Ricorsione e Divide et Impera

Sommario

- Paradigma di programmazione Divide et Impera
- Ricorsione e tempo di calcolo

Ricorsione

- Molti algoritmi hanno una struttura ricorsiva
- Struttura ricorsiva:
  - l'algoritmo è definito in termini di se stesso
  - un algoritmo è ricorsivo quando contiene una o più istruzioni di chiamata a se stesso
  - le chiamate ricorsive non possono succedersi in modo infinito altrimenti l'algoritmo non terminerebbe mai
  - deve sempre esistere una *condizione di terminazione* che determina quando l'algoritmo smette di richiamarsi
  - in questo caso la computazione prosegue eseguendo un insieme di istruzioni dette *passo base*

Tipi di ricorsione: Ricorsiva Lineare

- **Ricorsiva Lineare:** una sola chiamata a se stessa
- Un classico esempio e' la funzione fattoriale:

```
long factorial( long n )
{
   if ( n == 0 ) // base case
      return 1;
   else
      return n * factorial(n-1);
}
```

Tipi di ricorsione: Ricorsiva in coda

- **Ricorsiva in coda:** una sola chiamata a se stessa come ultima istruzione eseguita
  - *Nota: ci possono essere altre istruzioni dopo la chiamata basta che questa sia un return*
- Un classico esempio e' la funzione GCD (Greatest Common Divisor):
  - Il GCD di due numeri interi e' l'intero piu' grande che li divide entrambi senza resto

```
int gcd(int m, int n)
{
    if (m < n) return gcd(n,m);
    int r = m%n;
    if (r == 0) return(n);
    else return(gcd(n,r));
}
```

Tipi di ricorsione: Ricorsiva binaria

- **Ricorsiva binaria:** due o piu' chiamate a se stessa
- Un classico esempio e' la funzione di fibonacci:

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )

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

Tipi di ricorsione: Ricorsiva binaria

- Un altro esempio e' la funzione che restituisce il numero di combinazioni con cui prendere n elementi in un insieme di k:

```
int choose(int n, int k)
{
    if (k == 0 || n == k) return(1);
    else return(choose(n-1,k) + choose(n-1,k-1));
}
```

Tipi di ricorsione: Mutuamente ricorsiva

- **Mutuamente ricorsiva:** una prima funzione che ne chiama una seconda che e' definita in termini della prima
- Un esempio e' la funzione che determina se un numero e' pari o dispari: 0 e' pari e se n e' pari allora n-1 e' dispari

```
int is_even(unsigned int n)
{
    if (n==0) return 1;
    else return(is_odd(n-1));
}

int is_odd(unsigned int n)
{
    return (!iseven(n));
}
```

Tipi di ricorsione: Ricorsiva annidata

- **Ricorsiva annidata:** una funzione che ha almeno un argomento che e' una chiamata alla funzione stessa
- Un esempio e' la funzione di Ackerman che cresce (molto) piu' velocemente degli esponenziali

```
int ackerman(int m, int n)
{
    if (m == 0) return(n+1);
    else if (n == 0) return(ackerman(m-1,1));
    else return(ackerman(m-1,ackerman(m,n-1)));
}
```

Vantaggi e svantaggi

- La scrittura di codice ricorsivo e' piu' semplice ed elegante
- Spesso la natura del problema o della struttura dati e' inerentemente ricorsiva e la soluzione ricorsiva e' naturale

- La ricorsione puo' essere meno efficiente dell'iterazione
- La ricorsione viene implementata con chiamate successive alla stessa funzione -> le chiamate tendono a riempire lo stack frame
- Lo spazio dedicato agli stack frame e' minore di quello dedicato alla memoria nell'heap
- Si spreca memoria per memorizzare tutte le variabili locali alla funzione ricorsiva
- La ricorsione in coda si puo' trasformare semplicemente in iterazione (e molti compilatori lo fanno)

Divide et Impera

- Paradigma di programmazione Divide et Impera
- Si sfrutta la soluzione ricorsiva di un problema
- Il procedimento è caratterizzato dai seguenti passi:
  - Divide: suddivisione del problema in sottoprolemi
  - Impera: soluzione ricorsiva dei sottoproblemi. Problemi piccoli sono risolti direttamente.
  - Combina: le soluzioni dei sottoproblemi sono ricombinate per ottenere la soluzione del problema originale

