Costo di esecuzione e complessità

Modello di costo e complessità di un algoritmo

- Il costo di esecuzione di un algoritmo quantifica le risorse necessarie per l’esecuzione dell’algoritmo stesso
  - numero di operazioni
  - spazio di memoria

Modello di costo

- NON si misura la durata di esecuzione di un algoritmo
- NON si misura il numero di istruzioni eseguite nella CPU
  - saremmo dipendenti da dettagli di implementazione, caratteristiche della CPU, etc.
- Si usa un modello di costo (ne esistono diversi)

Modello di costo usato

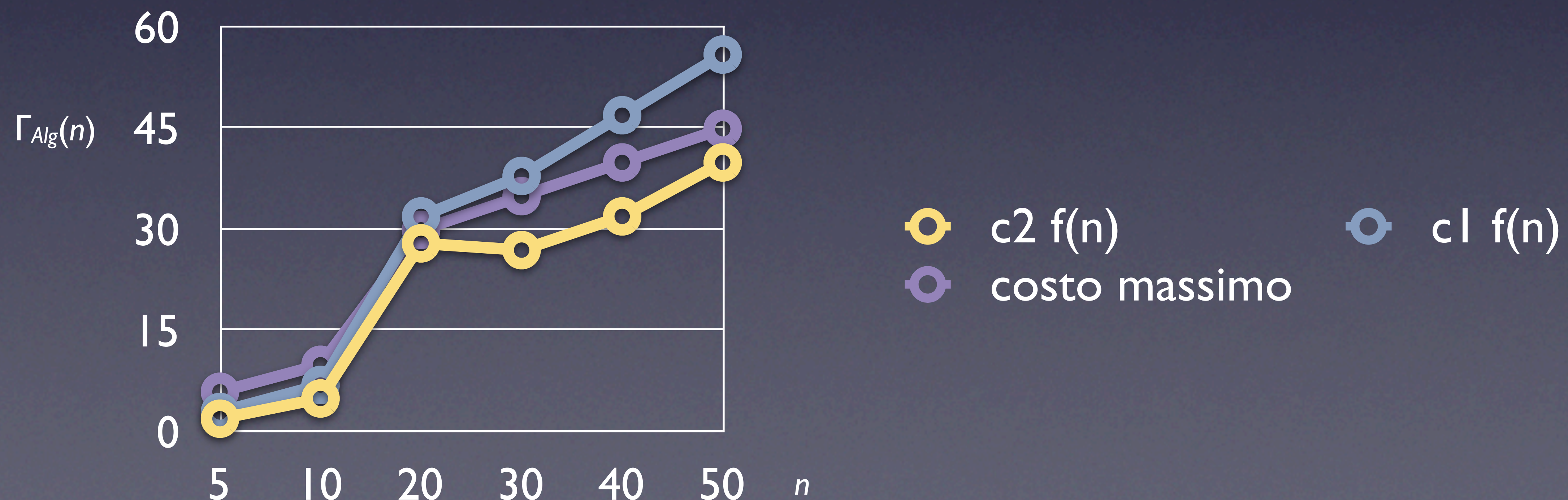
- il costo è dovuto solo agli operatori delle espressioni: valutato 1 per tutti
- si trascurano: dereferenziazione, selezione di un elemento di array, allocazioni di variabili locali/parametri formali, chiamate a funzione

- In realtà:
  - certe operazioni costano meno, secondo l'HW usato (es. somma vs. prodotto)
  - una chiamata a funzione richiede operazioni sullo stack (quantomeno nel C) e salti del program counter
  - il riferimento a variabile che comporta il calcolo di un indirizzo ha un costo dipendente da HW

- Dati: Γ modello di costo, *Alg* algoritmo, *n* dati di ingresso D_n
- $\Gamma_{Alg}(n, D_n)$
- il costo di esecuzione dipende dalla dimensione del dataset di ingresso
- $\Gamma_{Alg}(n) = Sup_{D_n}\, \Gamma_{Alg}(n, D_n)$
- Si qualifica il costo di *Alg* attraverso una funzione che associa ad ogni *n* del dataset il costo massimo di *Alg* al variare dei dataset

Complessità di un algoritmo

- La complessità è una stima sul costo massimo di un algoritmo $\Gamma_{Alg}(n)$
- L'algoritmo *Alg* ha complessità $C(Alg)=O(f(n))$ se: $\exists c_1, c_2, \tilde{n} \mid \forall n > \tilde{n} \;\; c_1 \cdot f(n) \geq \Gamma_{Alg}(n) \geq c_2 \cdot f(n)$



Proprietà algebriche

- $f(n)$ è dello stesso ordine di grandezza di $g(n)$, e si scrive $O(f(n))=O(g(n))$ se $\exists c_1, c_2, \tilde{n} \mid \forall n > \tilde{n} \quad c_1 \cdot f(n) \geq g(n) \geq c_2 \cdot f(n)$
- $f(n)$ domina $g(n)$, e si scrive $O(f(n))>O(g(n))$ se
  $\forall c_2 \; \exists \tilde{n} \mid \forall n > \tilde{n} \quad f(n) \geq c_2 \cdot g(n)$

