**9° lezione programmazione II**

**Complessità (2° parte)**

**Paradigma Divide et imper**

Secondo il paradigma divide et impera, risolviamo un problema in modo ricorsivo applicando tre passaggi ad ogni livello della ricorsione:

1. **Dividi:** dividere il problema in un numero di sotto problemi che sono istanze più piccole dello stesso problema.
2. **Conquista:** risolvere i sottoproblemi ricorsivamente. Se le dimensioni del sotto problema sono abbastanza piccole, si possono risolvere i sotto problemi in modo semplice.
3. **Combina:** combinare le soluzioni dei sotto problemi ottenendo la soluzione del problema originale.

**Relazione di ricorrenza**

La complessità di una funzione ricorsiva può essere espressa mediante una relazione di ricorrenza. L’analisi degli algoritmi ricorsivi si riduce spesso alla risoluzione di una o più equazioni di ricorrenza nelle quali si esprime il termine n-esimo di una sequenza in funzione dei termini precedenti. Questi descrivono in modo preciso

le prestazioni degli algoritmi ricorsivi in esame. Prima di capire quali sono i metodi fondamentali per l’analizzare questi algoritmi, vediamo come calcolare il tempo di esecuzione quando si ha un ciclo. Il tempo totale di esecuzione di un ciclo può essere espresso come sommatoria dei tempi impiegati in ogni esecuzione del ciclo.

**La linearità delle sommatorie**

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, diagramma

Descrizione generata automaticamente

**Quadrati e cubi di sommatorie**

**Immagine che contiene testo, Carattere, linea, diagramma

Descrizione generata automaticamente**

**Sommatorie e serie geometriche**

**Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente**

**Come si calcola il valore di una serie numerica?**

**Si calcola con utilizzando il metodo d’induzione (dimostrazione n= 0 ok, n+1 otutto n)**

**Immagine che contiene testo, Carattere, diagramma, bianco

Descrizione generata automaticamente**

**Per n=1 è vero, se vale anche per n+1 è vero per ogni**

**n naturale.Immagine che contiene testo, Carattere, diagramma, linea

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, ricevuta, schermata

Descrizione generata automaticamente**

**Immagine che contiene testo, diagramma, Carattere, linea

Descrizione generata automaticamente**

**Ricorrenze fondamentali**

* Questa formula si usa per programmi che ciclando sull’input ‘’eliminando’’ un elemento per volta.Immagine che contiene testo, Carattere, calligrafia, bianco