Divide et Impera

- Si continua la divisione in sottoproblemi di taglia minore fino a ottenere un problema che e' facilmente e direttamente risolubile
- In genere si usa un modello ricorsivo ma si possono memorizzare i sottoproblemi in strutture dati apposite come stack, queue o code con priorita'
- Questo offre il vantaggio di una maggiore flessibilita' nella selta di quale sottoproblema affrontare (es. Ricorsione per ampiezza o Branch and Bound)

Il merge sort come esempio

- L'algoritmo merge sort ordina un vettore di elementi nel modo seguente:
  - Divide: ogni sequenza di n elementi da ordinare è suddivisa in 2 sottosequenze di n/2 elementi
  - Impera: ordina (*sort*) ricorsivamente le sottosequenze
  - Combina: fonde (*merge*) le due sottosequenze per produrre la sequenza ordinata
- La condizione di terminazione si ha quando si deve ordinare una sequenza di lunghezza 1 (in quanto già ordinata)

Spiegazione intuitiva

- L’operazione chiave è la procedura di fusione
- L’idea è di fondere fra di loro due sottosequenze già ordinate (in ordine decrescente ad esempio)
  - Si confrontano in ordine gli elementi delle due sottosequenze
  - Si trascrive l’elemento più piccolo
  - Si reitera il procedimento a partire dall’elemento successivo a quello già considerato

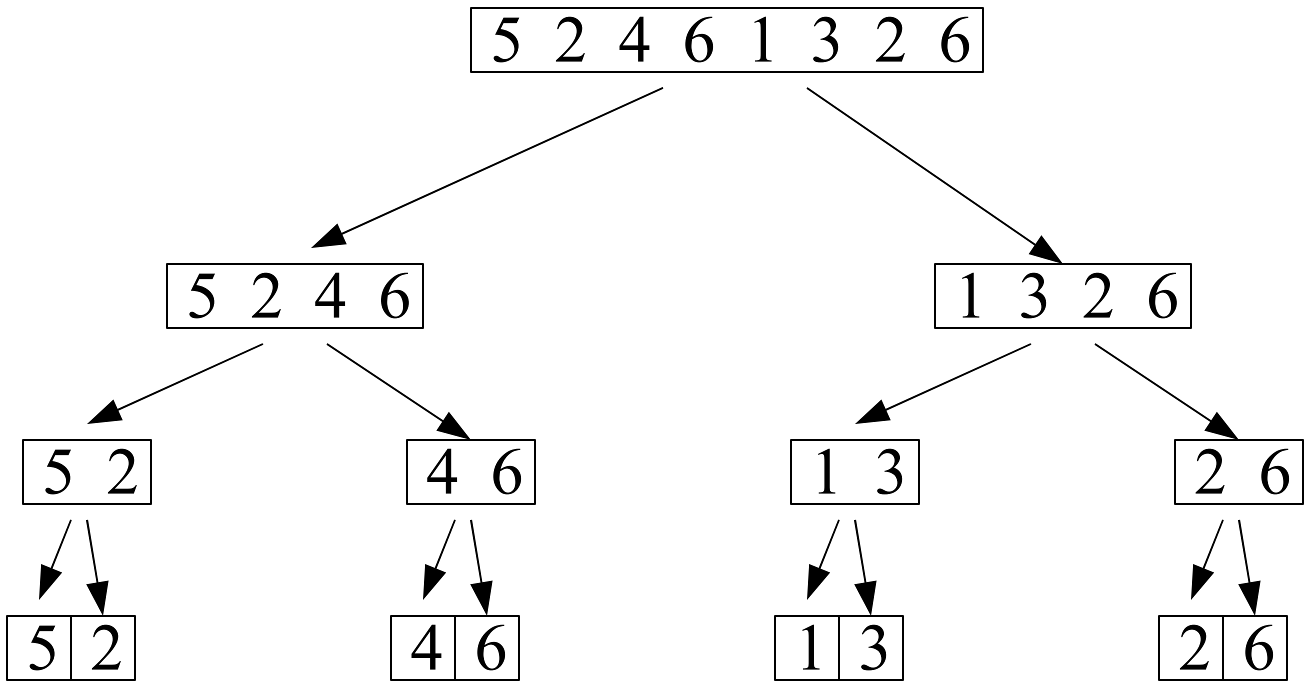
Visualizzazione del concetto di fusione

- Sequenze gia' ordinate:
  - A: 2 4 6 8
  - B: 1 3 5 7
  - R:
- Step di fusione:
- 1)
  - A: 2 4 6 8
  - B: _ 3 5 7
  - R: 1
- 2)
  - A: _ 4 6 8
  - B: _ 3 5 7
  - R: 1 2

