Sommario

[Algoritmo 5](#_Toc96123483)

[Dati 5](#_Toc96123484)

[Efficienza di un algoritmo 6](#_Toc96123485)

[Teoria della complessità computazionale 7](#_Toc96123486)

[Complessità di tempo 7](#_Toc96123487)

[Complessità di tempo lineare 7](#_Toc96123488)

[Complessità di tempo quadratica 7](#_Toc96123489)

[Complessità di tempo cubica 7](#_Toc96123490)

[Complessità di tempo logaritmica 7](#_Toc96123491)

[Classifica di complessità 7](#_Toc96123492)

[Complessità di spazio 7](#_Toc96123493)

[Complessità asintotica 7](#_Toc96123494)

[Ridefinizione delle classi di complessità 8](#_Toc96123495)

[Linguaggio C 9](#_Toc96123496)

[Processo di compilazione 9](#_Toc96123497)

[Compilatore 9](#_Toc96123498)

[Libreria <stdio.h> 9](#_Toc96123499)

[Tipi 10](#_Toc96123500)

[Funzione sizeof(<tipo>) 10](#_Toc96123501)

[Variabile 11](#_Toc96123502)

[Variabile indeterminata 11](#_Toc96123503)

[Variabile definita 11](#_Toc96123504)

[Variabile indefinita 11](#_Toc96123505)

[Variabili scalari 11](#_Toc96123506)

[Costante 11](#_Toc96123507)

[Azioni sulle variabili 11](#_Toc96123508)

[Associazione di un valore ad una variabile 11](#_Toc96123509)

[Istruzione di scrittura 11](#_Toc96123510)

[Operatori 12](#_Toc96123511)

[Costrutti di selezione 13](#_Toc96123512)

[Costrutto if-then-else 13](#_Toc96123513)

[Costrutto di selezione a vie 13](#_Toc96123514)

[Costrutti di ripetizione (o iterazione) 14](#_Toc96123515)

[Ciclo for 14](#_Toc96123516)

[Ciclo do-while 14](#_Toc96123517)

[Ciclo while 15](#_Toc96123518)

[Ciclo innestato 15](#_Toc96123519)

[Puntatori 16](#_Toc96123520)

[Doppio puntatore 16](#_Toc96123521)

[Function 17](#_Toc96123522)

[Intestazione 17](#_Toc96123523)

[Chiamata 17](#_Toc96123524)

[Argomento 17](#_Toc96123525)

[Parametro 17](#_Toc96123526)

[Sintassi dell’intestazione 17](#_Toc96123527)

[Parte dichiarativa 17](#_Toc96123528)

[Sintassi di una function 17](#_Toc96123529)

[Procedure 18](#_Toc96123530)

[Intestazione 18](#_Toc96123531)

[Sintassi dell’intestazione 18](#_Toc96123532)

[Chiamata 18](#_Toc96123533)

[Approccio incrementale 19](#_Toc96123534)

[Organizzazione e relazione tra i dati 20](#_Toc96123535)

[Array 21](#_Toc96123536)

[Dichiarazione di un array 21](#_Toc96123537)

[Inizializzazione array 2D 21](#_Toc96123538)

[Accesso agli elementi di un array 21](#_Toc96123539)

[Mappa di memorizzazione 21](#_Toc96123540)

[Algoritmo di fusione di due array – Merge 22](#_Toc96123541)

[Algoritmo di ricerca sequenziale 24](#_Toc96123542)

[Stringhe 25](#_Toc96123543)

[Permutazione di una k-stringa 25](#_Toc96123544)

[Sottostringa 25](#_Toc96123545)

[Libreria <string.h> 25](#_Toc96123546)

[String matching 26](#_Toc96123547)

[Best matching 27](#_Toc96123548)

[Algoritmi di ordinamento 28](#_Toc96123549)

[Ordinamento per inserimento (insertion sort) 28](#_Toc96123550)

[Ordinamento per selezione di minimo (selection sort di minimo) 29](#_Toc96123551)

[Ordinamento per selezione di massimo (selection sort di massimo) 30](#_Toc96123552)

[Divide et impera 33](#_Toc96123553)

[Bottom up 33](#_Toc96123554)

[Top down 33](#_Toc96123555)

[Ricerca binaria 33](#_Toc96123556)

[Algoritmo di somma con raddoppiamento 35](#_Toc96123557)

[Ricorsività 36](#_Toc96123558)

[Formula ricorrente 36](#_Toc96123559)

[Esempio formula ricorrente 36](#_Toc96123560)

[Formula ricorrente del secondo ordine 36](#_Toc96123561)

[Fibonacci 36](#_Toc96123562)

[Algoritmo ricorsivo 37](#_Toc96123563)

[Esempio algoritmo ricorsivo 37](#_Toc96123564)

[Esempio algoritmo ricorsivo 37](#_Toc96123565)

[Processo 37](#_Toc96123566)

[Stack dei processi 38](#_Toc96123567)

[Programmazione ricorsiva 38](#_Toc96123568)

[Ricerca binaria ricorsiva 38](#_Toc96123569)

[Somma degli elementi di un array ricorsiva – approccio incrementale 41](#_Toc96123570)

[Somma degli elementi di un array ricorsiva – approccio divide et impera 42](#_Toc96123571)

[Massimo degli elementi di un array ricorsivo – approccio incrementale 44](#_Toc96123572)

[Massimo degli elementi di un array ricorsivo – approccio divide et impera 44](#_Toc96123573)

[Algoritmo di Euclide – Massimo comun divisore (MCD) 45](#_Toc96123574)

[Algoritmo di Fibonacci – complessità esponenziale 45](#_Toc96123575)

[Algoritmo di Fibonacci – complessità lineare – programmazione dinamica 45](#_Toc96123576)

[Algoritmo per il calcolo della lunghezza di una stringa ricorsivo 46](#_Toc96123577)

[Algoritmo per contare i numeri pari di un array ricorsivo 46](#_Toc96123578)

[Algoritmo di ricerca sequenziale ricorsiva 46](#_Toc96123579)

[Algoritmo di uguaglianza tra due stringhe ricorsivo 46](#_Toc96123580)

[Algoritmo di string matching ricorsivo 47](#_Toc96123581)

[Algoritmo di ordinamento per inserimento ricorsivo (insertion sort ricorsivo) 47](#_Toc96123582)

[Struct 48](#_Toc96123583)

[Record 48](#_Toc96123584)

[Dichiarazione di un record 48](#_Toc96123585)

[Accesso ai campi di una struct 49](#_Toc96123586)

[Numeri casuali e pseudocasuali 50](#_Toc96123587)

[Fenomeno casuale 50](#_Toc96123588)

[Fenomeno pseudocasuale 50](#_Toc96123589)

[Successione di numeri casuale 50](#_Toc96123590)

[Successione di numeri pseudocasuale 50](#_Toc96123591)

[Generatore di numeri pseudocasuali 50](#_Toc96123592)

[Cambio di intervallo 50](#_Toc96123593)

[Function rand 50](#_Toc96123594)

[Generare un tipo reale – intervallo 51](#_Toc96123595)

[Generare un tipo reale – intervallo 51](#_Toc96123596)

[Memoria dinamica 52](#_Toc96123597)

[Memoria stack e heap 52](#_Toc96123598)

[Gestione della memoria dinamica 52](#_Toc96123599)

[Libreria <malloc.h> 52](#_Toc96123600)

## Algoritmo

Un *algoritmo* è un procedimento per la risoluzione di una classe di problemi, descritto da una sequenza finita di *costrutti di controllo* e di *istruzioni* non ambigue per l’esecutore, che specifica una sequenza finita di operazioni, elementari per l’esecutore, che agiscono su un insieme finito di oggetti. Tali oggetti sono definiti *dati*.

Le 5 *proprietà* di un algoritmo sono:

* *Finitezza*
* *Non ambiguità*
* *Input*
* *Output*
* *Efficacia*

### Dati

I dati che devono essere noti prima dell’esecuzione sono detti *dati di inputi*.

Il risultato dell’esecuzione è detto *dato/i di output.*

I dati che non sono né di input né di output sono definiti *dati locali* (o *intermedi*).

## Efficienza di un algoritmo

Un algoritmo con *costo minore* viene definito *efficiente*.

L’*efficienza* di un algoritmo è definita da:

* Quanto *tempo* impiega l’algoritmo a risolvere il problema.
* Quanta *memoria*, e quindi *spazio*,occupal’*algoritmo*.

Tempo e memoria sono definite *risorse di calcolo*.

Il numero di operazioni e dati di un algoritmo è *proporzionale* al tempo e alla memoria richiesti per l’esecuzione del programma che implementa l’algoritmo.

Tuttavia:

* Per quanto riguarda le *operazioni*, si presta attenzione alle *operazioni dominanti*.
* Per quanto riguarda i *dati*, si presta attenzione al *size degli array*.

Tutto ciò si può richiudere in *complessità computazionale* di un algoritmo, che ha un *impatto diretto* sulle prestazioni del software che implementa l’algoritmo.

Il numero complessivo delle operazioni da eseguire dipende dal *numero dei dati di input*. Quest’ultimo ci indica la *dimensione computazionale* di un problema.

Maggiore è la dimensione computazionale, maggiori saranno tempo e spazio richiesti dall’algoritmo.

## Teoria della complessità computazionale

La *teoria della complessità computazionale* investiga i problemi legati alla *quantità di risorse* richieste per l’esecuzione degli algoritmi e la *difficoltà intrinseca* di fornire algoritmi efficienti per risolvere specifici problemi. Quindi tale teoria investiga su:

* *Complessità di tempo* di un algoritmo.
* *Complessità di spazio* di un algoritmo.
* *Difficoltà intrinseca* di un algoritmo.

Individuate complessità di tempo e spazio è possibile *classificare* gli algoritmi in *classi di complessità*.

### Complessità di tempo

La *complessità di tempo* di un algoritmo è la funzione che esprime il *numero di operazioni* *dominanti* in dipendenza della *dimensione computazionale*  del problema.

Per individuare la complessità di tempo di un algoritmo, bisogna:

* Individuare la *dimensione computazionale* di un problema.
* Individuare le *operazioni dominanti* di un problema.

### Complessità di tempo lineare

La complessità di tempo *lineare* si ha quando la complessità di tempo *dipende da .*

### Complessità di tempo quadratica

La complessità di tempo *quadratica* si ha con i *cicli* for *annidati*. La formula è:

Dove corrisponde alla *formula di Gauss bambino*.

### Complessità di tempo cubica

La complessità di tempo *cubica* si ha con *tre cicli* for *annidati*. La formula è:

### Complessità di tempo logaritmica

La complessità di tempo *logaritmica* è migliore di quella lineare e si ha con i *cicli* do-while. La formula è:

### Classifica di complessità

È possibile stilare una classifica delle complessità, che va dal meno al più complesso:

1. Complessità di tempo logaritmica.
2. Complessità di tempo lineare.
3. Complessità di tempo quadratica.
4. Complessità di tempo cubica.

