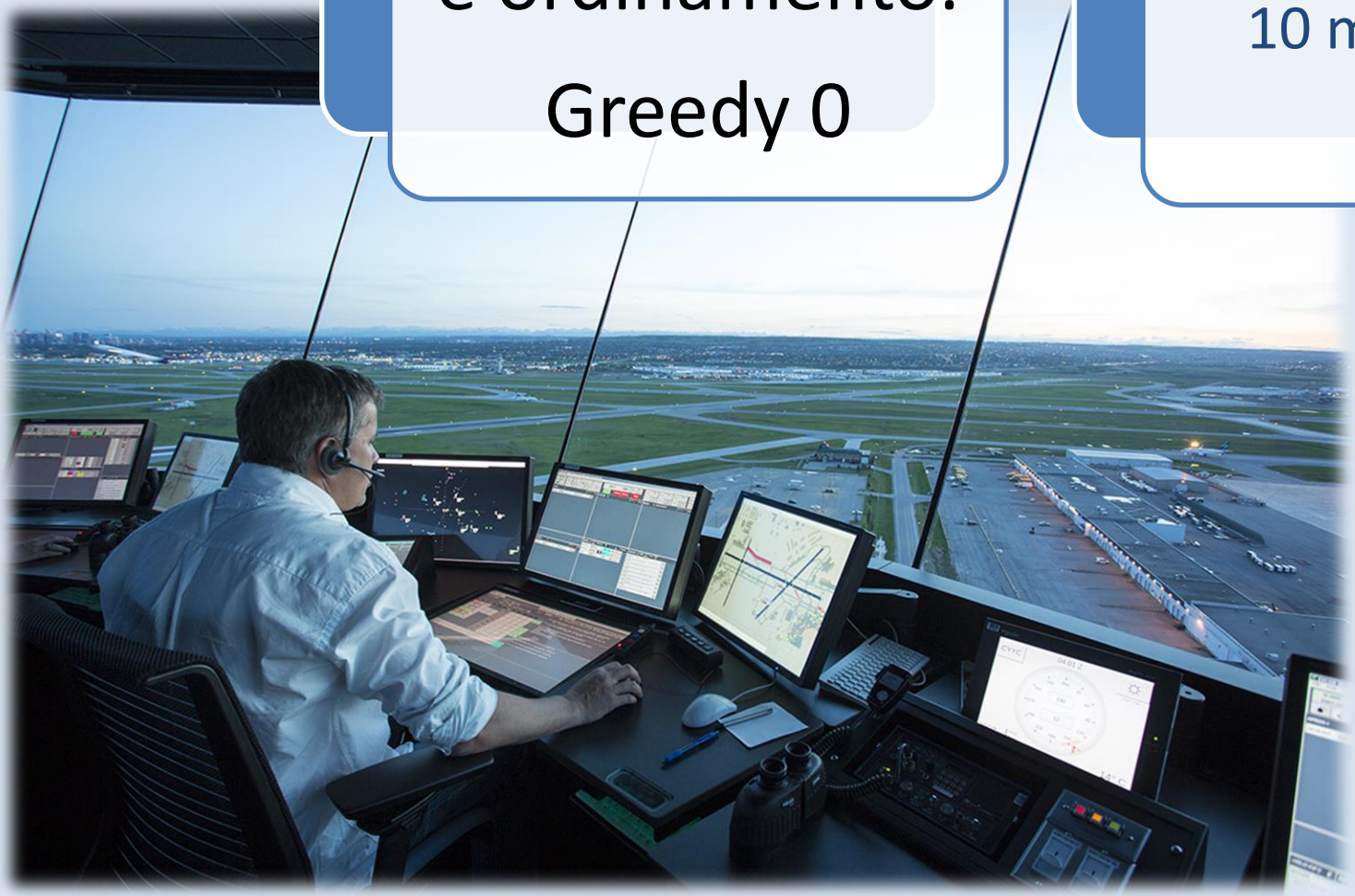
Code a priorità e ordinamento.

Greedy 0

10 maggio 2023



Priorita'

Studiamo una struttura dati per "gestire" elementi a cui è assegnata una priorità

- *Esempio*: centro di controllo aereo. Si stabiliscono delle priorità tra i voli per decidere quale deve atterrare prima. La lista è dinamica.

  Talvolta la priorità può cambiare (ad esempio, un aereo ha poco carburante, deve atterrare per primo)…

- Algoritmi su grafi per cammini minimi o per il MST: ai vertici sono abbinate delle priorità, in base alle quali vengono considerati nell'algoritmo.
- Più in generale per algoritmi "greedy".

Il TDA PriorityQueue

- Contenitore di oggetti
- Ogni oggetto è una coppia (key, element) [(chiave, elemento)], dove key è la priorità abbinata a element.
- Metodi principali
  - FindMin()
    restituisce l'oggetto (k, x) con chiave minima
  - Insert(k, x)
    inserisce un oggetto (k, x) (inserisce l'elemento x con priorità k)
  - ExtractMin()
    rimuove l'oggetto con chiave (key) più piccola, restituendo in output il relative elemento

(Analogamente la priorità potrebbe essere basata sul massimo)

Implementazioni

1. Manteniamo gli oggetti in una *lista* (senza nessun ordine)
   - FindMin() in tempo $O(n)$
   - Insert(k, x) in tempo $O(1)$
   - ExtractMin() in tempo $O(n)$

Supponiamo che il confronto tra due chiavi sia fatto in tempo $O(1)$.

2. Manteniamo in una *lista* gli oggetti ordinati sulle chiavi
   - FindMin() in tempo $O(1)$
   - Insert(k, x) in tempo $O(n)$
   - ExtractMin() in tempo $O(1)$

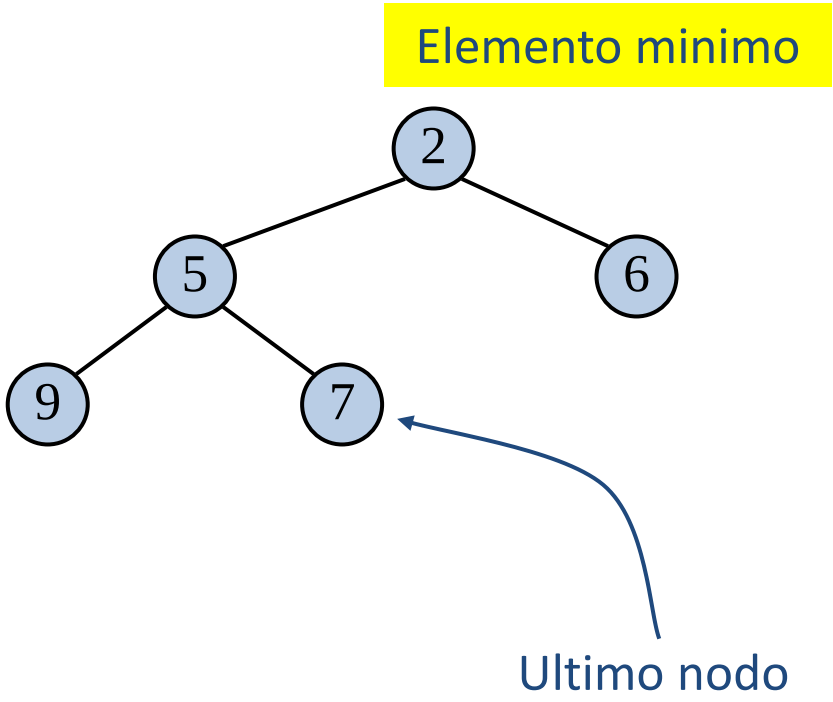
1. Manteniamo gli oggetti in una struttura dati chiamata heap, che usa il "potere gerarchico" degli alberi binari
   - FindMin() in tempo $O(1)$
   - Insert(k, x) in tempo $O(\log n)$
   - ExtractMin() in tempo $O(\log n)$

Heap binario

- Un min-heap (binario) è un albero binario che conserva una collezione di oggetti nei suoi nodi, e soddisfa le seguenti proprietà:
  - Albero binario completo:

    ogni livello è completo tranne al più l'ultimo che è riempito da sinistra a destra
  - Heap-Order:

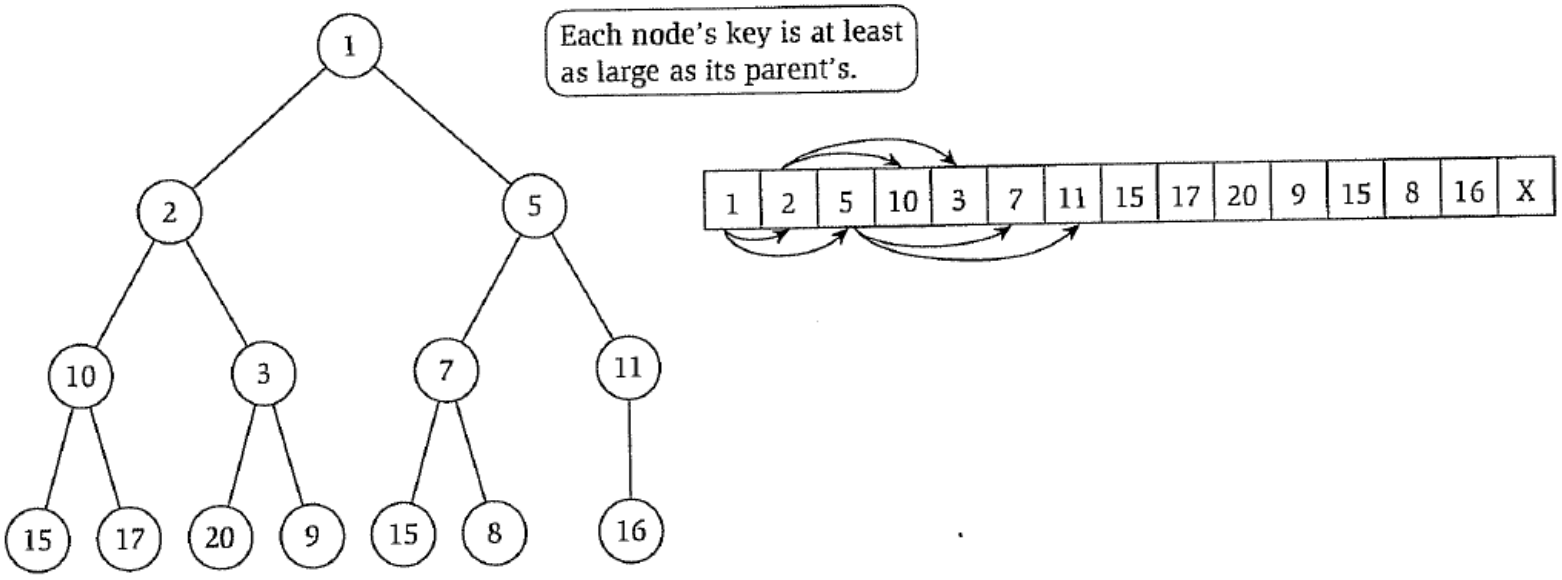
    per ogni nodo v, diverso dalla root, si ha:
    $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$