- 3)
  - A: _ 4 6 8
  - B: _ _ 5 7
  - R: 1 2 3
- 4)
  - A: _ _ 6 8
  - B: _ _ 5 7
  - R: 1 2 3 4
- 5)
  - A: _ _ 6 8
  - B: _ _ _ 7
  - R: 1 2 3 4 5

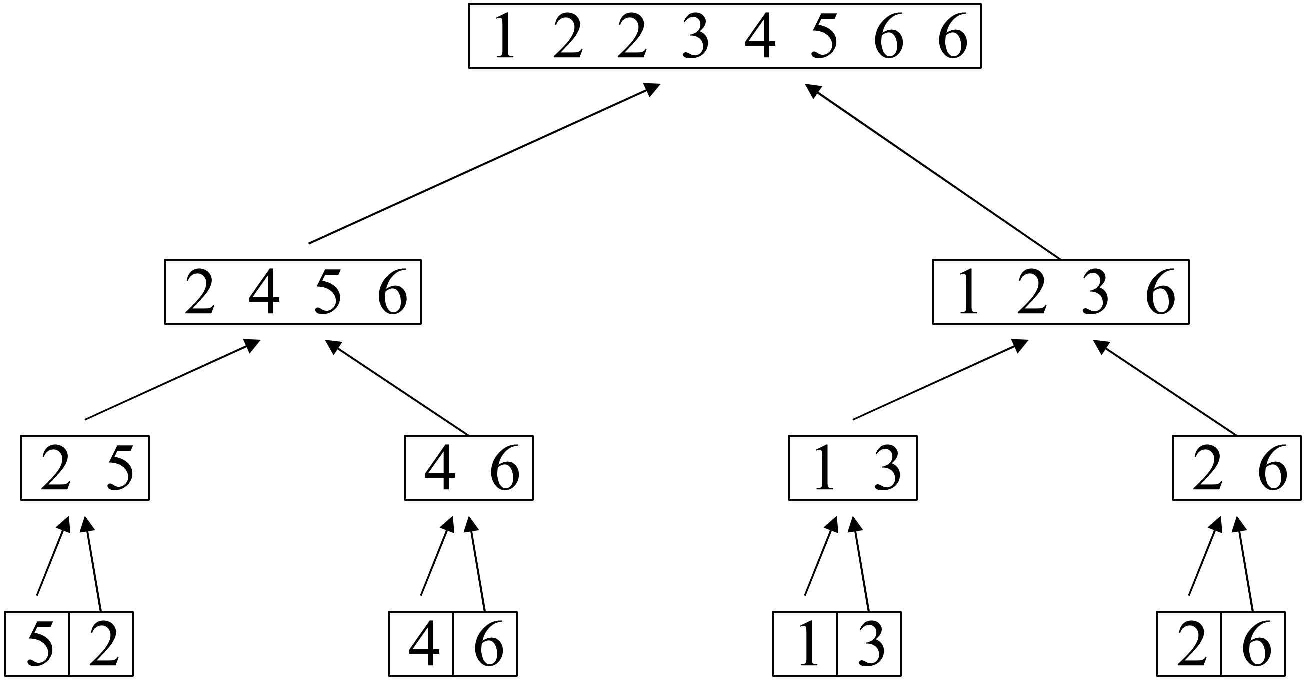
- 6)
  - A: _ _ _ 8
  - B: _ _ _ 7
  - R: 1 2 3 4 5 6
- 7)
  - A: _ _ _ 8
  - B: _ _ _ _
  - R: 1 2 3 4 5 6 7
- 8)
  - A: _ _ _ _
  - B: _ _ _ _
  - R: 1 2 3 4 5 6 7 8

Divide



Sequenza: 5 2 4 6 1 3 2 6

Impera e ricombina



Sequenza: 5 2 4 6 1 3 2 6

Pseudo Codice

```
MERGE-SORT(A,p,r)
if p<r
    then   q←(p+r)/2
           MergeSort(A,p,q)
           MergeSort(A,q+1,r)
           Merge(A,p,q,r)
```

Pseudo Codice (semplificato)

```
MERGE(A,p,q,r)
aux ← A
i ← p
j ← q+1
for k ← p to r
     do    if aux[i]<aux[j]
           then   A[k] ← aux[i++]
           else   A[k] ← aux[j++]
```

