Corso di Laurea

Ingegneria Gestionale

Lezione 6

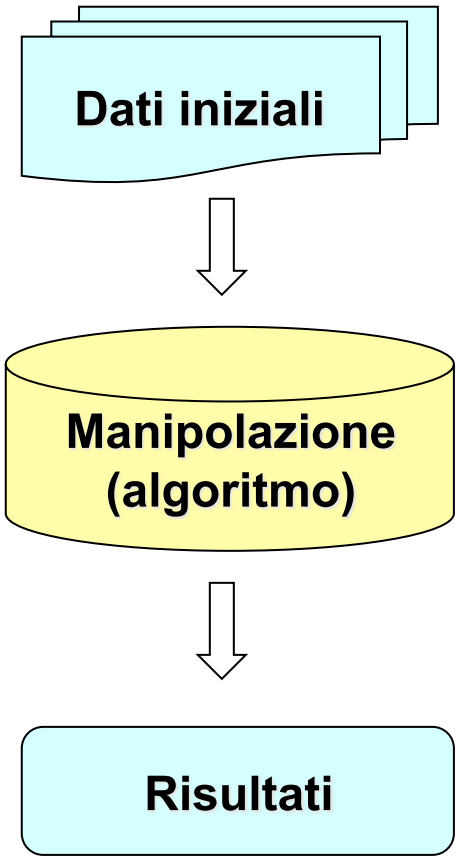
Algoritmi

Sistemi Informativi Aziendali

a.a. 2008/2009

Definizione di algoritmo

- Un algoritmo è l'insieme delle istruzioni che definiscono una sequenza di operazioni mediante le quali si risolve un problema
- Un programma traduce un algoritmo in una sequenza di istruzioni, scritte in un opportuno linguaggio di programmazione, comprensibile al calcolatore (*esecutore astratto*)



Caratteristiche degli algoritmi

- Un algoritmo deve essere:
  - finito (cioè consistere di numero *limitato* di passi da compiere in un tempo *finito*);
  - definito (ogni istruzione deve consentire un'interpretazione *univoca*);
  - eseguibile (la sua esecuzione deve essere possibile con gli strumenti a disposizione);
  - deterministico (ad ogni passo deve essere definita una operazione successiva)
  - comprensibile al suo esecutore
  - corretto
  - efficiente

Analisi della complessità degli algoritmi

- Un programma è tanto più efficiente quanto più basso è il suo costo (o complessità) computazionale
- La valutazione dell'efficienza del software è dunque legata alla individuazione della quantità di *risorse* computazionali richieste da un programma per la sua esecuzione
- Le risorse richieste per l'esecuzione di un programma sono (tradizionalmente) la CPU e la memoria centrale
- La complessità è quindi valutata in relazione al *tempo* (di esecuzione o di CPU) e allo *spazio* (cioè alla quantità di memoria centrale impegnata)

Analisi della complessità degli algoritmi

- La complessità spaziale viene tuttavia di norma considerata in subordine rispetto alla complessità temporale: il tempo è da privilegiare in quanto *risorsa non riutilizzabile*
- I bassi costi attuali di realizzazione fisica delle memorie e le elevate capacità di indirizzamento dei moderni elaboratori rendono peraltro meno significativa la valutazione dell'elemento spazio, tranne che in contesti specialistici e per applicazioni avanzate (sistemi embedded, architetture specializzate, etc.)

Analisi della complessità degli algoritmi

- Perché si parla di complessità o costo o efficienza degli algoritmi, e non dei programmi?
- Intuitivamente, un programma è più efficiente di un altro se la sua esecuzione richiede meno tempo
- I fattori che influenzano il tempo di esecuzione sono:
  - la velocità di esecuzione della macchina, intesa come esecutore astratto del linguaggio di programmazione adoperato, e quindi sia la potenza di calcolo fornita dall'hardware sia la qualità del compilatore (o dell'interprete) sia l'interazione con il sistema operativo
  - la quantità di dati in ingresso (dimensione) e (spesso) il modo in cui sono organizzati: è intuitivo che ordinare 10 numeri interi è meno costoso che ordinarne 1000000

Analisi della complessità degli algoritmi

Se l'esecuzione di un programma, avente i dati di ingresso *d*, su una macchina *M* richiede un tempo

$$t = f(d)$$

lo stesso programma, su una macchina *M'*, avrà un tempo di esecuzione

$$t' = c \cdot f(d)$$

Anche se il valore di *c* può essere notevole (e questo giustifica le differenze di costo tra elaboratori di classe diversa), la valutazione dell'efficienza di un programma può essere dunque effettuata , a meno di un fattore costante, prescindendo dalle caratteristiche della macchina e del software di base

Analisi della complessità degli algoritmi

- Per quanto riguarda la dipendenza dai dati, come detto, è naturale aspettarsi che il tempo di esecuzione di un programma aumenti al crescere della *dimensione del problema* (o *dimensione dell'input*), che in genere può essere espressa attraverso uno o più parametri
- La dimensione del problema corrisponde spesso alla *cardinalità* dei dati di ingresso, come il numero di elementi di un vettore, il numero di righe di una tabella, il numero di parole di un testo, etc.

Analisi della complessità degli algoritmi

- Per esempio:
  - nel caso dell'ordinamento di una sequenza di dati, la dimensione corrisponde al numero di valori nella sequenza,
  - nel caso della somma o della moltiplicazione di numeri interi, la dimensione dell'ingresso è data dal numero di cifre con cui sono rappresentati gli operandi

Analisi della complessità degli algoritmi

- Le considerazioni precedenti permettono di definire un approccio alla analisi approssimata della complessità asintotica degli algoritmi
- La misura di complessità può essere espressa formalmente utilizzando la cosiddetta notazione *O*
- Si tratta di un'analisi teorica che risulta più affidabile di quelle sperimentali, perché permette di stabilire risultati validi per *tutte* le possibili istanze di dati di ingresso al programma

Analisi della complessità degli algoritmi

- Essa risulta inoltre indipendente dalla macchina e dal linguaggio di programmazione, e, anche sulla base delle semplificazioni che saranno introdotte, risulta orientata agli algoritmi piuttosto che ai programmi che li implementano
- L'analisi teorica permette anzi di predire le prestazioni del software, prima ancora di scrivere linee di codice
- Va ribadito che si prendono in considerazione solo elaboratori di tipo tradizionale (noti anche come *Random Access Machine* o RAM)

Modello di costo

- Il modello di costo adottato fa cioè riferimento a una CPU mono-processore in cui tutte le istruzioni sono eseguite una dopo l'altra, senza alcuna forma di parallelismo
- L'accesso ad ogni cella di memoria avviene in tempo costante
- Il calcolo del tempo complessivo impiegato per l'esecuzione di tutte le istruzioni dell'algoritmo è effettuato evidenziando la sua dipendenza dalla dimensione dei dati di ingresso, *n*

Modello di costo

- A tal fine, vengono convenzionalmente considerate di costo unitario, in termini di tempo impiegato per la loro esecuzione (passi base), le istruzioni *semplici* quali:
  - Lettura
  - Scrittura
  - Assegnazione
  - Valutazione di una condizione logica comunque complessa
- Si assume dunque che il tempo di esecuzione delle istruzioni semplici non dipenda dai valori e dai tipi di dati delle variabili né dall'effettiva complessità matematica delle operazioni, e quindi dal numero di istruzioni macchina effettivamente eseguite

Modello di costo

- Sono invece considerate operazioni di costo *non* unitario:
  - Istruzione composta: somma dei costi delle istruzioni componenti
  - Istruzione ciclica a condizione: costo della condizione più costo totale del corpo del ciclo
  - Istruzione ciclica a conteggio: costo totale della testata del ciclo più costo totale del corpo del ciclo
  - Istruzione condizionale: costo della condizione più costo del ramo *then* (se la condizione è vera) oppure costo del ramo *else* (se la condizione è falsa)
  - Attivazione di modulo: costo di tutte le istruzioni che lo compongono (eventualmente tenendo conto di altre attivazioni al suo interno)

Analisi della complessità degli algoritmi

- In base alla teoria della complessità (sviluppata a partire dagli anni '70 del secolo scorso), il tempo di esecuzione di molte importanti classi di algoritmi non dipende dunque dalla tecnologia, e non risulta significativamente ridotto dal progresso tecnologico:
  - Algoritmi di ricerca (approssimativamente n passi base)
  - Algoritmi di ordinamento (approssimativamente n^2 passi base)
  - Algoritmi decisionali (approssimativamente 2^n passi base)
- E' importante notare quanto ai fini pratici possa essere importante la complessità dell'algoritmo, come mostra la seguente tabella

Analisi della complessità degli algoritmi

- Si supponga di avere, per uno stesso problema, più algoritmi di diversa complessità, e che un passo base venga eseguito in un μsec (10^{-6} sec).

