Ingegneria degli Algoritmi

Nicolò Bianchi e David Julian Belfiori

Indice

1 Introduzione

1.1 Che cosa è un algoritmo

Un algoritmo è una procedura per risolvere un problema ,ovvero produrre una risposta a partire dai dati che abbiamo ricevuto in input. Un algoritmo deve rispettare determinate caratteristiche:

- Definitezza (nessuna ambiguità)
- Input
- Output
- Finitezza
- Efficacia (ogni passo deve poter esser eseguito in un tempo finito)

riprendendo l'ultimo punto, per definizione di algoritmo, esso deve rispondere (produrre un output) in un tempo finito, altrimenti stiamo parlando di un metodo compuazionale.

N.B per ogni problema non esiste un solo algoritmo che lo risolve, non appena si dimostra che l'insieme degli algoritmi che risolvono il problema non è vuoto, si apre la discussione quale sia il 'migliore'

1.2 Come si valuta un algoritmo

Si può valutare la qualità di un algoritmo descrivendone:

- **Complessità temporale** : il tempo **T(n)** impiegato per risolvere un problema di dimensione n
- **Complessità spaziale**: la quantità di memoria impiegata **S(n)** per risolvere un problema di dimensione n

2 Correttezza degli algoritmi

Per qualunque algoritmo di deve dimostrare la terminazione (in un tempo finito) e la *correttezza*. In informatica per verificare la correttezza di un algoritmo spesso si dimostra per induzione utilizzando degli invarianti [1].

Es. Procedura Minsearch

Procedura Minsearch(A, n)

Input: Array contenente i valori $A[i]$, $i = 1, \ldots, n$;
$min \leftarrow A[1]$;
for $i \leftarrow 2$ **to** n **do**
  if $A[i] < min$ **then**
    $min \leftarrow A[i]$;
Result: min

Dimostrazione di corretezza

1. prima dell'inizio consideriamo solo il vettore A[1:1](che contiene solo un elemento), e la var. min è inizializzata bene.
2. All'iterazione i diamo per vero che *min* contenga il minimo del vettore A[1:i-1], allora considerando il valore A[i] ci sono due possibilità, che $A[i] < min$ oppure no, in ogni caso *min* contiene il minimo valore del vettore A[1:i], ossia la proprieta invariante che volevamo.

[1] Le invarianti di ciclo sono un utile strumento che permette di provare la correttezza e la terminazione di un algoritmo che esegue al suo interno cicli.

3 Complessità computazionale

Per risolvere un problema, spesso sono disponibili molti algoritmi diversi. Un criterio per scegliere il migliore consiste nel valutare la 'bontà' di un algoritmo in base alla quantità di risorse utilizzate (tempo e spazio) per il calcolo.

3.1 Quanto costa risolvere un problema?

In pratica, non è possibile misurare il tempo di calcolo di un algoritmo con il numero di secondi richiesti per eseguire una determinata procedura. Infatti tale tempo è fortemente dipendente dai dati in ingresso, dal linguaggio in cui la procedura è descritta e dalla natura e la velocità del calcolatore elettronico.
Poichè i problemi hanno una dimensione che dipende dalla grandezza dei dati in ingresso, il tempo di calcolo si può esprimere come il costo complessivo delle operazioni elementari (aritmetiche, logiche, di confronto e di assegnazione) in funzione della dimensione n dei dati di ingresso

E' se la dimensione n non è sufficiente a determinare completamente il tempo di esecuzione?

Il costo delle operazioni è principalmente valutato in 3 casi:

1. **Worst case:** $T(n)$ è $O(f(n))$ **per qualunque** input possibile di dimensione n
2. **Average case:** $T(n)$ è $\Theta(f(n))$ **in media** su tutti gli input possibili di dimensione n
3. **Best case:** $T(n)$ è $\Omega(f(n))$ questo valore viene raggiunto **per alcuni** degli input di dimensione n

Casi comuni:

1. $O(1)$: algoritmo che richiede un tempo costante
2. $O(log(n))$: algoritmo di ordine logaritmico
3. $O(n)$: algoritmo di ordine lineare
4. $O(n^k)$: algoritmo di ordine polinomiale
5. $O(a^n)$: algoritmo di ordine esponenziale

Gli algoritmi di **complessità polinomiale** sono **considerati inefficienti** e non sono accettabili. Una volta realizzato un algoritmo con complessità polinomiale, si cerca di migliorarne le prestazioni progettando algoritmi con complessità sempre inferiore

Esempio: un semplice algoritmo polinomiale è il *Selection Sort*, che è basato sulla proprietà che in una sequenza ordinata il primo elemento ha valore minimo

3.2 Problemi vs Algoritmi

Inanzi tutto distinguiamo la **complessità di un algoritmo** e la **complessità di un problema**

- **Complessità di un algoritmo** è la complessità di un particolare metodo per la soluzione di un problema
- **Complessità di un problema** è la complessità del **migliore** algoritmo che risolve quel problema

Ad esempio, l'ordinamento per inserzione è un algoritmo di ordine $O(n^2)$ mentre il problema dell'ordinamento per confronti ammette una soluzione in tempo $O(nlog(n))$

Esempio: Insertion Sorting (ordinamento per inserzione) Data una procedura, si valuta il

```
Procedura InsertionSorting(A, n)
Input: Array contenente le chiavi A[i], i = 1, ..., n;
for i ← 2 to n do
    temp ← A[i];
    j ← i;
    while j > 1 and A[j − 1] > temp do
        A[j] ← A[j − 1];
        j ← j − 1 ;
    A[j] ← temp;
```

costo del suddetto algoritmo attraverso delle "linee guida":

- le espressioni scalari [2] elementari hanno costo O(1)
- il costo di una sequenza di istruzioni è dato dalla somma dei costi delle singoli istruzioni
- il costo di un ciclo è la somma del costo delle singole iterazioni sull'insieme di tutte le iterazioni
- un istruzione condizionale (if) ha un costo nel **caso peggiore** che è il massimo tra i costi del ramo *if* e del ramo *else*; per il costo medio occorre fare la media pesata (con la stima delle probabilità che esca un caso o un altro

3.3 Costo di un ciclo

Per un ciclo bisogna calcolare:

$$\sum_{i \in I} C(i) \tag{1}$$

Dove:

- i è un iterazione
- I è l'insieme di tutte le iterazioni
- *C(i)* è il costo della i-esima iterazione

