Laboratorio per il corso di Algoritmi e Strutture Dati: regole d'esame, indicazioni generali e suggerimenti, consegne per gli esercizi

Indice

Regole d'esame

Importante: gli studenti che hanno nel piano di studi l'insegnamento di Algoritmi con un numero di CFU differente dal quello della corrente edizione (es. 12 CFU) sono pregati di contattare il docente al più presto, al fine di concordare un programma d'esame commisurato ai CFU.

Il progetto di laboratorio può essere svolto individualmente o in gruppo (al più 3 persone). **I membri di uno stesso gruppo devono appartenere tutti allo stesso turno di laboratorio**.

ll progetto di laboratorio va consegnato mediante Git (vedi sotto) entro e non oltre la data della prova scritta che si intende sostenere. E' vietato sostenere la prova scritta in caso di mancata consegna del progetto di laboratorio. In caso di superamento della prova scritta, la prova orale (discussione del laboratorio) va sostenuta, previa prenotazione mediante apposita procedura che sarà messa a disposizione sulla pagina i-learn del corso, **nella medesima sessione della prova scritta superata** (si ricorda che le sessioni sono giugno-luglio 2020, settembre 2020, dicembre 2020 e gennaio-febbraio 2021).

Si noti che, per la sola sessione di giugno-luglio saranno previsti due appelli e, pertanto, esisteranno due possibilità per la discussione del laboratorio (primo o secondo appello della sessione). Nelle altre sessioni, l'appello è unico. Ad esempio, se la studentessa/lo studente X supera la prova scritta a dicembre 2020, deve necessariamente sostenere la discussione di laboratorio con la prova orale di dicembre 2020 (non sarà possibile discutere a gennaio-febbraio 2021).

Esempio:

- la studentessa/lo studente X sostiene la prova scritta nel primo appello di giugno;
- la studentessa/lo studente X deve assicurarsi che il progetto su GitLab, alla data della prova scritta che intende sostenere (in questo esempio, quella del primo appello di giugno), sia aggiornato alla versione che vuole presentare al docente di laboratorio;
- se la studentessa/lo studente X supera la prova scritta nel primo appello di giugno, deve (pena la perdita del voto ottenuto nella prova scritta) iscriversi a uno degli appelli orali di giugno o luglio, prenotarsi su i-learn in

uno degli slot messi a disposizione dal docente del turno di appartenenza e sostenere l'orale nello slot temporale prenotato.

Le regole riportate sopra si applicano alla singola studentessa/al singolo studente. Per poter accedere alla discussione di laboratori è in ogni caso necessaria l'iscrizione alla prova orale corrispondente su myunito.

Studentesse/studenti diversi, appartenenti allo stesso gruppo, possono sostenere la prova **scritta** nello stesso appello o in appelli diversi. Se studentesse/studenti diversi, appartenenti allo stesso gruppo, superano la prova scritta nello stesso appello, **devono** sostenere l' **orale** nello stesso appello orale. Se studentesse/studenti diversi, appartenenti allo stesso gruppo, superano la prova scritta in appelli diversi, **possono** sostenere l'**orale** in appelli diversi.

Ad esempio, si consideri un gruppo di laboratorio costituito dalle studentesse/dagli studenti X, Y e Z, e si supponga che i soli X e Y sostengano la prova scritta nel primo appello di giugno, X con successo, mentre Y con esito insufficiente. Devono essere rispettate le seguenti condizioni:

- alla data della prova scritta del primo appello di giugno, il progetto di laboratorio del gruppo deve essere aggiornato alla versione che si intende presentare;
- il solo studente X deve sostenere la prova orale nella sessione giugno-luglio, procedendo come indicato nell'esempio riportato sopra, mentre Y e Z sosterranno la discussione quando avranno superato la prova scritta.
- Supponiamo che Y e Z superino la prova scritta nell'appello di gennaio: essi dovranno sostenere la prova orale nella sessione di gennaio-febbraio
- Gli studenti Y e Z dovranno, di norma, discutere la stessa versione del progetto di laboratorio che ha discusso lo studente X; i.e., eventuali modifiche al laboratorio successive alla discussione di X dovranno essere debitamente documentate (i.e., il log delle modifiche dovrà comparire su GitLab) e motivate.

