Domanda 43 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
10	6	2	8	19	31	15

Spiegare il processo di costruzione del codice.

L'algoritmo di Huffman è usato per la compressione dati e crea un codice prefisso ottimale basato sulla frequenza dei simboli. Ecco i passaggi per costruire il codice:

1. Ordinare i simboli per frequenza crescente: c(2), b(6), a(10), e(10), g(15), d(8), f(31)
2. Prendere i due simboli con frequenza minore e combinarli in un nodo, sommando le loro frequenze: c + b = 8
3. Ripetere il processo, includendo il nuovo nodo creato, fino a formare un albero completo: (c+b) + a = 18 ((c+b)+a) + e = 28 (((c+b)+a)+e) + g = 43 ((((c+b)+a)+e)+g) + d = 51 (((((c+b)+a)+e)+g)+d) + f = 82
4. Assegnare '0' ai rami sinistri e '1' ai rami destri dell'albero risultante.
5. Il codice per ogni simbolo sarà il percorso dalla radice alla foglia corrispondente.

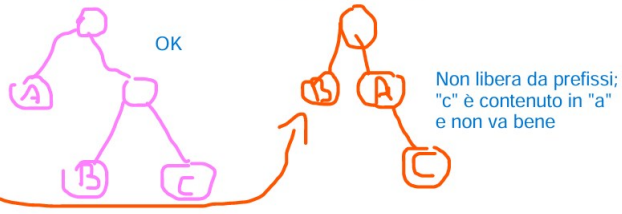
Senza poter disegnare l'albero completo, posso fornire una descrizione generale del risultato:

- I simboli meno frequenti (come 'c' e 'b') avranno codici più lunghi
- I simboli più frequenti (come 'f') avranno codici più corti
- Ogni codice sarà univoco e nessun codice sarà il prefisso di un altro

Per andare bene, (a) dovrebbe essere al livello superiore, perché in pratica Huffman è un albero somma senza nodi delle stesse frequenze allo stesso livello.

Esercizio 2 (9 punti) Si consideri un file definito sull'alfabeto $\Sigma = \{a, b, c\}$, con frequenze $f(a), f(b), f(c)$. Per ognuna delle seguenti codifiche si determini, se esiste, un opportuno assegnamento di valori alle 3 frequenze $f(a), f(b), f(c)$ per cui l'algoritmo di Huffman restituisce tale codifica, oppure si argomenti che tale codifica non è mai ottenibile.

1. $e(a) = 0,\ e(b) = 10,\ e(c) = 11$
2. $e(a) = 1,\ e(b) = 0,\ e(c) = 11$
3. $e(a) = 10,\ e(b) = 01,\ e(c) = 00$



Non libera da prefissi; "c" è contenuto in "a" e non va bene

Domanda 31 Si consideri una tabella hash di dimensione $m = 8$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \ mod \ m$ e $h_2(k) = 1 + k \ mod \ (m - 2)$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 12, 3, 22, 14, 38.

Doppio hash: h(k, i) = (h_1(k) + i * h_2(k)) mod m

Soluzione: Si ottiene

0	-
1	14
2	-
3	3
4	12
5	-
6	22
7	38

Nell'ordine:
1) h(12, 0) = (12 mod 8) + 0*(1 + 12 mod 6) = 4
Inseriamo quindi 12 in posizione 4
2) h(3, 0) = (3 mod 8) + 0*(1 + 3 mod 6) = 3
Inseriamo quindi 3 in posizione 3
3) h(22, 0) = (22 mod 8) + 0*(1 + 12 mod 7) = 6
Inseriamo quindi 22 in posizione 6
4) h(14, 0) = (14 mod 8) + 0*(1 + 14 mod 6) = 6
Collisione --> incremento il tentativo "i" di 1
h(14, 1) = (14 mod 8) + 1*(1 + 14 mod 6) = 9 mod 8 = 1
Inseriamo 14 in posizione 1
5) h(38, 0) = (38 mod 8) + 0*(1 + 38 mod 6) = 6
Collisione --> incremento il tentativo "i" di 1
h(38, 1) = (38 mod 8) + 1*(1 + 38 mod 6) = 9 mod 8 = 1
Altra collisione --> incremento il tentativo "i" di 2
h(38, 2) = (38 mod 8) + 2*(1 + 38 mod 6) = 12 mod 8 = 4
Altra collisione --> incremento il tentativo "i" di 3
h(38, 3) = (38 mod 8) + 3*(1 + 38 mod 6) = 15 mod 8 = 7

Domanda 34 Si consideri una tabella hash di dimensione $m = 8$, gestita mediante chaining (liste di trabocco) con funzione di hash $h(k) = k \ mod \ m$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 14, 10, 22, 18, 19.

