Materiale didattico preparato dal
dott. Stefano Ferilli

Corso di Programmazione
Algoritmi Fondamentali

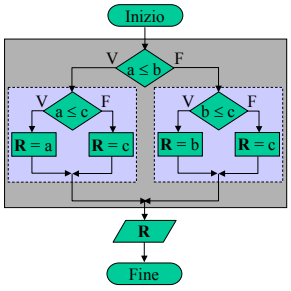
Dott. Pasquale Lops
lops@di.uniba.it

Minimo fra 3 valori

- Trovare il minore fra tre numeri reali *a*, *b*, *c*
  - Esempi:
    - *a* = 1, *b* = 2, *c* = 3
    - *a* = 20, *b* = 50, *c* = 55
    - *a* = 2, *b* = 1, *c* = 3
    - *a* = 7, *b* = 34, *c* = 13
- Operatore di confronto $x \leq y$
  - È verificato se il valore associato alla variabile *x* è minore di quello associato alla variabile *y*

Minimo fra 3 valori
Algoritmo

se $a \leq b$
  allora se $a \leq c$
    allora soluzione = *a*
    altrimenti soluzione = *c*
  altrimenti se $b \leq c$
    allora soluzione = *b*
    altrimenti soluzione = *c*



Minimo fra 3 valori
Considerazioni

- Analogo per qualunque dominio ordinale
  - Necessità di un operatore di confronto
  - Esempi:
    - Date 3 lettere, stabilire qual è la prima in ordine alfabetico
    - Dati 3 giorni della settimana, stabilire quale viene prima dal lunedì alla domenica

Scambio di valori

- Date due variabili *a* e *b*, scambiare i valori ad esse assegnati
  - Esempio:
    - $a = 12, b = 54 \quad \rightarrow \quad a = 54, b = 12$
- Operatore di assegnamento $x \leftarrow y$
  - Copia il valore associato alla variabile *y* nella memoria associata alla variabile *x*
    - Al termine i valori di *x* ed *y* coincidono
    - Il vecchio valore di *x* viene perso

Scambio di valori
Algoritmo (tentativo)

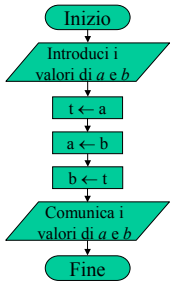
Assegna il valore di *b* ad *a*
Assegna il valore di *a* a *b*

- Trace:
  - $a = 12, b = 54$
  - $a = 54, b = 54$
  - $a = 54, b = 54$

- ***Non funziona!***
  - Dopo il primo passo il valore originario di *a* viene perso
  - Serve una variabile temporanea *t* in cui copiare il valore di *a* prima che sia sovrascritto

Scambio di valori
Algoritmo (corretto)

Assegna il valore di *a* a *t*
Assegna il valore di *b* ad *a*
Assegna il valore di *t* a *b*



– Trace:
  - *a* = 12, *t* = ??, *b* = 54
  - *a* = 12, *t* = 12, *b* = 54
  - *a* = 54, *t* = 12, *b* = 54
  - *a* = 54, *t* = 12, *b* = 12

Scambio di valori
Programma Pascal

```
program scambio(input, output);
    var a, b, t : integer;
begin
    write('Inserisci il valore di a: '); readln(a);
    write('Inserisci il valore di b: '); readln(b);
    t := a;
    a := b;
    b := t;
    writeln('Il nuovo valore di a e'' ', a);
    writeln('Il nuovo valore di b e'' ', b)
end.
```


Scambio di valori
Considerazioni

- Ad ogni passo dell'algoritmo una variabile assume sempre il valore definito dalla assegnazione più recente per essa
- L'applicazione di un algoritmo a casi particolari può aiutare a scoprire gli errori
  – Scelta oculata degli esempi critici
    - Non valori uguali per le due variabili
- Quante variabili temporanee servono per una rotazione dei valori contenuti in *n* variabili?

Conteggio

- Dato un insieme di *n* valori, contare il numero di valori maggiori di una soglia *s*
  - Esempio:
    Studenti che hanno superato un esame (*s* = 17)
    - Voti 22 30 17 25 4 18 27
    - Conteggio precedente 0 1 2 2 3 3 4
    - Conteggio attuale 1 2 2 3 3 4 **5**
- Accumulatore
  - Area di memoria interna contenente i risultati parziali dei calcoli

Conteggio

- Ad ogni passo, se il voto supera la soglia, si compiono le seguenti azioni:

  conteggio attuale ← conteggio precedente + 1
  conteggio precedente ← conteggio attuale

  - Alla destra del primo assegnamento si ha un'espressione e non una variabile
    - Il risultato andrà nell'accumulatore in attesa di assegnarlo alla variabile a sinistra

Conteggio

- Significato delle operazioni svolte dall'elaboratore per il primo assegnamento (esempio):

  conteggio attuale [4] conteggio precedente [4]
  accumulatore [4] + 1

  conteggio attuale [4] conteggio precedente [4]
  accumulatore [5]

  conteggio attuale [5] conteggio precedente [4]
  accumulatore [5]

Conteggio

- I due passi si possono condensare in:
  conteggio attuale ← conteggio attuale + 1
  – Esempio:
    conteggio attuale 4
    accumulatore 4 + 1

    conteggio attuale 5
    accumulatore 5

Conteggio
Considerazioni

- L'assegnamento è diverso dalla relazione matematica di uguaglianza
  – Il simbolo a sinistra dell'assegnamento deve essere un identificatore
    - Lo stesso simbolo, posto a destra, indica il valore prima dell'assegnamento
      – x = x + 1 non ha senso
      – x ← x + 1 ha senso
      – x + 1 ← x non ha senso

Conteggio
Algoritmo

leggi il numero *n* di voti da elaborare
inizializza il contatore a 0
mentre ci sono ancora voti da esaminare **esegui**
  leggi il voto successivo
  se voto > 17
    allora somma 1 al contatore
comunica il numero totale di promossi

Conteggio
Programma Pascal

```
program conteggio(input, output);
    const sufficienza = 18;
    var i, voto, n_voti, conteggio : integer;
begin
    write('Quanti voti vuoi inserire? '); readln(n_voti);
    conteggio := 0; i := 0;
    while i < n_voti do
        begin
            i := i + 1;
            write('Inserisci il ', i, '-mo voto: '); readln(voto);
            if voto >= sufficienza then conteggio := conteggio + 1
        end;
    writeln('Il numero di promossi e'' ', conteggio)
end.
```


Somma

- Dato un insieme di *n* numeri, progettare un algoritmo che li sommi e restituisca il totale risultante
  - Ipotesi: $n \geq 0$
  - Esempio:
    - Sommare 46, 2 e 284
      - somma ← 46 + 2 + 284
        Troppo specifico sui valori!
      - somma ← a + b + c
        Troppo specifico sul numero di valori!

Somma

- Non è possibile scrivere un'assegnazione che valga per qualunque insieme di *n* valori, ma:
  - Gli elaboratori sono adatti ai compiti ripetitivi
  - Il sommatore ha capienza di 2 numeri per volta

Somma

- Si può usare un accumulatore per contenere le somme parziali:
  - $s \leftarrow a_1 + a_2$
  - $s \leftarrow s + a_3$
  - …
  - $s \leftarrow s + a_n$
- Compattando i vari passi:

$s \leftarrow a_1 + a_2$
finché ci sono ancora numeri da sommare
  leggi il numero successivo a_{i+1}
  $s \leftarrow s + a_{i+1}$

- Non funziona per la somma di 0 o 1 valore

Somma
Algoritmo

considera il numero *n* di addendi
$s \leftarrow 0$
finché ci sono ancora numeri da sommare esegui
  leggi il numero successivo a_{i+1}
  $s \leftarrow s + a_{i+1}$
comunica che la somma finale è s

Somma
Programma Pascal

```
program somma(input, output);
    var   i, n_addendi : integer;
          addendo, somma : real;
begin
    write('Inserisci il numero di addendi: '), readln(n_addendi);
    somma := 0.0;
    for i := 1 to n do
          begin
                write('Inserisci il ', i, '-mo addendo'); readln(addendo);
                somma := somma + addendo
          end;
    writeln('La somma dei ', n_addendi, ' numeri e'' ', somma)
end.
```

Somma
Considerazioni

- Per sommare *n* numeri sono necessarie *n* – 1 addizioni
  - Se ne usano *n* per generalità
- Ad ogni iterazione *s* rappresenta la somma dei primi *i* valori
  - Per *i* = *n* si ha la somma totale
- Il valore di *i* cresce ad ogni iterazione
  - Prima o poi *i* = *n* e si uscirà dal ciclo (Terminazione)
- Non è assicurata la precisione della somma finale
  - Sommare i numeri per valore assoluto crescente

Successione di Fibonacci

- Generare i primi *n* termini della successione di Fibonacci ($n \geq 1$)
  - I primi due termini sono 0 e 1 ($F_0 = 0$; $F_1 = 1$)
  - Ogni termine successivo è ottenuto come somma degli ultimi 2 termini

    $F_i = F_{i-1} + F_{i-2}$ per $i \geq 2$
    - 1, 2, 3, 5, 8, 13, 21, 34, …

Successione di Fibonacci

- Necessario un ciclo di generazioni
  - In ogni istante, bisogna conoscere gli ultimi due termini generati per generare il successivo
    - Ultimo
    - Penultimo
  - Dopo la generazione
    - Il termine appena generato diventa l'ultimo
    - Il termine che prima era l'ultimo diventa penultimo

Successione di Fibonacci
Algoritmo

INIZIO
Acquisire il numero di termini da generare *n;*
primo (e penultimo) termine ← 0;
secondo (e ultimo) termine ← 1;
termini generati ← 2;
MENTRE (termini generati < *n)* **ESEGUI**
  nuovo termine ← penultimo + ultimo;
  visualizza nuovo termine;
  penultimo ← ultimo; ultimo ← nuovo termine;
  termini generati ← termini generati + 1
FINE_MENTRE
FINE

Successione di Fibonacci
Programma Pascal

```
program fibonacci(input, output);
   var i, n, nuovo, ultimo, penultimo: integer;
begin
   write('Inserisci il numero di termini da generare: '); readln(n);
   penultimo := 0; ultimo := 1; i := 2;
   writeln('I primi due termini sono: ', penultimo, ' e ', ultimo);
   while i < n do
        begin
                nuovo := ultimo + penultimo;
                writeln('Il termine successivo e'' ', nuovo);
                penultimo := ultimo; ultimo := nuovo; i := i + 1
        end
end.
```


Massimo Comune Divisore

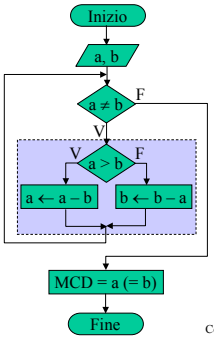
- Il MCD di due numeri naturali *x* e *y* si può ottenere per sottrazioni successive
  - MCD(15, 3) è 3
  - MCD(15, 2) è 1
- Valgono le relazioni:
  - MCD (x, y) = MCD (x-y, y) con x > y
  - MCD (x, x) = x

Algoritmo per il MCD

- ***Considera*** la coppia di numeri dati
- **Mentre** i numeri sono diversi **esegui**
  - **Se** il primo numero è minore del secondo **allora**
    - Scambiali
  - **Sottrai** il secondo dal primo
  - **Rimpiazza** i due numeri col sottraendo e con la differenza, rispettivamente
- Il ***risultato*** è il valore ottenuto

