Domanda A (8 punti) Si consideri la seguente funzione ricorsiva con argomento un intero $n \geq 0$

```
exp(n)
   if n = 0
      return 0
   else
      return exp(n-1) + exp(n-1) + 1
```

$] - 2^0$

$\alpha(n-1) + \alpha(n-1) + 1 \sim 2^{n-1} + 2^{n-1} + 1 - 2^0 \sim 2^n$

1 Dimostrare induttivamente che la funzione calcola il valore $2^n - 1$. Inoltre, determinare la ricorrenza che esprime la complessità della funzione e mostrare che la soluzione è $\Omega(2^n)$. È anche $O(2^n)$? Motivare le risposte.

2 3

$2^n - 1 \sim 2^n$

1

$n = 3 \qquad 2^2 + 2^2 + 1 = 9 \qquad 2^3 = 2^n$

Soluzione: Per quanto riguarda la funzione calcolata, procediamo per induzione su n. Se $n = 0$, la funzione ritorna 0 che è in effetti $2^0 - 1 = 1 - 1 = 0$. Se $n > 0$, allora per ipotesi induttiva, $exp(n-1) = 2^{n-1} - 1$. Quindi $exp(n) = exp(n-1) + exp(n-1) + 1 = 2^{n-1} - 1 + 2^{n-1} - 1 + 1 = 2 \cdot 2^{n-1} - 1 = 2^n - 1$, come desiderato.

2 $\quad \Omega(2^n) \longrightarrow T(n) \geq d2^n \qquad \begin{pmatrix} \forall n \geq 0 \\ d > 0 \end{pmatrix}$

$exp(n-1) + exp(n-1) + 1 \qquad \sim T(n)?$

$T(n-1) + T(n-1) + 1 = 2T(n-1) + 1$

$[T(n) = \Omega(2^n) \rightarrow T(n) \geq d2^n]$

$2d(2^n - 1) + 1 \geq d2^n$

$2d2^n - 2d + 1 \geq d2^n$

$d2^n - 2d + 1 \geq 0 \qquad \rightarrow \qquad d \geq 0, \forall n \geq 0$

$[T(n) = O(2^n) \rightarrow T(n) \leq c2^n]$

$$2T(n-1)+1 \sim 2c(2^n-1)+1$$

$$2c(2^n-1)+1 \le c2^n \qquad c>0,\ \forall n \ge 0$$

$$2c2^n - 2c + 1 \le c2^n$$

$$c2^n - 2c + 1 \le 0 \quad \Rightarrow \text{IMPOSSIBILE!}$$

Vale anche $T(n) = O(2^n)$. Tuttavia, il tentativo di provare direttamente per induzione che $d > 0$ e n_0 tali che $T(n) \le d2^n$ per un'opportuna costante $d > 0$ e $n \ge n_0$ fallisce. Si riesce invece a dimostrare che $T(n) \le d(2^n - 1)$. Infatti:

$$\begin{aligned} T(n) &= 2T(n-1) + c && \text{[dalla definizione della ricorrenza]} \\ &\le 2d(2^{n-1} - 1) + c && \text{[ipotesi induttiva]} \\ &= d2^n - 2d + c \\ &= d(2^n - 1) - d + c \\ &\le d2^n \end{aligned}$$

INDUZIONE FORTE
||
RAFFORZO ARGOMENTO INDUTTIVO...