### Complessità di spazio

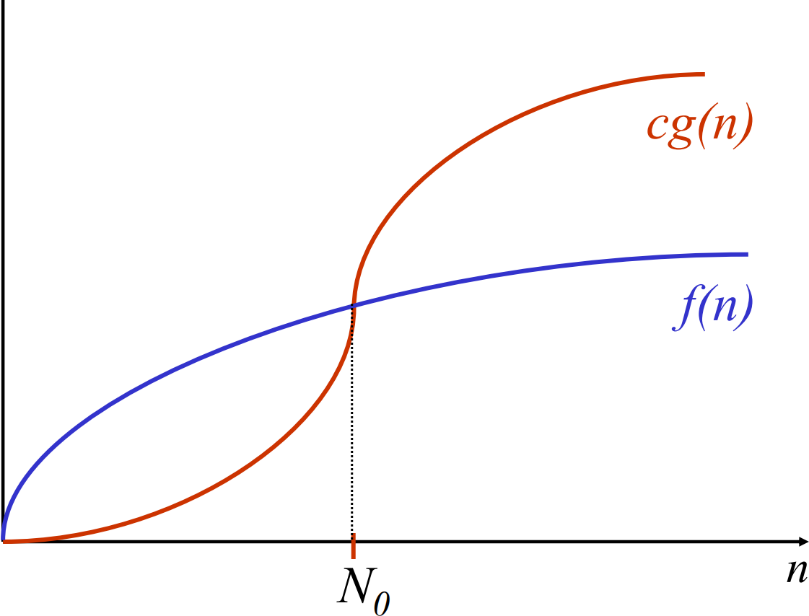
La *complessità di spazio*  di un algoritmo è la funzione che esprime il *size totale* delle *strutture dati* utilizzate per memorizzare dati di input, locali e di output, in dipendenza della *dimensione computazionale* di un problema.

Per individuare la complessità di spazio bisogna, quindi, individuare la *dimensione computazionale*  del problema.

### Complessità asintotica

Il termine *complessità asintotica* si riferisce al comportamento *asintotico*, cioè al tendere di all’*infinito*, delle funzioni e . Si noti che e , escludendo il caso di algoritmi che implementano una formula chiusa e la cui complessità è quindi costante, sono funzioni che tendono a per che tende a .

La notazione asintotica dell’ “O” (che si legge “O grande”) è formalmente definita nel seguente modo: “siano e due funzioni definite (dominio) sui numeri naturali e a valori (codominio) sui numeri naturali; e sono funzioni *non decrescenti*, cioè sono costanti oppure sono crescenti: allora significa che esistono due *costanti* (cioè non dipendenti da ) e , *non negative*, tali che .



In generale, se è un polinomio di grado , allora .

Da considerare: dato che è *limitata*.

### Ridefinizione delle classi di complessità

costante

logaritmica

lineare

lineare-logaritmica

quadratica

polinomiale di grado

esponenziale

Fattoriale

## Linguaggio C

Il *linguaggio C* è un linguaggio di programmazione *standardizzato*, riconosciuto a livello internazionale. Esso ha influenzato lo sviluppo di altri linguaggi di programmazione come Java, C#, Perl, PHP, Javascript.

### Processo di compilazione

Il *processo di compilazione* di un programma in C è:

* *Programma sorgente*: ha estensione “*.*c”. Corrisponde al programma che viene scritto. Viene dato in input al *compilatore.*
* *Programma compilatore*: corrisponde al programma in esecuzione. Viene dato in input al *linker*.
* *Programma oggetto*: ha estensione “*.o*”. Corrisponde al programma che deve essere sottoposto alla fase di *linking*, dove le funzioni presenti in altri file, vengono collegate a questo programma. Da qui si ottiene il *programma eseguibile*, che ha estensione “*.exe*”.

### Compilatore

Il *compilatore* controlla la sintassi del programma sorgente; se corretto, restituisce un programma oggetto.

Esso traduce da linguaggio ad alto livello (linguaggio C) in *linguaggio assembler* (sequenza di 0 e 1).

La parte di programma posta prima della funzione main() viene definita *precompilatore*.

La parte di programma posta dopo la funzione main() viene definita *compilatore*.

### Libreria <stdio.h>

La *libreria* <stdio.h> contiene i prototipi delle funzioni input/output, come la funzione printf(<stringa>) che stampa a video la stringa contenuta tra le parentesi.

## Tipi

Il *tipo* di un dato corrisponde a:

* Un *insieme di valori*.
* Un *insieme di operazioni* consentiti sui valori.
* Un *criterio di rappresentazione* in memoria, ovvero la *rappresentazione interna*.

I *tipi* presenti in C sono:

* *Intero*, che corrisponde alla rappresentazione dei numeri naturali; in C una variabile di tipo intero si definisce con int <variabile>. Tale tipo ha una grandezza di *32 bit*, ovvero *4 byte*; tuttavia, esistono tipi derivati dall’int, come:
  + short int, che viene rappresentato con.
  + long int, che viene rappresentato con in numero massimo di bit disponibili.
  + unsigned short int, che rappresenta solo numeri positivi e viene rappresentato con *16 bit*.
  + unsigned int, che rappresenta solo numeri positivi e viene rappresentato con *32 bit*.
  + unsigned long int, che rappresenta solo numeri positivi e viene rappresentato con il numero massimo di bit disponibili.
* *Carattere*, che corrisponde alla rappresentazione di un carattere (esempio “c”); in C una variabile di tipo carattere si definisce con char <variabile> e ha una grandezza di *8 bit*, ovvero *1 byte*. Per poter definire una stringa si usa un *array di caratteri*, che si definisce con char \*<variabile>.
* *Numero reale*, che corrisponde alla rappresentazione dei numeri reali; in C un numero reale si definisce con:
  + float <variabile>, a singola precisione, quindi con *8 cifre significative*; ha una grandezza di *32 bit*, ovvero *4 byte*.
  + double <variabile> a doppia precisione, quindi con *16 cifre significative*; ha una grandezza di *64 bit*, ovvero *8 byte*. Con questo tipo si ha anche il long double <variabile>, che corrisponde al numero più grande rappresentabile.

Definire un dato di un certo tipo significa definire quante celle di memoria andrà ad occupare quel dato.

Inoltre, in C è possibile definire nuovi tipi di dati, chiamati *user-defined*. Per fare ciò, generalmente, si usa la keyword typedef:

typedef enum{<valore 1>, <valore 2>, …, <valore n>} <nome tipo>;

### Funzione sizeof(<tipo>)

La *funzione* sizeof(<tipo>) restituisce il numero necessario di *byte* necessari per rappresentare il tipo che viene fornito in input alla funzione.

## Variabile

Una *variabile* è un nome a cui si può associare un dato, definito *valore* di una variabile. Essa occupa una cella di memoria che può cambiare nel tempo.

### Variabile indeterminata

Una *variabile indeterminata* corrisponde ad una variabile non dichiarata.

### Variabile definita

Una *variabile definita* corrisponde ad una variabile a cui è associato un valore del suo tipo.

### Variabile indefinita

Una *variabile indefinita* corrisponde ad una variabile dichiarata, ma non definita.

### Variabili scalari

Le *variabili scalari* corrispondono a quelle variabili a cui è possibile associare un solo valore

### Costante

Una *costante* è una variabile che occupa una cella di memoria, ma non cambia nel tempo. Viene così dichiarata:

const <tipo> <variabile>=<valore o espressione>;

### Azioni sulle variabili

Le *azioni* possibili su una variabile sono:

* *accesso* alla variabile, ovvero sapere il valore della variabile.
* *Cambiamento* del valore della variabile, ovvero sostituire il valore che contiene la variabile con un nuovo valore; quest’ultimo sovrascriverà il vecchio valore.
* *Visualizzazione* del valore della variabile, ovvero stampare a video il valore contenuto nella variabile.

### Associazione di un valore ad una variabile

Le operazioni che permettono di *associare* un valore ad una variabile sono:

* *Assegnazione*, che deve specificare:
  + *Variabile di destinazione*, ovvero la variabile a cui deve essere assegnato il valore.
  + *Valore* o *espressione*, ovvero ciò che deve essere assegnato alla variabile.

Un’*espressione* è una variabile di dati e di variabili connessi da operatori e deve:

* + - Coinvolgere variabili, dati e operatori rispettando la *congruenza* di tipo.
    - Non deve contenere variabili indeterminate o indefinite.
    - Deve restituire un *unico* valore.

L’operazione di assegnazione è così effettuata:

<variabile>=<valore>;

<variabile>=<espressione>;

* *Lettura da un dispositivo di input*, dove l’istruzione che gestisce questo tipo di assegnazione deve specificare la variabile di destinazione a cui deve essere assegnato il valore.

### Istruzione di scrittura

L’istruzione di *scrittura* su dispositivo esterno permette l’invio del valore associato alla variabile su un dispositivo di output (esempio: schermo).

Tale istruzione deve specificare la variabile da far visualizzare sul dispositivo esterno.

In C, la funzione di visualizzazione è printf(<variabile>);

## Operatori

Gli *operatori* in C sono:

* *+*, per l’addizione.
* *\**, per il prodotto.
* *-*, per la sottrazione.
* */*, per la divisione.
* *%*, per il modulo. Tale operazione effettua una divisione dando come risultato solamente il resto.
* *Incremento*:
  + i++
  + i=i+1
  + ++i
* *Decremento*:
  + i--
  + i=i-1
  + –i

## Costrutti di selezione

Un *costrutto di selezione* denota la scelta, ovvero la *selezione*, tra due insiemi di istruzioni, in dipendenza di una *condizione*, ovvero un *predicato*.

Un costrutto di selezione a due vie deve specificare:

* Una *condizione*, che può assumere valore *vero* (*true*) o falso (*false*).
* Due *insiemi di istruzioni*.

### Costrutto if-then-else

*Sintassi* del costrutto:

if (<predicato>) {

<corpo then>;

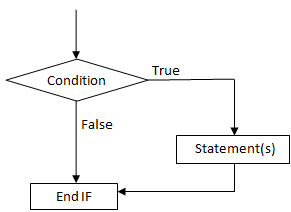
} else {

<corpo else>;

}

<corpo then> e <corpo else> sono due *blocchi di istruzioni*.

Il *diagramma* di questo costrutto è:



Quando questo costrutto ha più di due viene definito costrutto di selezione *nidificato*.

### Costrutto di selezione a vie

Per definire un costrutto di selezione a *vie* si usa uno *switch*.

*Sintassi*:

switch(<variabile>) {

case 1:

<corpo caso 1>;

break;

case 2:

<corpo caso 2>;

break;

case n:

<corpo caso n>;

break;

default:

<corpo default>;

break;

}

Se <variabile> ha un valore di un qualsiasi altro caso non specificato, vengono eseguite le istruzioni del corpo di default.