Compl.\ Dimens	n =10	n =100	n =1000
log n	3.32 μsec	6.64 μsec	9.96 μsec
n	10 μsec	100 μsec	1 msec
n^2	100 μsec	10 msec	1 sec
n^3	1 msec	1 sec	1000 sec
2^n	1.024 msec	10 milioni di volte l'età della terra	----

La notazione O

In termini matematici, si dice che una funzione $f(n)$ è di ordine $g(n)$ e si scrive:

$$f(n) = O(g(n))$$

se esiste una costante numerica C positiva tale che valga, salvo al più per un numero finito di valori di n, la seguente condizione:

$$f(n) \leq C * g(n)$$

cioè il rapporto $f(n)/g(n)$ è *limitato*, per $g(n)$ non identicamente nulla:

$$\lim_{n \to \infty} (f(n) / g(n)) = C$$

La notazione O

Esempi (n intero positivo):

$2n+5 = O(n)$ poichè $2n+5 <= 7n$ per ogni n

$2n+5 = O(n^2)$ poichè $2n+5 <= n^2$ per $n >= 4$,

$2n+5 = O(2^n)$ poichè $2n+5 <= 2^n$ per $n >= 4$

Invece:

$n^2 \neq O(n)$ poichè $n^2 / n = n$ non limitato

$e^n \neq O(n)$ poichè $e^n / n >= n^2 / n = n$ non limitato

Si ricordi che O misura la complessità *asintotica*

La notazione O

- Si dice che $f(n)$ ha complessità asintotica $g(n)$ se valgono le seguenti condizioni:
  - a) $f(n) = O(g(n))$
  - b) $g(n)$ è tra tutte le funzioni che soddisfano la condizione (a) quella di ordine inferiore
- Esempi:
  - $f(n)$ = cost. → complessità asintotica $O(1)$ (*costante*)
  - $f(n) = 2n + 5$ → complessità asintotica $O(n)$ (*lineare*)
  - $3n^2 + 5n$ → complessità asintotica $O(n^2)$ (*quadratica*)
  - $4 * 2^n$ → complessità asintotica $O(2^n)$ (*esponenziale* di base 2)
  - $3^n + 2n + 5^n$ → complessità asintotica $O(5^n)$ (*esponenziale* di base 5)
  - $2n + 5n + n!$ → complessità asintotica $O(n!)$ (*fattoriale*)

Algoritmi di ricerca

Algoritmi di ricerca

- Da ora in poi si assume che la struttura dati di riferimento sia un array di *n* elementi
- Un array è, come è noto, una struttura dati statica e omogenea, i cui elementi, cioè, non variano di numero durante l'esecuzione e sono tutti dello stesso tipo
- L'accesso a ciascun elemento di un array è *diretto*

Algoritmi di ricerca

- Problema della ricerca: dati un array e un valore, stabilire se il valore è contenuto in un elemento dell'array, riportando in caso affermativo l'indice di tale elemento, e in caso negativo un valore convenzionale, per esempio -1

Algoritmi di ricerca

- Una prima soluzione è data dall'algoritmo di ricerca lineare: gli elementi dell'array vengono attraversati uno dopo l'altro nell'ordine in cui sono memorizzati, finché il valore cercato non viene trovato o la fine dell'array non viene raggiunta
- Supponendo che il primo elemento dell'array *A* di *n* elementi corrisponda all'indice di valore 0, e che il valore cercato sia memorizzato nella variabile *key*, la rappresentazione in pseudo-codice di una prima versione, volutamente ridondante, dell'algoritmo è la seguente (*pos* rappresenta la risposta dell'algoritmo)

Algoritmi di ricerca

Algoritmo firstLinearSearch

```
pos = -1
for (i = 0; i <= n-1; i++)
   if (a[i] == key) then pos = i
          else pos = pos
endfor
return pos
```

la complessità asintotica è $O(n)$ *(il numero totale di passi base è $4*n + 4$)*

Algoritmi di ricerca

- Pertanto, se la dimensione del problema viene raddoppiata, si avrà un tempo di esecuzione "quasi" doppio, non esattamente doppio per la presenza delle costanti
- Per valori abbastanza grandi di *n*, è intuibile che la dipendenza lineare dai dati contribuisca al tempo di esecuzione molto più della dipendenza dalla macchina
- Si ritrova dunque per la complessità asintotica il risultato già determinato in precedenza, $O(n)$

Algoritmi di ricerca

Una possibile implementazione dell'algoritmo in forma di metodo Java è la seguente:

```
public int firstLinearSearch( int a[], int key )
{
int n = a.length;  // n.ro elementi dell'array
int pos = -1;  // valore da restituire se elemento non trovato
for (int counter = 0; counter < n; counter++)
   if ( a[counter] == key ) pos = counter; // elemento trovato
      else pos = pos;     // solo per bilanciare il n.ro di istruzioni eseguite
return pos
}
```

E' facile verificare che anche l'analisi di tale frammento di programma porterebbe alle medesime conclusioni

Algoritmi di ricerca

- Come si è già notato, il costo di un algoritmo può dipendere non solo dalla dimensione dei dati, ma anche dai particolari valori dei dati stessi, cioè dalla loro *configurazione*
- Ciò risulta anche dall'esempio seguente, che costituisce peraltro una implementazione più realistica della ricerca lineare
- La scansione dell'array viene infatti interrotta non appena si trova l'elemento cercato, se esiste
- Si assume pertanto che siano necessari *m* confronti

Algoritmi di ricerca

Algoritmo otherLinearSearch

```
for (i = 0; i <= n-1; i++)
  if (a[i] == key) then return i
endfor
return -1
```

L'espressione è analoga a quella ricavata nel caso precedente, ma *m* coincide con *n* solo se l'elemento cercato è l'ultimo della sequenza, oppure se esso non si trova nella sequenza

Il numero totale di passi base è $3*m + 3$

Algoritmi di ricerca

- In tutti gli altri casi, *m* è minore di *n*, e in alcuni casi molto minore (per esempio, se la sequenza contiene 10000 elementi e l'elemento cercato è il primo)
- Pertanto il tempo di esecuzione della ricerca dipende non solo dalla dimensione dei dati, ma anche dalla esistenza e dalla posizione dell'elemento cercato, cioè dalla distribuzione dei dati nell'array e dal loro valore, quindi dalla *configurazione* dei dati
- Riassumendo:
  - Se *key* non è presente nell'array, $m = n$
  - Se *key* è nell'array, *m* dipende dalla posizione di *key*:
    - Se *key* è il primo elemento dell'array, $m = 1$
    - Se *key* è l'ultimo elemento dell'array, $m = n$
    - In tutti gli altri casi, *m* è comunque minore di *n*

