccetera.

Credo l’idea sia: ad ogni chiamata faccio l’heap del tempo che ogni auto (solo libere, anche se non è ottimale) ci metterebbe, scelgo l’auto col tempo minimo, o col guadagno massimo (guarda specifiche) e la metto come occupata, poi modifico lo stato del sistema, e guardo l’evento successivo.

Per stabilire la posizione iniziale dei veicoli si usa un Greedy: devono risultare ognuno in un punto diverso.