DIPARTIMENTO
DI SCIENZE APPLICATE

sezione di
Matematica e Informatica

Università
degli
Studi
di Napoli
Parthenope

prof. Mariarosaria Rizzardi

Centro Direzionale, isola C4 - 80143 Napoli
tel.: 081 547 6545
mail: mariarosaria.rizzardi@uniparthenope.it

PROGRAMMA D'ESAME di
Programmazione II e Laboratorio di Programmaz. II

A.A. 2011/2012

I due moduli integrati di *Programmazione II e Laboratorio di Programmazione II* (esame unico) rappresentano la naturale prosecuzione del percorso didattico iniziato con gli omologhi di primo livello e pertanto i prerequisiti necessari consistono prevalentemente nella conoscenza degli argomenti trattati nel corso di Programmazione I e Laboratorio di Prog. I.

L'obiettivo principale di codesto corso consiste nell'approfondire alcuni aspetti fondamentali legati alle metodologie di sviluppo ed analisi degli algoritmi, all'organizzazione logica dei dati e alla relativa implementazione nel linguaggio C.

L'esame consiste in un colloquio orale individuale sugli argomenti di teoria trattati e sugli elaborati che il candidato presenta. L'elenco analitico degli elaborati assegnati e la modalità di presentazione all'esame sono specificati, in questo documento, dopo gli argomenti trattati.

Materiale didattico

Servizio di *web Learning*: http://e-scienzeetecnologie.uniparthenope.it/.

Testi consigliati

P. Aitken, B.L. Jones – *Programmare in C: guida completa* – Apogeo

R. Sedgewick – *Algoritmi in C++* – Addison-Wesley

Testi di consultazione

T.H. Cormen, C.E. Leiserson, R.L. Rivest – *Introduzione agli algoritmi* – Jackson Libri

E. Horowitz, S. Sahni, S. Anderson-Freed – *Strutture dati in C*. McGraw-Hill Libri Italia.

1) ARGOMENTI TRATTATI

Tipi di dati scalari

Rappresentazione in memoria dei tipi di dato scalari. Tipo logico. Operatori binari e booleani. Esempi d'uso. Tipi numerici. Richiami sulla rappresentazione posizionale. Algoritmi di cambiamento di base. Il *Sistema aritmetico degli Interi*: rappresentazione per segno e modulo, per complemento a 2, rappresentazione *biased*. *Range* degli interi. Il *Sistema Aritmetico Binario Floating-point Standard IEEE 754* e sua parametrizzazione. Rappresentazione della mantissa per bit implicito. Numeri normalizzati, denormalizzati, *Nan*, underflow, overflow. Range dei numeri floating-point. Schemi di arrotondamento. Massima accuratezza statica e massima accuratezza dinamica. Errori di *roundoff* ed esempi relativi. Tipo carattere e tipo stringa.

Tipi di dati strutturati

Dati strutturati. Strutture statiche (array, record) e strutture dinamiche lineari (lista, coda, pila). Rappresentazione in *C* di liste, code e pile. Operazioni sulle strutture lineari: visita, inserimento, eliminazione di nodi. Particolari liste: circolari, bidirezionali, multiple. Rappresentazione in memoria di matrici sparse mediante liste multiple. Strutture gerarchiche (alberi). Visita *per livelli* e visita *in ordine anticipato* di un albero qualsiasi. Alberi binari. Alberi binari completi. Algoritmi di visita su alberi binari: visita anticipata, simmetrica e differita. Rappresentazione di un albero binario mediante array oppure mediante lista multipla. Alberi binari di ricerca: algoritmo di costruzione. Struttura dati *heap* e sua rappresentazione mediante array. Strutture reticolari (grafi). Rappresentazione in memoria di grafi mediante matrice di adiacenze e mediante lista di adiacenze. Algoritmi di *visita in profondità (DFS)* e di *visita in ampiezza (BFS)* di un grafo non orientato.