Heap binario e array

Se è nota una limitazione superiore N sul numero di elementi che saranno nell'heap, si può evitare una struttura ad albero (con puntatori) e rappresentare un heap con un array H dove:

H[1] è la radice e per ogni nodo in posizione i,
leftChild(i) = 2i, rightChild(i) = 2i+1 e parent(i) = $\lfloor i/2 \rfloor$



Proprietà degli alberi binari completi (e quindi degli Heap)

Proprietà (da provare per esercizio).

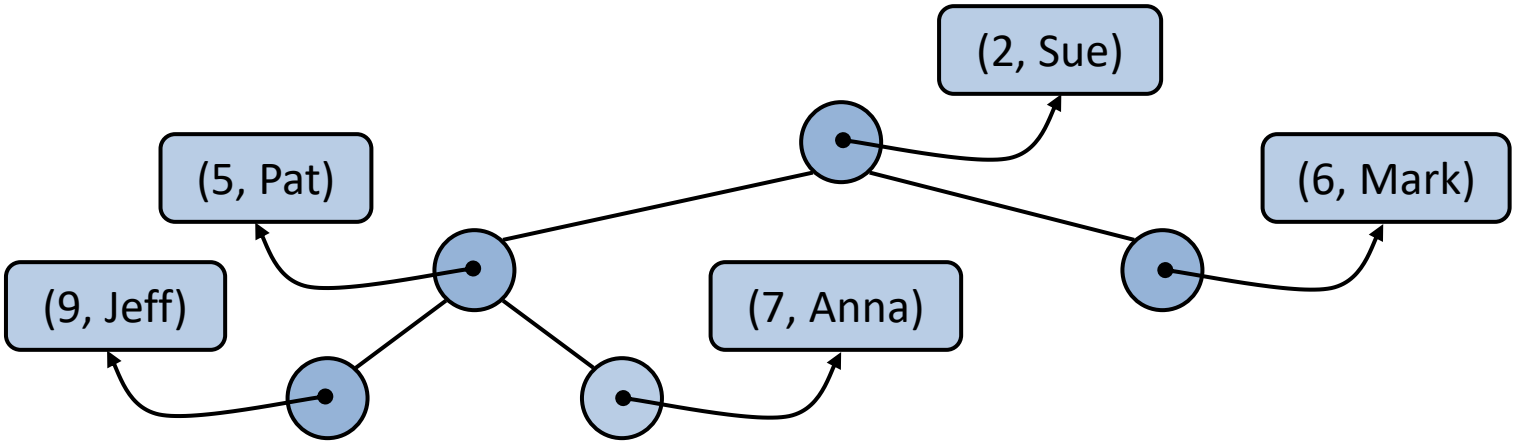
In un albero binario completo $A[1..n]$ con n nodi

- l'altezza è $\lfloor \log n \rfloor$
- gli elementi in $A[(\lfloor n/2 \rfloor+1).. n]$ sono tutte (e sole) le foglie dell'albero
- I nodi ad altezza h sono al più $\lceil n/2^{h+1} \rceil$

Nota: l'altezza di un nodo i è la massima distanza di i da una foglia sua discendente

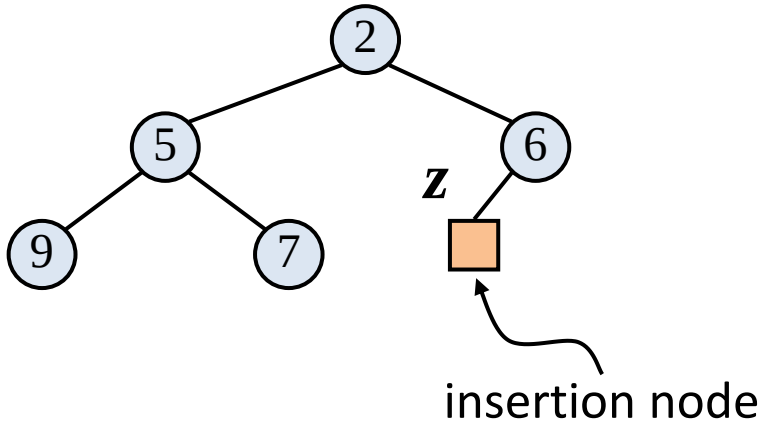
Heap e Code a Priorità

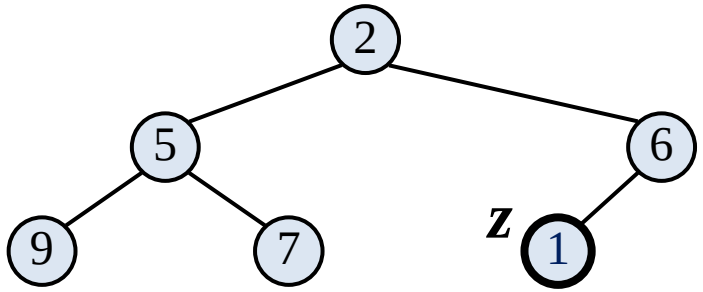
- Possiamo usare un heap per implementare una PQ
- Conserviamo (key, element) in ogni nodo
- Serve mantenere traccia dell'ultimo nodo
- Spesso per semplicità, indichiamo solo le key nei nodi



Inserimento in un Heap

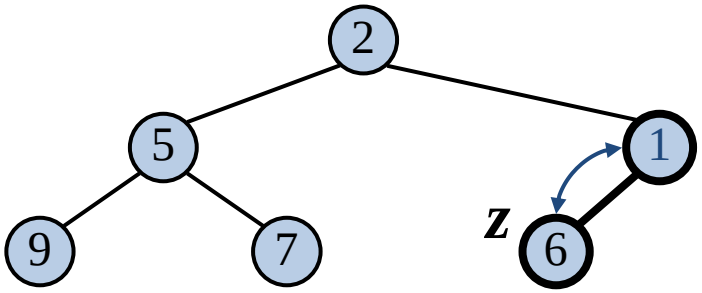
- Il metodo Insert della PQ corrisponderà all'inserimento di una key k nell'heap.
- L'algoritmo di inserimento è in 3 passi:
  - Trova il nodo per l'inserimento z (il nuovo **ultimo** nodo)
  - Conserva k in z
  - Ripristina la proprietà dell'heap-order

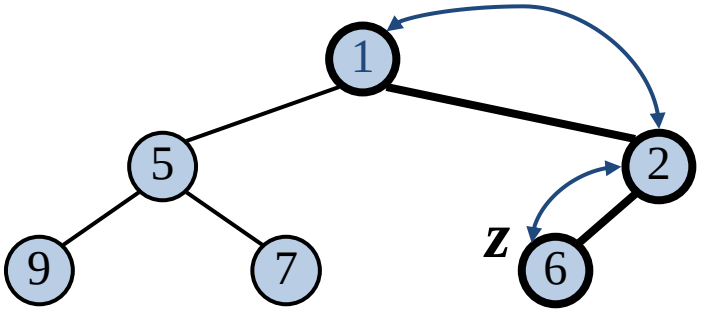




Heapify-up

- Infatti, dopo l'inserimento della nuova key ***k*** nell'ultimo nodo dell'heap, la proprietà dell'heap-order potrebbe essere violate fra ***z*** e i genitori
- L'algoritmo Heapify-up ripristina l'ordine delle key, effettuando degli *swap* tra il nodo che contiene ***k*** e i nodi lungo il cammino che sale verso la root.
- Heapify-up termina quando la key ***k*** raggiunge la radice oppure un nodo il cui parent ha una key minore o uguale a ***k.***
- Correttezza per induzione su i tale che ***z***=H[i]
- Poichè un heap ha altezza ***O***(log ***n***), Heapify-up ha complessità ***O***(log ***n***).

