Tecniche Algoritmiche

Damiano Macedonio
Università Ca' Foscari di Venezia

mace@unive.it

Introduzione

- Dato un problema
  - Non ci sono “ricette generali” per risolverlo in modo efficiente
- Tuttavia, è possibile evidenziare quattro fasi
  1. Classificazione del problema
  2. Caratterizzazione della soluzione
  3. Tecnica di progetto
  4. Utilizzo di strutture dati

Non sono fasi strettamente sequenziali!

Classificazione di un problema

Fa parte di una classe più ampia di problemi?

- Problemi decisionali
  - Il dato di ingresso soddisfa una certa proprietà?
  - Soluzione: risposta sì/no
  - Es: Stabilire se un grafo è connesso
- Problemi di ricerca
  - Spazio di ricerca: insieme di “soluzioni” possibili
  - Soluzione ammissibile: soluzione che rispetta certi vincoli
  - Es: posizione di una chiave in un dizionario

Introduzione

- Problemi di ottimizzazione
  - Ogni soluzione è associata ad una funzione di costo
  - Vogliamo trovare la soluzione di costo minimo
  - Es: cammino minimo fra due nodi
- Problemi di approssimazione
  - A volte, trovare la soluzione ottima è computazionalmente impossibile
  - Ci si accontenta di una soluzione approssimata:
    - costo basso, ma non sappiamo se ottimo
  - Es: problema del commesso viaggiatore

Caratterizzazione della soluzione

- Definire la soluzione dal punto di vista matematico
  - Spesso la formulazione è banale...
  - ... ma può suggerire una prima idea di soluzione
    - Es: Selection Sort: *"Data una sequenza di n elementi, una permutazione ordinata è data dall'elemento minimo seguito da una permutazione ordinata dei restanti n-1 elementi"*
- Le caratteristiche formali della soluzione possono suggerire una possibile tecnica algoritmica (Es. il Teorema Fondamentale degli MST)

Tecniche di progetto / 1

- Divide-et-impera
  - Un problema viene suddiviso in sotto-problemi *indipendenti*, che vengono risolti ricorsivamente (top-down)
  - Ambito: problemi di decisione, ricerca
- Programmazione dinamica
  - La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi *potenzialmente ripetuti*
  - Ambito: problemi di ottimizzazione

Tecniche di progetto / 2

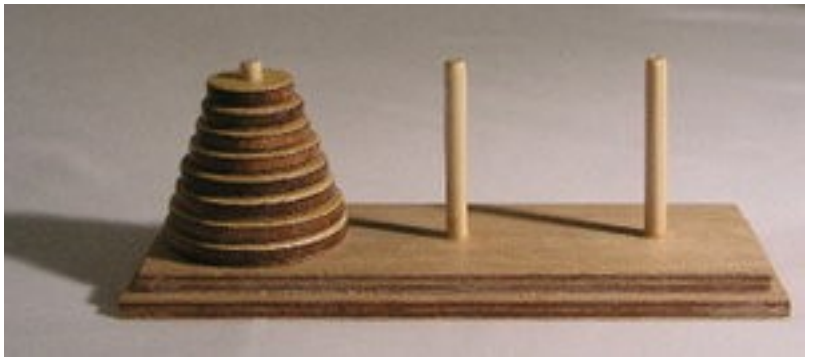
- Tecniche greedy (algoritmi “golosi”)
  - Ad ogni passo si fa sempre la scelta localmente ottima
- Backtrack
  - Procediamo per “tentativi”, tornando ogni tanto sui nostri passi
- Ricerca locale
  - La soluzione ottima viene trovata “migliorando” soluzioni esistenti, ma non ottime

divide-et-impera

Divide-et-impera

- Tre fasi:
  - *Divide*: Dividi il problema in sotto-problemi più piccoli e *indipendenti*
  - *Impera*: Risolvi i sotto-problemi ricorsivamente
  - *Combina*: “unisci” le soluzioni dei sottoproblemi
- Non esiste una ricetta “unica” per divide-et-impera:
  - Quick Sort: “divide” complesso, niente fase di “combina”
  - Merge Sort: “divide” banale, “combina” complesso
- È necessario uno sforzo creativo

