PROGETTI AP – percorso base

Si richiede di sviluppare un algoritmo e la sua implementazione come programma C.
Tutti i programmi devono contenere

- un insieme di commenti iniziali che spiega brevemente le finalità del programma;
- un insieme di commenti all'inizio di ogni function che spiega le finalità della function e il significato dei parametri di input output (*specifiche* della function);
- commenti esplicativi dei principali blocchi di istruzioni;

e devono essere corredati da

- un insieme di almeno 3 esecuzioni per testare il programma con diversi dati di input.

1. algoritmo del Gioco della Vita (Game of Life). Il Gioco della Vita consiste in una scacchiera in cui ogni casella rappresenta una cellula, che può essere viva o morta. Ogni cellula ha, in generale, quattro cellule vicine (a meno che la cellula non si trovi sul bordo della scacchiera). L'algoritmo usa un array 2D **`n`**x**`n`**, con **`n`**=45, per simulare la scacchiera. L'algoritmo esamina **`n`** volte la scacchiera. Durante ogni passo aggiorna lo stato di tutte le cellule sulla scacchiera nel seguente modo: se una cellula ha uno, due o tre cellule vicine vive allora la cellula deve essere posta nello stato "viva"; altrimenti deve essere posta nello stato "morta" (si noti che questa regola consente a una cellula morta di diventare viva). Dopo l'ultimo passo, l'algoritmo visualizza l'array 2D aggiornato. Ogni cellula viva è visualizzata con un asterisco; ogni cellula morta con uno spazio. Fare attenzione al fatto che la scacchiera deve essere aggiornata solo alla fine di ogni passo: ciò significa che l'algoritmo deve usare un array per memorizzare la scacchiera attuale e un array per memorizzare la scacchiera modificata. Usare le tre seguenti configurazioni iniziali(per i tre test): solo la cellula centrale è viva, ovvero quella di indice 22,22; solo le cellule 22,22 e 22,23 sono vive; solo le cellule 0,0 44,44 e 44,43 sono vive.

2. algoritmo per il calcolo della fusione di due insiemi (ordinati) di cognomi, che utilizzi la function **`strcmp`** per determinare quale tra due cognomi precede l'altro nell'ordine alfabetico. Assumere che l'intersezione dei due insiemi di input sia vuota.
   Ognuno dei tre insiemi di cognomi (due di input e uno di output) è un array di puntatori a **`char`**. Nei tre test usare insiemi di almeno 10 e 20 cognomi, 20 e 30 cognomi, 30 e 40 cognomi.

3. algoritmo per la generazione di domande a caso. L'algoritmo usa la function **`rand()`** in **`<stdlib>`** per generare numeri casuali; per esempio, se **`numero_casuale`** è un **`int`**, la chiamata **`numero_casuale=rand()%10;`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9). Dopo aver creato un elenco di 30 domande prestabilite (sul linguaggio C, diverse da quelle nei test online!, e raggruppate in 5 gruppi

tematici, per es. array 1D, array 2D, cicli, puntatori, stringhe), usare la **`rand()`** per selezionare a caso una domanda, visualizzarla e leggere la risposta da inserire da tastiera (la risposta deve essere SI oppure NO) ; dopo ogni risposta, si deve visualizzare se la risposta è corretta o sbagliata, chiedere se si vuole una nuova domanda o se si vuole concludere la seduta. Alla fine della seduta, visualizzare il numero delle risposte esatte, la percentuale di risposte esatte e il gruppo nel quale si sono registrati più errori.

4. algoritmo per il calcolo della fusione di due insiemi (ordinati) di float. Assumere che l'intersezione dei due insiemi di input possa essere non vuota; se l'intersezione è non vuota, solo un elemento, tra quelli uguali, deve comparire nell'array di output.
   Nei tre test usare insiemi di almeno 15 e 20 numeri, 25 e 30 numeri, 35 e 40 numeri.

5. algoritmo per crittografare e decrittografare un testo di 80x20 caratteri. L'algoritmo deve usare il seguente semplice codice a sostituzione: ogni consonante dell'alfabeto è sostituita dalla consonante precedente nell'ordine alfabetico, per es. b diventa z, p diventa n, c diventa b,..., v diventa t, z diventa v; ogni vocale è sostituita dalla vocale precedente; numeri e simboli di interpunzione rimangono inalterati.

6. algoritmo per crittografare e decrittografare un testo di 5x80 caratteri. L'algoritmo deve usare il seguente semplice codice a sostituzione: ogni consonante dell'alfabeto è sostituita dalla consonante seguente nell'ordine alfabetico, per es. b diventa c, d diventa f, n diventa p,..., v diventa z, z diventa b; ogni vocale è sostituita dalla vocale seguente; numeri e simboli di interpunzione rimangono inalterati. Il programma deve essere organizzato in una function che crittografa, in una function che decrittografa e in un main che legge da tastiera il testo, chiama la function che crittografa il testo e poi chiama la function che decrittografa il testo crittografato.

7. algoritmo per generare la matrice (array 2D) delle distanze tra un insieme di punti (di un piano). L'elemento di indici i,j della matrice è la distanza tra l'i-simo punto e il j-simo punto. Ogni punto del piano è individuato univocamente dalla sua ascissa e dalla sua ordinata; pertanto, un insieme di punti è rappresentato da un array 1D di ascisse e da un array 1D di ordinate. L'algoritmo deve essere organizzato come una function che ha in input: l'array delle ascisse, l'array delle ordinate, il loro size; e in output: l'array 2D delle distanze. Scrivere un main che legge da tastiera le ascisse e le ordinate dei punti (anche il numero dei punti deve essere immesso da tastiera); il main chiama la function per il calcolo delle distanze e poi calcola e visualizza le coordinate dei due punti più vicini e dei due punti più lontani.

8. Algoritmo per generare la matrice (array 2D) delle distanze tra un insieme di punti (di un piano). L'elemento di indici i,j della matrice è la distanza tra l'i-simo punto e il j-simo punto. Ogni punto del piano è individuato univocamente dalla

sua ascissa e dalla sua ordinata; pertanto, un insieme di punti è rappresentato da un array 1D di ascisse e da un array 1D di ordinate. L'algoritmo deve essere organizzato come una function che ha in input: l'array delle ascisse, l'array delle ordinate, il loro size; e in output: l'array 2D delle distanze. Scrivere un main che chiede se i dati devono essere immessi da tastiera oppure devono essere generati a caso (il numero dei punti deve comunque essere immesso da tastiera); il main chiama la function per il calcolo delle distanze e poi calcola e visualizza le coordinate dei due punti più vicini e dei due punti più lontani. Per generare a caso un punto del piano si deve generare a caso una ascissa e una ordinata.

9. algoritmo per la generazione di una "mano" di poker. L'algoritmo usa la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare numeri interi casuali. Per esempio, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%53;`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,..,52). Una carta del mazzo delle francesi è caratterizzata da due informazioni: il valore (un **`int`** tra 1 e 13) e il seme (cuori, quadri, fiori, picche. Usare la **`rand()`** per selezionare a caso dieci carte, cinque per il computer e cinque per il giocatore. Fare attenzione al fatto che una stessa carta non può essere estratta due volte. Fare in modo che il computer sia in grado di "dire" il proprio punto, il punto del giocatore e il vincente tra i due (visualizzando una frase del tipo "io ho una coppia di assi, tu hai un tris di jack. Tu vinci"), riconoscendo solo la coppia di.., il tris di ..., il full di …., e il poker di ...

10. algoritmo che, dato in input un numero intero positivo, verifichi se esso è un numero di Fibonacci; nel caso affermativo, l'algoritmo calcola anche l'indice del numero di Fibonacci. L'algoritmo deve essere organizzato come una function. Scrivere un main che per mille volte genera a caso un numero intero nell'insieme {0,1,2,..,MAX_RAND}, chiama la function e visualizza il numero solo nel caso sia un numero di Fibonacci (e visualizza l'indice); alla fine il main visualizza il numero di volte in cui è stato generato un numero di Fibonacci e anche la percentuale di successo.

11. Algoritmo per simulare il lancio di due dadi. I due dadi sono truccati, nel senso che l'uscita di ognuna delle sei facce non è equiprobabile. In particolare, il dado 1 è truccato nel seguente modo: faccia 1, 2, 3, ognuna 10% di probabilità di uscita, faccia 4 e 5, ognuna 20% di probabilità di uscita, faccia 6 30% di probabilità di uscita. Il dado 2 è truccato nel seguente modo: faccia 1 e 2, ognuna 25% di probabilità di uscita, faccia 3, 4 e 5, ognuna 15% di probabilità di uscita, faccia 6 5% di probabilità di uscita. Usare la **`rand()`** per simulare il lancio di un dado. L'algoritmo lancia 10000 volte la coppia di dadi; il punteggio di ogni lancio è dato dalla somma dei valori dei due dadi. Alla fine l'algoritmo visualizza il numero di volte che si è ottenuto ognuno degli undici punteggi possibili 2,3,4,5,..,12, e poi visualizza la lista ordinata mettendo in testa il punteggio che è uscito più volte e via decrescendo. L'algoritmo ripete poi lo stesso esperimento di diecimila lanci, usando però una coppia di dadi non truccati.

12. Algoritmo che, dato in input un numero intero positivo, verifichi se esso è un numero di Fibonacci; nel caso affermativo, l'algoritmo calcola anche l'indice del numero di Fibonacci. L'algoritmo deve essere organizzato come una function. Scrivere un main che per 10000 volte genera a caso un numero intero

nell'insieme {0,1,2,..,20000}, chiama la function e visualizza il numero solo nel caso sia un numero di Fibonacci (e visualizza l'indice); alla fine il main visualizza il numero di volte in cui è stato generato un numero di Fibonacci e anche la percentuale di successo.

13.

14. algoritmo per generare a caso un testo di 20 righe e 80 colonne. Il testo deve essere generato selezionando a caso una parola all'interno di un insieme di parole e simboli di interpunzione prefissati. Tale insieme deve comprendere almeno le seguenti parole:
    informatica, materia, interessante, è, a, me piace, non, università, difficile, facile, facili, sono, contento, difficile, facile, interessante, Parthenope, amo, corso, di, studio, insegnamento, programmazione, laboratorio, i, tutti, giorni, materie, le, impegnative, impegnativo, laurea, mondo, posto, società, software, del, lavoro, Napoli; e i simboli ,. . Aggiungere a propria scelta altre 30 parole all'insieme di cui sopra. L'insieme delle parole deve essere un array di puntatori a stringhe. Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare a caso un numero intero (devono essere numeri interi in {0,1,.., size dell'array delle parole}: si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora l'istruzione **`numero_casuale=rand()/(n+1);`**
    genera un numero casuale di tipo **`int`** (distribuzione uniforme) in {0,1,.., **`n`**}.

15. algoritmo per la simulazione del problema del compleanno. Il problema del compleanno è il seguente: se in una stanza ci sono 23 persone, qual è la probabilità che almeno due di loro siano nate nello stesso giorno dell'anno? L'algoritmo è organizzato come una function che ha in input 23 date di nascita e restituisce in output 0 se non ci sono date uguali e invece restituisce 1 se ci sono almeno due date uguali. L'insieme delle 23 date di nascita deve essere rappresentato mediante due array 1D di 23 elementi, uno per il giorno (che è un intero che può variare da 1 a 30) e uno per il mese (che è un intero che può variare da 1 a 12). Scrivere un main che chiama 10000 volte la function, passando ogni volta un differente insieme di 23 date di nascita. Ogni insieme di 23 date di nascita è generato a caso nel main. Si noti che bisogna generare a caso sia il giorno sia il mese (ovvero sia l'array dei 23 giorni sia l'array dei 23 mesi). Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare numeri interi casuali; per esempio, se **`numero_casuale`** è un **`int`**, la chiamata **`numero_casuale=rand()%(n+1);`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,.., **`n`**). Si supponga per semplicità che i mesi abbiano tutti 30 giorni. Alla fine il main visualizza il numero di volte in cui si è verificato che in un insieme di 23 date di compleanno si sono trovate date uguali e anche la percentuale di successo.