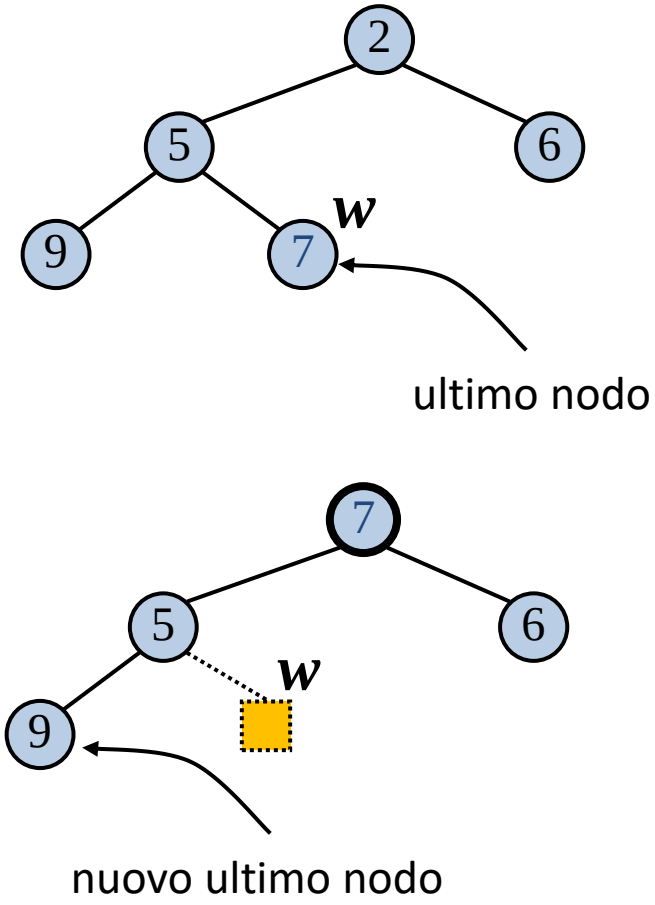




Cancellazione del minimo

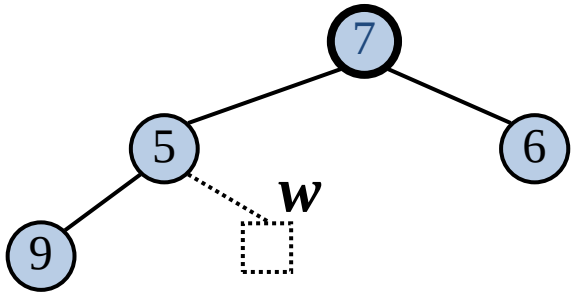
- Il metodo ExtractMin della PQ corrisponde alla rimozione della key conservata nella radice dell'heap.

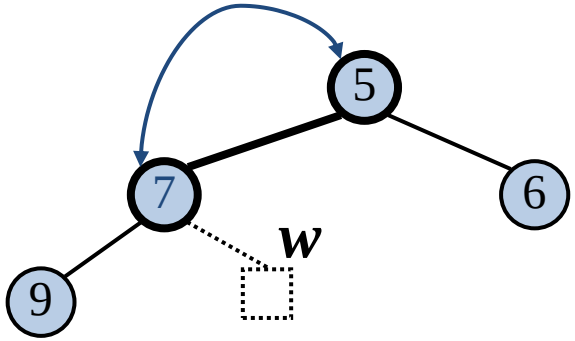
- L'algoritmo consiste di 3 passi:
  - Sostituisci la key della root con la key dell'ultimo nodo w
  - Cancella w
  - Ripristina la proprietà dell'heap-order



Heapify-down

- Dopo la sostituzione della key conservata nella root con la key k dell'ultimo nodo, l'ordine delle key potrebbe essere stato violato fra la radice e i figli.
- Più in generale, l'algoritmo Heapify-down chiamato su un nodo z ripristina la proprietà dell'heap-order effettuando lo *swap* tra il nodo z che contiene la key k e il figlio (dei due) che contiene una chiave minore, lungo il cammino dalla radice fino alle foglie (down).
- Heapify-down termina quando la key k raggiunge una foglia oppure un nodo i cui figli hanno entrambi key maggiori o uguali a k.





Heapify-down

- Correttezza:

L'algoritmo Heapify-down, chiamato su un nodo z, ripristina la proprietà dell'heap-order che potrebbe essere violata fra il nodo z e i figli.

Più precisamente: se il sotto-albero sinistro e il sotto-albero destro di z sono degli heap, dopo la chiamata di Heapify-down su z, l'intero albero radicato in z sarà un heap.

Dimostrazione per induzione inversa su i tale che z=H[i].

- Complessità:

la chiamata di Heapify-down su z, ha complessità di tempo $O(h(z))$, dove $h(z)$ è l'altezza del nodo z.

Se z è la radice, Heapify-down ha complessità $O(\log n)$, dato che un heap ha altezza $O(\log n)$.

Cancellazione di un qualsiasi nodo

Cancellazione di un nodo v in un heap H.

L'algoritmo consiste di 3 passi:

- Sostituisci la key del nodo v con la key dell'ultimo nodo w
- Cancella w
- Ripristina la proprietà dell'heap-order con Heapify-up se la proprietà è violata coi genitori, con Heapify-down se la proprietà è violata coi figli

La complessità è $O(\log n)$.

Decremento chiave

Se volessi modificare la chiave di un nodo:

- Se il nuovo valore è maggiore allora posso modificare la chiave e ripristinare l'heap-order con Heapify-down
- Se il nuovo valore è minore (decremento chiave) allora conviene cancellare il nodo e inserirne uno nuovo con la nuova chiave.
- La complessità è $O(\log n)$.

Costruzione di un heap

Data una sequenza S di elementi posso costruire un heap A con gli elementi di S, inserendo ogni elemento di S in un heap inizialmente vuoto con la procedura `INSERT`, in tempo totale O(n log n).

Inoltre, dato un array A[1..n] è possibile convertire A in un min-heap in tempo O(n) col seguente algoritmo (ricorda le proprietà di un heap).

```
BUILD-MIN-HEAP(A)
1 heap-size[A] ← length[A]
2 for i ← ⌊length[A]/2⌋ downto 1
3     do Heapify-down(A, i)
```

Costruzione di un heap

Inoltre, dato un array A[1..n] è possibile convertire A in un min-heap in tempo O(n) col seguente algoritmo (ricorda le proprietà di un heap).

```
BUILD-MIN-HEAP(A)
1 heap-size[A] ← length[A]
2 for i ← ⌊length[A]/2⌋ downto 1
3       do Heapify-down(A, i)
```

$$T(n) = \sum_{i=1}^{n/2} h(i) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

Utilizzando la formula $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

Applicazioni: ordinare con una PQ

Si può usare una coda a priorità per ordinare una sequenza S di elementi confrontabili.

L'algoritmo PQ-Sort ha due fasi:

1. inserisci un elemento alla volta nella coda a priorità P con Insert [key è l'elemento, value è nullo]
2. rimuovi gli elementi uno alla volta da P con ExtractMin e li reinserisci nella sequenza S

- La complessità di tempo di questo algoritmo dipende da come è implementata la PriorityQueue.

  Vedremo 3 specializzazioni

PQ-Sort con liste non ordinate

	***Sequence** S*	***Priority Queue** P (lista non ordinata)*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	((7,∅))
(b)	(8,2,5,3,9)	((7,∅),(4,∅))
..	..	..
(g)	()	((7,∅),(4,∅),(8,∅),(2,∅),(5,∅),(3,∅),(9,∅))
Phase 2		
(a)	(2)	((7,∅),(4,∅),(8,∅),(5,∅),(3,∅),(9,∅))
(b)	(2,3)	((7,∅),(4,∅),(8,∅),(5,∅),(9,∅))
(c)	(2,3,4)	((7,∅),(8,∅),(5,∅),(9,∅))
(d)	(2,3,4,5)	((7,∅),(8,∅),(9,∅))
(e)	(2,3,4,5,7)	((8,∅),(9,∅))
(f)	(2,3,4,5,7,8)	((9,∅))
(g)	**(2,3,4,5,7,8,9)**	()

Quale algoritmo ho ottenuto?

SelectionSort

PQ-Sort con liste ordinate

	Sequence *S*	***Priority queue*** *P* (*lista ordinata*)
Input:	(7,4,8,2,5,3,9)	( )
Phase 1		
(a)	(4,8,2,5,3,9)	((7,∅))
(b)	(8,2,5,3,9)	((4, ∅),(7,∅))
(c)	(2,5,3,9)	((4, ∅),(7,∅),(8,∅))
(d)	(5,3,9)	((2,∅),(4, ∅),(7,∅),(8,∅))
(e)	(3,9)	((2,∅),(4, ∅),(5,∅),(7,∅),(8,∅))
(f)	(9)	((2,∅),(3,∅),(4,∅),(5,∅),(7,∅),(8,∅))
(g)	()	((2,∅),(3,∅),(4,∅),(5,∅),(7,∅),(8,∅),(9,∅))
Phase 2		
(a)	(2)	((3,∅),(4,∅),(5,∅),(7,∅),(8,∅),(9,∅))
(b)	(2,3)	((4,∅),(5,∅),(7,∅),(8,∅),(9,∅))
..	..	..
(g)	**(2,3,4,5,7,8,9)**	( )

Quale algoritmo ho ottenuto?

InsertionSort

SelectionSort e InsertionSort

- Se usiamo liste **non ordinate**, si ottiene il **SelectionSort**
  Fase 1: n Insert in tempo O(n)
  Fase 2: n ExtractMin in tempo $O(n^2)$
  [lento nella seconda fase]

- Se usiamo liste **ordinate**, si ottiene **l'InsertionSort**
  Fase 1: n Insert in tempo $O(n^2)$
  Fase 2: n ExtractMin in tempo O(n)
  [lento nella prima fase]

Per entrambi il tempo di esecuzione è $O(n^2)$

Possiamo fare meglio?

- Bilanciare i tempi delle due fasi potrebbe migliorare il tempo di esecuzione di tutto l'algoritmo.

- Utilizzando la struttura dati *Heap* che consente di implementare una PQ in modo che inserimenti e cancellazioni possono essere eseguiti in tempo *logaritmico*.