Ordini di grandezza

- Si possono applicare le semplificazioni del calcolo simbolico, usate nel calcolo degli ordini di grandezza delle funzioni, al calcolo della complessità:

1. se $O(f(n))=O(g(n))$ allora $C(Alg)=O(f(n)) \Leftrightarrow C(Alg)=O(g(n))$

2. se $O(f(n))>O(g(n))$ allora $C(Alg)=O(f(n)+g(n)) \Leftrightarrow C(Alg)=O(g(n))$

3. $\forall c>0 \;\; O(f(n))=O(c \cdot f(n))$

1. $C(Alg)=O(1)$ se il numero di operazioni è invariante rispetto al numero di elementi del dataset

2. $C(Alg)=O(\ln_2(n))>O(1)$; $O(n^\alpha)>O(\ln_2(n))$ e $(\forall \alpha_1>\alpha_2\ O(n^{\alpha_1})>O(n^{\alpha_2}))$; $O(2^{\beta n})> O(n^\alpha)$ e $(\forall \beta_1>\beta_2\ O(2^{\beta_1 n})> O(2^{\beta_2 n}))$

3. $n!$ si riconduce all'esponenziale con Stirling: $n! \approx 2^{n \ln_2(n)}$, ovvero $2^n<n!<2^{n^2}$

Algoritmi di ricerca

Ricerca sequenziale e binaria

Ricerca: definizione

- Si deve decidere su un test di inclusione:
  D spazio sul quale è definita una relazione di uguaglianza (==)
  V multi-insieme di N elementi di D ($V \subseteq D^N$)
  v valore target ($v \in D$)
- Il problema è decidere se $\exists V_i \in V | v == V_i$
- Si tratta solo del caso in cui i dati sono in memoria primaria (il costo dominante dell'operazione è dato dalle operazioni di manipolazione)

Ricerca sequenziale

- Se non esistono ipotesi sui valori di V si devono controllare tutti uno ad uno
- C(*Alg*)=O(N): compariamo contro uno degli elementi, quindi si prosegue sui rimanenti N-1:
  $\Gamma_{seq}(N) = c_1 + \Gamma_{seq}(N-1)$ se N>0
  $\Gamma_{seq}(N) = c_2$ se N=0
  quindi $\Gamma_{seq}(N) = c \cdot N$ da cui C(*seq*)=O(N)

Esercizio

- Scrivere le funzioni di ricerca per le tre implementazioni di lista della lezione precedente

Ricerca binaria

- Su D è definita una relazione di ordine totale (≥), gli elementi di V possono essere numerati in modo univoco: V_i i=0,N-1
- Se i valori in V sono ordinati (i.e. $V_{i+1} \geq V_i$ per i=0,N-2) allora l'esito di un solo test può escludere più di un elemento di V dallo spazio della ricerca

Ricerca binaria

- Se i dati sono memorizzati su una struttura che consente l'accesso diretto ai valori in posizione intermedia (es. array o lista in forma sequenziale) è conveniente la ricerca binaria (ovvero dicotomica):

  il target viene comparato con l'elemento in posizione intermedia; se uguale si termina con successo, se maggiore si cerca nella metà di destra, altrimenti in quella di sinistra. Se la ricerca si applica ad un vettore di dimensione 0 allora il target non è presente.

Costo

- il caso pessimo si ha se il test di uguaglianza fallisce sempre: costo *c* più il costo per cercare nella metà selezionata
- $\Gamma_{Bin}(N) = c_1 + \Gamma_{Bin}(N/2)$ se $N>0$, c_2 se $N==0$ da cui $\Gamma_{Bin}(N) = c + \ln_2(N)$ e quindi $C_{bin}=O(\ln_2(N))$

Costo

- Se i dati fossero memorizzati su una lista collegata l'accesso all'elemento mediano sarebbe lineare...
- $\Gamma_{Bin}(N)= c_1 \cdot N/2+ \Gamma_{Bin}(N/2)$ se $N>0$, c_2 se $N==0$
  da cui $\Gamma_{Bin}(N)= c \cdot N$ e quindi $C_{bin}=O(N)$
- E' il costo della ricerca sequenziale !!

Implementazione ricorsiva

```
Boolean binarySearch(int *V, int N, int target)
{
  if (N>0)
  {
    if (V[N/2]==target)
      return TRUE;
    else if (V[N/2]<target)
      return binarySearch(V,N/2,target);
    else
      return binarySearch(&V[N/2+1],N-N/2-1, target);

  }
  else
    return FALSE;

}
```

Implementazione iterativa

- Jon Bentley in Programming Pearls:
“While the first binary search was published in 1946, the first binary search that works correctly for all values of n did not appear until 1962.”

