1. Dal documento "Programmazione_2022_Appello_Luglio.pdf":

Esercizio 1 (2 punti):
L'indirizzo di A[5][4] sarà 100 + (5*8 + 4) = 144.*
*Spiegazione: A è un array 6x8. Ogni riga occupa 8 elementi. Per arrivare alla 6a riga (indice 5), saltiamo 5*8=40 elementi. Poi aggiungiamo 4 per arrivare alla 5a colonna.

Esercizio 2 (5 punti):
Il codice compila e stampa:
15
9
16

Spiegazione:

- La prima chiamata a fun(2) restituisce 3, quindi y=3.
- printf(10/3*3) stampa 15.
- Dentro il blocco, fun(3) restituisce 6, poi fun(6) restituisce 12. 12+3=15.
- L'ultima chiamata fun(3) (y dalla prima chiamata) restituisce 6. 6+10=16.

Esercizio 3 (8 punti):

```
int check_presence(int* A, int* B, int sizeA, int sizeB) {
    if (sizeA == 0) return 1;  // Tutti gli elementi di A sono stati trovati
    if (sizeB == 0) return 0;  // B è finito ma restano elementi in A

    if (A[0] == B[0] || A[0] == -B[0]) {
        return check_presence(A+1, B+1, sizeA-1, sizeB-1);
    } else {
        return check_presence(A, B+1, sizeA, sizeB-1);
    }
}
```

Esercizio 4 (6 punti):

```
void clone_list(Lista *srcPtr, Lista **destPtr) {
    *destPtr = NULL;
    if (srcPtr == NULL) return;

    *destPtr = (Lista*)malloc(sizeof(Lista));
    if (*destPtr == NULL) return;  // Gestione errore di allocazione
```

```
    (*destPtr)->value = srcPtr->value;
    (*destPtr)->nextPtr = NULL;

    clone_list(srcPtr->nextPtr, &((*destPtr)->nextPtr));
}
```

Codice nel main (1 punto):

```
Lista *original = /* ... */;  // lista originale
Lista *clone = NULL;
clone_list(original, &clone);
```

Esercizio 5 (6 punti):

```
int search(BTree *ptr, int val) {
    if (ptr == NULL) return 0;
    if (ptr->valore == val) return 1;
    if (val < ptr->valore)
        return search(ptr->leftPtr, val);
    else
        return search(ptr->rightPtr, val);
}
```

Esercizio 6 (3 punti):

- Visita simmetrica (in-order): sinistra, radice, destra
- Visita anticipata (pre-order): radice, sinistra, destra
- Visita posticipata (post-order): sinistra, destra, radice

Esempio:

```
    4
   / \
  2   6
 / \ / \
1  3 5  7
```

Simmetrica: 1 2 3 4 5 6 7
Anticipata: 4 2 1 3 6 5 7
Posticipata: 1 3 2 5 7 6 4

2. Dal documento "Programmazione_2022_Appello_Giugno.pdf":

Esercizio 1 (2+4 punti):
a) Il codice non compila. L'operatore && non è valido in questo contesto.
b) Il codice stampa:
7
7

Spiegazione:

- q punta ad a, p viene impostato a q, quindi p punta ad a
- **pp = 7 imposta a a 7
- Cambiare q non influisce su p, quindi **pp = 3 imposta ancora a a 3

Esercizio 2 (1+8 punti):
PRE: A è un array di interi, dim è la sua dimensione, dim > 0

```
void rimuovi_triple(int A[], int *dim) {
    if (*dim <= 2) return;

    int write = 1;
    for (int read = 1; read < *dim - 1; read++) {
        if (A[read-1] != A[read] || A[read] != A[read+1]) {
            A[write++] = A[read];
        }
    }
    A[write++] = A[*dim - 1];
    *dim = write;
}
```

Esercizio 3 (6 punti):

```
void print_list(Lista *ptr) {
    if (ptr == NULL) return;
    print_list(ptr->nextPtr);
    printf("%d ", ptr->valore);
}
```

Esercizio 4 (2 punti):

```
enum giorni day = Mar;
printf("%d\n", day);
```

Esercizio 5 (6+2 punti):

```
void ord_insert(Lista **head, int value) {
    Lista *newNode = (Lista*)malloc(sizeof(Lista));
    if (newNode == NULL) return;  // Gestione errore di allocazione
    newNode->valore = value;
    newNode->nextPtr = NULL;

    if (*head == NULL || (*head)->valore >= value) {
        newNode->nextPtr = *head;
        *head = newNode;
        return;
    }

    Lista *current = *head;
    while (current->nextPtr != NULL && current->nextPtr->valore < value) {
        current = current->nextPtr;
    }
    newNode->nextPtr = current->nextPtr;
    current->nextPtr = newNode;
}
```

Nel main:

```
Lista *head = NULL;
int new_value = 5;
ord_insert(&head, new_value);
```