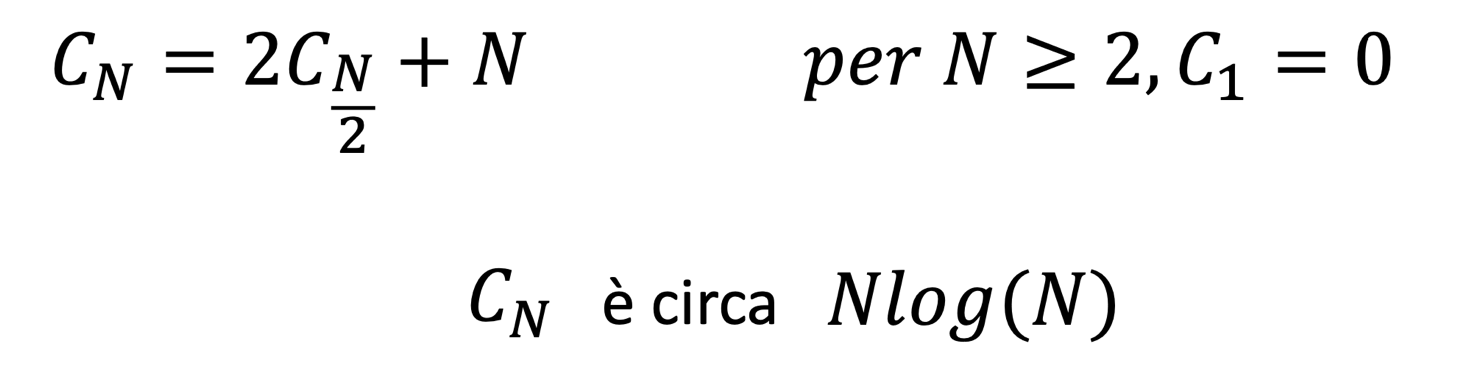
  Descrizione generata automaticamente

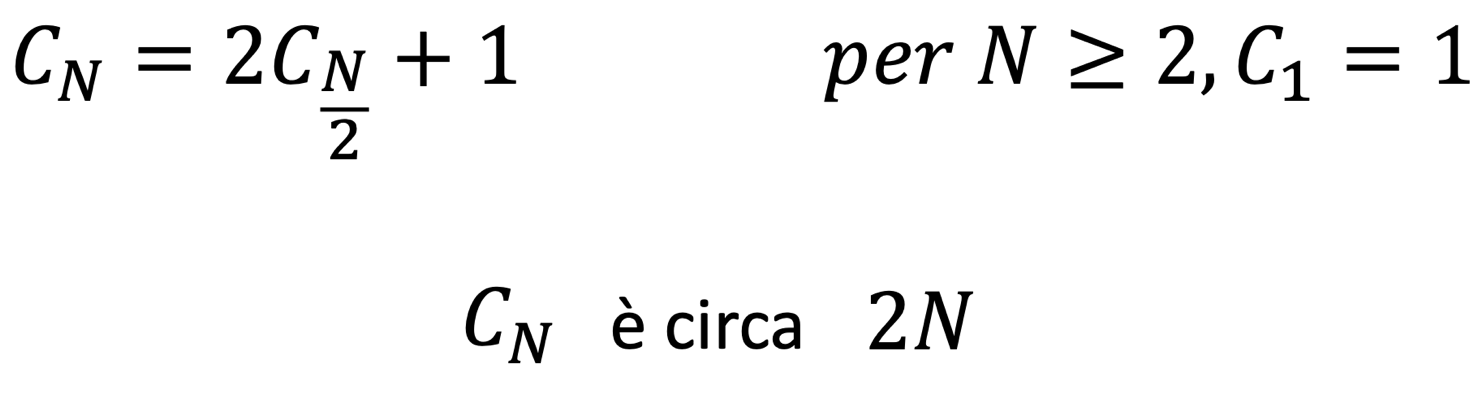
**Immagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamente**

**Immagine che contiene testo, Carattere, calligrafia, bianco

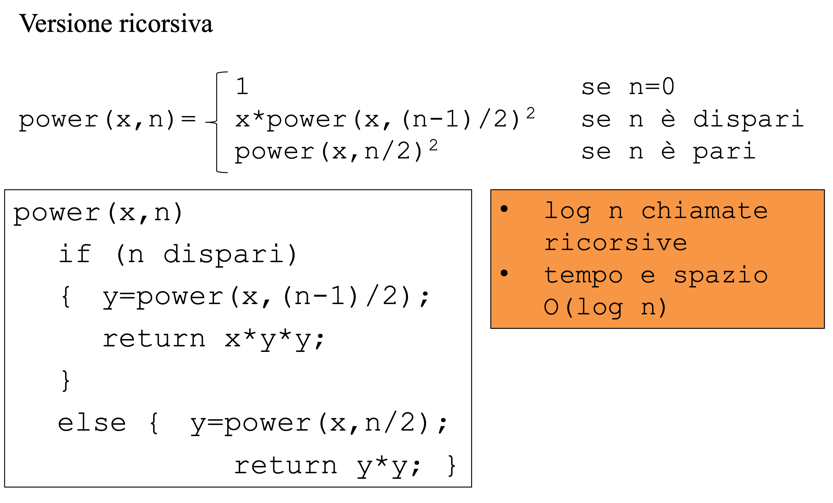
Descrizione generata automaticamente**

* Questa formula si usa tipicamente con un programma ricorsivo che dimezza l’input ad ogni passo(con N pari).
* Questa formula si usa tipicamente con un programma ricorsivo che esegue una scansione lineare dell’input prima, durante, oppure dopo aver suddiviso l’input in due parti.



* Questa formula si usa tipicamente con un programma ricorsivo che dimezza l’input ed esegue una quantità di lavoro addizionale costante.