- Con questa struttura dati otterremo un algoritmo di ordinamento HeapSort con complessità di tempo migliore:

  O(n log n)

HeapSort

- Usando una PQ implementata con heap, possiamo ordinare una sequenza di ***n*** elementi in tempo ***O***(***n*** log ***n***):
  - Ogni elemento della sequenza viene inserito in un **heap inizialmente vuoto**: n operazioni di Insert in tempo O(n log n)
  - L'heap viene svuotato: n operazioni di ExtractMin in tempo O(n log n)
- L'algoritmo risultante è chiamato HeapSort
- L'unica differenza con SelectionSort e InsertionSort è nell'implementazione della PQ
- Nota: il lower bound di Ω(n log n) per l'ordinamento (basato su confronti) ci dice che il bound di O(log n) è quello giusto per le operazioni di una coda a priorità

Animazione algoritmi su Heap

Potete trovarla su:

https://www.cs.usfca.edu/~galles/visualization/Heap.html

Esercizio

Sia H = [1, 6, 3, 12, 7, 9]. L'array H può rappresentare un *heap* binario (basato sulla priorità minima)?

A. No, perché gli elementi dovrebbero essere in ordine crescente

B. No, perché il massimo dovrebbe occupare l'ultima posizione

C. Sì

D. Nessuna delle risposte precedenti

Esercizio

a) **Definire** cosa è una coda a priorità

b) **Definire** cosa è un heap binario (basato sulla priorità minima)

c) Si supponga di voler mantenere un insieme V = {a, b, c, d, e, f } con una coda a priorità realizzata con un heap binario (basato sulla priorità minima) nelle ipotesi che le priorità degli elementi a, b, c, d, e, f, siano rispettivamente 6, 12, 1, 7, 9, 3.

c1) Si consideri l' heap binario H ottenuto inserendo con la procedura Insert studiata, in un heap inizialmente vuoto gli elementi a, b, c, d, e, f, nell'ordine. E' sufficiente rappresentare l'heap finale (in una delle due rappresentazioni studiate).

c2) Si esegua adesso la procedura ExtractMin studiata su H, mostrando gli aggiornamenti ad ogni passaggio.

c3) si continui fino a completare HeapSort di V

In un grafo connesso con n vertici ed m archi:

A. $\log m = \Theta(\log n)$

B. $\log m = O(\log n)$, ma non $\log m = \Theta(\log n)$

C. $\log n = O(\log m)$, ma non $\log n = \Theta(\log m)$

D. Nessuna delle risposte precedenti

Sia G=(V,E) un grafo con V={1,2,3,4,5,6,7} e E={(1,2), (1,3), (1,4), (2,5), (2,6), (3,6), (4,5), (5,7), (6,7)}. L'albero della visita in ampiezza **BFS** del grafo G=(V,E), a partire dal vertice 1, in cui i vertici adiacenti sono analizzati in ordine crescente, ha profondità

A. 3 B. 4 C. 5 D. Nessuna delle risposte precedenti

Sia G=(V,E) un grafo con V={1,2,3,4,5,6,7} e E={(1,2), (1,4), (2,4), (2,5), (3,6), (4,5)}. In seguito alla visita in ampiezza **BFS** del grafo G=(V,E), a partire dal vertice 1, quanti saranno i vertici scoperti (*Discovered*) (compreso 1)?

A. 6 B. 4 C. 5 D. Nessuna delle risposte precedenti

Se G=(V,E) è un grafo rappresentato con una matrice di adiacenza, verificare se (u,v) è un arco di E, richiede tempo

A. $\Theta(1)$ B. $\Theta(n)$ C. $\Theta(n^2)$ D. Nessuna delle risposte precedenti

Un ordinamento topologico per il grafo diretto G=(V,E) con V={u, v, x, y, z}, E={(u,x), (v,x), (v,y), (v,u), (x,y), (y,z)} è:

A. v, u, z, y, x

B. v, u, x, y, z

C. G non ha un ordinamento topologico

D. Nessuna delle risposte precedenti

Sia G=(V,E), il grafo in cui V={1,2,3,4,5,6,7} ed E={(1,2), (1,3), (1,4), (2,5), (2,6), (3,6), (3,7), (4,6), (4,7)}. Allora

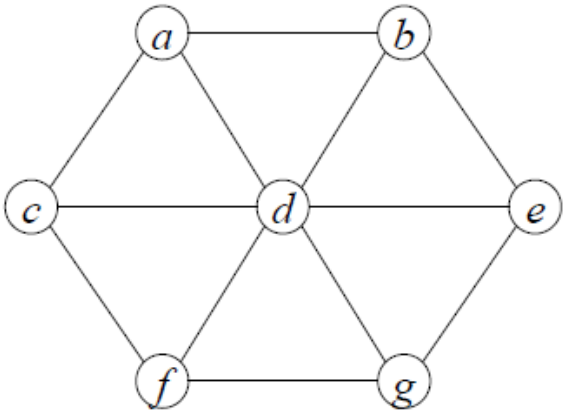
A. G non è bipartito perché non ha cicli di lunghezza dispari

B. G è bipartito perché contiene cicli di lunghezza dispari

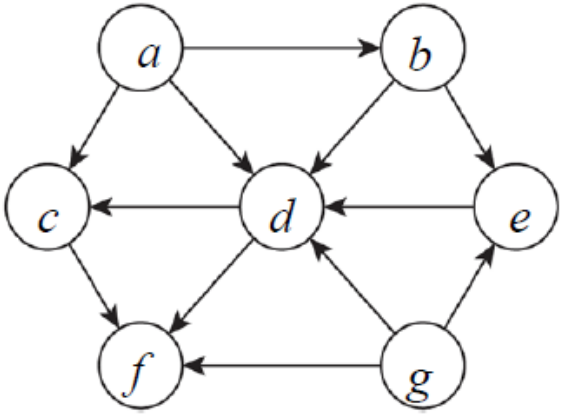
C. G è bipartito perché contiene cicli di lunghezza pari

D. Nessuna delle risposte precedenti

1)Descrivere **verbalmente** un algoritmo per decidere se un grafo **non orientato** è aciclico o no.
2) **Eseguire** l'algoritmo descritto sul grafo G1 in basso.
3)Descrivere **verbalmente** un algoritmo per decidere se un grafo **orientato** è aciclico o no.
4)Eseguire l'algoritmo descritto sul grafo G2 in basso.



G1



G2

Metodo *Greedy* (avido, goloso)

10 maggio 2022

Punto della situazione

Tecniche di progettazione di algoritmi:

- forza bruta, esaustivi, naif
- divide et impera;
- programmazione dinamica;
- tecnica greedy
- Algoritmi esaustivi intelligenti: backtracking e branch-and-bound

Esempi:

1. Selezione di intervalli (versione non pesata) ([KT] par. 4.1)
2. Partizionamento di intervalli ([KT] par. 4.1)
3. Scheduling che minimizza il ritardo ([KT] par. 4.2)
4. Codici di Huffman
5. e altri sui grafi: MST, cammini minimi, …

Tecnica greedy

- Serve per risolvere problemi di ottimizzazione = ricerca di una soluzione ammissibile che ottimizzi (max/min) un valore associato.

- Non sempre è possibile utilizzarla per ottenere una soluzione ottimale (nel caso, provare con la programmazione dinamica)

- Schema molto semplice

- Prova della correttezza… di meno

Schema algoritmo greedy

- Ordino secondo un criterio di convenienza
- Inizializzo SOL = insieme vuoto
- Considero ogni oggetto in ordine di convenienza
  - Se è compatibile con SOL lo aggiungo a SOL
- Restituisco SOL

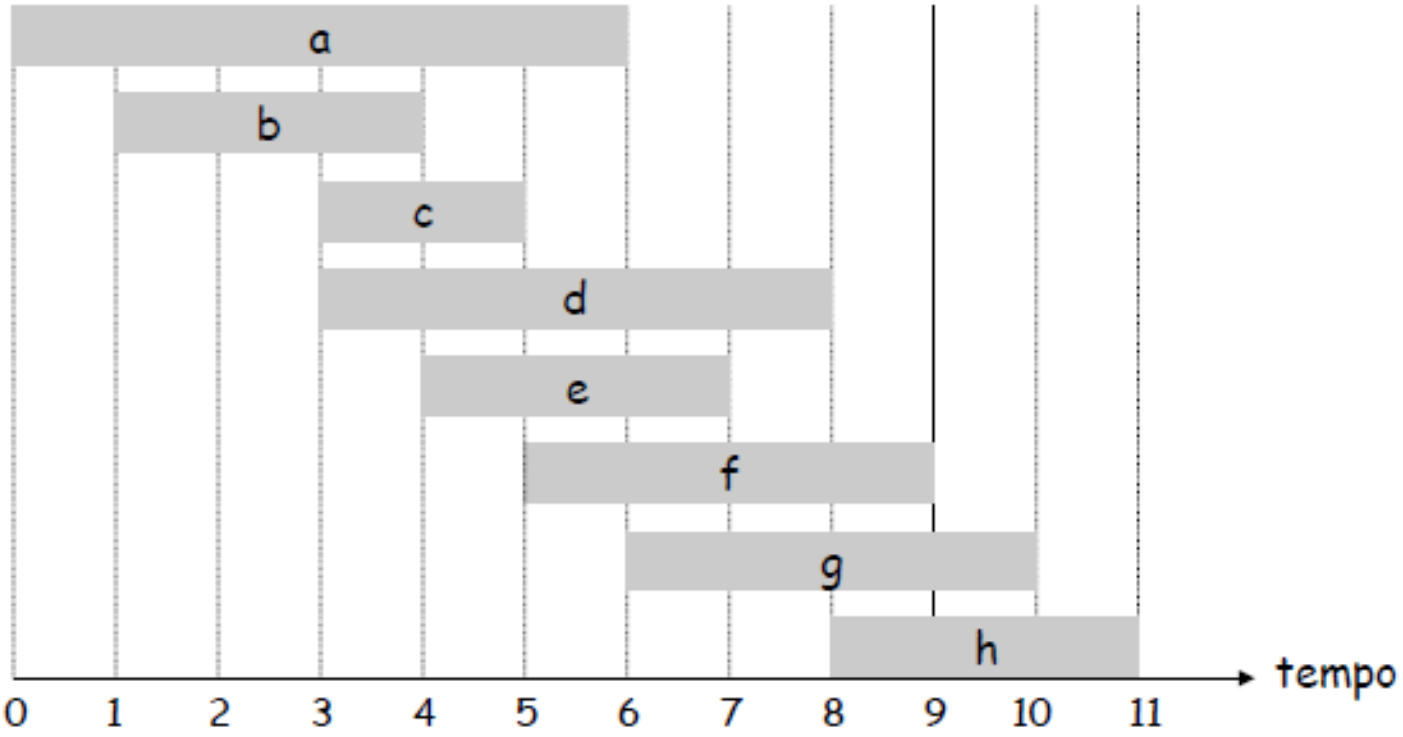
4.1 Interval Scheduling

(non pesato)

Schedulazione intervalli

Schedulazione intervalli.