Linguaggio C

Variabili booleane. Operatori booleani. Operatori *bitwise*. Operazioni sugli interi mediante *operatori bitwise*. Precedenza fra operatori. Tipi di dati numerici scalari: `signed` ed `unsigned short`, `long`, `int` per gli interi, `float`, `double` e `long double` per i reali. Header file: `limits.h` e `float.h`. Variabili carattere e variabili stringa. Uso di stringhe tramite puntatori. Funzioni C di manipolazione delle stringhe. Header file `string.h`. Gestione di file testo e file binari. Input/Output su file. Allocazione statica e dinamica della memoria: funzioni `malloc()`, `calloc()`, `realloc()` e `free()`. Funzioni per copiare o spostare blocchi di memoria. Gestione, tramite puntatori, delle matrici allocate per righe o per colonne. Differenza tra allocazione (per righe o per colonne) di una matrice ed accesso (per righe o per colonne) ai suoi elementi. Gestione delle matrici (statiche e dinamiche) nel passaggio dei parametri. Tipi di dati strutturati: implementazione di strutture dinamiche (uso di array e di struttura `struct`). Strutture autoriferenti statiche e dinamiche. Strutture dinamiche nel passaggio di parametri. Puntatori a `void` e liste generiche. Funzioni ricorsive.

Ricorsione

Funzioni ricorsive ed algoritmi ricorsivi. Struttura della ricorsione: ricorsione lineare, binaria, non lineare, mutua ricorsione. Analisi della profondità di ricorsione. Esempi di algoritmi ricorsivi in *C*: fattoriale, MCD, ricerca binaria, costruzione di una lista da un array, visita di alberi binari.

Problemi di base

Ricerca di una stringa in un testo. *String Matching* in un testo tutto in memoria oppure "bufferizzato da file": algoritmo di ricerca diretta. *Algoritmo di Knuth-Morris-Pratt*. Algoritmi di ordinamento ed analisi di complessità. Algoritmi: *SelectionSort*, *ExchangeSort* (*BubbleSort*), *InsertionSort*, *MergeSort*, *QuickSort*, *HeapSort*.

2) ESERCIZI D'ESAME

Risolvere in C, i problemi di seguito elencati, scegliendo, se non specificato, l'organizzazione dei dati più idonea (allocazione statica o dinamica, struttura dei dati,…) ed il tipo di algoritmo (iterativo o ricorsivo). Analizzare se necessario la relativa complessità.

Modalità di presentazione degli elaborati: *Per ogni esercizio l'elaborato (che va stampato) consiste nel listato del programma, con una documentazione essenziale, e di alcune esecuzioni dove siano stati selezionati insiemi significativi di dati di input. Va inoltre allegato al materiale cartaceo un CD contenente i codici simbolici (file .c) dei programmi presentati. È utile presentare stampato anche l'indice degli elaborati presentati con la stessa numerazione di codesto elenco. Gli studenti possono organizzarsi in* ***gruppi****: ciascun gruppo (al massimo di 4 componenti) partecipa alla medesima seduta d'esame e presenta una sola copia degli elaborati anche se la discussione orale è individuale.*

Livelli di approfondimento degli esercizi**:** *Per accedere all'esame è necessario aver completato almeno gli esercizi di* ***livello 1****.*

Livello1: n. 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 14, 18, 19, 20, 22, 23, 24, 25, 26, 28, 29, 32, 35, 36, 38, 40, 43, 48, 54, 55, 60, 61, 66, 68, 69.

Livello 2: tutto il liv. 1 ed in più n. 7, 16, 17, 21, 26, 27, 33, 34, 41, 42, 44, 46, 50, 52, 56, 57, 64, 70.

Livello 3: tutto il liv. 2 ed in più n. 13, 15, 30, 31, 37, 39, 45, 47, 49, 51, 53, 58, 59, 62, 63, 65, 67, 71, 72, 73.

P2_01_01_AC

1. [liv.1] Scrivere una *function C*

   ```
   char low_upp(char ch)
   ```

   che cambia il carattere in input da minuscolo a maiuscolo e viceversa automaticamente.

2. [liv.1] Scrivere una *function C*

   ```
   char rotate(char ch, char n_bit)
   ```

   per ruotare di n bit (`n_bit`), verso sinistra o verso destra (rispettivamente per `n_bit<0` e per `n_bit>0`), il contenuto di una variabile `char` mediante gli operatori bitwise.

P2_01_02_AC

3. [liv.1] Scrivere una *function C* che, dopo aver estratto i bit da una variabile intera X (tipo `char`, `short` o `long`), ne calcola il relativo valore dalla formula:

   $$Val\_X = b_{n-1}2^{n-1} + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

   dove b è l'array dei bit di X. Confrontare il risultato con il valore della variabile X dichiarata una volta *signed* ed un'altra *unsigned*.