## Costrutti di ripetizione (o iterazione)

Un costrutto di *ripetizione* (o *iterazione*) denota la *ripetizione* di un insieme di istruzioni.

I costrutti di ripetizione si distinguono in:

* Costrutti in cui il numero di ripetizioni è *noto a priori*. Un costrutto che si usa in questo caso è il for.
* Costrutti in cui il numero di ripetizioni non è noto, ma dipende da una *condizione*. Costrutti che si usano in questo caso sono while e do-while.

### Ciclo for

Il ciclo for consente di denotare la ripetizione volte di una sequenza di operazioni.

*Sintassi* del ciclo:

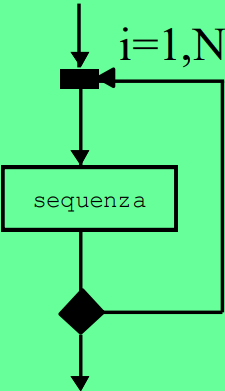
for(i=<valore>; i<n; i++) {

<corpo ciclo>;

}

* i si definisce variabile di ciclo.
* i<n definisce il *range* della variabile di ciclo.
* i++ definisce l’incremento della variabile di ciclo.

Il *diagramma* di questo ciclo è:



### Ciclo do-while

Il ciclo do-while è un costrutto di ripetizione che denota la ripetizione del valore di un predicato. Se quest’ultimo non cambia, si va incontro ad un *loop*.

*Sintassi* del ciclo:

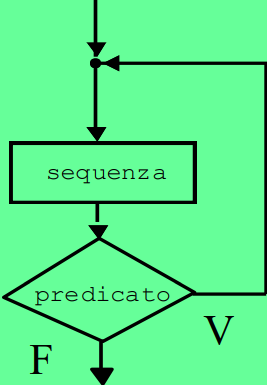
do {

<corpo del ciclo>;

}while(<predicato>);

il <corpo del ciclo> viene eseguito sicuramente una volta, mentre per essere eseguito nuovamente il <predicato> deve essere soddisfatto.

Il *diagramma* di questo ciclo è:



### Ciclo while

Il ciclo while è analogo al do-while, solo che il <corpo del ciclo> non viene *mai eseguito* se il predicato non è soddisfatto.

Per poter evitare un *loop*, almeno un valore delle variabili che compaiono nel predicato deve essere modificato da un’istruzione del <corpo del ciclo>.

*Sintassi* del ciclo:

while(<predicato) {

<corpo del ciclo>;

}

### Ciclo innestato

Un ciclo che si trova all’interno di un ciclo viene definito ciclo *innestato*.

## Puntatori

La memoria è divisa in *celle*, ognuna identificata attraverso un *indirizzo*.

L’*operatore indirizzo*, che si indica con &, viene usato per conoscere l’indirizzo di una variabile:

&<variabile>;

In C, è possibile assegnare l’indirizzo di una variabile ad una variabile *puntatore*, che consiste in una variabile che contiene l’indirizzo di un’altra variabile. Si dichiara così:

<tipo> \*<puntatore>;

L’operatore \* viene definito operatore di *deferenziazione* e indica il *valore memorizzato* nella cella di quell’indirizzo.

### Doppio puntatore

Con il *doppio puntatore* si va a definire una variabile che contiene *l’indirizzo di un puntatore*.

La dichiarazione è: char \*\*<nome variabile>.

Logicamente, in \*<nome variabile> sarà contenuto un *indirizzo*, mentre in \*\*<nome variabile> sarà contenuto un dato.

## Function

Una *function* prende in ingresso più input e fornisce un solo output.

Il suo algoritmo deve essere organizzato in modo da poter essere usato da altri algoritmi.

Per una function non è necessario conoscerne l’algoritmo che ne definisce il comportamento, bensì basta conoscerne i dati di input che essa richiede.

### Intestazione

L’*intestazione*, definito anche *prototipo* di una function deve specificare:

* *nome*;
* *parametri*, ovvero le variabili di input;
* *tipo* dei dati di input, ovvero il tipo dei parametri;
* *tipo*, che indica il tipo della variabile di output.

Il valore di output viene ritornato attraverso keyword return:

return <espressione>;

### Chiamata

L’operazione di *chiamata* di una function permette l’esecuzione di questa.

Lo scambio di informazioni tra *programma chiamante* e *function chiamata* avviene attraverso la corrispondenza tra *argomento* dichiamata, *parametro* della function e *risultato* restituito nell’espressione di chiamata. Tale corrispondenza è stabilita dalla *posizione* dei parametri nell’*intestazione* e dalla *posizione* degli argomenti nella *chiamata*.

### Argomento

Un *argomento* è una variabile che appare nella *chiamata* di una function.

### Parametro

Un *parametro* è una variabile che appare nell’*intestazione* di una function.

La lista di variabile che appaiono nell’intestazione di una function corrisponde alla *lista di parametri* dei dati di input della function.

### Sintassi dell’intestazione

La *sintassi* dell’intestazione di una function è:

<tipo> <nome function>(<lista parametri>);

### Parte dichiarativa

Anche nella function c’è una parte *dichiarativa*, dove vengono dichiarate le *variabili locali*, che hanno valore solamente nella function.

### Sintassi di una function

La *sintassi* di una function è:

<tipo> <nome function>(<tipo> <parametro 1>, <tipo> <parametro n>) {

<parte dichiarativa>; (dove vengono dichiarate le variabili locali)

<parte esecutiva>;

}

## Procedure

A differenza delle function, le *procedure* ammettono da a output.

### Intestazione

Nell’*intestazione* una procedura deve specificare:

* *nome* della procedura;
* *parametri* di input;
* *parametri* di output;
* *parametri* di input/output (i/o), ovvero parametri che sono sia di input che di output.

### Sintassi dell’intestazione

La *sintassi* dell’intestazione di una procedura è:

void <nome procedura>(<input 1>, <input n>, \*<i/o 1>, \*<i/o n>);

Da notare che nell’intestazione, i parametri di input/output di una procedura devono essere puntatori.

### Chiamata

La *chiamata* di una procedura si effettua nel modo seguente:

void <nome procedura>(<argomento 1>, <argomento n>, &<output 1>, <&output n>);

Da notare che nella chiamata, gli argomenti di input/output di una procedura devono essere indirizzi.

## Approccio incrementale

L’*approccio incrementale* permette di costruire la soluzione di un problema attraverso *incrementi* successivi della difficoltà dell’istanza di un problema.

Prendiamo come esempio la somma di numeri naturali, che in modo matematico si definisce:

Dove:

* corrisponde all’unico dato di input;
* corrisponde al dato di output.

L’*algoritmo*, usando l’approccio incrementale, è:

int somma(int n) {  
 int i, somma=0;

for(i=1; i<=n; i++)

somma=somma+i;

return somma;

}

Tuttavia, per risolvere questo problema, possiamo usare la *formula di Gauss bambino*:

## Organizzazione e relazione tra i dati

È possibile indicare con un unico nome un *insieme di dati*. In questo insieme i dati sono *organizzati*.

Dall’*organizzazione dei dati* viene definita una *struttura*, posseduta da *dati aggregati*.

Data una struttura, quindi, esiste una *relazione dei dati*. Varie forme di relazione sono:

* *Fila*: la posizione dei dati dipende dal tempo in cui il dato è entrato nella fila.
* *Albero*: la posizione dei dati dipende dalla loro *relazione gerarchica*.
* *Tabella*: formata da *caselle*, ogni dato occupa una cella. Tutti i dati nella tabella sono tutti dello stesso tipo. Ogni dato è individuabile attraverso un *indice*, che indica la *posizione* del dato all’interno della tabella.

Esistono tabelle formate da:

* + un’unica riga:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 34 | 51 | 87 | 19 | 26 | 14 | 74 |

* + *n* righe e *m* colonne: tali tabelle vengono definite tabelle a due dimensioni. Qui ogni elemento è individuato da due indici:
    - *n* indice di riga.
    - *m* indice di colonna.

## Array

Nei linguaggi di programmazione, l’insieme di dati organizzati a tabella prende il nome di *array*: è un tipo strutturato che può avere *n* dimensioni (array *nD*).

*Proprietà* degli array sono:

* gli elementi sono tutti dello *stesso tipo*.
* Il numero massimo di elementi, definito *size*, di un array è fissato al momento della dichiarazione e non può essere modificato.
* L’*intervallo* degli indici va da .

### Dichiarazione di un array

La *dichiarazione* di un array deve specificare:

* *Identificatore* della variabile di tipo array.
* *Size* dell’array.
* *Tipo* degli elementi dell’array.

La dichiarazione di un array a una dimensione è:

<tipo> <nome array>[<size>];

A due dimensioni è invece:

<tipo> <nome array>[<size righe>][<size colonne>];

a due dimensioni, la size massima è data dal prodotto delle due size:

### Inizializzazione array 2D

Per inizializzare un array a due dimensioni si usa la seguente sintassi:

int array[righe][colonne]={{<valori prima riga>},

{<valori n riga>}};

Stessa cosa vale per gli array a n dimensioni.

### Accesso agli elementi di un array

Per *accedere* agli elementi di un array bisogna indicare:

* *Identificatore* dell’array.
* *Indice/i* dell’elemento a cui vogliamo accedere.

Quindi, ad una dimensione, l’accesso è:

<nome array>[<indice>];

A due dimensioni, invece, è:

<nome array>[<indice riga>][<indice colonna>];

### Mappa di memorizzazione

Un array 2D può essere denotato:

* Con la *notazione standard*, quindi indicando i due indici.
* Con la *notazione a puntatore*, ovvero dereferenziando la *mappa di memorizzazione*:

avendo una matrice dichiarata:

int matrice[righe][colonne];

la sua mappa di memorizzazione sarà:

\*(matrice+colonne\*i+j);

## Algoritmo di fusione di due array – Merge

L’*algoritmo di fusione* risolve il cosiddetto problema del *merging*, ovvero la fusione di due array *ordinati* di dati (dello stesso tipo) in un terzo array *ordinato* che ha come elementi gli elementi dei due array, e quindi ha un size uguale alla somma dei size dei due array.

L’algoritmo di fusione ha come dati di input due array ordinati, che chiamiamo a e b, e i loro size che chiamiamo rispettivamente sizeA e sizeB, e ha come dato di output il terzo array che chiamiamo c, il cui size è noto ed è .

L’algoritmo di fusione è basato sull’idea di effettuare un numero di passi uguale al size del terzo array c, cioè passi; a ogni passo, si determina un elemento dell’array c. Detto countC l’indice per accedere agli elementi dell’array c, al generico passo countC lo scopo è determinare l’elemento di c di indice countC.