16. algoritmo per l'eliminazione dei duplicati in un insieme di **`int`**. Organizzare l'algoritmo come una function che ha in input l'array 1D e il suo size e in output lo stesso array senza i duplicati e il numero di elementi rimasti. Scrivere un main che legge in input il numero **`n`** di elementi dell'insieme e gli **`n`** numeri interi, visualizza l'array, richiama la function e poi visualizza l'array senza duplicati.
    Nei tre test usare insiemi di almeno 20 numeri, 30 numeri, 50 numeri.

17. Algoritmo per l'eliminazione dei duplicati in un array di interi. L'algoritmo lavora "in place", ovvero restituisce lo stesso array di input dopo aver "eliminato" i duplicati. Più precisamente, dato in input un array **a** di **n** elementi (effettivi), la function restituisce in output lo stesso array **a** e il numero **m** degli elementi effettivi (non duplicati); i primi **m** elementi di **a** sono proprio gli elementi non duplicati. Nei test usare array di size 20, 40, 100, i cui elementi devono essere numeri interi

18. algoritmo per la simulazione di una "mano" di poker. L'algoritmo usa la function **rand()** in **<stdlib.h>** per generare numeri casuali; se **numero_casuale** è un **int**, la chiamata **numero_casuale=rand()%53;** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,..,52). Una carta del mazzo delle francesi è caratterizzata da due informazioni: il valore (un **int** tra 1 e 13) e il seme (cuori, quadri, fiori, picche. Usare la **rand()** per selezionare a caso cinque carte (una "mano" di poker). Fare attenzione al fatto che una stessa carta non può essere estratta due volte. L'algoritmo deve ripetere l'azione 10000 volte (ovvero per 10000 volte si deve dare una "mano" di poker) e deve contare quante volte si è ottenuto un poker d'assi servito, quante volte si è ottenuto un poker di K servito, quante volte si è ottenuto un poker di Q servito, e così via fino a quante volte si è ottenuto un poker di due servito. Dividendo il numero di successi di ogni poker per 10000 si ha una stima della probabilità di avere quel poker servito. Ripetere la simulazione nell'ipotesi di giocare all'italiana, ovvero usando solo le carte 7,8,9,10, Jack, Queen, King, Asso.

19. algoritmo per la generazione a caso di nominativi di persone. La function a ogni chiamata visualizza un nominativo del tipo:
ROSSI ALDO nato a NAPOLI il 20 GENNAIO 1983. Si osservi che un nominativo è costituito da un campo Cognome, un campo nome, un campo Luogo di nascita un campo Giorno di nascita, un campo Mese di nascita e un campo Anno di nascita. In particolare un cognome deve essere scelto (a caso) in un insieme (da prefissare, di 30 cognomi); un nome deve essere scelto (a caso) in un insieme (da prefissare, di 40 nomi); un Luogo di nascita deve essere scelto (a caso) in un insieme (da prefissare, di 15 luoghi di nascita); e così via fino all'anno che deve essere un intero da scegliere (sempre a caso) nell'insieme che va da 1930 a 2004. L'algoritmo usa la function **rand()** in **<stdlib.h>** per generare numeri casuali e quindi per selezionare a caso un elemento in un certo insieme; si ricorda che, per esempio, se **numero_casuale** è un **int**, la chiamata **numero_casuale=rand()%11;** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,10). Usare la **rand()** per selezionare a caso ognuno dei campi che appaiono in un nominativo. Chiamare la function 50 volte e visualizzare i 50 nominativi casuali.

20. algoritmo per l'eliminazione dei duplicati in un insieme di cognomi. L'insieme dei cognomi è rappresentato attraverso un array di (puntatori a) stringhe. L'algoritmo lavora "in place", ovvero restituisce lo stesso array di input dopo aver "eliminato" i duplicati. Più precisamente, dato in input un array **a** di **n** elementi (effettivi), la function restituisce in output lo stesso array **a** e il numero

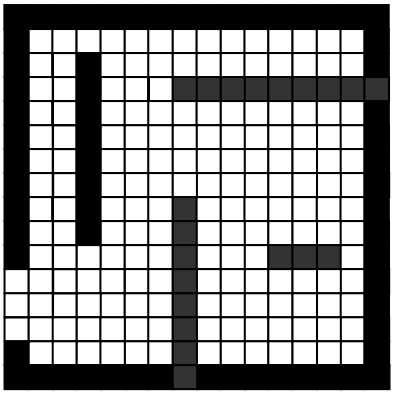
m degli elementi effettivi (non duplicati); i primi **m** elementi di **a** sono proprio gli elementi non duplicati. Nei test usare array di size 20, 40, 100.

21. algoritmo per il calcolo della somma degli elementi su una qualunque diagonale di un array rettangolare 2D; la diagonale è individuata da un indice (come nella tecnica vista con l'offset) e tale indice è il dato di input che permette di selezionare una particolare diagonale. Nei test usare array 5x6, 8x10, 10x15..

22. algoritmo che accetta in input un intero positivo (in C come **int**) e genera (e visualizza) una stringa di caratteri che contiene in ordine invertito le cifre del numero.

[Consiglio 1: se **mod** è l'operazione di modulo (in C %) allora per esempio 27953 **mod** 10 è 3; quindi l'operazione di **mod** 10 consente di ricavare l'ultima cifra di un numero.
Consiglio 2: l'operazione di divisione intera (in C la normale divisione tra dati **int**) consente di eliminare l'ultima cifra di un numero, per esempio 27953/10 è 2795.]

23. Algoritmo per la simulazione del movimento di un robot in una stanza. La stanza è pavimentata a tasselli quadrati (caselle) ed è dotata di pareti esterne e interne come in figura. Il robot si muove sempre solo di una casella alla volta, scelta tra una delle quattro caselle vicine. Il robot è in grado di "vedere", cioè di stabilire, guardando in una delle quattro direzioni (avanti, indietro, sinistra, destra), quante sono le caselle libere (in linea retta) fino alla parete. La legge con cui il robot si muove è la seguente: nel 20% dei casi il robot si muove a caso in una delle quattro caselle vicine possibili (parete permettendo); nell'80% dei casi il robot prima "vede" e individua la direzione (avanti, indietro, sinistra, destra) del movimento (scegliendo quella cui corrisponde il percorso possibile più lungo; nel caso di più percorsi di massima lunghezza, la direzione viene scelta a caso tra questi) e poi si muove (sempre di un solo passo) in quella direzione. La simulazione termina quando il robot "esce dalla porta" della stanza (in basso a sinistra nella figura). L'algoritmo deve visualizzare il percorso del robot dopo ogni passo, mostrando la stanza e la posizione del robot (comprimere le figure in modo da non aumentare a dismisura il numero di pagine della Relazione).



L'algoritmo usa la function **`rand()`** in **`stdlib`** per generare numeri casuali: si ricorda che, per esempio, se **`numero_casuale`** è un **`int`**, la chiamata **`numero_casuale=rand()%11;`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,10). Usare sempre la **`srand()`** per rendere automatica la scelta iniziale del *seed* della sequenza di numeri casuali. Effettuare almeno 5 test, variando la posizione iniziale del robot. Effettuare anche 3 test cambiando ogni volta la disposizione delle pareti nella stanza e cercando di trovare disposizioni critiche delle pareti (per esempio, un corridoio lungo e stretto può portare a movimenti oscillatori del tipo destra- sinistra).

24. Algoritmi per la cifratura/decifratura di un messaggio. Sviluppare una coppia di algoritmi, implementati come function, per crittografare e decrittografare un messaggio. L'algoritmo si basa sulla cosiddetta cifratura polialfabetica, che consiste nel trasformare il messaggio in un testo di lunghezza uguale a quella del messaggio, detto il "testo cifrato", utilizzando un altro testo (prefissato), detto "il testo delle lettere chiave". Ogni lettera del messaggio viene trasformate in una lettera del testo cifrato nel seguente modo: la lettera del messaggio viene sostituita dalla lettera dell'alfabeto successiva, se la corrispondente lettera chiave (nel testo delle lettere chiave) è B, da quella due volte successiva se la lettera chiave è C, da quella 3 volte successiva se la lettera chiave è D e così via; se la lettera chiave è A, la lettera del messaggio non viene modificata. L'alfabeto deve essere considerato come circolare, nel senso che la lettera successiva alla lettera Z è la lettera A. Per esempio, se il messaggio è THEPRESIDENT allora se il testo delle parole chiave è CODECODECODE allora il messaggio cifrato è VVHTTSVMFSQX; si osservi che in generale il testo delle lettere chiave si ottiene a partire da una parola chiave ripetuta più volte (CODE nel caso in esame). La corrispondenza: lettera del messaggio - lettera del testo cifrato - lettera chiave è evidente scrivendo i tre testi nel seguente modo (alfabeto:ABCDEFGHIJKLMNOPQRSTYVWXYZ)

ave	C	O	D	E	C	O	D	E	C	O	D	E
ssaggio	T	H	E	P	R	E	S	I	D	E	N	T

to rato	V	V	H	T	T	S	V	M	F	S	Q	X

Il main legge da tastiera il messaggio da crittografare (l'equivalente di THEPRESIDENT nell'esempio), legge da tastiera la parola chiave (l'equivalente di CODE nell'esempio; osservate che non è necessario leggere tutto il testo delle parole chiavi, perché questo può essere ottenuto a partire dalla parola chiave), chiama la function di cifratura (passando come parametro il messaggio e la parola chiave), che restituisce il teso cifrato, visualizza il testo cifrato, chiama la function di decifratura, passando come parametro il testo cifrato e la parola chiave, visualizza il messaggio decifrato, che deve coincidere con il messaggio di partenza. Usare solo lettere maiuscole e non usare la spaziatura. Usare le stringhe per rappresentare tutti i tipi di testo e le function in **`string`** (serve solo la **`strlen`**). E' utile sapere che in C, è possibile sommare "sommare i caratteri" sfruttando il fatto che il codice ASCII rappresenta i caratteri come interi e che a lettere consecutive corrispondono interi consecutivi (consiglio: determinate qual è la rappresentazione ASCII di A e poi "sottraetela" dalla rappresentazione di ogni lettera per sapere qual è la "posizione" di ogni lettera nell'alfabeto.). Fare una versione alternativa del main, in cui la parola chiave è una 5-stringa generata a caso, usando la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare gli interi casuali. Si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%(n+1);`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,..n). Usare sempre la **`srand()`** per rendere automatica la scelta iniziale del *seed* della sequenza di numeri casuali. Nella Relazione si deve riportare l'analisi della complessità di tempo dell'algoritmo (operazione dominante: confronto)

25. Algoritmo di calcolo (in simultanea) del minimo e del massimo tra gli elementi di un elenco non ordinato di stringhe. L'algoritmo usa la tecnica ricorsiva e l'approccio divide et impera. L'algoritmo deve essere implementato come function. Il main legge da tastiera la lunghezza effettiva dell'elenco (deve esser un numero tra 40 e 80), genera a caso le stringhe, fissando a caso anche la lunghezza di ciascuna stringa (deve essere un numero compreso tra 3 e 12; si osservi quindi che le stringhe non avranno la stessa lunghezza) e quindi chiama la function. La function usare le function della libreria **`string`**. Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare gli interi casuali.
26. Si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%(n+1);`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,..n). Usare sempre la **`srand()`** per rendere automatica la scelta iniziale del *seed* della sequenza di numeri casuali. Nella Relazione si deve riportare l'analisi della complessità di tempo dell'algoritmo (operazione dominante: confronto)