Algoritmi di ricerca

- Pertanto il numero totale di passi base non può essere in nessun caso superiore a $3*n + 3 \rightarrow O(n)$
- L'analisi effettuata mostra anche che l'algoritmo di ricerca lineare ha un caso migliore, quello in cui l'elemento cercato è il primo dell'array, per il quale la complessità è $O(1)$, e un caso peggiore, quello in cui l'elemento cercato non è presente nell'array o è l'ultimo, per il quale la complessità è appunto $O(n)$

Algoritmi di ricerca

- Oltre al caso migliore e al caso peggiore, spesso è di interesse il caso medio
- La stima del costo nel caso medio si ottiene valutando la media aritmetica dei costi rispetto a tutti i possibili dati di ingresso
- L'esempio conferma anche come ai fini della valutazione dell'efficienza di un algoritmo alcune operazioni sono più rilevanti di altre
- Nel caso della ricerca lineare, sono rilevanti in particolare i confronti tra il valore cercato e gli elementi dell'array

Algoritmi di ricerca

- Per l'algoritmo di ricerca lineare, il caso medio (solo se la ricerca ha esito positivo) richiede $(n + 1)/2$ confronti, se si assume che tutti gli elementi dell'array possano essere cercati con uguale probabilità (distribuzione *uniforme* dei valori)

Algoritmi di ricerca

- Applicando il criterio del caso peggiore alla ricerca lineare, si ottiene che i due algoritmi *firstLinearSearch* e *otherLinearSearch* sono equivalenti dal punto di vista dell'efficienza di esecuzione
- Però è intuitivo che di norma *otherLinearSearch* debba comportarsi meglio di *firstLinearSearch*, e l'analisi del caso medio lo conferma
- In generale, l'analisi del caso medio può fornire informazioni aggiuntive di grande utilità e pertanto nel seguito si farà riferimento al caso peggiore e, quando necessario, al caso medio

Algoritmi di ricerca

- Una seconda soluzione, *molto* più efficiente, al problema della ricerca è fornita dall'algoritmo di ricerca binaria o dicotomica
- L'algoritmo è applicabile se la struttura dati che contiene la sequenza di elementi è accessibile in modo diretto (come nel caso di un array) e se gli elementi sono *ordinati* secondo un certo criterio, per esempio numeri interi in ordine non decrescente
- In questo caso, l'algoritmo può procedere dimezzando ad ogni iterazione lo spazio di ricerca

Algoritmi di ricerca binaria

- L'algoritmo può essere illustrato informalmente nel modo seguente
  1. Si esegue un primo confronto tra l'elemento cercato e l'elemento *mediano* della sequenza. L'elemento mediano di una sequenza ordinata di *n* elementi è quello di posto $\lceil n/2 \rceil$. Se *n* è pari, l'elemento mediano è per convenzione quello di posto $n/2$
  2. Se l'elemento mediano coincide con l'elemento cercato, la ricerca termina con successo. Altrimenti si tratta uno dei due casi seguenti:
     a. L'elemento mediano è maggiore dell'elemento cercato: ciò implica che, se l'elemento cercato esiste nella sequenza, non può che trovarsi nella sua prima metà
     b. L'elemento mediano è minore dell'elemento cercato: ciò implica che, se l'elemento cercato esiste nella sequenza, non può che trovarsi nella sua seconda metà

Algoritmi di ricerca binaria

- In ogni caso, la ricerca continua restringendo l'analisi alla sola metà della sequenza potenzialmente in grado di contenere l'elemento cercato, e ripetendo i passi 1 e 2 fino alla individuazione dell'elemento cercato o finché la porzione di sequenza indagata si riduce ad un solo elemento che non è quello cercato
- Poiché ad ogni iterazione la lunghezza della sequenza in cui effettuare la ricerca si dimezza, il numero di confronti da effettuare, nel caso peggiore che è quello di ricerca infruttuosa, è dell'ordine di $\log_2 n$, quindi di un ordine di grandezza *inferiore* a quello della ricerca lineare.

Algoritmi di ricerca

Supponendo che *key* sia il valore cercato in un array *a* di *n* elementi, e indicando con *low*, *high* e *middle* i valori degli indici del primo elemento della porzione di array presa in considerazione, dell'ultimo e del mediano, si può utilizzare il seguente

Algoritmo BinarySearch

```
low = 0;  // indice basso della porzione di array analizzata
high = n - 1; // indice alto della porzione di array analizzata
while ( low <= high ) // ciclo di analisi, termina quando low > high
    middle = ( low + high ) / 2; // indice dell'elemento di mezzo
    if ( key == a[middle] ) then return middle; // elemento trovato
    else if ( key < a[middle] ) then high = middle – 1; //prima metà
            else low = middle + 1; // seconda metà
endwhile  // fine ciclo di analisi
return –1   // elemento non trovato
```

Algoritmi di ordinamento

Algoritmi di ordinamento

- L'ordinamento dei dati è una delle applicazioni informatiche basilari in pressoché tutti i contesti applicativi. Il suo scopo è anche quello di facilitare le ricerche dei dati all'interno di insiemi di dati ordinati
- In generale i dati possono essere molto complessi, per esempio si può trattare di oggetti caratterizzati da molti attributi. Si chiama chiave l'attributo utilizzato per l'ordinamento

- Informalmente, possono essere ordinati gli insiemi di elementi tra i quali possa essere stabilita una *relazione d'ordine totale*, ossia tale che per ogni coppia di elementi si possa dire che uno è minore od uguale (oppure maggiore od uguale) all'altro

Algoritmi di ordinamento

Per esempio, dato l'insieme degli abitanti dell'Italia e la relazione "abita nella stessa città“, la relazione non è di ordine

Invece, l’insieme dei numeri naturali può essere ordinato in base alla relazione “è maggiore di“, che è una relazione di ordine totale

Problema dell’ordinamento: dato un insieme di elementi contenenti una *chiave d'ordinamento* e data una relazione d'ordine totale sul dominio delle chiavi, determinare una permutazione degli elementi tale che nella nuova disposizione degli elementi i valori delle chiavi soddisfino la relazione d’ordine

Algoritmi di ordinamento

Dato pertanto l'insieme di elementi

$$a_1, a_2, \ldots, a_n$$

e la relazione d'ordine totale <= , ordinare significa individuare una permutazione degli elementi

$$a_{k1}, a_{k2}, \ldots, a_{kn}$$

stabilita dalla funzione di ordinamento *f* che realizza l'ordine tra le chiavi:

$$f(a_{k1}) <= f(a_{k2}) <= \ldots <= f(a_{kn})$$

Si noti che il simbolo <= è qui utilizzato, secondo la consuetudine, per denotare una relazione d'ordine generica, anche quando quest'ultima nulla ha a che fare con quella usata di solito per gli insiemi numerici

Algoritmi di ordinamento

- Nel seguito si suppone che le chiavi siano numeri interi (trascurando gli eventuali altri attributi degli elementi) e la relazione d'ordine totale sia “minore o uguale di”. La struttura dati di riferimento è ancora l’array

- Un metodo di ordinamento è detto *stabile* se mantiene inalterato l'ordine relativo dei dati aventi chiavi uguali, caratteristica spesso desiderabile se gli elementi sono già ordinati secondo una chiave secondaria
- Un algoritmo di ordinamento è detto *in loco* oppure *in place* quando è in grado di trasformare la struttura dati utilizzando soltanto un piccolo e costante spazio supplementare di memoria