Per costruire l’elemento c[countC], chiamiamo countA e countB i due indici per accedere rispettivamente agli elementi dell’array a e agli elementi dell’array b. Al primo passo countA, countB, countC indicano il primo elemento (dei rispettivi array).

Al passo countC, ci sono tre possibili scenari:

1. il primo è quello per così dire “normale”, cioè c’è almeno un elemento di a e un elemento di b da considerare, ovvero non ancora inseriti nell’array c; gli indici countA e countB indicano l’elemento di a e l’elemento di b che devono essere considerati al fine di determinare quale deve essere inserito in c; è facile convincersi che il più piccolo tra a[countA] e b[countB] deve essere inserito in c, nella posizione countC (nel caso gli elementi a[countA] e b[countB] fossero uguali, si inserirebbe uno qualunque dei due); L’indice dell’array il cui elemento viene inserito in c deve essere incrementato.
2. Il secondo scenario è quello in cui tutti gli elementi dell’array a sono stati già inseriti in c e rimangono da considerare solo gli elementi dell’array b; in tal caso, l’elemento b[countB] deve essere inserito in c, nella posizione countC, e l’indice countB deve essere incrementato.
3. Il terzo scenario è quello in cui tutti gli elementi dell’array b sono stati già inseriti in c e rimangono da considerare solo gli elementi dell’array a; in tal caso, l’elemento a[countA] deve essere inserito in c, nella posizione countC, e l’indice countA deve essere incrementato.

Si noti che, per una data istanza del problema, solo uno tra lo scenario 2 e lo scenario 3 può verificarsi.

Questa è l’implementazione in C dell’algoritmo di fusione, per due array di tipo char:

void fusione\_array(char a[], int sizeA, char b[], int sizeB, char c[]) {

int countA=0, countB=0, countC=0;

while(countA<sizeA && countB<sizeB) {

if(a[countA]<b[countB])

c[countC++]=a[countA++];

else

c[countC++]=b[countB++];

}

while(countA<sizeA)

c[countC++]=a[countA++];

while(countB<sizeB)

c[countC++]=b[countB++];

}

Si definiscono complessità di tempo, spazio e asintotica:

* ***Complessità di tempo***

L’operazione dominante è l’operazione di confronto tra un elemento dell’array a e un elemento dell’array b. L’*algoritmo di fusione* effettua passi, cioè quanti sono gli elementi di c che devono essere determinati. A ogni passo si costruisce un elemento dell’array c. Se si è nello scenario , allora è necessario esattamente un confronto per determinare un elemento di c. Se si è nello scenario o nello scenario , allora non si effettua alcun confronto. Quindi si può concludere che il numero totale di confronti è al più . Si noti che si tratta di una complessità di *caso peggiore*. In conclusione, la complessità di tempo dell’algoritmo di fusione è:

al più confronti, con

* ***Complessità di spazio***
* ***Complessità asintotica***

La complessità asintotica di questo algoritmo è:

## Algoritmo di ricerca sequenziale

L’algoritmo di *ricerca sequenziale* risolve questo problema:

“determinare l’appartenenza di un dato ad un insieme assegnato di dati, forniti da un dispositivo esterno”

Come dati abbiamo:

* *Chiave*, *n*, *n dati* che sono i dati di input;
* *True*, *false* che sono i dati di output.

Di seguito, viene riportato l’algoritmo:

int appartiene(char chiave, char a[], int n) {

int i=0, esito=0;

do {

if(chiave==a[i])

esito=1;

i++;

}while(esito=0 && i<n);

return esito;

}

Il *costo* dell’algoritmo di ricerca sequenziale e *al più*.

## Stringhe

Una *stringa* è una sequenza di caratteri (array di char; dichiarazione: char s[size]). Essa è costituita da un numero finito di caratteri, che ne definisce la lunghezza.

Una stringa vuota viene definita una stringa di lunghezza 0.

Una stringa di lunghezza k viene definita k-stringa e l’insieme delle k-stringhe su un alfabeto di n caratteri ha cardinalità .

Ogni carattere della stringa è identificato attraverso un indice, che ne indica la posizione all’interno della stringa.

### Permutazione di una k-stringa

La permutazione di una k-stringa è una k-stringa composta dagli stessi caratteri. L’insieme delle permutazioni di una k-stringa ha cardinalità k fattoriale .

### Sottostringa

Una sottostringa è una porzione di una stringa che, avendo una lunghezza p con inizio di indice i, viene definita p-sottostringa di inizio i.

### Libreria <string.h>

Per lavorare con le stringhe in C è presente la libreria <string.h>, contenente tutte le funzioni per gestire le stringhe.

## String matching

L’*algoritmo di string matching* risolve il cosiddetto problema del *matching*, ovvero l’individuazione di una certa stringa di caratteri (detta *stringa chiave*) all’interno di un’altra data stringa di caratteri (detta *stringa testo*). Da un punto di vista astratto, il problema è assimilabile a un problema di *ricerca* (*searching*).

L’algoritmo di string matching ha come dati di input due stringhe, cioè la stringa *chiave* e la stringa *testo*, e ha come dato di output il numero delle volte (detto numero delle *occorrenze*) in cui la stringa *chiave* compare come *sottostringa* all’interno della stringa *testo*.

Tale algoritmo è basato sull’idea di effettuare un numero di passi uguale alla lunghezza della stringa *testo* (più precisamente, , per evitare inutili confronti nelle iterazioni finali), risolvendo a ogni passo il problema di determinare l’uguaglianza tra la stringa *chiave* e la *sottostringa* della stringa *testo* che ha come posizione inziale il passo e come lunghezza la lunghezza della stringa *chiave*.

Questa è l’implementazione in C dell’*algoritmo di string matching*:

int string\_matching(char chiave[],char testo []) {

int n=strlen(chiave), m=strlen(testo), i, conta\_chiave=0;

for (i=0; i<=m-n; i++) {

if(strncmp(chiave, testo[i], n)==0)

conta\_chiave++;

}

return conta\_chiave;

}

Il for che va da i=0 a m-n gestisce i passi dell’algoritmo. Lo scopo del passo i è risolvere il problema di determinare l’uguaglianza tra la sottostringa della stringa *chiave* di lunghezza e la sottostringa della stringa *testo* che ha come posizione inziale i e come lunghezza .

Analizziamo adesso la complessità dell’algoritmo di string matching:

* ***Complessità di tempo***

L’operazione dominante è l’operazione di confronto tra un elemento della stringa testo e un elemento della stringa chiave. Tali confronti sono eseguiti all’interno della function strncmp. Poiché la strncmp determina l’uguaglianza di due sottostringhe, cioè di due array, il numero di confronti per ogni chiamata della function strncmp è al più la lunghezza delle sottostringhe in input. La function strncmp è chiamata una sola volta per ogni passo del ciclo for, che effettua passi. Quindi si può concludere che il numero totale di confronti è al più . Si noti che si tratta di una complessità di caso peggiore. In conclusione, la complessità di tempo dell’algoritmo di string matching è:

confronti al più.

* ***Complessità di spazio***

, ovvero i size delle due stringhe (*testo* e *chiave*).

* ***Complessità asintotica***

La complessità asintotica di questo algoritmo è:

confronti al più.

## Best matching

Per risolvere questo tipo di problema necessitiamo di un algoritmo che segni il punteggio del matching:

int punteggio\_matching(char \*a, char \*b, int n) {

int i, caratteri\_uguali=0;

for(i=0; i<n; i++) {

if(a[i]==b[i])

caratteri\_uguali++;

}

return caratteri\_uguali;

}

Adesso, possiamo scrivere l’algoritmo di best matching:

int best\_matching(char \*key, int n, char \*testo, int m) {

int i, punteggio, punteggio\_max=0, indice;

for(i=0; i<=m-n; i++) {

punteggio=punteggio\_matching(key, &testo[i], n);

if(punteggio>punteggio\_max) {

punteggio\_max=punteggio;

indice=i;

}

}

return indice;

}

Il costo dell’algoritmo di best matching è:

## Algoritmi di ordinamento

Il problema dell’ordinamento di un array viene risolto *in place*, ovvero lavorando sullo stesso array. In questo tipo di problemi abbiamo come dati di input l’array e il suo rispettivo size; come output, invece, lo stesso array ordinato.

### Ordinamento per inserimento (insertion sort)

L’*algoritmo di ordimento per inserimento* è basato sull’applicazione dell’approccio incrementale al problema dell’ordinamento. Infatti, a ogni passo l’algoritmo risolve una istanza del problema dell’ordinamento, cioè l’ordinamento di una “porzione” dell’array A; all’aumentare del passo aumenta la dimensione dell’istanza del problema, fino ad arrivare alla dimensione del problema dato. Diamo una rapida analisi della meccanica dell’algoritmo di ordinamento per inserimento. Al generico passo i, essendo già stata ordinata nei passi precedenti la porzione dell’array, l’algoritmo di ordinamento per inserimento ordinerà la porzione dell’array A. È importante osservare che tale ordinamento non è quello definitivo e che nei successivi passi tale ordinamento può essere modificato dall’inserimento di altri tra i rimanenti elementi dell’array. L’idea di base è che se i primi elementi sono ordinati, allora per ordinare i primi i elementi basta inserire l’i-esimo elemento nella sua “giusta” posizione, cioè nella posizione in cui tale elemento avrà alla sua sinistra gli elementi minori o uguali e alla sua destra gli elementi uguali o maggiori. Quindi, a ogni passo l’algoritmo di ordinamento per inserimento risolve il problema di “inserire” (di qui il nome dell’algoritmo) un elemento dell’array nella porzione già ordinata alla sua sinistra, in modo che la lunghezza della porzione ordinata aumenti di .

In generale, la “giusta” posizione viene determinata con un procedimento iterativo che procede a ritroso, esaminando successivamente le posizioni , , …. ed eventualmente fino alla prima pozione (indice ).

*Algoritmo:*

void insertion\_sort(char a[], int n) {

char el\_da\_ins;

int i, j;

for(i=1; i<n; i++) {

el\_da\_ins=a[i];

j=i-1;

while(j>=0 && el\_da\_ins<a[j]) {

a[j+1]=a[j];

j--;

}

a[j+1]=el\_da\_ins;

}

}

Andiamo ad analizzare le complessità di questo algoritmo:

* ***Complessità di tempo***

Si considerano come operazioni dominanti i confronti e gli scambi.