Le torri di Hanoi



http://en.wikipedia.org/wiki/Tower_of_Hanoi

- Gioco matematico
  - tre pioli
  - *n* dischi di dimensioni diverse
  - Inizialmente tutti i dischi sono impilati in ordine decrescente (più piccolo in alto) nel piolo di sinistra
- Scopo del gioco
  - Impilare in ordine decrescente i dischi sul piolo di destra
  - Senza mai impilare un disco più grande su uno più piccolo
  - Muovendo al massimo un disco alla volta
  - Utilizzando se serve anche il piolo centrale

Le torri di Hanoi
Soluzione divide-et-impera

Sposta n dischi da p1 a p3
usando p2 come appoggio

http://en.wikipedia.org/wiki/Tower_of_Hanoi

```
hanoi(Stack p1, Stack p2, Stack p3, int n)
  if (n == 1) then
    p3.push(p1.pop())
  else
    hanoi(p1, p3, p2, n-1)
    p3.push(p1.pop())
    hanoi(p2, p1, p3, n-1)
  endif
```

- Divide:
  - n-1 dischi da p1 a p2
  - 1 disco da p1 a p3
  - n-1 dischi da p2 a p3
- Impera
  - Esegui ricorsivamente gli spostamenti

Le torri di Hanoi
Soluzione divide-et-impera

```
hanoi(Stack p1, Stack p2, Stack p3, int n)
  if (n = 1) then
    p3.push(p1.pop())
  else
    hanoi(p1, p3, p2, n-1)
    p3.push(p1.pop())
    hanoi(p2, p1, p3, n-1)
  endif
```

- Costo computazionale:
  - T(1) = 1
  - T(n) = 2T(n-1)+1 per n>1
- **Domanda:** Quale è la soluzione della ricorrenza?

Metodo divide-et-impera

- Quando applicare divide-et-impera
  - I passi “divide” e “combina” devono essere semplici
  - Ovviamente, i costi devono essere migliori del corrispondente algoritmo iterativo
  - Esempio ok: sorting
  - Esempio non ok: ricerca del minimo

programmazione dinamica

Programmazione dinamica

- Definizione
  - Strategia di programmazione sviluppata negli anni '50 da Richard E. Bellman
  - Ambito: problemi di ottimizzazione
  - Trovare la soluzione ottima secondo un “indice di qualità” assegnato ad ognuna delle soluzioni possibili
- Approccio
  - Risolvere un problema combinando le soluzioni di sotto-problemi
  - Ma ci sono importanti differenze con divide-et-impera

Richard Ernest Bellman (1920—1984), http://en.wikipedia.org/wiki/Richard_Bellman

Programmazione dinamica vs divide-et-impera

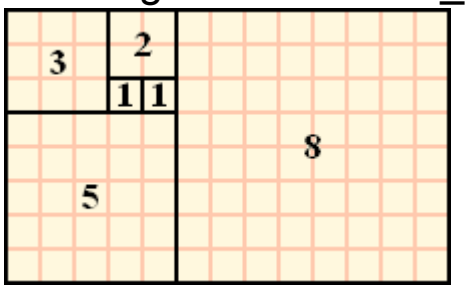
- Divide-et-impera
  - Tecnica ricorsiva
  - Approccio top-down (problemi divisi in sottoproblemi)
  - Vantaggioso solo quando i sottoproblemi sono indipendenti
    - Altrimenti gli stessi sottoproblemi possono venire risolti più volte
- Programmazione dinamica
  - Tecnica iterativa
  - Approccio bottom-up
  - Vantaggiosa quando ci sono sottoproblemi in comune
- Esempio semplice: i numeri di Fibonacci

Esempio:

Numero di Fibonacci (ancora loro!)

Numeri di Fibonacci

http://en.wikipedia.org/wiki/Fibonacci_number



- Definiti ricorsivamente
  - *F(0) = F(1) = 1*
  - *F(n) = F(n-2)+F(n-1)*
- In natura:
  - Pigne, conchiglie, parte centrale dei girasoli, etc.
- In informatica:
  - Alberi AVL minimi, Heap di Fibonacci, etc.

