1) Il problema della selezione delle attività prevede l’organizzazione di una serie di impegni con tempi di inizio e fine dedicati tali che siano compatibili tra di loro (riusciamo a combinare attività in maniera tale che non si intersechino tra di loro; quando finisce una, ne inizia almeno subito un’altra).

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamenteMatematicamente:

2)

Immagine che contiene testo, Carattere, schermata, ricevuta

Descrizione generata automaticamente|---------|



|------|



L’algoritmo si basa su una scelta greedy: date le attività, consideriamo la prima come ottima (perché inizia almeno da un punto >=0) e inseriamo tutte le altre attività con tempo di inizio >= all’attività subito precedente (quella greedy), costruendo così un insieme di attività ottimo.

3)

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteEsempi non ottimi misto ad altro esempio ottimo (= attività scelta per ultima)



Greedy-Sel-Reverse(S, f)

1 n = S.length

2 A = {a[n]}

3 first = n // indice della prima attività selezionata (partendo dalla fine)

4 for m = n-1 down to 1

5 if f[m] <= s[first]

6 A = A ∪ {a[m]}

7 first = m

8 return A

Esercizio (11 punti) Si consideri il problema della selezione delle attività compatibili. Si ha un insieme S di n attività, dove ogni attività ai ha un tempo di inizio s\_i e un tempo di fine f\_i. Due attività ai e aj sono compatibili se i loro intervalli di tempo non si sovrappongono, ovvero se f\_i ≤ s\_j o f\_j ≤ s\_i. L'obiettivo è selezionare il sottoinsieme di attività compatibili di cardinalità massima.

(a) Qual è la complessità dell'algoritmo esaustivo che esamina tutti i possibili sottoinsiemi di attività?

(b) Assumendo di conoscere un algoritmo che determina se due attività sono compatibili in tempo O(1), come si può modificare l'algoritmo del punto precedente per renderlo più efficiente?

(c) Progettare un algoritmo greedy efficiente per risolvere il problema. Sono richiesti:

* La strategia greedy utilizzata
* Lo pseudocodice dell'algoritmo
* La dimostrazione della correttezza (proprietà di scelta greedy e sottostruttura ottima)
* L'analisi della complessità
* Progetta un algoritmo su LCS e scrivi la ricorrenza

LCS 🡪 Due stringhe con lunghezza

Dato l’algoritmo, dopo scrivi la relazione di ricorrenza

* Algoritmo con due array come max heap in un unico array che era max heap (con pseudocodice)

Max heap:

* Parent >= Figli

🡪 Parent

funzione MERGE\_MAX\_HEAPS(heap1, heap2)

// Assumiamo che heap1 e heap2 siano array che rappresentano max heap

n1 = lunghezza(heap1)

n2 = lunghezza(heap2)

// Creiamo un nuovo array per contenere entrambi gli heap

risultato = nuovo array di dimensione (n1 + n2)

// Copiamo gli elementi di entrambi gli heap nel nuovo array

per i da 0 a n1-1

risultato[i] = heap1[i]

per i da 0 a n2-1

risultato[n1 + i] = heap2[i]

// Ora abbiamo un array che contiene tutti gli elementi, ma non è un max heap

// Dobbiamo heapify l'intero array dal basso verso l'alto

per i da ((n1 + n2) / 2) - 1 fino a 0 con passo -1

MAX\_HEAPIFY(risultato, n1 + n2, i)

return risultato

funzione MAX\_HEAPIFY(A, n, i)

largest = i

left = 2 \* i + 1

right = 2 \* i + 2

se left < n e A[left] > A[largest] allora

largest = left

se right < n e A[right] > A[largest] allora

largest = right

se largest != i allora

scambia A[i] con A[largest]

MAX\_HEAPIFY(A, n, largest)

doMerge(A,B,i,j)

l = 2\*i

r = 2\*i+1

if(A[l] >= B[l]

C[i] = A[l] // salvo max come posizione attuale

C[l] = B[l] // posiziono l’altro nel livello che sto toccando

else

C[i] = A[r]

C[r] = B[r]

return C

* Greedy delle attività ma con l’attività scelta per ultima