27. Algoritmi per la cifratura/decifratura di un messaggio. Sviluppare una coppia di algoritmi, implementati come function, per crittografare e decrittografare un messaggio. L'algoritmo si basa sulla cosiddetta cifratura polialfabetica, che consiste nel trasformare il messaggio in un testo di lunghezza maggiore o uguale a quella del messaggio, detto il "testo cifrato", utilizzando una matrice di caratteri

(prefissata), detta “matrice di sostituzione”. Il messaggio da crittografare viene dapprima partizionato in coppie di lettere adiacenti; se in tale partizionamento accade che una coppia è formata dalla stessa lettera, allora si inserisce una X tra le due. Per esempio, il messaggio è LET US MEET AT NOON viene partizionato in LE TU SM EX ET AT NO ON. Si è inserito una X tra le due E, ma non tra le due O, che si trovano in coppie diverse. Si consideri la seguente matrice di sostituzione:

8	J	E	Q	D	N	5	O
P	U	3	A	R	F	L	W
4	V	C	2	T	M	B	I
K	7	Z	S	G	X	H	Y

Ogni coppia di lettere viene crittografata nel seguente modo:

se le lettere sono nella stessa riga della matrice di sostituzione, le due lettere da inserire nel testo cifrato saranno le lettere immediatamente a destra nella stessa riga. Ogni riga è considerata circolare. Per esempio, la coppia TI viene crittografata come M4.
se le lettere sono nella stessa colonna della matrice di sostituzione, le due lettere da inserire nel testo cifrato saranno le lettere immediatamente sotto nella stessa colonna. Ogni colonna è considerata circolare. Per esempio, la coppia RG viene crittografata come TD.
se le lettere appaiono in differenti righe e colonne della matrice di sostituzione, ognuna delle due lettere sarà crittografata con la lettera nella stessa riga ma nella colonna dell’altra lettera.. Per esempio, la coppia LE viene crittografata come 35.
Il messaggio LET US MEET AT NOON viene quindi crittografato in 35VRX2NZDCR25885.
Il main legge da tastiera il messaggio da crittografare (l’equivalente di LET US MEET AT NOON nell’esempio), chiama la function di cifratura (passando come parametro il messaggio e la matrice di sostituzione), che restituisce il teso cifrato, visualizza il testo cifrato, chiama la function di decifratura, passando come parametro il testo cifrato e la matrice di sostituzione, visualizza il messaggio decifrato, che deve coincidere con il messaggio di partenza. Usare solo lettere maiuscole. Usare le stringhe per rappresentare il messaggio e il testo crittografato e decrittografato Fare una versione alternativa del main, in cui la matrice di sostituzione è una permutazione casuale della matrice precedente, usando la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare gli interi casuali per lo scambio a coppie di elementi della matrice. Si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%(n+1);`** genera un numero casuale intero (distribuzione uniforme) nell’insieme (0,1,2,..n). Usare sempre la **`srand()`** per rendere automatica la scelta iniziale del *seed* della sequenza di numeri casuali. Nella Relazione si deve riportare l’analisi della complessità di tempo dell’algoritmo (operazione dominante: confronto)

28. Algoritmo che, dato in input un numero intero positivo, verifichi se esso è un numero di Fibonacci; nel caso affermativo, l’algoritmo calcola anche l’indice del

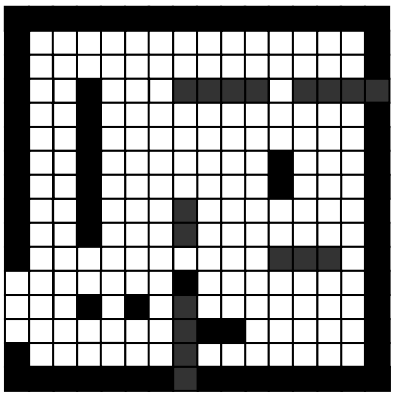
numero di Fibonacci. L'algoritmo deve essere organizzato come una function. Scrivere un main che per 10000 volte genera a caso un numero intero nell'insieme {0,1,2,..,20000}, chiama la function e visualizza il numero solo nel caso sia un numero di Fibonacci (e visualizza l'indice); alla fine il main visualizza il numero di volte in cui è stato generato un numero di Fibonacci, anche la percentuale di successo, il numero di Fibonacci generato il maggior numero di volte (se sono più di uno, devono essere visualizzati tutti), il numero di Fibonacci generato il minor numero (positivo) di volte (se sono più di uno, devono essere visualizzati tutti). Usare sempre la **`srand()`** per rendere automatica la scelta iniziale del *seed* della sequenza di numeri casuali. Nella Relazione si deve riportare l'analisi della complessità di tempo dell'algoritmo (operazione dominante: confronto).

29. Algoritmo per il calcolo della somma dei quadrati degli elementi del triangolo inferiore di un array quadrato 2D; nei test usare array 2D di size 6x6, 9x9, 11x11, i cui elementi sono numeri di tipo **`float`**
30.

31. Algoritmo legato al gioco della Dama. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra) e dispone a caso (usando la function **`rand`**) 5 Pedine Bianche e 5 Pedine Nere (per semplicità, possono stare in una qualunque casella non occupata da un'altra pedina). Si ricorda che nel gioco della Dama le Pedine di qualunque colore possono occupare solo le caselle nere della scacchiera. L'algoritmo deve visualizzare la scacchiera, mostrando una 'X' per le caselle nere non occupate, mostrando uno spazio (blank) per le caselle bianche, mostrando una 'B' per le caselle occupate dalle Pedine Bianche, una 'N' per le caselle occupate dalle Pedine Nere. L'algoritmo determina, in base alla posizione di tutte le Pedine sulla scacchiera, quali sono le Pedine Nere che possono essere catturate dalle Pedine Bianche. Fare attenzione al fatto che una Pedina Bianca cattura una Pedine Nera solo se quest'ultima si trova in una delle due caselle nere vicine lungo le due diagonali che passano per la casella dove di trova la Pedina Bianca e se la successiva casella nera esiste ed è libera. Infine l'algoritmo visualizza l'elenco delle Pedine Nere catturabili, precisando anche da quale Pedina Bianca sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco della Dama, ovvero scrivendo, per esempio: Pedina Nera in b2 catturabile da Pedina Bianca in a1, ….. Ricordare che nella notazione usuale le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga). Le Pedine possono catturare solo "in avanti" (dal basso verso l'alto, guardando la scacchiera), cioè le Pedine Bianche possono catturare muovendosi verso caselle nere contrassegnate da un indice maggiore di quello della casella nera dove di trova Pedina Bianca e che le Pedine Nere possono catturare muovendosi verso caselle nere contrassegnate da un indice minore di quello della casella nera dove di trova Pedina Nera (dall'alto verso il basso, guardando la scacchiera).

32. Algoritmo per la simulazione del movimento di un robot in una stanza. La stanza è pavimentata a tasselli quadrati (caselle) ed è dotata di pareti esterne e interne come in figura. Il robot si muove sempre solo di una casella alla volta, scelta tra una delle quattro caselle vicine. Il robot è in grado di "vedere", cioè di stabilire,

guardando in una delle quattro direzioni (avanti, indietro, sinistra, destra), quante sono le caselle libere (in linea retta) fino alla parete. La legge con cui il robot si muove è la seguente: nel 30% dei casi il robot si muove a caso in una delle quattro caselle vicine possibili (parete permettendo); nell'70% dei casi il robot prima "vede" e individua la direzione (avanti, indietro, sinistra, destra) del movimento (scegliendo quella cui corrisponde il percorso possibile più lungo; nel caso di più percorsi di massima lunghezza, la direzione viene scelta a caso tra questi) e poi si muove (sempre di un solo passo) in quella direzione. La simulazione termina quando il robot "esce dalla porta" della stanza (in basso a sinistra nella figura). L'algoritmo deve visualizzare il percorso del robot dopo ogni passo, mostrando la stanza e la posizione del robot (comprimere le figure in modo da non aumentare a dismisura il numero di pagine della Relazione).

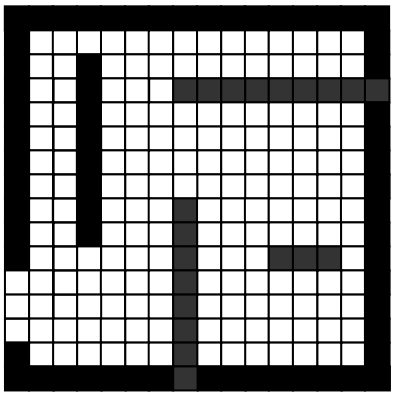


L'algoritmo usa la function **`rand()`** in **`stdlib`** per generare numeri casuali: si ricorda che, per esempio, se **`numero_casuale`** è un **`int`**, la chiamata **`numero_casuale=rand()%11;`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,10). Usare sempre la **`srand()`** per rendere automatica la scelta iniziale del *seed* della sequenza di numeri casuali. Effettuare almeno 5 test, variando la posizione iniziale del robot.

33. Algoritmo per il calcolo dell'unione di due insiemi di stringhe. Un insieme di stringhe deve essere rappresentato con un array di puntatori a **`char`**. Nei tre test usare insiemi di almeno 15 e 20 stringhe, 25 e 30 stringhe, 35 e 40 stringhe.

34. Algoritmo per la simulazione del movimento di un robot in una stanza. La stanza è pavimentata a tasselli quadrati (caselle) ed è dotata di pareti esterne e interne come in figura. Il robot si muove sempre solo di una casella alla volta, scelta tra una delle quattro caselle vicine. Il robot è in grado di "vedere", cioè di stabilire, guardando in una delle quattro direzioni (avanti, indietro, sinistra, destra), quante sono le caselle libere (in linea retta) fino alla parete. La legge con cui il robot si muove è la seguente: nel 30% dei casi il robot si muove a caso in una delle quattro caselle vicine possibili (parete permettendo); nel 70% dei casi il robot prima "vede" e individua la direzione (avanti, indietro, sinistra, destra) del

movimento (scegliendo quella cui corrisponde il percorso possibile più lungo; nel caso di più percorsi di massima lunghezza, la direzione viene scelta a caso tra questi) e poi si muove (sempre di un solo passo) in quella direzione. **In ogni caso, il robot non deve mai fare un passo in una direzione e allo step successivo un passo nella direzione opposta.** La simulazione termina quando il robot "esce dalla porta" della stanza (in basso a sinistra nella figura). L'algoritmo deve visualizzare il percorso del robot dopo ogni passo, mostrando la stanza e la posizione del robot (comprimere le figure in modo da non aumentare a dismisura il numero di pagine della Relazione).



L'algoritmo usa la function **`rand()`** in **`stdlib`** per generare numeri casuali: si ricorda che, per esempio, se **`numero_casuale`** è un **`int`**, la chiamata **`numero_casuale=rand()%11;`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,10). Usare sempre la **`srand()`** per rendere automatica la scelta iniziale del *seed* della sequenza di numeri casuali. Effettuare almeno 5 test, variando la posizione iniziale del robot. Effettuare anche 3 test cambiando ogni volta la disposizione delle pareti nella stanza e cercando di trovare disposizioni critiche delle pareti (per esempio, un corridoio lungo e stretto può portare a movimenti oscillatori del tipo destra-sinistra).