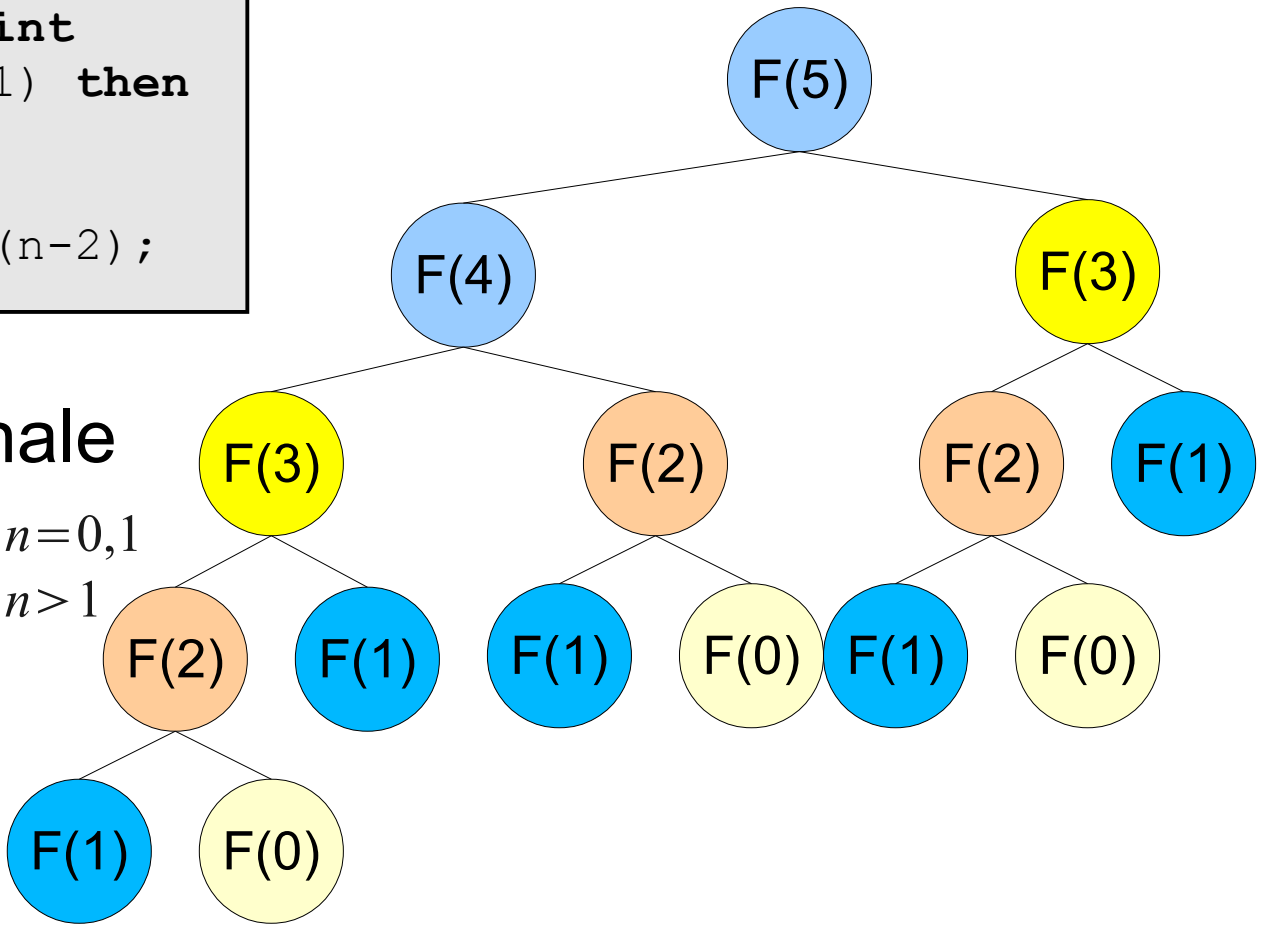
Implementazione ricorsiva

```
algoritmo Fib(int n) → int
  if (n == 0) or (n == 1) then
    return 1;
  else
    return Fib(n-1)+Fib(n-2);
```

- Costo computazionale

$$T(n)=\begin{cases} O(1) & n=0,1 \\ T(n-1)+T(n-2)+O(1) & n>1 \end{cases}$$

- Soluzione
  - $T(n) = O(2^n)$



Implementazione iterativa

```
algoritmo Fib(int n) → int
  f := new array[0..n] of int
  f[0] := 1;
  f[1] := 1;
  for i := 2 to n do
    f[i] := f[i-1] + f[i-2];
  endfor
  return f[n];
```