4. [liv.1] Scrivere una *function C* per estrarre dalla variabile intera X i k bit *più significativi* o *meno significativi*, dove X e k sono i parametri di input, usando:
   1) Una maschera.
   2) L'operatore di shift (>> o <<).
   3) Il prodotto o la divisione per 2.

P2_02_01_AT

5. [liv.1] Scrivere due *function C* di conversione di un intero positivo (`int`) da base 10 a base 2 mediante l'*algoritmo delle divisioni successive* realizzato rispettivamente:
   - Usando gli operatori di quoziente e resto della divisione intera;
   - Usando gli operatori bitwise.

6. [liv.1] Scrivere una *function C* di conversione di un intero positivo da base 2 a base 10, mediante l'*algoritmo delle divisioni successive*, che generi un array di caratteri contenenti le cifre decimali.
7. [liv.2] Ripetere l'esercizio precedente nel caso che l'input sia una stringa di caratteri contenenti i bit del numero.

P2_02_02_AT

8. [liv.1] Scrivere una *function C* per eseguire l'*addizione aritmetica binaria* di due numeri naturali p e q ($p,q \in \mathbb{N}$) mediante gli *operatori bitwise* (come da algoritmo in P-like in esercizio 11).
9. [liv.1] Scrivere una *function C* per eseguire la *sottrazione aritmetica* binaria* (* cioè primo operando maggiore del secondo) di due numeri naturali p e q ($p, q, p-q \in \mathbb{N}$) mediante gli *operatori bitwise*.

P2_02_04_AT

10. [liv.1] Scrivere una *function C* che, fissato il numero n di bit, calcoli la rappresentazione di un intero:
    - per complemento a 2;
    - eccesso B (B-biased).
11. [liv.1] Conoscendo la rappresentazione degli interi in C, riscrivere la *function C* per l'*addizione binaria* di due interi ($\mathbb{Z}$) mediante gli *operatori bitwise*, traducendo l'algoritmo di seguito riportato:

```
{Algoritmo di "addizione binaria" mediante operatori sui bit}
procedure binary_add(op1,op2)
rip:=1;
while rip>0
      sum:=bitXOR(op1,op2);
      rip:=bitAND(op1,op2);
      rip:=leftSHIFT(rip,1);
      op1:=sum; op2:=rip;
endwhile
```

Se l'operazione da implementare deve essere l'*addizione algebrica* (cioè deve valere anche per gli interi negativi rappresentati per complemento a 2), quale accorgimento va usato nella traduzione in *C* dell'algoritmo ... e perché.

P2_03_02_AT

12. [liv.1] Scrivere una *function C* per visualizzare la rappresentazione binaria (s,e,m) di un numero *float*. Verificare che il valore del numero ottenuto coincida con il dato iniziale.
13. [liv.3] Scrivere una *function C* di conversione di un numero reale da base 10 alla rappresentazione floating-point IEEE Std 754. L'input è una stringa di *char* del tipo `[±]X.Y` dove `X` e `Y` rappresentano rispettivamente la parte intera e la parte frazionaria del numero.

P2_03_03_AT

14. [liv.1] Scrivere delle *function C* per calcolare rispettivamente l'*epsilon macchina* del tipo `float`, del tipo `double` e del tipo `long double`, visualizzando ad ogni passo i singoli bit. Confrontare i risultati ottenuti con i valori delle variabili predefinite FLT_EPSILON, DBL_EPSILON e LDBL_EPSILON.
15. [liv. 3] Scrivere una *function C* per calcolare dalla definizione l'ULP(x) dove x è il parametro reale float di input.
16. [liv.2] Generando in modo random i bit [vedere uso di `rand()` in Materiale di supporto] di un numero reale `x` (`double x`), determinare i bit della corrispondente rappresentazione float `flx` (`float flx; flx=(float) x`). Se il numero `x` è rappresentabile nel tipo `float`, calcolarne l'errore assoluto E_A e l'errore relativo E_R di rappresentazione (considerando come esatto `double x` e come approssimante `float flx`) dalle formule:

$$E_A(flx) = |x - flx| \qquad E_R(flx) = \frac{|x - flx|}{|x|}$$

P2_03_04_AT

Visualizzare l'errore di roundoff (del tipo `float`) nei seguenti algoritmi applicando una volta l'algoritmo in aritmetica `float` ed un'altra volta in `double` oppure `long double` (come valore di riferimento), quando non sia noto il risultato esatto, per stabilire se il risultato `float` sia *di massima accuratezza*:

17. [liv.2] Scrivere una *function C* per valutare un polinomio mediante l'*algoritmo di Horner*. Applicare l'algoritmo ai dati dell'esempio 3 nelle dispense calcolando l'errore relativo. Usare una versione dell'algoritmo a precisione estesa per ottenere il valore "esatto".

