PROGRAMMAZIONE

Fusione
Partizionamento
Algoritmi avanzati di ordinamento

Algoritmi su Array

- Algoritmi di base (conteggio, ricerca max/min, elimina duplicati,…)
- Algoritmi di supporto
  - Partizionamento
  - Fusione
- Ricerca
  - Sequenziale
  - Binaria (Basata su un ordinamento)
- Ordinamento
  - Per selezione, Per scambio, Per inserzione
  - Per fusioni successive, Per partizionamento (v.iterativa e ricorsiva)

Tecniche Avanzate di Ordinamento

- Si può dimostrare che, in media, per dati casuali, gli elementi devono essere spostati di una distanza pari a $n/3$ rispetto alla loro posizione originaria
  - Gli algoritmi semplici tendono a spostare solo gli elementi vicini e quindi necessitano di molti spostamenti in più per ordinare la lista
    - Meno efficienza
  - Gli algoritmi migliori sono in grado di scambiare, nelle prime fasi di ordinamento, valori che occupano posizioni molto distanti nella lista

Partizionamento

- Dato un array non ordinato di *n* elementi, e un valore *x* dello stesso tipo degli elementi, partizionare gli elementi in due sottoinsiemi tali che
  - Gli elementi $\leq x$ siano in un sottoinsieme
  - Gli elementi $> x$ siano nell'altro sottoinsieme
- Esempio:

  a: | 4 | 13 | 28 | 18 | 7 | 36 | 11 | 24 | $x = 20$

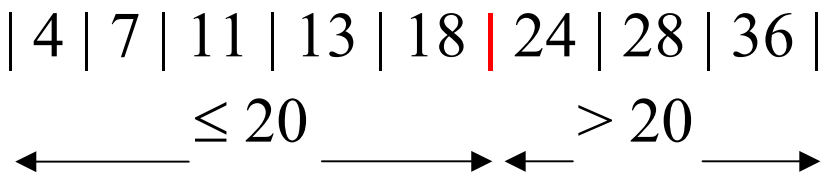
  una possibile soluzione:

  | 4 | 13 | 11 | 18 | 7 | 36 | 28 | 24 |

  ≤ 20 > 20

Partizionamento

- Si potrebbe ordinare l'array e trovare l'indice dell'elemento che separa i due sottoinsiemi



  - Non richiesto
    - suggerimento: l'algoritmo non deve fare più di quanto venga richiesto
  - Non necessario
- Da Matematica Discreta:

P_S unione P_D = intero insieme

(l'unione delle parti copre l'insieme originario e le parti a due a due non hanno intersezioni)

Partizionamento

- Se è nota la posizione p che divide i due sottoinsiemi
  - Alcuni elementi non vanno spostati
    - Scorrendo l'array da sinistra verso p, i valori $\leq x$
    - Scorrendo l'array da destra verso p, i valori $> x$
  - Alcuni elementi sono fuori posto
    - Scambiare un elemento fuori posto da una parte con uno fuori posto dall'altra parte.
- La posizione p può essere ricavata contando gli elementi $\leq x$ presenti nell'array
- In tal caso, occorrono 2 scansioni dell'array
  - Determinare la posizione p
  - Individuare e scambiare gli elementi fuori posto
    - Si termina quando si arriva a p

Partizionamento

- È possibile partizionare l'array mediante una sola scansione?
  - Si, procedendo dagli estremi verso il centro
- ***Mentre*** *le due partizioni non si incontrano*

  Estendere le partizioni sinistra e destra scambiando le coppie piazzate nella partizione sbagliata
  - La posizione *p* sarà individuata automaticamente al termine della scansione

Partizionamento

- Scansione effettuata tramite 2 indici
  - *i* per la partizione sinistra
    - Parte da 1
    - Incrementato
  - *j* per la partizione destra
    - Parte da *n*
    - Decrementato

Partizionamento
Algoritmo

- Individuazione dei primi valori fuori posto
  - Mentre l'i-mo elemento è $\leq x$ e $i < j$
    - Incrementare i
  - Mentre il j-mo elemento è $> x$ e $i < j$
    - Decrementare j
- Mentre le partizioni non si sono incontrate
  - Scambiare i valori nelle posizioni individuate
  - Cercare i successivi valori fuori posto
- Al termine, j è la posizione p
  - Limite inferiore per i valori $> x$ nella partizione

Partizionamento
Note Implementative

- Gestione del caso in cui *x* è fuori dall'intervallo dei valori assunti dall'array
  - Inserire un controllo supplementare dopo i cicli preliminari:

    Se il *j*-mo elemento è > *x* allora decrementa *j*

    - Ciò assicura che $\forall k \in [1..j]: a[k] \leq x$
      - Se *x* è maggiore di tutti gli elementi dell'array, al termine dei cicli si ha $i = j = n$ e il controllo non scatta
      - Se *x* è minore di tutti gli elementi dell'array, al termine del ciclo(1a) si ha $i = 1$, al termine del ciclo(1b) si ha $j = 1$ e a_1 non controllato, mentre dopo il controllo, $j = 0$

Partizionamento
Esempio

- Dati

  $x = 17$

  | 28 | 26 | 25 | 11| 16| 12 | 24 | 29 | 6 | $n = 9$

Il risultato è:

| **28** | 26 | 25 | 11| 16| 12 | 24 | 29 | **6** | *i=1, j=n*

| 6 | **26** | 25 | 11| 16| **12** | 24 | 29 | 28 | *i=2, j=6*

| 6 | 12| **25** | 11| **16**| 26 | 24 | 29 | 28 | *i=3, j=5*

| 6 | 12| 16 | 11| 25 | 26 | 24 | 29 | 28 | *j=4*

Dimensione di P_S: ***p = 4***

P_D Partizione di destra ha dimensione: $n-p$

Partizionamento
Casi limite

- $x = 17$

  | 28 | 32 | 39 | 37| 53| È già partizionato!

  P_S è vuota

  P_D coincide con l'intera lista

  Tutti gli a[*j*] sono > x

- $x = 17$

  | 12 | 6 | 5 | 4 | 11| È già partizionato!

  P_S coincide con l'intera lista

  P_D è vuota

  Tutti gli a[*i*] sono <= x

Partizionamento
Pseudo-codice

inizio

i = 1; j = n;

mentre (i < j) e (a_i <= x) **esegui** i = i + 1; /*estende P_S verso il centro*/
mentre (i < j) e (a_j > x) **esegui** j = j – 1; /*estende P_D verso il centro*/

se a_j > x **allora** j = j – 1; /* caso particolare: $x \leq a_k \; \forall k \in [1..n]$ */

mentre i < j **esegui**

t = a_i; a_i = a_j; a_j = t; /* scambia */

i = i + 1; j = j – 1; /* continua ad estendere P_S e P_D */

mentre (a_i <= x) **esegui** i = i + 1; /*ricerca elemento da scambiare*/

mentre (a_j > x) **esegui** j = j – 1; /*ricerca elemento da scambiare*/

fine_mentre

p = j; /* dimensione della P_S */

fine

Partizionamento
Codice C

Partition (a, 0, n-1, x, p)

Nota 1: al posto di *n* posso usare una funzione che ritorni la lunghezza di un array - *lenght(a)*

Partizionamento
Programma Pascal

```
begin
    i := 1; j := n;
    while (i < j) AND (a[i] <= x) do i := i + 1;
    while (i < j) AND (a[j] > x) do j := j – 1;
    if a[j] > x then j := j – 1;
    while i < j do
            begin
                    t := a[i]; a[i] := a[j]; a[j] := t;
                    i := i + 1; j := j – 1;
                    while (a[i] <= x) do i := i + 1;
                    while (a[j] > x) do j := j – 1;
            end;
    p := j;
end;
```

Ordinamento Veloce di Hoare
(per partizionamento)

- Ordinamento basato sul partizionamento (partition-exchange)
- Noto come Quicksort
  - in media è molto veloce
- Due versioni (iterativa, ricorsiva)

  | tutti gli elementi + piccoli | tutti gli elementi + grandi |
  - Il partizionamento aumenta il preordinamento
  - Si riduce la dimensione del problema
    - Ogni partizione può essere trattata separatamente

Ordinamento Veloce

- Il partizionamento richiede un valore *x* (pivot o perno) in base al quale separare gli elementi.
- Come scegliere il pivot?
  - Scelto a caso
    - Potrebbe essere esterno ai valori della lista
      - Avremmo una partizione vuota!
  - Prenderne uno della lista stessa
    - Scegliere il primo della lista (versione per EUCIP)
    - Il tempo di esecuzione dipende dal fatto che il partizionamento sia o meno bilanciato
      - Elemento *mediano*
        - » Elemento in posizione media
        - » Posizione media: (limite inferiore+limite superiore) /2

Ordinamento Veloce

- Situazione dopo il partizionamento:
  - Parte sinistra
    - Elementi minori dell'elemento mediano
  - Parte destra
    - Elementi maggiori dell'elemento mediano
- Sone escluse le interazioni fra partizioni
  - Nessun elemento della parte sinistra potrà mai, nell'ordinamento, finire nella parte destra
    - Ciascuna può essere trattata (ordinata) separatamente
    - Ciascuna contiene meno elementi della lista completa