Validità del progetto di laboratorio : le specifiche per il progetto di laboratorio descritte in questo documento resteranno valide fino all'ultimo appello della sessione gennaio-febbraio relativa al corrente anno accademico **(vale a dire, quella di gennaio-febbraio 2021)** e non oltre!. Gli appelli delle sessioni successive a questa dovranno essere sostenuti sulla base delle specifiche che verranno descritte nella prossima edizione del laboratorio di algoritmi.

Come unica eccezione si ammetterà, per il solo primo appello della sessione giugno-luglio dell'anno accademico successivo a quello corrente (vale a dire per il primo appello di giugno 2021), che venga discusso il laboratorio presentato in questo documento **a patto che i commit su gitlab dimostrino che il lavoro è stato completato entro la sessione di gennaio-febbraio relativa all'anno accademico corrente**.

Indicazioni generali e suggerimenti

Uso di Git

Durante la scrittura del codice è richiesto di usare in modo appropriato il sistema di versioning Git. Questa richiesta implica quanto segue:

- il progetto di laboratorio va inizializzato "clonando" il repository del laboratorio come descritto nel file Git.md;
- come è prassi nei moderni ambienti di sviluppo, è richiesto di effettuare commit frequenti. L'ideale è un commit per ogni blocco di lavoro terminato (es. creazione e test di una nuova funzione, soluzione di un baco, creazione di una nuova interfaccia, ...);
- ogni membro del gruppo dovrebbe effettuare il commit delle modifiche che lo hanno visto come principale sviluppatore;
- al termine del lavoro si dovrà consegnare l'intero repository.

Il file Git.md contiene un esempio di come usare Git per lo sviluppo degli esercizi proposti per questo laboratorio.

Nota importante: Su git dovrà essere caricato solamente il codice sorgente, in particolare nessun file dati dovrà essere oggetto di commit!

Si rammenta che la valutazione del progetto di laboratorio considererà anche l'uso adeguato di git da parte di ciascun membro del gruppo.

Linguaggio in cui sviluppare il laboratorio

Gli esercizi vanno implementati utilizzando il linguaggio C o Java come precisato di seguito:

- Esercizio 1: C
- Esercizio 2: Java
- Esercizio 3: C
- Esercizio 4: C o Java a discrezione dello studente

Come indicato sotto, alcuni esercizi chiedono di implementare codice generico. Seguono alcuni suggerimenti sul modo di realizzare codice con questa caratteristica nei due linguaggi accettati.

Nota : Con "codice generico" si fa riferimento al fatto che tale codice deve poter essere eseguito con tipi di dato non noti a tempo di compilazione.

Suggerimenti (C): Nel caso del C, è necessario capire come meglio approssimare l'idea di codice generico utilizzando quanto permesso dal linguaggio. Un approccio comune è far sì che le funzioni e le procedure presenti nel codice prendano in input

puntatori a void e utilizzino qualche funzione fornita dall'utente per accedere alle componenti necessarie.

Nota: chi è in grado di realizzare tipi di dato astratto tramite tipi opachi è incoraggiato a procedere in questa direzione.

Suggerimenti (Java): Sebbene in Java la soluzione più in linea con il moderno utilizzo del linguaggio richiederebbe la creazione di classi parametriche, tutte le scelte implementative (compresa la decisione di usare o meno classi parametriche) sono lasciate agli studenti. Inoltre, è possibile (e consigliato) usare gli ArrayList invece degli array nativi al fine di semplificare la realizzazione di codice generico.

Uso di librerie esterne e/o native del linguaggio scelto

È vietato (sia nello sviluppo in Java che in quello in C) l'uso di strutture dati native del linguaggio scelto o offerte da librerie esterne, quando la loro realizzazione è richiesta da uno degli esercizi proposti.

È, invece, possibile l'uso di strutture dati native del linguaggio o offerte da librerie esterne, se la loro realizzazione non è richiesta da uno degli esercizi proposti.

Es.: nello sviluppo in Java, l'uso di ArrayList è da ritenersi possibile, se nessun esercizio chiede la realizzazione in Java di un array dinamico.

Qualità dell'implementazione

È parte del mandato degli esercizi la realizzazione di codice di buona qualità.

Per "buona qualità" intendiamo codice ben modularizzato, ben commentato e ben testato.