- Job j inizia a s_j e finisce a f_j.
- Due job sono compatibili se hanno intersezione vuota.
- Obiettivo: trovare sottoinsieme massimale di job mutuamente compatibili.



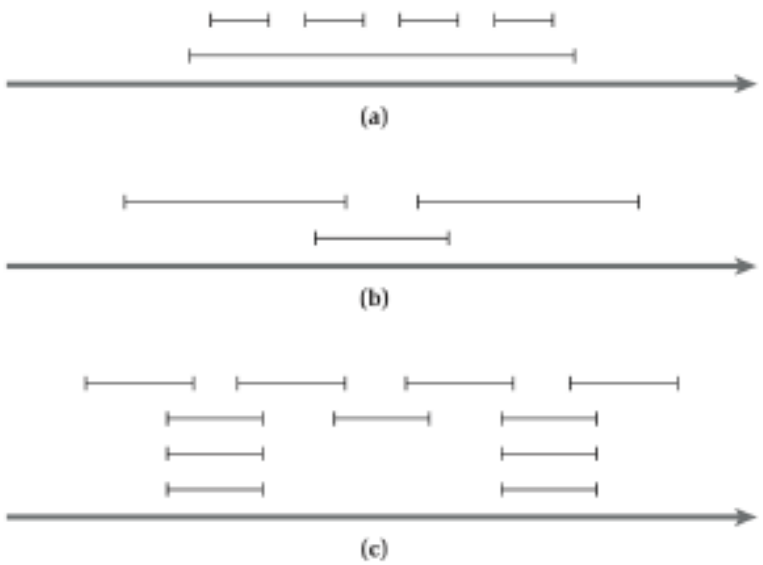
Schedulazione intervalli: Algoritmi Greedy

Scelta Greedy. Considera job in *qualche* ordine. Prendere job se è compatibile con quelli già presi.

- [tempo inizio minimo] Considera job in ordine crescente del tempo di inizio s_j.
- [tempo fine minimo] Considera job in ordine crescente del tempo di fine f_j.

Schedulazione intervalli: Algoritmi Greedy

Scelta Greedy. Considera job in *qualche* ordine. Prendere job se è compatibile con quelli già presi.



tempo inizio minimo

intervallo più corto

minori conflitti

non portano all'ottimo!

Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

Schedulazione intervalli: Algoritmo Greedy

Algoritmo Greedy. Considera job in ordine crescente del tempo di fine f_j. Prendere job se è compatibile con quelli già presi.

```
Ordina job per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.

   insieme job scelti
A ← {1}
for j = 2 to n {
   if (job j è compatibile con A)
      A ← A ∪ {j}
}
return A
```

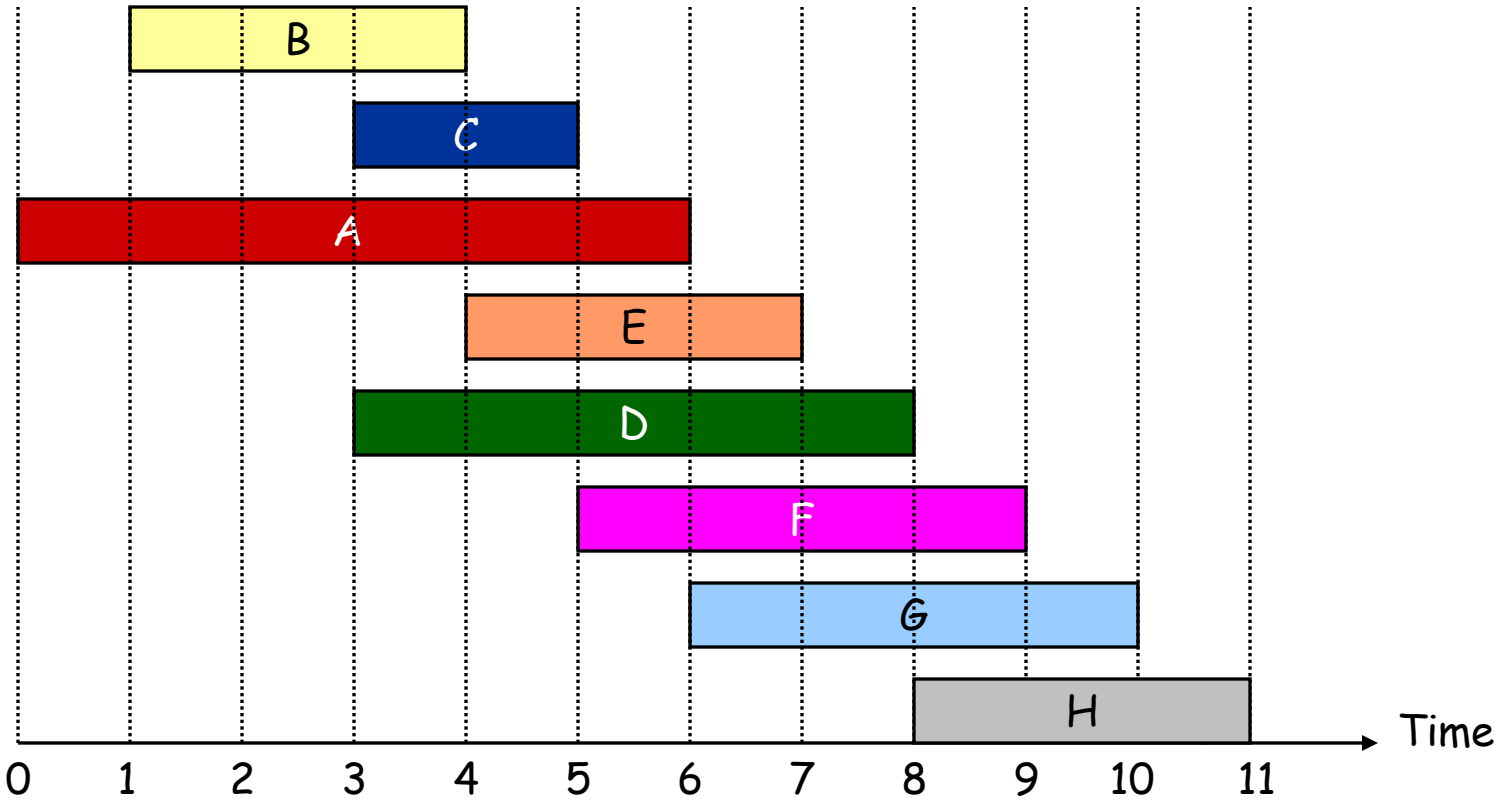
Schedulazione intervalli: Implementazione Algoritmo Greedy

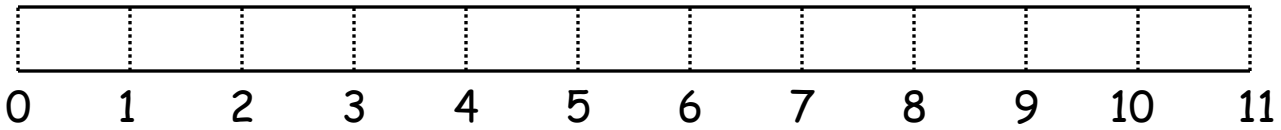
Implementazione Algoritmo Greedy. $O(n \log n)$.

- Denota j^* l'ultimo job aggiunto ad A.
- Job j è compatibile con A se $s_j \geq f_{j^*}$.

