Strutture Dati e Algortimi

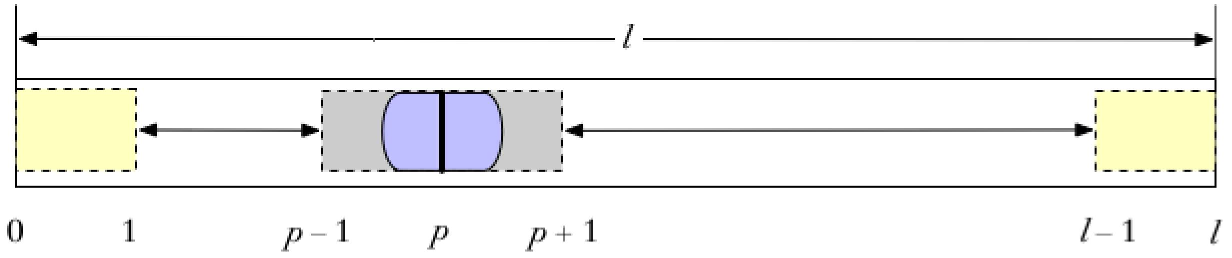
La ricorsione

Luca Di Gaspero, Università degli Studi di Udine

Il problema del parcheggio

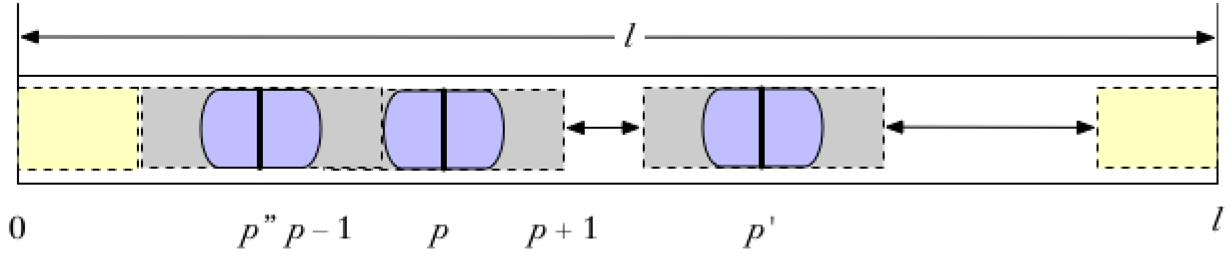
Il problema del parcheggio

Dobbiamo parcheggiare delle automobili di lunghezza unitaria in una via lunga l unità.



- ciascun'automobile arriva sulla via e parcheggia in un punto p (relativo al suo centro) scelto a caso fra quelli disponibili
- in particolare, può parcheggiare in un punto p consentre di avere almeno 0.5 unità di spazio a sinistra e a destra in modo da consentire le manovre
- eventuali automobili vicine possono condividere lo spazio di manovra

Il problema del parcheggio

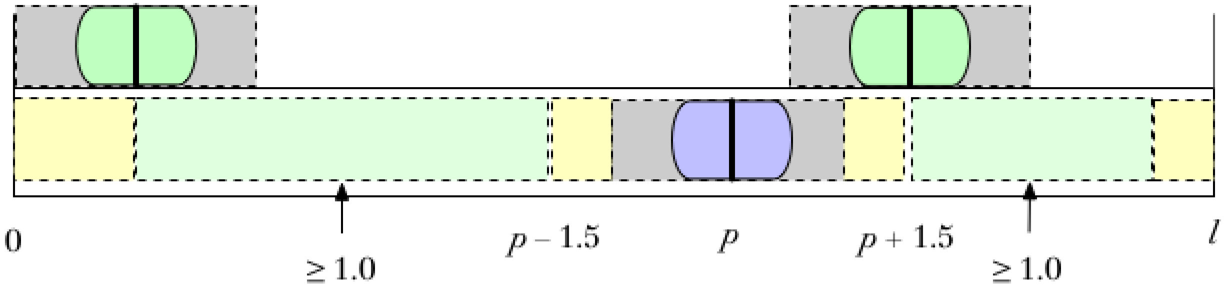


Quindi una nuova auto può parcheggiare in una posizione p' *valida* se $1 \leq p' \leq l - 1$ e, anche, $p' \geq p + 1.5$ oppure $p' \leq p - 1.5$.