Ordinamento Veloce

- Per ciascuna delle 2 partizioni ottenute valgono le stesse considerazioni precedenti
  - Problemi analoghi a quello iniziale (ordinamento)
  - Dimensione minore
- Riapplichiamo su ciascuna i concetti già esposti
  - Si ottengono, per ciascuna partizione, altre 2 partizioni
  - Situazione dopo un certo numero di passi:

elementi I partizione < elementi II partizione < … < elementi *n*-ma partizione

- Criterio di stop: partizioni di un solo elemento
  - Non ulteriormente partizionabili
  - Già ordinate

Ordinamento Veloce
Meccanismo Base

- **Mentre** tutte le partizioni non risultano ancora ridotte a dimensione 1 **esegui**
  - Scegli la prossima partizione da elaborare
  - Individua un nuovo valore di partizionamento per la partizione scelta (il mediano)
  - Partiziona la partizione attuale in due insiemi più piccoli parzialmente ordinati rispetto al mediano attuale

Ordinamento Veloce

- Ogni passo genera 2 nuove partizioni
  - Si può agire su una sola partizione alla volta!
    - L'elaborazione dell'altra resta in sospeso
  - Possibile esistenza, in un certo momento, di partizioni create in attesa di elaborazione
    - Necessità di ricordarne i limiti destro e sinistro
      - Salvare l'informazione sui limiti di ciascuna
        - » posizioni dell'array originale

Ordinamento Veloce

- Memorizzazione dei limiti delle partizioni lasciate in sospeso
  - Ogni partizione richiede la memorizzazione di 2 valori che la individuano (gli estremi)
    - Immagazzinati in un array di appoggio
      - Relativi agli indici dell'array di partenza
      - 2 elementi dell'array di appoggio per ogni partizione
- Scelta della prossima partizione
    - Simulazione di una struttura a pila
      - Last In, First Out

Ordinamento Veloce

- Man mano che il processo di partizionamento prosegue
  - Il numero delle partizioni di cui salvare l'informazione sui limiti aumenta (di 1)
  - Le partizioni diventano più piccole

Ordinamento Veloce

- Dimensione dell'array di servizio
  - Caso peggiore: 2 *(n–1) elementi
    - Il partizionamento è il più sbilanciato possibile
    - Si memorizzano n – 1 partizioni lasciate in sospeso
      - Ogni volta si elabora la partizione più grande
      - Ogni volta la partizione più piccola contiene un solo elemento

        | partizione elaborata per prima | x |
    - Ciascuna partizione richiede 2 elementi
  - Troppo Spazio! Cerchiamo di migliorare…

Ordinamento Veloce
Ottimizzazione

- Elaborare per prima la partizione più piccola
  - Minor numero di partizioni da tenere in sospeso
  - Partizioni di dimensione 1 non generano altre partizioni
    - Individuate e scartate subito dall'elaborazione
    - In ogni istante esiste una sola partizione in sospeso di cui vanno salvati i limiti
      - corrispondono alla partizione più grande

Ordinamento Veloce

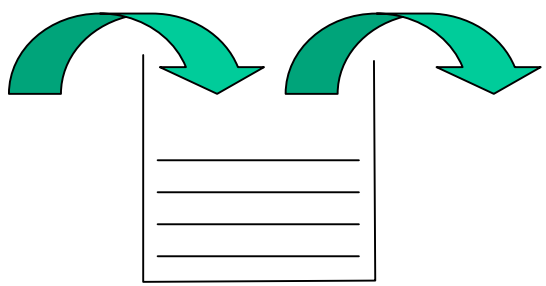
Ottimizzazione

- Ad ogni passo si elabora la più grande delle partizioni più piccole
  - Attuale caso peggiore: $2 \log_2 n$
    - Partizioni divise sempre a metà
      - Dimensione dimezzata
    - Caso peggiore senza ottimizzazione: $2*(n-1)$ elementi

Accettabile!