L’*operazione di confronto* appare nel predicato del while. Il ciclo while viene attivato volte, poiché il while è interno al ciclo for. Il numero di iterazioni di ogni attivazione del while non è costante, perché dipende sia dalla lunghezza della porzione dove si deve trovare la giusta posizione dell’elemento da inserire, sia dal valore degli elementi di tale porzione. Dobbiamo quindi fare un’analisi di complessità di caso peggiore. Il caso peggiore, cioè il massimo numero possibile di confronti, si ha quando si esce dal ciclo while sempre perché , ovvero quando la giusta posizione è sempre la prima posizione dell’array. Per inciso, questo accade quando l’array in input è ordinato nel senso decrescente. Quindi, al primo passo, , il massimo numero possibile di confronti nel while è ; al secondo passo, , il massimo numero possibile di confronti nel while è ; e così via, fino all’ultimo passo, , in cui il massimo numero possibile di confronti è . Il numero totale di confronti, quindi, si ottiene effettuando la somma degli numeri naturali; pertanto, può essere utilizzata la formula di *Gauss bambino*, ovvero . Per quanto concerne il numero di scambi, o meglio il numero di shift a destra, basta osservare che si ha uno spostamento a destra ogni volta che è vero il predicato del while. Pertanto, nel caso peggiore, il numero di shift a destra è uguale al numero dei confronti.

Pertanto, la complessità di tempo è:

confronti/scambi, al più.

* ***Complessità di spazio***

, ovvero il size dell’array

* ***Complessità asintotica***

La complessità asintotica di questo algoritmo è:

, ovvero quadratica

### Ordinamento per selezione di minimo (selection sort di minimo)

L’*algoritmo di ordinamento per selezione di minimo* ha come dati di input un array e il suo size e ha come dato di output lo stesso array con i dati ordinati.

L’algoritmo effettua un numero di passi (iterazioni) pari al . Cioè, detto il size dell’array che chiamiamo A, l’algoritmo effettua passi per ordinare l’array A. Ad ogni passo l’algoritmo risolve il sotto problema della ricerca dell’elemento minimo (e del suo indice) di una opportuna “porzione” dell’array A. Una volta trovato l’elemento minimo e la sua posizione, tale elemento viene scambiato con l’elemento che si trova al primo posto della porzione che si sta considerando. Al passo successivo, si considera una nuova porzione dell’array, che avrà un numero di elementi diminuito di una unità rispetto al numero di elementi della porzione considerata precedentemente.

Si noti che il sotto problema della determinazione del minimo può essere risolto mediante una opportuna chiamata della function min\_val\_ind, e che l’operazione di scambio può essere effettuata dalla function scambiare:

void scambiare(char \*a, char \*b) {

int temp=\*a;

\*a=\*b;

\*b=temp;

}

void min\_val\_ind(char a[], int n, char \*min\_array, int \*index) {

int i;

\*index=0;

\*min\_array=a[0];

for(i=1; i<n; i++) {

if(a[i]<\*min\_array) {

\*min\_array=a[i];

\*index=i;

}

}

}

Quindi, l’*algoritmo di ordinamento per selezione di minimo* sarà:

void selection\_sort\_min(char a[], int n) {

int index, i;

char min\_array;

for(i=0; i<n-1; i++) {

min\_val\_ind(&a[i], n-i, &min\_array, &index);

scambiare(&a[i], &a[index+i]);

}

}

È importante notare che il valore di index restituito dalla function min\_val\_ind è il cosiddetto “ valore locale” dell’indice, cioè si riferisce alla porzione che si sta considerando e non all’intero array. Per ricostruire il cosiddetto “valore globale” dell’indice, cioè per determinare la corretta posizione all’interno dell’array completo, è necessario effettuare l’operazione detta di “spiazzamento dell’indice locale” o “globalizzazione dell’indice locale”, che consiste semplicemente nell’aggiungere al valore dell’indice locale il valore dell’indice di inizio della porzione, cioè i. In altre parole, l’indice globale dell’elemento minimo della porzione è index+i.

Andiamo ad analizzare le complessità di questo algoritmo:

* ***Complessità di tempo***

Cominciamo, per semplicità, col determinare il numero di operazioni di scambio. In pratica, si tratta di contare il numero di chiamate della function scambiare. L’attivazione della scambiare viene fatta a ogni iterazione del ciclo for, che va da , cioè per un totale di iterazioni. Quindi l’algoritmo di ordinamento per selezione di minimo effettua sempre scambi; è una complessità cosiddetta assoluta, perché indipendente dal valore degli elementi dell’array. Determiniamo ora il numero delle operazioni di confronto tra due elementi dell’array. Poiché né nel testo della function di ordinamento né nella function scambiare compaiono operazioni di confronto tra due elementi dell’array, allora i soli confronti sono quelli contenuti nella function min\_val\_ind. Ricordiamo che la complessità di tempo dell’algoritmo di determinazione del minimo di un array è data dal operazioni di confronto. Tale complessità è assoluta, cioè indipendente dal valore dei dati. Nel nostro caso la function min\_val\_ind è chiamata volte, una per ogni iterazione del ciclo for, ma il size della porzione su cui agisce la function non è costante, perché diminuisce di a ogni iterazione. Alla prima iterazione, la porzione passata alla function min\_val\_ind coincide con l’intero array e ha lunghezza ; quindi, la function effettua confronti. Alla seconda iterazione, la porzione passata alla function min\_val\_ind ha una lunghezza diminuita di uno rispetto alla precedente, cioè ha lunghezza; quindi la min\_val\_ind effettua confronti. E così via, fino all’ultimo passo, in cui alla min\_val\_ind viene passata una porzione di lunghezza , per cui il numero di confronti è .

Per ottenere il numero di confronti totali bisogna eseguire la somma dei primi numeri naturali, che può essere effettuata con la formula di *Gauss bambino*: . Si parla di *complessità assoluta*.

In conclusione, la *complessità di tempo* dell’algoritmo di ordinamento per selezione di minimo è:

confronti e scambi

* ***Complessità di spazio***

, ovvero il size dell’array

* ***Complessità asintotica***

La complessità di questo algoritmo è:

scambi e confronti

Con questo algoritmo si parla di complessità assoluta, ovvero se in input abbiamo un array già ordinato, il costo resta invariato.

### Ordinamento per selezione di massimo (selection sort di massimo)

L’*algoritmo di ordinamento per selezione di massimo* ha come dati di input un array e il suo size e ha come dato di output lo stesso array con i dati ordinati. Tale algoritmo non fa uso di altri array o strutture dati e per questo motivo si dice che opera “in place”.

L’algoritmo effettua un numero di passi (iterazioni) pari a . Cioè, detto il size dell’array che chiamiamo A, l’algoritmo effettua passi per ordinare l’array A. Ad ogni passo l’algoritmo risolve il sotto problema della ricerca dell’elemento massimo (e del suo indice) di una opportuna “porzione” dell’array A. Una volta trovato l’elemento massimo e la sua posizione, tale elemento viene scambiato con l’elemento che si trova all’ultimo posto della porzione che si sta considerando. Al passo successivo, si considera una nuova porzione dell’array, che avrà un numero di elementi diminuito di una unità rispetto al numero di elementi della porzione considerata precedentemente.

Il sotto problema della determinazione del massimo può essere risolto mediante una opportuna chiamata della function max\_val\_ind, e l’operazione di scambio può essere effettuata con la function scambiare.

L’implementazione dell’*algoritmo* è:

void scambiare(char \*a, char \*b) {

char temp=\*a;

\*a=\*b;

\*b=temp;

}

void max\_val\_ind(char a[], int n, char \*max\_array, int \*index) {

int i;

\*index=0;

\*max\_array=a[0];

for(i=1; i<n; i++) {

if(a[i]>\*max\_array) {

\*max\_array=a[i];

\*index=i;

}

}

}

void selection\_sort\_max(char a[], int n) {

int i, index;

char max\_array;

for(i=n-1; i>0; i--) {

max\_val\_ind(&a[0], i+1, &max\_array, &index);

scambiare(&a[i], &a[index]);

}

}

Si noti che il valore di index restituito dalla function max\_val\_ind è il cosiddetto *valore locale* dell’indice, cioè si riferisce alla porzione che si sta considerando , ma coincide con il cosiddetto *valore globale* dell’indice, cioè quello che determina la corretta posizione all’interno dell’array completo, e quindi non si deve effettuare alcuna operazione di *spiazzamento dell’indice locale*.

Andiamo ad analizzare le complessità di questo algoritmo:

* ***Complessità di tempo***

Cominciamo, per semplicità, col determinare il numero di operazioni di scambio. In pratica, si tratta di contare il numero di chiamate della function scambiare. L’attivazione della scambiare viene fatta a ogni iterazione del ciclo for, che va da a , cioè per un totale di iterazioni. Quindi tale algoritmo effettua sempre scambi; è una complessità cosiddetta *assoluta*, perché indipendente dal valore degli elementi dell’array.

Determiniamo ora il numero delle operazioni di confronto tra due elementi dell’array. I soli confronti sono quelli contenuti nella function max\_val\_ind. Ricordiamo che la complessità di tempo dell’algoritmo di determinazione del massimo di un array è data da operazioni di confronto. Nel nostro caso la function max\_val\_ind è chiamata volte, una per ogni iterazione del ciclo for, ma il size della porzione su cui agisce la function non è costante, perché diminuisce di a ogni iterazione. Alla prima iterazione, la porzione passata alla function max\_val\_ind coincide con l’intero array e ha lunghezza ; quindi, la function effettua confronti. Alla seconda iterazione, la porzione passata alla function max\_val\_ind ha una lunghezza diminuita di rispetto alla precedente, cioè ha lunghezza ; quindi la max\_val\_ind effettua confronti. E così via, fino all’ultimo passo, in cui alla max\_val\_ind viene passata una porzione di lunghezza , per cui il numero di confronti è .

Per ottenere il numero di confronti totale bisogna effettuare la somma dei primi numeri naturali, che può essere effettuata con la formula di *Gauss bambino*: . Si parla di *complessità* *assoluta*.

In conclusione, la *complessità di tempo* dell’algoritmo di ordinamento per selezione di massimo è:

confronti e scambi.

Poiché la complessità di tempo dell’algoritmo di ordinamento per selezione di massimo non dipende dal valore dei dati, tale algoritmo assume lo stesso costo anche nel caso in cui l’array sia già ordinato.

* ***Complessità di spazio***

, ovvero il size dell’array

* ***Complessità asintotica***

La complessità asintotica di questo algoritmo è:

scambi e confronti

## Divide et impera

Viene definito *divide et impera* perché un problema si divide in istanze più semplici fino a quando non si arriva ad un’istanza che ha una *soluzione banale*.

### Bottom up

Il procedimento risolutivo *bottom-up* consiste di risolvere un problema dal basso verso l’alto.

### Top down

Il procedimento risolutivo *top-down* consiste di risolvere un problema dall’alto verso il basso.

### Ricerca binaria

L’algoritmo di *ricerca binaria* è un algoritmo di ricerca di una chiave in un array ordinato di dati. I dati possono essere numeri, caratteri, stringhe di caratteri o qualsiasi altro insieme di dati su cui sia definita una relazione di ordine. L’algoritmo di ricerca binaria è un algoritmo ottimale per tale problema e la sua complessità di tempo asintotica è logaritmica.