- Complessità
  - Tempo: $\Theta(n)$
  - Spazio: $\Theta(n)$

n	0	1	2	3	4	5	6	7
f[]	1	1	2	3	5	8	13	21

Implementazione iterativa - 2

```
algoritmo Fib(int n) → int
  if (n<2) then
    return 1;
  else
    f := new array[0..2] of int;
    f[1] := 1; f[2] := 1;
    for i := 2 to n do
      f[0] := f[1];
      f[1] := f[2];
      f[2] := f[1] + f[0];
    endfor
    return f[2];
  endif
```

- Complessità
  - Tempo: $\Theta(n)$
  - Spazio: $O(1)$
    - Array di *3* elementi

n	2	3	4	5
f[0]	1	1	2	3
f[1]	1	2	3	5
f[2]	2	3	5	8

Quando applicare la programmazione dinamica?

- Sottostruttura ottima
  - È possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più grande
    - PS: In tempo polinomiale!
  - Le decisioni prese per risolvere un problema rimangono valide quando esso diviene un sottoproblema di un problema più grande
- Sottoproblemi ripetuti
  - Un sottoproblema può occorrere più volte
- Spazio dei sottoproblemi
  - Deve essere polinomiale

Programmazione dinamica

1) Caratterizzare la struttura di una soluzione ottima
2) Definire ricorsivamente il valore di una soluzione ottima
   - La soluzione ottima ad un problema contiene le soluzioni ottime ai sottoproblemi
3) Calcolare il valore di una soluzione ottima “bottom-up” (cioè calcolando prima le soluzioni ai casi più semplici)
   - Si usa una tabella per memorizzare le soluzioni dei sottoproblemi
   - Evitare di ripetere il lavoro più volte, utilizzando la tabella
4) Costruire la (una) soluzione ottima.

Esempio:

Sottovettore di valore massimo

Esempio
sottovettore di valore massimo

- Consideriamo un vettore *V*[] di *n* elementi (positivi o negativi che siano)
- Vogliamo individuare il sottovettore di *V* la cui somma di elementi sia massima

3	-5	10	2	-3	1	4	-8	7	-6	-1

- Domanda: quanti sono i sottovettori di *V*?
  - 1 sottovettore di lunghezza n
  - 2 sottovettori di lunghezza n-1
  - 3 sottovettori di lunghezza n-2
  - ...
  - k sottovettori di lunghezza n-k+1
  - ...
  - n sottovettori di lunghezza 1

Totale: $n(n-1)/2 = \Theta(n^2)$

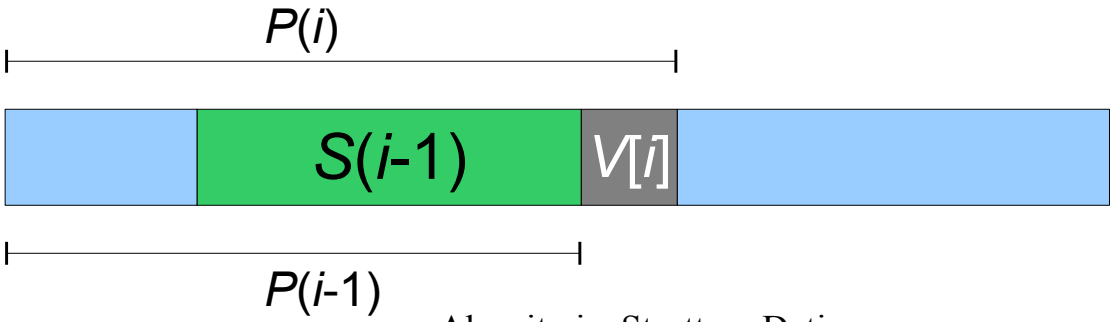
Approccio / 1

- Sia *P*(*i*) il problema che consiste nel determinare il valore massimo della somma degli elementi dei sottovettori del vettore *V*[1..*i*] che hanno *V*[*i*] come ultimo elemento
- Sia *S*(*i*) il valore della soluzione di *P*(*i*)
  - quindi *S*(*i*) è il valore massimo della somma dei sottovettori di *V*[1..*i*] che hanno *V*[*i*] come ultimo elemento
- La soluzione *S** al problema di partenza può essere espressa come