NB: termina quando i due numeri sono uguali

Algoritmo per il MCD



INIZIO

a ← x; b ← y;

oppure leggi a, b;

MENTRE a ≠ b **ESEGUI**

SE a > b

ALLORA a ← a-b

ALTRIMENTI b ← b-a

FINE

MCD

- Ci sono molte versioni dell'algoritmo per calcolare il MCD
- Algoritmo euclideo

  MCD (x, y) = MCD (x mod y, x)

  mod è il resto intero associato alla divisione intera fra x ed y

Sommatoria di Gauss

- Somma dei primi *n* numeri interi

$$1 + 2 + \ldots + (n-1) + n = \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

Altri algoritmi iterativi

RICORRERE ALLA DEFINIZIONE

- Somma di due numeri naturali *x* ed *y* (tramite incrementi unitari)
- Sottrazione di due numeri naturali *x* ed *y,* con $x > y$ (tramite decrementi unitari)
- Moltiplicazione di due numeri naturali *x* ed *y* (tramite addizioni successive)
- Divisione intera di due numeri naturali *x* ed *y* (tramite sottrazioni successive)

Inversione delle Cifre di un Intero

- Progettare un algoritmo che accetta un intero positivo e ne inverte l'ordine delle cifre
  - Esempio
    27953 → 35972
- Operatori DIV e MOD
  - Calcolano, rispettivamente, il quoziente intero ed il resto di una divisione fra interi

Inversione delle Cifre di un Intero

- Serve accedere alle singole cifre del numero
  - Non è nota a priori la lunghezza del numero
    - Iniziare dalla cifra meno significativa
- Eliminare progressivamente le cifre dal numero originario
  - Procedere da destra verso sinistra
- Accodarle progressivamente al numero invertito
  - Far scalare le precedenti verso sinistra

Inversione delle Cifre di un Intero

- Individuazione della cifra meno significativa
  - c = n MOD 10
- Rimozione della cifra meno significativa
  - n' = n DIV 10
- Scalare verso sinistra l'inversione parziale per aggiungere la nuova cifra
  - m * 10 + c
- Ripetere fino a quando non ci sono più cifre
  - Quoziente della divisione pari a 0

Inversione delle Cifre di un Intero Algoritmo

INIZIO
Stabilire *n;*
Predisporre a 0 l'intero rovesciato;
MENTRE l'intero da rovesciare è maggiore di 0 ESEGUI
  Estrarre dall'intero da rovesciare la cifra meno significativa come resto della divisione per 10;
  Aggiornare l'intero rovesciato moltiplicandolo per 10 e aggiungendo la cifra estratta;
  Eliminare dall'intero da rovesciare la cifra estratta dividendolo per 10
FINE_MENTRE
FINE

Inversione delle Cifre di un Intero
Programma Pascal

```
PROGRAM inverti_intero(input, output);
   VAR numero, inverso: integer;
BEGIN
   write('Inserisci l'intero da invertire: '),
   readln(numero);
   inverso := 0;
   WHILE numero > 0 do
        BEGIN
                inverso := inverso * 10 + numero MOD 10;
                numero := numero DIV 10
        END;
   writeln('Il numero invertito e'' ', inverso)
END.
```


Complessità di un Algoritmo

- Quantità di risorse usate dall'algoritmo
- Dipendente dalla dimensione del problema
  - Spazio
    - Quantità di memoria occupata durante l'esecuzione
  - Tempo
    - Quantità di tempo impiegata per ottenere la soluzione
      - Calcolabile in base al numero di volte in cui viene ripetuta l'operazione principale
        - » Esempio: Confronti, Scambi, Addizioni, …

Complessità di un Algoritmo

- Diversi casi
  - Migliore
    - Corrispondente alla configurazione iniziale che comporta il minimo numero di esecuzioni dell'operazione principale
  - Peggiore
    - Corrispondente alla configurazione iniziale che comporta il massimo numero di esecuzioni dell'operazione principale
  - Medio

Complessità di un Algoritmo
Livelli

- Complessità crescente:
  - Costante $O(1)$
  - Logaritmica $O(\log n)$
  - Lineare $O(n)$
  - nlog $O(n \log n)$
  - Polinomiale $O(n^m)$
    - Quadratica $O(n^2)$
    - Cubica $O(n^3)$
    - …
  - Esponenziale $O(k^n)$ $k > 1$

Stampa del set di caratteri

- Stampare il set di caratteri dell'elaboratore che si sta utilizzando
- I primi 32 caratteri non sono visualizzabili
- Utilizzare la funzione *chr*

Stampa del set di caratteri
Programma Pascal

```
PROGRAM CharacterSet (input,output);
VAR i:0..127;

BEGIN
   writeln(' ':20, 'Set di caratteri ASCII dal codice 32');
   FOR i:=32 TO 127 DO
        BEGIN
              write (i:5, chr(i):2);
              IF i mod 10 = 0 THEN writeln;
        END;
END.
```

Algoritmi su Array

- Algoritmi di base
- Algoritmi di supporto
  - Partizionamento
  - Fusione
- Ricerca
  - Basata su un ordinamento
- Ordinamento
  - Finalizzato alla ricerca

Trattamento degli Array

- Lettura di una variabile di tipo array: bisogna leggere una componente alla volta tramite ***iterazione*** (*while, repeeat, for*)
- Elaborazione
- Scrittura
- Ad ogni iterazione bisogna controllare che l'indice non superi il massimo valore consentito dal tipo dell'indice
- Il numero delle iterazioni è noto e coincide con il numero delle componenti dell'array

Trattamento degli Array

inizializza l'indice

Mentre c'è un nuovo dato e l'array non è pieno

incrementa l'indice

leggi il nuovo dato e inseriscilo nell'array

- Al termine l'indice specifica il numero di elementi

Trattamento degli Array

- Molto spesso è necessario trattare *tutte* le componenti di un array in modo uniforme (es. inizializzare tutte le componenti a zero, leggere tutte le componenti, etc.).
- In tali situazioni ha senso elaborare le componenti di un array in *sequenza*, partendo dal primo elemento fino all'ultimo
- Ciò si realizza utilizzando strutture iterative

Trattamento degli Array

- Inizializzare un array ad un valore costante
  - **FOR** i:=1 to 10 **DO** vet[i]:=0
- Copiare
  - **FOR** i:=1 to 10 **DO** vet_1[i]:=vet_2[i]
- Leggere
  - **FOR** i:=1 to 10 **DO** read(vet[i])
- Visualizzare
  - **FOR** i:=1 to 10 **DO** write(vet[i])

Massimo

- Trovare il massimo valore in un insieme di *n* elementi
  - Il computer non può guardare globalmente i valori nell'insieme
    - Analizzarli uno per volta
      - Necessità di esaminarli tutti per trovare il massimo
    - Ricordare in ogni momento il massimo parziale
      - Guardando solo il primo, è il più alto che si sia visto
      - Guardando i successivi, si aggiorna il massimo ogni volta che se ne trova uno maggiore del massimo parziale

Massimo
Algoritmo

- Inserire il numero di valori
- Leggere il primo
- Porre il massimo al primo
- Mentre non si sono letti tutti gli elementi
  - Leggere il successivo
  - Se è maggiore del massimo
    - Porre il massimo a questo numero
- Comunicare il massimo

Massimo
Considerazioni

- Necessari $n - 1$ confronti
- Si devono analizzare tutti gli elementi fino all'ultimo
  - Noto se l'insieme è conservato in un array
  - Ignoto se la sequenza di valori viene inserita progressivamente
    - Va chiesto quanti elementi compongono la sequenza

Massimo
Programmi Pascal

- Aggiungere dichiarazioni, (acquisizione dell'array) e stampa del risultato

```
begin
    massimo := valori[1]
    for i := 2 to n_valori do
        if valori[i] > massimo then
            massimo := valori[i]
end.
```

```
begin
    write('Inserisci il numero di
        valori: '); readln(n_valori);
    i := 1;
    write('Inserisci il prossimo
        valore: '); readln(valore);
    massimo := valore;
    for i := 2 to n_valori do
    write('Inserisci il prossimo
        valore: '); readln(valore);
        if valore > massimo then
            massimo := valore
end.
```

Inversione

- Risistemare gli elementi di un array di dimensione *n*, in modo tale che appaiano al contrario

  | a | k | h | e | x | b | h | → | h | b | x | e | h | k | a |

  – Inserirli in ordine inverso in un array di appoggio, quindi ricopiarlo in quello originale?
    - Spreco di memoria
    - Spreco di tempo

Inversione

- Effetto dell'inversione
  – Il primo diventa l'ultimo
  – Il secondo diventa il penultimo
  – …
  – Il penultimo diventa il secondo
  – L'ultimo diventa il primo
- Soluzione: scambiare il primo con l'ultimo, il secondo col penultimo, ecc.

Inversione

- 2 indici
  – Partono dagli estremi dell'array
  – Procedono verso l'interno
    - Ogni volta che il primo viene incrementato, il secondo viene decrementato
  – Si scambiano i valori via via incontrati
- Basta un solo indice
  – Conta la distanza dagli estremi (sia destro che sinistro)
    - (i, n – i)

Inversione

- Proviamo
  - $i = 1 \rightarrow (1, n - 1)$
    - *n* è stato saltato NON FUNZIONA!!!
    - Per considerarlo si può aggiungere 1
- Riproviamo
  - $i = 1 \rightarrow (1, n - 1 + 1) = (1, n)$
  - $i = 2 \rightarrow (2, n - 2 + 1) = (2, n - 1)$
  - …
- OK!

Inversione

- Quando fermarsi?
  - Ogni elemento si scambia col simmetrico
    - Se sono pari, gli scambi sono la metà degli elementi
    - Se sono dispari, quello centrale resterà al centro
      - E' già sistemato, non si scambia
  - $\lfloor n/2 \rfloor$ scambi

Inversione
Algoritmo e Programma Pascal

- Stabilire l'array di *n* elementi da invertire
- Calcolare il numero di scambi *r* = *n* DIV 2
- Mentre il numero di scambi *i* è minore di *r*
  - Scambiare l'*i*-mo elemento con l'(*n*–*i*+1)-mo
- Restituire l'array invertito

```
begin
   r := n DIV 2;
   for i := 1 to r do
         begin
                  t := a[i];
                  a[i] := a[n–i+1];
                  a[n–i+1] := t
         end
end.
```

- Aggiungere dichiarazioni, acquisizione dell'array e stampa del risultato

Inversione
Considerazioni

- Necessari $\lfloor n/2 \rfloor$ scambi
  - Cosa succederebbe se si facessero *n* scambi?
- L'algoritmo termina
  - Numero di iterazioni definito
- Spesso non è necessario eseguire lo scambio fisico
  - Sufficiente agire sugli indici

Rimozione dei Duplicati

- Dato un array ordinato, compattarlo in modo che i valori duplicati siano rimossi

  | 3 | 3 | 5 | 8 | 10 | 10 | 10 | 14 | 20 | 20 |

  → | 3 | 5 | 8 | 10 | 14 | 20 |

  - I doppi sono adiacenti
    - Saltarli
      - Confrontare elementi a coppie
    - Spostare il successivo valore distinto nella posizione seguente al più recente valore distinto
      - Contatore degli elementi distinti