L’algoritmo di ricerca binaria ha come dati di input la *chiave* di ricerca, un array e il suo size, e ha come dato di output un dato scalare intero che è la *posizione* della prima occorrenza della chiave nell’array oppure il valore convenzionale nel caso in cui la chiave non appartenga all’array. Tale algoritmo opera *in place*, ovvero non fa uso di altre strutture dati o di altri array.

L’algoritmo di ricerca binaria è basato sull’approccio *divide et impera*, che risulta particolarmente efficiente in quanto consente a ogni passo non solo di suddividere una data istanza in due istanze di dimensioni dimezzate, ma soprattutto di eliminare una delle due istanze continuando la ricerca solo su una delle due porzioni dimezzate. Infatti, a ogni passo l’algoritmo risolve una istanza del problema della ricerca in un array ordinato, cioè su una “porzione” dell’array A, e a ogni passo la dimensione della “porzione” si dimezza, fino ad arrivare eventualmente alla dimensione , che è la dimensione dell’*istanza banale* del problema della ricerca.

Diamo una rapida analisi della meccanica dell’algoritmo di ricerca binaria. Si indicheranno con gli indici primo e ultimo rispettivamente la *posizione iniziale* e la *posizione finale* della porzione di array su cui si effettua la ricerca. Al primo passo, primo vale e ultimo vale . Al generico passo di iterazione, si deve effettuare la ricerca sulla porzione primo...ultimo dell’array. Le azioni di un passo sono:

* la determinazione della *posizione* *centrale* della porzione (indice mediano);
* il *confronto a tre vie* tra la *chiave* e l’elemento dell’array che si trova nella posizione centrale:
  + se i due valori sono *uguali*, allora l’algoritmo termina restituendo tale posizione;
  + se la chiave è *minore* la ricerca proseguirà al passo successivo sulla semi-porzione di sinistra, cioè la porzione primo..(mediano-1);
  + se la chiave è *maggiore* la ricerca proseguirà al passo successivo sulla semi-porzione di destra, cioè la porzione (mediano+1)..ultimo.

Se la chiave *non appartiene* all’array, il processo iterativo terminerà quando la porzione su cui agire è vuota, ovvero costituita da nessun elemento, situazione questa indicata dal fatto che l’indice primo ha un valore maggiore di quello dell’indice ultimo.

Questa è l’implementazione in C dell’algoritmo di ricerca binaria, per una *chiave* e un array di tipo char:

int ricerca\_binaria(char chiave, char elenco[], int n) {

int mediano, primo=0; ultimo=n-1;

while(primo<=ultimo) {

mediano=(primo+ultimo)/2;

/\*inizio if a tre vie\*/

if(chiave==elenco[mediano])

return mediano;

else if(chiave<elenco[mediano])

ultimo=mediano-1; //considera porzione di sinistra

else

primo=mediano+1; //considera porzione di destra

/\*fine if a tre vie\*/

}

return -1;

}

Il while gestisce i passi dell’algoritmo di ricerca binaria. Il predicato di permanenza nel ciclo primo<=ultimo indica che il processo iterativo continua se la porzione di array su cui effettuare la ricerca è costituita da almeno un elemento. La nuova porzione su cui agire viene denotata modificando il valore di ultimo, se si dovrà operare sulla semi-porzione di sinistra, mentre invece modificando il valore di primo, se si dovrà operare sulla semi-porzione di destra. Si noti che l’uscita normale dal ciclo while significa che la chiave *non è stata trovata*, e quindi in tal caso si restituisce il valore convenzionale -1.

Vediamo ora l’implementazione dell’algoritmo di ricerca binaria con una chiave e un array di tipo *stringhe di* char:

int ricerca\_binaria(char chiave[], char \*elenco[], int n) {

int mediano, primo=0; ultimo=n-1;

while(primo<=ultimo) {

mediano=(primo+ultimo)/2;

/\*inizio if a tre vie\*/

if(strcmp(chiave, elenco[mediano])==0)

return mediano;

else if(strcmp(chiave, elenco[mediano])<0)

ultimo=mediano-1; //considera porzione di sinistra

else

primo=mediano+1; //considera porzione di destra

/\*fine if a tre vie\*/

}

return -1;

}

Si noti che sono cambiate solo le modalità di confronto tra *chiave* ed elemento dell’array. Si ricorda che la function strcmp restituisce:

* se le due stringhe sono uguali;
* un *numero negativo* se la prima stringa *precede* la seconda nell’ordine alfabetico;
* un *numero positivo* se la prima stringa *segue* la seconda nell’ordine alfabetico.

Andiamo ora ad analizzare le complessità dell’algoritmo di ricerca binaria:

* ***Complessità di tempo***

L’operazione dominante è l’operazione di confronto tra la *chiave* e un elemento dell’array. Nel caso dell’algoritmo di ricerca binaria l’operazione dominante è il *confronto a tre vie* tra la *chiave* e un elemento dell’array. L’operazione di confronto a tre vie è l’if-else-if che appare nel while.

La complessità di tempo dipende dal valore dei dati oltre che dalla dimensione computazionale e quindi si deve effettuare un’analisi di complessità di *caso peggiore*. Il modo più semplice per determinare la complessità di tempo dell’algoritmo di ricerca binaria consiste nell’esaminare il cosiddetto *albero binario* delle decisioni associato all’esecuzione dell’algoritmo per un dato problema. I *nodi* di tale albero sono gli elementi dell’array e il *livello* in cui appaiono nell’albero corrisponde al passo dell’algoritmo in cui sono confrontati con la *chiave*. In altre parole, la *radice dell’albero* è l’elemento che è confrontato con la chiave al primo passo dell’algoritmo, i due nodi del livello 1 sono i due elementi che possono essere confrontati con la chiave al secondo passo, ovvero l’elemento in posizione centrale della semi-porzione di sinistra oppure l’elemento in posizione centrale della semi-porzione di destra, e così via.

È chiaro che, poiché l’algoritmo a ogni passo effettua la ricerca solo una delle due semi-porzioni del passo precedente, a ogni livello corrisponde una sola operazione di confronto a tre vie. Quindi si può concludere che il numero totale di confronti a tre vie è uguale alla profondità (cioè il numero di livelli) del corrispondente albero delle decisioni.

L’albero ha esattamente nodi, dove è il *size dell’array*. Se l’albero fosse *completo* (cioè ogni nodo diverso da una foglia ha esattamente due nodi figli) la sua *profondità* sarebbe . È facile convincersi che l’albero delle decisioni dell’algoritmo di ricerca binaria, in generale, è *quasi-completo*, nel senso che solo il livello delle foglie può essere incompleto, e quindi la sua profondità è , dove , detto il *floor di a*, indica il *più grande intero minore o uguale di* . In conclusione, la complessità di tempo dell’algoritmo di ricerca binaria è:

al più.

Si noti che il caso peggiore è quello in cui la *chiave* *non appartiene* all’array, oppure quello in cui la *chiave* è *uguale* a una delle foglie del corrispondente albero delle decisioni.

* ***Complessità di spazio***

, ovvero il size dell’array.

* ***Complessità asintotica***

.

### Algoritmo di somma con raddoppiamento

Si va a definire l’algoritmo di somma con raddoppiamento con approccio divide et impera.

*Algoritmo:*

int somma(int a[], int n) {

int i, k, somma;

for(k=; k>0; k++) {

for(i=1; i<=; i++)

a[i-1]=a[2\*(i-1)]+a[2\*i-1];

}

somma=a[0];

return somma;

}

La complessità di tempo di questo algoritmo è uguale alla complessità di tempo dell’algoritmo di somma di un array: .

Tuttavia, questo algoritmo è *eccessivamente complicato*.

## Ricorsività

### Formula ricorrente

Una *formula ricorrente* consente di descrivere in modo formale alcuni algoritmi basati sull’approccio incrementale e corrisponde ad un’equazione che definisce in modo univoco una *successione*.

### Esempio formula ricorrente

*Problema:*

somma dei primi numeri naturali

*Formula ricorrente:*

### Formula ricorrente del secondo ordine

dipende solo dal termine precedente

*Algoritmo:*

void ricorrente\_1lc(int n, float a, float b, float y\_zero){

float y = y\_zero;

int k;

printf(“%f”, y);

for(k=1; k<=n; k++) {

y=a\*y+b; 🡪 *il nuovo valore di y sarà il vecchio valore di y moltiplicato per a e sommato a b*

printf(“%f”, y);

}

}

Usando questo algoritmo possiamo risolvere questo *problema*:

visualizzare il valore dei primi n termini e del valore assoluto della differenza di ogni termine con il precedente di una formula ricorrente

*Algoritmo****:***

void ricorrente\_1lc(int n, float a, float b, float y\_zero) {

float y\_att, y\_prec;

int k;

y\_prec = y\_zero ;

printf (“%f”, y\_prec);

for (k=1; k<=n; k++) {

y\_att = a\*y\_prec+b;

printf (y\_att,abs(y\_att-y\_prec));

y\_prec = y\_att; 🡪 *questa assegnazione viene definita aggiornamento della memoria*

}

}

### Fibonacci

La successione di Fibonacci è una formula ricorrente del secondo ordine con coefficienti costanti.

Tale formula è con

*Algoritmo di Fibonacci con approccio incrementale:*

int fibonacci(int n) {

int k, fibo, fibo\_prec\_1, fibo\_prec\_2;

if (n==0 || n==1) {

fibo=n;

} else {

fibo\_prec\_1=1;

fibo\_prec\_2=0;

for (k=2; k<=n; k++) {

fibo=fibo\_prec\_1+fibo\_prec\_2;

fibo\_prec\_2=fibo\_prec\_1;

fibo\_prec\_1=fibo;

}

}

return fibo;

}

La complessità di tempo è: somme.

Prendiamo in esempio il problema:

somma di n numeri naturali

e applichiamo Fibonacci.

L’equazione risolutiva sarà:

con

### Algoritmo ricorsivo

Un algoritmo ricorsivo consente di ritrovare la soluzione di un problema richiamando sé stesso.

Se ho un algoritmo ricorsivo esiste un equivalente algoritmo senza approccio ricorsivo.

Per un algoritmo ricorsivo necessitiamo di una struttura a pila, definita stack, che congela tutti i risultati parziali e setta come output l’ultimo risultato salvato.

### Esempio algoritmo ricorsivo

*Problema:*

somma dei primi n numeri naturali

*Formula ricorsiva:*

con

*Algoritmo ricorsivo:*

int somma(int n) {

if(n==1) 🡪 *istanza banale*

return 1;

else

return n + somma(n-1); 🡪 *auto attivazione*

}

### Esempio algoritmo ricorsivo

*Problema:*

fattoriale dei primi n numeri naturali

*Formula ricorsiva:*

con

*Algoritmo ricorsivo:*

int P(int n) {

if(n<=1)

return 1;

else

return n\*P(n-1);

}

### Processo

Un processo è un programma in esecuzione. Esso ha vari stati:

* Attivo, ovvero sta eseguendo operazioni.
* Sospeso, ovvero interrompe momentaneamente la sua esecuzione per dar spazio ad altri processi.
* Creato, ovvero un processo inizia la sua esecuzione.
* Concluso, ovvero un processo termina la sua esecuzione.