18. [liv.1] Scrivere una *function C* per calcolare una somma di molti addendi dello stesso ordine di grandezza. A scelta la versione iterativa o ricorsiva dell'algoritmo di somma a gruppi. Applicare l'algoritmo al particolare problema test

$$\sum_{k=1}^{10^8} 10^{-6} = 100$$

di cui già è nota la soluzione.

19. [liv.1] Scrivere una *function C* per calcolare iterativamente la somma $\sum_{k=1}^{n} \frac{x^k}{k!} \approx e^x$ con il *criterio di arresto naturale*.

20. [liv.1] Scrivere una *function C* per calcolare una somma di addendi ordinati, rispettivamente in ordine crescente ed in ordine decrescente, mostrando qual è il modo migliore di sommare in questo caso.

21. [liv.2] Scrivere una *function C* per calcolare una somma di addendi di segno alternato, evitando l'eventuale cancellazione catastrofica.

P2_04_01_AC

22. [liv.1] Confrontando i risultati con quelli delle relative funzioni del *C* ed utilizzando per le stringhe
    - l'allocazione statica
    - l'allocazione dinamica

    scrivere una *function C* che accetta in input il numero *n* e legge da tastiera *n* caratteri (uno alla volta) costruendo la stringa che li contiene (output) senza usare `strcat(...)`.

23. [liv.1] Confrontando i risultati con quelli delle relative funzioni del C ed utilizzando per le stringhe
    - l'allocazione statica
    - l'allocazione dinamica

    scrivere una *function C* che restituisca la concatenazione di due stringhe date in input senza usare `strcat(...)`. È a scelta restituire la concatenazione delle due stringhe in una terza variabile (parametro di output o function stessa) oppure nella prima delle due variabili di input.

P2_04_02_AC

24. [liv.1] Confrontando i risultati con quelli delle relative funzioni del C, scrivere una *function C* che restituisca la prima occorrenza di una sottostringa in una stringa senza usare `strstr(...)`.

25. [liv.1] Usando l'allocazione dinamica e le funzioni *C* per manipolare le stringhe, scrivere una *function C* che restituisca la posizione di tutte le occorrenze di una sottostringa in una stringa ed il loro numero totale.

26. [liv.2] Utilizzando per le stringhe
    - l'allocazione statica
    - l'allocazione dinamica

    scrivere una *function C* che elimini tutte le occorrenze di una data sottostringa in una stringa col minimo numero di spostamenti di blocchi di memoria.

27. [liv.2] Utilizzando per le stringhe
    - l'allocazione statica
    - l'allocazione dinamica

    scrivere una *function C* che sostituisca in un testo tutte le occorrenze di una data sottostringa S_1 con un'altra S_2 (le due sottostringhe possono avere anche lunghezze diverse). [liv.3]: … con il *minimo* numero di spostamenti di blocchi di memoria.

P2_05_02_C

28. [liv.1] A partire dalla matrice A(m×n), del tipo sotto indicato, allocata staticamente [risp. dinamicamente] per righe visualizzarne gli elementi per colonne

$$A_{4\times 6} = \begin{pmatrix} 11 & 12 & 13 & 14 & 15 & 16 \\ 21 & 22 & 23 & 24 & 25 & 26 \\ 31 & 32 & 33 & 34 & 35 & 36 \\ 41 & 42 & 43 & 44 & 45 & 46 \end{pmatrix}$$

P2_05_03_C

29. [liv.1] Scrivere una *function C* che restituisca la matrice C *prodotto righe×colonne* [vedi pdf delle dispense] di due matrici rettangolari A e B le cui dimensioni sono stabilite in input (usare per tutte le matrici l'allocazione dinamica e generarle come numeri reali random). C'è qualche preferenza nell'usare `malloc()` o `calloc()` rispettivamente per A, B o C? Verificare se i tempi di esecuzione, per la sola allocazione e totali, sono gli stessi (vedere `calcola_tempo.c` in "Materiale utile!" per misurare il tempo in un programma C).