- Nell’implementazione iterativa si deve tenere traccia del range in cui cercare il target, finché si trova o finché tale range è vuoto

Pseudo-codice

```
inizializza il range tra 0..n-1

loop

  { invariante: mustbe(range) }
  if range è vuoto,
    break e riporta che t non è nell’array
  calcola m, metà del range
  usa m per valutare la riduzione del range
    if t è trovato durante la riduzione, break
    e rendi la posizione
```

Si useranno due indici: l e u per rappresentare il range (prima e ultima posizione, ma si può implementare anche come posizione iniziale e lunghezza).

la funzione logica mustbe() indica che se t è nell’array allora deve essere in [l,u]

```
l=0; u=n-1;

loop

  { invariante: mustbe(l,u) }
  if l > u
    p=-1; break;
  m=(l+u)/2;   /* può essere un GROSSO problema ! */
  usa m per valutare la riduzione del range
      if t è trovato durante la riduzione, break
      e rendi la posizione
```

```
l=0; u=n-1;

loop

  { invariante: mustbe(l,u) }
  if l > u
    p=-1; break;
  m=(l+u)/2;   /* può essere un GROSSO problema ! */
```

```
l=0; u=n-1;

loop

  { invariante: mustbe(l,u) }
  if l > u
    p=-1; break;
  m=(l+u)/2;   /* può essere un GROSSO problema ! */
  case:
    x[m]<t: mustbe(m+1,u)
    x[m]==t: p=m; break;
    x[m]>t: mustbe(l,m-1)
```

```
l=0; u=n-1;

loop

   { invariante: mustbe(l,u) }
   if l > u
     p=-1; break;
   m=(l+u)/2;    /* può essere un GROSSO problema ! */
```

```
l=0; u=n-1;

loop

  { invariante: mustbe(l,u) }
  if l > u
    p=-1; break;
  m=(l+u)/2;   /* può essere un GROSSO problema ! */
  case:
    x[m]<t: l=m+1
    x[m]==t: p=m; break;
    x[m]>t: u=m-1
```

```
{mustbe(0,n-1)}
l=0; u=n-1;
{mustbe(l,u)}
loop
  {mustbe(l,u)}
  if l > u
    {l>u && mustbe(l,u)}
    {t non è nell'array}
    p=-1; break;
  {l<=u && mustbe(l,u)}
  m=(l+u)/2;
  {mustbe(l,u) && l<=m<=u}
  case:
    x[m]<t:
      {mustbe(l,u) && cantbe(0,m)}
      {mustbe(m+1,u)}
      l=m+1
      {mustbe(l,u)}
    x[m]==t:
      {x[m]==t}
      p=m; break;
    x[m]>t:
      {mustbe(l,u) && cantbe(m,n)}
      {mustbe(l,m-1)}
      u=m-1
      {mustbe(l,u)}
{mustbe(l,u)}
```

Dallo pseudocodice al codice

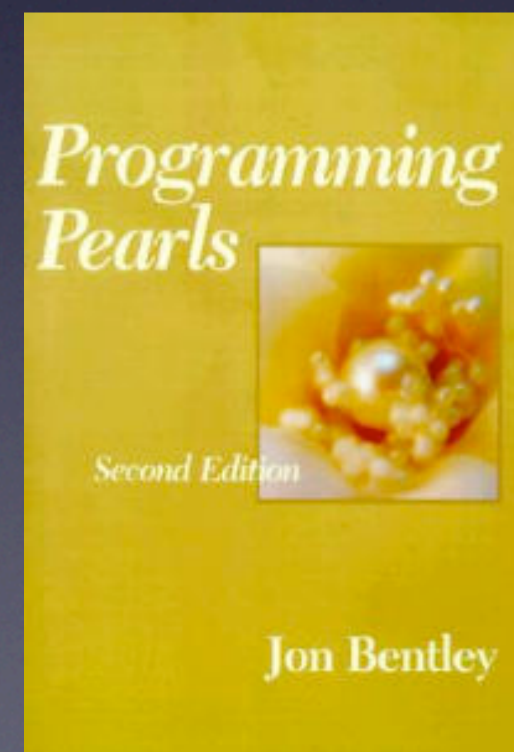
```
typedef int DataType;
int n;
DataType x[MAXN];

int binarySearch(DataType t)
{
  int l,u,m;
  l = 0; u = n-1;
  while (l<=u) {
    m = (l+u)/2;  /* può essere un GROSSO problema! */
    if (x[m]<t)
      l=m+1;
    else if (x[m]==t)
      return m;
    else /* x[m]>t */
      u=m-1;
  }
  return -1;
}
```

- `m = (l+u)/2` è un problema quando l+u è più grande del massimo int rappresentabile (2^{31}-1)
- Un array di un miliardo di elementi (~2^{30}) può dare questi problemi
- Una possibile soluzione:
  - `m = ((unsigned int)low + (unsigned int)high)) >> 1;`

Bibliografia

- "Algoritmi in C"
  Sedgewick Robert
  Pearson Education
- "Programming Pearls"
  Jon Bentley
  Addison Wesley



- Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken: http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html