### Stack dei processi

Lo *stack dei processi* è la pila (*stack*) che contiene i processi sospesi, rispettando l’ordine temporale di esecuzione.

Quando uno stack si svuota, l’algoritmo termina la sua esecuzione.

### Programmazione ricorsiva

La *programmazione ricorsiva* permette di descrivere gli algoritmi basati sull’approccio *divide et impera*.

Consiste nella suddivisione dell’*istanza principale* in *istanze più semplici*, e nella loro successiva risoluzione attraverso le *auto-attivazioni*.

L’istanza più semplice viene definita *istanza banale*.

### Ricerca binaria ricorsiva

L’*algoritmo* *di ricerca binaria ricorsivo* è un algoritmo di ricerca di una chiave in un array ordinato di dati. I dati possono essere numeri, caratteri, stringhe di caratteri o qualsiasi altro insieme di dati su cui sia definita una relazione di ordine. L’algoritmo di ricerca binaria è un algoritmo ottimale per tale problema e la sua complessità di tempo asintotica è logaritmica.

Ha come dati di input la *chiave* di ricerca, un *array* e il suo *size*, e ha come dato di output un dato scalare intero che vale 1 (true) se la *chiave appartiene* all’array, oppure il valore 0 (false) se la *chiave non appartiene* all’array.

L’algoritmo di ricerca binaria ricorsivo è basato sull’approccio *divide et impera*, che risulta particolarmente efficiente in quanto consente a ogni passo non solo di suddividere una data istanza in due istanze di dimensioni dimezzate, ma soprattutto di eliminare una delle due istanze continuando la ricerca solo su una delle due porzioni dimezzate. Infatti, a ogni *auto-attivazione*, si crea un processo che risolve una istanza del problema della ricerca in un array ordinato, cioè su una “porzione” dell’array A, e tale processo agisce su una “porzione” di dimensione dimezzata rispetto alla precedente, fino ad arrivare eventualmente alla dimensione , che è la dimensione dell’*istanza banale* del problema della ricerca.

Diamo una rapida analisi della meccanica dell’algoritmo di ricerca binaria ricorsivo. Alla prima attivazione si crea un processo che agisce sull’intero array A. Alla generica auto-attivazione, si crea un processo che effettua la ricerca su una porzione dell’array A. Tale porzione è individuata dall’indirizzo base della porzione e dal size della porzione.

Il *caso base* della ricorsione è costituito da due distinte situazioni:

* la prima è che il *size* della porzione è , e in tal caso si deve restituire 0, cioè false, in quanto la *chiave non appartiene* alla porzione e quindi non appartiene all’array;
* la seconda è che l’elemento in *posizione centrale* della porzione (non vuota) che si sta considerando è *uguale alla chiave*, e in tal caso si deve restituire , cioè true.

Se l’istanza che il processo sta risolvendo non rientra nel caso base, allora si deve confrontare la *chiave* e l’*elemento dell’array* che si trova nella *posizione centrale*:

* se la *chiave* è *minore* la ricerca proseguirà con l’auto-attivazione sulla semi-porzione “di sinistra”, cioè la porzione che ha lo stesso indirizzo base della porzione in input e size dimezzato;
* se la *chiave* è *maggiore* la ricerca proseguirà con l’auto-attivazione sulla semi-porzione “di destra”, cioè la porzione che ha come indirizzo base l’indirizzo della posizione immediatamente a destra della posizione centrale e size dimezzato.

Questa è l’implementazione in C dell’algoritmo di ricerca binaria ricorsiva, per una *chiave* e un *array* di tipo char (vengono usati 0 per false e 1 per true):

int ricerca\_binaria(char chiave, char elenco[], int n) {

int mediano;

if(n==0) //caso base – prima situazione

return 0;

mediano=(n-1)/2;

if(chiave==elenco[mediano]) //caso base – seconda situazione

return 1;

else if(chiave<elenco[mediano])

/\*auto-attivazione porzione di sinistra\*/

return ricerca\_binaria(chiave, elenco, mediano);

else

/\*auto-attivazione porzione di destra\*/

return ricerca\_binaria(chiave, elenco+mediano+1,n-mediano-1);

}

Questa è l’implementazione in C dell’algoritmo di ricerca binaria ricorsiva, per una *chiave* e un *array* di tipo char (viene definito il tipo booleano):

typedef enum{false, true} boolean; //definizione del tipo booleano

boolean ricerca\_binaria(char chiave, char elenco[], int n) {

int mediano;

if(n==0) //caso base – prima situazione

return false;

mediano=(n-1)/2;

if(chiave==elenco[mediano]) //caso base – seconda situazione

return true;

else if(chiave<elenco[mediano])

/\*auto-attivazione porzione di sinistra\*/

return ricerca\_binaria(chiave, elenco, mediano);

/\*elenco è l’indirizzo base della porzione di sinistra\*/

/\*mediano è il size della porzione di sinistra\*/

else

/\*auto-attivazione porzione di destra\*/

return ricerca\_binaria(chiave, elenco+mediano+1, n-mediano-1);

/\*elenco+mediano+1 è l’indirizzo base della porzione di destra\*/

/\*n-mediano-1 è il size della porzione di destra\*/

}

Vediamo adesso l’implementazione in C dell’algoritmo di ricerca binaria dando in input una *chiave* di tipo *array* *di* char, un *array* di tipo *stringa di* char e il suo size, e usando sempre la versione booleana:

typedef enum{false, true} boolean;

boolean ricerca\_binaria(char chiave[], char \*elenco[], int n) {

int mediano;

if(n==0) //caso base – prima situazione

return false;

mediano=(n-1)/2;

if(strcmp(chiave, elenco[mediano])==0) //caso base – seconda situazione

return true;

else if(strcmp(chiave, elenco[mediano])<0)

/\*auto-attivazione porzione di sinistra\*/

return ricerca\_binaria(chiave, elenco, mediano);

/\*elenco è l’indirizzo base della porzione di sinistra\*/

/\*mediano è il size della porzione di sinistra\*/

else

/\*auto-attivazione porzione di destra\*/

return ricerca\_binaria(chiave, elenco+mediano+1, n-mediano-1);

/\*elenco+mediano+1 è l’indirizzo base della porzione di destra\*/

/\*n-mediano-1 è il size della porzione di destra\*/

}

Un breve commento al codice. La function ha la classica struttura delle function ricorsive, cioè un if-then-else che distingue il *caso base* dalle *istanze non banali*. Si noti che:

* il primo if si riferisce alla prima situazione del caso base;
* il secondo if alla seconda situazione del caso base;
* il blocco else gestisce l’auto-attivazione:
  + la prima auto-attivazione agisce sulla semi-porzione di sinistra, che ha lo stesso indirizzo base della porzione ricevuta in input e size uguale al valore dell’indice mediano;
  + la seconda auto-attivazione agisce sulla semi-porzione di destra, che ha come indirizzo base quello della posizione immediatamente a destra della posizione centrale della porzione ricevuta in input e size uguale al valore n-mediano-1.

Analizziamo ora le complessità dell’algoritmo di ricerca binaria ricorsiva:

* ***Complessità di tempo***

Negli algoritmi di ricerca l’operazione dominante è l’operazione di confronto tra la *chiave* e un *elemento dell’array*. Nel caso dell’algoritmo di ricerca binaria l’operazione dominante è il *confronto a tre vie* tra la *chiave* e un *elemento dell’array*. L’operazione di confronto a tre vie è l’ if-else-if che appare nel corpo della function. Come per tutti gli algoritmi di ricerca, la complessità di tempo dipende dal valore dei dati oltre che dalla dimensione computazionale e quindi si deve effettuare un’analisi di complessità di *caso peggiore*. Il modo più semplice per determinare la complessità di tempo dell’algoritmo ricorsivo di ricerca binaria consiste nell’esaminare il cosiddetto *albero binario* delle decisioni associato all’esecuzione dell’algoritmo per un dato problema. I *nodi* di tale albero sono gli elementi dell’array e il *livello* in cui appaiono nell’albero corrisponde al livello dell’auto-attivazione in cui sono confrontati con la chiave. In altre parole, la *radice* dell’albero è l’*elemento* che è confrontato con la *chiave* alla prima attivazione dell’algoritmo, i due nodi del livello sono i due elementi che possono essere confrontati con la chiave al secondo livello di auto-attivazione (che può essere sulla semi-porzione di sinistra o sulla semi-porzione di destra), ovvero l’elemento in *posizione centrale* della semi-porzione di sinistra oppure l’elemento in posizione centrale della semi-porzione di destra, e così via.

È chiaro che, poiché l’algoritmo a ogni auto-attivazione genera un processo che effettua la ricerca solo una delle due semi-porzioni della porzione dell’auto-attivazione precedente, a ogni livello corrisponde una sola operazione di confronto a tre vie. Quindi si può concludere che il numero totale di confronti a tre vie è uguale alla *profondità* (cioè il numero di livelli) del corrispondente albero delle decisioni.

Se l’albero fosse *completo* (cioè ogni nodo diverso da una foglia ha esattamente due nodi figli) la sua profondità sarebbe . È facile convincersi che l’albero delle decisioni dell’algoritmo di ricerca binaria, in generale, è *quasi-completo*, nel senso che solo il livello delle foglie può essere incompleto, e quindi la sua profondità è , dove , detto il *floor* di , indica il *più grande intero minore o uguale* di .

In conclusione, la complessità di tempo dell’algoritmo della ricerca binaria è:

al più

Si noti che il caso peggiore è quello in cui la *chiave non appartiene* all’array, oppure quello in cui la *chiave* è *uguale* a una delle *foglie* del corrispondente albero delle decisioni.

* ***Complessità di spazio***

, ovvero il size dell’array, dato che tale algoritmo opera *in place*.

* ***Complessità asintotica***

La complessità asintotica è: .

### Somma degli elementi di un array ricorsiva – approccio incrementale

L’*algoritmo ricorsivo di somma di un array*, *approccio incrementale*, calcola la somma degli elementi di un array di dati. I dati possono essere numeri interi, numeri reali o qualsiasi altro insieme di dati su cui sia definita una operazione di somma.

Ha come dati di input un *array* e il suo *size*, e ha come dato di output un dato scalare, che è il valore della *somma* di tutti gli elementi dell’array. Non fa uso di altri array o strutture dati.