Algoritmi di ordinamento

- Le prestazioni degli algoritmi di ordinamento sono normalmente confrontate considerando operazioni dominanti quali il numero di confronti fra gli elementi ed il numero di trasposizioni di elementi
- I metodi di ordinamento più semplici da capire e implementare sono i metodi di ordinamento diretto, che però nel caso peggiore sono $O(n^2)$, essendo n il numero di elementi da ordinare. Sono pertanto adatti ad ordinare sequenze di elementi di piccola dimensione

Algoritmi di ordinamento

- Tra i metodi di ordinamento diretto si ricordano il metodo di ordinamento per selezione, il metodo di ordinamento per inserzione lineare e il metodo di ordinamento a bolle o bubblesort
- Prestazioni migliori offrono alcuni algoritmi di ordinamento non diretto, tra i quali si ricorda il metodo di ordinamento veloce o quicksort, che nel caso *medio* è $O(n*\log n)$

Algoritmi di ordinamento

Ordinamento per selezione (selectionSort)

Si procede per iterazioni successive

Durante il generico passo *i*, l'array *a* viene suddiviso in una *sequenza di destinazione* *a*[0], …, *a*[*i* - 1] già ordinata e in una *sequenza di origine* *a*[*i*],…, a[*n* - 1] ancora da ordinare

A tal fine, si seleziona l'elemento di valore minimo della sequenza di origine individuata al passo precedente e lo si scambia con il valore contenuto in *a*[*i* -1]

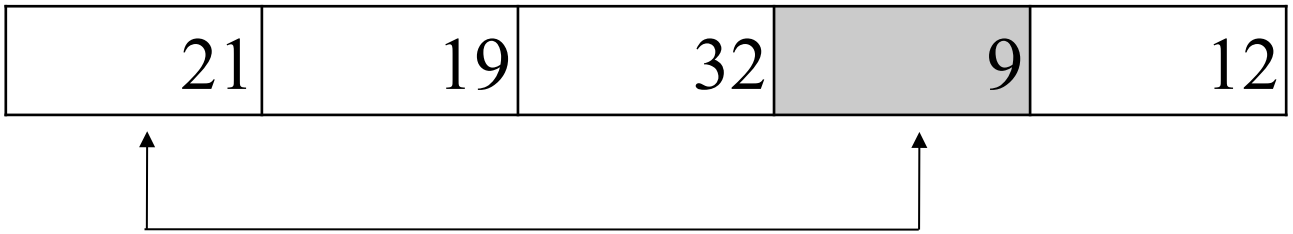
Ad ogni passo, la sequenza di origine risulta quindi ridotta di un elemento, e quella di destinazione accresciuta di un elemento. Si procede allo stesso modo fino al completo ordinamento dell'array

Algoritmi di ordinamento

Si supponga per esempio di dover ordinare il seguente array:

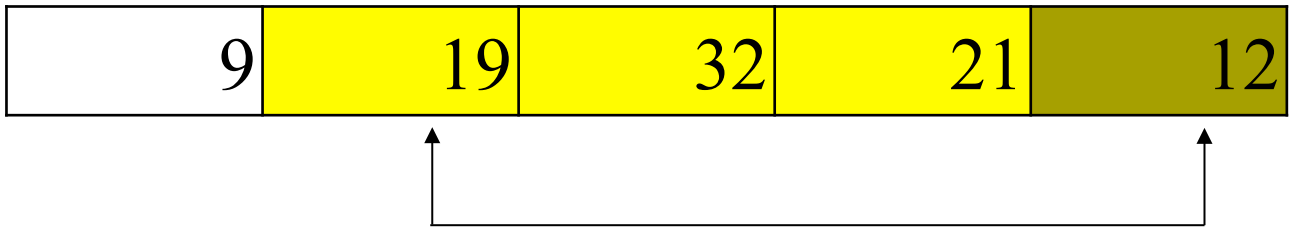
21	19	32	9	12

Durante il primo passo dell'algoritmo, la sequenza di origine coincide con l'intero array, mentre la sequenza di destinazione è ancora vuota, quindi occorre trovare il minimo degli elementi dell'array e scambiarlo con *a*[0]:



Algoritmi di ordinamento

Durante il secondo passo, la sequenza di destinazione precedente coincide con il primo elemento dell'array, che adesso è il più piccolo, mentre la sequenza di origine, all'interno della quale bisogna cercare il minimo, da scambiare con *a*[1], è costituita dagli altri elementi (segnati in giallo):



Analogamente si procede per i passi successivi

Il numero complessivo dei passi da eseguire è pari alla lunghezza *n* dell'array meno 1

Algoritmi di ordinamento

- Il seguente metodo *selectionSort* implementa l'ordinamento per selezione di un array
- Esso si serve di due altri metodi:
  - *minPos* per trovare la posizione del minimo di una parte dell'array (cui vanno passati come argomenti l'array e la posizione da cui cominciare la ricerca del minimo)
  - *swap* capace di scambiare di posto due elementi di un array (cui vanno passati come argomenti l'array e gli indici degli elementi da scambiare)

L'algoritmo SelectionSort

```
for ( int i=0; i < a.length-1; i++ )
    {
    //trova il minimo nell'array, da i alla fine
    // con i che varia da 0 alla lunghezza-1 di a
    mPos = minPos(a, i);
    // scambia gli elementi
    swap(a, mPos, i);
    }
```

Algoritmi di ordinamento

```
public void selectionSort (int[] a)
{
   for ( int i=0; i < a.length-1; i++ )
      {
        int mPos = minPos(a, i); //trova il minimo nell'array,
da i alla fine
        swap(a, mPos, i);  // scambia gli elementi
      }
 }//fine metodo selectionSort
public int minPos (int[] a, int from)
{
   int posMinimo=from;
   for (int i=from+1; i< a.length; i++)
   if ( a[i] < a[posMinimo]) posMinimo=i;
      return posMinimo;
}//fine metodo minPos
```

Continua →

Algoritmi di ordinamento

```
public void swap (int[] v, int i, int j)
{
    int temp=v[i];
    v[i]=v[j];
    v[j]=temp;
}// fine metodo swap
```

Ad ogni passo l'algoritmo effettua la ricerca del minimo di una porzione di array, seguita dallo scambio tra due elementi dell'array

La complessità asintotica dell'algoritmo di ordinamento per selezione può essere analizzata individuando eventuali operazioni dominanti

Algoritmi di ordinamento

Mentre quest'ultima operazione (scambio di 2 elementi) ha un tempo di esecuzione costante, la ricerca del minimo richiede l'esecuzione di un ciclo for, il cui corpo è a sua volta basato sul confronto fra due elementi dell'array

Questo confronto può pertanto essere considerato come l'operazione dominante dell'intero algoritmo

Basta quindi determinare quante volte viene eseguito il confronto fra due elementi dell'array

Se l'array ha *n* elementi, l'algoritmo esegue *n* – 1 passi, come si è visto

Algoritmi di ordinamento

Durante il primo passo, la porzione di array da visitare per la ricerca del minimo è di n elementi, e il numero di confronti da effettuare è n - 1

Durante il secondo passo occorre invece eseguire n – 2 confronti per trovare il minimo tra gli n – 1 elementi rimasti nella sequenza di origine individuata al passo precedente

Durante il k-simo passo si eseguono n – k confronti per trovare il minimo di n – k + 1 elementi

Algoritmi di ordinamento

- Il numero totale di confronti da effettuare è dunque: $confr = \sum_{k=1}^{n-1}(n-k) = n(n-1) - \sum_{k=1}^{n-1} k$
- Poiché $\sum_{j=1}^{m} j = \frac{m(m+1)}{2}$
- si ottiene: $confr = n(n-1) - \frac{(n-1)n}{2} = \frac{n(n-1)}{2}$

Algoritmi di ordinamento

Il numero totale di confronti è pertanto dell'ordine di n^2, per cui la complessità asintotica di questo algoritmo è $O(n^2)$ e non dipende dalla disposizione iniziale dei valori negli elementi dell'array

