Lezione 21

Sommario della Lezione

Esercizi

Partizionamento di attività

Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

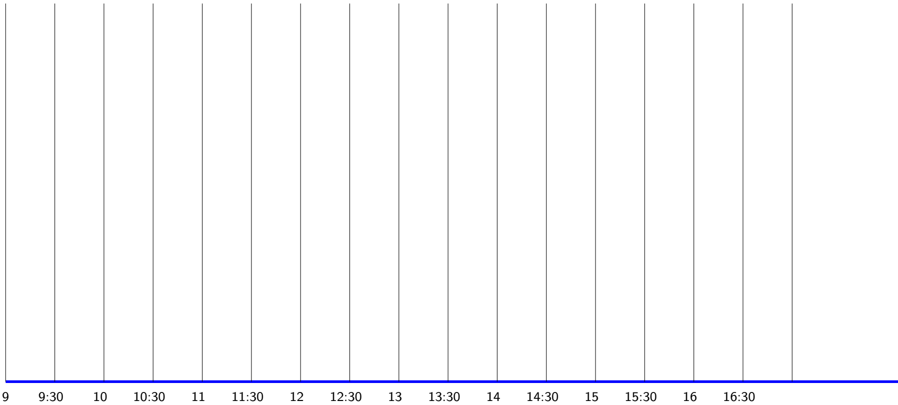
Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