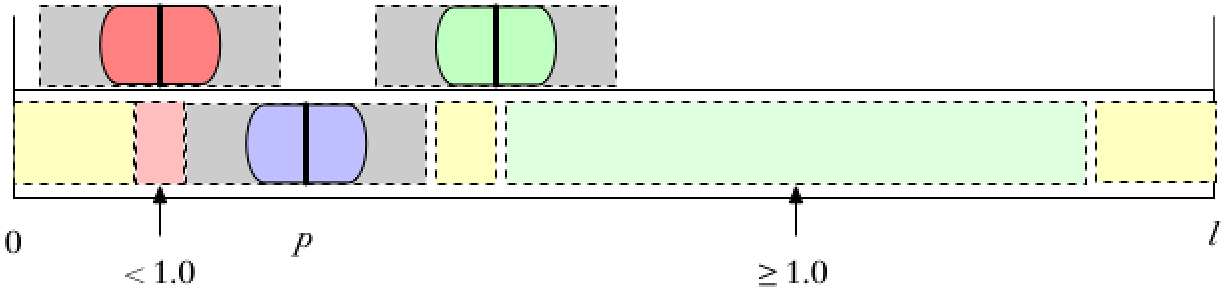
- Come possiamo definire un algoritmo per generare delle posizioni di parcheggio casuali ma valide?
- **Nota 1**: *non importa quante macchine riusciamo a parcheggiare, l'importante è che i parcheggi siano validi.*
- **Nota 2**: *si parte da un parcheggio completamente vuoto.*

Il problema del parcheggio: bozza di algoritmo

1. generare un numero a caso p compreso tra 1 e $l - 1$ per parcheggiare la prima automobile
2. guardare al lato sinistro rispetto all'automobile appena parcheggiata:
   a) c'è spazio sufficiente per parcheggiare ancora delle automobili



   b) non c'è spazio sufficiente per parcheggiare delle automobili

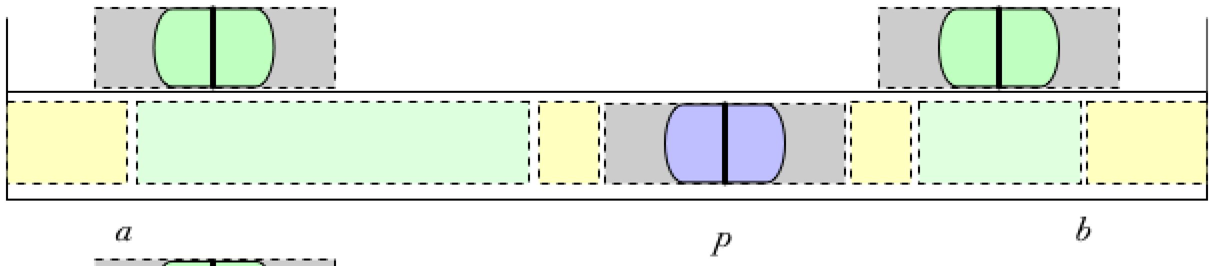


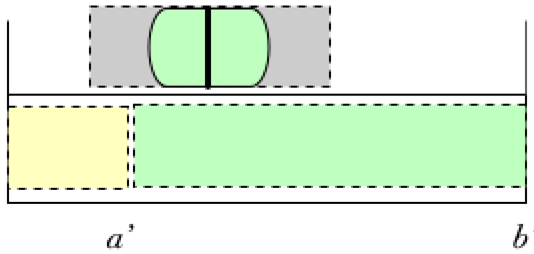
3. applicare lo stesso ragionamento sul lato destro

Il problema del parcheggio: bozza di algoritmo

Indichiamo con a e b i limiti inferiore e superiore delle possibili posizioni del parcheggio.

- Che caratteristiche ha il problema di parcheggiare sul lato sinistro (o sul lato destro) rispetto all'automobile appena parcheggiata?





- È esattamente lo stesso problema ma con dei limiti diversi per il parcheggio: $a' = 1$ e $b' = p - 1.5$

Il problema del parcheggio: bozza di algoritmo

1. generare un numero a caso p compreso tra $a = 1$ e $b = l - 1$ per parcheggiare la prima automobile
2. guardare al lato sinistro rispetto all'automobile appena parcheggiata e verificare se c'è spazio sufficiente per parcheggiare ancora delle automobili
   2.1) in caso affermativo **riapplicare l'algoritmo del parcheggio** sull'intervallo $a = a$, $b = p - 1.5$
3. guardare al lato destro rispetto all'automobile appena parcheggiata e verificare se c'è spazio sufficiente per parcheggiare ancora delle automobili
   3.1) in caso affermativo **riapplicare l'algoritmo del parcheggio** sull'intervallo $a = p + 1.5$