Spesso il costo di un ciclo è costante, trovare l'insieme di tutte le iterazioni I è facile nei ciclo *for* , ma non necessariamente nei cicli *while*
N.B A cicli innestati corrispondono somme multiple

```
for i ← 1 to m do
    for j ← 1 to n do
        <statement>;
```

$$Opcnt = \sum_{i=1}^{m} \sum_{j=1}^{n} C(\text{statement}_{ij}).$$

[2] Qualsiasi entità numerica a cui corrisponda unicamente un valore e non una direzione

3.4 Complessità computazionale InsertionSorting

```
Procedura InsertionSorting(A, n)
Input: Array contenente le chiavi A[i], i = 1, ..., n;
for i ← 2 to n do
    temp ← A[i];
    j ← i;
    while j > 1 and A[j − 1] > temp do
        A[j] ← A[j − 1];
        j ← j − 1 ;
    A[j] ← temp;
```

Per il ciclo interno *while*:

- Nel caso migliore, esegue il controllo ed esce immediatamente (nel caso migliore sarà lineare)
- nel caso peggiore viene eseguito i-1 volte

Abbiamo quindi una complessità $\Omega(n)$ o $O(n^2)$

3.5 Complessità computazionale BinarySearch

```
Procedura BinarySearch(A, v, i, j)
Input: Array ordinato A[i], i = 1, ..., n, chiave v, estremi del sottovettore corrente i, j;
if i > j then
    Result: 0
else
    m ← ⌊(i + j)/2⌋;
    if A[m] = v then
        Result: m
    else if A[m] < v then
        Result: BinarySearch(A, v, m + 1, j)
    else
        Result: BinarySearch(A, v, i, m − 1)
```

Al primo passo la dimensione del vettore è n , ad ogni passo la dimensione si dimezza , nel caso peggiore è quindi necessario eseguire un numero di passi k tale che

$$n/2^k \leq 1 \rightarrow k \geq \log_2(n) \tag{2}$$

La procedura di **BinarySearch** è quindi è di **ordine** $O(\log(n)$

3.6 Funzione Ricorsiva

In una funziona ricorsiva ciascuna istanza applicata ad un problema è di dimensione n=1 (di cui possiamo calcolare il costo separatamente), oppure suddividere il suddetto problema di dimensione n in sotto problemi di dimensione più piccola, le cui soluzioni saranno poi combinate per costituire la soluzione complessiva
nell'analisi delle funzioni ricorsive si usano spesso delle relazioni di ricorrenza ,ossia delle equazioni del tipo

$$T(n) = G(T(n-n_1), T(n-n_2), T(n-n_3), ...T(n-n_k) \tag{3}$$

Ci sono due tipi di ricorrenze :

- Lineari a termini costanti.
  - Una ricorrenza a termini lineari sarà del tipo

$$T(n) = (\sum_{i=1}^{k} a_i T(n-i)) + cn^{\beta} \quad n > k \tag{4}$$

- A partizione bilanciata : dove si divide il problema principale di dimensione n in un insieme di sotto problemi di dimensione n/b, le cui soluzioni vengono ricombinate per costruire la soluzione complessiva

$$T(n) = \begin{cases} 1 & \text{se n=1} \\ aT(\frac{n}{b}) + n^{\beta} & se \;\; n > 1 \end{cases} \tag{5}$$

Il termine n^{β} misura il costo per suddividere il problema in sotto problemi e ricomporre le loro soluzioni

4 Nozioni base di C

4.1 Variabili

Ad ogni variabile in C deve essere associato un tipo, quindi quando si dichiara una variabile bisogna specificare sempre di che tipo è. In C le variabili possono essere di 5 tipi base:

- **Int**: numeri interi a 16 bit
- **Float**: sono numeri a virgola mobile 32 bit
- **Char**: sono le variabili che contengono un carattere 8 bit
- **Double** Numeri a virgola mobile a precisone doppia 64 bit
- **Void** *Speciale* : le variabili void sono usate per dichiarare dei puntatori [3] e per definire delle funzioni che non ritornano alcun valore

4.1.1 Regole di visibilità

Ogni variabile ha un compito e una conseguente visibilità

- Variabili globali
- Variabili locali
- Variabili statiche

4.1.2 Modificatori

I modificatori sono parole chiave che appunto modificano le proprietà predefinite delle variabili int e char e sono di 4 tipi:

- **Signed** : Lo specificatore signed indica che una data variabile va trattata con segno positivo o negativo, nei calcoli aritmetici. Può essere impiegato sia come modificatore di alcuni tipi base, che direttamente come tipo di dati, applicandosi in modo predefinito al tipo int
- **Unsigned** : Lo specificatore unsigned indica che una data variabile va trattata sempre con segno positivo, nei calcoli aritmetici.
- **Short** : dal nome si evince che riduce lo spazio assegnato alla variabile.
  Limita l'utente a memorizzare piccoli valori interi da -32768 a 32767. Può essere utilizzato solo su tipo di dati int.
- **Long**: al contrario di Short aggiunge lo spazio assegnato alla variabile .
  Consente all'utente di memorizzare numeri molto grandi da -9223372036854775808 a 9223372036854775807

[3] Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile (il puntatore x punta alla variabile y).

4.2 Operatori

Sulle variabili agiscono degli operatori per costiture dele espressioni:

- **Operatore di assegnazione**: =
- **Operatori aritmetici**: gli usuali
- **Operatori relazionali** : < > == !=
- **Operatori di incremento**: ++ –
- **Operatori logici**:| || & &&

4.3 Strutture di controllo: Switch

Il blocco switch-case permette di selezionare un'istruzione da eseguire in base ad una variabile. L'istruzione break interrompe l'elaborazione del blocco switch, in modo da uscire subito dal blocco se una condizione è vera. L'istruzione break è opzionale, ma se non viene usata, vengono eseguite le istruzioni dei casi seguenti.

```
switch (variabile)
{
 case valore1:
  // istruzione1
  break;

 case valore2:
  // istruzione2
  break;

 case valoreN:
  // istruzioneN
  break;
}
```

4.3.1 Istruzione switch-case-default

```
int main(void)
{
 int scelta;

 printf("Inserisci Una Scelta: ");
 scanf("%d", &scelta);

 switch(scelta){
        case  1: printf("Scelta 1\n");     break;
        case  2: printf("Scelta 2\n");     break;
        case  3: printf("Scelta 3\n");     break;
        default: printf("Altra Scelta\n"); break;
 }
}
```

4.4 Tipi di dati derivati: Array e puntatori

In linguaggio C, i tipi di dati derivati sono tipi di dati che vengono creati a partire dai tipi di dati primitivi (come int, float, char, ecc.) 4.1 attraverso varie operazioni. Questi tipi di dati derivati consentono di adattare e organizzare i dati in modi specifici per soddisfare le esigenze di programmazione

4.4.1 Array

un array è in tipo di dato aggregato costituito a molteplici dati elementari i quali