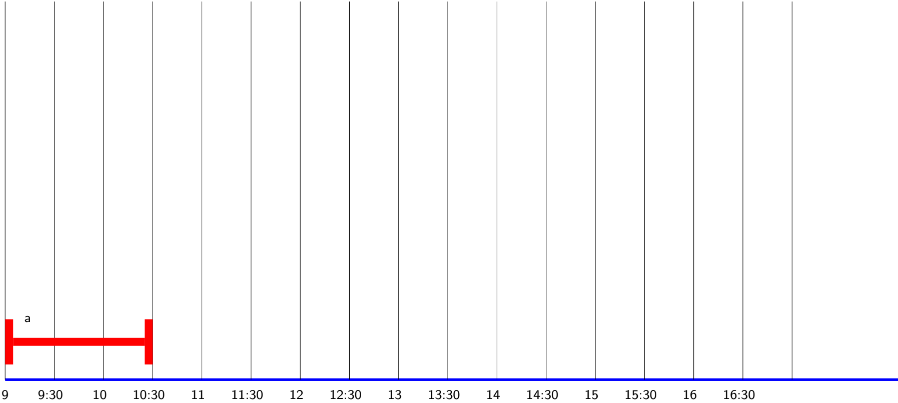


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

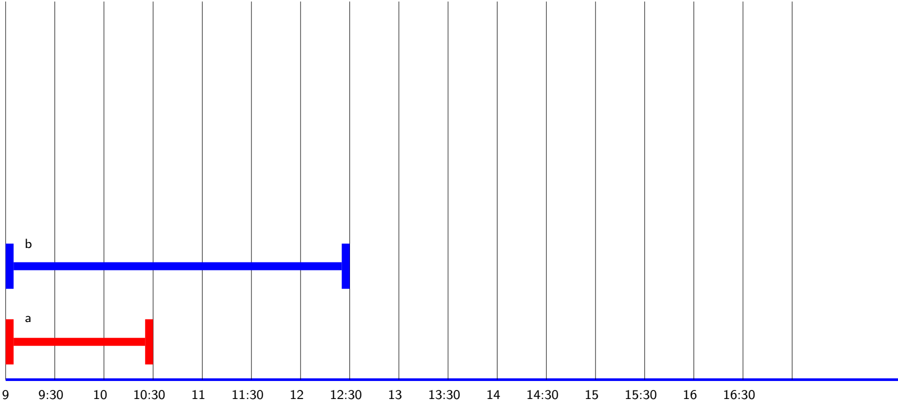


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

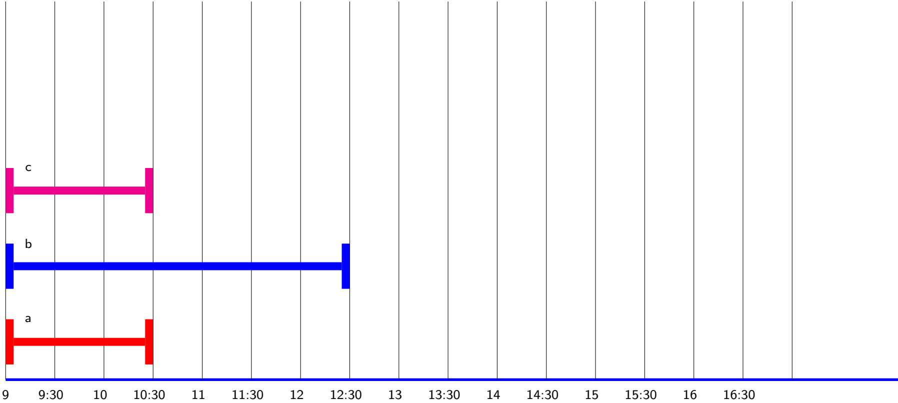


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

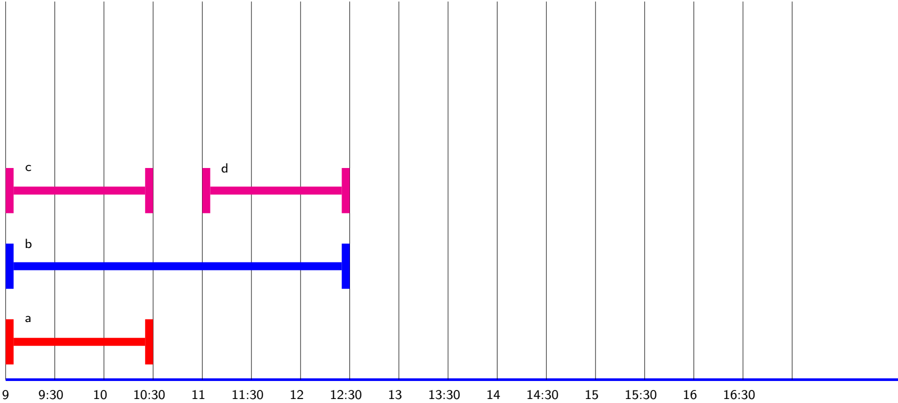


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

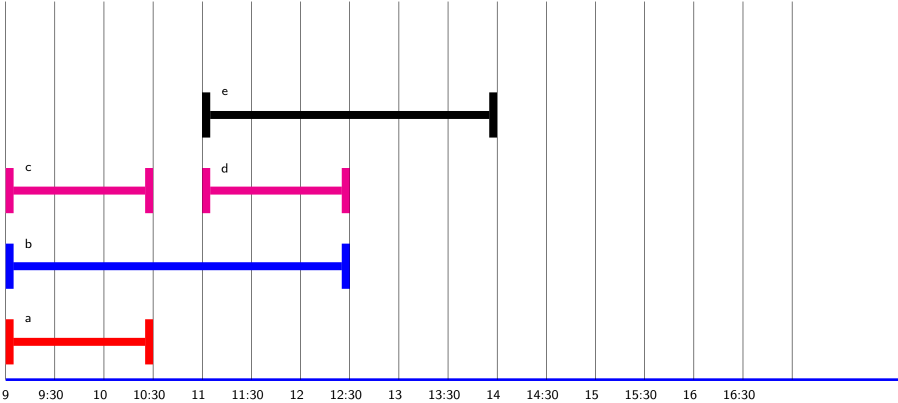


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

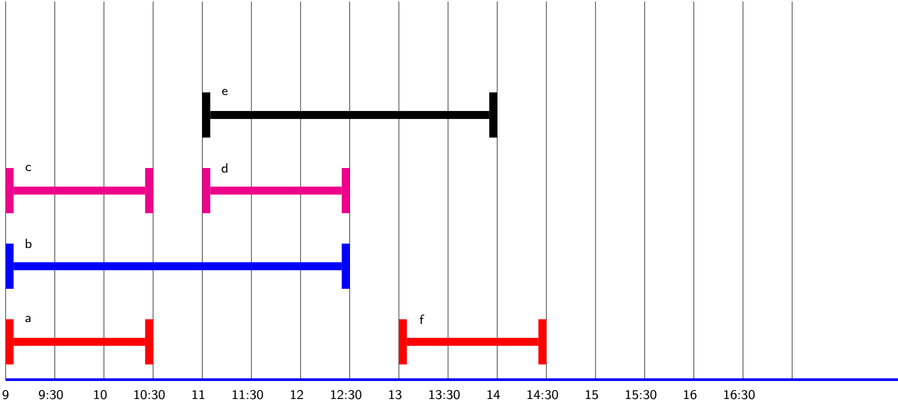


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

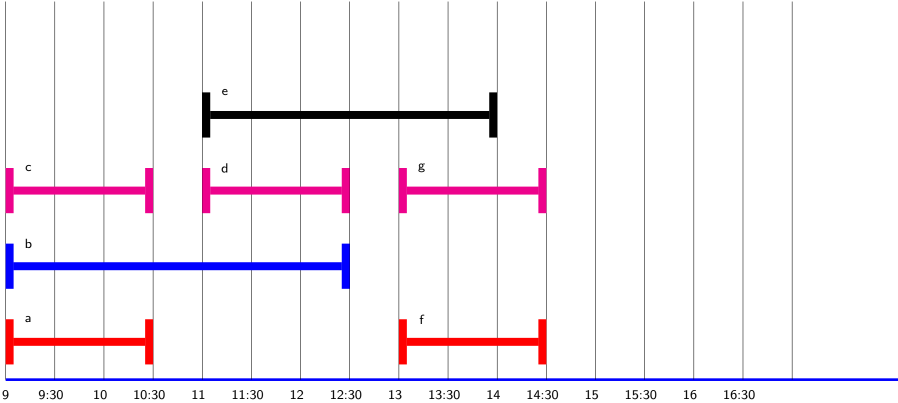


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

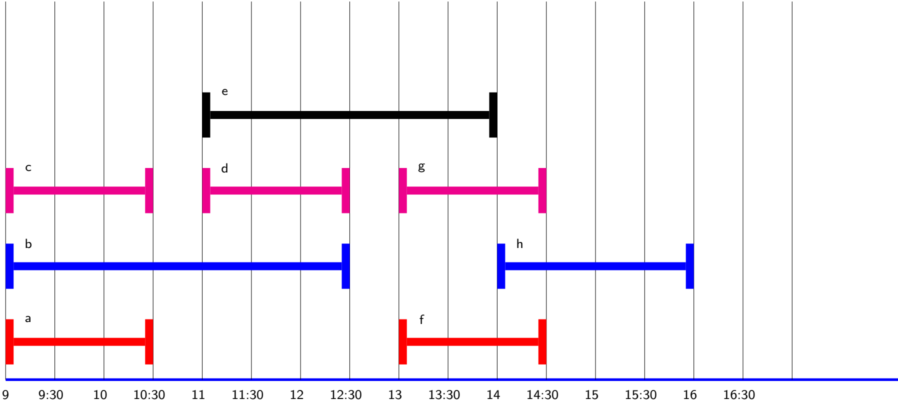


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

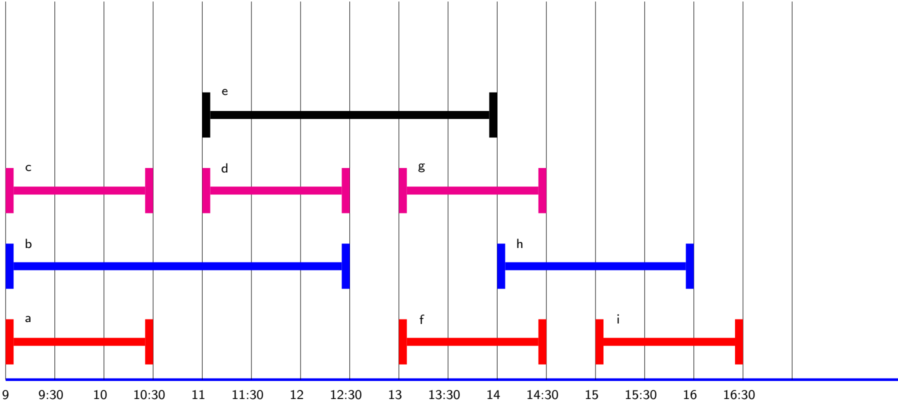


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:

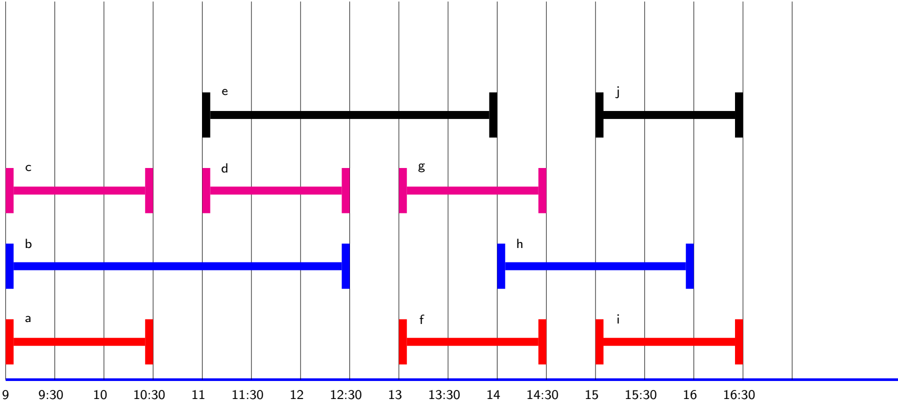


Partizionamento di attività

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

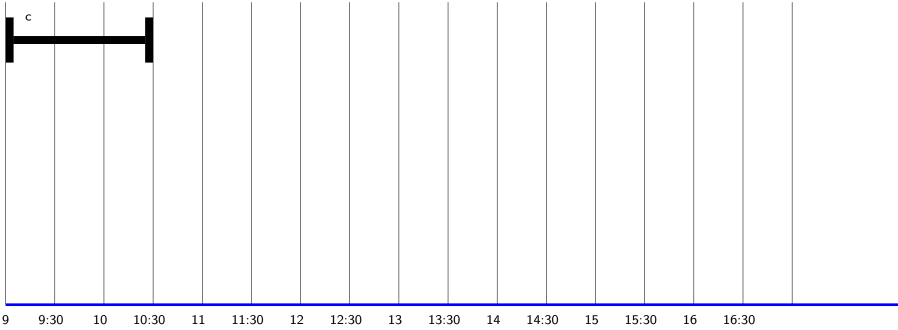
Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi

Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

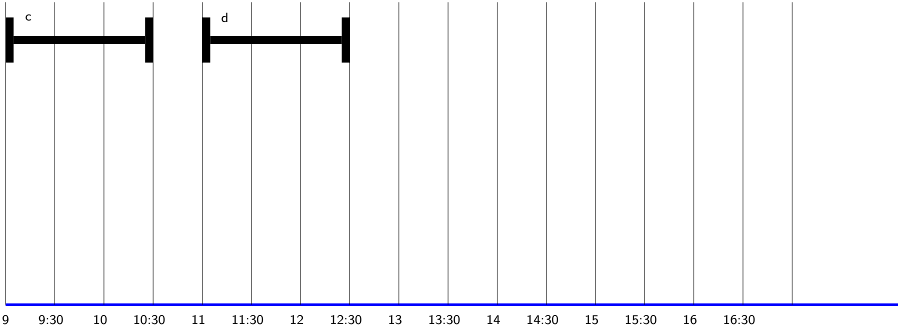
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

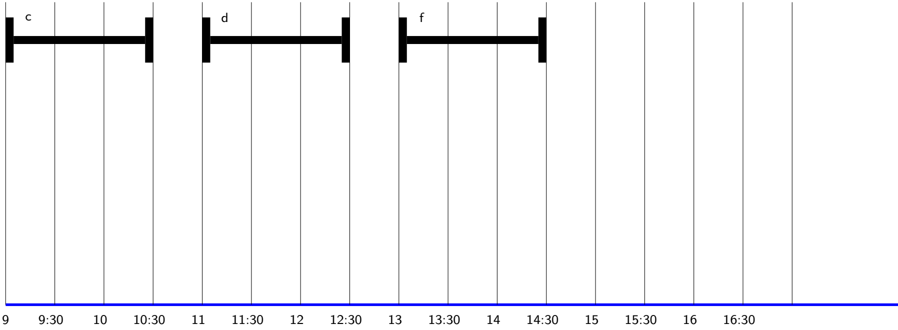
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

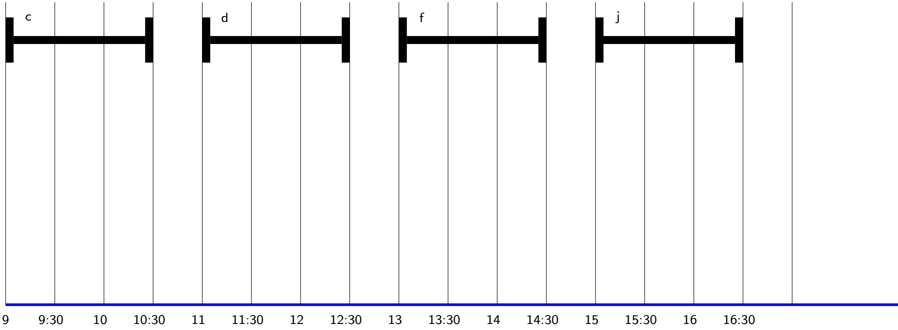
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

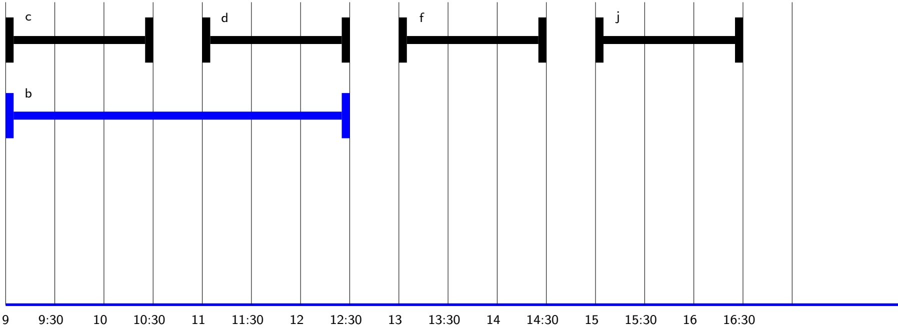
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

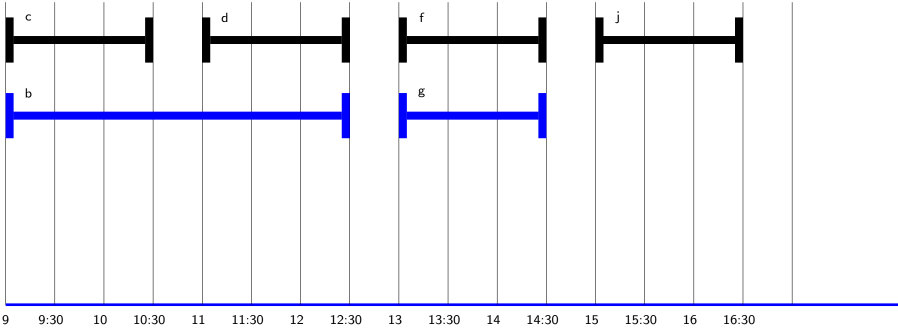
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

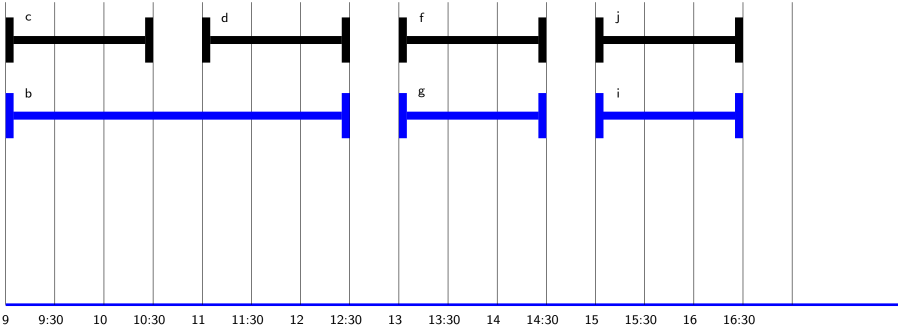
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