$> T(n) \le d(2^n)$

$\Downarrow$ FORTE

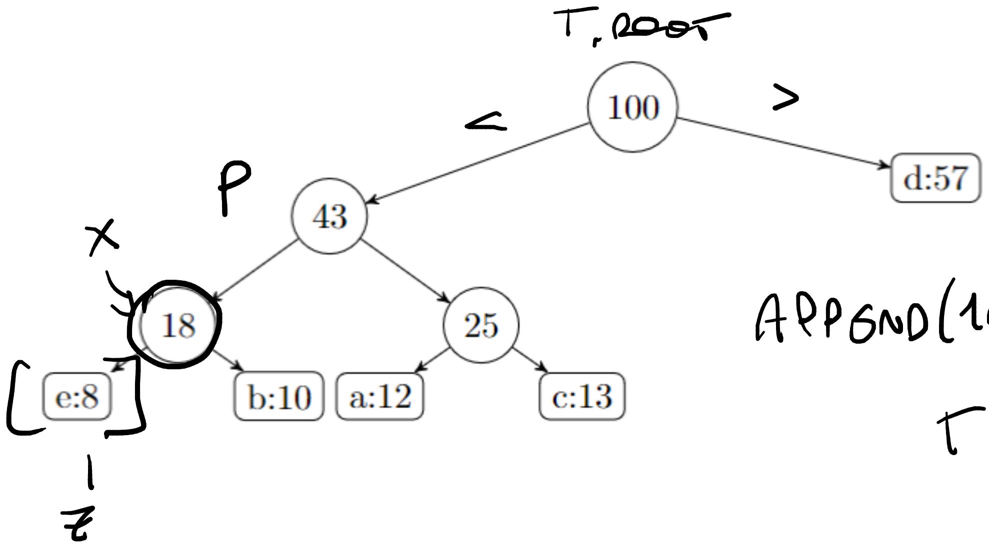
$T(n) \le d(2^n - 1)$

È facile dimostrare con il metodo di sostituzione che $T(n) = \Omega(2^n)$, ovvero che esistono $d > 0$ e n_0 tali che $T(n) \ge d2^n$, per $n \ge n_0$. Si procede per induzione su n:

$$\begin{aligned} T(n) &= 2T(n-1) + c && \text{[dalla definizione della ricorrenza]} \\ &\ge 2d2^{n-1} + c && \text{[ipotesi induttiva]} \\ &= d2^n + c \\ &\ge d2^n \end{aligned}$$

qualunque sia $d > 0$ e n.

Esercizio 1 (9 punti) Realizzare una funzione *append*(T, x, z) che dato un albero binario di ricerca T, un suo nodo *foglia* x ed un nuovo nodo z, verifica se è possibile inserire z come figlio (destro o sinistro) di x in T. In caso affermativo ritorna l'albero esteso, altrimenti ritorna `nil`. (Assumere che i nodi dell'albero abbiano gli usuali campi $x.left$, $x.right$, $x.p$, $x.key$ e la radice sia $T.root$.) Valutarne la complessità.



APPEND(100, 18, 8)

T, x, z

IF (T.ROOT == NIL) RETURN NIL

100 [R = T.ROOT] [P = X.P] 43

WHILE (R <> NIL)

IF (R > P)

R = R.LEFT

ELSE

R = R.RIGHT

IF (Z < X)

X.LEFT = Z

ELSE

X.RIGHT = Z

```
append(T,x,z)

    # verifica che z.key sia <= delle chiavi degli antenati di x
    # per i quali x e' nel sottoalbero dx e
    y = x

    while (y.p <> nil) and
          ( (y == y.p.left   and  z.key <= y.p.key) or
            (y == y.p.right  and  z.key >= y.p.key) )
        y = y.p

    # se siamo arrivati alla radice, significa che la condizione e' soddisfatta
    if (y.p == nil)
        # e z viene inserito come figlio dx o sx
        z.p = x
        if (z.key <= x.key)
            x. left = z
        else
            x.right = z

        return T
    else                                    # altrimenti si ritorna nil
        return nil
```

→ O(h)

$O(h) = \left(\Theta(\lfloor 2 \log_n + 1 \rfloor) \right)$ PUÒ ESSERE SBILANCIATO ...

La complessità è chiaramente $O(h)$, dove h è l'altezza dell'albero. Infatti, il ciclo while effettua un numero di iterazioni pari al numero di antenati di x, e quindi limitato dall'altezza dell'albero. E ciascuna iterazione ha costo costante.

SCRIVI QUESTO!

Esercizio 1 (9 punti) Realizzare una funzione booleana `checkSum(A,B,C,n)` che dati tre array `A[1..n]`, `B[1..n]` e `C[1..n]` con valori numerici, verifica se esistono tre indici i, j, k con $1 \leq i, j, k \leq n$ tali che $A[i] + B[j] = C[k]$. Valutarne la complessità.

$i < j \leq k$

$(A) + (B) = (C)$

A[i] + B[j] <> C[k]

```
CHECKSUM (A, B, C, n)
SORT(A)
SORT(B)
SORT(C)
i=0, j=0, k=0
WHILE (i < N AND j < N AND k < N)
  IF (A[i] + B[j] < C[k]) {
      i++;
      j--; }
  IF (A[i] + B[j] > C(k)) {
      i--;
      j++; }
  k++;
}
RETURN TRUE;
```

```
checkSum(A,B,C,n):
    # ordina B e C
    Sort(B)
    Sort(C)

    i = 1
    found = false

    while (i <= n) and not found
        # per ogni A[i] verifica se per qualche elemento B[j] vale che
        # A[i]+B[j]=C[k] scorrendo B e C

        j = k = 1
        while (j <= n) and (k <= n) and (A[i] + B[j] <> C[k])
            if (A[i] + B[j] < C[k]):
                j++
            else
                k++
        if (j<=n and k<=n)
            found = true
        else
            i++

    return found
```

Esercizio 2 (8 punti) Per $n > 0$, siano dati due vettori a componenti intere $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$. Si consideri la quantità $c(i,j)$, con $0 \leq i \leq j \leq n-1$, definita come segue:

$$c(i,j) = \begin{cases} a_i & \text{se } 0 < i \leq n-1 \text{ e } j = n-1, \\ b_j & \text{se } i = 0 \text{ e } 0 \leq j \leq n-1, \\ c(i-1,j-1) \cdot c(i,j+1) & 0 < i \leq j < n-1. \end{cases}$$

1

2

Si vuole calcolare la quantità $m = \min\{c(i,j) : 0 \leq i \leq j \leq n-1\}$.