Ordinamento Veloce

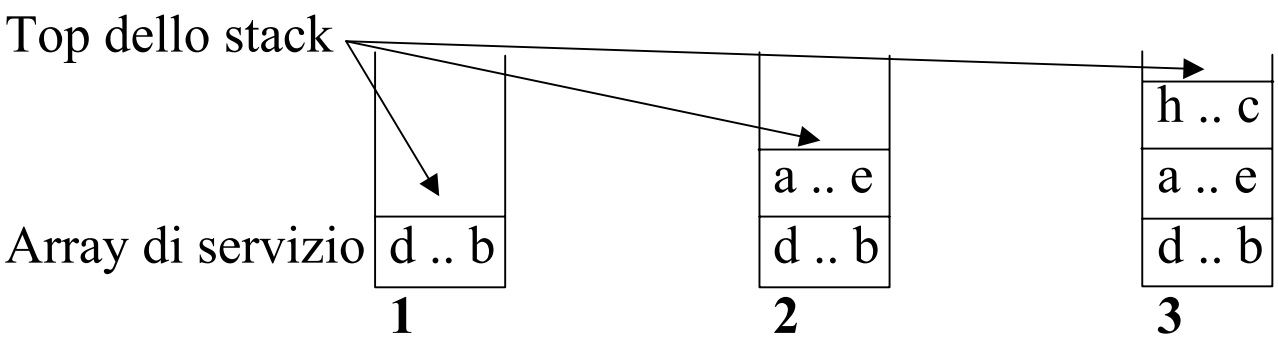
- Test per decidere qual è la partizione più grande
  - Se le partizioni “cadono” (si congiungono) a sinistra del medio
    - Elabora la partizione sinistra (che è la più piccola)
    - Salva i limiti di quella destra
  - Altrimenti
    - Elabora la partizione destra (che è la più piccola)
    - Salva i limiti di quella sinistra
- La partizione più grande è sempre messa in attesa di elaborazione
  - Gestione a pila



Ordinamento Veloce

- Meccanismo di partizionamento e accodamento

1 | | |
a cd b a..c < d..b
2 | | |
a e f c f..c < a..e
3 | | |
f gh c f..g < h..c



Ordinamento Veloce

- Quando si raggiunge una partizione di dimensione 1
  - si ricomincia il processo di partizionamento sulla partizione più recentemente memorizzata
    - Limiti della partizione rimossi dalla cima della pila
- Quando tutte le partizioni sono state ridotte a dimensione 1
  - Pila vuota
  - **Terminazione** del processo di partizionamento

Ordinamento Veloce
Algoritmo iterativo

Definire l'array *a*[1..*n*] da ordinare

Inizializzare a 2 il puntatore al top dello stack

Impostare i limiti *sup* e *inf* dell'array nello stack a 1..*n* (si parte con 1 sola partizione coincidente con l'intero array di partenza)

Mentre lo stack non è vuoto **esegui**

Estrai dal top dello stack la prossima partizione (limiti *sup* e *inf* del segmento di array corrispondente)

Mentre il corrente segmento non è ridotto a dimensione 1 **esegui**

Scegli l'elemento mediano del segmento di array

Partiziona il segmento in 2 rispetto al suo valore mediano

Salva nello stack i limiti della partizione più grande

Segmento corrente ← partizione più piccola

Ordinamento Veloce
Programma Pascal

```
begin
top := 2; stack[1] := 1; stack[2] := n;
while top > 0 do
  begin
  right := stack[top]; left := stack[top–1]; top := top – 2;
  while left > right do
    begin
    medio := (left + right) DIV 2; mediano := a[medio];
    partiziona(a,left,right,mediano,pos);
    if pos < medio then
      begin stack[top+1] := pos+1; top := top + 2; stack[top] := right; right := pos end
    else
      begin stack[top+1] := left; top := top + 2; stack[top] := pos; left := pos+1 end
    end
  end
end.
```

Ordinamento veloce
Versione Ricorsiva

- Algoritmo per natura ricorsivo
  - Implementazione ricorsiva molto semplice
    - Strategia *divide et impera* (applicazione dello stesso processo ad istanze del problema via via più piccole)
      - Divide: l'insieme di dati è ripartito in 2 partizioni in modo tale che *tutti* gli elementi della partizione sinistra (prime *pos* posizioni) siano ≤ di *tutti* gli elementi della parte destra (posizioni da *pos*+1 fino ad *n*)
      - Impera: ordina le 2 partizioni riapplicando lo *stesso* meccanismo di *partizionamento* a[1..*pos*] e a[*pos*+1..*n*] fino ad ottenere segmenti di dimensione 1
      - Ricombina: Non è richiesto alcuno sforzo per ricombinare le 2 partizioni
        - » A questo punto l'intero array a[1..*n*] sarà ordinato

Quicksort (v. ricorsiva)

se card(A) = 1

allora A è ordinato /* passo base */

altrimenti

-scegli un elemento *pivot* in A e ripartisci gli elementi di A in due sottoinsiemi, disponendo nella parte bassa gli elementi di A < del pivot e nella parte alta gli elementi di A > del pivot.