Esempio del calcolo di potenze



**Esempio: segmenti di somma massima (1° soluzione)**

* **Segmento: sequenza di elementi consecutivi in una sequenza S**

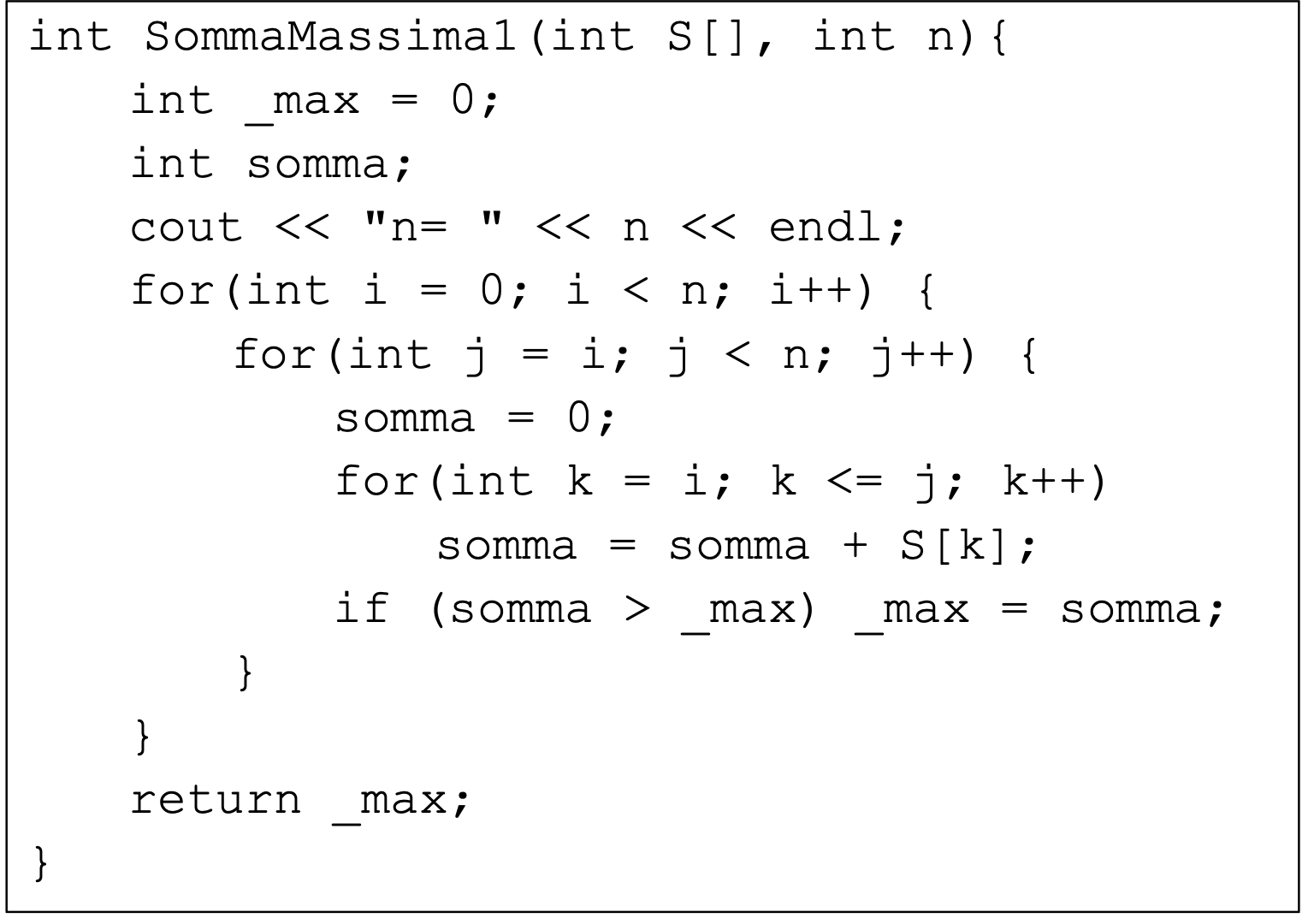
**– S array di n interi**

**– S[i,j] segmento se 0 ≤ i ≤ j ≤ n-1**

**– almeno un intero è positivo**

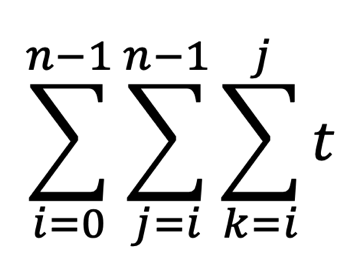
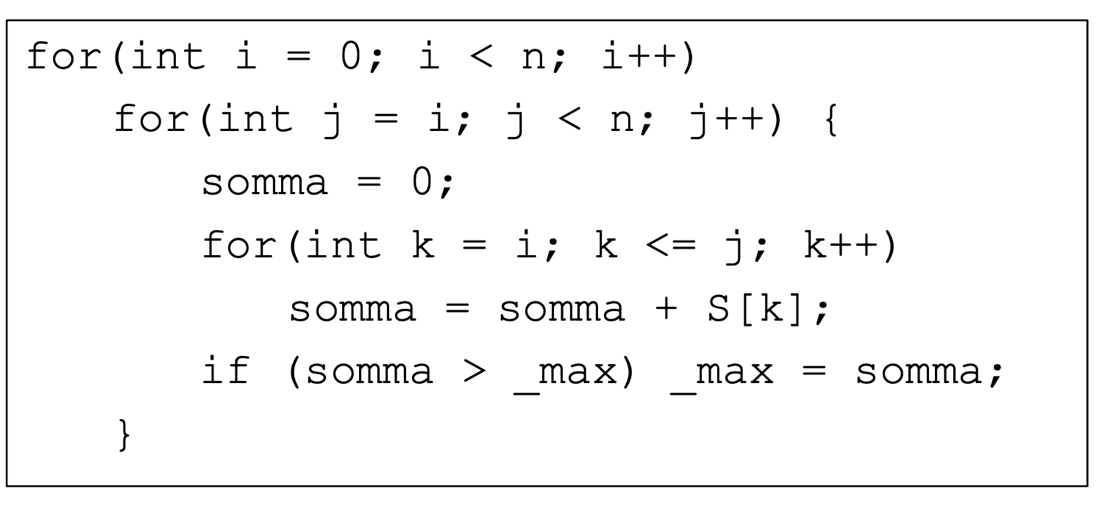
**Determinare il segmento di somma massima**

**– A parità di somma si predilige il segmento più corto.