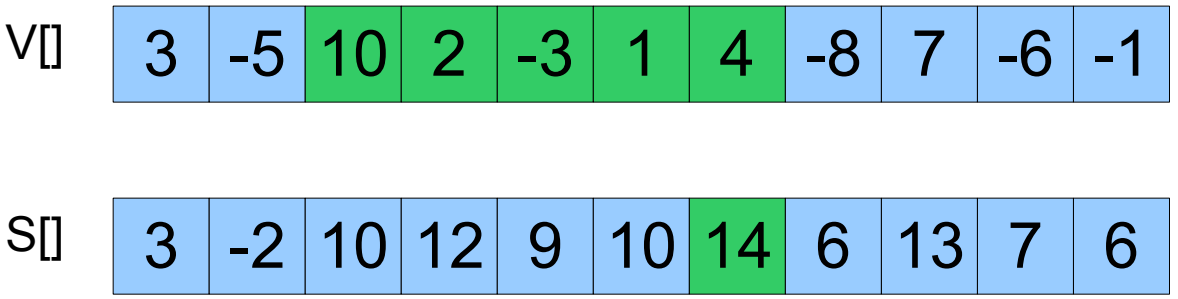
$$S^* = max_{1 \le i \le n} S(i)$$

Approccio / 2

- $P(1)$ ammette una unica soluzione
  - $S(1) = V[1]$
- Consideriamo $P(i)$
  - Supponiamo di avere già risolto $P(i\text{-}1)$, e quindi di conoscere $S(i\text{-}1)$
  - Se $S(i\text{-}1)+V[i] \geq V[i]$
    - allora la soluzione ottima è $S(i) = S(i\text{-}1)+V[i]$
  - Se $S(i\text{-}1)+V[i] < V[i]$
    - allora la soluzione ottima è $S(i) = V[i]$



Esempio



$$S[i] = \max \{ V[i], V[i]+S[i-1] \}$$

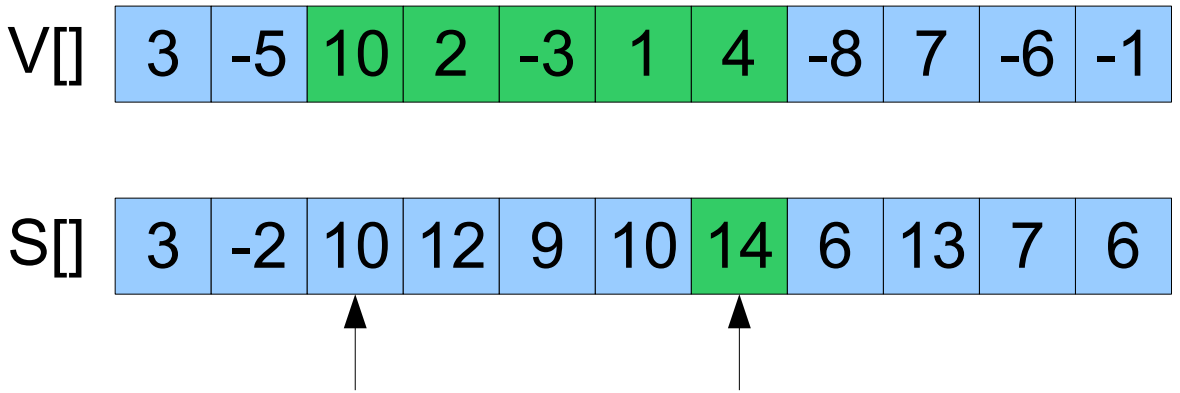
L'algoritmo

```
algoritmo sottovettoreMax(V, n) → intero
    S := new array [1..n] di interi;
    S[1] := V[1];
    max := 1;
    for i:=2 to n do
        if ( S[i-1]+V[i]≥V[i] ) then
            S[i] := S[i-1]+V[i];
        else
            S[i] := V[i];
        endif
        if ( S[i]>S[max] ) then
            max := i;
        endif
    endfor
    return S[max];
```

Come individuare il sottovettore?