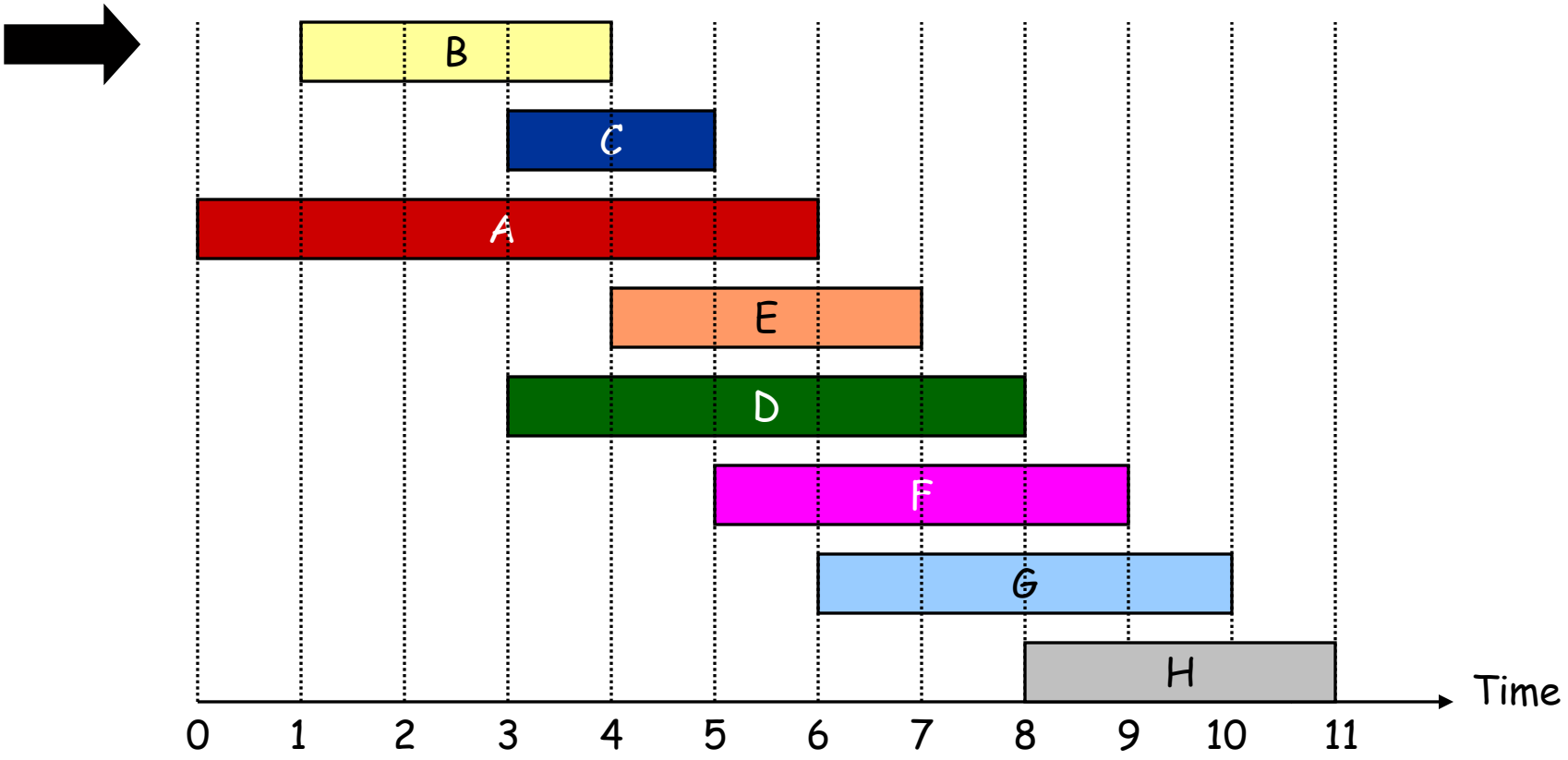
```
Ordina job per tempo di fine f1 ≤ f2 ≤ ... ≤ fn.

A ← {1}
j* = 1
for j = 2 to n {
   if sj ≥ fj*
      {A ← A ∪ {j}
       j* = j
      }
}
return A
```

