12 - Algoritmi

CAP.11: Algoritmi di Ordinamento

Introduzione all'Ordinamento

L'ordinamento è un problema classico dell'informatica che consiste nel disporre gli elementi di un insieme secondo una relazione d'ordine predefinita, come quella numerica o alfabetica. Il risultato è una permutazione degli elementi che soddisfa questa relazione. L'ordinamento può essere crescente o decrescente e può applicarsi a elementi semplici o complessi, come record identificati da una chiave. La chiave determina l'ordine e la posizione dell'elemento.

Tipologie di Algoritmi

Gli algoritmi di ordinamento si dividono in due categorie principali:

- **Algoritmi esterni**, che utilizzano strutture ausiliarie (es. array di appoggio).
- **Algoritmi interni**, che operano direttamente sull'array da ordinare.
  Non esiste un algoritmo universalmente migliore: le prestazioni dipendono dalla dimensione dei dati e dal grado di pre-ordinamento. Gli algoritmi basati su confronto e scambio sono comuni e includono metodi come **selezione**, **inserzione** e **scambio**.

Ordinamento per Selezione (Selection Sort)

L'algoritmo divide l'array in due parti: una ordinata e una non ordinata. Ad ogni iterazione si seleziona l'elemento minimo dalla parte non ordinata e lo si scambia con il primo elemento di quella sezione. Questo processo viene ripetuto fino a ordinare l'intero array.

Pseudocodice

1. Dividi l'array in parte ordinata e non ordinata.
2. Trova il minimo nella parte non ordinata.
3. Scambia il minimo con il primo elemento della parte non ordinata.
4. Ripeti finché tutti gli elementi non sono ordinati.

Codice in C

```
for (i = 0; i < n - 1; i++) {
    min = a[i];
    p = i;
    for (j = i + 1; j < n; j++) {
        if (a[j] < min) {
            min = a[j];
```

```
            p = j;
        }
    }
    a[p] = a[i];
    a[i] = min;
}
```

Analisi

Il tempo di esecuzione è quadratico, $O(n2)O(n^2)$, indipendentemente dal pre-ordinamento, poiché ogni iterazione richiede n−in-i confronti. L'algoritmo effettua sempre lo stesso numero di confronti, ma può evitare scambi inutili se l'indice dell'elemento minimo coincide con quello corrente.

Ordinamento per Inserzione (Insertion Sort)

Questo algoritmo inserisce ogni elemento nella posizione corretta all'interno della porzione già ordinata. È simile al modo in cui si ordinano le carte in mano.

Metodo

1. Considera il primo elemento come già ordinato.
2. Prendi un elemento dalla parte non ordinata e inseriscilo nella posizione corretta nella parte ordinata.
3. Ripeti fino a ordinare l'intero array.

Codice in C

```
for (i = 1; i < n; i++) {
    x = a[i];
    j = i - 1;
    while (j >= 0 && a[j] > x) {
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = x;
}
```

Analisi

L'insertion sort è efficiente per piccoli dataset o array quasi ordinati. Nel migliore dei casi (array già ordinato), la complessità è $O(n)O(n)$, mentre nel peggiore è $O(n2)O(n^2)$ per array ordinati al contrario.

Ordinamento a Bolle (Bubble Sort

Il bubble sort confronta elementi adiacenti e li scambia se non sono nell'ordine corretto. Gli elementi più grandi "affondano" verso la fine, mentre quelli più piccoli "risalgono" verso l'inizio, come bolle.

Metodo

1. Confronta ogni coppia di elementi adiacenti.
2. Scambia se necessario.
3. Ripeti finché l'intero array è ordinato.

Codice in C

```
for (i = 0; i < n - 1; i++) {
    for (j = n - 1; j > i; j--) {
        if (a[j] < a[j - 1]) {
            t = a[j];
            a[j] = a[j - 1];
            a[j - 1] = t;
        }
    }
}
```

Miglioramenti

Un indicatore può interrompere l'algoritmo se una passata non comporta scambi, segnalando che l'array è già ordinato. La complessità varia da $O(n)O(n)$ (array ordinato) a $O(n2)O(n^2)$ (array decrescente).

Ordinamento per Fusione (Merge Sort)

Il merge sort divide l'array in due metà, le ordina ricorsivamente e le unisce in modo ordinato. Si basa sul principio "divide et impera".