- Hanno tutti lo stesso tipo base
- sono identificati da un indice numerico (o da un inseme di indici)

il numero di indici è necessario per identificare un elemento e per comprendere la dimensione complessiva dell'array: monodimensionale , bidimensionale , etc.

```
#inlude <stdio.h>
#define N 100
int main(int argc, char *argv[]){
    int a[N],i;

    for (i=0,i<N,i++){
        a[i]=i
    }
}
```

```
#include <stdio.h>

int main() {
    // Dichiarazione di un array bidimensionale 2x2
    int arrayBidimensionale[2][2];

    // Accesso agli elementi dell'array
    arrayBidimensionale[0][0] = 1;
    arrayBidimensionale[0][1] = 2;
    arrayBidimensionale[1][0] = 3;
    arrayBidimensionale[1][1] = 4;

    // Stampa dell'intero array
    printf("Array bidimensionale:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", arrayBidimensionale[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

- il primo elemento corrisponde a [0][0] e l'ultimo a [M-1][N-1]

4.5 Puntatori

Un puntatore è una variabile speciale che contiene l'indirizzo di memoria di un'altra variabile. In altre parole, anziché contenere un valore direttamente, un puntatore contiene l'informazione su dove trovare il valore in memoria. I puntatori sono un concetto fondamentale in C e sono utilizzati per lavorare con dati complessi e strutture di dati dinamiche.
Esempio: come dichiarare un puntatore

```
int x = 10; // Dichiarazione di una variabile intera
int *ptr;   // Dichiarazione di un puntatore a un intero

ptr = &x;    // Assegna all'indirizzo del puntatore l'indirizzo di x

printf("Il valore di x: %d\n", x);
printf("Il valore a cui punta il puntatore: %d\n", *ptr);
```

L'operatore **&** consente di ottenere l'indirizzo di memoria di un altra variabile, e quindi assegnarlo ad un puntatore.

Perché i puntatori hanno generalmente un tipo associato ? Specificare il tipo di dato a cui si riferisce il puntatore è un modo per garantire che il compilatore sappia come interpretare i dati memorizzati all'indirizzo di memoria puntato dal puntatore. Ad esempio, se si dichiara un puntatore a un intero (int), il compilatore sa che dovrebbe trattare i dati a quell'indirizzo come valori interi.

4.5.1 Operazioni sui puntatori

Cosa **C** ci permette di fare con un puntatore:

- Dichiarare un vettore: `type *ptr;`
- Accedere al valore puntato: `val= *ptr;`
- Accedere ad un indirizzo : `type *ptr= &val;`

```
//Esempio: algoritmo per il calcolo del massimo comune divisore  utilizzano i
  puntatori
#include <stdio.h>
#include <stdlib.h>

// utlizzio di puntatori per scambiare due variabili e le dichiaro unsigned
  perche non possono essere negative
void swap(unsigned int *a,unsigned int *b){
  unsigned temp;
  temp=*a;
  //temp prende il valore di a che a sua volta prende il valore di m e quindi
  temp //prende il valore di m proveniente dalla funzione MCD
  *a=*b;
  *b=temp;
}

//le dichiaro unsigned perche non possono essere negative
unsigned int MCD(unsigned m,unsigned n){
  if (m==0 || n==0){
    printf("errore");
    exit(1);
  }
  while (m!=n){
    if (m<n){
      swap(&m,&n);
    }
    m=m-n;
  }
  return m;
}

int main(int argc, char **argv){
  unsigned int k = MCD(12, 18);
  printf("%d",k);}
```

4.5.2 La funzione malloc

La funzione malloc è utilizzata per allocare dinamicamente memoria nell'heap, consentendo al programma di gestire la memoria in modo flessibile. L'allocazione dinamica della memoria è utile quando non si conosce a priori la dimensione esatta della memoria necessaria o quando si desidera utilizzare la memoria in modo dinamico durante l'esecuzione del programma.

La funzione malloc (che sta per "memory allocation") **prende** un unico argomento, che rappresenta **il numero di byte di memoria da allocare**, e **restituisce un puntatore a un'area di memoria allocata**. Ad esempio:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *p;
  p = (int *)malloc(sizeof(int)); // Alloca memoria per un intero (4 byte su
  molti sistemi)
  if (p == NULL) {
    printf("Errore di allocazione di memoria.\n");
    return 1;
  }

  // Ora puoi usare la variabile puntatore p per accedere alla memoria allocata
  dinamicamente

  *p = 42;
  printf("Il valore di p : %d\n", *p);

  // Quando hai finito di usare la memoria,e'importante deallocarla per evitare
  memory leak
  free(p);

  return 0;
}
```

4.6 Strutture

In linguaggio C, le strutture (o "struct") sono utilizzate per creare tipi di dati personalizzati che possono contenere uno o più membri di diversi tipi. Una struttura è una collezione di variabili che possono rappresentare un oggetto o una entità complessa. Ogni membro di una struttura può essere di qualsiasi tipo di dati C, compresi tipi di dati primitivi (come int, float, char) o altre strutture.
Ecco come dichiarare e utilizzare una struttura in C:

```c
#include <stdio.h>

// Dichiarazione di una struttura
struct Student {
  char name[50];
  int age;
  float gpa;
};

int main() {
  // Dichiarazione di una variabile di tipo struct Student
  struct Student student1;

  // Accesso ai membri della struttura e assegnazione di valori
  strcpy(student1.name, "Alice");
  student1.age = 20;
  student1.gpa = 3.5;

  // Stampa dei membri della struttura
  printf("Nome: %s\n", student1.name);
  printf("Eta: %d\n", student1.age);
  printf("GPA: %.2f\n", student1.gpa);

  return 0;
}
```

Esempio: Un algoritmo che gestisce informazioni su persone (Joe e Frank) utilizzando strutture e funzioni.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

// Definizione di una struttura "person" per rappresentare informazioni su una
  persona.
struct person {
  char *name;       // Nome della persona
  int age;          // Eta'della persona
  float height;     // Altezza della persona
  float weight;     // Peso della persona
};

// Funzione per creare una nuova persona e restituire un puntatore a essa.
struct person *person_create(char *name, int age, float height, float weight) {
  struct person *who = malloc(sizeof(struct person)); // Alloca memoria per una
  persona
  assert(who != NULL); // Assicurati che l'allocazione di memoria sia riuscita
  who->name = strdup(name); // Copia il nome nella nuova struttura
  who->age = age;     // Imposta l'eta
  who->height = height; // Imposta l'altezza
  who->weight = weight; // Imposta il peso
  return who; // Restituisce il puntatore alla nuova persona
}