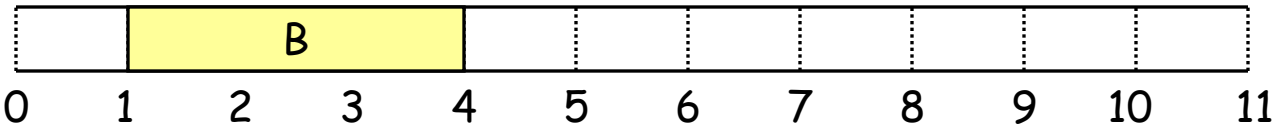
Interval Scheduling



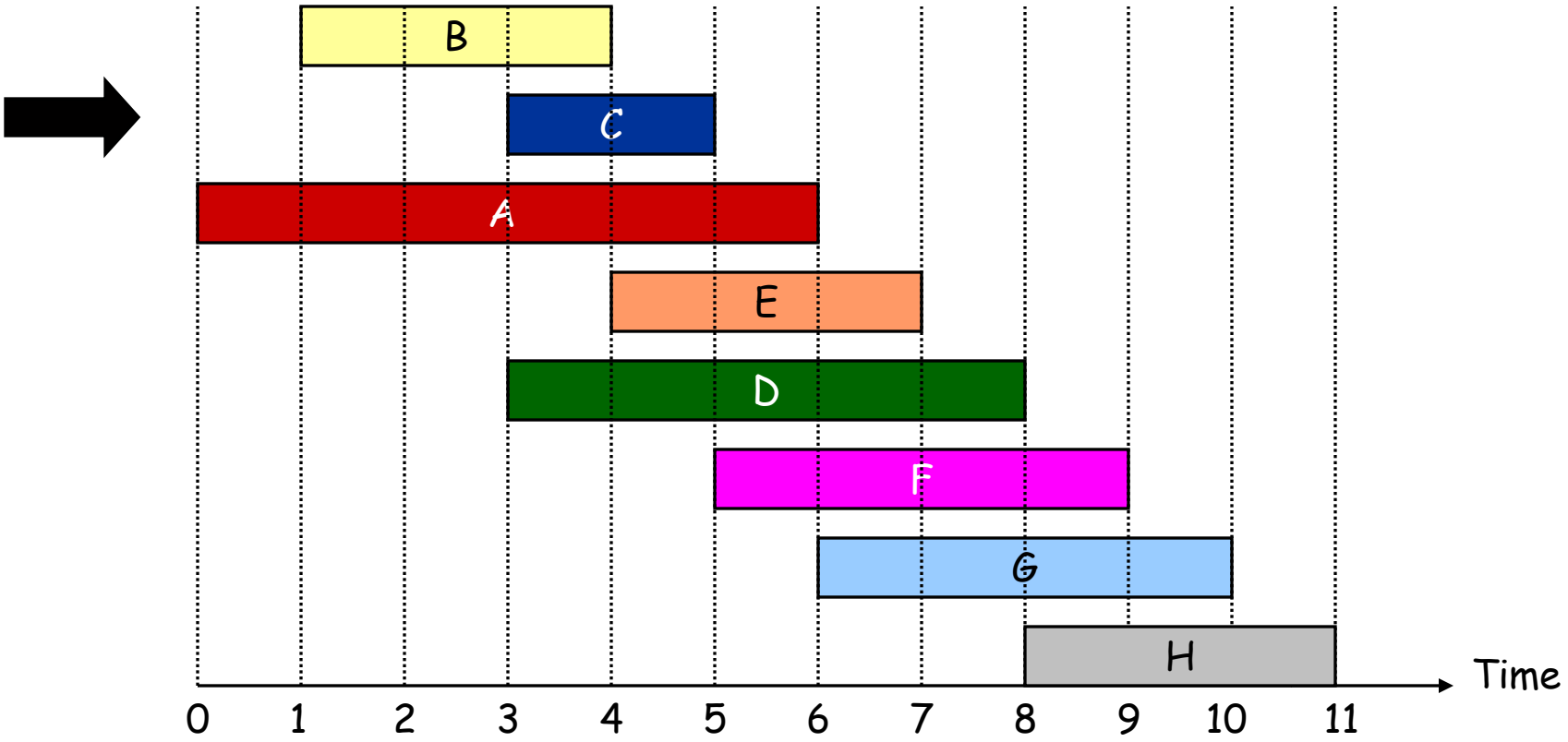


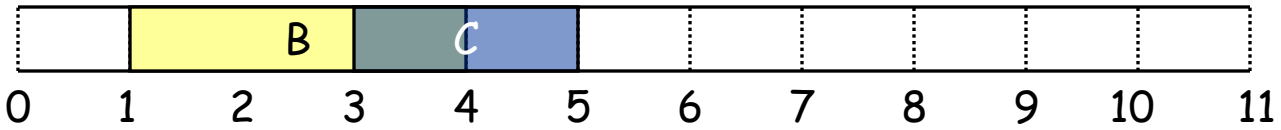
Interval Scheduling



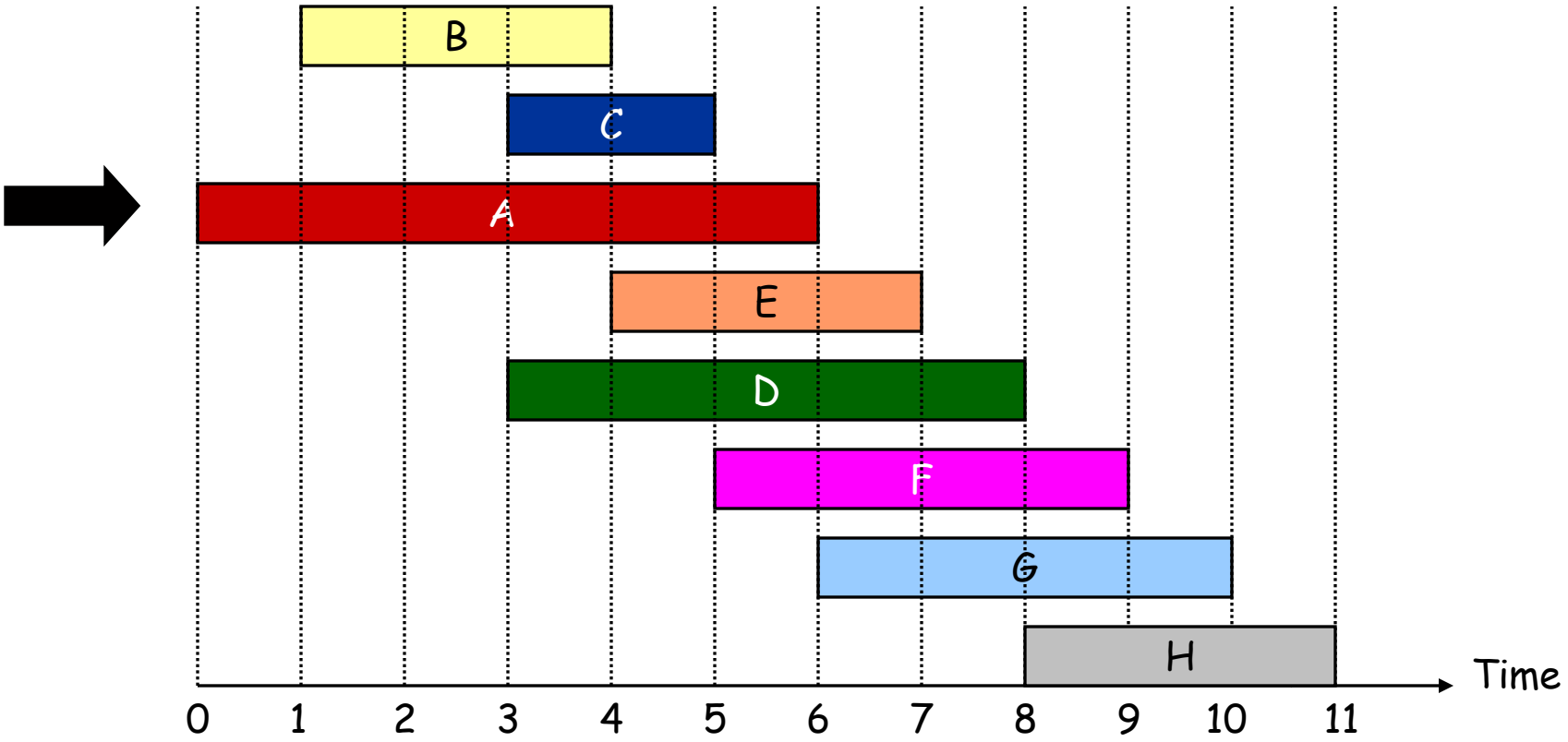


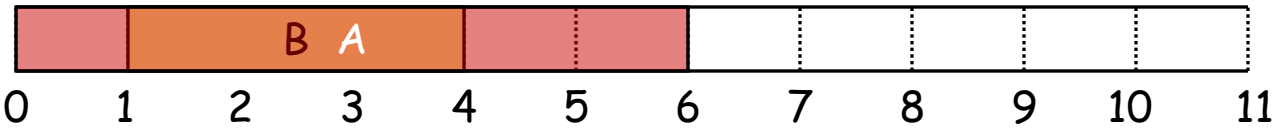
Interval Scheduling



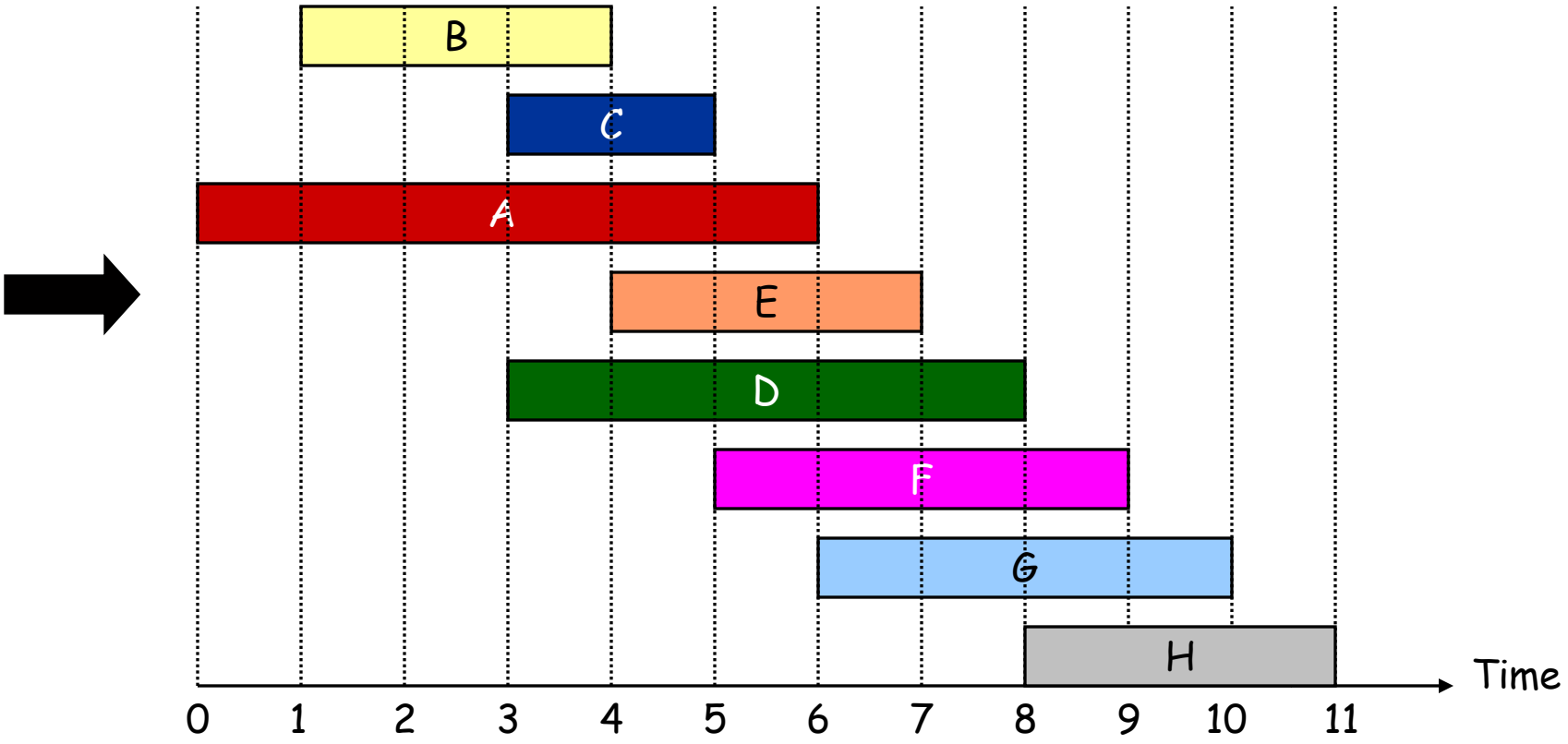


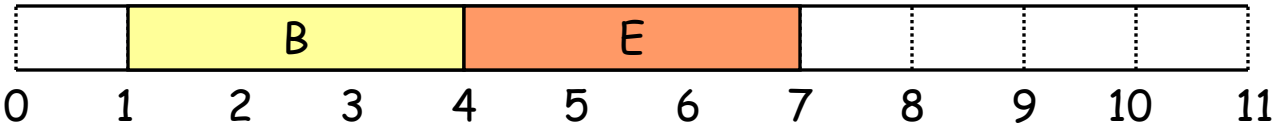
Interval Scheduling