La ricorsione

Ricorsione

- È sia una tecnica di programmazione che uno strumento per costruire soluzioni di problemi

- Una funzione è detta *ricorsiva* se nella sua definizione compare un riferimento (chiamata) a se stessa
  - la funzione ricorsiva sa risolvere direttamente dei casi particolari del problema, detti *casi di base*, in tal caso è in grado di restituire immediatamente un risultato
  - se invece le vengono passati dei dati che non costituiscono uno dei casi di base chiama se stessa passando dei dati *"ridotti"* o *"semplificati"*

- Ad ogni chiamata i dati vengono si riducono così da arrivare, ad un certo punto, ad uno dei casi di base

L'algoritmo del parcheggio

- Quali sono i casi di base?
- In questo caso sono dei casi con una **connotazione negativa**: sono quelli in cui **non è possibile parcheggiare ulteriormente**

Esempio: l'algoritmo per il parcheggio, traduzione in C

```c
#include <stdio.h>
#include <stdlib.h> // per random

double drand(double low, double high);

void park(double a, double b) {
  double p;
  p = drand(a, b);
  printf("Auto parcheggiata nella posizione %f\n", p);
  if (p - a >= 1.0) // c'è spazio sufficiente a sinistra almeno per una macchina
    park(a, p - 1.5);
  if (b - p >= 1.0) // c'è spazio sufficiente a destra almeno per una macchina
    park(p + 1.5, b);
}

int main() {
  double l = 10.0;
  park(1, l - 1);
  return EXIT_SUCCESS;
}
```

Esempio: l'algoritmo del parcheggio, rendere esplicito il caso di base

```c
void park(double a, double b) {
  double p;
  if (b - a < 1.0) // non c'è spazio sufficiente per parcheggiare
    return;
  p = drand(a, b);
  printf("Auto parcheggiata nella posizione %.1f\n", p);
  park(a, p - 1.5); // tento il parcheggio a sinistra
  park(p + 1.5, b); // tento il parcheggio a destra
}
```

Esempio: l'algoritmo per il parcheggio, la funzione `drand`

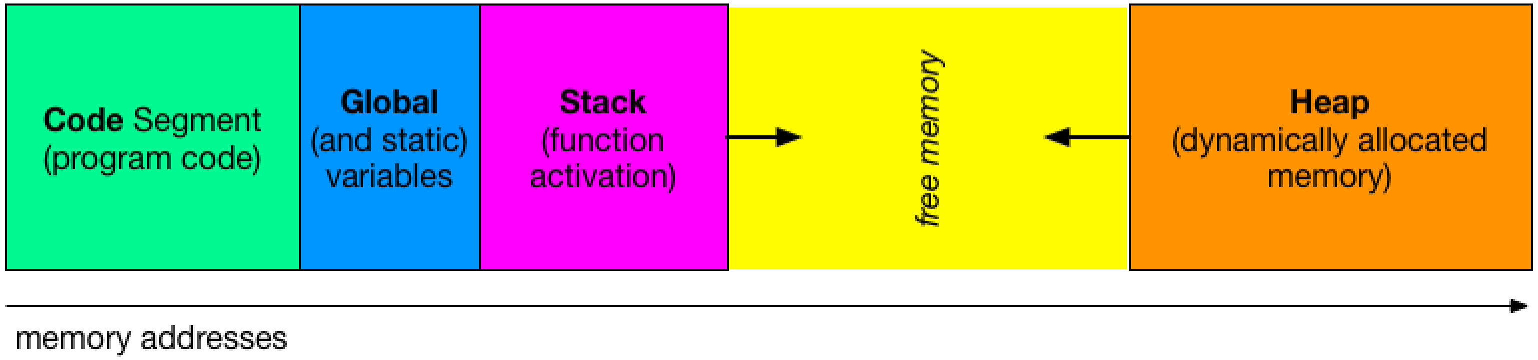
```
double drand(double low, double high) {
  int randomNum;
  double d;
  randomNum = rand(); // valore compreso fra 0 e RAND_MAX
  d = randomNum / (double)RAND_MAX; // valore reale compreso fra 0.0 e 1.0
  return low + (high - low) * d;
}
```

$$f(n) = \begin{cases} 1 & \text{se}\, n = 0 \\ n \cdot f(n-1) & \text{se}\, n > 0 \end{cases}$$

```
long fattoriale(long n) {
  if (n == 0)
    return 1; /* caso di base */
  else   /* semplificazione ☟ */
    return n * fattoriale(n - 1);
}
```