// Funzione per liberare la memoria allocata per una persona.
void person_destroy(struct person *who) {
  assert(who != NULL);
  free(who->name); // Libera la memoria del nome
  free(who); // Libera la memoria della struttura "person" stessa
}

// Funzione per stampare le informazioni di una persona.
void person_print(struct person *who) {
  printf("Name: %s\n", who->name);
  printf("\tAge: %d\n", who->age);
  printf("\tHeight: %f\n", who->height);
  printf("\tWeight: %f\n", who->weight);
}

int main(int argc, char **argv) {
  // Creazione di due persone
  struct person *joe = person_create("Joe Alex", 32, 1.80, 64);
  struct person *frank = person_create("Frank Blank", 20, 1.90, 72);

  // Stampa le informazioni sulle due persone
  printf("Joe is at memory location %p:\n", joe);
  person_print(joe);
  printf("Frank is at memory location %p:\n", frank);
  person_print(frank);

  // Modifica alcune informazioni su Joe e Frank e stampa di nuovo
  joe->age += 20;
  joe->height -= 2;
  joe->weight += 40;
  person_print(joe);

  frank->age += 20;
  frank->weight += 20;
  person_print(frank);

  // Liberazione della memoria allocata per le due persone
  person_destroy(joe);
  person_destroy(frank);

  return 0;
}
```

Cosa fa questo codice:

- Include le librerie necessarie, tra cui `stdio.h`, `stdlib.h`, `assert.h`, e `string.h`.
- Definisce una struttura chiamata **person** che ha quattro membri: **name** (nome), **age** (età), **height** (altezza) e **weight** (peso). Questa struttura è utilizzata per memorizzare le informazioni su una persona.
- La funzione `person_creat` e è definita per creare una nuova persona. Accetta il nome, l'età, l'altezza e il peso come argomenti e restituisce un puntatore a una nuova struttura person inizializzata con queste informazioni.
- La funzione `person_destroy` è definita per liberare la memoria allocata per una persona. Accetta un puntatore a una struttura person come argomento e libera sia la memoria allocata per il nome che per la struttura stessa.
- La funzione `teperson_prinxtt` è definita per stampare le informazioni di una persona. Accetta un puntatore a una struttura person e stampa il nome, l'età, l'altezza e il peso.
- La funzione `main` è la funzione principale del programma e svolge le seguenti azioni:
  - Crea due persone, "Joe Alex" e "Frank Blank", utilizzando la funzione `person_create`.
  - Stampa le informazioni sulle due persone, inclusi i nomi, le età, le altezze e i pesi.
  - Modifica le informazioni su Joe e Frank (aumentando l'età, modificando l'altezza e il peso) e stampa le informazioni aggiornate.
  - Libera la memoria allocata per le due persone utilizzando la funzione `person_destroy`

5 Tipi e strutture dati

5.1 Definizione: Dato

In un linguaggio di programmazione il dato è un valore che una determinata variabile può assumere.

5.2 Definizione: Tipo di Dato (astratto)

Un **tipo di dato** è costituito da un inseme di valori e dagli operatori che su questi valori hanno senso.

Esempio: interi
Il numero sette è un dato, che può essere assegnato ad una variabile di tipo intero. La collezione dei numeri interi , con le sue usuali operazioni aritmetiche , è un tipo di dato.

5.3 Definizione: Struttura dati

Una struttura dati è la collezione di dati cosi come memorizzati nel calcolatore, insieme con i programmi che su di loro agiscono

Esempio: Vettore
Un vettore è una sequenza di elementi dello stesso tipo. Sugli elementi si possono effettuare le operazioni di lettura e scrittura. In entrambe le operazioni si accede direttamente al generico elemento specificandone òa posizione (indice) all'interno della sequenza

5.4 Tipo di dato (astratto): Sequenza

Una sequenza è l'insieme degli elementi caratterizzati da una posizione pos_i:

$$S = s_1, s_2, s_3, ..., s_n.$$

5.4.1 Dichiarazione di una sequenza in C: Codice

```
#define MAX_SIZE 100
typedef struct {
  int data[MAX_SIZE];
  int length;
} Sequence;

Sequence createSequence() {
  Sequence seq;
  seq.length = 0;
  return seq;
}
```

5.4.2 Metodi per la sequenza

- `boolean empty()`:

```
bool isEmpty(Sequence seq) {
  return seq.length == 0;
}
```

- `boolean finished(Pos p)`:

```
bool finisched(int p,Sequence seq){
  if (p==0){
    return true;
  } else{
    if (seq.data[p-1]==seq.data[p]){
      return true;
    } else{
      return false;
    }
  }}
```

- head():

```
int head(Sequence seq) {
  if (isEmpty(seq)) {
    printf("Error: empty sequence\n");
    return -1;
  } else {
    return seq.data[0];
  }
}
```

- tail():

```
int tail(Sequence seq) {
  if (isEmpty(seq)) {
    printf("Error: empty sequence\n");
    return -1;
  } else {
    return seq.data[seq.length - 1];
  }
}
```

- nexy(Pos p):

```
int next(Sequence seq , int p){
  if (finisched(p,seq)){
    printf("Error: empty sequence\n");
    return -1;
  } else{
    return seq.data[p+1];
  }
}
```

- prev(Pos p):

```
int prev (Sequence seq, int p){
  if (finisched(p,seq)){
    printf("Error: empty sequence\n");
    return -1;
  } else{
    return seq.data[p-1];
  }
}
```

- insert(Pos p,Item v):

```
void insert(int p, int x, Sequence *seq){
  if (seq->length==MAX_SIZE){
    printf("errore sequenza piena");
  }else{
    for (int i=seq->length;i>p;i--){
      seq->data[i]=seq->data[i-1];
    }
    seq->data[p]=x;
    seq->length++;
  }
}
```

- remove(Pos P):