**

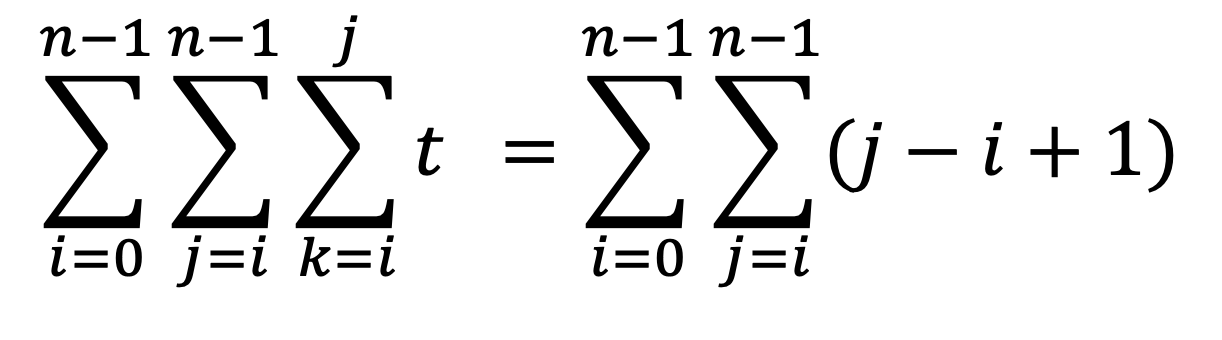


**In questo frame di codice per ogni coppia di indici (i,j) considero la somma del segmento S[i…j]. (Costo: ~n3)**

**Come si calcola il costo di questo frame di codice?**

****

* **t ha un costo costante (es. 1).**
* **Il ciclo for più interno viene eseguito al massimo j – i + 1 volte (lunghezza del segmento esaminato).**
* **Si hanno tre sommatorie perché vi sono 3 cicli for annidati**