Ricorsione: meccanismo computazionale

- Quando la funzione chiama se stessa la sua esecuzione si sospende per eseguire la nuova chiamata
  - L'esecuzione riprende quando la chiamata interna termina
  - La sequenza di chiamate ricorsive termina quando quella *più annidata* incontra uno dei casi di base
- Ogni chiamata alloca sullo *stack* nuove istanze dei parametri e delle variabili locali
  - chiamati **record di attivazione**



Ricorsione: esempio di esecuzione

```c
long fattoriale(long n) {
  long risultato; // introduciamo una variabile temporanea
  if (n == 0)
    risultato = 1;
  else
    risultato = n * fattoriale(n - 1);
  return risultato;
}

int main() {
  long x = fattoriale(4);
  printf("Il fattoriale è: %ld\n", x);
  return 0;
}
```

Proviamo il codice sul sito Python Tutor (in modalità C) https://goo.gl/n6zSSK

Ricorsione e induzione

La ricorsione è basta sul principio di *induzione matematica*:

- Se una proprietà P vale per $n = n_0$ (**caso base**)
- e si può provare che, *assumendola valida* per n, vale anche per $n + 1$ (**passo induttivo**)
- allora P vale per ogni $n \geq n_0$

Analogamente, risolvere il problema con un approccio ricorsivo comporta:

- l'identificazione del **caso di base** in cui la soluzione sia nota
- la capacità di **esprimere il caso generico** in termini dello **stesso problema** ma in uno o più casi più semplici

Esempio di ricorsione

```
int f(int a, int b) {
  assert(b >= 0); // assumiamo b >= 0
  if (b == 0)
    return a;
  else
    return f(a, b - 1) + 1;
}
```

Cosa fa questa funzione? https://goo.gl/TnsSt3

Esempio: numeri di Fibonacci

n	0	1	2	3	4	5	6	7	8	9	10	11
F_n	1	1	2	3	5	8	13	21	34	55	89	144

La successione è definita come:

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$

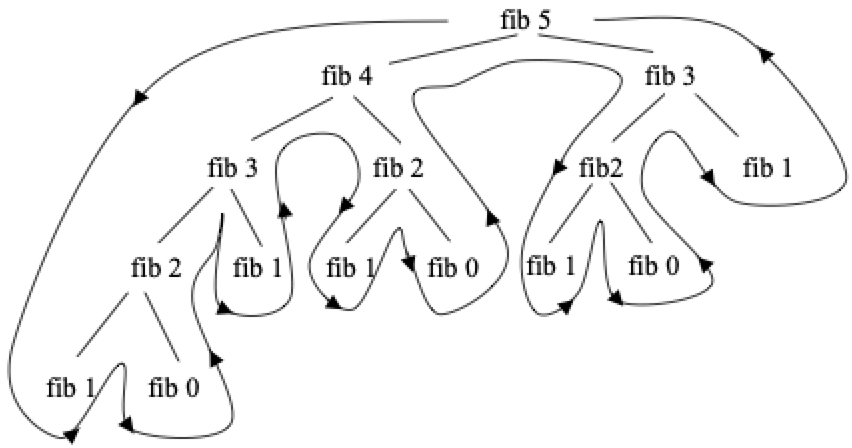
Numeri di Fibonacci

```
// Scriviamo il codice assieme
```

Numeri di Fibonacci

```
int fibonacci(int n) {
  if (n == 0 || n == 1)
    return 1;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Numeri di Fibonacci: albero delle chiamate



Esempio: massimo comun divisore

$$\mathrm{mcd}(m, n) = \begin{cases} m & \text{se } n = m \\ \mathrm{mcd}(m - n, n) & \text{se } m > n \\ \mathrm{mcd}(m, n - m) & \text{se } m < n \end{cases}$$

Massimo comun divisore

```
// Scriviamo il codice assieme
```

Massimo comun divisore

```c
int mcd(int m, int n) {
  if (n == m)
    return n;
  if (m > n)
    return mcd(m - n, n);
  else
    return mcd(m, n - m);
}
```

Ricorsione in coda (tail recursion)

Ricorsione in coda

- La ricorsione in coda si ha quando la funzione ricorsiva chiama se stessa come ultima istruzione prima di terminare
- Il risultato di ciascuna chiamata non di base viene calcolato solo attraverso la restituzione del valore ottenuto dalla chiamata ricorsiva