```
void removeElement (int p, Sequence *seq){
  if (isEmpty(*seq)){
    printf("Error: empty sequence\n");
  } else{
    for (int i=p;i<seq->length;i++){
      seq->data[i]=seq->data[i+1];
    }
    seq->length--; }}
```

Nei metodi sopra riportati hanno tutti (ad eccezione dei metodi `insert()` e `remove()`) un costo nell'ordine di 1 $o(1)$, invece i metodi `insert()` e `remove()` hanno un costo nell'ordine di n $o(n)$ che varia in base agli elementi da spostare

5.5 Liste

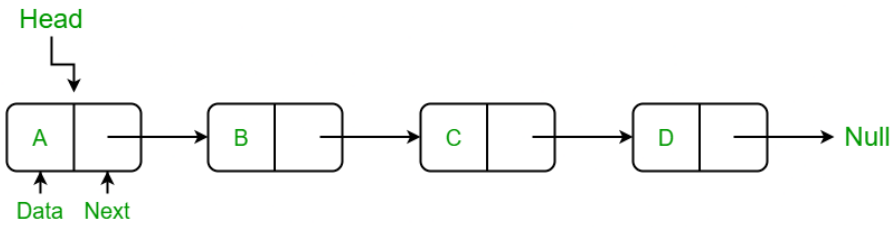
Definizione: Le liste sono delle *strutture dati* che realizzano il tipo di astratto "Sequence".

5.5.1 Liste con puntatori singoli

Una **lista con puntatori singoli** ci consente di realizzare gli operatori `empty , head, next` e `insert` con una complessità nel ordine di 1 ($o(1)$) e gli operatori `prev` e `remove` con una complessità nel ordine di n ($o(n)$).
ciascun nodo nella lista ha due componenti:

- Un campo `value`
- Un campo `next` che contiene il puntatore al prossimo elemento

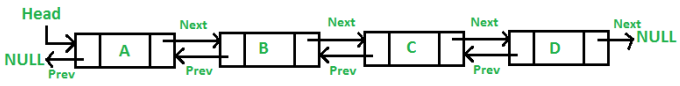
La lista stessa è realizzata con un puntatore al primo elemento (se esiste, altrimente **nil** e l'ultimo elemento della lista contiene nel campo `next` (che è quello in cui viene inserito il puntatore alla prossima "cella") il valore `nil`



5.5.2 Liste con puntatore doppio

Una lista a puntatore doppio è una struttura dati dinamica che consiste di una sequenza di nodi, ognuno dei quali contiene un campo di dati e due riferimenti, uno al nodo precedente e uno al nodo successivo.
Quindi la struttura della lista sarà:

```
typedef struct node{
  struct node *prev; \\contiene il puntatore all'elemento precedente
  int data;
  struct node *next; \\contiene il puntatore all'elemento successivo
}Node;
```



Ecco alcuni esempi di operazioni che possono essere eseguite su una lista a puntatore doppio:

- **Inserimento di un elemento**: Per inserire un elemento alla fine della lista, è sufficiente creare un nuovo nodo e collegarlo al nodo finale. Per inserire un elemento all'inizio della lista, è necessario collegare il nuovo nodo al nodo iniziale

```
void insert_at_head(list_node **head, int value) {
 // Crea un nuovo nodo
 list_node *new_node = malloc(sizeof(struct list_node));
 new_node->value = value;

 // Collega il nuovo nodo al nodo iniziale
 new_node->next = *head;
 new_node->prev = NULL;

 // Collega il nodo iniziale al nuovo nodo
 if (*head != NULL) {
   (*head)->prev = new_node;
 }
 *head = new_node;// Imposta il nuovo nodo come nodo iniziale
}
```

- **Eliminazione di un elemento**:Per eliminare un elemento dalla lista, è sufficiente scollegare il nodo desiderato dai nodi adiacenti.

```
void delete_element(struct list_node **head, int value) {
  // Trova il nodo da eliminare
  struct list_node *node = *head;
  while (node != NULL && node->value != value) {
    node = node->next;
  }

  // Controlla se il nodo e' stato trovato
  if (node == NULL) {
    return;
  }

  // Scollega il nodo dai nodi adiacenti
  if (node->prev != NULL) {
    node->prev->next = node->next;
  }
  if (node->next != NULL) {
    node->next->prev = node->prev;
  }