Soluzione: Si ottiene

0	
1	
2	$18 \rightarrow 10$
3	19
4	
5	
6	$22 \rightarrow 14$
7	

Quindi:
14 mod 8 = 6
Inserisco 14 in posizione 6
10 mod 8 = 2
Inserisco 10 in posizione 2
22 mod 8 = 6
Essendo chaining, concatenato nella posizione 6 una lista che incorpora come primo elemento il 22
18 mod 8 = 2
Inseriamo 18 in posizione
Essendo chaining, concatenato nella posizione 2 una lista che incorpora come primo elemento il 18
19 mod 8 = 3
Inseriamo 19 in posizione 3

Esercizio 2 (10 punti) Dato un insieme di n numeri reali positivi e distinti $S = \{a_i, a_2, \ldots, a_n\}$, con $0 < a_i < a_j < 1$ per $1 \leq i < j \leq n$, un *(2,1)-boxing* di S è una partizione $P = \{S_1, S_2, \ldots, S_k\}$ di S in k sottoinsiemi (cioè, $\bigcup_{j=1}^{k} S_j = S$ e $S_r \cap S_t = \emptyset, 1 \leq r \neq t \leq k$) che soddisfa inoltre i seguenti vincoli:

$$|S_j| \leq 2 \qquad \text{e} \qquad \sum_{a \in S_j} a \leq 1, \quad 1 \leq j \leq k.$$

In altre parole, ogni sottoinsieme contiene al più due valori la cui somma è al più uno. Dato S, si vuole determinare un (2,1)-boxing che minimizza il numero di sottoinsiemi della partizione.

1. Scrivere il codice di un algoritmo greedy che restituisce un (2,1)-boxing ottimo in tempo lineare. (Suggerimento: si creino i sottoinsiemi in modo opportuno basandosi sulla sequenza ordinata.)

2. Si enunci la proprietà di scelta greedy per l'algoritmo sviluppato al punto precedente e la si dimostri, cioè si dimostri che esiste sempre una soluzione ottima che contiene la scelta greedy.

```
def optimal_2_1_boxing(S):
    S.sort()  # Ordina l'insieme in ordine crescente
    result = []
    i = 0
    n = len(S)
        while i < n:
        if i == n - 1 or S[i] + S[i+1] > 1:
            result.append([S[i]])
            i += 1
        else:
            result.append([S[i], S[i+1]])
            i += 2
        return result
```

Proprietà di scelta greedy: La scelta ottimale locale consiste nel raggruppare i due elementi più piccoli consecutivi se la loro somma non supera 1, altrimenti si prende l'elemento più piccolo da solo.

Dimostrazione: Consideriamo due casi:

a) Se i due elementi più piccoli hanno somma ≤ 1: Raggrupparli insieme è ottimale perché:

Non possiamo aggiungere un terzo elemento (violerebbe il vincolo $|S_j| \leq 2$).

Separandoli, aumenteremmo il numero di sottoinsiemi senza benefici.

b) Se i due elementi più piccoli hanno somma > 1: Prendere il più piccolo da solo è ottimale perché:

Non può essere raggruppato con nessun altro elemento (tutti gli altri sono maggiori).

Raggrupparlo con qualsiasi altro elemento violerebbe il vincolo $\sum a \leq 1$.

Induzione: Base: Con 1 o 2 elementi, l'algoritmo produce chiaramente la soluzione ottimale. Passo induttivo: Supponiamo che l'algoritmo sia ottimale per k elementi. Aggiungendo il (k+1)-esimo elemento:

Se può essere raggruppato con il k-esimo, lo fa ottimalmente.

Se non può, inizia un nuovo sottoinsieme, che è l'unica scelta possibile.

In entrambi i casi, la soluzione rimane ottimale per k+1 elementi.

Quindi, per induzione, l'algoritmo produce sempre una soluzione ottimale che contiene la scelta greedy.