-riapplica l'algoritmo ai due sottoinsiemi

/* due chiamate ricorsive */

fine_se

Quicksort (v. ricorsiva)

- Situazione iniziale:

Dati originali
inf = 1	sup = n

  – Dopo il primo partizionamento:

partizione sinistra	partizione destra
inf	pos	sup

- Passi fondamentali dell'algoritmo ricorsivo
  – **Se** nell'insieme corrente c'è più di 1 elemento
    - Partiziona i dati in partizione sinistra e destra
    - Riapplica il partizionamento alla partizione sinistra
    - Riapplica il partizionamento alla partizione destra

Quicksort (v. ricorsiva)

- Indicatori di indice necessari:

inf	pos	sup

  - partition(a, inf, sup, mediano, pos)
- Ricorsione
  - quicksort2(a, inf, pos)
  - quicksort2(a, pos+1, sup)
- Terminazione della ricorsione
  - Chiamata ad un segmento di 1 elemento
    - Test: **se** inf < sup **allora** continua

Nota: il modulo di partizionamento del quicksort può fare a meno del parametro *mediano*

Quicksort
Algoritmo Ricorsivo

Se il segmento corrente da ordinare contiene più di 1 elemento **allora**

Partizionalo in 2 segmenti più piccoli tali che tutti gli elementi del segmento di sinistra siano ≤ di tutti gli elementi del segmento di destra

Se il segmento di sinistra è più piccolo del segmento di destra **allora**

Quicksort il segmento di sinistra

Quicksort il segmento di destra

Altrimenti

Quicksort il segmento di destra

Quicksort il segmento di sinistra

Quicksort
Una implementazione C

```
void quicksort2 (int a[MAX], int inizio, int fine)
int  q;
    if (inizio<fine)
          { partition (a, inizio, fine, q);
            quicksort2 (a, inizio, q);
            quicksort2 (a, q+1, fine);
          }
}
```

(Versione che non ottimizza la scelta della partizione)

La libreria standard del C fornisce una implementazione efficiente del QuickSort denominata qsort( )

QuickSort
versione per EUCIP **(C++)**

```
void QuickSort (int *vettore, int IndiceSinistro, int IndiceDestro) {
if (IndiceSinistro  < IndiceDestro)
{
    int Pivot = vettore[IndiceSinistro];  /*scelta del primo come pivot*/
    int IS = IndiceSinistro;
    int ID = IndiceDestro +1;
    do
    {
        do IS++; while (vettore[IS] < Pivot;
        do ID++; while (vettore[ID] < Pivot;
        if (IS< ID) Scambia (vettore[IS], vettore[ ID]);
     } while (IS < ID);
     Scambia (vettore[IS], vettore [ID]);
     QuickSort (vettore, IndiceSinistro, ID-1);
     QuickSort (vettore, ID+1, IndiceDestro);
} } // QuickSort
```

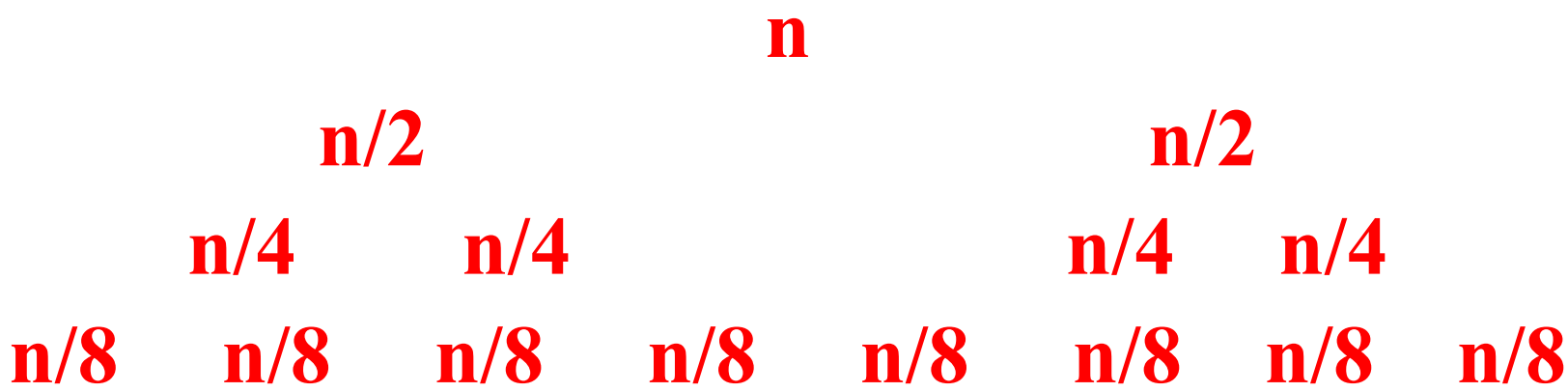
Ordinamento Veloce
Programma Pascal

```
procedure quicksort2(var a: tipo_array; inf, sup: integer);
    var   pos : integer;
begin
if inf < sup then
    begin
    medio := (inf + sup) DIV 2;
    mediano := a[medio];
    partition(a, inf, sup, mediano, pos);
    if (pos – inf) < (sup – pos+1) then
            begin quicksort2(a, inf, pos); quicksort2(a, pos+1, sup) end
    else
            begin quicksort2(a, pos+1, sup); quicksort2(a, inf, pos) end
    end
end;
```