L'ordinamento per selezione ha quindi complessità $O(n^2)$ in ogni caso, in particolare anche se l'array fosse inizialmente già ordinato

Algoritmi di ordinamento

Ordinamento per inserzione lineare

Anche in questo caso, al generico passo *i* l'array è suddiviso in una *sequenza di destinazione* *a*[0], …, *a*[*i* - 1] già ordinata e in una *sequenza di origine* *a*[*i*],…, *a*[*n* - 1] ancora da ordinare

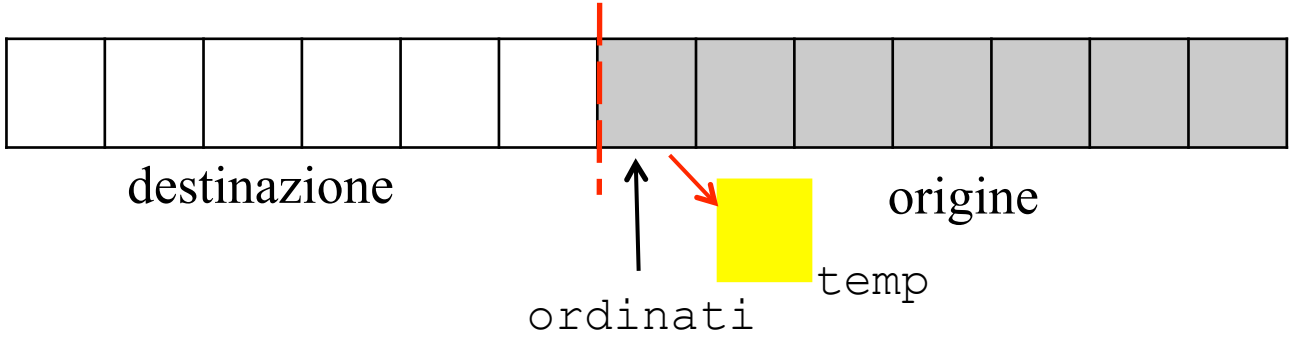
Il valore contenuto in *a*[*i*] viene inserito *al posto giusto* nella sequenza di destinazione facendolo scivolare a ritroso, in modo da ridurre di un elemento la sequenza di origine

L'algoritmo termina quando si esaurisce la sequenza di origine

Algoritmi di ordinamento

Il funzionamento dell'algoritmo può essere così illustrato

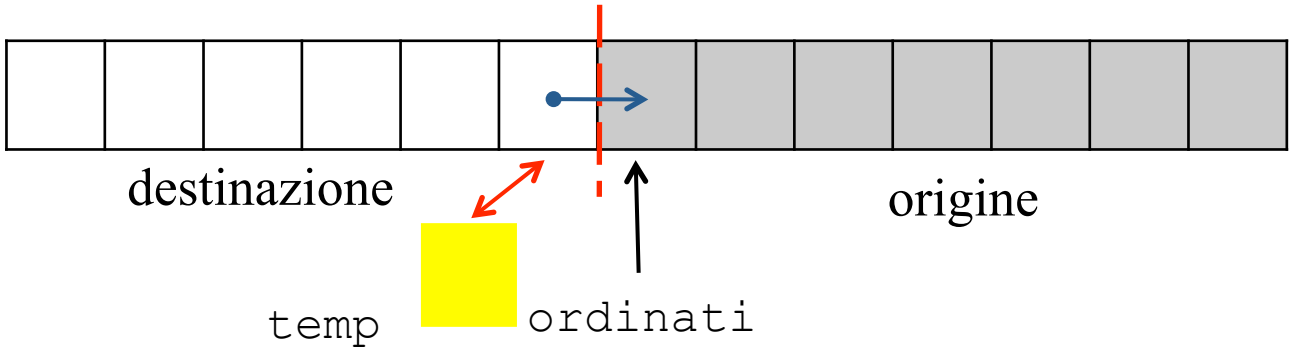
All'inizio dell'esecuzione di un generico passo la situazione sia la seguente:



Il riferimento `ordinati` punta al primo elemento della sequenza di origine. Tale elemento viene memorizzato nella variabile `temp`

Algoritmi di ordinamento

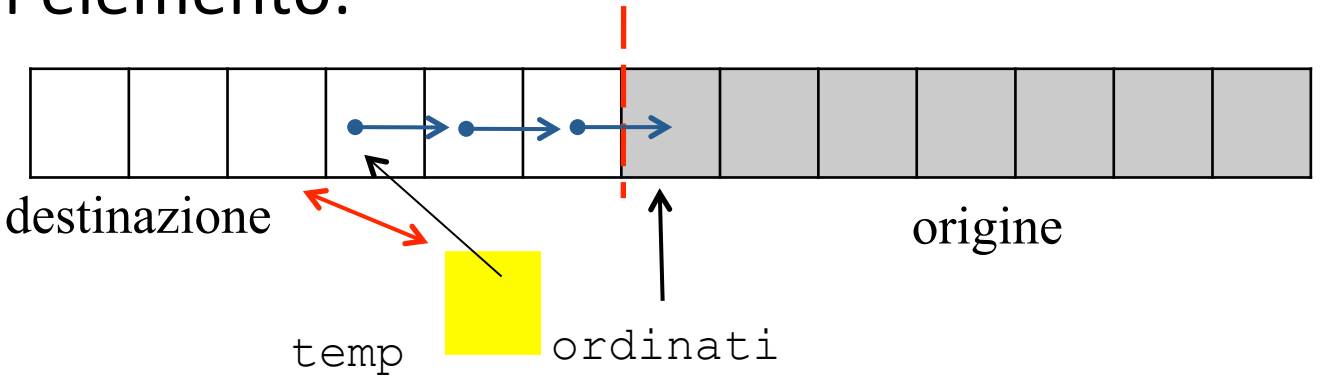
La variabile `temp` è confrontata con ciascun elemento della sequenza di destinazione, a partire dall'ultimo:



Se `temp` è minore dell'elemento, quest'ultimo viene spostato verso destra di una posizione, e si passa a confrontare `temp` con l'elemento successivo verso sinistra

Algoritmi di ordinamento

Non appena si trova un elemento della sequenza di destinazione di valore minore di `temp`, `temp` viene copiato nella posizione immediatamente a destra dell'elemento:



Si sposta `ordinati` di una posizione verso destra, e si inizia il passo successivo

Ad ogni passo, dunque, la sequenza di destinazione cresce di un elemento, e quella di origine diminuisce di un elemento, fino al completo ordinamento

Algoritmi di ordinamento

Il seguente metodo *insertionSort* implementa l'ordinamento per inserzione lineare di un array

```
public void insertionSort (int[] v)
{
 int i;      // scandisce i dati nella parte ordinata
 int ordinati;     //punta al primo elemento della parte
     // non ordinata (n.ro elementi ordinati)
 boolean ins;      //indica se e' possibile inserire
     //l'elemento fra quelli ordinati
 for ( ordinati=1; ordinati < v.length; ordinati ++)
 {
   temp=v[ordinati];
   ins=false;
   i=ordinati;     // i punta al primo elemento
    // della parte non ordinata
```

Algoritmi di ordinamento

```
        while (!ins && i>0) //si scorre la parte ordinata
          if (temp < v[i-1])
              {
               v[i]=v[i-1]; //si sposta l'elemento a destra
               i--;
              }
          else
              {
                ins=true;   //altrimenti si inserisce
                            //al posto dell'elemento corrente
                v[i]=temp;
              }
    } // fine for
}
```

Algoritmi di ordinamento

Per la determinazione della complessità asintotica dell'algoritmo, si osservi che il ciclo esterno viene eseguito $n - 1$ volte, se n è la lunghezza dell'array