30. [liv.3] Ripetere l'esercizio sul *prodotto righe×colonne*, una prima volta, allocando tutte le matrici in memoria per colonna ed, una seconda volta, per riga. Per ciascun tipo di allocazione in memoria, usare due *function C* per il prodotto righe×colonne: una che acceda a tutte le matrici per colonne e l'altra per righe. Confrontare i tempi d'esecuzione delle due modalità di accesso alle matrici rispetto alla loro allocazione in memoria, deducendo quindi il tipo di accesso più efficiente rispetto al criterio di memorizzazione.

P2_06_01_AT

31. [liv.3] Determinare il numero totale di occorrenze di un pattern in un testo, usando per la ricerca l'*algoritmo KMP*.

P2_07_01_AC

32. [liv.1] Scrivere una *function C* che legga, mediante una variabile "buffer" di 200char, un file testo e lo visualizzi sullo schermo 40 char per riga e 25 righe per ogni schermata, fermandosi finché non viene premuto un tasto per continuare.

33. [liv.2] Scrivere un *programma C* che crei un file binario "studenti1.dat" contenente le seguenti informazioni:
    - cognome e nome (30c) c = char
    - matricola (ccc/ccccc)
    - numero degli esami superati (short)
    - media pesata degli esami (float)
    - crediti acquisiti (short).

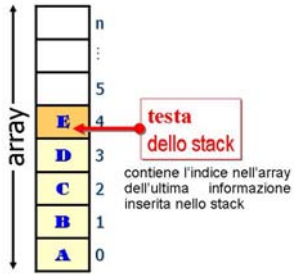
    Il file contiene le informazioni già ordinate per matricola. Scrivere una *function C* che, a partire da un file di aggiornamento relativo ad un certo esame (per esempio, "esameProg2.dat") contenente gli studenti che l'hanno superato ed i relativi voti, crei il file "studenti2.dat" aggiornato.

P2_07_02_AC

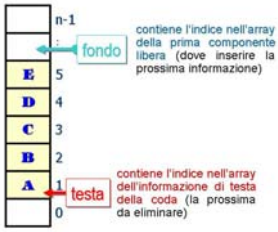
34. [liv.2] Scrivere una *function C* per la *ricerca diretta* di tutte le occorrenze di un pattern in un testo dove:
    - il testo è memorizzato in un file;
    - la lettura del file avviene "a pezzi" mediante un array-buffer;
    - il pattern è definito in input (e costruito dinamicamente).

P2_08_03_T

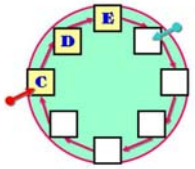
35. [liv.1] Simulare in *C* la gestione di una **pila** (**stack**) tramite array statico (può essere anche un array di struct) creando le funzioni di manipolazione `push()` [inserimento] e `pop()` [eliminazione]. Il programma deve prevedere un menù che consenta di scegliere l'operazione da eseguire.



36. [liv.1] Simulare in *C* la gestione di una **coda** (**queue**) tramite array statico (può essere anche un array di struct) creando le funzioni di manipolazione `enqueue()` [inserimento] e `dequeue()` [eliminazione]. Il programma deve prevedere un menù che consenta di scegliere l'operazione da eseguire. Le informazioni NON vanno spostate!



37. [liv.3] Simulare in *C* la gestione di una **coda** (**queue**) tramite array statico **circolare** (può essere anche un array di struct) creando le funzioni di manipolazione `enqueue()` [inserimento] e `dequeue()` [eliminazione]. Il programma deve prevedere un menù che consenta di scegliere l'operazione da eseguire.



P2_08_04_T

38. [liv.1] Simulare in *C* l'algoritmo di visita di una *lista lineare* già memorizzata mediante un array statico di `struct` (come nella tabella in basso) in cui il primo campo contiene l'informazione ed il secondo contiene il *link* al nodo successivo (in questo caso il *link* è l'indice di una componente dell'array). Memorizzando nell'array i dati come mostrato nella figura che segue, l'output del programma consiste nell'elenco di nomi ordinato alfabeticamente.