Rimozione dei Duplicati
Algoritmo

Definire l'array di *n* elementi
Impostare a 2 l'indice di scansione
Finché si trovano valori distinti
    Confrontare coppie di elementi adiacenti
Impostare il numero di valori distinti
Finché ci sono coppie da esaminare
    Se la prossima coppia non ha duplicati
        Incrementa il contatore dei valori distinti
        Sposta l'elemento di destra della coppia in tale posizione

Rimozione dei Duplicati
Note Implementative

- Inizializzazione dell'indice di scansione
  - Prima posizione dell'array
    - Confronti col valore successivo
  - Seconda posizione dell'array
    - Confronti col valore precedente
- Seconda opzione più intuitiva
  - Consente di terminare il ciclo ad *n*

Rimozione dei Duplicati
Programma Pascal

```
begin
    i := 2;
    while (a[i–1] <> a[i]) and (i < n) do i := i + 1;
    if a[i–1] <> a[i] then i := i + 1;
    j := i – 1;
    while i < n do
        begin
            i := i + 1;
            if a[i–1] <> a[i] then
                begin    j := j + 1; a[j] := a[i]        end
        end;
    n := j
end.
```


Ricerca

- Determinare se (e dove) un certo elemento *x* compare in un certo insieme di *n* dati
  - Supponiamo gli elementi indicizzati 1 … *n*
    - Il fatto che l'elemento non sia stato trovato è rappresentabile tramite il valore di posizione 0
    - Deve funzionare anche per 0 elementi
  - Possibili esiti:
    - Elemento trovato nell'insieme
      - Restituirne la posizione
    - Elemento non presente nell'insieme

Ricerca

- Tipicamente effettuata sulla chiave di un record
  - Reperimento delle restanti informazioni
    - Esempio:
      Ricerca delle informazioni su un libro per codice ISBN

Ricerca Sequenziale

- Scorrimento di tutti gli elementi dell'insieme, memorizzando eventualmente la posizione in cui l'elemento è stato trovato
  - Nessuna ipotesi di ordinamento
  - Utilizzabile quando si può accedere in sequenza agli elementi della lista

Ricerca Lineare Esaustiva
Algoritmo

- Scandisce tutti gli elementi della lista
  - Restituisce l'ultima (posizione di) occorrenza
  - Utile quando si vogliono ritrovare tutte le occorrenze del valore

```
j ← 0
posizione ← 0
mentre j < n
   j ← j + 1
   se lista(j) = x allora posizione ← j
```

Ricerca Lineare Esaustiva
Programma Pascal

- Completare con le dichiarazioni

```
begin
    j := 0;
    posizione := 0;
    while j < n do
        begin
            j := j + 1;
            if a[j] = x then posizione := j
        end
end
```


Ricerca Lineare Esaustiva
Considerazioni

- Complessità
  - Basata sul numero di confronti
    - In ogni caso: *n*
      - Si devono controllare comunque tutti gli elementi fino all'ultimo
- Soluzione più naturale
  - A volte non interessa scandire tutta la lista
    - Ci si può fermare appena l'elemento viene trovato
    - Possibilità di migliorare l'algoritmo

Ricerca Lineare con Sentinella
Algoritmo

- Si ferma alla prima occorrenza del valore
  - Restituisce la prima (posizione di) occorrenza
  - Utile quando
    - Si è interessati solo all'esistenza, oppure
    - Il valore, se esiste, è unico

```
j ← 0
posizione ← 0
mentre (j < n) e (posizione = 0)
    j ← j + 1
    se lista(j) = x allora posizione ← j
```

Ricerca Lineare con Sentinella
Programma Pascal

- Completare con le dichiarazioni

```
begin
    j := 0;
    posizione := 0;
    while j < n and posizione = 0 do
        begin
            j := j + 1;
            if a[j] = x then posizione := j
        end
end
```


Ricerca Lineare con Sentinella
Considerazioni

- Complessità
  - Basata sul numero di confronti
    - Caso migliore: 1
      - Elemento trovato in prima posizione
    - Caso peggiore: *n*
      - Elemento in ultima posizione o assente
    - Caso medio: (*n* + 1) / 2
      - Supponendo una distribuzione casuale dei valori
- Ottimizzato per i casi in cui è applicabile

Ricerca Binaria
o Dicotomica

- Analizzare un valore interno alla lista, e se non è quello cercato basarsi sul confronto per escludere la parte superflua e concentrarsi sull'altra parte
  - Applicabile a insiemi di dati ordinati
    - Guadagno in efficienza
    - Incentivo all'ordinamento
  - Richiede l'accesso diretto

Ricerca Binaria

- Scelta della posizione da analizzare
  - Più vicina ad uno dei due estremi
    - Caso migliore: restringe più velocemente il campo
    - Caso peggiore: elimina sempre meno elementi
  - Centrale
    - Compromesso che bilancia al meglio i casi possibili
- Necessità di ricordare la porzione valida
  - Prima posizione
  - Ultima posizione

Ricerca Binaria
Algoritmo

Mentre la parte di lista da analizzare contiene più di un elemento e quello cercato non è stato trovato
  Se l'elemento centrale è quello cercato
    allora è stato trovato in quella posizione
    altrimenti se è minore di quello cercato
      allora analizzare la metà lista successiva
      altrimenti analizzare la metà lista precedente

Ricerca Binaria
Esempio

$x = 29$ | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | 38 | 44 | 53 | 61 |
12 elementi

I tentativo ~~| 2 | 4 | 7 | 11 | 24 | 25 |~~ 29 | 32 | 38 | 44 | 53 | 61 |
↑ Eliminati 6

II tentativo | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | ~~38 | 44 | 53 | 61 |~~
↑ Eliminati 4

III tentativo | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | 38 | 44 | 53 | 61 |
↑ Trovato!

- Se fosse stato 31: non trovato in 4 tentativi

Ricerca Binaria
Programma Pascal

```
begin
    posizione := 0;
    first := 1; last := n;
    while (first <= last) and (posizione = 0) do
        begin
            j := (first + last) DIV 2;     (* per difetto *)
            if lista[j] = x then
                posizione := j
            else
                if lista[j] < x then first := j + 1 else last := j – 1
        end
end
```


Ricerca Binaria
Considerazioni

- Complessità
  - Numero minimo di accessi: 1
    - Valore trovato al centro della lista
  - Numero massimo di accessi: $\lfloor \log_2 n \rfloor + 1$
    - Massimo numero possibile di divisioni per 2 partendo da n
      - Esempio: $n = 128 \rightarrow \log_2 n = 7$
    - Tempo per il calcolo di j trascurabile
- Usata per consultare dizionari, elenchi telefonici, …

Ordinamento

- Disporre gli elementi di un insieme secondo una prefissata relazione d'ordine
  - Dipendente dal tipo di informazione
    - Numerica
      - Ordinamento numerico
    - Alfanumerica
      - Ordinamento lessicografico
  - 2 possibilità
    - Crescente
    - Decrescente
  - Altri criteri personali

Ordinamento

- Possibilità di avere elementi costituiti da più componenti
  - Associazione a ciascuno di una *chiave*
    - Lo identifica univocamente
    - Stabilisce la sua posizione
    - Unica componente rilevante per l'ordinamento
  - Esempio: Record costituito da più campi
- Supponiamo, nel seguito, di richiedere un ordinamento numerico crescente
  - Indicazione della sola chiave

Ordinamento

- Gran varietà di algoritmi
  - Basati su confronti e scambi fra gli elementi
    - Relazione d'ordine, criterio
  - Non esiste uno migliore in assoluto
    - La bontà dipende da fattori connessi ai dati su cui deve essere applicato
      - Dimensione dell'insieme di dati
        - » Numerosità
      - Grado di pre-ordinamento dell'insieme di dati
        - » Già ordinato, parzialmente, ordine opposto, casuale

Ordinamento

- Una delle attività di elaborazione più importanti
  - Stima: 30% del tempo di calcolo di un elaboratore
- Obiettivo: efficienza
  - Sfruttare "al meglio" i confronti ed i conseguenti spostamenti degli elementi
    - Piazzare gli elementi prima possibile più vicino alla loro posizione finale nella sequenza ordinata

Ordinamento

- Algoritmi esterni
  - Usano un array di appoggio
    - Occupazione di memoria doppia
    - Necessità di copiare il risultato nell'array originale
- Algoritmi interni
  - Eseguono l'ordinamento lavorando sullo stesso array da ordinare
    - Basati su scambi di posizione degli elementi

Ordinamento Esterno

- Enumerativo
  - Ciascun elemento confrontato con tutti gli altri per determinare il numero degli elementi dell'insieme che sono più piccoli in modo da stabilire la sua posizione finale
- Non verranno trattati

Ordinamento Interno

- Per Selezione
  - Elemento più piccolo localizzato e separato dagli altri, quindi selezione del successivo elemento più piccolo, e così via
- A bolle
  - Coppie di elementi adiacenti fuori ordine scambiate, finché non è più necessario effettuare alcuno scambio
- Per Inserzione
  - Elementi considerati uno alla volta e inseriti al posto che gli compete all'interno degli altri già ordinati

Ordinamento per Selezione

- Minimi successivi
  - Trovare il più piccolo elemento dell'insieme e porlo in prima posizione
    - Scambio con l'elemento in prima posizione
  - Trovare il più piccolo dei rimanenti ($n - 1$) elementi e sistemarlo in seconda posizione
  - …
  - Finché si trovi e collochi il penultimo elemento
    - Ultimo sistemato automaticamente

Ordinamento per Selezione
Esempio

	Inizio	**I**	**II**	**III**	**IV**	**V**
array(1)	44	44	11	11	11	11
array(2)	33	33	33	22	22	22
array(3)	66	66	66	66	33	33
array(4)	11	(11)	44	44	(44)	44
array(5)	55	55	55	55	55	(55)
array(6)	22	22	(22)	(33)	66	66

Ordinamento per Selezione
Algoritmo

$i \leftarrow 1$
mentre $i < n$ **esegui**
  trova il minimo di lista (i … n)
  scambia la posizione del minimo con lista(i)
  $i \leftarrow i + 1$

- Algoritmi già noti
  - Ricerca del minimo
  - Scambio

Ordinamento per Selezione
Programma Pascal

```
begin
    for i := 1 to n – 1 do
        begin
            min := a[i]; p := i;
            for j := i + 1 to n do
                if a[j] < min then
                    begin    min := a[j]; p := j end;
            a[p] := a[i];
            a[i] := min
        end
end.
```


Ordinamento per Selezione
Complessità

- Confronti
  - Sempre $(n – 1) * n / 2 \sim O(n^2)$
    - $n – 1$ al I passo di scansione
    - $n – 2$ al II passo di scansione
    - …
    - 1 all'$(n – 1)$-mo passo di scansione
- Scambi
  - Al più $(n – 1)$
    - 1 per ogni passo

Ordinamento per Selezione
Considerazioni

- Ogni ciclo scorre tutta la parte non ordinata
- Numero fisso di confronti
  - Non trae vantaggio dall'eventuale preordinamento
- Pochi scambi

Ordinamento a Bolle

- Far "affiorare" ad ogni passo l'elemento più piccolo fra quelli in esame
  - Confronto fra coppie di elementi adiacenti e, se sono fuori ordine, scambio, ripetendo il tutto fino ad ottenere la sequenza ordinata
  - Simile alle bolle di gas in un bicchiere
- Il passo di esecuzione coincide con:
  - Il numero di elementi già ordinati
  - Elemento a cui fermarsi ad ogni passo