Invece il numero di iterazioni del ciclo interno dipende da `temp` e dai `v[i]`, mentre il tempo di esecuzione c_2 di una iterazione del ciclo interno è costante rispetto a questi elementi

Si ha pertanto:

$$f(n) = c_1(n-1) + \sum_{k=1}^{n-1} c_2 m_k$$

Qui c_1 è il tempo di esecuzione della parte del ciclo esterno che precede il ciclo interno, mentre m_k è il numero di iterazioni del ciclo interno alla k-esima iterazione del ciclo esterno

Algoritmi di ordinamento

- m_k coincide peraltro con il numero di confronti da effettuare, per cui il confronto è ancora una volta l'operazione dominante dell'algoritmo
- Ha senso determinare il valore di m_k nel caso peggiore, ossia quello in cui, ad ogni iterazione del ciclo esterno, l'elemento preso in esame, cioè *temp*, è minore di tutti gli elementi della porzione non ancora ordinata dell'array
- E' il caso che, per esempio, si verifica se l'array è inizialmente ordinato in maniera decrescente e lo si vuole ordinare in maniera crescente

Algoritmi di ordinamento

Nel caso peggiore occorre eseguire un confronto durante la prima iterazione del ciclo esterno, due confronti durante la seconda, … , *k* nella *k*-esima

Pertanto:

$$f(n) = c_1(n-1) + \sum_{k=1}^{n-1} c_2 k = c_1(n-1) + c_2 \frac{(n-1)n}{2}$$

La complessità dell'ordinamento per inserzione lineare, nel caso peggiore, è pertanto ancora $O(n^2)$

Se invece l'array fosse inizialmente già ordinato (caso migliore), si dovrebbe eseguire un solo confronto ad ogni iterazione del ciclo esterno, e pertanto la complessità si ridurrebbe a $O(n)$

Algoritmi di ordinamento

Riepilogando:

- Il comportamento dell'algoritmo di ordinamento per inserzione lineare è pertanto *mediamente* migliore di quello per selezione
- La complessità dell'ordinamento per inserzione lineare, nel caso peggiore, è infatti $O(n^2)$
- Se invece l'array fosse inizialmente già ordinato (caso migliore), si dovrebbe eseguire un solo confronto ad ogni iterazione del ciclo esterno, e pertanto la complessità si ridurrebbe a $O(n)$

Algoritmi di ordinamento

Ordinamento a bolle (bubblesort)

L'algoritmo sfrutta il fatto che in un array ordinato per valori non decrescenti, l'elemento in posizione *i*-esima è non maggiore dell'elemento in posizione *i* + 1-esima, per ogni *i* compreso tra 0 e *n* - 2, essendo *n* gli elementi dell'array

Algoritmi di ordinamento

Pertanto l'ordinamento può essere ottenuto con una serie di iterazioni, durante ognuna delle quali ciascun elemento della sequenza di origine è confrontato con il successivo, e scambiato con esso se la relazione di ordinamento non è già soddisfatta

Ad ogni passo vengono quindi confrontate tutte le coppie di elementi contigui della sequenza di origine: se in una coppia il primo valore è minore o uguale al secondo, gli elementi restano inalterati, viceversa se il primo valore è maggiore del secondo, i due elementi vengono scambiati di posto

Algoritmi di ordinamento

In questo modo, gli elementi più piccoli della sequenza salgono gradualmente verso le prime posizioni della sequenza stessa, come le bolle d'aria immerse in un liquido che salgono verso la superficie

L'algoritmo richiede in totale $n - 1$ iterazioni, durante ciascuna delle quali vengono effettuati $n - 1$ confronti

La complessità asintotica di questo algoritmo è pertanto $O(n^2)$ e non dipende dalla disposizione iniziale dei valori negli elementi dell'array.

Il metodo seguente, che si avvale del metodo `swap` visto in precedenza, implementa l'algoritmo

Algoritmi di ordinamento

```
public void firstBubbleSort (int[] b)
{
 for ( int pass = 1; pass < b.length; pass++) // ciclo
                // esterno (n.ro passate)
    {
      for ( int i = b.length - 2; i >= 0; i--) // ciclo
                              // interno (n.ro confronti)
        {
          if ( b[i] > b[i + 1] ) swap( b, i, i + 1);
                             // eventuale scambio
         } // fine ciclo intefine ciclo esterno
     }
} // fine metodo firstBubbleSort
```

Algoritmi di ordinamento

E' evidente come l'istruzione dominante sia il confronto tra due elementi dell'array, seguito dall'eventuale scambio

L'istruzione viene eseguita $(n - 1)(n - 1)$ volte, per cui la complessità è $O(n^2)$

Si può osservare, però, che al termine della prima iterazione, che inizia confrontando l'ultimo elemento con il penultimo, l'elemento più piccolo dell'array sarà correttamente collocato nella prima posizione dell'array stesso

La seconda iterazione potrà quindi prendere in considerazione solo gli n - 1 elementi più grandi non ancora ordinati

Algoritmi di ordinamento

Al termine della seconda iterazione, il secondo elemento più piccolo sarà collocato correttamente in seconda posizione. La terza iterazione potrà quindi prendere in considerazione solo gli *n* - 2 più grandi elementi non ancora ordinati

Al termine della *k*-esima passata, il *k*-esimo elemento più piccolo sarà collocato correttamente in *k*-esima posizione, e così via, fino al completo ordinamento dell'array

Inoltre, non è detto che tutte le *n* – 1 iterazioni del ciclo esterno siano necessarie. Se durante una iterazione, infatti, non si rende necessario alcuno scambio, l'array risulta ordinato e non occorre proseguire

Algoritmi di ordinamento

E' evidente come l'istruzione dominante sia il confronto tra due elementi dell'array, seguito dall'eventuale scambio

L'istruzione viene eseguita $(n - 1)(n - 1)$ volte, per cui la complessità è $O(n^2)$

Inoltre, non è detto che tutte le $n - 1$ iterazioni del ciclo esterno siano necessarie. Se durante una iterazione, infatti, non si rende necessario alcuno scambio, l'array risulta ordinato e non occorre proseguire

Algoritmi di ordinamento

La seguente modifica dell'algoritmo tiene conto delle considerazioni precedenti, ed è tale da adattarne la complessità alla configurazione dei dati

```
public void bubbleSort (int[] b)
   {boolean noswap;  // flag vero per non avvenuto scambio
     for ( int pass = 1; pass < b.length; pass++) // ciclo
                              // esterno (n.ro passate)
     { noswap = true;
      for ( int i = b.length - 2; i >= pass - 1; i--)
                  //ciclo interno (n.ro confronti)
       {if ( b[i] > b[i + 1] ) // confronto e eventuale
          { swap( b, i, i + 1); noswap = false; }//scambio
       } // fine ciclo interno
      if ( noswap ) return;
     } // fine ciclo esterno
   } // fine metodo bubbleSort
```

Algoritmi di ordinamento

In questa formulazione, se l'array è inizialmente già ordinato (caso migliore), l'algoritmo esegue solo la prima iterazione del ciclo esterno, e poi termina

La complessità è determinata quindi solo dal ciclo interno, ed è pertanto $O(n)$

Se l'array è inizialmente ordinato a rovescio (caso peggiore), sono necessarie tutte le $n - 1$ iterazioni del ciclo esterno, perché dopo ognuna di esse solo un elemento si troverà nella posizione corretta

Algoritmi di ordinamento

Durante la *pass*-esima iterazione del ciclo esterno, peraltro, sono necessarie $n - 1 - (pass - 1) = n - pass$ iterazioni del ciclo interno