Alcuni suggerimenti:

- verificare che il codice sia suddiviso correttamente in package o moduli;
- aggiungere un commento, prima di una definizione, che spiega il funzionamento dell'oggetto definito. Evitare quando possibile di commentare direttamente il codice interno alle funzioni/metodi implementati (se il codice è ben scritto, i commenti in genere non servono);
- la lunghezza di un metodo/funzione è in genere un campanello di allarme: se essa cresce troppo, probabilmente è necessario rifattorizzare il codice spezzando la funzione in più parti. In linea di massima si può consigliare di intervenire quando la funzione cresce sopra le 30 righe (considerando anche commenti e spazi bianchi);
- sono accettabili commenti in italiano, sebbene siano preferibili in inglese;
- tutti i nomi (es., nomi di variabili, di metodi, di classi, ecc.) *devono* essere significativi e in inglese;
- il codice deve essere correttamente indentato; impostare l'indentazione a 2 caratteri (un'indentazione di 4 caratteri è ammessa ma scoraggiata) e

impostare l'editor in modo che inserisca "soft tabs" (cioè, deve inserire il numero corretto di spazi invece che un carattere di tabulazione);

- per dare i nomi agli identificatori, seguire le convenzioni in uso per il linguaggio scelto:
  - Java: i nomi dei package sono tutti in minuscolo senza separazione fra le parole (es. thepackage); i nomi dei tipi (classi, interfacce, ecc.) iniziano con una lettera maiuscola e proseguono in camel case (es. TheClass), i nomi dei metodi e delle variabili iniziano con una lettera minuscola e proseguono in camel case (es. theMethod), i nomi delle costanti sono tutti in maiuscolo e in formato snake case (es. THE_CONSTANT);
  - C: macro e costanti sono tutti in maiuscolo e in formato snake case (es. THE_MACRO, THE_CONSTANT); i nomi di tipo (e.g. struct, typedefs, enums, ...) iniziano con una lettera maiuscola e proseguono in camel case (e.g., TheType, TheStruct); i nomi di funzione iniziano con una lettera minuscola e proseguono in snake case (e.g., the_function());
- i file vanno salvati in formato UTF-8.

Consegne per gli esercizi

Nota : la presente sezione contiene alcune formule descritte usando la sintassi LaTeX. È possibile convertire l'intero documento in formato pdf - di più facile lettura - usando l'utility pandoc. Da riga di comando (Unix):

```
pandoc --shift-heading-level-by=-1 README.md -o README.pdf
```

Importante: Gli esercizi 1 e 3 richiedono (fra le altre cose) di sviluppare codice generico. Nello sviluppare questa parte, si deve assumere di stare sviluppando una libreria generica intesa come fondamento di futuri programmi. Non è pertanto lecito fare assunzioni semplificative; in generale, l'implementazione della libreria generica non deve essere influenzata in alcun modo dagli usi di essa eventualmente richiesti negli esercizi (ad esempio, se un esercizio dovesse richiedere l'implementazione della struttura dati grafo e quello stesso o un altro esercizio dovesse richiedere l'implementazione, a partire da tale struttura dati, di un algoritmo per il calcolo delle componenti connesse di un grafo, l'implementazione della struttura dati *non* dovrebbe contenere elementi – variabili, procedure, funzioni, metodi, ecc. – eventualmente utili per il calcolo delle componenti connesse, ma non essenziali alla struttura dati; analogamente, se un esercizio dovesse richiedere di operare su grafi con nodi di tipo stringa, l'implementazione della struttura dati grafo dovrebbe restare generica e non potrebbe quindi assumere per i nodi il solo tipo stringa).

In sede di discussione d'esame, sarà facoltà del docente chiedere di eseguire gli algoritmi implementati su dati forniti dal docente stesso. Nel caso questi dati siano memorizzati su file, questi saranno dei csv con la medesima struttura dei dataset forniti e descritti nel testo dell'esercizio. I codici sviluppati dovranno consentire un rapido e semplice adattamento agli input forniti: ad esempio, una buona implementazione consentirà di inserire in input il nome del file su cui eseguire il test, mentre una peggiore richiederà di modificare il codice sorgente e una successiva compilazione a fronte della sola modifica del nome del file contenente il dataset.

Unit Testing

Come indicato esplicitamente nei testi degli esercizi, il progetto di laboratorio comprende anche la definizione di opportune suite di unit tests.

Si rammenta, però, che il focus del laboratorio è l'implementazione di strutture dati e algoritmi. Relativamente agli unit-test sarà quindi sufficiente che gli studenti dimostrino di averne colto il senso e di saper realizzare una suite di test sufficiente a coprire i casi più comuni (compresi, in particolare, i casi limite).