Esercizio 2 (10 punti) Abbiamo n programmi da eseguire sul nostro computer. Ogni programma j, dove $j \in \{1, 2, \ldots, n\}$, ha lunghezza ℓ_j, che rappresenta la quantità di tempo richiesta per la sua esecuzione. Dato un ordine di esecuzione $\sigma = j_1, j_2, \ldots, j_n$ dei programmi (cioè, una permutazione di $\{1, 2, \ldots, n\}$), il tempo di completamento $C_{j_i}(\sigma)$ del j_i-esimo programma è dato quindi dalle somma delle lunghezze dei programmi $j_1, j_2, \ldots, j_i$. L'obiettivo è trovare un ordine di esecuzione σ che minimizza la somma dei tempi di completamento di tutti i programmi, cioè $\sum_{j=1}^{n} C_j(\sigma)$.

(a) Dare un semplice algoritmo greedy per questo problema, e valutarne la complessità.

(b) Dimostrare la proprietà di scelta greedy dell'algoritmo del punto (a), cioè che esiste un ordine di esecuzione ottimo $\sigma^\star$ che contiene la scelta greedy.

a)

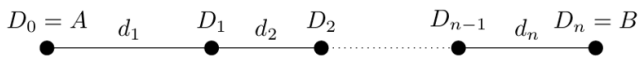
1. Creare una lista di coppie (j, ℓ_j) per ogni programma j

2. Ordinare la lista in base a ℓ_j in ordine crescente

3. L'ordine di esecuzione σ è dato dall'ordine dei j nella lista ordinata

```
Algoritmo: OrdinamentoOptProgrammi

Input: Un array L di n elementi, dove L[j] rappresenta la lunghezza ℓ_j
del programma j

Output: Un array σ che rappresenta l'ordine ottimale di esecuzione dei
programmi

1. // Creazione dell'array di coppie (indice, lunghezza)

   Sia P un array di n coppie (j, ℓ_j)

   Per j = 1 fino a n:

       P[j] = (j, L[j])

2. // Ordinamento dell'array P basato sulle lunghezze ℓ_j

   OrdinaPerSecondoElemento(P)  // Utilizzo di un algoritmo di
ordinamento efficiente come QuickSort o MergeSort

3. // Estrazione dell'ordine ottimale

   Sia σ un nuovo array di dimensione n

   Per i = 1 fino a n:

       σ[i] = P[i].primo  // Estrae l'indice j dalla coppia ordinata

4. Restituisci σ

Funzione OrdinaPerSecondoElemento(P):

    // Implementazione di un algoritmo di ordinamento (es. QuickSort)

    // che ordina le coppie in P basandosi sul loro secondo elemento (ℓ_j)
```

Esercizio 33 Si supponga di voler viaggiare dalla città A alla città B con un'auto che ha un'autonomia pari a d km. Lungo il percorso si trovano $n-1$ distributori $D_1, \ldots, D_{n-1}$, a distanze di $d_1, \ldots, d_n$ km ($d_i \leq d$) come indicato in figura



L'auto ha inizialmente il serbatoio pieno e l'obiettivo è quello di percorrere il viaggio da A a B, minimizzando il numero di soste ai distributori per il rifornimento.

i. Introdurre la nozione di soluzione per il problema e di costo della una soluzione. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `stop(d,n)` che dato in input l'array delle distanze `d[1..n]` restituisce una soluzione ottima.

iii. Valutare la complessità dell'algoritmo.

i. Nozione di soluzione e proprietà di scelta greedy:

Soluzione: Una sequenza di distributori presso cui l'auto si ferma per rifornirsi.

Costo: Il numero di soste effettuate durante il viaggio.

Proprietà della sottostruttura ottima:

Se abbiamo una soluzione ottima per il viaggio da A a B che include una sosta al distributore Di, allora la parte della soluzione da A a Di e da Di a B devono essere entrambe ottime per i rispettivi sotto-percorsi.

Scelta greedy:

La scelta greedy consiste nel guidare il più lontano possibile prima di fermarsi per il rifornimento. Ovvero, ci si ferma solo quando non è possibile raggiungere il distributore successivo con il carburante rimanente.

Dimostrazione della proprietà della scelta greedy:

Supponiamo per assurdo che esista una soluzione ottima S che non segue la scelta greedy. Ciò significa che in S esiste una sosta a un distributore Di quando sarebbe stato possibile raggiungere un distributore successivo Dj. Possiamo costruire una nuova soluzione S' identica a S ma saltando la sosta a Di. S' avrà un numero minore di soste rispetto a S, contraddicendo l'ottimalità di S. Quindi, la scelta greedy è sempre parte di una soluzione ottima.

```
Funzione stop(d, n, distanze[1..n])
    soste = []
    carburante_rimanente = d
    posizione_attuale = 0
    Per i da 1 a n:
        Se distanze[i] > carburante_rimanente:
            Aggiungi i-1 a soste
            carburante_rimanente = d - (distanze[i] - distanze[i-1])
        Altrimenti:
```

```
            carburante_rimanente -= (distanze[i] - distanze[i-1])


    Restituisci soste
```

L'algoritmo ha una complessità temporale di O(n), dove n è il numero di distributori. Questo perché esegue un singolo passaggio attraverso l'array delle distanze, effettuando operazioni costanti per ogni distributore.