35. Algoritmo per il calcolo dell'unione di due insiemi di stringhe. Un insieme di stringhe deve essere rappresentato con un array di puntatori a **`char`**. Nei tre test usare insiemi di almeno 15 e 20 stringhe, 25 e 30 stringhe, 35 e 40 stringhe. L'algoritmo deve essere implementato come function. La function deve **necessariamente** utilizzare una function (da sviluppare) che effettua (agendo su stringhe) una azione simile a quella della function **`appartiene_a`** utilizzata nelle lezioni del Corso per l'algoritmo di unione.

36. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **`rand`**) 8 Pedoni bianchi (per semplicità,

possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **rand**) un Alfiere nero (può stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni e una 'A' per la casella occupata dall'Alfiere. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti dell'Alfiere. Fare attenzione al fatto che un Pedone potrebbe essere "protetto" da un altro Pedone e che i Pedoni "protetti" non possono essere catturati. Infine l'algoritmo visualizza l'elenco dei Pedoni catturabili, precisando anche da quale pezzo sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Alfiere in b4, Pedone in d6 catturabile da Alfiere in b4,….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).

37. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **rand**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **rand**) una Regina nera (può stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni e una 'R' per la casella occupata dalla Regina. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti della Regina. Fare attenzione al fatto che un Pedone potrebbe essere "protetto" da un altro Pedone e che i Pedoni "protetti" non possono essere catturati. Infine l'algoritmo visualizza l'elenco dei Pedoni catturabili, precisando anche da quale pezzo sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Regina in b4, Pedone in d6 catturabile da Regina in b4,….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).
38. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **rand**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **rand**) due Alfieri neri (uno può stare in una qualunque casella bianca non occupata da un altro pezzo e l'altro in una qualunque casella nera non occupata). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni e una 'A' per la casella occupata da ognuno dei due Alfieri. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti dell'Alfiere. Fare attenzione

al fatto che un Pedone potrebbe essere “protetto” da un altro Pedone o da un altro pezzo e che i Pedoni “protetti” non possono essere catturati. Infine l’algoritmo visualizza l’elenco dei Pedoni catturabili, precisando anche da quale pezzo sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Alfiere in b4, Pedone in d6 catturabile da Alfiere in f4,….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l’alto), mentre le colonne sono indicate mediante lettere dell’alfabeto (da ‘a’ ad ‘h’ da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).

39. algoritmo per il calcolo della fusione di due insiemi (ordinati) di cognomi, che utilizzi la function **`strcmp`** per determinare quale tra due cognomi precede l’altro nell’ordine alfabetico. L’intersezione dei due insiemi di input può essere non vuota; gli eventuali elementi uguali devono apparire ripetuti nell’array “fuso”.
    Ognuno dei tre insiemi di cognomi (due di input e uno di output) è un array di puntatori a **`char`**. Nei tre test usare insiemi di almeno 10 e 20 cognomi, 20 e 30 cognomi, 30 e 40 cognomi.
40. Algoritmo per la generazione a caso di informazioni su esami. La function a ogni chiamata visualizza le informazioni di uno studente universitario, del tipo:
    insegnamento PROGRAMMAZIONE 1 anno PRIMO semestre SECONDO Crediti 10 professore ROSSI assistente BIANCHI. Si osservi che un nominativo è costituito da un campo Insegnamento, un campo Anno, un campo Semestre, un campo Crediti, un campo Professore e un campo Assistente. In particolare un insegnamento deve essere scelto (a caso) in un insieme (da prefissare, di 20 insegnamenti); un anno deve essere scelto (a caso) in un insieme (PRIMO, SECONDO,TERZO); un semestre deve essere scelto (a caso) tra (PRIMO,SECONDO); i Crediti deve essere scelto a caso tra (3,5,6,9,10); il campo professore è scelto a caso in un elenco prefissato di 20 cognomi, e il campo assistente è scelto a caso in un altro elenco prefissato di 15 cognomi. L’algoritmo usa la function **`rand()`** in **`<stdlib.h>`** per generare numeri casuali e quindi per selezionare a caso un elemento in un certo insieme; si ricorda che, per esempio, se **`numero_casuale`** è un **`int`**, la chiamata **`numero_casuale=rand()%11;`** genera un numero casuale intero (distribuzione uniforme) nell’insieme (0,1,2,3,4,5,6,7,8,9,10). Usare la **`rand()`** per selezionare a caso ognuno dei campi che appaiono in un nominativo. Chiamare la function 50 volte e visualizzare le 50 informazioni sugli esami.

41. algoritmo per il gioco della roulette. L’algoritmo simula il lancio della pallina nella ruota generando un numero casuale tra 0 e 36.
    Usare la function **`rand()`** in **`<stdlib.h>`** per generare numeri casuali; per esempio, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%(n+1);`** genera un numero casuale intero (distribuzione uniforme) nell’insieme (0,1,2,..,**`n`**).
    Simulare 1000 lanci, ovvero mille puntate ipotizzando di giocare 20 euro a puntata e puntando sempre sulla stesso numero, da fissare in input, (solo numero

secco, che vince 36 volte la puntata). Alla fine, visualizzare il numero di volte in cui si è vinto, il numero di volte in cui si è perso e la cifra totale vinta o persa. Ripetere altre due volte il test precedente (1000 lanci) cambiando il numero su cui puntare

42. Algoritmo di simulazione di un viaggio a velocità non costante. Si consideri il percorso Napoli – Milano di 850 km. L'algoritmo simula il viaggio di una automobile che si muove secondo la seguente legge: l'auto può viaggiare solo a 3 velocità: 80, 100, 130 km/h.; in particolare nel 60% dei casi viaggia a 100 km/h, nel 25% dei casi viaggia a 130 km/h e nel 15% dei casi a 80 km/h. Ogni 30 minuti l'auto cambia la sua velocità; la velocità rimane costante nei successivi 30 minuti. Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare casualmente una delle 3 velocità per ogni "fetta" di tempo di 30 minuti. L'algoritmo calcola il tempo necessario per andare da Napoli a Milano e visualizza (per il primo viaggio) tutte le velocità che ha avuto. L'algoritmo ripete poi per 100 volte il viaggio Napoli – Milano (senza visualizzare tutte le velocità) visualizzando solo la velocità media del singolo viaggio e poi infine la velocità media dei 100 viaggi.

43. Algoritmo di calcolo (in simultanea) del minimo e del massimo tra gli elementi di un array 2D rettangolare e dei loro indici, con tecnica iterativa e approccio incrementale. L'algoritmo deve essere implementato come function. Nei test usare array di **`int`** (di size almeno 10x10) i cui valori sono generati a caso in (-200,….,200).

44. Algoritmo elementare di classificazione. L'algoritmo considera una scacchiera di 30x30 caselle e dispone a caso (usando la function **`rand`**) 50 oggetti (la posizione è una coppia di numeri interi). L'algoritmo posiziona sulla scacchiera, sempre generando a caso la loro posizione (usando la function **`rand`**), 2 nuovi oggetti che chiameremo centro1 e centro2. L'algoritmo deve visualizzare la scacchiera, mostrando un ' ' per le caselle non occupate, mostrando una 'X' per le caselle occupate dagli oggetti, un '1' per la casella occupata da centro1 e un '2' per la casella occupata da centro2. L'algoritmo determina i seguenti due insiemi A e B: A è l'insieme degli oggetti che sono più vicini a centro1 piuttosto che a centro2; B è l'insieme degli oggetti che sono più vicini a centro2 piuttosto che a centro1. La "vicinanza" è determinata calcolando la usuale distanza geometrica. Infine l'algoritmo visualizza di nuovo la scacchiera, ma questa volta indicando con la lettera 'A' ognuno degli oggetti dell'insieme A, con la lettera 'B' ognuno degli oggetti dell'insieme B e con la lettera 'U' gli eventuali oggetti che hanno uguale distanza sia rispetto a centro1 che a centro2. La posizione di centro1 è ancora indicata con '1' e quella di centro2 con '2'.
45. Algoritmo elementare di classificazione. L'algoritmo considera una scacchiera di 30x30 caselle e dispone a caso (usando la function **`rand`**) 50 oggetti (la posizione è una coppia di numeri interi). L'algoritmo posiziona sulla scacchiera, sempre generando a caso la loro posizione (usando la function **`rand`**), 2 nuovi oggetti che chiameremo centro1 e centro2. L'algoritmo deve visualizzare la scacchiera, mostrando un ' ' per le caselle non occupate, mostrando una 'X' per le caselle occupate dagli oggetti, un '1' per la casella occupata da centro1 e un '2' per la casella occupata da centro2. L'algoritmo determina i seguenti due insiemi A e B: A è l'insieme degli oggetti che sono più vicini a centro1 piuttosto che a centro2; B è l'insieme degli oggetti che sono più vicini a centro2 piuttosto che a

centro1. La “vicinanza” è determinata calcolando la usuale distanza geometrica. Infine l’algoritmo visualizza di nuovo la scacchiera, ma questa volta indicando con la lettera ‘A’ ognuno degli oggetti dell’insieme A, con la lettera ‘B’ ognuno degli oggetti dell’insieme B e con la lettera ‘U’ gli eventuali oggetti che hanno uguale distanza sia rispetto a centro1 che a centro2. La posizione di centro1 è ancora indicata con ‘1’ e quella di centro2 con ‘2’.

46.

47. Algoritmo di calcolo del centroide. L’algoritmo considera una scacchiera di 30x30 caselle e dispone a caso (usando la function **`rand`**) 50 oggetti (la posizione è una coppia di numeri interi). L’algoritmo deve visualizzare la scacchiera, mostrando un ‘ ’ per le caselle non occupate e mostrando una ‘X’ per le caselle occupate dagli oggetti. L’algoritmo determina il centroide dell’insieme dei 50 oggetti: il centroide è un punto della scacchiera che ha per ascissa l’intero più vicino alla media delle ascisse di tutti gli oggetti e per ordinata l’intero più vicino alla media delle ordinate di tutti gli oggetti. L’algoritmo deve visualizzare di nuovo la scacchiera, indicando, oltre alle caselle vuote e a quelle occupate, anche la posizione del centroide (indicandola con la lettera ‘C’). Infine, l’algoritmo calcola e visualizza la distanza di ognuno degli oggetti dal centroide, e determina e visualizza sia la minima distanza (indicando anche la posizione dell’oggetto a minima distanza) sia la massima distanza (indicando anche la posizione dell’oggetto a massima distanza).

48. Algoritmo per il calcolo del massimo e minimo (in valore assoluto) e lo scambio di righe di un array 2D. L’algoritmo deve essere implementato come function che ha in input un array 2D quadrato, il numero di righe e un indice di colonna; l’algoritmo calcola il massimo e minimo (in valore assoluto) elemento nella colonna di quell’indice e poi scambia la riga dove si trova il massimo con la riga in cui si trova il minimo. L’algoritmo lavora in place.
49. Algoritmo di simulazione di popolazioni di cellule. L’algoritmo considera una scacchiera di 80x80 caselle e dispone a caso (usando la function **`rand`**) 1000 oggetti (la posizione è una coppia di numeri interi). L’algoritmo deve visualizzare la scacchiera, mostrando un ‘ ’ per le caselle non occupate e mostrando una ‘*’ per le caselle occupate dagli oggetti. L’algoritmo effettua la seguente simulazione.
    Ogni casella rappresenta una “cellula”, e in particolare una casella non occupata è una cellula “bianca” e una casella occupata è una cellula “nera”. La scacchiera è dunque una “fotografia” a un certo istante (passo) della popolazione di cellule bianche e nere. Le cellule evolvono secondo la seguente legge: a ogni passo, ogni cellula considera il numero totale di cellule nere presenti nelle 9 caselle vicine (la casella della cellula stessa e le 8 caselle adiacenti, comprese le 4 sulle diagonali). Se il totale è minore di 4, allora la cellula diventa bianca al prossimo passo, mentre se il totale è maggiore di 6 allora diventa nera. Se il totale è esattamente 5, allora la cellula diventa bianca, e se il totale è esattamente 4, allora diventa nera, se totale è esattamente 6, la cellula rimane inalterata. Fare attenzione al fatto che la scacchiera deve essere aggiornata solo alla fine di ogni passo: ciò significa che l’algoritmo deve usare un array per memorizzare la scacchiera attuale e un array per memorizzare la scacchiera modificata. Inoltre si deve considerare che la scacchiera “non ha bordi”, ovvero il bordo nord e quello sud devono essere

considerati contigui e il bordo est e quello ovest devono essere considerati contigui. L'algoritmo deve visualizzare la scacchiera ai seguenti passi:
1,2,3,4,5,10,20,30,40,50,100,50,200,250,300,350,400,450,500,550,600,700,800,
indicando sempre con ' ' una cellula bianca e con '*' una cellula nera.