Ordinamento Veloce
Considerazioni

- Il tempo di esecuzione dipende dal bilanciamento delle partizioni effettuate dall'algoritmo di partizionamento (scelta del pivot)
  - caso migliore: partizioni perfettamente bilanciate, entrambe di dimensione $n/2$
  - caso peggiore: una partizione è sempre di dimensione 1 (l'altra sarà di dimensione n-1)
- Suddivisione successiva della sequenza in sottosequenze
  - Rappresentazione tramite un albero binario
- Complessità: $n \log_2 n$ *(tempo asintotico medio)*
  - **Algoritmo ottimo!**

Ordinamento Veloce
Considerazioni



sino ad ottenere

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

$n \log_2 n$

Fusione (Merge)

- Fondere due array, già ordinati secondo il medesimo criterio, in un unico array ordinato secondo sempre lo stesso criterio

  a: | 2 | 5 | 9 | 13 | 24| ***b:*** | 3 | 4 | 11 | 15 | 22 |

  c: | 2 | 3 | 4 | 5 | 9 | 11 | 13 | 15 | 22 | 24|

  – Numero di elementi dell'array ***c*** risultante pari alla somma degli elementi dei due array, ***m +n***

  – Necessità di esaminare tutti gli elementi dei due array dati

Fusione

- I due array possono considerarsi come costituiti da sottosequenze da concatenare opportunamente nell'array finale
  - Suddividere ciascuno dei due array dati in una parte già fusa e una ancora da fondere
    - In ogni istante, esaminare il primo elemento della parte da fondere dei due array
      - Inserire il minore nella prima posizione dell'array fuso
      - Avanzare al prossimo elemento dell'array da cui è stato preso

Fusione

- Uno dei due array finirà per primo
  - Copiare fino ad esaurimento gli elementi restanti nell'array finale
  - Individuabile dall'inizio confrontando gli ultimi elementi dei due array
- Caso particolare: se l'ultimo elemento di un array è minore del primo elemento dell'altro, la fusione si riduce alla copiatura degli elementi dell'uno seguita dalla copiatura degli elementi dell'altro

Fusione
Esempio

a: | 15 | 18 | 42 | 51|

i=2

b: | 8 | 11 | 16 | 17 | 44 | 58 | 71| 74 |

j=3

c: | 8 | 11 | 15 | |…

k=4

Fusione
Algoritmo ingenuo

- Mentre ci sono elementi da considerare nei due array *(i<m) e (j <n)*
  - Confronta i loro primi elementi non fusi e metti il più piccolo nell'array finale
  - Aggiorna l'indice dell'array appropriato
- Se è finito il primo array, allora
  - Copia il resto del secondo nell'array finale

  Altrimenti
  - Copia il resto del primo nell'array finale

Fusione
Una implementazione C

```
void merge( int a[ ], int b[ ], int c[ ], int m, int n)
{
   int i = 0, j= 0, k = 0;
   while ((i < m)&& (j < n))
        if (a[i] < b[j])
                {c[k] = a[i]; i=i+1; k=k+1};
        else {c[k] = b[ j]; j=j+1; k=k+1};
   while (i < m) c[k++] = a[ i++];
   while (j < n) c[k++] = b[ j++];
}
```

Fusione
Modularizzazione

- Procedura *merge*
  - Decide quale array finisce prima, quindi esegue la fusione di conseguenza
- Procedura *mergecopy*
  - Distingue il caso di "accodamento" dalla fusione classica e fonde l'array che finisce prima con l'altro
- Procedura *shortmerge*
  - Fonde le parti degli array ricadenti nello stesso intervallo
- Procedura *copy*
  - Copia nell'array risultato gli elementi di un array dalla posizione attuale fino alla fine

La procedura da chiamare nel main è *merge*

Fusione
Procedura *merge*

Definire gli array $a[1..m]$ e $b[1..n]$

- Se l'ultimo elemento di a è ≤ dell'ultimo elemento di b allora
  - Fondi tutto a con b *mergecopy(a, b, c, m, n)*
  - Copia il resto di b *copy*
- Altrimenti
  - Fondi tutto b con a *mergecopy(b, a, c, n, m)*
  - Copia il resto di a *copy*

