ASD Homework 4

Francesco Iannaccone e Matteo Conti

6 dicembre 2021

1 Homework Esercitativi

In questo problema dobbiamo minimizzare il numero di aule per programmare nel modo più efficiente possibile un certo numero di lezioni n. Ci sono pochi possibili metodi di ordinamento in cui possiamo organizzare le lezioni. Il primo è ordinarle in base all'ora di inizio, il secondo è ordinarle in base all'ora di fine, il terzo è ordinarle in base alla durata minima dell'attività e il quarto è in base al numero minimo di task in conflitto. Si dimostra, attraverso dei contro-esempi, che la soluzione ottimale è scegliere le lezioni in base all'ora di inizio. Quindi, in qualsiasi momento, scegliamo la lezione che inizia prima e che non è ancora stata assegnata e la assegniamo alla prima aula disponibile. Una volta assegnate tutte le lezioni, il numero totale di aule sarà il numero minimo di aule richieste.

Algorithm 1: assignLessons-Greedy(lessons, classrooms)

```
1  sort(lessons);
   // ordina in modo crescente in base all'orario di inizio
2  numero_classi ← 0 ;
3  foreach lesson ∈ lessons do
4      if ∃ una classroom i disponibile per lesson then
5          classrooms[i] ← lesson;
6      else
7          numero_classi ← numero_classi + 1 ;
8          classrooms[numero_classi] ← lesson ;
9      end
10 end
11 return numero_classi;
```

dove lessons e classrooms, in ingresso all'algoritmo assignLessons-Greedy, sono rispettivamente il vettore delle lezioni, formato da n elementi, che tiene traccia per ogni lezione dell'orario di inizio e di fine, e il vettore delle classi, che per ogni classe mantiene l'informazione di quale lezione si sta tenendo in un determinato intervallo di tempo. Chiaramente, il vettore delle classi all'inizio dell'algoritmo è vuoto e verrà riempito mano a mano che le lezioni saranno elaborate.

Infine, per trovare una classroom disponibile per una certa lezione, bisogna verificare per ogni aula se l'ora di fine della lezione in quell'aula è inferiore all'ora di inizio della nuova lezione da schedulare. Se sì, allora la classroom analizzata è compatibile.

2 Homework di verifica

Una prima strategia di soluzione greedy che potrebbe venire in mente è quella di vendere i biglietti prima ai gruppi più numerosi. La complessità di un algoritmo del genere sarebbe pari a quella dell'ordinamento più quella necessaria per scegliete, tra tutti i gruppi, quelli più numerosi. La complessità complessiva è quindi pari a $O(n \lg(n) + n) = O(n \lg(n))$ supponendo di aver usato un ordinamento come il quicksort.

Algorithm 2: maxGuadagno-Greedy(groups, posti_rimanenti)

```
1 sort(groups);
  // ordina in modo decrescente dal gruppo più grande al più
     piccolo
2 biglietti_venduti ← 0 ;
3 foreach group ∈ groups do
4 |   if group.size ≤ posti_rimanenti then
5 |   |   posti_rimanenti ← posti_rimanenti − group.size;
6 |   |   biglietti_venduti ← group.size + biglietti_venduti;
7 |   end
8 end
9 return biglietti_venduti;
```

L'algoritmo però non fornisce sempre una soluzione ottimale. Supponiamo di avere un gruppo di $N-1$ tifosi, dove N è pari alla capienza complessiva dello stadio, e due gruppi di $\frac{N}{2}$ persone. Seguendo la strategia golosa e vendendo i biglietti prima al gruppo più numeroso, si venderebbero, nel complesso, meno biglietti rispetto a quanti se si scegliesse di venderli ai due gruppi più piccoli.

L'approccio corretto prevede di risolvere il problema tramite la programmazione dinamica, ottenendo una soluzione che, seppur meno efficiente, garantisce l'ottimo per ogni combinazione di ingressi.

L'algoritmo proposto è il seguente:

Algorithm 3: maxGuadagno(groups, pos, posti_rimanenti, dp)

```
1  if dp[pos][posti_rimanenti] ≠ -1 then
2  |  return dp[pos][posti_rimanenti];
3  end
4  if pos ≥ groups.size then
5  |  return 0;
6  else if groups[pos].size ≤ posti_rimanenti then
7  |  dp[pos][posti_rimanenti] = max(maxGuadagno(groups, pos+1,
   |   posti_rimanenti - groups[pos].size, dp) + groups[pos].size,
   |   maxGuadagno(groups, pos+1, posti_rimanenti, dp));
8  else
9  |  dp[pos][posti_rimanenti] = maxGuadagno(groups, pos+1,
   |   posti_rimanenti, dp);
10 end
11 return dp[pos][posti_rimanenti];
```

L'algoritmo opera suddividendo il problema in uno più piccolo utilizzando la tecnica dei suffissi. E' però necessario memorizzare, oltre alla posizione da cui inizia il suffisso, anche il numero di posti rimanenti.

Il numero totale di sottoproblemi è pari a $O(NM)$, dove N corrisponde al numero di posti e M al numero di gruppi di tifosi. Il tempo per sotto problema, al netto delle ricorsioni, è costante in quanto ogni sottoproblema richiede di calcolare il massimo tra due valori. Non essendo necessarie operazioni extra, il tempo di esecuzione complessivo è pari a $O(NM)$.