50. Algoritmo di simulazione di popolazioni di cellule. L'algoritmo considera una scacchiera di 80x80 caselle e dispone un oggetto ("cellula") nella posizione centrale della scacchiera. L'algoritmo effettua la seguente simulazione.
La scacchiera è una "fotografia" a un certo istante (passo) della popolazione di cellule. Le cellule evolvono secondo la seguente legge: a ogni passo una nuova cellula è aggiunta alla popolazione ed è inserita a caso in una qualunque delle caselle libere che sono però adiacenti a tutte le cellule che si trovano in quel momento sulla scacchiera. Fare attenzione al fatto che la scacchiera deve essere aggiornata solo alla fine di ogni passo: ciò significa che l'algoritmo deve usare due array, uno per memorizzare la scacchiera attuale e uno per memorizzare la scacchiera modificata. L'algoritmo deve effettuare 1000 passi e deve visualizzare la scacchiera al termine dei seguenti passi:
1,2,3,4,5,7,10,50,100,300,500,700,800,1000, indicando sempre con ' ' una casella non occupata con '*' una casella occupata da una cellula.
51. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **`rand`**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **`rand`**) una Regina nera (può stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni e una 'R' per la casella occupata dalla Regina. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti della Regina. Fare attenzione al fatto che un Pedone potrebbe essere "protetto" da un altro Pedone e che i Pedoni "protetti" non possono essere catturati. Infine l'algoritmo visualizza l'elenco dei Pedoni catturabili, precisando anche da quale pezzo sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Regina in b4, Pedone in d6 catturabile da Regina in b4,….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).
52. Algoritmo di simulazione di popolazioni di cellule. L'algoritmo considera una scacchiera di 80x80 caselle e dispone a caso (usando la function **`rand`**) 1000 oggetti (la posizione è una coppia di numeri interi). L'algoritmo deve visualizzare la scacchiera, mostrando un ' ' per le caselle non occupate e mostrando una '*' per le caselle occupate dagli oggetti. L'algoritmo effettua la seguente simulazione.
Ogni casella rappresenta una "cellula", e in particolare una casella non occupata è una cellula "bianca" e una casella occupata è una cellula "nera". La scacchiera è dunque una "fotografia" a un certo istante (passo) della popolazione di cellule bianche e nere. Le cellule evolvono secondo la seguente legge: a ogni passo, ogni cellula considera il numero totale di cellule nere presenti nelle 9 caselle vicine (la casella della cellula stessa e le 8 caselle adiacenti, comprese le 4 sulle diagonali).

Se il totale è minore di 4, allora la cellula diventa bianca al prossimo passo, mentre se il totale è maggiore di 6 allora diventa nera. Se il totale è esattamente 5, allora la cellula diventa bianca, e se il totale è esattamente 4, allora diventa nera. Fare attenzione al fatto che la scacchiera deve essere aggiornata solo alla fine di ogni passo: ciò significa che l'algoritmo deve usare un array per memorizzare la scacchiera attuale e un array per memorizzare la scacchiera modificata. Inoltre si deve considerare che la scacchiera "non ha bordi", ovvero il bordo nord e quello sud devono essere considerati contigui e il bordo est e quello ovest devono essere considerati contigui. L'algoritmo deve visualizzare la scacchiera ai seguenti passi:
1,2,3,4,5,10,20,30,40,50,100,50,200,250,300,350,400,450,500,550,600,700,800,
indicando sempre con ' ' una cellula bianca e con '*' una cellula nera.

53. Algoritmo di simulazione del gioco della tombola. L'algoritmo simula 5 giocatori, ognuno con una cartella. Ogni cartella deve essere costruita a caso dall'algoritmo e assegnata a un giocatore. Una cartella della tombola è formata da tre righe, ognuna con 5 numeri; in ogni riga può esserci un solo numero per ogni decina (si ricorda che la tombola considera i numeri da 1 a 90); inoltre, in una cartella non possono esserci numeri duplicati. Dopo aver generato le cartelle, l'algoritmo simula l'estrazione in sequenza (a caso) di un numero dal cestello (ovvero l'algoritmo genera a caso un numero intero tra 1 e 90, evitando di generare due volte lo stesso numero) e termina quando uno dei giocatori fa tombola, visualizzando il nome del giocatore. L'algoritmo visualizza anche il giocatore che ha fatto ambo, il giocatore che ha fatto terno, il giocatore che ha fatto quaterna, il giocatore che ha fatto quintina. L'algoritmo deve anche visualizzare le cartelle di tutti i giocatori sia all'inizio sia dopo l'occorrenza di ambo, terno, quaterna, quintina e tombola.
54. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **`rand`**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **`rand`**) un Alfiere e da una Torre neri (possono stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni, una 'A' per la casella occupata dall'Alfiere e una 'T' per la casella occupata dalla Torre. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti dell'Alfiere e della Torre. Fare attenzione al fatto che un Pedone potrebbe essere "protetto" da un altro Pedone o da un altro pezzo e che i Pedoni "protetti" non possono essere catturati. Infine l'algoritmo visualizza l'elenco dei Pedoni catturabili, precisando anche da quale pezzo sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Alfiere in b4, Pedone in d6 catturabile da Torre in f6,….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).

55. Algoritmo che, dato in input un numero intero positivo, verifichi se esso è un numero di Fibonacci; nel caso affermativo, l'algoritmo calcola anche l'indice del numero di Fibonacci. L'algoritmo deve essere organizzato come una function. Scrivere un main che per 10000 volte genera a caso un numero intero nell'insieme {0,1,2,..,20000}, chiama la function e visualizza il numero solo nel caso sia un numero di Fibonacci (e visualizza l'indice); alla fine il main visualizza il numero di volte in cui è stato generato un numero di Fibonacci, anche la percentuale di successo, il numero di Fibonacci generato il maggior numero di volte (se sono più di uno, devono essere visualizzati tutti), il numero di Fibonacci generato il minor numero (positivo) di volte (se sono più di uno, devono essere visualizzati tutti).
56. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **`rand`**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **`rand`**) un Alfiere e da una Torre neri (possono stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni, una 'A' per la casella occupata dall'Alfiere e una 'T' per la casella occupata dalla Torre. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti dell'Alfiere e della Torre. Fare attenzione al fatto che un Pedone potrebbe essere "protetto" da un altro Pedone o da un altro pezzo e che i Pedoni "protetti" non possono essere catturati. Infine l'algoritmo visualizza l'elenco dei Pedoni catturabili, precisando anche da quale pezzo sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Alfiere in b4, Pedone in d6 catturabile da Torre in f6,….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).
57. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **`rand`**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **`rand`**) la Regina e una Torre neri (possono stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni, una 'R' per la casella occupata dalla Regina e una 'T' per la casella occupata dalla Torre. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti della Regina e della Torre. Fare attenzione al fatto che un Pedone potrebbe essere "protetto" da un altro Pedone o da un altro pezzo e che i Pedoni "protetti" non possono essere catturati. Infine l'algoritmo visualizza l'elenco dei Pedoni catturabili, precisando anche da quale pezzo sono catturabili e utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Regina in b4, Pedone in d6 catturabile da Torre in f6,….. Ricordare che nella notazione usuale degli scacchi le righe della

scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).

58. Algoritmo di calcolo (in simultanea) del minimo e del massimo tra gli elementi di un array, con tecnica ricorsiva e approccio incrementale (**non** divide et impera). L'algoritmo deve essere implementato come function.
59. Algoritmo per determinare se un numero intero positivo **`n`** è un numero primo. Un numero intero è primo se è divisibile (resto della divisione intera = 0) solo per 1 e per sé stesso. Ecco l'inizio della sequenza dei numeri primi: 2, 3, 5, 7, 11, 17, 19, 23, 29, 31, 37,….
    Usare la function **`mod(n,x)`**, il cui prototipo è in **`<math.h>`**, per determinare il resto della divisione intera di **`n`** per **`x`**. Usare anche i seguenti accorgimenti: i numeri pari non sono primi (escluso il solo 2); è sufficiente controllare se **`n`** è divisibile per i numeri (dispari) compresi tra 3 e il più piccolo intero maggiore della radice quadrata di **`n`** (**`ceil(sqrt(n))`**). Quindi l'algoritmo deve solo controllare se **`n`** è dispari e poi controllare se **`n`** non è divisibile per tutti i numeri dispari compresi tra 3 e **`ceil(sqrt(n))`**; in tal caso il numero **`n`** è primo.
    L'algoritmo deve essere organizzato come function. Usare tale function per generare la sequenza dei primi 100 numeri primi. Inoltre il main deve dare la possibilità di inserire da tastiera un numero intero positivo e poi controllare se è primo; il main deve anche dare la possibilità di inserire da tastiera due numeri a e b (con a<b ed entrambi interi positivi) per poi identificare (visualizzandoli in output) tutti i numeri primi compresi tra a e b. **Effettuare almeno 15 test**.
60. algoritmo del cammino casuale. Un cammino si svolge su una scacchiera in cui ogni casella rappresenta una possibile posizione del cammino. Stando in una casella, è possibile muoversi solo in una delle 8 caselle vicine (caselle: nord, sud, est, ovest, nordest,nordovest, sudest,sudovest). La scelta di una tra queste otto caselle deve essere fatta a caso (generando per esempio un numero intero in {0,1,2,3,4,5,6,7}).
    Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare a caso un numero intero: si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora l'istruzione **`numero_casuale=rand()%(n+1);`**
    genera un numero casuale di tipo **`int`** (distribuzione uniforme) in {0,1,.., **`n`**}. Ogni movimento da una casella all'altra è detto passo del cammino. L'algoritmo usa un array 2D **`n`**x**`n`**, con **`n`**=40, per simulare la scacchiera. Il cammino procede finché non viene raggiunto il bordo della scacchiera. Organizzare l'algoritmo come una function che restituisce il numero di passi (lunghezza) del cammino. Il cammino inizia sempre partendo dal centro della scacchiera. Scrivere un main che per mille volte chiama la function e poi alla fine visualizza la media delle lunghezze dei cammini effettuati, la lunghezza del cammino più breve e la lunghezza del cammino più lungo. (Effettuare anche due test in cui il main chiama, 5000 volte e 10000 volte la function).
61. algoritmo di somma degli elementi di un array con tecnica ricorsiva. L'algoritmo deve essere implementato come function e deve avere, tra i dati di input, una variabile (**`flag`**) che permette di scegliere se l'algoritmo deve usare l'approccio incrementale o l'approccio divide et impera per il calcolo della somma.