Metodo

1. Dividi l'array in due metà.
2. Ordina ricorsivamente ciascuna metà.
3. Unisci le due metà ordinate.

Codice in C

```
void merge(int a[], int primo, int mid, int ultimo) {
    int i = primo, j = mid + 1, k = 0;
    int b[ultimo - primo + 1];
    while (i <= mid && j <= ultimo) {
        if (a[i] < a[j]) b[k++] = a[i++];
```

```
            else b[k++] = a[j++];
        }
        while (i <= mid) b[k++] = a[i++];
        while (j <= ultimo) b[k++] = a[j++];
        for (i = primo, k = 0; i <= ultimo; i++, k++) a[i] = b[k];
    }

    void mergeSort(int a[], int primo, int ultimo) {
        if (primo < ultimo) {
            int mid = (primo + ultimo) / 2;
            mergeSort(a, primo, mid);
            mergeSort(a, mid + 1, ultimo);
            merge(a, primo, mid, ultimo);
        }
    }
```

Analisi

La complessità è $O(nlogn)O(n \log n)$, poiché ogni divisione riduce l'array di metà e ogni fusione richiede $O(n)O(n)$. Il merge sort è indipendente dal pre-ordinamento, ma richiede memoria aggiuntiva per le fusioni.

CAP. 12: Algoritmi di Ricerca

Introduzione alla Ricerca

La ricerca è un problema fondamentale dell'informatica, che consiste nel determinare se un elemento specifico (chiamato chiave) è presente in un insieme di dati e, in caso affermativo, restituirne la posizione. Gli input sono l'insieme di dati e la chiave, mentre l'output è la posizione dell'elemento trovato o un valore indicativo che segnala la sua assenza (es. 0 o -1). Per esempio, una funzione di ricerca può essere rappresentata come `int ricerca(int valore, int vettore[], int n)`.

Ricerca Lineare Esaustiva

La ricerca lineare esaustiva è il metodo più semplice e non richiede che i dati siano ordinati. Consiste nello scorrere tutti gli elementi dell'insieme uno per uno, confrontandoli con la chiave. Se l'elemento viene trovato, la sua posizione viene memorizzata e restituita. In caso di più occorrenze, restituisce l'ultima posizione.

Algoritmo

1. Inizia dal primo elemento e scansiona fino alla fine dell'array.
2. Se un elemento corrisponde alla chiave, aggiorna la posizione.
3. Restituisci la posizione al termine della scansione.

Codice in C

```c
int ricerca(int a[], int n, int x) {
    int posizione = 0;
    for (int j = 0; j < n; j++) {
        if (a[j] == x) {
            posizione = j;
        }
    }
    return posizione;
}
```

Ricerca Lineare con Sentinella

Questa variante della ricerca lineare si interrompe al primo ritrovamento dell'elemento cercato. È utile quando si vuole solo sapere se un elemento esiste o se è unico.

Algoritmo

1. Scansiona l'array dall'inizio.
2. Se trovi la chiave, restituisci immediatamente la posizione.
3. Se la scansione termina senza trovare l'elemento, restituisci un valore indicativo (es. -1).

Codice in C

```c
int ricerca(int a[], int n, int x) {
    int posizione = -1;
    for (int j = 0; j < n && posizione < 0; j++) {
        if (a[j] == x) {
            posizione = j;
        }
    }
    return posizione;
}
```

Ricerca Binaria (o Dicotomica)

La ricerca binaria è un algoritmo molto più efficiente, ma applicabile solo a insiemi di dati ordinati. Il principio base è confrontare l'elemento cercato con il valore centrale dell'insieme, escludendo metà dell'insieme in base al risultato del confronto.

Metodo

1. Confronta la chiave con l'elemento al centro dell'insieme.

2. Se la chiave è uguale al valore centrale, l'elemento è trovato.
3. Se è minore, analizza la metà sinistra; se è maggiore, analizza la metà destra.
4. Ripeti fino a trovare l'elemento o esaurire l'insieme.

Esempio

Considerando il vettore ordinato `[2, 4, 7, 11, 24, 25, 29, 32, 38, 44, 53, 61]` e cercando `29`:

- Primo confronto: elemento centrale `25` (indice 5). `29 > 25`, analizza la metà destra.
- Secondo confronto: elemento centrale `32` (indice 8). `29 < 32`, analizza la metà sinistra.
- Terzo confronto: elemento centrale `29` (indice 6). Elemento trovato.