Ordinamento a Bolle

- Se in una passata non viene effettuato nessuno scambio, l'insieme è già ordinato
  - L'algoritmo può già terminare
    - Meno di $n - 1$ passi
- Miglioramento:
  - Usare un indicatore di scambi effettuati
    - Impostato a vero all'inizio di ogni passata
    - Impostato a falso non appena si effettua uno scambio
  - Si termina se alla fine di un passo è rimasto inalterato

Ordinamento a Bolle

Esempio

	Inizio	**I/1**	**I/2**	**I/3**	**I/4**	**II**
array(1)	4	4	4	4	4	1
array(2)	3	3	3	3	1	4
array(3)	6	6	6	1	3	3
array(4)	1	1	1	6	6	6
array(5)	5	2	2	2	2	2
array(6)	2	5	5	5	5	5

Ordinamento a Bolle
Esempio

	II	II/1	II/2	II/3	III
array(1)	1	1	1	1	1
array(2)	4	4	4	4	2
array(3)	3	3	3	2	4
array(4)	6	6	2	3	3
array(5)	2	2	6	6	6
array(6)	5	5	5	5	5

Ordinamento a Bolle
Esempio

	III	III/1	III/2	IV	IV/1	Fine
array(1)	1	1	1	1	1	1
array(2)	2	2	2	2	2	2
array(3)	4	4	4	3	3	3
array(4)	3	3	3	4	4	4
array(5)	6	5	5	5	5	5
array(6)	5	6	6	6	6	6

Ordinamento a Bolle
Algoritmo

```
p ← 0    (* passo di esecuzione *)
ordinato ← falso
mentre p < n e non ordinato esegui
    p ← p + 1
    ordinato ← vero
    i ← n
    mentre i > p esegui
        se lista(i) < lista (i – 1) allora
                scambia lista(i) con lista(i – 1)
                ordinato ← falso
        i ← i – 1
```

Ordinamento a Bolle
Programma Pascal

```
begin
    sorted := false; p := 0;
    while (p < n) and not sorted do
        begin
            sorted := true; p := p + 1;
            for i := n downto p + 1 do
                if a[i] < a[i–1] then
                    begin
                        t:= a[i]; a[i] := a[i–1]; a[i–1] := t;
                        sorted := false
                    end
        end
end.
```


Ordinamento a Bolle
Complessità

- Caso migliore (lista già ordinata): 1 passo
  - $n - 1$ confronti, 0 scambi
- Caso peggiore (ordine opposto): $n - 1$ passi
  - All'i-mo passo
    - $n - i + 1$ confronti $\rightarrow$ in tutto $(n - 1) * n / 2 \sim O(n^2)$
      - Come per Selezione
    - $n - i + 1$ scambi $\rightarrow$ in tutto $(n - 1) * n / 2 \sim O(n^2)$
      - Molto maggiore della Selezione
- Caso medio
  - Scambi pari alla metà dei confronti $\rightarrow O(n^2)$

Ordinamento a Bolle
Considerazioni

- Ogni ciclo scorre tutta la parte non ordinata
- Prestazioni medie inferiori agli altri metodi
  - Nel caso peggiore, numero di confronti uguale all'ordinamento per selezione, ma numero di scambi molto maggiore
  - Molto veloce per insiemi con alto grado di preordinamento

Ordinamento per Inserzione

- Ricerca la giusta posizione d'ordine di ogni elemento rispetto alla parte già ordinata
  - Inizialmente già ordinato solo il primo elemento
  - Elementi da ordinare considerati uno per volta
    - Necessari $n - 1$ passi
- Metodo intuitivo
  - Simile all'ordinamento eseguito sulle carte da gioco

Ordinamento per Inserzione

- Determinare la posizione in cui inserire la chiave nella sequenza ordinata, facendo scalare le altre
  - Scansione sequenziale
- Soluzione completa costruita inserendo un elemento della parte non ordinata nella parte ordinata, estendendola di un elemento

| parte ordinata | x | parte non ordinata | $\rightarrow$

| $\leq x$ | x | $> x$ | parte non ordinata |

Ordinamento per Inserzione
Note Implementative

- Strategia di scelta del prossimo elemento da inserire nella parte ordinata
  - Primo elemento della parte non ordinata
- Variante (Dromey)
  - Inserire subito il più piccolo in prima posizione
    - Evita di dover effettuare appositi controlli sull'indice per evitare che esca fuori dall'array

Ordinamento per Inserzione
Esempio

	Inizio	I	II	III	IV	V
array(1)	40	30	30	10	10	10
array(2)	(30)	40	40	30	30	20
array(3)	60	(60)	60	40	40	30
array(4)	10	10	(10)	60	50	40
array(5)	50	50	50	(50)	60	50
array(6)	20	20	20	20	(20)	60

Ordinamento per Inserzione
Algoritmo

per ogni elemento dal secondo fino all'ultimo **esegui**
    inserito ← falso
    mentre non è stato inserito **esegui**
        se è minore del precedente **allora**
            fai scalare il precedente
            se sei arrivato in prima posizione **allora**
                piazzalo; inserito ← vero
            fine_se (necessario per evitare ambiguità)
        altrimenti
            piazzalo; inserito ← vero
        fine_se
    fine_mentre
Fine_perogni

Ordinamento per Inserzione
Implementazione Pascal

```
begin
for i := 2 to n do
    begin
    x := a[i]; j := i–1; inserito := false;
    while not inserito do
        if x < a[j] then
            begin
            a[j+1] := a[j]; j := j – 1;
            if j <= 0 then    begin    a[1] := x; inserito := true    end
            end
        else
            begin    a[j + 1] := x; inserito := true          end
    end
end.
```

Ordinamento per Inserzione
Algoritmo

Cerca il minimo
Prima posizione ← minimo
mentre c'è una parte non ordinata
  Considera il primo elemento di tale parte
  Confrontalo a ritroso con i precedenti, facendoli via via scalare finché sono maggiori

- Gli elementi via via incontrati scandendo a ritroso la parte ordinata scalano per far spazio all'elemento da inserire

Ordinamento per Inserzione
Programma Pascal

```
begin
    min := a[1]; p := 1;
    for i := 2 to n do
      if a[i] < min then
        begin    min := a[i]; p := i          end;
    a[p] := a[1]; a[1] := min;        (* i primi 2 sono ordinati *)
    for i := 3 to n do
      begin
        x := a[i]; j := i;
        while x < a[j–1] do
                begin    a[j] := a[j–1]; j := j – 1     end;
        a[j] := x
      end
end.
```


Ordinamento per Inserzione
Complessità

- Sempre $n - 1$ passi
  - Uno scambio per ogni confronto, salvo (eventualmente) l'ultimo
  - Caso ottimo (lista già ordinata)
    - $n - 1$ confronti, 0 scambi
      - Come il metodo a bolle
  - Caso pessimo (ordine opposto)
    - i-mo passo
      - $i - 1$ confronti e scambi -> $(n - 1) * n / 2 \sim O(n^2)$
      - Caso medio: metà confronti e scambi

Ordinamento per Inserzione
Considerazioni

- Per sequenze con distribuzione casuale
  - Molti confronti
  - Molti scambi
- Caso migliore come l'ordinamento a bolle
  - Valido per
    - Piccole sequenze ($n \leq 25$)
    - Sequenze note a priori essere parzialmente ordinate
- Ogni ciclo scorre una porzione della parte ordinata

Ordinamento
Considerazioni

- Scambio più costoso del confronto
  - Confronto operazione base del processore
  - Scambio composto da tre assegnamenti
    - Un assegnamento richiede due accessi alla memoria
- Ad ogni passo
  - La porzione ordinata cresce di una unità
  - La porzione disordinata decresce di una unità

Matrici
Parte dichiarativa

```
const
  N = 2;
  P = 3;
  M = 4;

var
  mat1: array[1..N, 1..P] of real;     { prima matrice }
  mat2: array[1..P, 1..M] of real;     { seconda matrice }
  pmat: array[1..N, 1..M] of real;     { matrice prodotto }
```

Matrici
Inizializzazione

```
writeln('INIZIALIZZAZIONE DELLA PRIMA MATRICE');

for i:=1 to N do
 for j:=1 to P do
 begin
  write('Inserisci elemento in riga ', i, ' colonna ', j, ' val: ');
  readln(mat1[i,j]);
 end;
```


Matrici
Stampa

```
writeln('STAMPA MATRICE');

for i:=1 to N do
        begin
        for j:=1 to M do
                write(pmat[i,j]:5:2);
        writeln
        end
end;
```


Matrici
Matrice Trasposta

- Esempio
  - Matrice A: 2 righe, 3 colonne
  - Matrice T(A): 3 righe, 2 colonne
    gli elementi sono trasposti dalle righe alle colonne

$$\mathbf{T}\left(\begin{array}{|c|c|c|}\hline 2 & 3 & 4 \\ \hline 1 & 4 & 8 \\ \hline \end{array}\right) = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline 4 & 8 \\ \hline \end{array}$$

Matrici
Matrice Trasposta - Calcolo

Input: mat **matrice** N x M
Output: **t**mat **matrice** M x N

Per i **da** 1 **a** M
  Per j **da** 1 **a** N
    Assegna all'elemento
    $tmat_{i\ j}$ l'elemento $mat_{j\ i}$
  cicla
cicla

in Pascal:

tmat: array[1..M,1..N] of real;

```
for i:=1 to M do
  for j:=1 to N do
    tmat[i, j] := mat[j,i]
```


Matrici
Somma

- Esempio
  - Matrice A: 2 righe, 3 colonne
  - Matrice B: 2 righe, 3 colonne
  - Matrice A + B: 2 righe, 3 colonne

2	3	4
-6	4	8

+

0	-4	5
-1	3	2

=

2	-1	9
-7	7	10

Matrici
Somma - Algoritmo

Input: mat1 **e** mat2
Output: smat

Per i **da** 1 **a** N
  Per j **da** 1 **a** M
    Assegna all'elemento $smat_{i\ j}$ la somma dell'elemento $mat1_{i\ j}$ con l'elemento $mat2_{j\ j}$
  cicla
cicla

In Pascal:

smat: array[1..N,1..M] of real;

```
for i:=1 to N do
  for j:=1 to M do
    smat[i, j] := mat1[i, j] + mat2[i, j];
```

Matrici
Somma: programma Pascal

```
for i:=1 to N do
  for j:=1 to M do
     pmat[i, j] := mat1[i, j] + mat2[i, j];
```


Matrici
Prodotto

- Calcolare il prodotto di due matrici
  - Matrice A: *N* righe, *P* colonne
  - Matrice B: *P* righe, *M* colonne
  - Matrice Prodotto = A x B: *N* righe, *M* colonne

Matrici
Prodotto (2)

- Matrice A: 2 righe, 3 colonne
- Matrice B: 3 righe, 4 colonne
- Matrice A x B: 2 righe, 4 colonne

2	3	4
1	4	8

x

1	2	5	2
0	1	7	3
4	0	1	2

=

18	7	35	21
33	6	41	30

Matrici
Prodotto - Algoritmo

Input: mat1 **e** mat2

Output: pmat

Per i **da** 1 **fino a** N
  Per j **da** 1 **fino a** M
    Inizializza accumulatore a zero
    Per k **da** 1 **fino a** P
      Aggiungi all'accumulatore il prodotto
        dell'elemento $mat1_{i\ k}$ per l'elemento $mat2_{k\ j}$
      Assegna il valore finale dell'accumulatore all'elemento $pmat_{i\ j}$
    cicla
  cicla
cicla