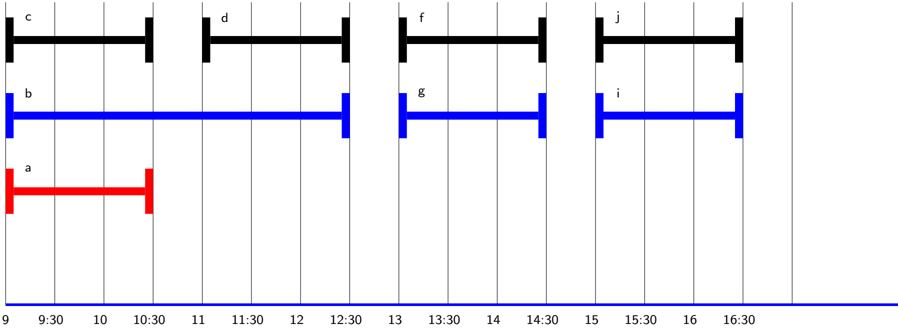
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

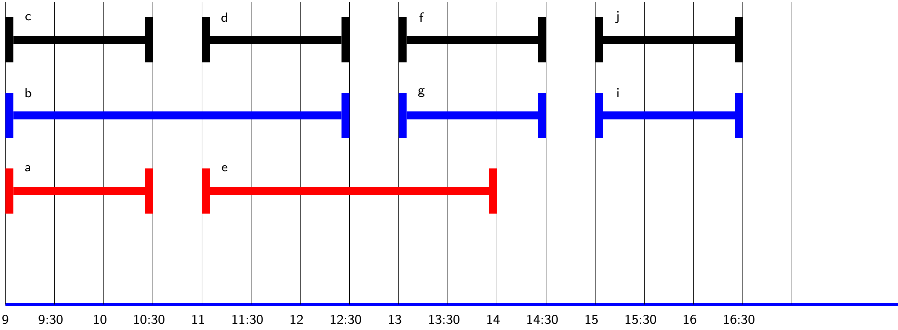
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

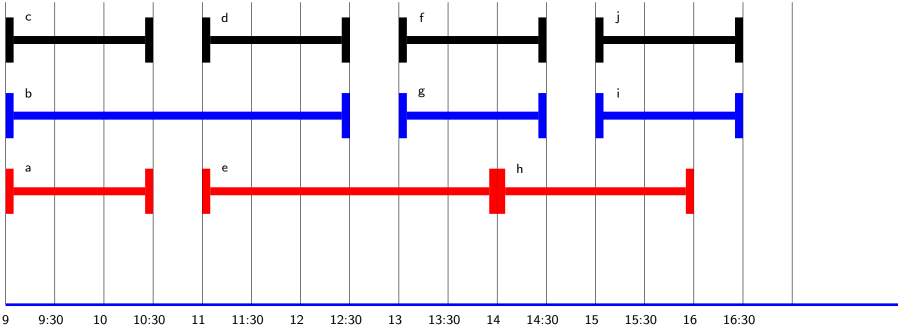
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

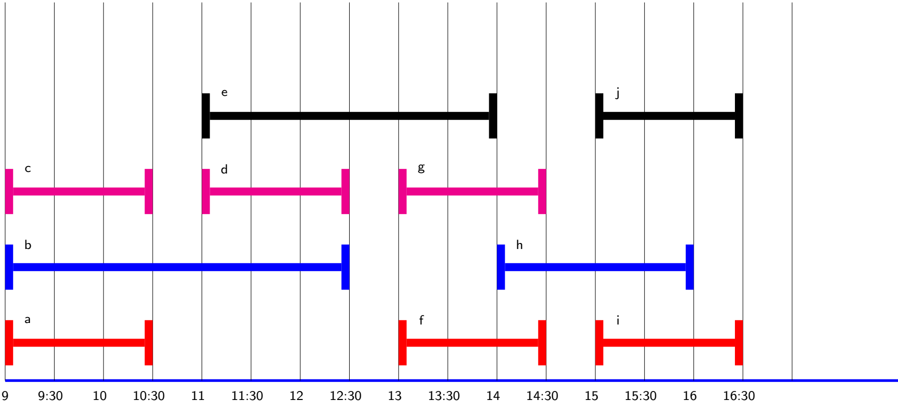
Esempio: quest'altra assegnazione utilizza solo 3 aule per gli stessi 10 corsi



Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

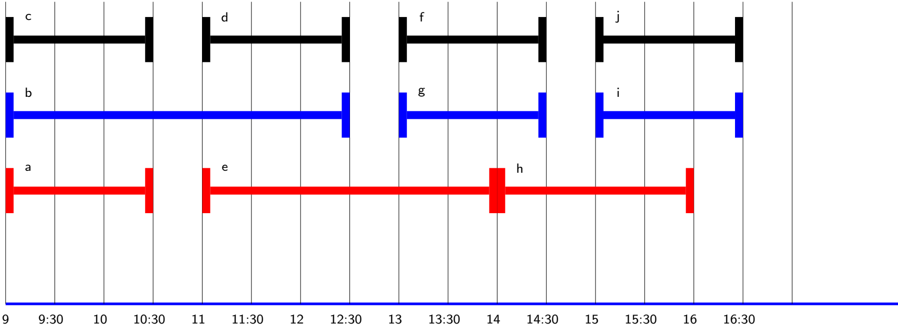
Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

Esempio:



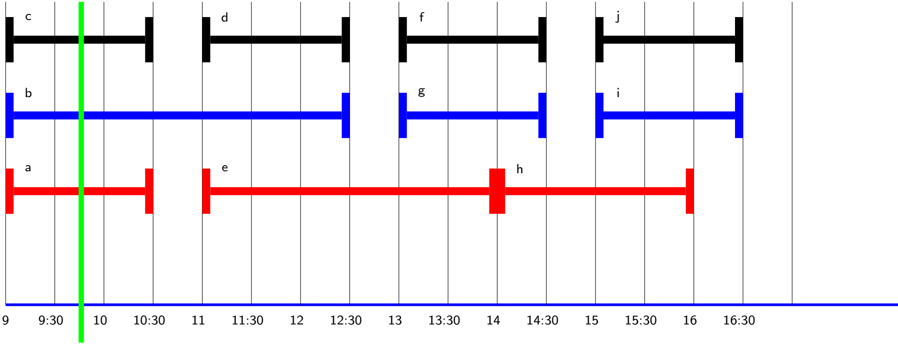
È possibile fare meglio (cioè usare *meno* di 3 aule)?

È possibile fare meglio (cioè usare *meno* di 3 aule)?



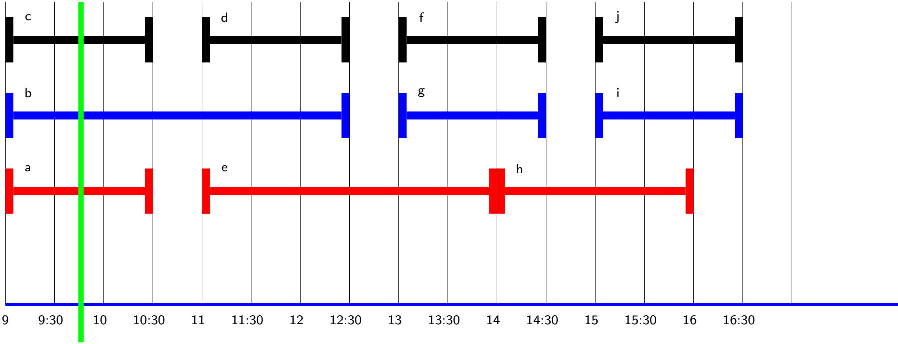
È possibile fare meglio (cioè usare *meno* di 3 aule)?

No, Infatti nell'intervallo dalle 9:30 alle 10:00 vi sono 3 corsi in contemporanea, che necessitano evidentemente di 3 aule distinte



L'ultima osservazione può essere generalizzata nella seguente:

Osservazione chiave: se in un dato istante ci sono *d* corsi in contemporanea, allora *ogni* partizionamento di attività richiede *almeno d* aule



Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano,

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano.

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

Partizionamento_greedy $(s_1 \ldots s_n, f_1, \ldots, f_n)$
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

Partizionamento_greedy $(s_1 \ldots s_n, f_1, \ldots, f_n)$
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1, ..., d},) {
```

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

Partizionamento_greedy ($s_1 \ldots s_n, f_1, \ldots, f_n$)
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)
3. FOR(j=1, j<n+1, j=j+1) {
4. IF (corso j non si sovrappone ai corsi di qualche aula $k \in \{1, \ldots, d\}$,) {
   assegna il corso j all'aula k

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1, ..., d},) {
              assegna il corso j all'aula k
      } ELSE {
5.      apri l'aula d + 1
```

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
             aula k ∈ {1, ..., d},) {
             assegna il corso j all'aula k
      } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
```

Algoritmo Greedy per partizionare i corsi tra il *minor* numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se ciò non è possibile, utilizza una nuova aula.

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1, ..., d},) {
              assegna il corso j all'aula k
      } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
7.      d = d + 1
   }  }
```

Analisi: sia D il numero di aule usate dall'algoritmo

Partizionamento_greedy $(s_1 \ldots s_n, f_1, \ldots, f_n)$
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)
3. FOR(j=1, j<n+1, j=j+1) {
4.   IF (corso j non si sovrappone ai corsi di qualche
          aula $k \in \{1, \ldots, d\}$,) {
          assegna il corso j all'aula k
    } ELSE {
5.   apri l'aula $d+1$
6.   assegna il corso j all'aula $d+1$
7. $d = d + 1$
  } }

Analisi: sia D il numero di aule usate dall'algoritmo

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
                aula k ∈ {1, ..., d},) {
                assegna il corso j all'aula k
       } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
7.      d = d + 1
    }  }
```

L'algoritmo **non** assegna mai due corsi che si svolgono in uno stesso momento alla stessa aula.

Analisi: sia D il numero di aule usate dall'algoritmo

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
                aula k ∈ {1, ..., d},) {
                assegna il corso j all'aula k
     } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
7.      d = d + 1
    }  }
```

L'algoritmo **non** assegna mai due corsi che si svolgono in uno stesso momento alla stessa aula. L'aula D è stata usata in quanto esisteva un corso j che **non** poteva essere inserito in **nessuna** delle aule $1, \ldots, D-1$