Codice in C

```c
int ricerca_binaria(int a[], int n, int x) {
    int first = 0, last = n - 1, posizione = -1;
    while (first <= last && posizione == -1) {
        int mid = (first + last) / 2;
        if (a[mid] == x) {
            posizione = mid;
        } else if (x > a[mid]) {
            first = mid + 1;
        } else {
            last = mid - 1;
        }
    }
    return posizione;
}
```

Analisi

La ricerca binaria richiede un numero di operazioni proporzionale a $log2(n)\log_2(n)$, rendendola estremamente efficiente per grandi dataset. Tuttavia, richiede che i dati siano già ordinati, il che può comportare un overhead aggiuntivo.

CAP.13: Algoritmo di Merge

Introduzione alla Fusione

L'algoritmo di fusione (merge) è un metodo utilizzato per combinare due array già ordinati in un unico array anch'esso ordinato. Questo processo assicura che l'ordine dei dati sia mantenuto. Ad esempio, dato l'array `a = [2, 5, 9, 13, 24]` e l'array `b = [3, 4, 11, 15, 22]`, il risultato della fusione sarà `c = [2, 3, 4, 5, 9, 11, 13, 15, 22, 24]`. La

dimensione del nuovo array `c` è pari alla somma degli elementi dei due array di partenza, cioè m+nm + n.

Procedura di Fusione

L'algoritmo considera ciascun array come diviso in una parte già fusa e una ancora da fondere. Durante il processo, viene esaminato il primo elemento della porzione non fusa di entrambi gli array e quello minore viene aggiunto al nuovo array. L'indice del valore appena inserito viene incrementato, e il processo si ripete finché tutti gli elementi non sono stati aggiunti all'array risultato.

Esempio

- Array iniziali: `a = [15, 18, 42, 51]`, `b = [8, 11, 16, 17, 44, 58, 71, 74]`.
- Fusione: Si inizia confrontando `a[0]` e `b[0]`, e si inserisce il minore, in questo caso `8`, nel nuovo array `c`.
- Risultato parziale dopo vari step: `c = [8, 11, 15, 16, 17, 18, 42, 44, 51]`.
- Quando uno dei due array si esaurisce (ad esempio, `a`), gli elementi rimanenti dell'altro array vengono copiati direttamente in `c`.

Caso Particolare

Se l'ultimo elemento di un array è minore del primo elemento dell'altro, la fusione si riduce a una semplice concatenazione. Per esempio, se `a = [2, 3, 4]` e `b = [5, 6, 7]`, il risultato sarà semplicemente `c = [2, 3, 4, 5, 6, 7]`.

Algoritmo Ingenuo

L'algoritmo base per la fusione funziona così:

1. Finché ci sono elementi in entrambi gli array:
   - Confronta i primi elementi rimanenti di entrambi.
   - Inserisci il più piccolo nell'array finale.
   - Incrementa l'indice dell'array da cui è stato preso l'elemento.
2. Quando uno dei due array si esaurisce:
   - Copia i restanti elementi dell'altro array nell'array finale.

Implementazione in C

```c
void merge(int a[], int b[], int c[], int m, int n) {
    int i = 0, j = 0, k = 0;

    // Finché ci sono elementi in entrambi gli array
    while (i < m && j < n) {
        if (a[i] < b[j]) {
            c[k++] = a[i++];
```

```
        } else {
            c[k++] = b[j++];
        }
    }

    // Copia gli elementi rimanenti di a (se ce ne sono)
    while (i < m) {
        c[k++] = a[i++];
    }

    // Copia gli elementi rimanenti di b (se ce ne sono)
    while (j < n) {
        c[k++] = b[j++];
    }
}
```

Limitazioni dell’Algoritmo

L'algoritmo di fusione presentato è efficiente quando la cardinalità (cioè la dimensione) degli array di input è nota. Tuttavia, per applicazioni come file sequenziali o dataset di dimensioni non predefinite, sono necessarie modifiche per gestire correttamente i dati.

L'algoritmo di merge è un componente fondamentale di tecniche di ordinamento più avanzate, come il Merge Sort. La sua semplicità lo rende una scelta ideale per combinare due insiemi ordinati in modo efficiente. Tuttavia, richiede che gli array di partenza siano già ordinati, altrimenti sarà necessario un passo di pre-ordinamento. La complessità temporale dell'algoritmo è $O(m+n)O(m+n)$, dove mm e nn rappresentano le dimensioni dei due array di input.