```
<tipo> f(/*parametro */) {
  ...
  return f(/* parametro ridotto */;
}
```

- Il risultato della chiamata finale (caso di base) coincide con il risultato di tutte le chiamate

Ricorsione in coda: esempio

Formula di Euclide per il massimo comun divisore:

$$\mathrm{mcd}(m,n) = \begin{cases} m & \text{se } n = 0 \\ \mathrm{mcd}(n, m \mod n) & \text{altrimenti} \end{cases}$$

```
int mcd_euclide(int m, int n) {
  if (n == 0)
    return m;
  return mcd_euclide(n, m % n); // il valore non subisce elaborazioni
}
```

Ricorsione in coda: il fattoriale è una funzione ricorsiva in coda?

```
long fattoriale(long n) {
  if (n <= 1)
    return 0;
  else
    return n * fattoriale(n - 1);
}
```

- **No**: il valore restituito nelle chiamate ricorsive consiste anche in un'operazione di moltiplicazione

Qualora fosse necessario calcolare valori intermedi è necessario utilizzare dei parametri che costituiscono degli **accumulatori**

```c
long fattoriale_tail(long n, long acc) {
  if (n > 0) {
    acc = acc * n;
    return fattoriale_tail(n - 1, acc);
  }
  return acc; // caso di base
}
```

```c
// la prima chiamata dev'essere:
// fattoriale_tail(n, 1):
// la incapsuliamo in una funzione `wrapper`
long fattoriale(long n) {
  return fattoriale_tail(n, 1);
}
```

Ricorsione in coda: proprietà

- Escluso il risultato, tutti i dati della chiamata della funzione non servono più e possono essere scartati
- Il *risultato* calcolato sarà quello calcolato dalla sua chiamata *più profonda*
    - infatti la chiamata ricorsiva non fa altro che restituire il valore della funzione sul caso ridotto senza ulteriori elaborazioni
- Alcuni linguaggi ottimizzano l'esecuzione eliminando i record di attivazione dallo stack
- Le funzioni con ricorsione in coda, tuttavia, possono essere riscritte, eventualmente eliminando la ricorsione in coda e trasformandola in costrutti iterativi

ricorsiva generica

```
<tipo> funzione_ricorsiva(<tipo> x) {
  if (caso_base(x)) {
    istruzioni_caso_base;
    return risultato;
  }
  istruzioni_non_base_di_accumulazione;
  return funzione_ricorsiva(riduci(x));
}
```

La funzione ricorsiva ha tipicamente questa struttura

```
<tipo> funzione_iterativa(<tipo> x) {
  while (!caso_base(x)) {
    istruzioni_non_base_di_accumulazione;
    x = riduci(x);
  }
  istruzioni_caso_base;
  return risultato;
}
```

E questa è una sua possibile traduzione

Ricorsione in coda: traduzione del fattoriale

```
long fattoriale_iterativo(long n) {
  long acc = 1;
  while (n > 0) {
    acc = acc * n; /* istruzioni non base di accumulazione */
    n = n - 1; /* riduci */
  }
  /* nessuna istruzione specifica per il caso base */
  return acc;
}
```

Funzioni ricorsive: pregi e difetti

Pro:

- Metodo particolarmente utile su strutture dati inerentemente *ricorsive* (alberi, grafi)
- Qualora applicabili, le funzioni ricorsive sono più chiare (ed eleganti), più semplici, più brevi e più comprensibili rispetto alle versioni iterative
    - consentono di risolvere un problema, anche complesso, con poche linee di codice

Contro:

- Le risorse di memoria richieste sono generalmente superiori a quelle della corrispondente soluzione iterativa (a causa dell'allocazione sullo stack dei record di attivazione)
    - richiede tempo per la gestione dello stack (allocare e passare i parametri, salvare l'indirizzo di ritorno)
    - richiede memoria (alloca un nuovo record di attivazione ad ogni chiamata)

Quando utilizzare la ricorsione

- In generale qualunque problema ricorsivo può essere risolto in modo non ricorsivo (ossia iterativo)
    - Talvolta, però, al prezzo di una maggiore difficoltà o complessità di scrittura
- È utile quando la soluzione iterativa è difficile da individuare o in generale più complessa
- In caso sia possibile definire una versione con ricorsione in coda è certamente da preferirsi