- Fino ad ora siamo in grado di calcolare il valore della massima somma di un sottovettore di *V*[]
- Come facciamo a determinare *quale* sottovettore produce tale somma?
  - L'indice finale del sottovettore già ce l'abbiamo
  - L'indice iniziale lo possiamo ricavare procedendo “a ritroso”
    - Se *S*[*i*] = *V*[*i*], il sottovettore massimo inizia nella posizione *i*

Esempio




```
algoritmo indiceInizio(V, S, max) → intero
    i := max;
    while( S[i] != V[i] ) do
        i := i-1;
    endwhile
    return i;
```

algoritmi greedy

Introduzione

- La programmazione dinamica
  - Valuta tutte le decisioni possibili in maniera bottom-up
  - Evita di ricalcolare soluzioni più volte, usando un vettore/tabella per memorizzare le soluzioni ai sottoproblemi
- Un algoritmo greedy (ingordo, goloso)
  - Seleziona una sola delle possibili decisioni...
  - ...quella che sembra ottima (localmente)...
  - ...”sperando” così di ottenere un ottimo globale

Introduzione

- Quando applicare la tecnica greedy?
  - Quando è possibile *dimostrare* che esiste una *scelta ingorda*
    - “Fra le molte scelte possibili, ne può essere facilmente individuata una che porta sicuramente alla soluzione ottima”
  - Quando il problema ha sottostruttura ottima
    - “Fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale”
- Non tutti i problemi hanno una scelta ingorda
  - Quindi non tutti i problemi si possono risolvere con una tecnica greedy
  - in alcuni casi, soluzioni non ottime possono essere comunque interessanti

Algoritmo greedy generico

```
algoritmo paradigmaGreedy(insieme di candidati C) → soluzione
    S := ∅
    while ( (not ottimo(S)) and (C ≠ ∅) ) do
        x := seleziona(C)
        C := C - {x}
        if (ammissibile(S ∪ {x})) then
            S := S ∪ {x}
        endif
    endwhile
    if (ottimo(S)) then
        return S
    else
        errore ottimo non trovato
    endif
```

Ritorna *true* sse la soluzione S è ottima

Estrae un candidato dall'insieme C

Ritorna *true* sse la soluzione candidata è una soluzione ammissibile (anche se non necessariamente ottima)

Problema del resto

Problema del resto

- Input
  - Un numero intero positivo *n*
- Output
  - Il più piccolo numero intero di monete necessarie per erogare un resto di *n* centesimi, supponendo che i tagli delle monete siano 50c, 20c, 10c, 5c, 2c e 1c
  - Supponiamo di avere una quantità finita di monete da poter erogare
- Esempi
  - *n* = 78, 5 pezzi: 50+20+5+2+1
  - *n* = 18, 4 pezzi: 10+5+2+1

Algoritmo greedy per il resto

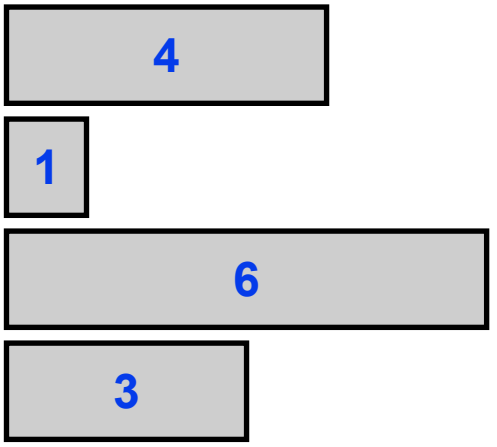
- Insieme dei candidati *C*
  - Insieme finito delle monete a disposizione nel serbatoio
- Soluzione *S*
  - Insieme delle monete da restituire
- `ottimo(S)`
  - true sse la somma dei valori di *S* è esattamente uguale al resto
- `ammissibile(S)`
  - true sse la somma dei valori di *S* è minore o guale al resto
- `seleziona(C)`
  - scegli la moneta di valore massimo in *C*

Algoritmo greedy per il resto

- L'algoritmo greedy sceglie tra le monete disponibili quella di valore massimo che risulti minore o uguale al residuo da erogare
- Si può dimostrare che l'algoritmo greedy potrebbe fallire in sistemi monetari “particolari”
  - Esempio: tagli da 1, 3, 4
  - Dobbiamo erogare un resto di 6
  - L'algoritmo greedy produce 4, 1, 1
  - La soluzione che minimizza il numero di pezzi sarebbe 3, 3
  - L'algoritmo greedy potrebbe NON trovare la soluzione; ad esempio, se il serbatoio contiene 4, 3, 3 l'algoritmo greedy sceglie dapprima la moneta 4, e poi non puo' proseguire