39. [liv.3] Simulare in *C* la *gestione delle camere di un albergo* mediante liste lineari rappresentate su un array di `struct`: i principali campi sono le "informazioni" (numero di camera, cliente, etc.) ed i "link" (puntatori ai nodi della lista). [Suggerimento: L'array di struct corrisponde alla memoria in cui allocare la lista delle camere libere

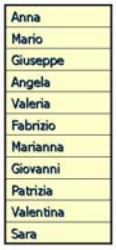
(`ListaLibera`) e la lista delle camere occupate (`ListaDati`). È necessario creare prima la `ListaLibera`, inizializzando l'array dei link in modo che ogni componente punti alla componente successiva. Ogni nodo da inserire nella `ListaDati`, quando una camera viene assegnata ad un cliente, è prelevato dalla testa della `ListaLibera` ed inserito nella testa della `ListaDati`; mentre il nodo da eliminare dalla `ListaDati`, quando una particolare camera viene liberata, è restituito alla `ListaLibera` (in testa) per poter essere riutilizzato in seguito.]

P2_08_06_C

40. [liv.1] Realizzare la gestione di una lista lineare mediante menù (visualizzazione mediante visita, inserimento in testa, inserimento in mezzo, eliminazione in testa, eliminazione in mezzo) implementando la lista lineare con una struttura autoriferente dinamica.

41. [liv.2] A partire dalla versione ricorsiva di costruzione di una lista in *C*, scriverne la versione iterativa.

42. [liv.2] Scrivere una *function C* per costruire una *lista ordinata in ordine alfabetico* a partire da un elenco di nomi in ordine casuale, come nel seguente.



P2_08_08_AT

43. [liv.1] Realizzare in *C* le funzioni per la gestione, mediante menù, delle strutture dati *pila* e *coda* mediante *lista lineare dinamica e generica* con [rispettivamente senza] nodo sentinella.

44. [liv.2] Realizzare in *C* le funzioni per la gestione, mediante menù, delle strutture dati *catena* e *lista bidirezionale* mediante *lista lineare dinamica e generica* con [rispettivamente senza] nodo sentinella.

45. [liv.3] Implementare in *C* il prodotto righe×colonne di due matrici sparse rappresentate tramite liste multiple. Scegliere per ciascuna matrice la rappresentazione più idonea.

P2_09_01_T

46. [liv.2] Scrivere *function C* per la costruzione e visita per livelli di un *albero qualsiasi* rappresentato mediante array. [Suggerimento: la struct che definisce il generico nodo dell'albero, come nella figura sotto, deve contenere i seguenti campi: l'informazione, il suo grado ed un array di puntatori (ai nodi figli) di dimensione pari al massimo grado dei nodi che si suppone noto]

47. [liv.3] Scrivere *function C* per la costruzione e visita per livelli di un *albero qualsiasi* rappresentato mediante liste multiple. [Suggerimento: i puntatori ai figli risiedono in un array dinamicamente allocato e indirizzato dall'unico campo puntatore del nodo dell'albero (`pt_figli`) come nella figura che segue]

P2_09_03_T

48. [liv.1] Scrivere le *function C* per la visita (*preorder*, *inorder* e *postorder*) di un *albero binario* rappresentato mediante array.

49. [liv.3] Scrivere *function C* per la costruzione e visita (*preorder*, *inorder* e *postorder*) di un *albero binario* rappresentato mediante liste multiple.

P2_09_05_T

50. [liv.2] Scrivere *function C* iterativa per la costruzione di un *albero binario di ricerca* rappresentato mediante array.

51. [liv.3] Scrivere *function C* iterativa per la costruzione di un *albero binario di ricerca* rappresentato mediante liste multiple.

P2_09_06_T

52. [liv.2] Scrivere *function C* iterativa per la costruzione di un *heap* rappresentato mediante array.

53. [liv.3] Scrivere *function C* iterativa per la trasformazione in un *heap* di un *albero binario* rappresentato mediante array.

P2_10_02_T

54. [liv.1] Scrivere *function C* per la costruzione di un *grafo non orientato* mediante *matrice di adiacenze*: in input per ogni nodo sono specificati quelli *adiacenti*. Scegliendo in input un nodo, scrivere una *function C* che restituisca il suo grado.