Analisi: sia D il numero di aule usate dall'algoritmo

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1, ..., d},) {
              assegna il corso j all'aula k
       } ELSE {
5.      apri l'aula d+1
6.      assegna il corso j all'aula d+1
7.      d = d+1
    }  }
```

L'algoritmo **non** assegna mai due corsi che si svolgono in uno stesso momento alla stessa aula. L'aula D è stata usata in quanto esisteva un corso j che **non** poteva essere inserito in **nessuna** delle aule $1, \ldots, D-1 \Longrightarrow$ poichè i corsi sono ordinati per tempo di inizio, questa impossibilità è causata da $D-1$ corsi che iniziano a tempi $\leq s_j$

Analisi: sia D il numero di aule usate dall'algoritmo

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
                aula k ∈ {1,...,d},) {
                assegna il corso j all'aula k
      } ELSE {
5.      apri l'aula d+1
6.      assegna il corso j all'aula d+1
7.      d = d+1
    }  }
```

L'algoritmo **non** assegna mai due corsi che si svolgono in uno stesso momento alla stessa aula. L'aula D è stata usata in quanto esisteva un corso j che **non** poteva essere inserito in **nessuna** delle aule $1, \dots, D-1$ $\Longrightarrow$ poichè i corsi sono ordinati per tempo di inizio, questa impossibilità è causata da $D-1$ corsi che iniziano a tempi $\leq s_j$ $\Longrightarrow$ al tempo $s_j + \Delta$ ci sono D corsi che si sovrappongono

Analisi: sia D il numero di aule usate dall'algoritmo

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1,...,d},) {
              assegna il corso j all'aula k
     } ELSE {
5.      apri l'aula d+1
6.      assegna il corso j all'aula d+1
7.      d = d+1
    }  }
```

L'algoritmo **non** assegna mai due corsi che si svolgono in uno stesso momento alla stessa aula. L'aula D è stata usata in quanto esisteva un corso j che **non** poteva essere inserito in **nessuna** delle aule $1, \ldots, D-1$ $\Longrightarrow$ poichè i corsi sono ordinati per tempo di inizio, questa impossibilità è causata da $D-1$ corsi che iniziano a tempi $\leq s_j$ $\Longrightarrow$ al tempo $s_j + \Delta$ ci sono D corsi che si sovrappongono $\Longrightarrow$ dalla Osservazione Chiave, **ogni** assegnazione di corsi ad aule richiede D aule,

Analisi: sia D il numero di aule usate dall'algoritmo

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
               aula k ∈ {1, ..., d},) {
               assegna il corso j all'aula k
     } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
7.      d = d + 1
    }  }
```

L'algoritmo **non** assegna mai due corsi che si svolgono in uno stesso momento alla stessa aula. L'aula D è stata usata in quanto esisteva un corso j che **non** poteva essere inserito in **nessuna** delle aule $1, \dots, D-1$ $\Longrightarrow$ poichè i corsi sono ordinati per tempo di inizio, questa impossibilità è causata da $D-1$ corsi che iniziano a tempi $\leq s_j$ $\Longrightarrow$ al tempo $s_j + \Delta$ ci sono D corsi che si sovrappongono $\Longrightarrow$ dalla Osservazione Chiave, **ogni** assegnazione di corsi ad aule richiede D aule, quindi l'algoritmo utilizza il *minor numero possibile* di aule.

Complessità

Complessità

Partizionamento_greedy ($s_1 \dots s_n, f_1, \dots, f_n$)
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \dots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)
3. FOR(j=1, j<n+1, j=j+1) {
4. IF (corso j non si sovrappone ai corsi di qualche
aula $k \in \{1, \dots, d\}$) {
assegna il corso j all'aula k
} ELSE {
5. apri l'aula $d+1$
6. assegna il corso j all'aula $d+1$
7. $d = d+1$
} }

Complessità

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1, ..., d}) {
              assegna il corso j all'aula k
      } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
7.      d = d + 1
    }  }
```

1. richiede $O(n \log n)$.

Complessità

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d è il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1, ..., d}) {
              assegna il corso j all'aula k
      } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
7.      d = d + 1
    }  }
```

1. richiede $O(n \log n)$. Il `FOR 3.` viene eseguito n volte.

Complessità

Partizionamento_greedy $(s_1 \ldots s_n, f_1, \ldots, f_n)$
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)
3. FOR(j=1, j<n+1, j=j+1) {
4. IF (corso j non si sovrappone ai corsi di qualche
   aula $k \in \{1, \ldots, d\}$) {
   assegna il corso j all'aula k
   } ELSE {
5. apri l'aula $d + 1$
6. assegna il corso j all'aula $d + 1$
7. $d = d + 1$
   } }

1. richiede $O(n \log n)$. Il FOR 3. viene eseguito n volte. Per ogni aula, possiamo ricordarci il tempo di fine dell'ultimo corso assegnatoli (cosicchè per verificare che il corso j non si sovrappone ai corsi già inseriti in un' aula basta solo controllare che s_j sia maggiore di tale tempo di fine)

Complessità

Partizionamento_greedy ($s_1 \ldots s_n, f_1, \ldots, f_n$)
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)
3. FOR(j=1, j<n+1, j=j+1) {
4. IF (corso j non si sovrappone ai corsi di qualche
   aula $k \in \{1, \ldots, d\}$) {
   assegna il corso j all'aula k
   } ELSE {
5. apri l'aula $d + 1$
6. assegna il corso j all'aula $d + 1$
7. $d = d + 1$
   } }

1. richiede $O(n \log n)$. Il FOR 3. viene eseguito n volte. Per ogni aula, possiamo ricordarci il tempo di fine dell'ultimo corso assegnatoli (cosicchè per verificare che il corso j non si sovrappone ai corsi già inseriti in un' aula basta solo controllare che s_j sia maggiore di tale tempo di fine) Il numero di aule è $O(n)$, quindi l'istruzione 4. richiede tempo $O(n)$.

Complessità

Partizionamento_greedy ($s_1 \ldots s_n, f_1, \ldots, f_n$)
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)
3. FOR(j=1, j<n+1, j=j+1) $\{$
4. IF (corso j non si sovrappone ai corsi di qualche aula $k \in \{1, \ldots, d\}$) $\{$
   assegna il corso j all'aula k
   $\}$ ELSE $\{$
5. apri l'aula $d+1$
6. assegna il corso j all'aula $d+1$
7. $d = d+1$
   $\}$ $\}$

1. richiede $O(n \log n)$. Il FOR 3. viene eseguito n volte. Per ogni aula, possiamo ricordarci il tempo di fine dell'ultimo corso assegnatoli (cosicchè per verificare che il corso j non si sovrappone ai corsi già inseriti in un' aula basta solo controllare che s_j sia maggiore di tale tempo di fine) Il numero di aule è $O(n)$, quindi l'istruzione 4. richiede tempo $O(n)$. Il resto richiede tempo $O(1)$, quindi in totale l'algoritmo greedy richiede tempo $O(n^2)$

Complessità

Partizionamento_greedy $(s_1 \ldots s_n, f_1, \ldots, f_n)$
1. Ordina i corsi in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$
2. $d = 0$ %(d è il numero di aule utilizzate finora)
3. FOR(j=1, j<n+1, j=j+1) {
4. IF (corso j non si sovrappone ai corsi di qualche aula $k \in \{1, \ldots, d\}$) {
   assegna il corso j all'aula k
   } ELSE {
5. apri l'aula $d + 1$
6. assegna il corso j all'aula $d + 1$
7. $d = d + 1$
   } }

1. richiede $O(n \log n)$. Il `FOR 3.` viene eseguito n volte. Per ogni aula, possiamo ricordarci il tempo di fine dell'ultimo corso assegnatoli (cosicchè per verificare che il corso j non si sovrappone ai corsi già inseriti in un' aula basta solo controllare che s_j sia maggiore di tale tempo di fine) Il numero di aule è $O(n)$, quindi l'istruzione `4.` richiede tempo $O(n)$. Il resto richiede tempo $O(1)$, quindi in totale l'algoritmo greedy richiede tempo $O(n^2)$ (vedremo in seguito che un'implementazione più furba dell'algoritmo richiederà tempo $O(n \log n)$)

Input: Vettore di numeri $A = A[1]...A[n]$
Output: più lunga sottosequenza (non necessariamente contigua) di elementi *crescenti* di A

Input: Vettore di numeri $A = A[1]...A[n]$
Output: più lunga sottosequenza (non necessariamente contigua) di elementi *crescenti* di A

Detto in altri termini, cerchiamo il più grande valore di k per cui esistono interi $1 \leq i_1 < \ldots < i_k \leq n$ tali che $A[i_1] < \ldots < A[i_k]$.

Input: Vettore di numeri $A = A[1]...A[n]$
Output: più lunga sottosequenza (non necessariamente contigua) di elementi *crescenti* di A

Detto in altri termini, cerchiamo il più grande valore di k per cui esistono interi $1 \leq i_1 < \ldots < i_k \leq n$ tali che $A[i_1] < \ldots < A[i_k]$.

Ad esempio se abbiamo la sequenza

$$0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15$$

Input: Vettore di numeri $A = A[1]...A[n]$
Output: più lunga sottosequenza (non necessariamente contigua) di elementi *crescenti* di A

Detto in altri termini, cerchiamo il più grande valore di k per cui esistono interi $1 \leq i_1 < \ldots < i_k \leq n$ tali che $A[i_1] < \ldots < A[i_k]$.

Ad esempio se abbiamo la sequenza

$$\color{red}{0}, 8, 4, 12, \color{red}{2}, 10, \color{red}{6}, 14, 1, \color{red}{9}, 5, 13, 3, \color{red}{11}, 7, \color{red}{15}$$

una tra le più lunghe sottosequenze crescenti è quella in rosso

Sia

$$L(j) = \text{lunghezza della più lunga sottosequenza crescente}$$
$$\text{che termina nella posizione } j.$$

Sia

$$L(j) = \text{lunghezza della più lunga sottosequenza crescente che termina nella posizione } j.$$

Noi siamo ovviamente interessati alla lunghezza della più lunga sottosequenza crescente che finisce in una qualunque posizione, cioè al valore

$$\max\{L(1), L(2), \dots L(n)\}.$$

Sia

$$L(j) = \text{lunghezza della più lunga sottosequenza crescente}$$
$$\text{che termina nella posizione } j.$$

Noi siamo ovviamente interessati alla lunghezza della più lunga sottosequenza crescente che finisce in una qualunque posizione, cioè al valore

$$\max\{L(1), L(2), \ldots L(n)\}.$$

Per il calcolo di $L(j)$ chiedamoci: Quale puo è essere il *penultimo* elemento (sia esso $A[i]$) nella sottosequenza di lunghezza massima $L(j)$?