Pseudo Codice (caso generico completo)

```
MERGE(A,p,q,r)
aux ← A
i ← p
j ← q+1
for k ← p to r
      do     if i=q+1 ►CASO DI FINE PRIMA SOTTOSEQUENZA
             then   A[k] ← aux[j++] ►SI COPIA II SOTTO-SEQ
             else if j=r+1 ►CASO DI FINE SECONDA SOTTOSEQUENZA
             then   A[k] ← aux[i++] ►SI COPIA I SOTTO-SEQ
             else if aux[i]<aux[j] ►CASO GENERICO
             then   A[k] ← aux[i++]
             else   A[k] ← aux[j++]
```

Il merge sort in C

```
typedef int Item; // tipo di ogni elemento da ordinare

void merge(Item A[], int p, int q, int r)
{
  const int MaxItems=1000;
  static Item aux[MaxItems];
  int i,j;
  for (i = q+1; i > p; i--) aux[i-1] = A[i-1];
  for (j = q; j < r; j++) aux[r+q-j] = A[j+1];//COPIA DECR

  for (int k = p; k <= r; k++)
       if (aux[j] < aux[i]) A[k] = aux[j--];
       else A[k] = aux[i++];
}
```

Nota:

- Si è copiata la seconda sottosequenza in ordine decrescente in modo da poter gestire il caso in cui la sequenza originale è formata da un numero dispari di elementi e quindi una sottosequenza è più piccola dell’altra
- in questo modo non si devono effettuare controlli sugli indici
  - infatti quando l’indice *i* si riferisce ad un elemento della seconda sottosequenza (B) questo risulterà sicuramente ≥ degli elementi di B ancora da elaborare

Il merge sort in C++

```
void mergesort(Item A[], int p, int r) {
if (p<r) { //passo base: quando p==r
   int q = (r+p)/2;
   mergesort(A, p, q);
   mergesort(A, q+1, r);
   merge(A, p, q, r);
   }
}
```

II merge sort in C++

```
int main() {
   const int N=100;
   int A[N];
   for (int i=0; i<N; ++i) A[i] = rand()%1000;

  mergesort(A,0,N-1);

   for (int i=0; i<N; ++i)
        cout << A[i] <<((i+1)%10==0)? endl :" ";
   return 0;
}
```

Il tempo di calcolo del merge sort

- Per semplificare l'analisi:
  - numero di elementi da ordinare sia una potenza di 2
  - così per ogni applicazione del merg sort si lavora con sotto sequenze di lunghezza identica
- Tempo per elaborare sequenze di 1 elemento $\Theta(1)$
- quando si hanno $n>1$ elementi
  - *Divide*: calcolo dell'indice mediano della sequenza $\Theta(1)$
  - *Impera*: si risolvono ricorsivamente due sotto problemi di dimensione $n/2$, quindi $2T(n/2)$
  - *Combina*: il tempo di calcolo per la procedura Merge fra due sotto sequenze di $n/2$ elementi è $\Theta(n)$

Il tempo di calcolo del merge sort

- Si ha pertanto:

  $T(n)= \Theta(1)+2T(n/2)+\Theta(n) = 2T(n/2)+\Theta(n)$

- O più precisamente:

  $T(n)=\Theta(1)$ per $n=1$

  $T(n)= 2T(n/2)+\Theta(n)$

- Vedremo che $T(n)= \Theta(n \lg n)$

Il tempo di calcolo di un algoritmo ricorsivo

- Per risolvere il calcolo di ricorrenze del tipo

$$T(n)=aT(n/b)+f(n)$$

- Si usa il Teorema Principale (non lo dimostreremo)
- Le ricorrenze del tipo indicato descrivono i tempi di calcolo di algoritmi che
  - dividono un problema di dimensione n in *a* sottoproblemi di dimensione *n/b*
  - risolvono i sotto problemi in tempo T(n/b)
  - determinano come dividere e come ricomporre i sottoproblemi in un tempo f(n)

Il Teorema principale

- Sia $T(n)=aT(n/b)+f(n)$ con $a,b>1$ e $f(n)$ asintoticamente positiva
- allora $T(n)$ può essere asintoticamente limitato
  - 1. se $f(n)=O(n^{\log_b a - e})$ per qualche costante e, allora: $T(n)=\Theta(n^{\log_b a})$
  - 2. se $f(n)=\Theta(n^{\log_b a})$ allora: $T(n)=\Theta(n^{\log_b a} \lg n)$
  - 3. se $f(n)=\Omega(n^{\log_b a + e})$ per qualche costante e, e inoltre se