La complessità spaziale è O(n) nel caso peggiore, dove potrebbe essere necessario fermarsi a ogni distributore. Tuttavia, in pratica, lo spazio utilizzato sarà generalmente molto inferiore, poiché l'obiettivo è minimizzare il numero di soste.

Esercizio 34 L'ufficio postale offre un servizio di ritiro pacchi in sede su prenotazione. Il destinatario, avvisato della presenza del pacco, deve comunicare l'orario preciso al quale si recherà allo sportello. Sapendo che gli impiegati dedicano a questa mansione turni di un'ora, con inizio in un momento qualsiasi, si chiede di scrivere un algoritmo che individui l'insieme minimo di turni di un'ora sufficienti a soddisfare tutte le richieste. Più in dettaglio, data una sequenza $\vec{r} = r_1, \ldots, r_n$ di richieste, dove r_i è l'orario della i-ma prenotazione, si vuole determinare una sequenza di turni $\vec{t} = t_1, \ldots, t_k$, con t_j orario di inizio del j-mo turno, che abbia dimensione minima e tale che i turni coprano tutte le richieste.

i. Formalizzare la nozione di soluzione per il problema e il relativo costo. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `time(R,n)` che dato in input l'array delle richieste `r[1..n]` restituisce una soluzione ottima.

iii. Valutare la complessità dell'algoritmo.

i. Formalizzazione e proprietà:

Soluzione: Una sequenza di orari di inizio dei turni t = t1, ..., tk che copre tutte le richieste. Costo: Il numero k di turni nella soluzione.

Proprietà della sottostruttura ottima: Se abbiamo una soluzione ottima per le richieste r1, ..., rn che include un turno che inizia all'orario ti, allora le parti della soluzione che coprono le richieste prima di ti e dopo ti+1 ora devono essere ottime per i rispettivi sottoinsiemi di richieste.

Scelta greedy: La scelta greedy consiste nel selezionare come inizio del prossimo turno l'orario della prima richiesta non ancora coperta.

Dimostrazione della proprietà della scelta greedy: Supponiamo per assurdo che esista una soluzione ottima S che non segue la scelta greedy. Ciò significa che in S esiste un turno che inizia dopo la prima richiesta non coperta ri. Possiamo costruire una nuova soluzione S' anticipando l'inizio di questo turno all'orario di ri. S' copre tutte le richieste di S e potenzialmente anche altre, senza aumentare il numero di turni. Quindi, S' è almeno altrettanto buona di S, dimostrando che la scelta greedy è sempre parte di una soluzione ottima.

ii. Algoritmo greedy time(R,n):

```
Funzione time(R, n)

    turni = []

    i = 1

    Mentre i ≤ n:
```

```
        Aggiungi R[i] a turni
        fine_turno = R[i] + 1  // Il turno dura un'ora
        Mentre i ≤ n e R[i] < fine_turno:
            i = i + 1
    Restituisci turni
```

Questo algoritmo seleziona sempre l'orario della prima richiesta non coperta come inizio di un nuovo turno, e poi avanza fino alla prima richiesta non coperta dal turno appena aggiunto.

iii. Complessità dell'algoritmo:

L'algoritmo ha una complessità temporale di O(n), dove n è il numero di richieste. Questo perché ogni richiesta viene esaminata una sola volta nel ciclo esterno o in quello interno.

La complessità spaziale è O(k), dove k è il numero di turni nella soluzione ottima. Nel caso peggiore, potrebbe essere O(n) se ogni richiesta richiedesse un turno separato.

Questo algoritmo greedy è molto efficiente, fornendo una soluzione ottima al problema con una singola scansione dell'array delle richieste. La sua linearità lo rende ideale per gestire anche grandi volumi di richieste in modo rapido ed efficace.

```
Funzione time(R, n)
    turni = []
    i = 1
    Mentre i ≤ n:
        Aggiungi R[i] a turni
        fine_turno = R[i] + 1  // Il turno dura un'ora
        Mentre i ≤ n e R[i] < fine_turno:
            i = i + 1
    Restituisci turni
```