Immagine che contiene testo, Carattere, ricevuta, schermata

Descrizione generata automaticamente

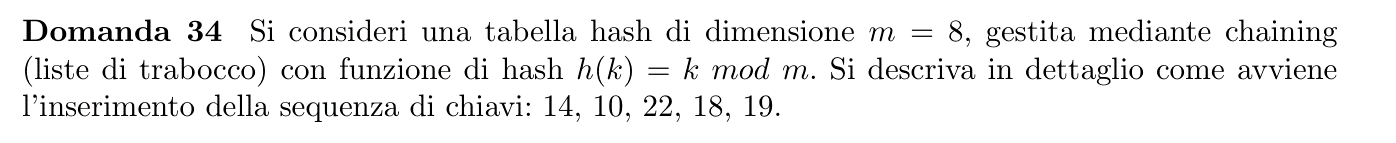
L'algoritmo di Huffman è usato per la compressione dati e crea un codice prefisso ottimale basato sulla frequenza dei simboli. Ecco i passaggi per costruire il codice:

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
* Immagine che contiene testo, schermata, Carattere, cartone animato

  Descrizione generata automaticamenteOgni codice sarà univoco e nessun codice sarà il prefisso di un altro

Immagine che contiene testo, schermata, Carattere, numero

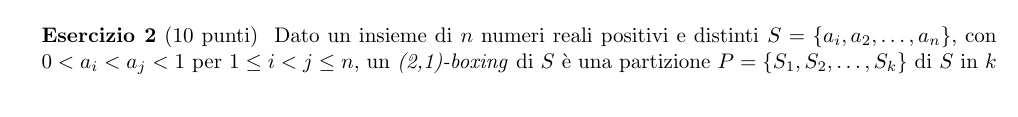
Descrizione generata automaticamente

Immagine che contiene testo, diagramma, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente



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 |Sj| ≤ 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 ∑a ≤ 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.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

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)

Immagine che contiene testo, schermata, Carattere, bianco e nero

Descrizione generata automaticamente

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.

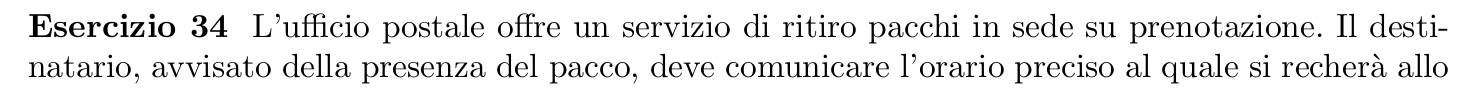
La complessità spaziale è O(n) nel caso peggiore, dove potrebbe essere necessario fermarsi a ogni distributore. Tuttavia, in pratica, lo spazio utilizzato sarà generalmente molto inferiore, poiché l'obiettivo è minimizzare il numero di soste.

Immagine che contiene testo, Carattere, schermata, bianco e nero

Descrizione generata automaticamente

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