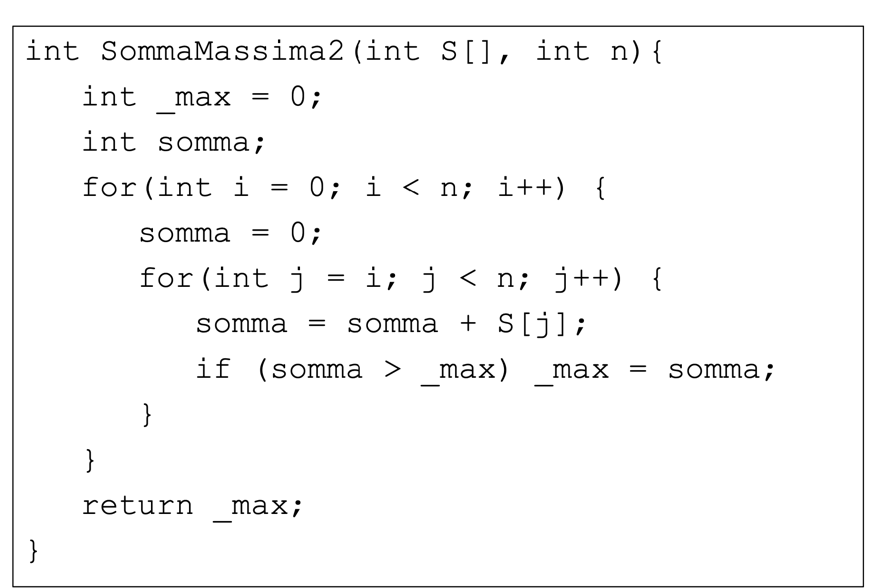
****

**Esempio: segmenti di somma massima (2° soluzione)**

**Una volta calcolata somma(a[i,j-1]) evitiamo di ripartire da capo per somma(a[i,j]**

**Utilizziamo il fatto che somma(a[i,j])=somma(a[i,j-1]) + a[j]**

**Questo ci permette di risparmiare un ciclo for. Dunque, otteniamo una soluzione quadratica**

****

li ciclo esterno seleziona l'elemento iniziale, il ciclo interno trova la somma massima possibile con il primo elemento selezionato dal ciclo esterno e confronta questo massimo con il massimo complessivo.

Infine, restituisce il massimo complessivo. La complessità temporale è O(n2).

**Algoritmi di ordinamento e ricerca**

**Ricerca sequenziale**

Si utilizza quando gli elementi non sono ordinati. Confronta la chiave con tutti gli elementi presenti.

**Immagine che contiene testo, ricevuta, Carattere, bianco

Descrizione generata automaticamente**

**La ricerca binaria (Binary Search)**

**La ricerca binaria (Binary Search)** è un algoritmo efficiente per trovare un elemento all'interno di un array ordinato. Ecco una descrizione dettagliata del processo e i passaggi fondamentali:

1. **Controllo dell'elemento centrale**:
   * Si inizia controllando l'elemento situato al centro dell'array.
   * Si confronta l'elemento centrale con l'elemento che si desidera trovare (chiamiamolo **target**).
2. **Confronto e scelta della parte dell'array**:
   * Se l'elemento centrale è uguale al **target**, la ricerca termina con successo.
   * Se il **target** è minore dell'elemento centrale, significa che, se presente, si troverà nella metà sinistra dell'array. Si aggiorna quindi il limite superiore dell'array alla posizione appena a sinistra del centro.
   * Se il **target** è maggiore dell'elemento centrale, significa che, se presente, si troverà nella metà destra dell'array. Si aggiorna quindi il limite inferiore dell'array alla posizione appena a destra del centro.
3. **Dimezzamento dell'array**:
   * Si dimezza il numero di elementi da controllare aggiornando i limiti superiore e inferiore dell'array secondo il risultato del confronto precedente.
4. **Ripetizione del processo**:
   * Questo procedimento viene ripetuto (tornando al passo 1) con la nuova porzione di array fino a che l'elemento **target** non viene trovato oppure finché la porzione di array da controllare non è vuota (cioè, il limite inferiore supera il limite superiore).

L'algoritmo termina con uno dei due risultati:

* **Elemento trovato**: Se l'elemento **target** è stato trovato durante una delle iterazioni.
* **Elemento non trovato**: Se la porzione di array da controllare diventa vuota senza che l'elemento **target** sia stato trovato.

**Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente**

**Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente**

**Ordinamento di sequenze**

**Ci sono svariati algoritmi di ordinamento tra i quali:**

* **Bubble Sort (algoritmo di ordinamento a bolle)**
* **Selection Sort (algoritmo di ordinamento per selezione)**
* **Inserction Sort (algoritmo di ordinamento per inserzione)**
* **Marge Sort**
* **Quick Sort**

**Bubble Sort**

Per ogni iterazione si confrontano gli elementi adiacenti e si scambiano i loro valori quando il primo è maggiore del secondo. Come conseguenza, abbiamo che il maggiore ‘’risale’’ fino alla cima del vettore ad ogni iterazione. (per tale motivo si chiama bubble bolle).Dopo *n-1* iterazioni l’array risulterà̀ ordinato.