Si ha pertanto:

$$f(n) = c_1(n-1) + \sum_{pass=1}^{n-1} c_2(n - pass) =$$

$$= c_1(n-1) + c_2 n(n-1) - c_2 \sum_{pass=1}^{n-1} pass =$$

$$= c_1(n-1) + c_2 n(n-1) - c_2 \frac{(n-1)n}{2} =$$

$$= c_1(n-1) + c_2 \frac{(n-1)n}{2}$$

Algoritmi di ordinamento

La complessità dell'algoritmo di ordinamento a bolle, nel caso peggiore, è pertanto $O(n^2)$

FINE

Algoritmi di ordinamento (quicksort)

Ordinamento veloce (quicksort)

E' l'algoritmo *in loco* che ha, in generale, prestazioni *mediamente* migliori (la sua complessità nel caso medio è $O(n*\log n)$) tra quelli basati sul confronto tra gli elementi da ordinare

E' molto popolare dato che richiede poche risorse ed è facilmente implementabile, tanto che, nella versione base, è presente in molte librerie di linguaggi

Algoritmi di ordinamento (quicksort)

Gli svantaggi sono dati dal fatto che non è stabile, nel caso peggiore ha un comportamento quadratico ed è particolarmente fragile: un semplice errore nella sua implementazione può passare inosservato ma causare in certe situazioni un drastico peggioramento nelle prestazioni

Algoritmi di ordinamento (quicksort)

L'idea base si può illustrare agevolmente in termini ricorsivi. Ad ogni passo, assunto un elemento come perno o pivot della sottosequenza da ordinare, si confrontano con esso gli altri elementi e si posizionano alla sua sinistra i minori e a destra i maggiori, senza tener conto del loro ordine

Algoritmi di ordinamento (quicksort)

Alla fine del passo il perno risulta collocato nella sua posizione definitiva, e separa due sottosequenze di elementi rimasti non ordinati

Si procede allo stesso modo all'ordinamento delle due sottosequenze, continuando fino a quando si ottengono sottosequenze costituite da un solo elemento, e quindi implicitamente ordinate

Quicksort può essere implementato utilizzando la classe seguente, ripresa da [3]

Algoritmi di ordinamento (quicksort)

```
public class QuickSorter {
 private int[] a;
 public QuickSorter(int[] anArray)
 {
   a = anArray;
 }
 public void sort()
 {
   sort(0, a.length - 1);
 }
 public void sort(int from, int to)
 {
   if (from >= to) return;
   int p = partition(from, to);
   sort(from, p-1);
   sort(p + 1, to);
 }
```

Algoritmi di ordinamento (quicksort)

```
private int partition(int from, int to)
{
   int pivot = a[from];
   int i = from;
   int j = to;
   while (i < j)
    {
      while (a[i] < pivot) i++;
      while (a[j] > pivot) j--;
      if (i < j) swap(i, j);
    }
   return j;
}
private void swap(int i, int j)
 {int temp = a[i]; a[i] = a[j]; a[j] = temp;
 }
}                    \\fine classe quicksorter
```

Algoritmi di ordinamento (quicksort)

Il metodo `sort` divide l'array in due parti e viene chiamato (ricorsivamente) su ciascuna di esse

Le due parti dell'array sono determinate dal metodo `partition` che, posto il pivot uguale al primo elemento dell'array da ordinare (`pivot = a[from]`), organizza la suddivisione del vettore in due parti, tali che la parte sinistra contenga solo elementi minori del pivot e la parte destra solo elementi maggiori del pivot

Il metodo `swap` è utilizzato per scambiare due elementi, quando la relazione d'ordine non è rispettata

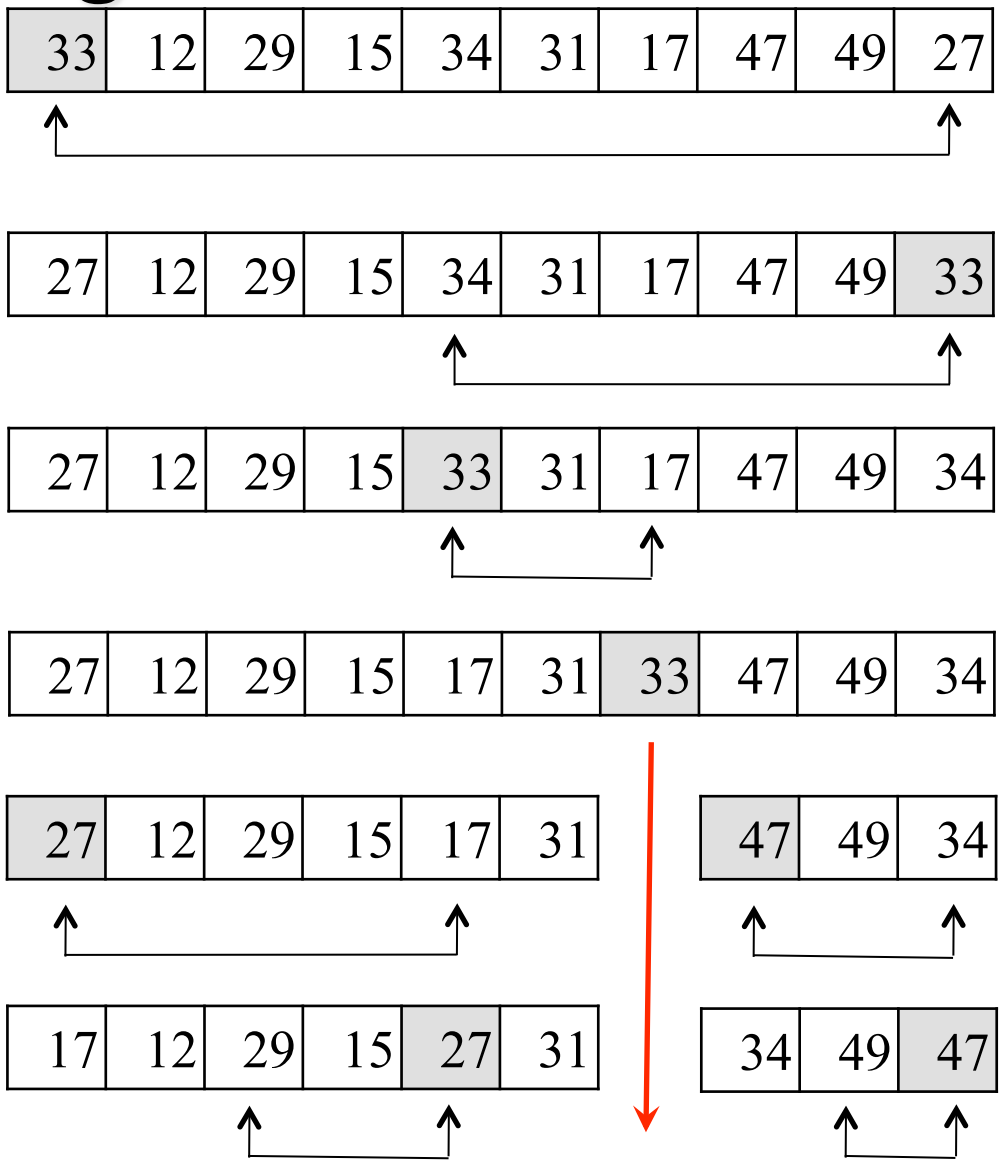
Algoritmi di ordinamento (quicksort)

Poiché non vi è motivo per ritenere che le due parti in cui risulta suddiviso l'array siano ordinate al loro interno, `partition` non ordina il vettore