Matrici
Prodotto: programma Pascal

```
for i:=1 to N do
  for j:=1 to M do begin
    pmat[i, j] := 0;
    for k:=1 to P do
      pmat[i, j] := pmat[i, j] + mat1[i, k] * mat2[k, j];
  end;
```


Matrici
Ricerca

- Cercare un elemento “x” in una matrice
  - se “x” è all’interno della matrice, stampare le coordinate della sua prima occorrenza
- altrimenti stampare l’esito negativo

Matrici
Ricerca - Algoritmo

Input: mat matrice N x M
x numero reale
Output: (riga, colonna) ∈ [1..N] x [1..M]

i ← 0
trovato ← falso

mentre i < N **e non** trovato **esegui**
incrementa i
j ← 0
mentre j < M **e non** trovato **esegui**
incrementa j
se mat(i,j) è uguale a x **allora** trovato ← vero
fine_mentre
fine_mentre
se non trovato **allora stampa** non trovato
altrimenti stampa x trovato in posizione (i,j)

Matrici
Ricerca: programma Pascal

```
var     mat: array [1..N,1..M] of real;
        x: real;
...
i := 0; trovato := false;
while (i < N) and not trovato do
        begin
        i:=i+1; j := 0;
        while (j < M) and not trovato do
            begin
            j := j+1;
            if mat[i,j] = x then trovato := true
            end;
        end;
if not trovato thenwriteln('non trovato')
                else writeln(x,'trovato in posizione (',i,',',j,')');
```

per esercizio. usare repeat..until anziché while..do

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
  - Algoritmi migliori 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 ≤ *x* siano in un sottoinsieme
  - Gli elementi > *x* siano nell'altro sottoinsieme
- Esempio
  | 4 | 13 | 28 | 18 | 7 | 36 | 11 | 24 | x = 20

Partizionamento

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

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

  ←——≤ 20——→ ←—> 20—→
  - Non richiesto
  - Non necessario
- Il problema sarebbe risolto anche con la seguente configurazione finale

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

  ←——≤ 20——→ ←—> 20—→

Partizionamento

- Nota la posizione finale *p* che divide i due sottoinsiemi
  - Alcuni elementi non vanno spostati
    - Scorrendo l'array da sinistra verso *p*, i valori ≤ *x*
    - Scorrendo l'array da destra versi *p*, i valori > *x*
  - Alcuni elementi sono fuori posto
    - Per ognuno da una parte, uno dall'altra
      - Scambio
- La posizione *p* può essere ricavata
  - Contare gli elementi ≤ *x*

Partizionamento

- Occorrono 2 scansioni dell'array
  - Determinazione della posizione *p*
  - Individuazione e scambio degli elementi fuori posto
    - Si termina quando si arriva a *p*
- È possibile scandire l'array una sola volta

  Mentre le due partizioni non si incontrano

  Estendere le partizioni sinistra e destra scambiando le coppie piazzate male
  - La posizione *p* sarà individuata automaticamente al termine

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 dei cicli si ha $i = j = 1$ e, dopo il controllo, $j = 0$