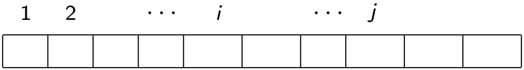
Sia

$$L(j) = \text{lunghezza della più lunga sottosequenza crescente}$$
$$\text{che termina nella posizione } j.$$

Noi siamo ovviamente interessati alla lunghezza della più lunga sottosequenza crescente che finisce in una qualunque posizione, cioè al valore

$$\max \{L(1), L(2), \ldots L(n)\}.$$

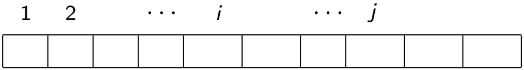
Per il calcolo di $L(j)$ chiedamoci: Quale puo è essere il *penultimo* elemento (sia esso $A[i]$) nella sottosequenza di lunghezza massima $L(j)$? (l'ultimo è ovviamente $A[j]$)



Sicuramente varrà che $A[i] < A[j]$, dato che la sottosequenza deve essere composta da elementi crescenti.

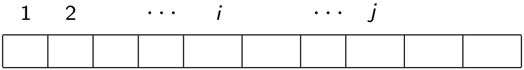
1	2		$\cdots$	i		$\cdots$	j		

Sicuramente varrà che $A[i] < A[j]$, dato che la sottosequenza deve essere composta da elementi crescenti. Inoltre, nella posizione i terminerà la più lunga sottosequenza crescente con ultimo elemento $A[i]$.



Sicuramente varrà che $A[i] < A[j]$, dato che la sottosequenza deve essere composta da elementi crescenti. Inoltre, nella posizione i terminerà la più lunga sottosequenza crescente con ultimo elemento $A[i]$. Pertanto

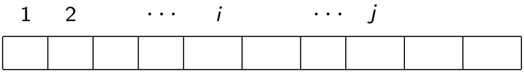
$$L(j) = \max\{L(i) : i < j \text{ e } A[i] < A[j]\} + 1.$$



Sicuramente varrà che $A[i] < A[j]$, dato che la sottosequenza deve essere composta da elementi crescenti. Inoltre, nella posizione i terminerà la più lunga sottosequenza crescente con ultimo elemento $A[i]$. Pertanto

$$L(j) = \max\{L(i) : i < j \text{ e } A[i] < A[j]\}+1.$$

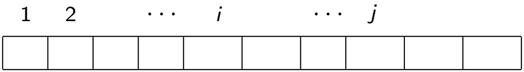
Valutiamo la complessità di un tale algoritmo. Ci basta un tempo $O(n)$ per il calcolo di *ciascun* $L(j)$, per $j = 1, \ldots, n$,



Sicuramente varrà che $A[i] < A[j]$, dato che la sottosequenza deve essere composta da elementi crescenti. Inoltre, nella posizione i terminerà la più lunga sottosequenza crescente con ultimo elemento $A[i]$. Pertanto

$$L(j) = \max\{L(i) : i < j \text{ e } A[i] < A[j]\} + 1.$$

Valutiamo la complessità di un tale algoritmo. Ci basta un tempo $O(n)$ per il calcolo di *ciascun* $L(j)$, per $j = 1, \ldots, n$, ed un tempo $O(n)$ per il calcolo di $\max\{L(1), L(2), \ldots L(n)\}$.



Sicuramente varrà che $A[i] < A[j]$, dato che la sottosequenza deve essere composta da elementi crescenti. Inoltre, nella posizione i terminerà la più lunga sottosequenza crescente con ultimo elemento $A[i]$. Pertanto

$$L(j) = \max\{L(i) : i < j \text{ e } A[i] < A[j]\} + 1.$$

Valutiamo la complessità di un tale algoritmo. Ci basta un tempo $O(n)$ per il calcolo di *ciascun* $L(j)$, per $j = 1, \ldots, n$, ed un tempo $O(n)$ per il calcolo di $\max\{L(1), L(2), \ldots L(n)\}$. In totale, ci basta tempo $O(n^2)$.

```
MaxSottosequenzaCrescente(A)
1. FOR (j=1, j<n+1, j=j+1) {
2.   L(j)=1
   }
```

```
MaxSottosequenzaCrescente(A)
1. FOR (j=1, j<n+1, j=j+1) {
2.   L(j)=1
   }
3. FOR (j=2, j<n+1, j=j+1) {
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
 }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
 }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
    IF ((A[i]<A[j])&&(L(j)<L(i)+1))
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
 }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
    IF ((A[i]<A[j])&&(L(j)<L(i)+1))
      L(j)=L(i)+1
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
 }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
    IF ((A[i]<A[j])&&(L(j)<L(i)+1))
      L(j)=L(i)+1
      }
    }
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
  }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
    IF ((A[i]<A[j])&&(L(j)<L(i)+1))
      L(j)=L(i)+1
      }
    }
max=L(1)
```

```
MaxSottosequenzaCrescente(A)
1. FOR (j=1, j<n+1, j=j+1) {
2.   L(j)=1
   }
3. FOR (j=2, j<n+1, j=j+1) {
4.   FOR (i=1, i<j, i=i+1) {
5.     IF ((A[i]<A[j])&&(L(j)<L(i)+1))
6.       L(j)=L(i)+1
       }
     }
7. max=L(1)
8. FOR (j=2, j<n+1, j=j+1) {
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
 }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
    IF ((A[i]<A[j])&&(L(j)<L(i)+1))
      L(j)=L(i)+1
      }
   }
max=L(1)
FOR (j=2, j<n+1, j=j+1) {
  IF (L(j)>max) {
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
 }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
    IF ((A[i]<A[j])&&(L(j)<L(i)+1))
      L(j)=L(i)+1
     }
    }
max=L(1)
FOR (j=2, j<n+1, j=j+1) {
  IF (L(j)>max) {
   max=L(j)
 }
```

```
MaxSottosequenzaCrescente(A)
1. FOR (j=1, j<n+1, j=j+1) {
2.   L(j)=1
   }
3. FOR (j=2, j<n+1, j=j+1) {
4.   FOR (i=1, i<j, i=i+1) {
5.     IF ((A[i]<A[j])&&(L(j)<L(i)+1))
6.       L(j)=L(i)+1
         }
      }
7. max=L(1)
8. FOR (j=2, j<n+1, j=j+1) {
9.   IF (L(j)>max) {
10.    max=L(j)
   }
11. RETURN max
```

```
MaxSottosequenzaCrescente(A)
FOR (j=1, j<n+1, j=j+1) {
  L(j)=1
  }
FOR (j=2, j<n+1, j=j+1) {
  FOR (i=1, i<j, i=i+1) {
    IF ((A[i]<A[j])&&(L(j)<L(i)+1))
      L(j)=L(i)+1
      }
    }
max=L(1)
FOR (j=2, j<n+1, j=j+1) {
  IF (L(j)>max) {
   max=L(j)
  }
RETURN max
```

Esercizio: trovare la sequenza crescente più lunga, e non solo la sua lunghezza.

Ancora un altro esercizio

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremità   quindi ogni taglio riduce la sua lunghezza della misura asportata.

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremità quindi ogni taglio riduce la sua lunghezza della misura asportata. Vogliamo determinare il numero massimo di segmenti che è possibile ottenere.

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremità quindi ogni taglio riduce la sua lunghezza della misura asportata. Vogliamo determinare il numero massimo di segmenti che è possibile ottenere.

Esempio: Sia $L = 10$, $n = 5$ e
$S[1] = 8, S[2] = 7, S[3] = 2, S[4] = 3, S[5] = 5.$

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremità quindi ogni taglio riduce la sua lunghezza della misura asportata. Vogliamo determinare il numero massimo di segmenti che è possibile ottenere.

Esempio: Sia $L = 10$, $n = 5$ e
$S[1] = 8, S[2] = 7, S[3] = 2, S[4] = 3, S[5] = 5$. Possiamo ottenere le seguenti possibili soluzioni: $(8, 2)$,

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremità quindi ogni taglio riduce la sua lunghezza della misura asportata. Vogliamo determinare il numero massimo di segmenti che è possibile ottenere.

Esempio: Sia $L = 10$, $n = 5$ e $S[1] = 8, S[2] = 7, S[3] = 2, S[4] = 3, S[5] = 5$. Possiamo ottenere le seguenti possibili soluzioni: $(8, 2), (7, 2)$,

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremità quindi ogni taglio riduce la sua lunghezza della misura asportata. Vogliamo determinare il numero massimo di segmenti che è possibile ottenere.

Esempio: Sia $L = 10$, $n = 5$ e
$S[1] = 8, S[2] = 7, S[3] = 2, S[4] = 3, S[5] = 5$. Possiamo ottenere le seguenti possibili soluzioni: $(8, 2), (7, 2), (7, 3)$,

Ancora un altro esercizio

Disponiamo di un tubo metallico di lunghezza L da cui vogliamo ottenere al più n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremità quindi ogni taglio riduce la sua lunghezza della misura asportata. Vogliamo determinare il numero massimo di segmenti che è possibile ottenere.

Esempio: Sia $L = 10$, $n = 5$ e $S[1] = 8, S[2] = 7, S[3] = 2, S[4] = 3, S[5] = 5$. Possiamo ottenere le seguenti possibili soluzioni: $(8, 2), (7, 2), (7, 3), (2, 3, 5)$

Idea per Greedy:

Ordiniamo le sezioni in senso non decrescente rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima.

Idea per Greedy:

Ordiniamo le sezioni in senso non decrescente rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima.

Procediamo quindi a segare prima il segmento più corto, poi quello successivo e così via finchè possibile

Idea per Greedy:

Ordiniamo le sezioni in senso non decrescente rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima.

Procediamo quindi a segare prima il segmento più corto, poi quello successivo e così via finchè possibile (cioè fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento).