Fornire il risultato $c[1..n+m]$

Fusione
Procedura *mergecopy*

Definire gli array $a[1..m]$ e $b[1..n]$ con $a[m] \leq b[n]$

- **Se** l'ultimo elemento di a è ≤ al primo elemento di b **allora** (vengono evitati tutti i confronti)
  - Copia tutto a nei primi m elementi di c *copy*
  - Copia tutto b in c, partendo dalla posizione m+1 *copy*
- **Altrimenti**
  - Fondi in c tutto a con b *shortmerge*
  - Copia il resto di b in c partendo dalla posizione in cui è finita la fusione *copy*

Fusione

Procedura *shortmerge*

Definire gli array $a[1..m]$ e $b[1..n]$ con $a[m] \leq b[n]$

- **Mentre** tutto *a* non è ancora fuso **esegui**
  - **Se** il corrente di $a \leq$ del corrente di *b* **allora**
    - Copia il corrente di *a* nella corrente posizione di *c*
    - Avanza l'indice di *a*
  - **Altrimenti**
    - Copia il corrente di *b* nella corrente posizione di *c*
    - Avanza l'indice di *b*
  - Avanza l'indice di *c*

Fusione

Procedura *shortmerge*

a: | 15 | 18 | 42 | 51|

b: | 8 | 11 | 16 | 17 | 44 | 58 | 71| 74 | da fondere

c: | 8 | 11 | 15 | 16 | 17 | 18 | 42 | 44 | 51|…

Situazione della fusione dopo che l'array *a* si è esaurito

Fusione
Procedura *copy*

Definire gli array $b[1..n]$ e $c[1..n+m]$ e definire da dove cominciare la copia da b (al corrente j) e in c (al corrente k)

- **Mentre** non è ancora finito b **esegui**
  - Copia l'elemento dalla corrente posizione in b nella corrente posizione in c
  - Avanza l'indice di b (ossia j)
  - Avanza l'indice di c (ossia k)

Fusione
Considerazioni

- L'algoritmo non è adatto per fondere due insiemi di dati di cui non si conosca la cardinalità
  - È una situazione frequente
  - Occorre apportare modifiche

Ordinamento per Fusioni Successive

- Algoritmo di *MergeSort*
  - L'idea di base è che l'ordinamento di una lista di *n* elementi può essere ottenuto
    - Dividendo la lista in due sequenze di *n*/2 elementi ciascuna
      - Dimensione inferiore (esattamente dimezzata)
    - Ordinando singolarmente ogni sequenza
      - Problema di ordine inferiore
      - Risolubile secondo la stessa tecnica
        - » Procedura ricorsiva
    - Fondendo le due metà ordinate in un'unica sequenza
      - Lista risultante ordinata
  - Strategia fondata sul principio del ***divide et impera***

Ordinamento per Fusione
Esempio

- Insieme iniziale

| 33 | 21 | 7 | 48 | 28 | 13 | 65 | 17 |

- Suddivisione in 2 sottoinsiemi

| 33 | 21 | 7 | 48 | | 28 | 13 | 65 | 17 |

- Ordinamento di ogni singolo sottoinsieme

| 7 | 21 | 33 | 48 | | 13 | 17 | 28 | 65 |

- Fusione dei sottoinsiemi ordinati

| 7 | 13 | 17 | 21 | 28 | 33 | 48 | 65 |

MergeSort (sulla Ricorsione)

- Se l'array da ordinare
  - contiene **n= 1** elemento, allora (per definizione) l'array è già ordinato: è l'assioma
    - (anche se n=0 l'array è ordinato)
  - contiene **n>1** elementi, si suddivide a metà l'array iniziale, si ordinano le due metà e si fondono.
- Si arriva a suddividere l'array iniziale in *n* sottoinsiemi costituiti da 1 elemento ciascuno, si riaggregano i sottoinsiemi fondendoli ordinatamente sino a ricostruire una unica sequenza ordinata.