  // Libera la memoria allocata per il nodo
  free(node);
}
```

- **Ricerca di un elemento nella lista**:Per cercare un elemento nella lista, è possibile utilizzare un iteratore per navigare attraverso la lista fino a trovare l'elemento desiderato.

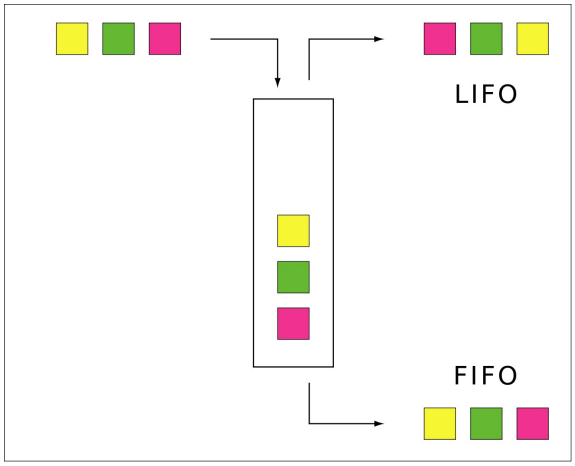
```
int cerca(int n,nodo* head){
  int posizione = -1;
  int trovato = 0;
  int i=0;
  while(head != NULL && trovato == 0){
    if(head -> elemento == n){
      posizione = i;
      trovato = 1;
    }
    i++;
    head = head -> successivo;
  }
  return posizione;
}
```

5.6 Pile

Una pila è un particolare tipo di lista in cui gli inserimenti e le cancellazioni avvengono ad un estremo della struttura , ovvero:

- Gli inserimenti avvengono solo dopo l'ultimo elemento
- La cancellazione avviene solo sull'ultimo elemento

Generalmente le liste utilizzano il principio di `LIFO` (Last In First Out), ovvero l'elemento inserito per ultimo è il primo elemento a uscire dalla lista.



Funzioni classiche quando si ha a che fare con le pile sono:

- `push(P,v)` per l'operazione di inserimento
- `pop(P)` per l'operazione di cancellazione del primo elemento(pop restituisce il contenuto dell'elemento cancellato)
- `top(P)` per l'operazione che legge l'ultimo elemento in cima alla pila

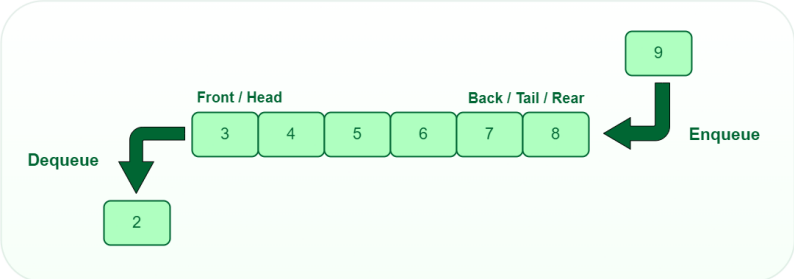
5.7 Code

il tipo di dato coda è una specializzazione della lista in cui:

- Gli eventi di inserimento avvengono solo all'estremo della lista(prima dalla testa, o dopo la coda)
- Gli eventi di cancellazione avvengono solo all'altro estremo (alla coda, oppure alla testa)

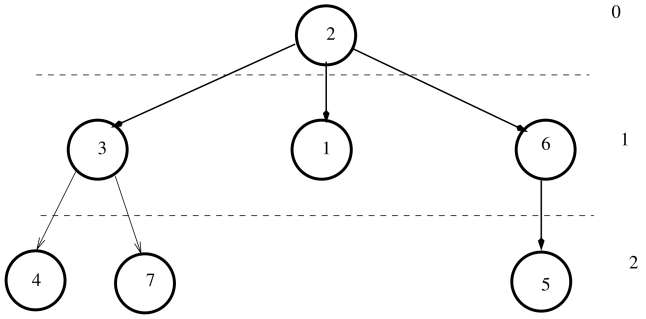
Le procedure per l'inserimento e la rimozione di un dato dato il contesto prendono il nome:

- `enqueue(Q,v)` per l'operazione di inserimento
- `dequeque(Q)` per l'operazione di cancellazione (che restituisce il contenuto dell'elemento cancellato)



5.8 Strutture ad albero

Un albero è una struttura dati gerarchica che consiste di nodi collegati tra loro da archi. Ogni nodo(tranne la radice) ha uno e un solo padre e può avere zero o più nodi figli. Il nodo senza figli è chiamato nodo foglia.



Profondità dei nodi: è la lunghezza del cammino semplice della radice al nodo(misurato in numero di archi)
Ex.: la radice ha profondità 0, i suoi figli hanno profondità 1 e i figli dei figli hanno profondità 2

Livello: l'inseme dei nodi alla stessa profondità

Altezza albero:La profondità massima delle sue foglie.

5.8.1 Albero binario

Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, identificati come figlio destro e figlio sinistro.

5.8.2 Alberi binari di ricerca

Un albero binario di ricerca (BST) è una struttura dati gerarchica in cui ogni nodo può avere al massimo due figli. I figli di un nodo sono chiamati figlio sinistro e figlio destro.

In un BST, i valori dei nodi sono ordinati in modo crescente. Ciò significa che il valore di ogni nodo è maggiore o uguale al valore del suo figlio sinistro e minore o uguale al valore del suo figlio destro.

Gli alberi binari di ricerca sono spesso utilizzati per implementare algoritmi di ricerca efficienti, come la ricerca binaria. La ricerca binaria funziona confrontando il valore dell'elemento da cercare con il valore del nodo radice. Se i due valori sono uguali, l'elemento è stato trovato. In caso contrario, il valore viene confrontato con il valore del figlio sinistro o del figlio destro del nodo radice, a seconda di quale dei due valori sia minore. Il processo viene ripetuto fino a quando l'elemento viene trovato o fino a quando viene raggiunto un nodo foglia, che indica che l'elemento non è presente nell'albero.

5.8.3 Alberi bilanciati

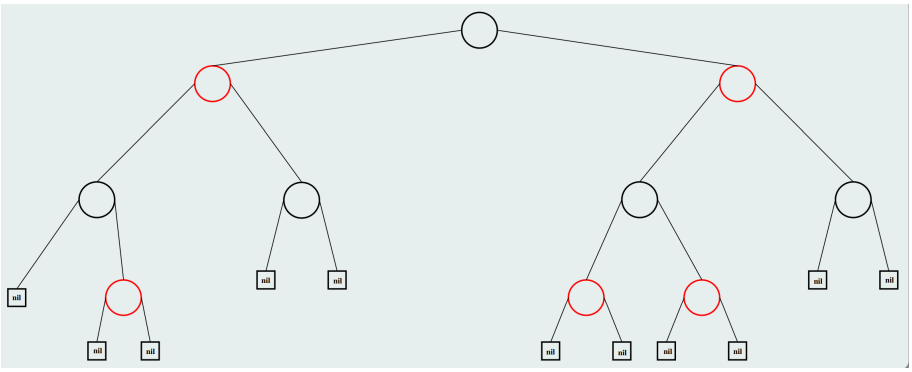
La bilanciatura di un albero binario di ricerca (BST) consiste nel mantenere la struttura dell'albero in modo tale che la profondità di ogni nodo sia il più vicina possibile alla profondità media dell'albero, il **fattore di bilanciamento** si indica con $\beta(v)$. Ciò migliora l'efficienza delle operazioni di ricerca e inserimento, che sono operazioni $O(log\, n)$ in un BST bilanciato.

Esistono diversi tipi di alberi bilanciati:

- **Alberi AVL** : Gli alberi AVL sono BST bilanciati in cui la differenza tra la profondità dei sottoalberi sinistro e destro di ogni nodo non supera mai 1 ($\beta(v) \leq 1$). Il bilanciamento è ottenuto tramite **rotazioni**.

- **B-alberi**: Un B-albero (in inglese: B-tree) è una struttura dati che permette la rapida localizzazione dei file (record o chiavi), specie nelle basi di dati, riducendo il numero di volte che un utente necessita per accedere alla memoria in cui il dato è salvato. I B-alberi sono generalmente "piatti" ovvero la loro altezza generalmente è tra 1 e 2
- **Alberi 2-3**: Gli alberi 2-3 sono BST bilanciati in cui ogni nodo può avere al massimo due figli, uno sinistro e uno destro. I nodi interni degli alberi 2-3 possono contenere al massimo due valori.
- **Alberi rosso - neri**:Gli alberi rosso-neri sono un tipo di albero binario di ricerca (BST) bilanciato. Gli alberi rosso-neri soddisfano le seguenti proprietà:
  - Ogni nodo è rosso o nero.
  - Le **chiavi** vengono mantenute solo nei nodi interni dell'albero
  - Le foglie sono costituite da nodi speciali `Nil`
  - Vengono rispettati i seguenti vincoli:
    * la radice è nera
    * tutte le foglie sono nere
    * entrambi i figli di un nodo rosso sono neri
    * Ogni cammino semplice da un nodo u ad una delle foglie contenute nel suo sottoalbero ha lo stesso numero di nodi neri

  Queste proprietà garantiscono che l'altezza di un albero rosso-nero sia sempre $O(log\, n)$, dove n è il numero di nodi dell'albero.



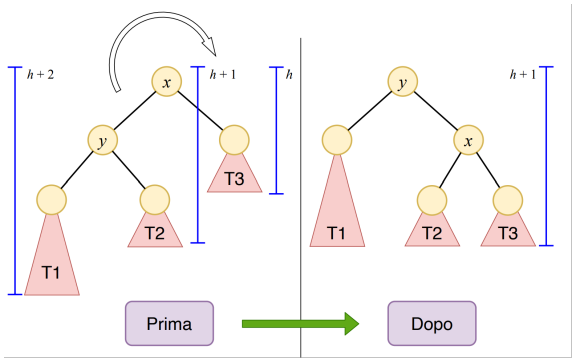
Memorizzazione alberi Red-Black: in aggiunta ai campi classici di un albero binario ovvero:

- Tree `parent`
- Tree `left`
- Tree `right`
- **int** `color`
- Item `key`
- Item `value`

Altezza nera di un nodo v**)**: l'altezza nera $bh(v)$ di un nodo v è il numero di nodi neri lungo ogni cammino da v(escluso) ad ogni foglia (inclusa) del suo sottoalbero
Altezza nera di un albero Red-Black: L'altezza nera di un albero Red-Black è pari all'altezza nera della sua radice

Inserimento: E' possibile che durante la modifica di un albero Red-Black le condizioni di bilanciamento risultino violate.
Quando i vincoli sopracitati vengono violati si può agire nei seguenti modi :

- Modificando i colori nella zona della violazione
- Operando dei ribilanciamenti dell'albero tramite **rotazioni** rispettivamente **rotazione destra** e **rotazione sinistra**



Algorithm 1 Rotazione a sinistra