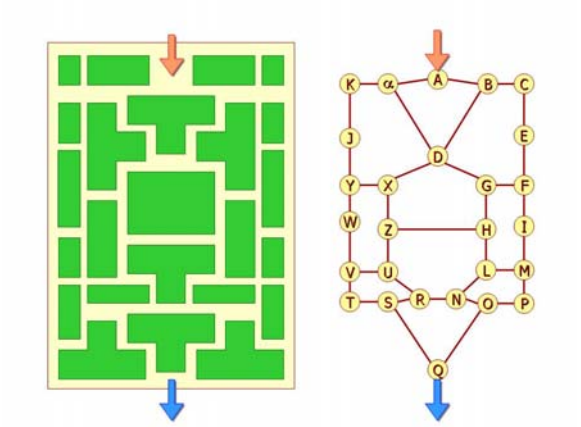
55. [liv.1] Scrivere *function C* per la costruzione di un *grafo orientato* mediante *matrice di adiacenze*: in input per ogni nodo sono specificati quelli *raggiungibili*. Scegliendo in input un nodo, scrivere una *function C* che restituisca il numero degli archi uscenti e quello degli archi entranti.

56. [liv.2] Scrivere *function C* per la costruzione di un *grafo non orientato* mediante liste di adiacenze: in input per ogni nodo sono specificati quelli *adiacenti*.

P2_10_03_T

57. [liv.2] Scrivere *function C* per la *visita in ordine anticipato* di un *albero qualsiasi*.

58. [liv.3] Scrivere *function C* per la visita di un *grafo* mediante l'algoritmo *Depth First Search* iterativo. Applicare l'algoritmo al grafo che descrive il seguente labirinto per determinare un cammino che parte dall'ingresso A e termina nell'uscita Q.



P2_10_04_T

59. [liv.3] Scrivere *function C* per la visita di un *grafo* mediante l'algoritmo *Breadth First Search*. Applicare l'algoritmo per stabilire se esiste un cammino che unisce due nodi qualsiasi di un grafo stabiliti in input.

P2_11_01_T

60. [liv.1] Scrivere delle *function C* (rispettivamente iterativa e ricorsiva) per calcolare (con *ricorsione* sia *lineare* sia *binaria*) la somma delle componenti di un array.

61. [liv.1] Scrivere delle *function C* (rispettivamente iterativa e ricorsiva) per calcolare (con *ricorsione* sia *lineare* sia *binaria*) la potenza intera x^n di un numero reale.

P2_11_03_AT

62. [liv.3] Scrivere due *function C* (rispettivamente iterativa e ricorsiva) per valutare un polinomio mediante *algoritmo di Horner*.
63. [liv.3] Scrivere due *function C* (rispettivamente iterativa e ricorsiva) per visitare una lista lineare, stampando le informazioni.
64. [liv.2] Scrivere due *function C* (rispettivamente iterativa e ricorsiva) per visitare un albero binario (risp. In ordine anticipato, simmetrico e differito) stampando le informazioni.
65. [liv.3] Approssimare lo zero di una funzione, monotona in un intervallo [a,b] e tale che f(a)f(b)<0, mediante *algoritmo di bisezione* ricorsivo.

P2_12_01_T

66. [liv.1] Scrivere due *function C* (rispettivamente iterativa e ricorsiva) per implementare l'*algoritmo Selection Sort* su un array di struttura, sia mediante scambi reali sia mediante scambi virtuali.
67. [liv.3] Scrivere una *function C* per implementare l'*algoritmo Selection Sort* su una lista bidirezionale.
68. [liv.1] Scrivere una *function C* per implementare l'*algoritmo Bubble Sort* su un array di struttura, sia mediante scambi reali sia mediante scambi virtuali.
69. [liv.1] Scrivere una *function C* per implementare l'*algoritmo Insertion Sort* su un array di struttura.

P2_12_02_T

70. [liv.2] Scrivere *function C* per implementare l'*algoritmo Merge Sort* su un array in versione iterativa e ricorsiva.
71. [liv.3] Scrivere una *function C* per implementare l'*algoritmo Merge Sort* su una lista lineare.
72. [liv.3] Scrivere una *function C* per implementare l'*algoritmo Quick Sort*.
73. [liv.3] Scrivere una *function C* per implementare l'*algoritmo Heap Sort*.