**Data questa sequenza di numeri mostriamo i passi che si eseguono con questo algoritmo per ordinarlo.**

**(sequenza)**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

**Immagine che contiene testo, Carattere, schermata, diagramma

Descrizione generata automaticamente**

**Si ha una variabile bolena chiamata scambio che quando è avvenuto una scambia diventa true in mode che se non ci sono piu scambi finisce si ece dal while()**

**Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente**

**Selection Sort**

**L’ordinamento per selezione diretta seleziona l’elemento minore del sottoarray non ordinato e lo posiziona al primo posto.**

****

**Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente**

**void selectionSort(std::vector<int>& arr) {  
 int n = arr.size();  
 for (int i = 0; i < n - 1; ++i) {  
 // Trova l'indice del minimo elemento nella parte non ordinata  
 int min\_idx = i;  
 for (int j = i + 1; j < n; ++j) {  
 if (arr[j] < arr[min\_idx]) {  
 min\_idx = j;  
 }  
 }  
 // Scambia il minimo elemento trovato con l'elemento all'inizio della parte non ordinata  
 swap(arr[i], arr[min\_idx]);  
 }  
}**

**Selection Sort**

**L’ordinamento per inserimento** **diretto** consiste nell’inserire un elemento alla volta nella posizione che gli spetta in un vettore già ordinato, partendo da un vettore che contiene un solo elemento. Si aggiungono gli altri elementi uno per volta, posizionandoli direttamente nella posizione corretta.

**Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamente**

**Immagine che contiene calligrafia, Carattere

Descrizione generata automaticamente**

**Immagine che contiene testo, ricevuta, schermata, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente**

**Come si può migliorare questo algoritmo di ordinamento?**

Possiamo migliorarlo utilizzando **la ricerca binaria** **per collocare correttamente l’elemento a[i] nel sottoarray ordinato**.

Questo riduce il numero di passi per trovare la posizione corretta da O(n2) ad O(nlogn), tuttavia, ognuno degli elementi deve essere spostato di una posizione e ciò richiede comunque un costo totale di O(n2) nel caso medio e peggiore.

**Merge Sort**

Questo algoritmo implementa il **paradigma *divide et impera****:*

* **L’input di dimensione *n* viene partizionato in due parti di lunghezza *n/2*.**
* **Le due sottosequenze vengono ordinate in maniera ricorsiva fino a quando si ottengono delle sequenze composte da un solo elemento.**
* **A questo punto la procedura *merge* unisce due sottosequenze ordinate.**

**Immagine che contiene testo, Carattere, schermata, calligrafia

Descrizione generata automaticamente**

**Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente**

La **complessità è la medesima in tutti e tre i casi** **perché l’algoritmo divide sempre le sequenze a metà impiegando un tempo O(log n)** e **le unisce impiegando un tempo lineare.**

**Quick sort**

**Il Quicksort è l’algoritmo di ordinamento più efficiente. Si basa sulla divisione del vettore in tre partizioni:**

* **Centrale**: contenente un solo elemento detto *pivot*
* **Sinistra:** contenente tutti gli elementi minori del *pivot*
* **Destra:** contenente tutti gli elementi maggiori del *pivot*

Come conseguenza avremo che **tutti gli elementi della partizione sinistra saranno minori del più piccolo della partizione di destra. Si applica ricorsivamente l’algoritmo sulle partizioni sinistra e destra fino ad ordinare tutto il vettore. Il *pivot* può essere scelto a caso. Si può definire una funzione int *partition()* che si occupa di effettuare la partizione e restituire la posizione del pivot.**

**Nel seguente modo:**

**Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

* **Caso peggiore:** quando le due partizioni sono formate da 0 ed n-1 elementi. In questo caso il partizionamento costa O(n) e se questo caso si verifica ad ogni chiamata ricorsiva avremo un costo totale di O(n2).
* **Caso migliore:** bilanciamento massimo. Si verifica quando i due sottoproblemi hanno dimensione circa n/2. In questo caso il costo è O(nLogn)
* **Caso medio:** è possibile dimostrare che anche con una ripartizione

sproporzionata ad ogni livello di ricorsione, il quicksort viene eseguito nel tempo O(nLogn). Questo perché qualsiasi ripartizione con proporzionalità costante produce una ricorsione di profondità O(logn) il cui costo unitario è O(n).