MassimoNumero$(L, S[1 \ldots n])$

```
MassimoNumero(L, S[1 ... n])
1.  Ordina S in senso crescente
```

```
MassimoNumero(L, S[1...n])
    Ordina S in senso crescente
    i=1
    WHILE(i<n+1&&L ≥ S[i]){
```

```
MassimoNumero(L, S[1 ... n])
Ordina S in senso crescente
i=1
WHILE(i<n+1&&L ≥ S[i]){
    L = L − S[i]
```

```
MassimoNumero(L, S[1...n])
  Ordina S in senso crescente
  i=1
  WHILE(i<n+1&&L ≥ S[i]){
     L = L − S[i]
     i=i+1
  }
```

```
MassimoNumero(L, S[1...n])
1.  Ordina S in senso crescente
2.  i=1
3.  WHILE(i<n+1&&L ≥ S[i]){
4.      L = L − S[i]
5.      i=i+1
    }
6.  return i-1
```

```
MassimoNumero(L, S[1...n])
   Ordina S in senso crescente
   i=1
   WHILE(i<n+1&&L ≥ S[i]){
       L = L − S[i]
      i=i+1
   }
   return i-1
```

L'operazione di ordinamento può essere fatta in tempo $\Theta(n \log n)$.

```
MassimoNumero(L, S[1 ... n])
Ordina S in senso crescente
i=1
WHILE(i<n+1&&L ≥ S[i]){
     L = L − S[i]
    i=i+1
}
return i-1
```

L'operazione di ordinamento può essere fatta in tempo $\Theta(n \log n)$. Il successivo ciclo while ha costo $\Theta(n)$ nel caso peggiore.

```
MassimoNumero(L, S[1...n])
Ordina S in senso crescente
i=1
WHILE(i<n+1&&L ≥ S[i]){
    L = L − S[i]
    i=i+1
}
return i-1
```

L'operazione di ordinamento può essere fatta in tempo $\Theta(n \log n)$. Il successivo ciclo while ha costo $\Theta(n)$ nel caso peggiore. Il costo complessivo dell'algoritmo risulta quindi $\Theta(n \log n)$.

```
MassimoNumero(L, S[1...n])
    Ordina S in senso crescente
    i=1
    WHILE(i<n+1&&L ≥ S[i]){
        L = L − S[i]
        i=i+1
    }
    return i-1
```

L'operazione di ordinamento può essere fatta in tempo $\Theta(n \log n)$. Il successivo ciclo while ha costo $\Theta(n)$ nel caso peggiore. Il costo complessivo dell'algoritmo risulta quindi $\Theta(n \log n)$.

Proviamo che l'algoritmo ritorna una soluzione ottima, ovvero per cui il numero di segmenti ritornati è il massimo possibile.

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0,

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$.

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima.

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$.

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$.

Proviamo che esiste una soluzione della stessa cardinalità k, ma della forma $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$, dove $S[1]$ è la prima scelta effettuata dall'algoritmo Greedy.

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$.

Proviamo che esiste una soluzione della stessa cardinalità k, ma della forma $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$, dove $S[1]$ è la prima scelta effettuata dall'algoritmo Greedy.

Infatti, poichè $S[1] = \min\{S[1], \ldots, S[n]\}$, segue che $S[1] \leq S[i_1]$,

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$.

Proviamo che esiste una soluzione della stessa cardinalità k, ma della forma $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$, dove $S[1]$ è la prima scelta effettuata dall'algoritmo Greedy.

Infatti, poichè $S[1] = \min\{S[1], \ldots, S[n]\}$, segue che $S[1] \leq S[i_1]$, da cui $S[1] + \sum_{j=2}^{k} S[i_j] \leq \sum_{j=1}^{k} S[i_j] \leq L$

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$.

Proviamo che esiste una soluzione della stessa cardinalità k, ma della forma $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$, dove $S[1]$ è la prima scelta effettuata dall'algoritmo Greedy.

Infatti, poichè $S[1] = \min\{S[1], \ldots, S[n]\}$, segue che $S[1] \leq S[i_1]$, da cui $S[1] + \sum_{j=2}^{k} S[i_j] \leq \sum_{j=1}^{k} S[i_j] \leq L$ (quindi $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$ è una soluzione corretta)

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$.

Proviamo che esiste una soluzione della stessa cardinalità k, ma della forma $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$, dove $S[1]$ è la prima scelta effettuata dall'algoritmo Greedy.

Infatti, poichè $S[1] = \min\{S[1], \ldots, S[n]\}$, segue che $S[1] \leq S[i_1]$, da cui $S[1] + \sum_{j=2}^{k} S[i_j] \leq \sum_{j=1}^{k} S[i_j] \leq L$ (quindi $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$ è una soluzione corretta) con un numero di segmenti pari a $(k-1) + 1 = k$, cioè il massimo possibile.

Osserviamo che se $\min\{S[1],\ldots,S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo è ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1\ldots n]$.

Supponiamo quindi che $\min\{S[1],\ldots,S[n]\} \leq L$. Sia k il massimo numero di segmenti che è possibile ottenere, e siano $S[i_1],\ldots,S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$.

Proviamo che esiste una soluzione della stessa cardinalità k, ma della forma $(\{S[i_1],\ldots,S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$, dove $S[1]$ è la prima scelta effettuata dall'algoritmo Greedy.

Infatti, poichè $S[1] = \min\{S[1],\ldots,S[n]\}$, segue che $S[1] \leq S[i_1]$, da cui $S[1] + \sum_{j=2}^{k} S[i_j] \leq \sum_{j=1}^{k} S[i_j] \leq L$ (quindi $(\{S[i_1],\ldots,S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$ è una soluzione corretta) con un numero di segmenti pari a $(k-1)+1 = k$, cioè il massimo possibile.

A questo punto, possiamo usare induzione sulla lunghezza $L - S[1]$ e provare che l'algoritmo è ottimo.

ed un altro esercizio ancora:

ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli.

ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$.

ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$. Diremo che il rettangolo i *contiene* il rettangolo j se e solo se sia la base che l'altezza del rettangolo i sono maggiori della base ed altezza del rettangolo j,

ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$. Diremo che il rettangolo i *contiene* il rettangolo j se e solo se sia la base che l'altezza del rettangolo i sono maggiori della base ed altezza del rettangolo j, ovvero se e solo se

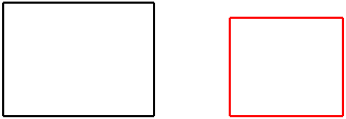
$$b(i) \geq b(j) \ \text{ e } \ h(i) \geq h(j).$$

ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$. Diremo che il rettangolo i *contiene* il rettangolo j se e solo se sia la base che l'altezza del rettangolo i sono maggiori della base ed altezza del rettangolo j, ovvero se e solo se

$$b(i) \geq b(j) \;\text{ e }\; h(i) \geq h(j).$$

Ad esempio, il rettangolo rosso è contenuto nel nero,



ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$. Diremo che il rettangolo i *contiene* il rettangolo j se e solo se sia la base che l'altezza del rettangolo i sono maggiori della base ed altezza del rettangolo j, ovvero se e solo se

$$b(i) \geq b(j) \;\; \text{e} \;\; h(i) \geq h(j).$$

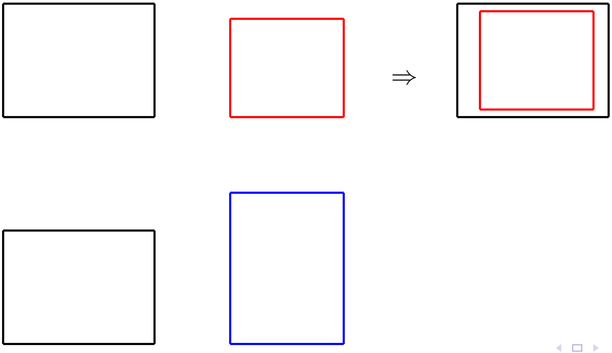
Ad esempio, il rettangolo rosso è contenuto nel nero,

ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$. Diremo che il rettangolo i *contiene* il rettangolo j se e solo se sia la base che l'altezza del rettangolo i sono maggiori della base ed altezza del rettangolo j, ovvero se e solo se

$$b(i) \geq b(j) \;\text{ e }\; h(i) \geq h(j).$$

Ad esempio, il rettangolo rosso è contenuto nel nero, mentre il blu **no**

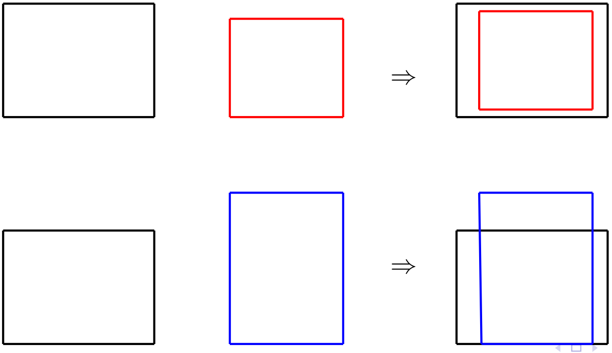


ed un altro esercizio ancora:

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$. Diremo che il rettangolo i *contiene* il rettangolo j se e solo se sia la base che l'altezza del rettangolo i sono maggiori della base ed altezza del rettangolo j, ovvero se e solo se

$$b(i) \geq b(j) \ \text{ e } \ h(i) \geq h(j).$$

Ad esempio, il rettangolo rosso è contenuto nel nero, mentre il blu **no**



Diremo che il sottoinsieme di rettangoli $S \subseteq R = \{1, 2, \ldots, n\}$ contiene tutto l'insieme R se
$\forall$ rettangolo $j \in R$ esiste un rettangolo $i \in S$ che contiene j.

Diremo che il sottoinsieme di rettangoli $S \subseteq R = \{1, 2, \ldots, n\}$ contiene tutto l'insieme R se
$\forall$ rettangolo $j \in R$ esiste un rettangolo $i \in S$ che contiene j.

Problema:
Input: Insieme di rettangoli $R = \{1, 2, \ldots, n\}$
Output: $S \subseteq R$ di cardinalità *minima* che contiene tutto R.