1. Si fornisca il codice di un algoritmo iterativo bottom-up per il calcolo di m.
2. Si valuti la complessità esatta dell'algoritmo, associando costo unitario ai prodotti tra numeri interi e costo nullo a tutte le altre operazioni.

① COMPUTE (A, B)

```
N = LENGTH(A)
FOR I = 1 TO N-1
    C[I, N-1] = A[I]         M = MIN(M, C[I, N-1])
FOR J = 0 TO N-1
    C[0, J] = B[J]           M = MIN(M, C[0, J])
FOR I = 1 TO N-2
    FOR J = N-2 DOWNTO 1
        C[I, J] = C[I-1, J-1] · C[I, J+1]
        M = MIN(M, C[I, J])
RETURN M
```

```
COMPUTE(a,b)
n <- length(a)
m = +infinito
for i=1 to n-1 do
   C[i,n-1] <- a_i
   m <- MIN(m,C[i,n-1])
for j=0 to n-1 do
   C[0,j] <- b_j
   m <- MIN(m,C[0,j])
for i=1 to n-2 do
   for j=n-2 downto i do
      C[i,j] <- C[i-1,j-1] * C[i,j+1]
      m <- MIN(m,C[i,j])
return m
```

2

$$\sum_{i=1}^{n-2} \sum_{j=i}^{n-2} 1 = \sum_{i=1}^{n-2} n-1-i$$

$$= \sum_{k=1}^{(n-2)} k = \frac{(n-2)(n-1)}{2}$$

$$\left[k = \frac{n(n-1)}{2} \right] \equiv \text{GAUSS}$$

Domanda B (7 punti) Si consideri il problema di selezione di attività compatibili:

1. Definire il problema.
2. Descrivere brevemente l'algoritmo ottimo GREEDY_SEL visto in classe.
3. Fornire un esempio di algoritmo greedy *non* ottimo, motivandone la non ottimalità.

1. Selezione di attività mutualmente compatibili, ciascuna con tempo di inizio e fine.

1. Definizione del problema Il problema di selezione di attività compatibili consiste nel selezionare il massimo numero possibile di attività da un insieme dato, dove ogni attività è caratterizzata da un tempo di inizio e un tempo di fine. Due attività sono considerate compatibili se i loro intervalli temporali non si sovrappongono. L'obiettivo è trovare il sottoinsieme più grande di attività mutuamente compatibili.

2.

```
GREEDY_SEL(A)

n = length(A)
a_opt = {a_1}
for i=2 to n
        if(a_i > a_opt)
                a_opt = a_opt U {a_i}
```

MISSING: PUNTATORE ALL'ULTIMA

GREEDY-SEL(S, f)

```
n = S.length
A = {a_1}
last = 1 // indice dell'ultima attività selezionata
for m = 2 to n
    if s_m ≥ f_last
        A = A ∪ {a_m}
        last = m
return A
```

2. Algoritmo ottimo GREEDY_SEL L'algoritmo GREEDY_SEL risolve questo problema in modo ottimo seguendo questa strategia: ordina le attività per tempo di fine crescente e seleziona iterativamente l'attività compatibile con tempo di fine più piccolo. In dettaglio:

INIZIO

- Ordina le attività secondo il tempo di fine crescente
- Seleziona la prima attività (quella con tempo di fine minimo)
- Per ogni attività successiva, la seleziona solo se il suo tempo di inizio è maggiore o uguale al tempo di fine dell'ultima attività selezionata Questo approccio garantisce di trovare sempre una soluzione ottima al problema.

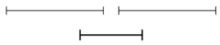
3

8.2.4 Scelte Greedy Alternative

Oltre alla scelta greedy vista precedentemente, ne esistono altre:

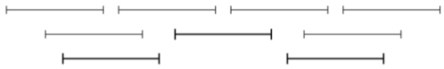
- Scegli l'attività di durata inferiore → non è ottima.

  Controesempio:

- Scegli l'attività col minor numero di sovrapposizioni → non è ottima.

  Controesempio:

- Scegli l'attività che inizia per prima → non è ottima.

  Controesempio:

- Scegli l'attività che inizia per ultima → è ottima.