62. algoritmo del massimo di un array 1D (di size qualunque, non necessariamente potenza intera di 2) di **`char`**, approccio divide et impera, implementato con tecnica ricorsiva e variante casuale. L'idea è quella dell'algoritmo implementato in **`max_radd`**, ma l'implementazione deve essere fatto con tecnica ricorsiva. La variante casuale prevede che, a ogni passo, una porzione di array (per esempio quella individuata dal primo indice =8 e dal secondo indice = 20) non venga divisa esattamente a metà, ma venga suddivisa in due parti per esempio (da 8 a k) e (da k+1 a 20) dove k è un numero casuale in (9,10,11,…,19). Quindi, per es., la prima suddivisione di array (1..n) avverrà scegliendoa caso un intero tra 2 e n-1 e non necessariamente scegliendo l'indice (n+1)/2.
63. algoritmo per la simulazione di una "mano" di poker. L'algoritmo usa la function **`rand()`** in **`<stdlib.h>`** per generare numeri casuali; se **`numero_casuale`** è un **`int`**, la chiamata **`numero_casuale=rand()%53;`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,..,52). Una carta del mazzo delle francesi è caratterizzata da due informazioni: il valore (un **`int`** tra 1 e 13) e il seme (cuori, quadri, fiori, picche. Usare la **`rand()`** per selezionare a caso cinque carte (una "mano" di poker). Fare attenzione al fatto che una stessa carta non può essere estratta due volte. L'algoritmo deve ripetere l'azione 10000 volte (ovvero per 10000 volte si deve dare una "mano" di poker) e deve contare quante volte si è ottenuto un tris d'assi servito, quante volte si è ottenuto un tris di K servito, quante volte si è ottenuto un tris di Q servito, e così via fino a quante volte si è ottenuto un tris di due servito. Dividendo il numero di successi di ogni poker per 10000 si ha una stima della probabilità di avere quel poker servito. Ripetere la simulazione nell'ipotesi di giocare all'*italiana*, ovvero usando solo le carte 7,8,9,10, Jack, Queen, King, Asso.

64. algoritmo per l'eliminazione dei duplicati in un array di interi. L'algoritmo lavora "in place", ovvero restituisce lo stesso array di input dopo aver "eliminato" i duplicati. Più precisamente, dato in input un array **`a`** di **`n`** elementi (effettivi), la function restituisce in output lo stesso array **`a`** e il numero **`m`** degli elementi effettivi (non duplicati); i primi **`m`** elementi di **`a`** sono proprio gli elementi non duplicati. Nei test usare array di size 20, 40, 100.

65. Algoritmo con numeri casuali. Data una scacchiera 8x8, con caselle bianche e nere, l'algoritmo genera a caso una posizione di una casella nera (che viene detta casella nera occupata) finché accade che tutte le caselle nere sono state occupate. Il numero complessivo di volte che è stato necessario generare una posizione a caso (una casella nera) per ricoprire completamente la scacchiera (caselle nere) è il risultato dell'esperimento. Solo le caselle nere vengono coinvolte nella simulazione. Ripetere 1000 volte l'esperimento e determinare il massimo, il minimo e la media dell'insieme dei risultati dei 1000 esperimenti. Per uno solo dei 1000 esperimenti, visualizzare, ogni 3 caselle nere generate a caso, la situazione della scacchiera, indicando con una x le caselle nere occupate, con una b le caselle bianche e con una n le caselle nere non occupate. Per generare una casella a caso è necessario generare una ascissa e una ordinata a caso. Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare le ascisse casuali; per esempio, se **`ascissa_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`ascissa_casuale=rand()%11;`** genera un numero casuale intero

(distribuzione uniforme) nell'insieme (0,1,2,..10). Fare attenzione al fatto che bisogna lavorare solo sulle caselle nere.

66. Algoritmo che, dato in input un array non ordinato di cognomi (rappresentato come un array di puntatori a **`char`**), ne elimina tutti gli elementi duplicati. Nei test (**che per questo esercizio devono essere almeno 8**) usare una array di 15, 25, 40 cognomi, con almeno 3, 6, 10 cognomi duplicati e alcuni anche triplicati. In C, usare array di puntatori a **`char`**. Usare la function **`strcmp`** per il confronto tra stringhe, il cui prototipo è in **`<string.h>`** .
67. Algoritmo per "mischiare" un mazzo di carte napoletane. L'algoritmo si basa sull'idea di scambiare effettivamente a coppie le carte del mazzo; una variabile in input permette di indicare quante volte si devono effettuare gli scambi (nei test usare i valori: 10,20,50,100,200). Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare a ogni passo gli indici delle due carte da scambiare. Nei test, partire sempre dal mazzo "ordinato" e poi visualizzare il mazzo "mischiato".
    Si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%(n+1);`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,..n).

68. Algoritmo di analisi di un testo. L'algoritmo legge da tastiera un testo in italiano di almeno 5 linee (80 caratteri per linea); poi costruisce e visualizza un array 1D dove viene memorizzato, nella componente i-sima, il numero di volte in cui la i-sima consonante dell'alfabeto è presente nel testo. L'algoritmo costruisce e visualizza anche, un array di **`char`** in cui le consonanti dell'alfabeto sono ordinate (in ordine crescente) base al loro numero di occorrenze nel testo. L'algoritmo determina se il testo è un pangramma di consonanti, ovvero contiene, almeno una volta, tutte le consonanti dell'alfabeto. Infine l'algoritmo visualizza, in ordine alfabetico, tutte le parole che compaiono nel testo e che iniziano con le prime 5 consonanti più frequenti.
69. Algoritmo l'analisi di un testo. L'algoritmo legge da tastiera un testo in italiano di almeno 5 linee (80 caratteri per linea); poi costruisce e visualizza un array 1D dove viene memorizzato, nella componente i-sima, il numero di volte in cui la i-sima vocale dell'alfabeto è presente nel testo. L'algoritmo visualizza anche, in ordine crescente, le vocali dell'alfabeto in base al loro numero di occorrenze nel testo. L'algoritmo determina se il testo è un pangramma, ovvero contiene, almeno una volta, tutte le lettere dell'alfabeto. Infine l'algoritmo visualizza, in ordine alfabetico, tutte le parole che compaiono nel testo e che iniziano con le prime 2 vocali più frequenti.
70.

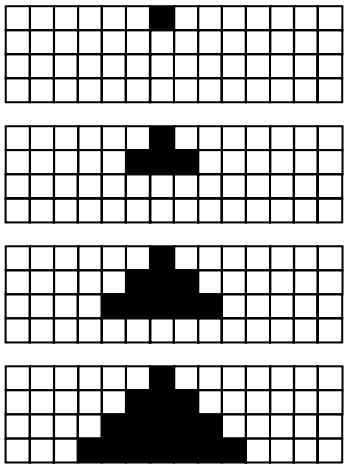
71. Algoritmo che, dato in input un numero intero positivo, verifichi se esso è un numero di Fibonacci; in caso affermativo, l'algoritmo visualizza l'indice del numero di Fibonacci (per esempio, se il numero è 8 visualizza una frase del tipo "8 è il 6-sto numero di Fibonacci") e anche il precedente numero di Fibonacci e il rapporto il numero di Fibonacci e il numero di Fibonacci precedente; in caso negativo, visualizza i due numeri di Fibonacci consecutivi tra cui si trova il numero, i loro indici e il loro rapporto (più grande su più piccolo). Nei test usare i seguenti numeri: 20, 6831, 10946, 21009, 2178309, 3324578. Effettuare anche un test in cui vengono generati a caso 50 numeri interi in (0,1,2, ….20000) e vengono forniti alla function per verificare se sono numeri di Fibonacci. Usare la

function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare gli interi casuali.

72. Si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%(n+1);`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,..n).

73. Algoritmo per la simulazione di dinamica di 'cellule'. Si consideri un array 2D 80x80. Ogni elemento dell'array è detto 'cellula'. Una cellula può essere bianca o nera. Una cellula bianca viene visualizzata con uno spazio (blank ' '), una cellula nera con 'X'. All'inizio vi è solo una cellula nera, al centro della prima riga; tutte le altre cellule sono bianche. Sviluppare un algoritmo iterativo che, a partire dalla prima riga, a ogni passo aggiorna una sola riga dell'array, ovvero al secondo passo aggiorna la seconda riga, al terzo passo aggiorna la terza riga e così via, fino all'ultima riga. Al passo i-simo, la riga i-sima viene aggiornata considerando solo la riga (i-1)-sima, secondo la seguente regola:
    la cellula j della riga i-sima diventa nera se
    a. nella riga (i-1)-sima la cellula j e la cellula j-1 sono nere e la cellula j+1 è bianca;
    b. nella riga (i-1)-sima la cellula j-1 è nera e le cellule j e j+1 sono bianche;
    c. nella riga (i-1)-sima la cellula j e la cellula j+1 sono nere e la cellula j-1 è bianca;
    d. nella riga (i-1)-sima la cellula j+1 è nera e le cellule j e j-1 sono bianche;

    Per dare un esmpio del funzionamento di algoritmi che operano su cellule, si consideri il caso in cui la regola è semplicemente:
        la cellula j della riga i-sima diventa nera se
        e. nella riga (i-1)-sima la cellula j o la cellula j-1 o la cellula j+1 sono nere;
    Si avrebbero, dopo i passi 1,2 3 e 4, le 'fotografie' dell'array mostrate in figura



    Visualizzare tutto l'array sia dopo il quarto passo, sia dopo l'ottavo passo, sia dopo il 16-simo, il 32-simo, il 64-simo, l'80-simo passo.

2. Algoritmo per la simulazione di dinamica di 'cellule'. Si consideri un array 2D 40x40. Ogni elemento dell'array è detto 'cellula'. Una cellula può essere bianca o nera. Una cellula bianca viene visualizzata con uno spazio (blank ' '), una cellula

nera con ‘X’. All’inizio vi è solo una cellula nera, al 60-simo posto della prima riga; tutte le altre cellule sono bianche. Sviluppare un algoritmo iterativo che, a partire dalla prima riga, a ogni passo aggiorna una sola riga dell’array, ovvero al secondo passo aggiorna la seconda riga, al terzo passo aggiorna la terza riga e così via, fino all’ultima riga. Al passo i-simo, la riga i-sima viene aggiornata considerando solo la riga (i-1)-sima, secondo la seguente regola:

la cellula j della riga i-sima diventa nera se

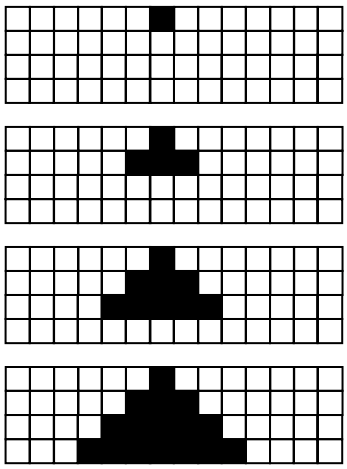
a. nella riga (i-1)-sima la cellula j e la cellula j-1 sono nere e la cellula j+1 è bianca;
b. nella riga (i-1)-sima le cellule j-1 e j+1 sono nere e la cellula j è bianca;
c. nella riga (i-1)-sima la cellula j e la cellula j+1 sono nere e la cellula j-1 è bianca;
d. nella riga (i-1)-sima la cellula j è nera e le cellule j+1 e j-1 sono bianche;
e. nella riga (i-1)-sima la cellula j+1 è nera e le cellule j e j-1 sono bianche;

Per dare un esempio del funzionamento di algoritmi che operano su cellule, si consideri il caso in cui la regola è semplicemente:

la cellula j della riga i-sima diventa nera se

f. nella riga (i-1)-sima la cellula j o la cellula j-1 o la cellula j+1 sono nere;

Si avrebbero, dopo i passi 1,2 3 e 4, le ‘fotografie’ dell’array mostrate in figura



Visualizzare tutto l’array sia dopo il quarto passo, sia dopo l’ottavo passo, sia dopo il 16-simo, il 32-simo, il 40-simo passo.