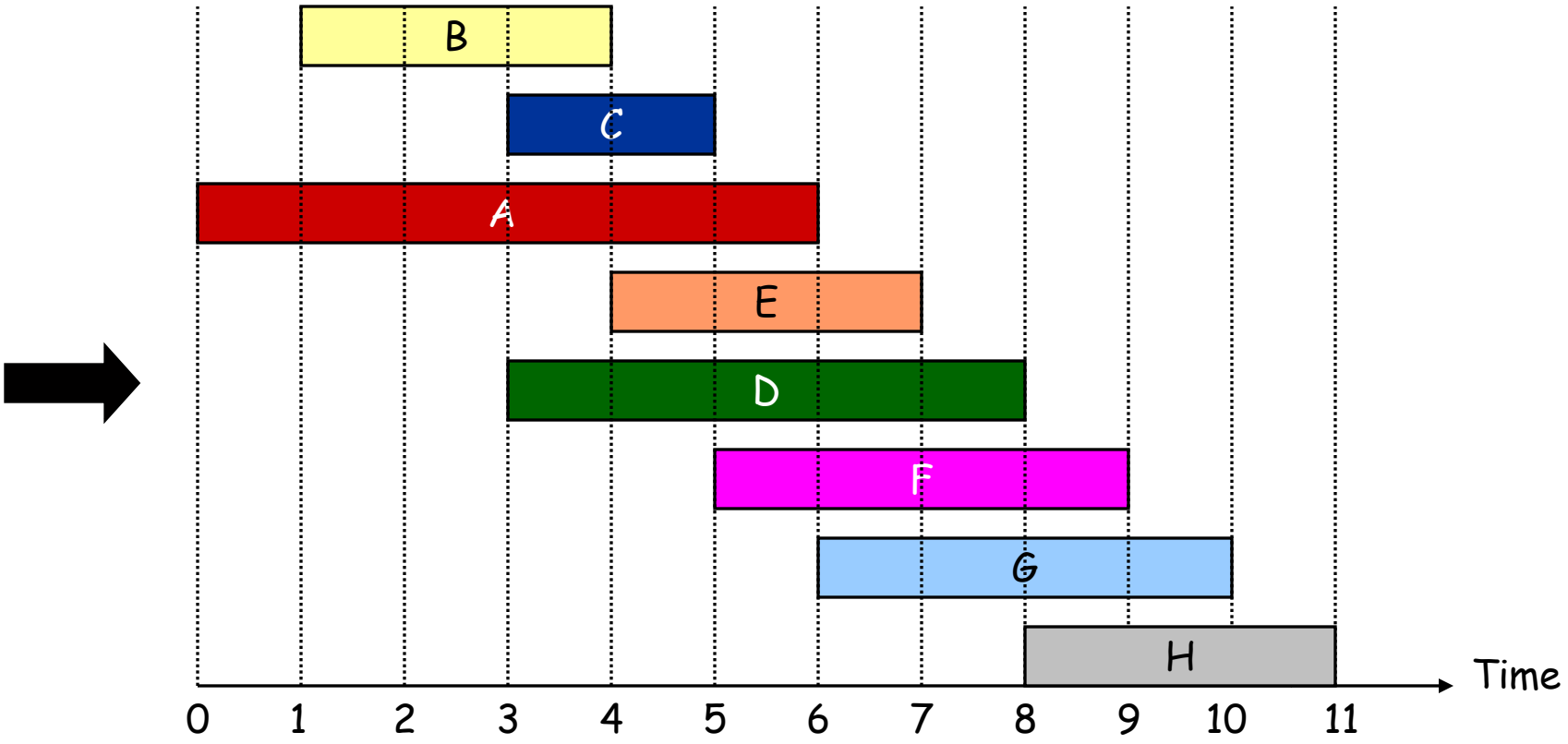


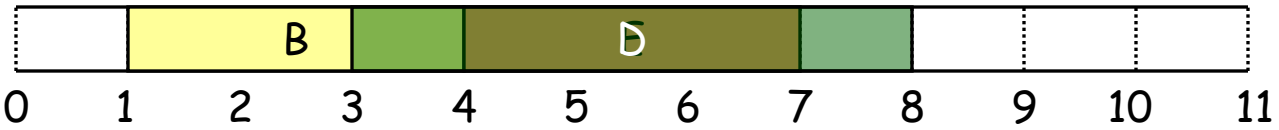
Interval Scheduling



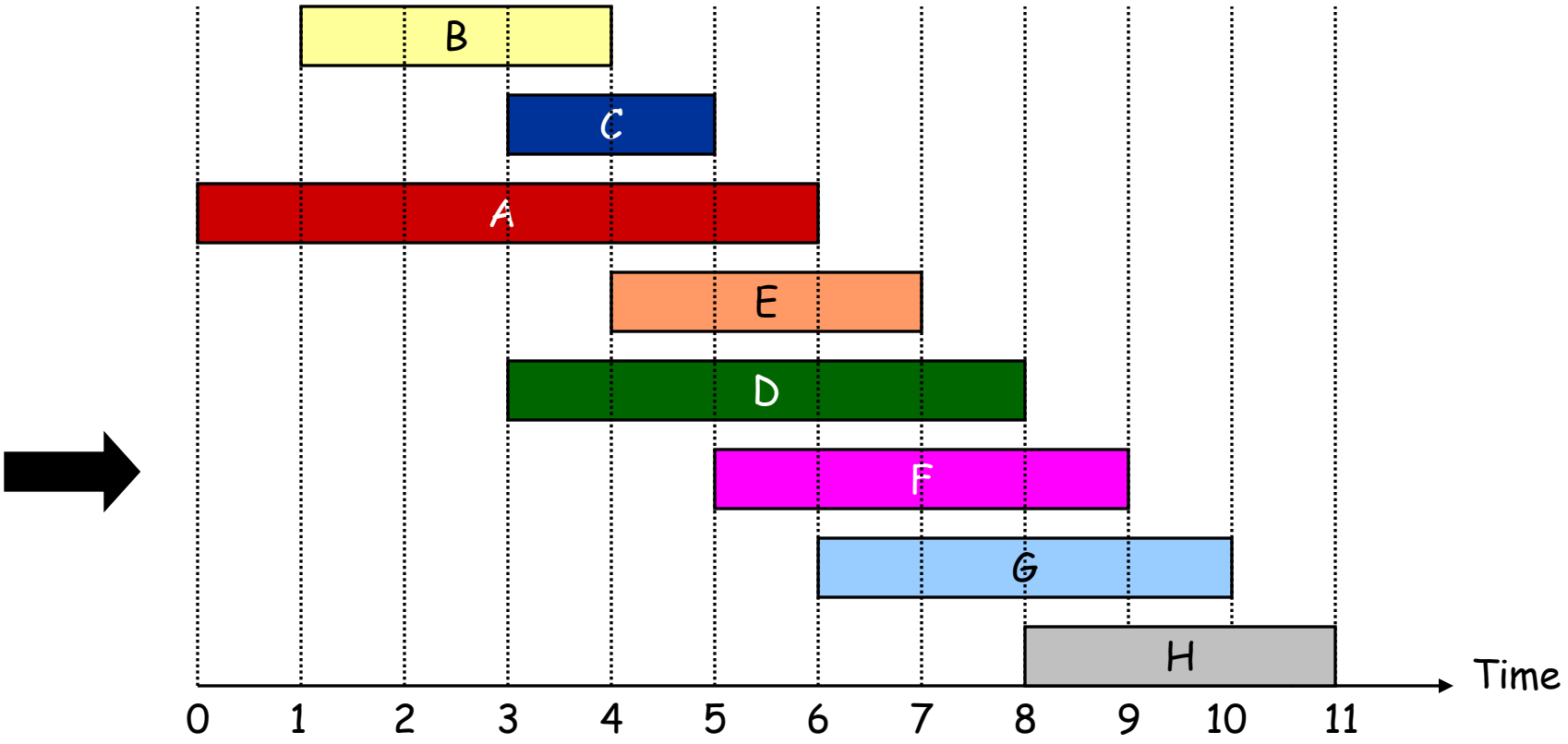


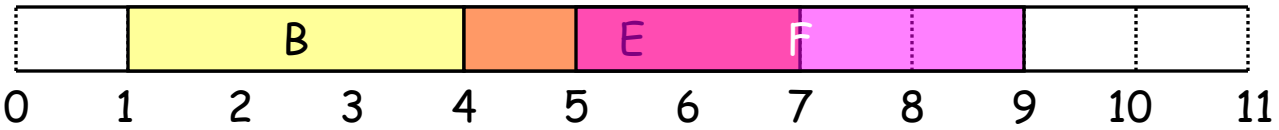
Interval Scheduling



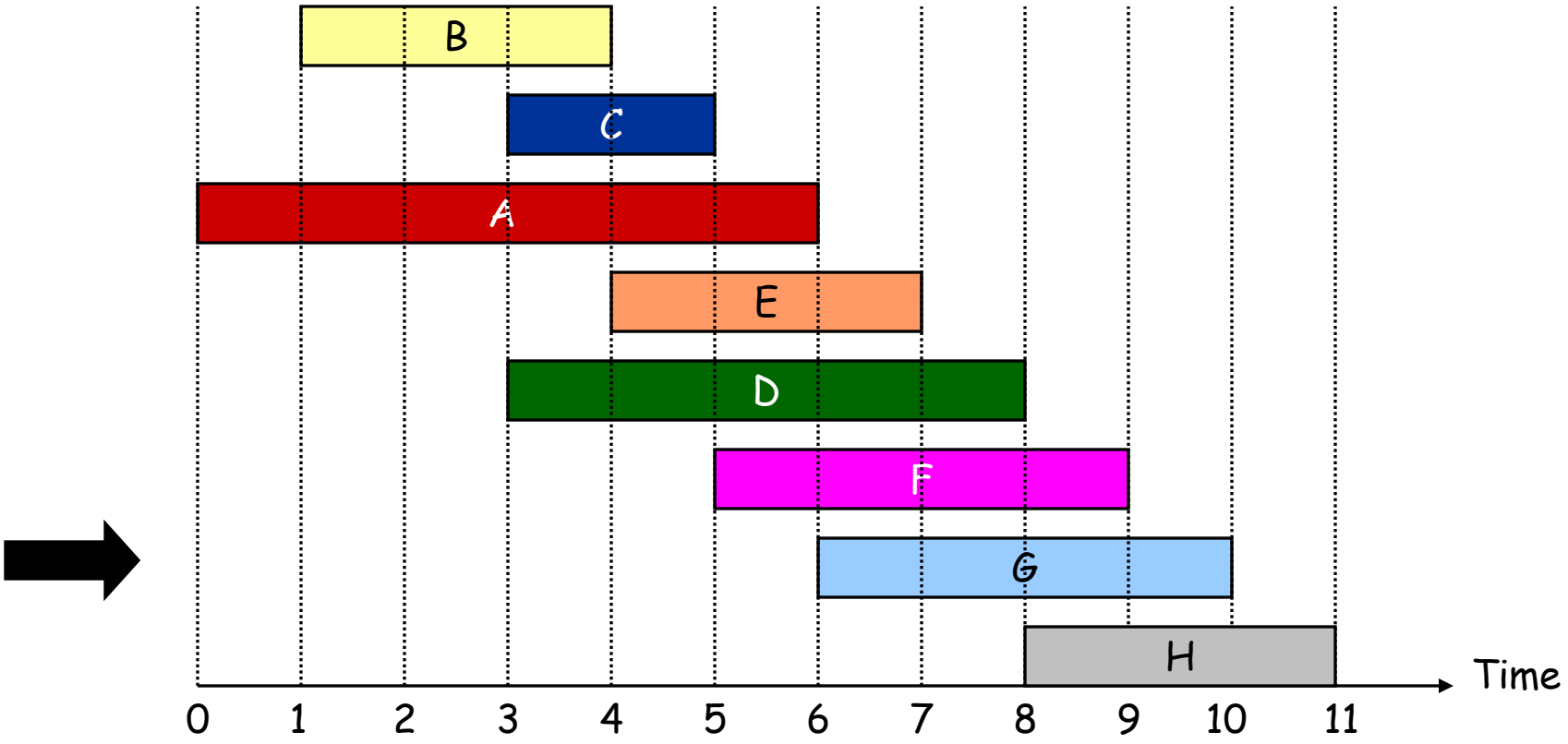


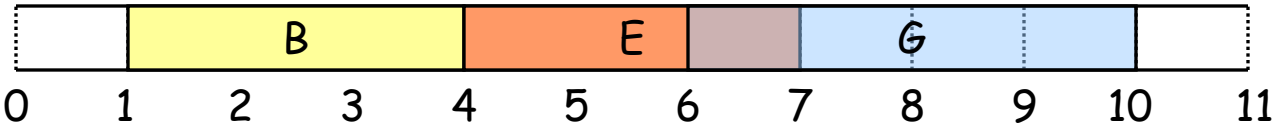
Interval Scheduling





Interval Scheduling





Interval Scheduling