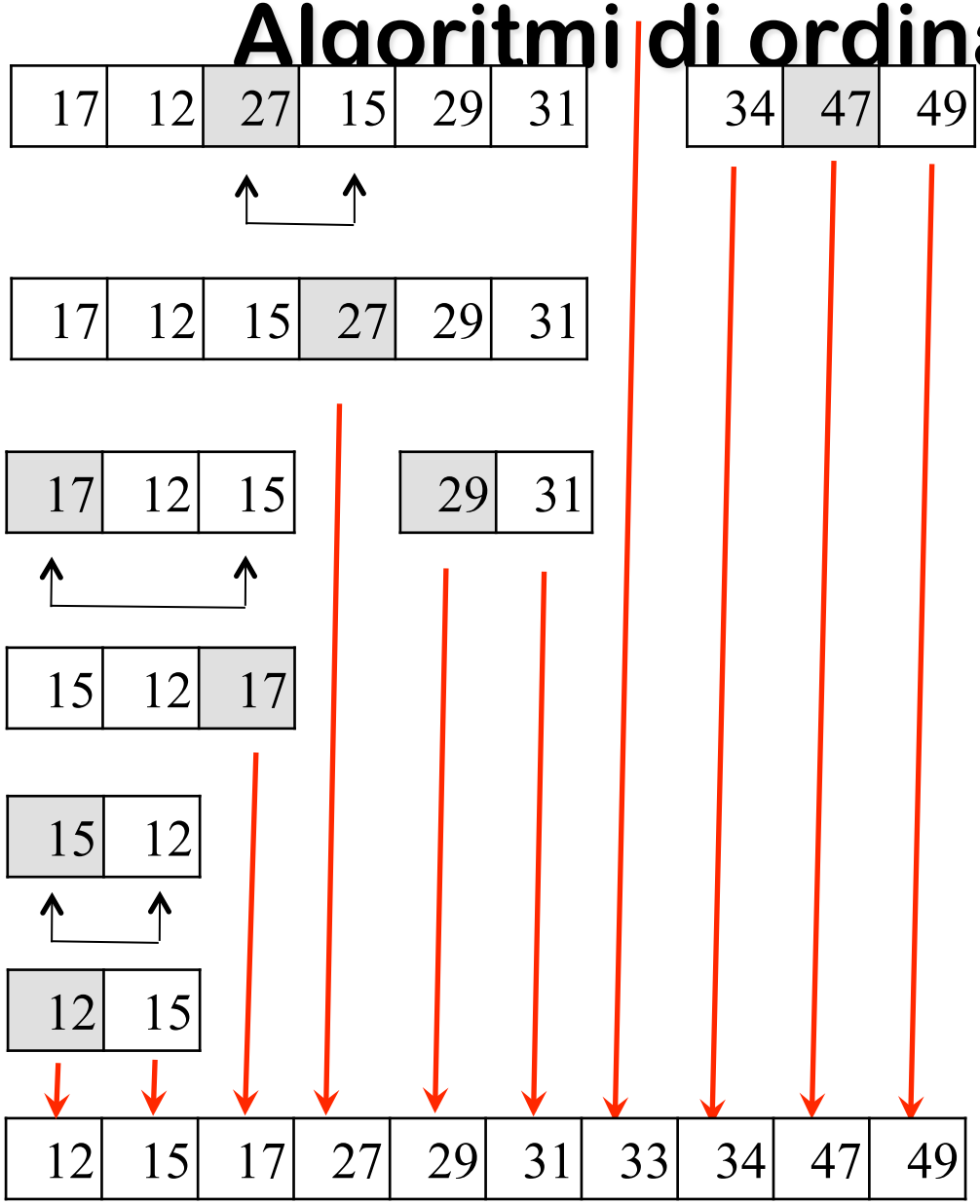
L'ordinamento complessivo dell'array si ottiene per il fatto che le parti via via ottenute diventano sempre più piccole, rimanendo ordinate fra di loro

Le operazioni dell'algoritmo possono essere illustrate dall'esempio seguente (il pivot è marcato in grigio)

Algoritmi di ordinamento (quicksort)



Algoritmi di ordinamento



Algoritmi di ordinamento

Per l'analisi della complessità dell'algoritmo quicksort, si può notare che `sort`, a parte le due attivazioni ricorsive, esegue un numero di operazioni che dipende da `partition`

A sua volta, `partition` ha una complessità che dipende dal numero di iterazioni del ciclo `while` esterno

Questo numero è al più uguale a `to - from`, dato che ad ogni iterazione viene incrementato `i` e decrementato `j`. Pertanto la complessità di `partition` è lineare con il numero di elementi sui quali opera

Algoritmi di ordinamento

Il numero delle attivazioni ricorsive di `sort` dipende invece dalla scelta del pivot

Indicando con $f(n)$ la funzione che esprime il tempo di esecuzione necessario per ordinare un array di n elementi, si può scrivere:

$$f(n) = \begin{cases} c & \text{se } n = 1 \\ cn + f(t) + f(n-t-1) & \text{se } n > 1 \end{cases}$$

Si è supposto che la prima attivazione di `sort` richieda di ordinare un array di t elementi e uno di $n - t - 1$ elementi

Algoritmi di ordinamento

Questa *equazione di ricorrenza* ammette soluzioni diverse al variare di *t*

Il caso peggiore dell'algoritmo è quello in cui ogni volta viene scelto come pivot l'elemento più grande (o più piccolo) del vettore

Se, come nella implementazione mostrata sopra, il pivot coincide ogni volta con il primo elemento dell'array, il caso peggiore è dunque quello di un array già ordinato: ad ogni passo l'array viene suddiviso in due parti, una costituita da un solo elemento, l'altra da tutti i rimanenti

Algoritmi di ordinamento

Si ha pertanto, nel caso peggiore:

costo di

`sort(0,n-1)` = $c*n + c$ + costo di `sort(0,n-2)`

`sort(0,n-2)` = $c*(n-1) + c$ + costo di `sort(0,n-3)`

`sort(0,n-3)` = $c*(n-2) + c$ + costo di `sort(0,n-4)`

...

`sort(0,1)` = $c*2 + c$ + costo di `sort(0,0)`

`sort(0,0)` = c

Algoritmi di ordinamento

Sostituendo ed eseguendo le somme, si ottiene, per il caso peggiore, la complessità $O(n^2)$, dello stesso ordine, cioè, degli algoritmi di ordinamento diretti visti in precedenza

Il caso migliore si ha quando il pivot è scelto ogni volta in modo da suddividere l'array in due parti di uguale lunghezza

Algoritmi di ordinamento

L'equazione di ricorrenza diviene infatti:

$$f(n) = \begin{cases} c & \text{se } n = 1 \\ cn + f\left(n/2\right) + f\left(n/2\right) & \text{se } n > 1 \end{cases}$$

Algoritmi di ordinamento

Si può dimostrare che la soluzione di questa equazione è $O(n * \log n)$, e si dimostra altresì che questa è anche la complessità dell'algoritmo nel caso medio

Questo spiega il buon comportamento pratico dell'algoritmo: è improbabile che la scelta del pivot sia così sfortunata da suddividere ogni volta l'array in due parti, di cui l'una molto più grande dell'altra

E' più probabile che l'array venga suddiviso in parti aventi un numero di elementi che realizzano situazioni più vicine al caso medio o al caso migliore

Bibliografia

1. N. Wirth "Algoritmi+Strutture Dati=Programmi", Tecniche Nuove, 1987.
2. L. Cabibbo "Fondamenti di Informatica, Oggetti e Java", Mc Graw-Hill, 2004.
3. C.S. Horstmann "Concetti di informatica e fondamenti di Java 2", Apogeo, 2002.
4. C. Batini et al. “Fondamenti di programmazione dei calcolatori elettronici”, F. Angeli, 1992.
5. S. Ceri, D. Mandrioli, L. Sbattella “Informatica arte e mestiere”, McGraw-Hill, 1999.