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

```
1. S={1}
2. j=1
```

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

```
1. S={1}
2. j=1
3. FOR (i=2, i<n+1, i=i+1) {
```

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

```
1. S={1}
2. j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.   IF (h(i)> h(j)) {
```

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

```
1. S={1}
2. j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.   IF (h(i)> h(j)) {
5.     S= S∪{i}
```

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

```
S={1}
j=1
FOR (i=2, i<n+1, i=i+1) {
  IF (h(i)> h(j)) {
    S= S∪{i}
    j= i
```

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

```
S={1}
j=1
FOR (i=2, i<n+1, i=i+1) {
  IF (h(i)> h(j)) {
    S= S∪{i}
    j= i
      }
    }
```

Ordiniamo i rettangoli in R per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$).

Idea per un algoritmo greedy:

Passo 1: 1 è il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i è contenuto in j ignoriamo i, altrimenti mettiamo i in S.

```
1. S={1}
2. j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.   IF (h(i)> h(j)) {
5.     S= S∪{i}
6.     j= i
        }
      }
7. RETURN S
```

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare.

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare. Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare.
Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.
Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$,

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare.
Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.
Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$, da cui discende necessariamente che $b(i) = b(1)$.

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare.
Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.
Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$, da cui discende necessariamente che $b(i) = b(1)$.

Ricordiamo anche che i rettangoli di base uguale erano ordinati per altezza decrescente, quindi $h(i) \leq h(1)$,

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare.
Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.
Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$, da cui discende necessariamente che $b(i) = b(1)$.

Ricordiamo anche che i rettangoli di base uguale erano ordinati per altezza decrescente, quindi $h(i) \leq h(1)$, e di nuovo ne segue che $h(i) = h(1)$.

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare. Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.
Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$, da cui discende necessariamente che $b(i) = b(1)$.

Ricordiamo anche che i rettangoli di base uguale erano ordinati per altezza decrescente, quindi $h(i) \leq h(1)$, e di nuovo ne segue che $h(i) = h(1)$. Pertanto i rettangoli i e 1 hanno le stesse dimensioni, da cui si ha che anche $S' = S - \{i\} \cup \{1\}$ copre tutto R,

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare.
Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.
Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$, da cui discende necessariamente che $b(i) = b(1)$.

Ricordiamo anche che i rettangoli di base uguale erano ordinati per altezza decrescente, quindi $h(i) \leq h(1)$, e di nuovo ne segue che $h(i) = h(1)$. Pertanto i rettangoli i e 1 hanno le stesse dimensioni, da cui si ha che anche $S' = S - \{i\} \cup \{1\}$ copre tutto R, con $1 \in S'$ e S' ottimo (in quanto $|S'| = |S|$).

Proviamo innazitutto che *esiste* una soluzione di cardinalità minima (ottima) che contiene il rettangolo 1.

Sia S una soluzione ottima, se $1 \in S$ allora non vi è nulla da provare. Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$.
Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$, da cui discende necessariamente che $b(i) = b(1)$.

Ricordiamo anche che i rettangoli di base uguale erano ordinati per altezza decrescente, quindi $h(i) \leq h(1)$, e di nuovo ne segue che $h(i) = h(1)$. Pertanto i rettangoli i e 1 hanno le stesse dimensioni, da cui si ha che anche $S' = S - \{i\} \cup \{1\}$ copre tutto R, con $1 \in S'$ e S' ottimo (in quanto $|S'| = |S|$).

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'è nulla da provare.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'è nulla da provare.
Se $k \notin S$, allora S deve possedere un rettangolo j che contiene k, ovvero per cui $b(j) \geq b(k)$ e $h(j) \geq h(k) > h(1)$.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'è nulla da provare.
Se $k \notin S$, allora S deve possedere un rettangolo j che contiene k, ovvero per cui $b(j) \geq b(k)$ e $h(j) \geq h(k) > h(1)$. Essendo k il più piccolo intero per cui $h(k) > h(1)$, abbiamo che $j > k$.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'è nulla da provare.
Se $k \notin S$, allora S deve possedere un rettangolo j che contiene k, ovvero per cui $b(j) \geq b(k)$ e $h(j) \geq h(k) > h(1)$. Essendo k il più piccolo intero per cui $h(k) > h(1)$, abbiamo che $j > k$. A causa dell'ordinamento decrescente delle basi, si ha che $b(j) \leq b(k)$ e quindi $b(j) = b(k)$.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'è nulla da provare.
Se $k \notin S$, allora S deve possedere un rettangolo j che contiene k, ovvero per cui $b(j) \geq b(k)$ e $h(j) \geq h(k) > h(1)$. Essendo k il più piccolo intero per cui $h(k) > h(1)$, abbiamo che $j > k$. A causa dell'ordinamento decrescente delle basi, si ha che $b(j) \leq b(k)$ e quindi $b(j) = b(k)$.

A causa dell'ordinamento decrescente delle altezze di rettangoli con basi uguali, si ha che $h(j) \leq h(k)$, da cui, si ha che $h(j) = h(k)$.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'è nulla da provare.
Se $k \notin S$, allora S deve possedere un rettangolo j che contiene k, ovvero per cui $b(j) \geq b(k)$ e $h(j) \geq h(k) > h(1)$. Essendo k il più piccolo intero per cui $h(k) > h(1)$, abbiamo che $j > k$. A causa dell'ordinamento decrescente delle basi, si ha che $b(j) \leq b(k)$ e quindi $b(j) = b(k)$.

A causa dell'ordinamento decrescente delle altezze di rettangoli con basi uguali, si ha che $h(j) \leq h(k)$, da cui, si ha che $h(j) = h(k)$. Pertanto, i rettangoli k e j hanno le stesse dimensioni.

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy.

Proviamo che esiste una soluzione ottima che contiene 1 e k. Se k è la seconda scelta effettuata, k è il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'è nulla da provare.
Se $k \notin S$, allora S deve possedere un rettangolo j che contiene k, ovvero per cui $b(j) \geq b(k)$ e $h(j) \geq h(k) > h(1)$. Essendo k il più piccolo intero per cui $h(k) > h(1)$, abbiamo che $j > k$. A causa dell'ordinamento decrescente delle basi, si ha che $b(j) \leq b(k)$ e quindi $b(j) = b(k)$.

A causa dell'ordinamento decrescente delle altezze di rettangoli con basi uguali, si ha che $h(j) \leq h(k)$, da cui, si ha che $h(j) = h(k)$. Pertanto, i rettangoli k e j hanno le stesse dimensioni.

Ne segue che anche $S' = S - \{j\} \cup \{k\}$ copre tutto R, con $1, k \in S'$ e S' ottimo (in quanto $|S'| = |S|$).

Iterando... possiamo concludere che esiste una soluzione ottima che contiene **tutte** le scelte effettuate dall'algoritmo Greedy, *ergo*, l'algoritmo Greedy produce una soluzione ottima al problema.

Iterando... possiamo concludere che esiste una soluzione ottima che contiene **tutte** le scelte effettuate dall'algoritmo Greedy, *ergo*, l'algoritmo Greedy produce una soluzione ottima al problema.

Complessità $O(n)$ se i rettangoli sono già ordinati come richiesto, alrimenti $O(n \log n)$.

ed un altro:

Supponiamo di avere n oggetti $\{1, \ldots, n\}$.

ed un altro:

Supponiamo di avere n oggetti $\{1, \ldots, n\}$.
Ognuno di essi deve essere immagazzinato e tenuto nell'opportuno range di temperatura. In altri termini, l'oggetto i deve essere conservato ad una temperatura che deve essere compresa nell'intervallo $[s_i, f_i]$, $i = 1, \ldots n$.

ed un altro:

Supponiamo di avere n oggetti $\{1, \ldots, n\}$.
Ognuno di essi deve essere immagazzinato e tenuto nell'opportuno range di temperatura. In altri termini, l'oggetto i deve essere conservato ad una temperatura che deve essere compresa nell'intervallo $[s_i, f_i]$, $i = 1, \ldots n$.

Determinare il minimo numero di celle frigorifere in cui possiamo immagazzinare i nostri oggetti.

ed un altro:

Supponiamo di avere n oggetti $\{1, \ldots, n\}$.
Ognuno di essi deve essere immagazzinato e tenuto nell'opportuno range di temperatura. In altri termini, l'oggetto i deve essere conservato ad una temperatura che deve essere compresa nell'intervallo $[s_i, f_i]$, $i = 1, \ldots n$.

Determinare il minimo numero di celle frigorifere in cui possiamo immagazzinare i nostri oggetti.
Formalizziamo del problema nel modo seguente:

Input: n segmenti $[s_1, f_1], \ldots, [s_n, f_n]$;
Output: un insieme $S = \{x_1, \ldots, x_m\}$ di cardinalità minima tale che *per ogni* segmento $[s_i, f_i]$ esiste un elemento $x \in S$ per cui $s_i \leq x \leq f_i$.

Diciamo che x *copre* un intervallo generico $[s, t]$ se $s \leq x \leq f$.

Diciamo che x *copre* un intervallo generico $[s, t]$ se $s \leq x \leq f$.

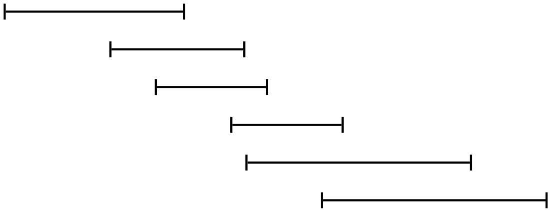
Un primo algoritmo Greedy che potremmo progettare per il problema potrebbe essere quello che sceglie i punti x da inserire in S in modo tale che, ad ogni scelta, x sia il punto che copre il maggior numero di intervalli non ancora coperti.

Diciamo che x *copre* un intervallo generico $[s, t]$ se $s \leq x \leq f$.

Un primo algoritmo Greedy che potremmo progettare per il problema potrebbe essere quello che sceglie i punti x da inserire in S in modo tale che, ad ogni scelta, x sia il punto che copre il maggior numero di intervalli non ancora coperti.
Purtoppo, ciò non porterebbe ad una soluzione di cardinalià minima.

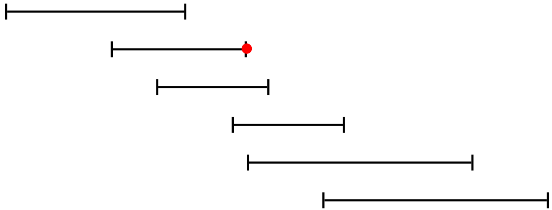
Diciamo che x *copre* un intervallo generico $[s, t]$ se $s \leq x \leq f$.