```
1:  function TREE ROTATELEFT(Tree x)
2:      Treey ← x.right;
3:      Treep ← x.parent;
4:      x.right ← y.left;
5:      if y.left ≠ nil then
6:          y.left.parent ← x;
7:      end if
8:      y.left ← x;
9:      x.parent ← y;
10:     y.parent ← p;
11:     if p ≠ nil then
12:         if p.left = x then
13:             p.left ← y;
14:         else
15:             p.right ← y;
16:         end if
17:     end if
18: end function
```

```
Function insertNode (Tree t, item x, item v)
    Tree p ← nil;
    Tree n ← u ← t;
    while u ≠ nil and u.key ≠ x do
        p ← u;
        u ← if (x < u.key, u.left, u.right);
    if u ≠ nil then
        u.value ← v;
    else
        Tree n ← Tree(x, v);
        link (p, n, x);
        balanceInsert (n);
    while n.parent ≠ nil do
        n ← n.parent;
    Risultato n
```

```
Function balanceInsert (Tree t)
    t.color ← RED;
    while t ≠ nil do
        Tree p ← t.parent;
        Tree n ← if(p ≠ nil, p.parent, nil);
        Tree z ← if(n = nil, nil, if(n.left = p, n.right, n.left));
        if p = nil then
            t.color ← BLACK; t ← nil;
        else if p.color = BLACK then
            t ← nil;
        else if z.color = RED then
            p.color ← z.color ← BLACK; n.color ← RED; t ← n;
        else
            if (t = p.right) and (p = n.left) then
                rotateLeft(p); t ← p;
            else if (t = p.left) and (p = n.right) then
                rotateRight(p); t ← p;
            else
                if (t = p.left) and (p = n.left) then rotateRight(p) ;
                else if (t = p.right) and (p = n.right) then rotateLeft(p) ;
                p.color ← BLACK; n.color ← RED; t ← nil;
```

```
Function removeNode (Tree T, item x)
    Tree u ← lookupNode(T, x);
    if u ≠ nil then
        if u.left ≠ nil and u.right ≠ nil then
            Tree s ← u.right;
            while s.left ≠ nil do
                s ← s.left;
            u.key ← s.key; u.value ← s.value; u ← s; x ← s.key;
        Tree t;
        if u.left ≠ nil and u.right = nil then
            t ← u.left;
        else
            t ← u.right;
        link(u.parent, t, x);
        if u.color=BLACK then balanceDelete(T, t) ;
        if u.parent = nil then T ← t ;
        delete u;
    while T.parent ≠ nil do
        T ← T.parent;
    return T;
```