Esercizio 1

Linguaggio richiesto: C

Testo

Implementare una libreria che offre i seguenti algoritmi di ordinamento:

- Insertion Sort
- Quicksort

Il codice che implementa ciascun algoritmo deve essere generico. Inoltre, ogni algoritmo deve permettere di specificare (cioè deve accettare in input) il criterio secondo cui ordinare i dati.

Unit Testing

Implementare gli unit-test degli algoritmi secondo le indicazioni suggerite nel documento Unit Testing.

Uso della libreria di ordinamento implementata

Il file `records.csv` può essere scaricato dalle macchine del laboratorio seguendo il path

- `/usr/NFS/Linux/labalgoritmi/datasets/`

oppure dalla pagina web:

- https://datacloud.di.unito.it/index.php/s/MN5QZAPE4t5Na6F.

Il file contiene 20 milioni di record da ordinare. Ogni record è descritto su una riga e contiene i seguenti campi:

- id: (tipo intero) identificatore univoco del record;
- field1: (tipo stringa) contiene parole estratte dalla divina commedia, potete assumere che i valori non contengano spazi o virgole;
- field2: (tipo intero);
- field3: (tipo floating point);

Il formato è un CSV standard: i campi sono separati da virgole; i record sono separati da `\n`.

Usando ciascuno degli algoritmi implementati, si ordinino i *record* (non è sufficiente ordinare i singoli campi) contenuti nel file `records.csv` in ordine non decrescente secondo i valori contenuti nei tre campi "field" (cioè, per ogni algoritmo, è necessario ripetere l'ordinamento tre volte, una volta per ciascun campo).

Si misurino i tempi di risposta e si produca una breve relazione in cui si riportano i risultati ottenuti insieme a un loro commento. Nel caso l'ordinamento si protragga per più di 10 minuti potete interrompere l'esecuzione e riportare un fallimento dell'operazione. I risultati sono quelli che vi sareste aspettati? Se sì, perché? Se no, fate delle ipotesi circa il motivo per cui gli algoritmi non funzionano come vi aspettate, verificatele e riportate quanto scoperto nella relazione.

Ricordiamo che il file `records.csv` non deve essere oggetto di commit su git!

Esercizio 2

Linguaggio richiesto: Java

Testo

Si consideri il problema di determinare la distanza di edit tra due stringhe (Edit distance): date due stringhe s1 e s2, non necessariamente della stessa lunghezza, determinare il minimo numero di operazioni necessarie per trasformare la stringa s2 in s1. Si assuma che le operazioni disponibili siano: cancellazione e inserimento. Esempi:

- "casa" e "cassa" hanno edit distance pari a 1 (1 cancellazione);
- "casa" e "cara" hanno edit distance pari a 2 (1 cancellazione + 1 inserimento);
- "vinaio" e "vino" hanno edit distance=2 (2 inserimenti);
- "tassa" e "passato" hanno edit distance pari a 4 (3 cancellazioni + 1 inserimento);
- "pioppo" e "pioppo" hanno edit distance pari a 0.

1. Si implementi una versione ricorsiva della funzione edit_distance basata sulle seguenti osservazioni (indichiamo con $|s|$ la lunghezza di s e con rest(s) la sottostringa di s ottenuta ignorando il primo carattere di s):

- se $|s1| = 0$, allora edit_distance$(s1, s2) = |s2|$;
- se $|s2| = 0$, allora edit_distance$(s1, s2) = |s1|$;
- altrimenti, siano:
  - $d_{\text{no-op}} = \begin{cases} \text{edit\_distance}(\text{rest}(s1), \text{rest}(s2)) & \text{se } s1[0] = s2[0] \\ \infty & \text{altrimenti} \end{cases}$
  - $d_{\text{canc}} = 1 + \text{edit\_distance}(s1, \text{rest}(s2))$
  - $d_{\text{ins}} = 1 + \text{edit\_distance}(\text{rest}(s1), s2)$

Si ha: edit_distance$(s1, s2) = \min\{d_{\text{no-op}}, d_{\text{canc}}, d_{\text{ins}}\}$

1. Si implementi una seconda versione edit_distance_dyn della funzione, adottando una strategia di programmazione dinamica. Tale versione deve essere anch'essa ricorsiva (in particolare, essa può essere facilmente ottenuta a partire dall'implementazione richiesta al punto precedente).

Nota: Le definizioni sopra riportate non corrispondono al modo usuale di definire la distanza di edit. Sono del tutto sufficienti però per risolvere l'esercizio e sono quelle su cui dovrà essere basato il codice prodotto.