Un primo algoritmo Greedy che potremmo progettare per il problema potrebbe essere quello che sceglie i punti x da inserire in S in modo tale che, ad ogni scelta, x sia il punto che copre il maggior numero di intervalli non ancora coperti.
Purtoppo, ciò non porterebbe ad una soluzione di cardinalià minima.



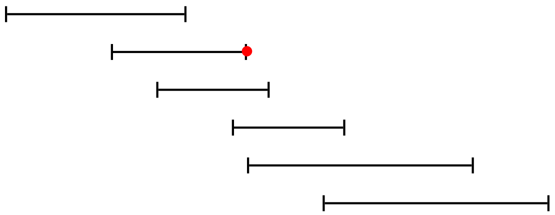
Diciamo che x *copre* un intervallo generico $[s, t]$ se $s \leq x \leq f$.

Un primo algoritmo Greedy che potremmo progettare per il problema potrebbe essere quello che sceglie i punti x da inserire in S in modo tale che, ad ogni scelta, x sia il punto che copre il maggior numero di intervalli non ancora coperti.
Purtoppo, ciò non porterebbe ad una soluzione di cardinalià minima.



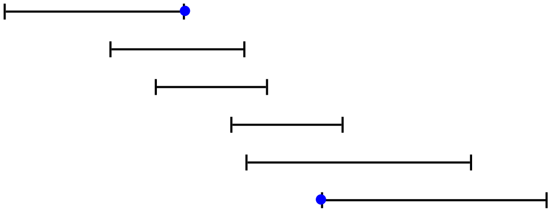
Diciamo che x *copre* un intervallo generico $[s, t]$ se $s \leq x \leq f$.

Un primo algoritmo Greedy che potremmo progettare per il problema potrebbe essere quello che sceglie i punti x da inserire in S in modo tale che, ad ogni scelta, x sia il punto che copre il maggior numero di intervalli non ancora coperti.
Purtoppo, ciò non porterebbe ad una soluzione di cardinalià minima.



Avremmo poi bisogno di altri due punti per coprire il primo ed ultimo intervallo, per un totale di 3 punti.

Che questa non sia una soluzione ottima lo si evince dal fatto che i due punti in **blu** coprono tutti gli intervalli



1. Ordina gli intervalli in base ai valori f_i, dal più piccolo al più grande

1. Ordina gli intervalli in base ai valori f_i, dal più piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.

1. Ordina gli intervalli in base ai valori f_i, dal più piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.
3. Elimina tutti gli intervalli coperti da f_1.

1. Ordina gli intervalli in base ai valori f_i, dal più piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.
3. Elimina tutti gli intervalli coperti da f_1.
4. Itera sugli intervalli rimanenti.

1. Ordina gli intervalli in base ai valori f_i, dal più piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.
3. Elimina tutti gli intervalli coperti da f_1.
4. Itera sugli intervalli rimanenti.
5. Restituisci S.

1. `Ordina gli intervalli in base ai valori` f_i`, dal più piccolo al più grande`
2. `Inserisci nella soluzione` S `il punto` f_1`.`
3. `Elimina tutti gli intervalli coperti da` f_1`.`
4. `Itera sugli intervalli rimanenti.`
5. `Restituisci` S`.`

L'algoritmo si può implementare in modo che abbia complessità $O(n \log n)$.

```
1. Ordina gli intervalli in base ai valori f_i, dal più
   piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.
3. Elimina tutti gli intervalli coperti da f_1.
4. Itera sugli intervalli rimanenti.
5. Restituisci S.
```

L'algoritmo si può implementare in modo che abbia complessità $O(n \log n)$.
Proviamo innanzitutto che esiste una soluzione di cardinalità minima che contiene il punto f_1.

1. Ordina gli intervalli in base ai valori f_i, dal più piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.
3. Elimina tutti gli intervalli coperti da f_1.
4. Itera sugli intervalli rimanenti.
5. Restituisci S.

L'algoritmo si può implementare in modo che abbia complessità $O(n \log n)$.
Proviamo innanzitutto che esiste una soluzione di cardinalità minima che contiene il punto f_1. Sia S una generica soluzione di cardinalità minima.

```
1. Ordina gli intervalli in base ai valori fi, dal più
   piccolo al più grande
2. Inserisci nella soluzione S il punto f1.
3. Elimina tutti gli intervalli coperti da f1.
4. Itera sugli intervalli rimanenti.
5. Restituisci S.
```

L'algoritmo si può implementare in modo che abbia complessità $O(n \log n)$.
Proviamo innanzitutto che esiste una soluzione di cardinalità minima che contiene il punto f_1. Sia S una generica soluzione di cardinalità minima. Se essa contiene f_1, non c'è null'altro da provare. Se $f_1 \notin S$, allora poichè ci deve essere un punto x in S che copre l'intervallo $[s_1, f_1]$, deve necessariamente valere che $x < f_1$.

```
1. Ordina gli intervalli in base ai valori f_i, dal più
   piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.
3. Elimina tutti gli intervalli coperti da f_1.
4. Itera sugli intervalli rimanenti.
5. Restituisci S.
```

L'algoritmo si può implementare in modo che abbia complessità $O(n \log n)$.
Proviamo innanzitutto che esiste una soluzione di cardinalità minima che contiene il punto f_1. Sia S una generica soluzione di cardinalità minima. Se essa contiene f_1, non c'è null'altro da provare. Se $f_1 \notin S$, allora poichè ci deve essere un punto x in S che copre l'intervallo $[s_1, f_1]$, deve necessariamente valere che $x < f_1$. L'insieme $S' = (S \setminus \{x\}) \cup \{f_1\}$ ha la stessa cardinalità di S e chiaramente copre gli stessi intervalli (cioè , tutti) di S.

1. `Ordina gli intervalli in base ai valori` f_i, `dal più piccolo al più grande`
2. `Inserisci nella soluzione` S `il punto` f_1.
3. `Elimina tutti gli intervalli coperti da` f_1.
4. `Itera sugli intervalli rimanenti.`
5. `Restituisci` S.

L'algoritmo si può implementare in modo che abbia complessità $O(n \log n)$.
Proviamo innanzitutto che esiste una soluzione di cardinalità minima che contiene il punto f_1. Sia S una generica soluzione di cardinalità minima. Se essa contiene f_1, non c'è null'altro da provare. Se $f_1 \notin S$, allora poichè ci deve essere un punto x in S che copre l'intervallo $[s_1, f_1]$, deve necessariamente valere che $x < f_1$. L'insieme $S' = (S \setminus \{x\}) \cup \{f_1\}$ ha la stessa cardinalità di S e chiaramente copre gli stessi intervalli (cioè , tutti) di S. Il resto per esercizio...

l'ultimo:

Input: n brani musicali $\{1, 2, \ldots, n\}$ di durata $d_1, d_2, \ldots, d_n$, ed un CD di dimensione D.

l'ultimo:

Input: n brani musicali $\{1, 2, \ldots, n\}$ di durata $d_1, d_2, \ldots, d_n$, ed un CD di dimensione D.
Output: il maggior numero di brani che possiamo memorizzare sul CD.

l'ultimo:

Input: n brani musicali $\{1, 2, \ldots, n\}$ di durata $d_1, d_2, \ldots, d_n$, ed un CD di dimensione D.
Output: il maggior numero di brani che possiamo memorizzare sul CD.

Supponiamo per semplicità che le durate siano distinte e ordinate in senso crescente $d_1 < d_2 < \ldots < d_n$.

l'ultimo:

Input: n brani musicali $\{1, 2, \ldots, n\}$ di durata $d_1, d_2, \ldots, d_n$, ed un CD di dimensione D.
Output: il maggior numero di brani che possiamo memorizzare sul CD.

Supponiamo per semplicità che le durate siano distinte e ordinate in senso crescente $d_1 < d_2 < \ldots < d_n$.
Un possibile algoritmo consiste nel memorizzare i brani nell'ordine dal più piccolo al più grande,

l'ultimo:

Input: n brani musicali $\{1, 2, \ldots, n\}$ di durata $d_1, d_2, \ldots, d_n$, ed un CD di dimensione D.
Output: il maggior numero di brani che possiamo memorizzare sul CD.

Supponiamo per semplicità che le durate siano distinte e ordinate in senso crescente $d_1 < d_2 < \ldots < d_n$.
Un possibile algoritmo consiste nel memorizzare i brani nell'ordine dal più piccolo al più grande, fin quando il CD non ne può contenere più

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Esiste una soluzione ottima che contiene il brano di durata d_1.

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Esiste una soluzione ottima che contiene il brano di durata d_1.

Sia S un insieme di brani ottimo per cui $1 \notin S$,

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Esiste una soluzione ottima che contiene il brano di durata d_1.

Sia S un insieme di brani ottimo per cui $1 \notin S$, detto $j > i$ il primo brano per cui $j \in S$,

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Esiste una soluzione ottima che contiene il brano di durata d_1.

Sia S un insieme di brani ottimo per cui $1 \notin S$, detto $j > i$ il primo brano per cui $j \in S$, allora $d_j > d_1$, e posso sostituire j con 1,

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Esiste una soluzione ottima che contiene il brano di durata d_1.

Sia S un insieme di brani ottimo per cui $1 \notin S$, detto $j > i$ il primo brano per cui $j \in S$, allora $d_j > d_1$, e posso sostituire j con 1, rispettando il vincolo sulla somma delle durate dei brani.

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Esiste una soluzione ottima che contiene il brano di durata d_1.

Sia S un insieme di brani ottimo per cui $1 \notin S$, detto $j > i$ il primo brano per cui $j \in S$, allora $d_j > d_1$, e posso sostituire j con 1, rispettando il vincolo sulla somma delle durate dei brani.

Otteniamo quindi un'altro insieme ottimo di $|S|$ brani che contiene questa volta il brano musicale 1.

Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo.

Esiste una soluzione ottima che contiene il brano di durata d_1.

Sia S un insieme di brani ottimo per cui $1 \notin S$, detto $j > i$ il primo brano per cui $j \in S$, allora $d_j > d_1$, e posso sostituire j con 1, rispettando il vincolo sulla somma delle durate dei brani.

Otteniamo quindi un'altro insieme ottimo di $|S|$ brani che contiene questa volta il brano musicale 1.

Il resto per esercizio....