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;
```


Fusione

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

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

  | 2 | 3 | 4 | 5 | 9 | 11 | 13 | 15 | 22 | 24|
  - Numero di elementi dell'array risultante pari alla somma degli elementi dei due array dati
  - 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 i suoi elementi restanti nell'array finale
  - Individuabile dall'inizio confrontando gli ultimi elementi dei due array
    - 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
Algoritmo

- Mentre ci sono elementi da considerare nei due array
  - 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
Modularizzazione

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

Fusione
Procedura *merge*

- Definire gli array $a[1..m]$ e $b[1..n]$
- Se l'ultimo elemento di a è ≤ all'ultimo elemento di b allora
  - Fondi tutto a con b
  - Copia il resto di b
- Altrimenti
  - Fondi tutto b con a
  - Copia il resto di a
- Dà 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
  - Copia tutto a nei primi m elementi di c
  - Copia tutto b in c, partendo dalla posizione m+1
- Altrimenti
  - Fondi tutto a con b in c
  - Copia il resto di b in c partendo dalla posizione in cui è finita la fusione

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 $a \leq b$ allora
    - Copia il corrente a nella corrente posizione di c
    - Avanza l'indice di a
  - Altrimenti
    - Copia il corrente b nella corrente posizione di c
    - Avanza l'indice di b
  - Avanza l'indice di c

Fusione
Procedura *copy*

- Definire gli array $a[1..n]$ e $c[1..n+m]$ e definire dove bisogna cominciare in 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 (j)
  - Avanza l'indice di c (k)

Ordinamento Veloce di Hoare
(per partizionamento)

- Ordinamento per partizionamento-scambio
  - 2 versioni
    - Iterativa
    - Ricorsiva

Ordinamento Veloce

- Meglio un metodo di partizionamento

  | tutti gli elementi più piccoli | tutti gli elementi più grandi |

  – Aumenta il preordinamento
  – Riduce la dimensione del problema
    - Ogni partizione può essere affrontata separatamente

Ordinamento Veloce

- Il partizionamento richiede un valore in base al quale distinguere gli elementi
  – Scelto a caso
    - Potrebbe essere esterno ai valori della lista
  – Prenderne uno della lista stessa
    - Indifferente quale
      – Elemento *mediano*
        » Elemento in posizione media
      – Posizione *media*
        » Limite inferiore + limite superiore DIV 2

Ordinamento Veloce

- Situazione dopo il partizionamento:
  – Parte sinistra
    - Elementi minori dell'elemento mediano
  – Parte destra
    - Elementi maggiori dell'elemento mediano
- Escluse 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, 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
    - Una si elabora
    - 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
      - Dove iniziano
      - Dove finiscono
    - Salvare l'informazione sui limiti di ciascuna
      - Riferita alle posizioni (indice) 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
      - 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
    - 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 da elaborare per prima | x |
    - Ciascuna richiede 2 elementi
  - Troppo Spazio!

Ordinamento Veloce
Ottimizzazione

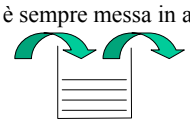
- Elaborare per prima la partizione più piccola
  - Minor numero di partizioni da tenere in sospeso
  - Partizioni di dimensione 1 non generano altre partizioni
    - Individuate e scartate subito dall'elaborazione
    - In ogni istante esiste una sola partizione in sospeso di cui vanno salvati i limiti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

Ordinamento Veloce
Ottimizzazione

- Ad ogni passo si elabora la più grande delle partizioni più piccole
  - Nel precedente caso peggiore: 2 elementi
  - Nuovo caso peggiore: $2 \log_2 n$
    - Partizioni divise sempre a metà
      - Dimensione dimezzata
  - Accettabile

Ordinamento Veloce

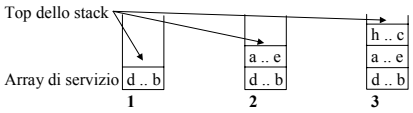
- Test per decidere qual è la partizione più grande
  - Se le partizioni si incontrano a sinistra del medio
    - Elabora la partizione sinistra
    - Salva i limiti di quella destra (che è la più grande)
  - Altrimenti
    - Elabora la partizione destra
    - Salva i limiti di quella sinistra (che è la più grande)
- 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 non ci saranno più limiti nella pila
  - Pila vuota
  - Terminazione del processo di partizionamento

Ordinamento Veloce
Algoritmo

Acquisire l'array *a*[1..*n*] da ordinare
Inizializzare il puntatore al top dello stack al valore 2
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 basata sull'applicazione dello stesso processo a problemi via via più piccoli
      - Divide l'insieme di dati in 2 partizioni in modo tale che *tutti* gli elementi della partizione sinistra (prime *pos* posizioni) siano ≤ a *tutti* gli elementi della parte destra (posizioni da *pos*+1 fino ad *n*)
      - Riapplica lo *stesso* meccanismo di *partizionamento* ad *entrambe* le *partizioni* (che sono di dimensioni via via più piccole) a[1..*pos*] e a[*pos*+1..*n*] fino ad ottenere segmenti di dimensione 1
        - » A questo punto l'array sarà completamente ordinato

Ordinamento Veloce

- 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
    - Ripeti il processo di partizionamento per la partizione sinistra
    - Ripeti il processo di partizionamento per la partizione destra

Ordinamento Veloce

- 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 unico elemento
      - Test: **se** inf < sup **allora** continua

Ordinamento Veloce
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

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

- Suddivisione successiva delle partizioni
  - Caso peggiore: suddivisione a metà
    - Profondità dell'albero: $\log_2 n$
      - Numero di volte che si può spaccare successivamente a metà una lista di n elementi
    - Ciascun livello dell'albero contiene n elementi
  - Complessità: $n \log_2 n$
    - Algoritmo ottimo!

Tecnica Hash

- Obiettivo
  - Ritrovare un elemento esaminando solo pochi elementi della tabella
  - Esempio:

a: |10|12|20|23|27|30|31|39|42|44|45|49|53|57|60|

a(1) a(8) a(15)

- Trovare il record con campo chiave uguale a 44 accedendo al massimo a 3 elementi prima di terminare con successo

Tecnica Hash

- Caso estremo
  - Trovare l'elemento cercato in un solo tentativo
    - Tutto ciò che si sa è il valore dell'elemento cercato
  - Inserire ciascun elemento in posizione coincidente col suo valore
    - Poco pratico
    - Esempio: Numeri telefonici di 7 cifre
      10000000 di elementi per memorizzarne solo 15!
  - Ottimizzazione dell'occupazione di memoria
    - Array di dimensione vicina al numero massimo di elementi che saranno memorizzati

Tecnica Hash
Funzioni di Accesso

- Necessità di ricondurre ciascun valore ad un indice valido dell'array
  - Normalizzazione dei dati
    - Esempio: 15 elementi, valore massimo nella lista 60
      - Applicare alla chiave la seguente trasformazione:
        Chiave diviso 4 & Arrotondamento
  - Potrebbe non essere noto il massimo valore degli elementi che si tratteranno
    - Impossibile stabilire a priori una funzione di normalizzazione
    - Necessità di un metodo più generale

Tecnica Hash
Funzioni di Accesso

- Soluzione migliore
  - Sufficiente una *trasformazione* delle chiavi che dia un indice valido dell'array
    - Esempio: array indicizzato da 1 a 15
      - Calcolare il modulo 15 della chiave
        - » Compreso fra 0 e 14
      - Aggiungere un'unità
        - » Valori compresi fra 1 e 15

Tecnica Hash
Funzioni di Accesso

- h: K $\rightarrow$ {0, 1, 2, …, *m*–1}
  - *K* insieme dei valori distinti che il campo chiave può assumere
  - *m* dimensione del vettore in cui si intende memorizzare gli elementi della tabella
- *K* sottoinsieme dei numeri naturali
  - Possibile funzione di accesso:

    h(k) = k MOD m
    - Sempre compreso fra 0 e *m* – 1

Tecnica Hash
Rappresentazione della Chiave

- *K* Non sottoinsieme dei numeri naturali
  - Insieme di stringhe alfanumeriche
    - La funzione hash è numerica
      - Per applicarla ad una chiave non numerica, bisogna associare alla chiave un valore numerico
- Necessità di definire funzioni hash generali
  - Associazione di un valore numerico ad una chiave di qualunque tipo
  - Applicazione della funzione hash numerica a tale valore

Tecnica Hash
Rappresentazione della Chiave

- Un possibile metodo di codificazione
  - Considerare la rappresentazione binaria della chiave *k*, *bin*(*k*)
    - Se la chiave non è numerica, è data dalla concatenazione della rappresentazione binaria di ciascun carattere che la compone
  - Calcolare il numero intero positivo corrispondente alla rappresentazione binaria della chiave, *int*(*bin*(*k*))
    - Esempio: $K = A^4$ con A = {'A','B',…,'Z'}

      B A R I

      ASCII code 1000010 1000001 1010010 1001001

      int(bin(k)) = ord('B') · 128^3 + ord('A') · 128^2 + ord('R') · 128 + ord('I') = 66 · 128^3 + 65 · 128^2 + 82 · 128 + 73 = …

Tecnica Hash
Rappresentazione della Chiave

- Un altro possibile metodo di codificazione
  - Lasciare invariate le altre cifre numeriche e trasformare le lettere A, B, …, Z nei numeri decimali 11, 12, …, 36
  - Sommare i numeri associati ai caratteri che compongono la chiave
    - Esempio:

      A.S.L. 'BA16' →

      $12 \cdot 36^3 + 11 \cdot 36^2 + 1 \cdot 36 + 6 = 574.170$

Tecnica Hash
Collisioni

- Associazione, da parte di una trasformazione, della stessa posizione a chiavi distinte
  - *Sinonimi*
    - Esempio: (chiave MOD 15) + 1
      - Posizione 1 ← 30, 45, 60
      - Posizione 9 ← 23, 53
      - Posizione 13 ← 12, 27, 42, 57
  - Ciascuna posizione dell'array può contenere al più un elemento
    - Ridurre al massimo le collisioni
    - Gestirle quando si verificano

Tecnica Hash
Collisioni

- Funzioni di *hashing perfetto* (che evitano i duplicati) abbastanza rare, anche per grandi tabelle
  - Esempio: *paradosso del compleanno*
    - Dato un gruppo di 23 persone, ci sono più del 50% di probabilità che due di esse siano nate nello stesso giorno dell'anno
    - In altre parole, se scegliamo una funzione aleatoria che trasforma 23 chiavi in un indirizzo di una tabella di 365 elementi, la probabilità che due chiavi NON collidano è solo 0.4927 (meno della metà)
- Individuare una funzione di accesso che porti ad un numero ridotto di collisioni è un problema complesso

Tecnica Hash
Collisioni

- Esempio
  - $41^{31} \approx 10^{50}$ possibili funzioni da un insieme K di 31 elementi in un insieme $\{1, 2, \ldots, 41\}$ di 41 posizioni
  - Solo $41 \cdot 40 \cdot 39 \cdot \ldots \cdot 11 = 41!/10! \approx 10^{43}$ ingettive
    - Daranno valori distinti per ciascun argomento
  - Solo una ogni 10 milioni eviterà le collisioni
    - Scelta dipendente dai particolari valori in K
- Occorre studiare
  - Buone funzioni di accesso
  - Valide strategie per gestire le collisioni

Tecnica Hash
Collisioni

- Scelta di m critica
  - Influenza fortemente il numero di possibili collisioni
- b base di rappresentazione della chiave
  - $m = b^n$ è una scelta insoddisfacente
    - Resto della divisione composto sempre dalle n cifre di più basso ordine della chiave
  - Meglio utilizzare come divisore un numero primo di valore vicino al numero di elementi che si desiderano riservare per il vettore

Tecnica Hash
Collisioni

- Esempi
  - Chiave rappresentata in base 10, $m = 1000$
    - Il resto sarà rappresentato dalle 3 cifre di più basso ordine della chiave
      - Tutte le chiavi che terminano per 893 daranno luogo a collisioni o sinonimie

        2893 MOD 1000 = 1893 MOD 1000 = 893
  - Chiave rappresentata in base 2, $m = 2^p$
    - Due numeri con gli stessi p bit finali darebbero sempre luogo ad una collisione

Tecnica Hash
Collisioni

- Numero di collisioni ridotto drasticamente se accettiamo uno spreco del 20% di memoria extra
  - Esempio: array di 19 elementi invece che di 15 (indicizzati da 0 a 18)

Posizione 0 ← 57	Posizione 8 ← 27
Posizione 1 ← 20, 39	Posizione 10 ← 10
Posizione 3 ← 60	Posizione 11 ← 30, 49
Posizione 4 ← 23, 42	Posizione 12 ← 12, 31
Posizione 6 ← 44	Posizione 15 ← 53
Posizione 7 ← 45	

- Collisioni non eliminate del tutto

Tecnica Hash
Gestione delle Collisioni

- Memorizzazione dei valori che collidono
  - Necessità di mantenere alta anche l'efficienza media nel ritrovamento
  - Tecniche
    - A scansione interna
      - Sinonimi memorizzati nella stessa tabella
    - A scansione esterna
      - Uso di aree di trabocco

Tecnica Hash
Tecnica a Scansione Interna

- Possono esistere molte locazioni "libere" nella tabella
  - Individuate da un valore speciale "vuoto"
  - Utilizzabili per memorizzare gli elementi che collidono
- Possibile criterio
  - Assegnare ad un elemento che collide con un altro la successiva posizione disponibile

Tecnica Hash
Metodo di Scansione Lineare

- (*Linear probing*)
  - Ricerca del prossimo elemento vuoto effettuata esaminando le posizioni contigue successive secondo la seguente funzione *lineare* in *i*:

    $pos_i \leftarrow (h(k) + i) \bmod m \qquad i \geq 1$

    - Posizione a cui si accede la *i*-esima volta che una posizione del vettore viene trovata occupata
  - Per $i = 0$: $pos_0 = h(k)$
    - Posizione iniziale da controllare

Tecnica Hash
Metodo di Scansione Lineare

- Generalizzazione

$$pos_i \leftarrow (h(k) + h \cdot i) \text{ MOD } m \qquad i \geq 1$$

  - *h* numero intero positivo primo con *m*
    - Garanzia che la scansione tocchi tutte le posizioni del vettore prima di riconsiderare posizioni già esaminate
  - Inconveniente: fenomeno dell'*agglomerazione*
    - Sequenza di indirizzi scanditi per la chiave *k* uguale a quella per le chiavi *k'* tali che

$$h(k') = h(k) + n \cdot h$$

Tecnica Hash

- Esempio
- Se la posizione 12 è già occupata dalla chiave 12, allora al 31 assegneremo la prima posizione libera in avanti, ovvero la 13

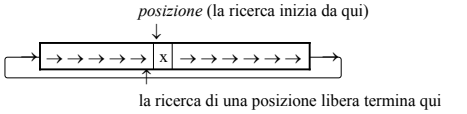
57 20 39 60 23 42 44 45 47 10 30 12 31 49 53

  - Numero medio di passi compiuti per posizionare tutti i 15 elementi: 21/15 ≈1.4
    - 11 elementi posizionati in un solo passo
    - 3 elementi in 2 passi
    - 1 solo elemento in 4 passo

Tecnica Hash
Memorizzazione degli Elementi

- Dalla chiave di ricerca deriva un valore hash usando la funzione modulo (dimensione tabella)
  - posizione ← chiave MOD m
- Se la posizione indicata dalla funzione hash è già occupata, compi una ricerca lineare in avanti a partire dalla posizione corrente
  - se tabella(posizione) ≠ chiave allora …
- Se si raggiunge la fine della tabella senza aver trovato una posizione libera la ricerca deve riprendere dall'inizio
  - Posizione ← (posizione + 1) MOD m

Tecnica Hash



- 2 possibili condizioni di terminazione della ricerca
  - Si incontra una cella "vuota"
  - Si arriva alla posizione di partenza della ricerca dopo aver controllato tutti gli elementi
- 2 controlli (uno per ogni condizione)

Tecnica Hash

- Possibilità di ridurre tale numero attraverso un artificio
  - Se la posizione indicata dalla funzione hash è già occupata, vi si pone il valore "cella vuota"
    - La condizione 2 (tabella piena) non si potrà mai verificare
    - La ricerca terminerà forzatamente per via della condizione 1
  - Alla fine si dovrà ripristinare la situazione preesistente

Tecnica Hash

- Bisognerà distinguere la vera causa della terminazione della ricerca
  - Perché effettivamente è stata trovata una cella vuota
  - Perché la tabella era piena
    - In tal caso si dovrà segnalare l'errore di "tabella piena"
- Uso di una variabile booleana
  - *errore*
    - Vera nel caso in cui la tabella sia piena e non abbia spazio per effettuare l'inserimento

Tecnica Hash

Algoritmo per l'Inserimento

calcola l'indice hash in base alla chiave da memorizzare
errore ← false
se l'indice associato alla chiave corrisponde ad un elemento "vuoto" **allora**
    poni l'elemento da memorizzare in tabella
altrimenti
    poni in posizione iniziale il valore "vuoto"
    ripeti
        calcola la posizione successiva (modulo dimensione tabella)
    finché non trovi un elemento "vuoto"
    se la posizione dell'elemento "vuoto" coincide con quella iniziale **allora**
        segnala l'errore (errore ← true)
    altrimenti
        poni in tabella l'elemento da memorizzare
    ripristina il valore modificato della tabella

Tecnica Hash

- Dopo aver memorizzato la tabella, possiamo effettuare la ricerca mediante funzioni di accesso
  - Il ritrovamento avverrà con lo stesso procedimento
    - Altrettanto valido
- Data la chiave dell'elemento da cercare, si genera un indirizzo mediante la funzione hash
  - Se a tale indirizzo si trova la chiave cercata allora la ricerca si può arrestare immediatamente
  - Altrimenti bisogna procedere per ricerca lineare come nella fase di memorizzazione

Tecnica Hash

- 3 possibili condizioni di terminazione della ricerca
  - Si trova la chiave cercata in tabella
    - Esito positivo
  - Si incontra una cella "vuota"
    - Esito negativo (elemento non trovato)
  - Si arriva alla posizione di partenza della ricerca dopo aver controllato tutti gli elementi
    - Esito negativo (elemento non trovato)
- 3 controlli (uno per ogni condizione)
  - Possibilità di ridurre tale numero attraverso un artificio analogo a quello per evitare il controllo tabella piena

Tecnica Hash

- Artificio per la riduzione del numero di controlli
  - Dopo aver confrontato la chiave cercata con la posizione indicata dalla funzione hash e aver scoperto che non sono uguali si pone nella posizione di partenza della ricerca proprio la chiave cercata
    - La condizione 3 di elemento non trovato non si potrà mai verificare
    - In caso di tabella piena ed elemento non trovato, la ricerca terminerà forzatamente per la condizione 1
  - Alla fine si dovrà ripristinare la situazione preesistente

Tecnica Hash

- Bisognerà distinguere la vera causa della terminazione della ricerca:
  - Perché effettivamente esisteva la chiave in tabella
  - Perché la chiave non esisteva e la tabella era piena
    - In tal caso si dovrà segnalare che l'elemento non è stato trovato
- Necessità di 2 variabili booleane
  - Attivo
    - Falsa quando si trova o la chiave o una posizione vuota
  - Trovato
    - Vera solo nel caso in cui si trovi la chiave

Tecnica Hash
Algoritmo di Ricerca

calcola l'indice hash come "chiave modulo dimensione tabella"
imposta *attivo* e *trovato* in modo da terminare la ricerca
se la chiave si trova nella posizione indicata dall'indice hash **allora**
    poni le condizioni per la terminazione
altrimenti
    poni la chiave nella posizione data dall'indice hash
mentre non è soddisfatta alcuna condizione di terminazione
    calcola l'indice successivo modulo dimensione
    se la chiave si trova nella posizione corrente **allora**
        Poni le condizioni di terminazione e valuta se *trovato* è vera
     altrimenti
        se la posizione è vuota segnala la terminazione
ripristina il valore modificato della tabella

Tecnica Hash
Programma Pascal