Unit Testing

Implementare gli unit-test degli algoritmi secondo le indicazioni suggerite nel documento Unit Testing.

Uso delle funzioni implementate

I file `dictionary.txt` e `correctme.txt` possono essere scaricati dalle macchine del laboratorio seguendo il path

- `/usr/NFS/Linux/labalgoritmi/datasets/`

oppure dalla pagina web:

- https://datacloud.di.unito.it/index.php/s/tEgL74ExJNHmXQr.

Il file `dictionary.txt` contiene l'elenco (di una parte significativa) delle parole italiane. Le parole sono scritte di seguito, ciascuna su una riga.

Il file `correctme.txt` contiene una citazione di John Lennon. La citazione presenta alcuni errori di battitura.

Si implementi un'applicazione che usa la funzione edit_distance_dyn per determinare, per ogni parola w in correctme.txt, la lista di parole in dictionary.txt con edit distance minima da w. Si sperimenti il funzionamento dell'applicazione e si riporti in una breve relazione (circa una pagina) i risultati degli esperimenti.

Si ricorda che i file `dictionary.txt` e `correctme.txt` non devono essere oggetto di commit su git!

Esercizio 3

Linguaggio richiesto: C

Testo

Si implementi una libreria per la struttura dati Hash Map, tenendo conto delle seguenti indicazioni:

- Una Hash Map rappresenta un insieme di associazioni del tipo `<K,V>`, dove K è una chiave e V è il valore ad essa associato;
- in una Hash Map, non possono esservi chiavi ripetute;
- l'implementazione sfrutta un meccanismo di hashing;
- L'implementazione deve offrire le seguenti operazioni:
  - creazione di una Hash Map vuota;
  - distruzione di una Hash Map (con conseguente deallocazione della memoria associata);
  - verifica se una Hash Map è vuota;
  - recupero del numero di associazioni presenti in una Hash Map;
  - cancellazione di tutte le associazioni di una Hash Map;
  - verifica se la chiave specificata è presente in una Hash Map;
  - inserimento in una Hash Map di un'associazione di tipo `<K,V>`;
  - recupero da una Hash Map dell'eventuale valore, associato alla chiave specificata
  - cancellazione da una Hash Map dell'eventuale associazione con una chiave specificata
  - recupero dell'insieme delle chiavi presenti in una Hash Map
- Il codice che implementa la Hash Map deve essere generico (nel senso che deve consentire di inserire associazioni `<K,V>` di cui non è noto a tempo di compilazione né il tipo della chiave K, né quello del valore V) e non deve assumere alcuna cardinalità massima per l'insieme di associazioni che possono essere ospitate nella Hash Map.

Unit Testing

Implementare gli unit-test degli algoritmi secondo le indicazioni suggerite nel documento Unit Testing.

Uso delle funzioni implementate

Il file `hashes.txt` contiene circa sei milioni di coppie di interi. Il primo elemento di ogni coppia rappresenta una chiave, il secondo elemento rappresenta un valore. Ogni coppia è scritta su una linea. Gli elementi della coppia sono separati da una virgola. Il file può essere scaricato dalle macchine del laboratorio seguendo il path

- `/usr/NFS/Linux/labalgoritmi/datasets/`

oppure dalla pagina web:

- https://datacloud.di.unito.it/index.php/s/QyyPSzi28B8q4rr.

Si esegua quanto segue e si scriva una breve relazione riguardante i risultati ottenuti:

- si carichi il contenuto del file in un oggetto di tipo Hash Map; si misurino i tempi di caricamento;
- si carichi il contenuto del file in un array allocato staticamente (`hashes.txt` contiene 6 321 078 coppie di interi) ordinato per chiave (la decisione a proposito di quando e come effettuare l'ordinamento è lasciata allo studente e deve essere presa seguendo un criterio di economia dei tempi di esecuzione); si misurino i tempi di caricamento;
- si estraggano a caso 10 000 000 numeri interi tra 0 e 10 000 000 e li si memorizzi in un array `keys`;
- si misuri il tempo necessario per recuperare i valori associati alle chiavi contenute nell'array `keys` usando la Hash Map;
- si misuri il tempo necessario a recuperare i valori associati alle chiavi contenute nell'array `keys` usando una ricerca binaria sull'array ordinato;
- si controlli che il numero di chiavi reperite con successo sia identico nei due casi presi in considerazione.