  $af(n/b)<cf(n)$ per qualche $c<1$ e per $n>n_0$, allora: $T(n)=\Theta(f(n))$

Significato intuitivo

- Si confronta f(n) con $n^{\log_b a}$
- La soluzione è determinata dalla più grande fra le due funzioni
- se prevale $n^{\log_b a}$ allora la soluzione è $\Theta(n^{\log_b a})$
  - cioè il tempo è dominato dalla soluzione ricorsiva dei sottoproblemi
- se prevale f(n) allora la soluzione è $\Theta(f(n))$
  - cioè il tempo è dominato dalle procedure di divisione e ricombinazione delle soluzioni parziali
- se hanno lo stesso ordine di grandezza allora la soluzione tiene conto di entrambi i costi $\Theta(f(n)\ln n)$

Nota

- Cosa significa che una funzione *prevale* sull'altra?
- Nel primo caso non solo si deve avere $f(n) < n^{\log_b a}$ ma deve esserlo in *modo polinomiale*, cioè deve esserlo per un fattore n^e per un qualche *e*
- Nel terzo caso $f(n)$ deve essere polinomialmente più grande di $n^{\log_b a}$ e inoltre devono valere le condizioni di regolarità $af(n/b)<cf(n)$ – in pratica ci si deve guadagnare a dividere/ricombinare un numero a di sotto-problemi di dimensione n/b rispetto ad un unico problema di dimensione n

Nota

- ATTENZIONE! Se f(n) è più piccola (o più grande nel terzo caso) di $n^{\log_b a}$ ma non in modo polinomiale allora non si può utilizzare il teorema per risolvere la ricorrenza

Applicazione del Teorema Principale

- $T(n)=2T(n/2)+n$
  - $f(n)=n$, $a=2$, $b=2$
  - $n^{\log_b a} = n^{\log_2 2} = n$
  - pertanto, dato che $f(n)=\Theta(n^{\log_b a})=\Theta(n)$ allora (caso 2)
  - $T(n)=\Theta(n \lg n)$
- $T(n)=9T(n/3)+n$
  - $f(n)=n$, $a=9$, $b=3$
  - $n^{\log_b a} = n^{\log_3 9} = n^2$
  - pertanto, dato che $f(n)=O(n^{\log_b a-1})=O(n^1)$ allora (caso 1)
  - $T(n)= \Theta(n^2)$

Applicazione del Teorema Principale

- $T(n)=T(2n/3)+1$
  - $f(n)=1$, $a=1$, $b=3/2$
  - $n^{\log_b a} = n^{\log_b 1} = n^0 = \Theta(1)$
  - pertanto, dato che $f(n)=\Theta(n^{\log_b a})=\Theta(1)$ allora (caso 2)
  - $T(n)= \Theta(\lg n)$
- $T(n)=3T(n/4)+n \lg n$
  - $f(n)=n \lg n$, $a=3$, $b=4$
  - $n^{\log_b a} = n^{\log_4 3} = n^{0.79..}$
  - pertanto, dato che $f(n)=(n^{\log_b a+e})= \Omega(n^1)$ allora
  - se f(n) è regolare siamo nel caso 3
  - $af(n/b)<cf(n)$ ovvero $3\, n/4 \lg n/4 < c\, n \lg n$, ok se $c=3/4$
  - $T(n)= \Theta(n \lg n)$

Applicazione del Teorema Principale

- $T(n)=2T(n/2) + n \lg n$
  - $f(n)=n \lg n$, $a=2$, $b=2$
  - $n^{\log_b a} = n^{\log_2 2} = n$
  - pertanto, dato che f(n) è più grande di n ovvero $f(n)=(n^{\log_b a+e})= \Omega(n^1)$ allora
  - se f(n) è polinomialmente maggiore allora potremo essere nel caso 3
  - deve essere $f(n) > n^{\log_b a+e}$
  - ovvero $f(n) / n^{\log_b a} > n^e$ per qualche *e*
  - ma $f(n)/ n^{\log_b a} = (n \lg n)/n = \lg n$
  - dato che lg n è sempre minore di n^e (per qualsiasi *e)*
  - allora non possiamo usare il Teorema Principale