```
Function balanceDelete (Tree T, Tree t)
    while t ≠ T and t.color = BLACK do
        Tree p ← t.parent;
        if t = p.left then
            Tree f ← p.right;
            Tree ns ← f.left;
            Tree ns ← f.right;
            if (f.color = RED) then
                p.color ← RED; f.color ← BLACK; rotateLeft(p);
            else
                if (ns.color = nd.color = BLACK) then
                    f.color ← RED; t ← p;
                else if (ns.colorRED and nd.color = BLACK) then
                    ns.color ← BLACK; f.color ← RED; rotateRight(f);
                else if nd.color = RED then
                    f.color ← RED; p.color ← BLACK;
                    nd.color ← BLACK; rotateLeft(p);
                    t ← T;
        else
            % Codice speculare al precedente;
    if t ≠ nil then t.color = BLACK;
```

6 Ordinamento

6.1 Problema dell'ordinamento

Problema dell'ordinamento: Abbiamo una collezione di oggetti R_i ciascuno con una chiave K_i. Le chiavi ammettono una relazione di ordinamento: **Tricotomia** e la **transitività**

Perché abbiamo bisogno di un inseme ordinato ? Perché la ricerca in un inseme ordinato ha un costo di ordine $0(log\, n)$, quindi avere una sequenza in ordine può far risparmiare tempo.

6.1.1 Permutazioni

Una **Permutazione** è una sequenza di n di ogetti distinguibili tra loro in uno specifico ordine.

Se l'insieme degli oggetti è l'inseme 0,1,2,3,... possiamo usarli come *indici* per accedere un altro insieme di oggetti

6.2 Bubble sort

Il **bubble sort** è un semplice algoritmo di ordinamento che ripetutamente scorre la lista, o vettore, confronta i due elementi vicini e li scambia se si trovano nell'ordine sbagliato. Lo scorrimento attraverso gli elementi prosegue fino a quando la lista non è completamente ordinata. Analizzando

Algorithm 1: Bubble sort

Input: Elementi dell'insieme $V = [V[0], V[1], \ldots, V[n-1]]$;
change $\leftarrow$ *true* ;
while *change* **do**
    change $\leftarrow$ *false*;
    for $i \leftarrow 0$ **to** $n-2$ **do**
        if $V[i] > V[i+1]$ **then**
            Scambia $V[i] \leftrightarrow V[i+1]$;
            change $\leftarrow$ *true*;

la performance dell'algoritmo di ordinamento possiamo osservare che:

- nel caso peggiore, cioè quando tutti gli elementi contenuti sono da ordinare, presenta una complessità computazionale $O(n^2)$ dove n corrisponde al numero di elementi da ordinare nella lista.
- nel caso migliore, ovvero quando la lista è completamente ordinata e non si deve svolgere alcuno scambio, possiede una complessità computazionale $O(n)$. In questo caso il bubble sort risulta più efficiente di altri algoritmi di ordinamento, in quanto l'abilità di verificare che la lista sia ordinata è implementata direttamente nell'algoritmo.

6.3 Insertion Sorting

L'insertion sort, detto anche algoritmo a inserimento, è un algoritmo di ordinamento che ordina una lista, o vettore, passando un elemento alla volta, analogamente a quando ordiniamo un mazzo di carte.
Nonostante sia molto meno efficiente su vettori molto grandi rispetto ad algoritmi più avanzati, può avere alcuni vantaggi, tra i quali: è semplice da implementare, è più efficiente su data set più piccoli e già parzialmente ordinati, è più efficiente di altri algoritmi di ordinamento con complessità quadratica come il bubble sort e selection sort.

Algorithm 2: Insertion Sorting

Input: Array contenente le chiavi $V[i]$, $i = 0, \ldots, n-1$;
for $i \leftarrow 1$ **to** $n-1$ **do**
    $temp \leftarrow V[i]$;
    $j \leftarrow i$;
    while $j > 0$ **and** $V[j-1] > temp$ **do**
        $V[j] \leftarrow V[j-1]$;
        $j \leftarrow j-1$;
    $V[j] \leftarrow temp$;

Quale è il costo?

- ad ogni iterazione del ciclo `For␣i␣=␣0...␣n-1`, abbiamo un ciclo intero,ed un paio di assegnazioni
- Il ciclo interno viene eseguito per valori `j` che possono andare (nel caso peggiore) i a 0, mentre nel caso migliore non verrà eseguita nessuna operazione.

Ogni iterazione del ciclo esterno ha un costo potenzialmente diverso. Quindi nel caso peggiore

$$opcnt = \left(\sum_{i=0}^{n-1} 1 + 1\right) + \left(\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1\right) \leq 2n + \sum_{i=0}^{n-1}(i+1) = 3n + \frac{(n-1)n}{2} = O(n^2)$$

mentre nel caso migliore sopravvive solo il ciclo esterno e quindi il costo è $\Omega(n)$

6.4 Quicksort

Il **quick sort** è un algoritmo di ordinamento ricorsivo in loco (o in place), in particolare utilizza il metodo **divide et impera** per ordinare i dati di un vettore. La procedura ricorsiva avviene nel seguente modo: viene scelto un elemento dall'array, detto pivot, si posizionano gli elementi minori a sinistra del pivot, mentre gli elementi più grandi a destra. In questo modo ci saranno due sottogruppi divisi dal pivot, il quale si troverà nella posizione finale. Il processo verrà ripetuto in ciascun sottogruppo fino a che il vettore non sarà completamente ordinato.

Algorithm 4: QuickSort

Input: Elementi dell'insieme $V = (V[0], V[1], \ldots, V[n-1])$, *primo*, *ultimo* ;
$k \leftarrow$ perno $(V, primo, ultimo)$;
Quicksort$(V, primo, k-1)$;
Quicksort$(V, k+1, ultimo)$;

Algorithm 5: Perno

Input: Elementi dell'insieme $V = (V[0], V[1], \ldots, V[n-1])$, *primo*, *ultimo* ;
$x \leftarrow V[primo]$;
$k \leftarrow primo$;
for $i \leftarrow primo$ **to** *ultimo* **do**
  if $V[i] < x$ **then**
    $k \leftarrow k+1$;
    Scambia $V[i] \leftrightarrow V[k]$;
$V[primo] \leftarrow V[k]$;
$V[k] \leftarrow x$;
Risultato k

6.4.1 Complessità Computazionale del Quicksort

La complessità dell'algoritmo quicksort dipende dal caso in cui viene eseguito.

- Caso migliore: $O(n \log n)$
  Il caso migliore si verifica quando il pivot viene scelto in modo tale da dividere l'array in due parti uguali ogni volta. In questo caso, quicksort impiega O(n log n) operazioni per ordinare l'array, che è la stessa complessità di heapsort e merge sort.

- Caso medio: $O(n \log n)$
  Il caso medio si verifica quando il pivot viene scelto in modo casuale. In questo caso, la probabilità che il pivot non sia un buon pivot è molto bassa. Di conseguenza, la complessità di quicksort nel caso medio è ancora O(n log n).

- Caso peggiore: $O(n^2)$ Il caso peggiore si verifica quando il pivot viene scelto in modo tale da dividere l'array in due parti sbilanciate ogni volta. In questo caso, quicksort impiega $O(n^2)$ operazioni per ordinare l'array, che è la stessa complessità di bubble sort e insertion sort.

6.5 Mergesort

Merge sort è un algoritmo basato sul principio del divide et impera (come il quick sort), che suddivide un problema in sottoproblemi più piccoli e poi li risolve ricorsivamente. Merge sort è un algoritmo di ordinamento stabile, il che significa che l'ordine degli elementi uguali viene preservato nell'output ordinato

Algorithm 6: MergeSort

Input: Elementi dell'insieme $V = (V[0], V[1], \ldots, V[n-1])$, *primo*, *ultimo* ;
if *primo* $<$ *ultimo* **then**
    mezzo $\leftarrow \lfloor (primo + ultimo)/2 \rfloor$;
    Mergesort(V, *primo*, *mezzo*);
    Mergesort(V, *mezzo* $+ 1$, *ultimo*);
    Merge(V, *primo*, *ultimo*, *mezzo*);

Algorithm 7: Merge

Input: Elementi dell'insieme $V = (V[0], V[1], \ldots, V[n-1])$, *primo*, *ultimo*, *mezzo* ;
$i \leftarrow primo$;
$j \leftarrow mezzo + 1$;
$k \leftarrow primo$;
while $i \leq mezzo$ **and** $j \leq ultimo$ **do**
    if $V[i] \leq V[j]$ **then**
        $T[k] = V[i]$;
        $i \leftarrow i + 1$;
    else
        $T[k] = V[j]$;
        $j \leftarrow j + 1$;
    $k \leftarrow k + 1$;
for $h \leftarrow mezzo$ **downto** i **do**
    $V[j] \leftarrow V[h]$;
    $j \leftarrow j - 1$;
for $j \leftarrow primo$ **to** $k - 1$ **do**
    $V[j] \leftarrow T[j]$;

6.5.1 Complessità Computazionale del Mergesort

La complessità computazionale del mergesort è analoga a quella del Quicksort, dato che ad ogni passo ricorsivo sto spezzando la sequenza iniziale in due sottosequenze eguali, per ciò $T(n) = O(nlog(n)))$

Osservazione: Il **Mergesort** è un algroritmo migliore del quicksort in termini di complessità nel caso peggiore $O(nlog(n))$ per il mergesort e $O(n^2)$ per il Quicksort. Tuttavia, nel caso medio il Quicksort è più veloce