Nota: Il numero esatto di chiavi che riuscirete a reperire dipende dai valori che avete generato casualmente. Dato però che il dataset contiene $6\,321\,078 \simeq (1 - 1/e) \cdot 10^7$ chiavi distinte estratte nell'intervallo $[0 \dots 10^7)$, un semplice ragionamento probabilistico suggerisce che il numero di chiavi che dovreste riuscire a reperire facendo N tentativi è circa $(1 - 1/e) \cdot N$ e quindi, nello specifico dell'esercizio proposto, tale numero è di nuovo $(1 - 1/e) \cdot 10^7$.

Nota 2: Non è necessario che entrambe le prove (il test con array e quello con la Hash Map) siano eseguite durante la stessa esecuzione del programma. Nel caso si proceda eseguendo i due test in due esecuzioni separate, ci si accerti che il seme usato per la generazione dei valori contenuti nell'array `keys` sia lo stesso in entrambi i casi.

Esercizio 4

Linguaggio richiesto: C o Java

Testo

Si consideri un grafo connesso con N nodi e $N - 1$ archi bidirezionali pesati con un peso intero W. Ci si pone il problema di trovare un algoritmo efficiente per rispondere a Q distinte interrogazioni.

Una interrogazione consiste in un nuovo arco pesato q. L'algoritmo deve rispondere `YES` se q permette di ridurre il peso complessivo del grafo, `NO` altrimenti. L'arco q soddisfa questa condizione se esiste un arco e tale per cui sia possibile sostituire q a e lasciando il grafo connesso e diminuendone il peso complessivo.

L'esecuzione della singola interrogazione non deve modificare il grafo (i.e., il grafo di partenza è sempre lo stesso).

I file di input iniziano con una linea contenente il numero N di nodi del grafo a cui seguono $N-1$ linee contenenti gli archi. Ogni linea che specifica un arco contiene `3` interi separati da spazi: il nodo sorgente, il nodo destinazione e il peso dell'arco.

I file continuano con una linea contenente il numero Q di interrogazioni a cui rispondere. Seguono Q linee contenenti le interrogazioni. Ogni interrogazione è nello stesso formato usato per descrivere gli archi.

L'output del programma deve consistere in esattamente Q linee contenenti `YES` o `NO` a seconda che la risposta alla corrispondente interrogazione sia positiva (l'arco oggetto dell'interrogazione riduce il peso del grafo) o negativa (viceversa).

Potete assumere quanto segue:

- $1 \leq N \leq 100\,000$
- $1 \leq Q \leq 100\,000$
- i nodi sono interi che assumono valori nel range $[1, 100\,000]$
- per ogni arco $(u, v, w) : u \neq v \wedge w \in [1, 1\,000\,000\,000]$.

Esempio:

Input:

```
6
1 2 2
1 3 3
3 4 5
3 5 4
2 6 4
4
1 4 4
4 5 6
2 3 8
1 6 3
```

Output atteso:

```
YES
NO
NO
YES
```

Unit Testing

Per questo esercizio non siete tenuti a scrivere unit test. Siete liberi di farlo se lo ritenete opportuno.

Verifica delle prestazioni

Insieme a questo esercizio vengono forniti 11 dataset di test. Ogni dataset consiste in un file nel formato descritto sopra e un file risultato nel formato richiesto come output. Si può considerare un test superato se l'algoritmo implementato completa l'elaborazione del file di input in meno di 2 secondi.

I file di test (contenuti nell'archivio `graph_substitution_tests.zip`) possono essere scaricati accedendo dalle macchine del laboratorio al seguente path:

- `/usr/NFS/Linux/labalgoritmi/datasets/`

oppure dalla pagina web:

- https://datacloud.di.unito.it/index.php/s/5AM3H3BsBsPrTMM

Note importanti

- Tutti i test possono essere completati nei tempi richiesti su una macchina ragionevomente attrezzata (es., i computer in laboratorio);
- Superare tutti i test *non* è facile e *non* è richiesto. In fase d'esame verrà valutato lo sforzo fatto per andare oltre la soluzione più banale, ma non ci si aspetta che tutti riescano a superare tutti i test.
- Si ricorda che i file di test *non* devono essere oggetto di commit su git!
- Lo script `run-tests.rb` fornito insieme a questo progetto può essere usato per verificare se la propria soluzione restituisce un output corretto e se termina in meno di due secondi. Se lanciato con l'opzione `-h` mostra a video un breve testo che indica come usarlo e cosa si aspetta dal programma che implementa la soluzione.