74. Algoritmo per la simulazione di dinamica di ‘cellule’. Si consideri una scacchiera in cui ogni casella rappresenta una cellula, che può essere viva o morta. Ogni cellula ha, in generale, **otto** cellule vicine (a meno che la cellula non si trovi sul bordo della scacchiera). L’algoritmo usa un array 2D **n**x**n**, con **n**=45, per simulare la scacchiera. L’algoritmo esamina **n** volte la scacchiera. Durante ogni passo aggiorna lo stato di tutte le cellule sulla scacchiera nel seguente modo: se una cellula ha esattamente tre cellule vicine vive allora la cellula deve essere posta nello stato “viva”; se una cellula ha 1,2 o 4 cellule vicine vive allora la cellula permane nello stato in cui si trova; se una cellula ha 5 o più cellule vicine vive allora la cellula deve essere posta nello stato “morta” (si noti che questa regola consente a una cellula morta di diventare viva). Fare attenzione al fatto che la scacchiera deve essere aggiornata solo alla fine di ogni passo: ciò significa che l’algoritmo deve usare un array per memorizzare la scacchiera attuale e un array

per memorizzare la scacchiera modificata. L'algoritmo visualizza tutto l'array 2D al temine del secondo, terzo, quarto, sesto, ottavo, decimo, 20-simo, 30-simo e 45-simo passo. Ogni cellula viva è visualizzata con un asterisco; ogni cellula morta con uno spazio. Usare le tre seguenti configurazioni iniziali (per i tre test): solo 7 cellule centrali della riga centrale sono vive; solo 7 cellule centrali della colonna centrale sono vive; solo le 4 cellule in posizione 22,22 22,23 23,22 23,23 sono vive.

75. Algoritmo per la simulazione di dinamica di 'cellule'. Si consideri una scacchiera in cui ogni casella rappresenta una cellula, che può essere viva o morta. Ogni cellula ha, in generale, quattro cellule vicine (a meno che la cellula non si trovi sul bordo della scacchiera). L'algoritmo usa un array 2D **n**x**n**, con **n**=45, per simulare la scacchiera. L'algoritmo esamina **n** volte la scacchiera. Durante ogni passo aggiorna lo stato di tutte le cellule sulla scacchiera nel seguente modo: se una cellula è viva, rimane viva; se una cellula ha esattamente una o esattamente quattro cellule vicine vive allora la cellula deve essere posta nello stato "viva"; altrimenti deve essere posta nello stato "morta" (si noti che questa regola consente a una cellula morta di diventare viva). Fare attenzione al fatto che la scacchiera deve essere aggiornata solo alla fine di ogni passo: ciò significa che l'algoritmo deve usare un array per memorizzare la scacchiera attuale e un array per memorizzare la scacchiera modificata. L'algoritmo visualizza tutto l'array 2D al temine del secondo, terzo, quarto, sesto, ottavo, decimo, 20-simo, 30-simo e 45-simo passo.. Ogni cellula viva è visualizzata con un asterisco; ogni cellula morta con uno spazio. Usare le tre seguenti configurazioni iniziali(per i tre test): solo la cellula centrale è viva, ovvero quella di indici 22,22; solo le cellule 22,22 e 22,23 sono vive; solo le cellule 0,0 44,44 e 44,43 sono vive
76. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **`rand`**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **`rand`**) un Cavallo nero (può stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n' per le caselle bianche e nere non occupate, mostrando una 'P' nelle caselle occupate dai Pedoni e una 'C' per la casella occupata dal Cavallo. L'algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni che possono essere catturati da uno qualunque dei possibili movimenti del Cavallo. Infine l'algoritmo visualizza l'elenco dei Pedoni catturabili, utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone in a3 catturabile da Cavallo in c4, ….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l'alto), mentre le colonne sono indicate mediante lettere dell'alfabeto (da 'a' ad 'h' da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).

77. Algoritmo per il calcolo della fusione di due insiemi (ordinati) di stringhe. Un insieme di stringhe deve essere rappresentato con un array di puntatori a **`char`**. Assumere che l'intersezione dei due insiemi di input possa essere **non** vuota; se l'intersezione è non vuota, solo un elemento, tra quelli uguali, deve comparire nell'array di output.

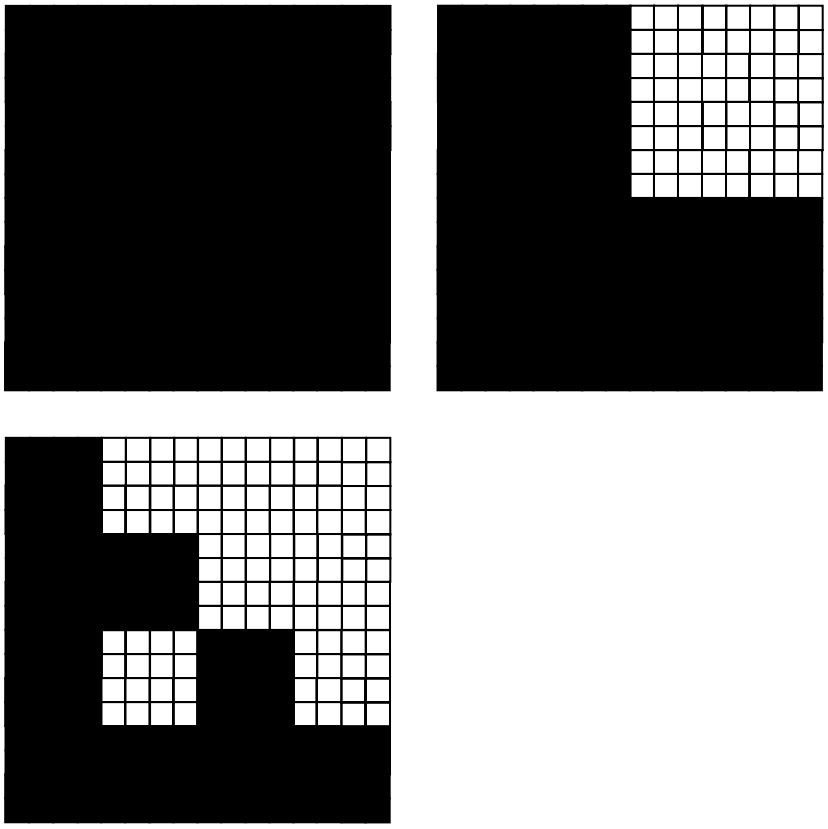
Nei tre test usare insiemi di almeno 15 e 20 stringhe, 25 e 30 stringhe, 35 e 40 stringhe.

78. Algoritmo per generare un vettore sparso di **`char`**, memorizzandolo nel formato a coordinate, senza passare per la memorizzazione standard (array 1D con tutte le componenti). Il numero complessivo **`n`** di componenti del vettore (componenti nulle (spazio) e non nulle) è un dato di input. L'algoritmo genera direttamente la rappresentazione a coordinate del vettore sparso. Il formato a coordinate memorizza un vettore utilizzando due array 1D, uno di tipo **`char`** per i valori degli elementi non nulli del vettore e l'altro di tipo **`int`** per gli indici degli elementi non nulli. Entrambi gli array hanno nnz componenti, dove nnz è il numero di componenti non nulle del vettore. Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare gli elementi non nulli del vettore (devono essere i caratteri dell'alfabeto latino). Gli indici degli elementi non nulli del vettore sono anch'essi casuali (numeri interi nell'insieme 0,1,2,...,**`n-1`**): si ricorda che, se **`i_numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata;
`i_numero_casuale=rand()%(n+1);` genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,..,**`n`**). L'array 1D degli indici deve poi essere ordinato. Nei test usare vettori con un numero di componenti **`n`** (compresi gli elementi nulli (ovvero gli spazi)) pari a 50, 80, 100. Il coefficiente di sparsità dei vettori da usare nei test deve essere del 70%, ovvero i due array 1D per la rappresentazione avranno solo il 30% di elementi rispetto al numero **`n`** di componenti del vettore.

79. Algoritmo per determinare se un numero intero positivo **`n`** è un numero primo. Un numero intero è primo se è divisibile (resto della divisione intera = 0) solo per 1 e per sé stesso. Ecco l'inizio della sequenza dei numeri primi: 2, 3, 5, 7, 11, 17, 19, 23, 29, 31, 37,….
Usare la function **`mod(n,x)`**, il cui prototipo è in **`<math.h>`**, per determinare il resto della divisione intera di **`n`** per **`x`**. Usare anche i seguenti accorgimenti: i numeri pari non sono primi (escluso il solo 2); è sufficiente controllare se **`n`** è divisibile per i numeri (dispari) compresi tra 3 e il più piccolo intero maggiore della radice quadrata di **`n`** (**`ceil(sqrt(n))`**). Quindi l'algoritmo deve solo controllare se **`n`** è dispari e poi controllare se **`n`** non è divisibile per tutti i numeri dispari compresi tra 3 e **`ceil(sqrt(n))`**; in tal caso il numero **`n`** è primo.
L'algoritmo deve essere organizzato come function. Usare tale function per generare la sequenza dei primi 100 numeri primi. Inoltre il main deve dare la possibilità di inserire da tastiera un numero intero positivo e poi controllare se è primo; il main deve anche dare la possibilità di inserire da tastiera due numeri a e b (con a<b ed entrambi interi positivi) per poi identificare (visualizzandoli in output) tutti i numeri primi compresi tra a e b. Infine il main deve generare a caso 1000 numeri interi in 0,..,, MAX_RAND e deve visualizzare la percentuale di numeri primi tra questi.
80. Algoritmo legato al gioco degli scacchi. L'algoritmo considera una scacchiera regolamentare (8x8 caselle bianche e nere, casella nera in basso a sinistra), dispone a caso (usando la function **`rand`**) 8 Pedoni bianchi (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo) e poi dispone a caso (usando la function **`rand`**) un Cavallo nero e un Pedone nero (per semplicità, possono stare in una qualunque casella non occupata da un altro pezzo). L'algoritmo deve visualizzare la scacchiera, mostrando una 'b' e una 'n'

per le caselle bianche e nere non occupate, mostrando una ‘p’ nelle caselle occupate dai Pedoni e una ‘C’ per la casella occupata dal Cavallo, una ‘P’ per la casella occupata dal Pedone nero. L’algoritmo determina, in base alla posizione di tutti i pezzi sulla scacchiera, quali sono i Pedoni bianchi che possono essere catturati da uno qualunque dei possibili movimenti del Cavallo e del Pedone neri. Infine l’algoritmo visualizza l’elenco dei Pedoni catturabili, utilizzando la notazione usuale delle posizioni nel gioco degli scacchi, ovvero scrivendo, per esempio: Pedone bianco in a3 catturabile da Cavallo nero in c4, ….. Ricordare che nella notazione usuale degli scacchi le righe della scacchiera sono indicate mediante numeri interi (da 1 ad 8, dal basso verso l’alto), mentre le colonne sono indicate mediante lettere dell’alfabeto (da ‘a’ ad ‘h’ da sinistra a destra) e che la notazione prevede di indicare prima la lettera (colonna) e poi il numero (riga).