MergeSort (Implementazione C)

```
void MergeSort(int a[ ]; int primo, ultimo)
 {
   int mid;
    if primo < ultimo     /* a ha almeno 2 elementi */
            {
             mid = (primo + ultimo) / 2;
             MergeSort(a, primo, mid);    /* prima chiamata ricorsiva */
             MergeSort(a, mid + 1, ultimo); /* seconda chiamata
     ricorsiva */
             merge(a, primo, mid, ultimo);   /* Att.ne: procedura
                       iterativa diversa da quella definita in precedenza */
             }
}
```

Ordinamento per Fusioni Successive (Pascal)

```
procedure MergeSort(var a: nelements; primo, ultimo:
    integer);
    var  q: integer;
begin
    if primo < ultimo then
            begin
                    q := (primo + ultimo) DIV 2;
                    MergeSort(a, primo, q);
                    MergeSort(a, q + 1, ultimo);
                    merge(a, primo, q, ultimo)
            end
end;
```

Procedura Merge per MergeSort

- La procedura di fusione deve lavorare su segmenti diversi di uno stesso vettore
  - Non può essere la stessa già definita per la fusione di due vettori distinti
- Occorre definire una nuova procedura che fonde due sottosequenze contigue di uno stesso vettore

void merge(int a[ ]; int primo, mezzo, ultimo)

Occorrono:

- un array di appoggio *b* per il risultato della fusione (alla fine *b* viene copiato in *a*)
- variabili locali usate come indici

Merge per MergeSort (pseudo-codice)

Merge(a, primo, mid, ultimo)

inizio
Sia i= primo e j = mid + 1 /* fondere *a(primo..mid)* con *a(mid+1..ultimo*)*/
k = primo
mentre (i <= mid) e (j <= ultimo) **esegui**
    se $a_i < a_j$ **allora** $b_k = a_i$; i = i + 1;
        altrimenti $b_k = a_j$; j = j + 1;
    fine_se
    k = k + 1
fine_mentre
se i <= mid **allora** copia ($a_i \ldots a_{mid}$)
        altrimenti copia ($a_j \ldots a_{ultimo}$)
fine_se
copia l'array *b* nell'array *a*
fine Merge

Merge per il MergeSort (Pascal)

```
begin
i:= primo; k := primo; j := mezzo + 1;
while (i <= mezzo) and (j <= ultimo) do
    begin
        if a[i] < a[j] then
                begin    b[k] := a[i]; i := i + 1          end
        else
                begin    b[k] := a[j]; j := j + 1          end;
        k := k + 1
    end;
if i <= mezzo then
    begin j := ultimo – k; for h := j downto 0 do a[k+h] := a[i+h]      end;
for j := primo to k-1 do a[j] := b[j]
end;
```

Ordinamento per Fusioni Successive (Pascal)

```
procedure ordina_per_fusioni(var a: nelements; n: integer);
    var    b: nelements;
 procedure mergesort(var a,b: nelements; primo, ultimo: integer);
            var         q: integer;
            procedure merge(var a,b: nelements; primo, ultimo, mezzo: integer);
                        var i, j, k, h: integer;
            begin       {merge}
                        …
            end;        {merge}
    begin {mergesort}
            …
    end    {mergesort}
begin      {chiamante}
    mergesort(a, b, 1, n)
end        {chiamante}
```

MergeSort
Considerazioni

- Durante il processo di suddivisione, l'insieme non viene in alcun modo modificato, i suoi elementi non sono né confrontati, né spostati.
- Viene operata una suddivisione dicotomica ricorsiva della sequenza iniziale da ordinare sino ad ottenere *n* sequenze di dimensione 1.
- Quando la procedura ricorsiva MergeSort viene chiamata per una sottosequenza di lunghezza 1, sarà *primo=ultimo* e la procedura termina senza compiere alcuna operazione.
- Il punto di forza dell'algoritmo sta nel ***backtracking*** della ricorsione.

MergeSort Considerazioni

- Durante il ***backtracking*** vengono fusi a due a due i sottoinsiemi ottenuti con la suddivisione ricorsiva.
- Durante la fusione gli elementi vengono posizionati in modo da ottenere una unica sottosequenza ordinata da 2 sottosequenze ordinate
  - Si inizia la fusione con sequenze costituite da 1 solo elemento e, perciò, ordinate.
- La complessità computazionale è O($n\log_2 n$)
- Non esistono casi più favorevoli di altri
  - l'algoritmo si comporta nello stesso modo a prescindere dalla disposizione iniziale degli elementi nella sequenza da ordinare.

MergeSort
Esempio

$n = 8 = 2^3$

da ordinare: {3,15,6,2,7,4,12,8}

{3,15,6,2} {7,4,12,8}

{3,15} {6,2} {7,4} {12,8}

{3} {15} {6} {2} {7} {4} {12} {8}

{3, 15} {2, 6} *lungh. 2* {4, 7} {8, 12}

{2, 3, 6, 15} *lungh. 4* {4, 7, 8, 12}

ordinato: {2, 3, 4, 6, 7, 8, 12, 15} *lungh.8*

Le sottosequenze ordinate hanno lunghezza fissa:

da 2^0 *sino a* $2^{\log_2 n}$ $(\log_2 n = \log_2 2^3 = 3)$