```
begin    attivo := true; trovato := false;
start := chiave MOD dimensione; posizione := start;
if tabella[start] = chiave then
  begin attivo := false; trovato := true; temp := tabella[start] end
else
  begin temp := tabella[start]; tabella[start] := chiave end;
while attivo do
  begin  posizione := posizione + 1 MOD dimensione;
  if tabella[posizione] = chiave then
    begin attivo := false; if posizione <> start then trovato := true end
  else if tabella[posizione] = vuoto then attivo := false
  end;
tabella[start] := temp
end.
```


Tecnica Hash
Considerazioni

- Prestazione valutabile in termini del numero di elementi della tabella che devono essere esaminati prima della terminazione della ricerca
  - Funzione del *fattore di carico* α
    - Percentuale di posizioni occupate in tabella
  - Si dimostra che, sotto opportune ipotesi statistiche, sono esaminati mediamente $\frac{1}{2}[1 + 1/(1 - \alpha)]$ elementi prima che la ricerca termini con successo
    - Per $\alpha = 0.8$ si hanno mediamente 3 accessi, qualunque sia la dimensione della tabella
    - Il costo per una ricerca infruttuosa è maggiore, mediamente $\frac{1}{2}[1 + 1/(1 - \alpha)^2]$

Tecnica Hash
Tecnica a Scansione Esterna

- Uso di *aree di trabocco*
  - Aree di memoria destinate alla memorizzazione degli elementi che, in inserimento, hanno portato ad una collisione
- Ricerca di un elemento di chiave *k*
  - Si calcola h(k)
  - Se si verifica una collisione allora si accede all'area di trabocco associata alla posizione h(k) e si comincia una nuova ricerca di una posizione libera in tale area

Tecnica Hash

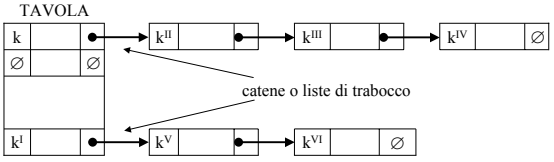
Aree di trabocco

- Una unica per tutta la tavola
  - Tabelle in cui la ricerca può essere effettuata con un'ulteriore funzione hash
- Una per ogni posizione occupata della tabella originaria
  - Si utilizza una rappresentazione con liste concatenate
    - Il problema di tabelle piene non si pone

Tecnica Hash

Gestione delle Collisioni

- Esempio di liste di trabocco



Tecnica Hash

Algoritmo di Inserimento

calcola l'indirizzo corrente applicando la funzione di accesso al campo chiave dell'elemento da memorizzare
se il campo chiave dell'elemento corrente di tabella è vuoto **allora**
    poni l'elemento da memorizzare in tabella
    crea una lista vuota di trabocco e associala all'elemento corrente
altrimenti (* aggiungere un nuovo elemento alla lista di trabocco *)
    inizializza il puntatore ULTIMO a inizio lista di trabocco associata all'elemento corrente
    mentre non è raggiunta la fine lista di trabocco **esegui**
        Aggiorna ultimo puntando al successivo elemento
    inserisci un nuovo elemento alla lista di trabocco

Ordinamento per Fusioni Successive

- Algoritmo di *MergeSort*
  - L'idea alla 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 (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

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 |

- Combinazione (fusione) dei sottoinsiemi ordinati

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

Ordinamento per Fusioni Successive

```
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, ultimo, q)
        end
end;
```

Ordinamento per Fusione

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

```
procedure merge(var a: nelements; primo, ultimo, mezzo: integer);
    var   i, j, k, h: integer;
          b: nelements;
```


Ordinamento per Fusioni Successive

```
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

```
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     {main}
    mergesort(a, b, 1, n)
end       {main}
```

Algoritmi su File

- Algoritmi basilari di creazione e modifica, ed inoltre
  - Fusione
  - Ordinamento
- Supponiamo di operare su file di record
  - Identificazione tramite campo chiave numerico intero
    - Struttura del record: | *chiave* | …altri dati… |
- Operazione dominante rispetto alla quale valutare la complessità
  - Lettura/scrittura di un elemento di un file

File

- Aggiunta di elementi ad un file esistente
  - Non vanno cancellati gli elementi esistenti
    - Dopo averli scorsi in lettura non si può scrivere
  - Necessità di un file di appoggio

Aprire in lettura il file originale ed in scrittura il file di appoggio
Finché non si è raggiunta la posizione di inserimento ed il file non è finito
    copiare un elemento dal file originale a quello di appoggio
Inserire l'elemento da aggiungere nel file di appoggio
Finché non si è raggiunta la fine del file
    copiare un elemento dal file originale a quello di appoggio
Riversare il file di appoggio nel file originale (copia)

File

- Modifica di un elemento
  - Bisogna individuarne la posizione
    - Dopo averlo scorso il file in lettura non vi si può scrivere
  - Necessità di un file di appoggio

Aprire in lettura il file originale ed in scrittura il file di appoggio
Finché non si è raggiunto l'elemento da modificare
    copiare un elemento dal file originale a quello di appoggio
Inserire l'elemento modificato nel file di appoggio
Finché non si è raggiunta la fine del file
    copiare un elemento dal file originale a quello di appoggio
Riversare il file di appoggio nel file originale (copia)

Algoritmi su File

- Dichiarazione dei tipi usati per il file

```
type    tipo_chiave = 0..maxint;
        qualche_tipo = record
                        (* …dati…*)
                        end;
         tipo_rec = record
                        chiave: tipo_chiave;
                        resto_record: qualche_tipo
                end;
        tipo_file = file of tipo_rec;
```


Algoritmi su File

- Indicheremo con < la relazione d'ordine definita sui record del file
  - Basata sul campo chiave
  - Implementabile come funzione booleana
    function minore(x, y: tipo_rec): **boolean**; **true** sse x < y
- Per generalità definiamo un tipo *tipo_chiave* per il campo chiave
  - Può essere non strutturato o una stringa o comunque un tipo sui cui valori cui poter usare l'operatore <
    - Nel caso più semplice coinciderà con l'elemento del file

Fusione di File

- Dati due file di record, entrambi ordinati in modo crescente sul campo chiave, produrre un unico file ottenuto dalla fusione dei primi due
  - Il risultato è ancora un file di record ordinato in modo crescente sul campo chiave
  - La fase di fusione vera e propria termina quando finisce uno dei file
    - Uso di una variabile booleana

```
procedure file_merge(var in1, in2, out: tipo_file);
   var   fine_in1_in2: boolean;
```

Fusione di File

- Dimensioni dei file non note a priori
  - Algoritmo ottimizzato per la fusione di array ordinati inadatto
  - Più appropriato il primo algoritmo presentato
    - Più semplice della procedura su array
  - Sostituzione delle condizioni
    - $i \leq m$ (fine primo array)
    - $j \leq n$ (fine secondo array)

  con dei test EOF per riconoscere la terminazione degli insiemi di dati

Fusione di File

```
begin
reset(in1); reset(in2); rewrite(out); fine_in1_in2 := eof(in1) or eof(in2);
while not fine_in1_in2 do
   begin
   if in1^.chiave < in2^.chiave then
        begin   out^ := in1^; get(in1); fine_in1_in2 := eof(in1)        end
   else
        begin   out^ := in2^; get(in2); fine_in1_in2 := eof(in2)        end;
   put(out);
   end;
while not eof(in2) do     begin   out^ := in2^; put(out); get(in2)     end;
while not eof(in1) do     begin   out^ := in1^; put(out); get(in1)     end
end;
```


Ordinamento NaturalMerge

- Ordinare un file di record cosicché i record risultino sistemati in base all'ordine ascendente del campo chiave
  - Possibilità di riusare le idee relative all'ordinamento di array attraverso fusioni successive
    - *Natural merging*

Ordinamento NaturalMerge

- Sequenza di record iniziale (non ordinata) posta in un file di nome *file_dati*
  - Ordinamento applicato ad una copia *C* del file originale
    - Possibilità di recuperare eventuali cadute di sistema durante l'ordinamento
- Dichiarative di variabili globali

```
var     file_dati: tipo_file;
        C: tipo_file
```


Ordinamento NaturalMerge

- Schema di programma che incorpora l'ordinamento per fusioni successive
  - Copiare il file da ordinare in *C*
  - Ordinare la copia
  - Alla fine ritrasferire il file ordinato nel file originale

```
begin
   copia_file(file_dati, C);
   NaturalMergeSort(C);
   copia_file(C, file_dati)
end.
```


Ordinamento NaturalMerge

Procedura *copia_file*

- Trasferisce un elemento alla volta da un file ad un altro
  - Molto semplice
- Si può fare in modo che elenchi in uscita i valori dei campi chiave man mano che li elabora
  - Controllo ulteriore sul comportamento

Ordinamento NaturalMerge
Procedura *copia_file*

```
procedure copia_file(var F, G: tipo_file);
begin
    reset(F); rewrite(G);
    while not eof(F) do
        begin
            writeln(F^.chiave);
            G^ := F^;
            put(G);
            get(F);
        end
end;
```


Ordinamento NaturalMerge
Algoritmo

- Ogni passo dell'ordinamento consiste in 2 fasi
  - Distribuzione
    - Pone alternativamente i record in due file *A* e *B*
      - File ausiliari dello stesso tipo di *C*
    - Si basa su una valutazione dei dati elementari così da ordinare parzialmente
      - Non si cambia file in cui si travasa fintantoché è mantenuto l'ordine ascendente
  - Fusione
    - Riaggrega *A* e *B* in *C*

Ordinamento NaturalMerge
Esempio

- *Run*
  - Sottosequenza del file costituita da elementi già ordinati secondo il criterio scelto

- Situazione iniziale (run separati da ||):

C || 82 || 48 || 14 | 15 | 84 || 25 | 77 || 13 | 72 || 4 | 51 || 19 | 27 | 43 | 57 ||53 ||

  - Prima distribuzione

A || 82 || 14 | 15 | 84 || 13 | 72 || 19 | 27 | 43 | 57 ||
B || 48 || 25 | 77 || 4 | 51 | 53 ||

  - Si fondono *A* e *B* e si ottiene un nuovo *C*

C || 48 | 82 || 14 | 15 | 25 | 77 | 84 || 4 | 13 | 51 | 53 | 72 || 19 | 27 | 43 | 57 ||

Ordinamento NaturalMerge
Esempio

- Eravamo arrivati a:

C || 48 | 82 || 14 | 15 | 25 | 77 | 84 || 4 | 13 | 51 | 53 | 72 || 19 | 27 | 43 | 57 ||

  - Si ridistribuisce

A || 48 | 82 || 4 | 13 | 51 | 53 | 72 ||

B || 14 | 15 | 25 | 77 | 84 || 19 | 27 | 43 | 57 ||

  - Si fondono nuovamente in

C || 14 | 15 | 25 | 48 | 77 | 82 | 84 || 4 | 13 | 19 | 27 | 43 | 51 | 53 | 57 | 72 ||

  - Si distribuisce per l'ultima volta

A ||14 | 15 | 25 | 48 | 77 | 82 | 84 ||

B || 4 | 13 | 19 | 27 | 43 | 51 | 53 | 57 | 72 ||

  - Infine (file ordinato)

C || 4 | 13 | 14 | 15 | 19 | 25 | 27 | 43 | 48 | 51 | 53 | 57 | 72 | 77 | 82 | 84 ||

Ordinamento NaturalMerge

- Supponendo di aver introdotto le precedenti definizioni globali di file, la procedura ad alto livello
  - Agisce sul parametro formale *C*
    - Esegue operazioni successive di fusione e distribuzione
      - Usa due file *A* e *B* dello stesso tipo di *C* come appoggio
    - Si ferma quando il file è ordinato (composto da un unico run)
      - Usa una variabile *numero_run* per controllarlo

Ordinamento NaturalMerge
Procedura ad Alto Livello

```
procedure NaturalMergeSort(var C: tipo_file);
    var   numero_run: integer;
          A, B: tipo_file;
          fine_run: boolean;
begin
    repeat
          reset(C); rewrite(A); rewrite(B);
          distribuisci;
          reset(A); reset(B); rewrite(C);
          numero_run := 0;
          fondi
    until numero_run = 1
end;
```

Ordinamento NaturalMerge

Procedura *distribuisci*

- Copia run da *C* in *A* e *B* alternativamente finché *C* non è completamente letto
  - *A*, *B* e *C* variabili non locali

```
procedure distribuisci;
begin
    repeat
        copia_un_run(C,A);
        if not eof(C) then copia_un_run(C,B)
    until eof(C)
end;
```


Ordinamento NaturalMerge

Procedura *copia_un_run*

- Copia un run di record dal file che figura come primo parametro a quello che appare come secondo parametro
- Chiamata ripetutamente fino alla fine di un run
  - *fine_run* variabile non locale
    - Indica che si è giunti alla fine di un run
    - Dichiarata nella procedura di alto livello per essere visibile a chi ne ha bisogno
    - Impostata (aggiornata) ad ogni chiamata della procedura *copia*

Ordinamento NaturalMerge

Procedura *copia_un_run*

```
procedure copia_un_run(var sorgente, destinazione: tipo_file);
begin
    repeat
        copia(sorgente, destinazione)
    until fine_run
end;
```

Ordinamento NaturalMerge

Procedura *copia*

- Trasferisce il valore di un singolo elemento dal file che appare come primo parametro al file che appare come secondo parametro
- Controlla l'eventuale fine di un run
  - Fine del file
  - Chiave dell'ultimo record copiato maggiore di quella del successore
    - Necessità di conservare l'ultimo record in una variabile *elemento_copiato* per il confronto con il suo successore

Ordinamento NaturalMerge

Procedura *copia*

```
procedure copia(var sorgente, destinazione: tipo_file);
    var   elemento_copiato: tipo_rec;
begin
    elemento_copiato := sorgente^;
    get(sorgente);
    destinazione^ := elemento_copiato;
    put(destinazione);
    if eof(sorgente) then
          fine_run := true
    else
          fine_run := (elemento_copiato.chiave > sorgente^.chiave)
end;
```


Ordinamento NaturalMerge

Procedura *fondi*

- Implementa la fase di fusione
  - Combina run da ciascuno di *A* e *B* fino alla fine di uno dei due
    - *A* e *B* variabili non locali
  - Conta il numero di run incontrate
    - *numero_run* variabile non locale
  - Copia il resto della coda dell'altro file in *C*
    - Non si entra nel ciclo relativo al file già finito

Ordinamento NaturalMerge
Procedura *fondi*

```
procedure fondi;
begin
while not (eof(A) or eof(B)) do      begin
    fondi_un_run_da_A_e_B; numero_run := numero_run + 1
    end;
while not (eof(A)) do      begin
    copia_un_run(A,C); numero_run := numero_run + 1
    end;
while not (eof(B)) do      begin
    copia_un_run(B,C); numero_run := numero_run + 1
    end;
end;
```


Ordinamento NaturalMerge
Procedura *fondi_un_run_da_A_e_B*

- Opera comparando i campi chiave nei corrispondenti run di *A* e *B*
  - *A* e *B* variabili non locali
- Chiama la procedura *copia* per trasferire i record selezionati a *C*
  - *C* variabile non locale
- Il processo termina quando uno dei run è esaurito
  - *fine_run* variabile non locale
- Questo provoca la ricopiatura in *C* della coda dell'altro run

Ordinamento NaturalMerge
Procedura *fondi_un_run_da_A_e_B*

```
procedure fondi_un_run_da_A_e_B;
begin
repeat
    if a^.chiave < b^.chiave then      begin
            copia(A,C);
            if fine_run then copia_un_run(B,C)
            end
    else   begin
            copia(B,C);
            if fine_run then copia_un_run(A,C)
            end
until fine_run
end;
```

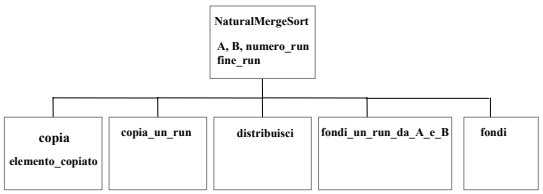
Ordinamento NaturalMerge
Chiamate di Procedure

- Procedura principale: *NaturalMergeSort*
  - Unica visibile all'esterno
- Procedure *distribuisci* e *fondi*
  - Chiamate solo da *NaturalMergeSort*
- Procedura *fondi_un_run_da_A_e_B*
  - Chiamata da *fondi*
- Procedura *copia_un_run*
  - Chiamata da *distribuisci*, *fondi* e *fondi_un_run_da_A_e_B*
- Procedura *copia*
  - Chiamata da *copia_un_run*, *fondi_un_run_da_A_e_B*

Ordinamento NaturalMerge
Uso di Variabili Non Locali

- *A*, *B*
  - Usati da tutte le procedure
- *fine_run*
  - *fondi_un_run_da_A_e_B*, *copia*, *copia_un_run*
- *numero_run*
  - *NaturalMergeSort*, *fondi*
- *elemento_copiato*
  - Usato solo da *copia*

Ordinamento NaturalMerge
Nidificazione



N.B.: conta l'ordine di dichiarazione (da sinistra verso destra)

Ricerca di una Parola Chiave

- Contare il numero di volte che una parola data appare in un testo
  - Non ci si curi del problema dell'immissione del testo
    - Si ipotizzi di digitare il testo in input dalla tastiera
  - Esempio

    "Questo è un segmento di testo"
    - Verificare quante volte appare la parola "un"

Ricerca di una Parola Chiave

- L'esame va fatto un carattere alla volta
- (Tranne la prima,) una parola in un testo può essere preceduta o seguita
  - Da un blank
  - Da caratteri speciali
- Due parole possono essere simili anche per una frazione di parola
  - La discordanza può apparire in qualunque momento, durante il processo di confronto
  - Esempio: un *un*a

Ricerca di una Parola Chiave

- Gestione del confronto
  - Tenere la parola da ricercare in un array
  - Portare avanti la lettura del testo carattere per carattere
    - Il processo di lettura del testo continua fino alla fine del testo
    - Contemporaneamente avviene il confronto con la parola ricercata
      - Incremento dell'indice dell'array

Ricerca di una Parola Chiave

- Dunque bisogna:
  - Leggere un carattere del testo
  - Confrontarlo con la prima lettera della parola cercata
    - Se vi è accordo si legge il secondo carattere e si confronta con la seconda lettera della parola cercata per verificare se l'accordo continua

Ricerca di una Parola Chiave

- Ogni volta che si stabilisce che la parola è completamente eguale allo spezzone confrontato durante la lettura del testo, va definito se lo spezzone è una parola o una substringa di una parola nel testo
  - Si assume che una parola sia preceduta e seguita da un carattere non alfabetico
    - Necessità di "ricordare" il carattere che precede una parola nel testo
    - Necessità di tener conto delle linee nel testo

Ricerca di una Parola Chiave

- Parte "centrale" dell'intera strategia

Metti a 1 l'indice *i* dell'array di parola
Mentre non è vera la condizione che segnala la fine del testo **esegui**
    Leggi il prossimo carattere
    Se il carattere è identico all'*i*-esimo carattere dell'array contenente la parola **allora**
        Estendi il confronto di 1
        Se vi è concordanza **allora**
            Verifica che l'intera parola concordi e prendi azioni opportune
    Altrimenti
        Prendi le azioni conseguenti un caso di discordanza

Ricerca di una Parola Chiave

- Necessità di ricordare il carattere che precede uno spezzone di testo
  - N.B.: Un carattere su cui si è verificata una condizione di disaccordo *può* essere un carattere che precede una parola
    - Bisogna salvarlo
    - Sarà cambiato solo quando vi è una condizione di disaccordo
      - Non varierà durante il confronto

Ricerca di una Parola Chiave

- Quando si rileva un disaccordo
  - Salvare il carattere corrente nel testo come *pre*
  - Reimpostare l'indice di parola a 1
- Se vi è accordo completo
  - Incrementare il contatore di confronto di 1
  - Lasciare in *pre* il più recente
  - Reimpostare l'indice di parola a 1

Ricerca di una Parola Chiave
Algoritmo

- Stabilire la parola e la sua lunghezza
- Inizializzare il contatore di concordanze *nmatches*, definire il carattere precedente e mettere a 1 l'indice di array della parola da cercare
- **Mentre** ci sono caratteri nel file **esegui**
  - **Mentre** la linea non è terminata **esegui**
    - Leggi il prossimo carattere *chr*
    - (vd. lucido successivo)
  - Passa all'altra linea di testo
- *nmatches* contiene il numero di concordanze

Ricerca di una Parola Chiave
Algoritmo

- **Se** *chr* coincide con l'*i*-esimo carattere nella parola **allora**
  - **Se** l'intera parola concorda **allora**
    - Leggi il prossimo *chr* nel testo *post*
    - **Se** il carattere precedente ed il seguente non sono alfabetici **allora**
      - Aggiorna *nmatches*
    - Reimposta l'indice di parola *i*
    - Salva il carattere *post* come prossimo precedente
- **Altrimenti**
  - Salva il prossimo carattere precedente
  - Reimposta *i* a 1

Ricerca di una Parola Chiave
Considerazioni

- L'istruzione fondamentale è il paragone tra il corrente carattere del testo ed il corrente carattere di parola
  - Viene fatto un numero di volte uguale al numero di caratteri nel testo
- Con ogni iterazione è letto un altro carattere
  - Si raggiunge sicuramente la fine del file
  - L'algoritmo termina

Ricerca di una Parola Chiave
Considerazioni

- Allo stadio di ricerca in cui sono stati letti i primi *j* caratteri sono state contate tutte le parole nel testo che corrispondono alla parola ricercata
- È possibile realizzare algoritmi che ottimizzino la ricerca
  - In funzione della lunghezza della stringa da ricercare, o
  - In funzione della particolare parola
    - Saltando alla parola successiva già al primo disaccordo