81. Algoritmo per il calcolo dell’intersezione di due insiemi di stringhe. Un insieme di stringhe deve essere rappresentato con un array di puntatori a **`char`**. Nei tre test usare insiemi di almeno 15 e 20 stringhe, 25 e 30 stringhe, 35 e 40 stringhe.
82. Algoritmo per il calcolo dell’unione di due insiemi di stringhe. Un insieme di stringhe deve essere rappresentato con un array di puntatori a **`char`**. Nei tre test usare insiemi di almeno 15 e 20 stringhe, 25 e 30 stringhe, 35 e 40 stringhe
83. Algoritmo per la costruzione di un frattale. Si consideri un array 2D **`n`**x**`n`**, con **`n`**=32, in cui ogni casella può essere bianca (rappresentata da uno spazio ‘ ‘) o nera (rappresentata da una ’X’). All’inizio l’array è costituito solo da caselle nere. L’algoritmo esamina **`m=6`** volte la scacchiera. Durante ogni passo aggiorna lo stato di tutte le caselle usando un criterio di suddivisione nel seguente modo: al primo passo l’array 2D viene visto come costituito da 4 blocchi, ognuno di 16x16 caselle; al secondo passo, ogni blocco di 16x16 è visto come costituito da 4 blocchi di 8x8 caselle; e così via. La regola di suddivisione è la seguente: quando un blocco ‘grande’ nero viene suddiviso in 4 blocchi ‘più piccoli’ allora tre blocchi ‘piccoli’ rimangono neri e il blocco piccolo in alto a destra diventa bianco; quando un blocco ‘grande’ bianco viene suddiviso in 4 blocchi ‘più piccoli’ allora tutti i blocchi ‘piccoli’ rimangono bianchi. La figura mostra i primi tre passi dell’algoritmo, nel caso **`n=16`**.



L’algoritmo visualizza tutto l’array 2D al temine di ogni passo. Sviluppare due versioni dell’algoritmo, una iterativa e l’altra ricorsiva. Per questo esercizio è necessario solo un test per ogni versione.

84. Algoritmo per simulare il lancio di due dadi. I due dadi sono truccati, nel senso che l’uscita di ognuna delle sei facce non è equiprobabile. In particolare, il dado 1 è truccato nel seguente modo: faccia 1, 2, 3, ognuna 10% di probabilità di uscita, faccia 4 e 5, ognuna 20% di probabilità di uscita, faccia 6 30% di probabilità di uscita. Il dado 2 è truccato nel seguente modo: faccia 1 e 2, ognuna 25% di probabilità di uscita, faccia 3, 4 e 5, ognuna 15% di probabilità di uscita, faccia 6 5% di probabilità di uscita. Usare la **`rand()`** per simulare il lancio di un dado. L’algoritmo lancia 10000 volte la coppia di dadi; il punteggio di ogni lancio è dato dalla somma dei valori dei due dadi. Alla fine l’algoritmo visualizza il numero di volte che si è ottenuto ognuno degli undici punteggi possibili 2,3,4,5,..,12, e poi visualizza la lista ordinata mettendo in testa il punteggio che è uscito più volte e via decrescendo. L’algoritmo ripete poi lo stesso esperimento di diecimila lanci, usando però una coppia di dadi non truccati.
85. Algoritmo per la simulazione del lancio di tre dadi. L’algoritmo usa la function **`rand()`** in **`<stdlib.h>`** per generare numeri casuali; per esempio, se **`numero_casuale`** è dichiarato di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%10;`** genera un numero casuale intero (distribuzione uniforme) nell’insieme (0,1,2,3,4,5,6,7,8,9). Un dado ha sei facce con valori da 1 a 6. Usare la **`rand()`** per lanciare a caso 3 dadi (una chiamata **`rand`** per ogni lancio di un dado). Ripetere 30000 volte il lancio e alla fine visualizzare il numero di volte in cui si è ottenuto 2, il numero di volte in cui si è ottenuto 3, e così via fino al numero di volte in cui si è ottenuto 18. Dividere i

numeri così ottenuti per 300 (ovvero dividere per 30000 e moltiplicare per 100, che fornisce la percentuale di successo) e visualizzarli scrivendo:

2 è stato ottenuto n volte (percentuale m%);
3 è stato ottenuto p volte (percentuale q%);
……
18 è stato ottenuto r volte (percentuale s%);

86. Algoritmo di ricerca/sostituzione di una parola in un testo. L'algoritmo legge da tastiera un testo in italiano di almeno 5 linee (80 caratteri per linea), la parola (di almeno 5 caratteri) da ricercare e sostituire nel testo, la parola con cui sostituire la precedente. La sostituzione non avviene se la stringa da sostituire è parte di una parola più lunga (per esempio se si ricerca "mente", non si deve modificare "talmente").
87. Algoritmo per il calcolo dell'array 2D delle medie di un array 2D. L'algoritmo riceve in input un array 2D quadrato **A**, **m**x**n**, e genera in output un array 2D quadrato **A_diffuso**, dello stesso size di **A**. L'elemento i,j di **A_diffuso** è la media dei quattro elementi vicini dell'elemento i,j di **A** (ovvero quello sopra, quello sotto, quello a sinistra e quello a destra). Fare attenzione al caso in cui l'elento i,j si trova sul bordo dell'array (in tal caso gli elementi vicini sono tre oppure due). Nei test (almeno 4) usare array 2D di size 4x4, 8x8, 10x10, 15x15.
88. Algoritmo per la generazione di domande a caso. L'algoritmo usa la function **rand()** in **<stdlib>** per generare numeri casuali; per esempio, se **numero_casuale** è un **int**, la chiamata **numero_casuale=rand()%10;** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9). Dopo aver creato un elenco di 30 domande prestabilite (sul linguaggio C, diverse da quelle nei test ondine sul sito del corso di laurea!, e raggruppate in 5 gruppi tematici, per es. array 1D, array 2D, cicli, puntatori, stringhe), usare la **rand()** per selezionare a caso una domanda, visualizzarla e leggere la risposta da inserire da tastiera (la risposta deve essere SI oppure NO) ; dopo ogni risposta, si deve visualizzare se la risposta è corretta o sbagliata, chiedere se si vuole una nuova domanda o se si vuole concludere la seduta. Alla fine della seduta, visualizzare il numero delle risposte esatte, la percentuale di risposte esatte e il gruppo nel quale si sono registrati più errori.
89. Algoritmo per la generazione a caso di informazioni su esami. La function a ogni chiamata visualizza le informazioni di uno studente universitario, del tipo:
insegnamento PROGRAMMAZIONE 1 anno PRIMO semestre SECONDO Crediti 10 professore ROSSI assistente BIANCHI. Si osservi che un nominativo è costituito da un campo Insegnamento, un campo Anno, un campo Semestre, un campo Crediti, un campo Professore e un campo Assistente. In particolare un insegnamento deve essere scelto (a caso) in un insieme (da prefissare, di 20 insegnamenti); un anno deve essere scelto (a caso) in un insieme (PRIMO, SECONDO,TERZO); un semestre deve essere scelto (a caso) tra (PRIMO,SECONDO); i Crediti deve essere scelto a caso tra (3,5,6,9,10); il campo professore è scelto a caso in un elenco prefissato di 20 cognomi, e il campo assistente è scelto a caso in un altro elenco prefissato di 15 cognomi. L'algoritmo usa la function **rand()** in **<stdlib.h>** per generare numeri casuali e quindi per selezionare a caso un elemento in un certo insieme; si ricorda che, per esempio, se **numero_casuale** è un **int**, la chiamata **numero_casuale=rand()%11;** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,10). Usare la **rand()**

per selezionare a caso ognuno dei campi che appaiono in un nominativo. Chiamare la function 50 volte e visualizzare le 50 informazioni sugli esami.

90. Algoritmo per giocare una "mano" di scopa. L'algoritmo usa la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare numeri interi casuali. Per esempio, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora la chiamata **`numero_casuale=rand()%40;`** genera un numero casuale intero (distribuzione uniforme) nell'insieme (0,1,2,3,4,5,6,7,8,9,..,39). Una carta del mazzo delle napoletane è caratterizzata da due informazioni: il valore (un **`int`** tra 1 e 10) e il seme (bastoni, coppe, spade, denari). Usare la **`rand()`** per selezionare a caso sei carte, tre per il computer e tre per il giocatore. Fare attenzione al fatto che una stessa carta non può essere estratta due volte. Fare in modo che il computer sia in grado di "giocare" la mano; il primo a giocare è il giocatore, che da tastiera precisa quale delle sue (tre) carte giocare per prima; il computer gioca la sua carta cercando sempre di "prendere"; se non può prendere, allora gioca a caso una delle sue carte. Il giocatore gioca la sua seconda carta, precisando, sempre da tastiera, quale delle sue (due rimanenti) carte giocare; il computer gioca la sua carta cercando sempre di "prendere". Alla fine il giocatore gioca la sua ultima carta e il computer risponde. Quando il computer o il giocatore "prendono" l'algoritmo visualizza le carte prese (per esempio se a terra ci sono 3 e 5, il computer deve giocare 8 e deve essere visualizzato che il computer ha "preso" un 3, un 5 e un 8 (precisando anche il seme delle carte).
91. Algoritmo della somma degli elementi di un array 1D (di size qualunque, non necessariamente potenza intera di 2) di **`float`**, approccio divide et impera, implementato con tecnica ricorsiva e variante casuale. L'idea è quella dell'algoritmo implementato in **`somma_radd`**, ma l'implementazione deve essere fatta con tecnica ricorsiva. La variante casuale prevede che, a ogni passo, una porzione di array (per esempio quella individuata dal primo indice =8 e dal secondo indice = 20) non venga divisa esattamente a metà, ma venga suddivisa in due parti per esempio (da 8 a k) e (da k+1 a 20) dove k è un numero casuale in (9,10,11,…,19). Quindi, per es., la prima suddivisione di array (1..n) avverrà scegliendoa caso un intero tra 2 e n-1 e non necessariamente scegliendo l'indice (n+1)/2.
    Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare a caso un numero intero: si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora l'istruzione **`numero_casuale=rand()%(n+1);`**
    genera un numero casuale di tipo **`int`** (distribuzione uniforme) in {0,1,.., **`n`**}. Nei test (che devono essere almeno 6) usare i seguenti valori di **`n`**: 20,55, 64, 73, 102, 120. Per ogni test, confrontare il risultato della function sviluppata con quello restituito dalla usuale function di somma degli elementi di un array 1D, basato sull'approccio incrementale.

92. Algoritmo del cammino casuale di una pedina. Un cammino si svolge su una scacchiera in cui ogni casella rappresenta una possibile posizione del cammino della pedina. Stando in una casella, una pedina può muoversi solo nella casella a destra, nella casella a sinistra, nella casella sopra o nella casella sotto. La scelta di una tra queste quattro caselle deve essere fatta a caso (generando per esempio un numero intero in {0,1,2,3}).
    Usare la function **`rand()`**, il cui prototipo è in **`<stdlib.h>`**, per generare a caso un numero intero: si ricorda che, se **`numero_casuale`** è dichiarata di tipo **`int`**, allora l'istruzione **`numero_casuale=rand()%(n+1);`**

genera un numero casuale di tipo **`int`** (distribuzione uniforme) in {0,1,.., **`n`**}. Ogni movimento da una casella all'altra è detto passo del cammino. L'algoritmo usa un array 2D **`n`**x**`n`**, con **`n`**=30, per simulare la scacchiera. Ci sono due pedine sulla scacchiera, e all'inizio una si trova nella casella in alto a sinistra e l'altra nella casella in basso a destra. A ogni movimento di una pedina, segue il movimento dell'altra pedina. Fare attenzione al movimento di una pedina quando si trova sul bordo della scacchiera. L'algoritmo procede finché accade che le due pedine tentano di muoversi sulla stessa casella; in tal caso l'algoritmo termina (cammino terminato), restituendo il numero di passi effettuati. Organizzare l'algoritmo come una function che restituisce il numero di passi (lunghezza) del cammino. Scrivere un main che per 100 volte chiama la function e visualizza la lunghezza di ogni cammino effettuato e alla fine calcoli e visualizzi la lunghezza media del cammino casuale.