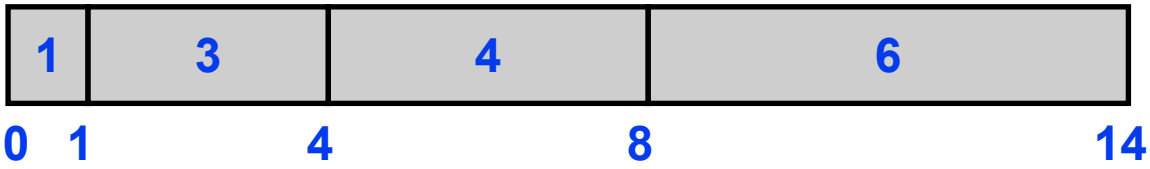
Problema di scheduling (Shortest Job First)

Algoritmo di scheduling—Shortest Job First



- Definizione:
  - 1 processore, *n* job p_1, p_2, ..., p_n
  - Ogni job p_i ha un tempo di esecuzione *t*[*i*]
  - Minimizzare il tempo medio di completamento
    - Il libro considera il problema di minimizzare il tempo medio di attesa

(4+5+11+14)/4 = 34/4



(1+4+8+14)/4 = 27/4

Algoritmo greedy di scheduling

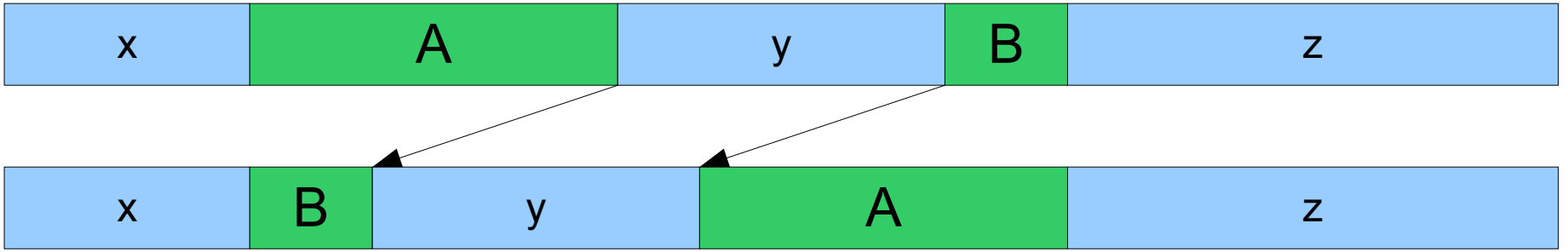
- Siano p_1, p_2, … p_n gli *n* job che devono essere schedulati
- L'algoritmo greedy esegue *n* passi
  - ad ogni passo sceglie e manda in esecuzione il job più breve tra quelli che rimangono

Algoritmo greedy per lo scheduling Shortest-job-first

- Insieme dei candidati C
  - Job da schedulare, inizialmente { p_1, p_2, … p_n }
- Soluzione S
  - Ordine dei job da schedulare: p_{i1}, p_{i2}, … p_{in}
- `ottimo(S)`
  - True sse l'insieme S contiene tutti i job
- `ammissibile(S)`
  - Restituisce sempre true
- `seleziona(C)`
  - Sceglie il job di durata minima in C

Dimostrazione di ottimalità

- Consideriamo un ordinamento dei job in cui un job "lungo" A viene schedulato prima di uno "corto" B
  - x, y e z sono sequenze di altri job



- Osserviamo:
  - Il tempo di completamento dei job in x non cambia
  - Il tempo di completamento dei job in z non cambia
  - Il tempo di completamento di A nella seconda soluzione è uguale al tempo di completamento di B nella prima soluzione
  - Il tempo di completamento di B si riduce
  - Il tempo di completamento dei job in y si riduce

Algoritmi greedy

- Vantaggi
  - Semplici da programmare
  - Solitamente efficienti
  - Quando è possibile dimostrare la proprietà di scelta greedy, danno la soluzione ottima
  - La soluzione sub-ottima può essere accettabile
- Svantaggi
  - Non sempre applicabili se si vuole la soluzione ottima