L’*approccio incrementale* è una metodologia di progetto di algoritmi in cui la soluzione del problema viene determinata risolvendo *iterativamente*, a partire dall’*istanza banale*, istanze di dimensione *crescente* (incrementata ogni volta di ) del problema, fino ad arrivare all’istanza in input. A ogni iterazione, la soluzione dell’istanza è calcolata utilizzando la soluzione (già calcolata) dell’istanza precedente.

La tecnica di programmazione ricorsiva, basata sull’auto-attivazione di una function, consente di descrivere facilmente un algoritmo basato sull’approccio incrementale, usando la cosiddetta *tail recursion*.

Nel caso del problema della somma degli elementi di un array, la soluzione del problema, cioè la somma, viene calcolata come la somma tra l’*ultimo elemento* dell’array e la somma degli elementi della porzione dell’array che va *dal primo al penultimo elemento*. Si noti che si tratta di una sorta di “variante” rispetto all’approccio divide et impera: qui, la porzione “di sinistra” invece di avere size dimezzato (come nel divide et impera) ha size diminuito di e la porzione “di destra” invece di avere size dimezzato (come nel divide et impera) è costituita da un unico elemento (in particolare, l’*ultimo elemento*). Di qui il nome *tail recursion*.

Ogni auto-attivazione suddivide una data istanza in due istanze:

* una di *size* , la cui soluzione è proprio l’unico elemento che la costituisce;
* una di dimensione *diminuita* di

Successivamente crea un processo che risolve l’istanza del problema della somma sulla porzione di *size diminuito* di ; e quindi restituisce la *somma* tra il *valore dell’ultimo elemento* della porzione e il *valore ottenuto* da tale processo. A ogni auto-attivazione, il processo creato agisce su una “porzione” di size *diminuito* di rispetto al precedente, fino ad arrivare al *size* , che è il *size* dell’*istanza banale* del problema della somma.

Diamo una rapida analisi della meccanica dell’algoritmo di somma degli elementi di un array ricorsivo. Alla prima attivazione si crea un processo che agisce sull’intero array A, di size ; esso crea a sua volta un processo che effettua la somma sulla porzione dell’array A costituita dai primi elementi; il valore calcolato da questo processo è *sommato all’ultimo elemento* e il risultato è restituito. La porzione è individuata dall’*indirizzo base* e dal *size* .

Il *caso base* della ricorsione è costituito dalla porzione di *size* , e in tal caso si deve restituire il valore dell’unico elemento della porzione. Se l’istanza che il processo sta risolvendo non rientra nel caso base, allora si deve auto-attivare sulla porzione , cioè la porzione che ha lo *stesso indirizzo base* della porzione in input e *size diminuito* di , e sommare il risultato di tale auto-attivazione con l’elemento in *ultima posizione* della porzione in input, restituendo tale somma al chiamante.

Questa è l’implementazione in C dell’algoritmo di somma degli elementi di un array ricorsivo, con approccio incrementale, per un array di tipo int:

int somma\_ricorsiva(int a[], int n) {

if(n==1)

return a[0]; //caso base

else

/\*auto-attivazione\*/

return a[n-1]+somma\_ricorsiva(a, n-1);

}

Un breve commento al codice. La function ha la classica struttura delle function ricorsive, cioè un if-then-else che distingue il *caso base* dalle *istanze non banali*.

Il blocco then gestisce il caso base, restituendo il valore dell’unico elemento di una porzione di *size* .

Il blocco else gestisce l’auto-attivazione. L’auto-attivazione agisce sulla porzione che ha lo *stesso indirizzo base* della porzione ricevuta e *size* uguale a , cioè un *size diminuito* di rispetto al *size* della porzione ricevuta. La soluzione dell’istanza è calcolata *sommando* il *valore dell’ultimo elemento* della porzione ricevuta con il *valore restituito dall’auto-attivazione*, ed è restituita al *chiamante*.

È possibile, tuttavia, modificare l’ordine in cui vengono eseguite le somme, facendo partire le somme da destra verso sinistra. Ciò è possibile andando a modificare l’auto-attivazione.

Questa è l’implementazione in C dell’algoritmo di somma degli elementi di un array ricorsivo, con approccio incrementale, per un array di tipo int, avendo modificato l’ordine:

int somma\_ricorsiva(int a[], int n) {

if(n==1)

return a[0]; //caso base

else

/\*auto-attivazione\*/

return a[0]+somma\_ricorsiva(a+1, n-1);

}

Analizziamo le complessità dell’algoritmo di somma degli elementi di un array ricorsivo con approccio incrementale:

* ***Complessità di tempo***

L’operazione dominante è l’operazione di somma tra due numeri. C’è un’unica operazione di somma, che è effettuata nel blocco else a ogni auto-attivazione, con esclusione di quella relativa al *caso base*. Poiché il numero totale di attivazioni, esclusa quella del caso base, è , è immediato concludere che il numero di totale di somme è .

In conclusione, la complessità di tempo dell’algoritmo di somma degli elementi di un array ricorsivo con approccio incrementale è:

somme

Si noti che si tratta di una *complessità assoluta*, cioè indipendente dal valore degli elementi dell’array, e che tale complessità è identica a quella del classico algoritmo non ricorsivo di somma degli elementi di un array basato sull’approccio incrementale. In particolare, in entrambi gli algoritmi (ricorsivo e iterativo) l’ordine secondo cui sono sommati i vari numeri è esattamente lo stesso. In altre parole, i due algoritmi descrivono la *medesima sequenza computazionale di operazioni*.

* ***Complessità di spazio***

, ovvero il size dell’array, dato che questo algoritmo opera *in place*.

* ***Complessità asintotica***

La complessità asintotica è: .

### Somma degli elementi di un array ricorsiva – approccio divide et impera

L’*algoritmo ricorsivo di somma di un array*, *approccio divide et impera*, calcola la somma degli elementi di un array di dati. I dati possono essere numeri interi, numeri reali o qualsiasi altro insieme di dati su cui sia definita una operazione di somma.

Ha come dati di input un *array* e il suo *size*, e ha come dato di output un dato scalare, che è il *valore della somma di tutti gli elementi* dell’array. Non fa uso di altri array o strutture dati.

L’algoritmo di somma degli elementi di un array ricorsivo, approccio divide et impera, è basato sull’applicazione dell’approccio divide et impera al problema della somma degli elementi di un array. Tale approccio è una metodologia di progetto di algoritmi in cui la soluzione del problema viene determinata suddividendo ricorsivamente il problema in due istanze di dimensione dimezzata (fase *divide*) e poi esprimendo la soluzione combinando opportunamente le due soluzioni delle istanze dimezzate (fase *impera*).

Nel caso del problema della somma degli elementi di un array, la soluzione del problema, cioè la somma, viene calcolata come la somma tra la somma degli elementi della porzione “di sinistra” e la somma degli elementi della porzione “di destra” in cui si può suddividere l’array. Ogni auto-attivazione suddivide una data istanza in due istanze di dimensioni dimezzate, crea un processo che risolve l’istanza del problema della somma sulla “porzione di sinistra” e un processo che risolve l’istanza del problema della somma sulla “porzione di destra” della porzione ricevuta in input e quindi restituisce la somma dei valori ottenuti dai due processi. A ogni auto-attivazione, i due processi creati agiscono ognuno su una “porzione” di *size dimezzato* rispetto al precedente, fino ad arrivare al *size* , che è il *size dell’istanza banale* del problema della somma.

Diamo una rapida analisi della meccanica dell’algoritmo di somma degli elementi di un array ricorsivo, con approccio divide et impera. Alla prima attivazione si crea un processo che agisce sull’intero array A; esso crea a sua volta un processo che effettua la *somma sulla porzione di sinistra* dell’array A e un processo che effettua la *somma sulla porzione di destra* dell’array A; i valori restituiti da questi due processi sono sommati e restituiti. Le porzioni sono individuate dall’*indirizzo base e dal size*.

Il *caso base* della ricorsione è costituito dalla porzione di *size* , e in tal caso si deve restituire il valore dell’unico elemento della porzione. Se l’istanza che il processo sta risolvendo non rientra nel caso base, allora si deve calcolare la *posizione centrale* della porzione; poi si deve auto-attivare sulla porzione “di sinistra”, cioè la porzione che ha lo *stesso indirizzo base* della porzione in input e *size dimezzato*, e auto-attivare sulla porzione “di destra”, cioè la porzione che ha come indirizzo base l’*indirizzo della posizione immediatamente a destra della posizione centrale* e *size dimezzato*; i due risultati di tali auto-attivazioni devono essere sommati e restituiti al processo chiamante.

Questa è l’implementazione in C dell’algoritmo di somma degli elementi di un array ricorsivo, con approccio divide et impera, per un array di tipo int:

int somma\_divideEtImpera(int a[], int n) {

int mediano;

if(n==1) //caso base

return a[0];

else {

mediano=(n-1)/2;

/\*auto-attivazione\*/

return somma\_divideEtImpera(a, mediano+1)+somma\_divideEtImpera(a+mediano+1, n-mediano-1);

}

Un breve commento al codice. La function ha la classica struttura delle function ricorsive, cioè un if-then-else che distingue il caso base dalle istanze non banali. Il blocco else gestisce l’auto-attivazione. La prima auto-attivazione agisce sulla semi-porzione di sinistra, che ha lo *stesso indirizzo base* della porzione ricevuta in input e *size* uguale al valore dell’indice mediano+1; la seconda auto-attivazione agisce sulla semi-porzione di destra, che ha come *indirizzo base* quello della *posizione immediatamente a destra della posizione centrale* (a+mediano+1) della porzione ricevuta in input e *size* uguale al valore n-mediano-1. La soluzione dell’istanza è calcolata sommando i valori restituiti dalle due auto-attivazioni ed è restituita al chiamante. Poiché un’espressione è eseguita da “sinistra verso destra”, l’auto-attivazione sulla porzione di sinistra (che appare come primo addendo nell’espressione somma\_divideEtImpera(a, mediano+1)+somma\_divideEtImpera(a+mediano+1, n-mediano-1)) è eseguita prima dell’auto-attivazione sulla porzione di destra.

Analizziamo le complessità dell’algoritmo della somma degli elementi di un array ricorsivo, con approccio divide et impera.

* ***Complessità di tempo***

L’operazione dominante è l’operazione di somma tra due numeri. Il modo più semplice per determinare la complessità di tempo consiste nell’esaminare il cosiddetto *albero binario* delle operazioni associato all’esecuzione dell’algoritmo per un dato problema. I *nodi foglia* di tale albero sono gli *elementi dell’array*, i nodi interni contengono i valori delle somme delle varie porzioni in cui l’array viene suddiviso durante le successive auto-attivazioni, e il *livello* in cui appaiono nell’albero corrisponde al *livello di auto-attivazione*. In altre parole, la *radice* dell’albero contiene la *soluzione del problema*, ottenuta sommando dal basso verso l’alto le varie coppie di numeri.

È chiaro esaminando l’albero che per ottenere il valore di ciascuno dei *nodi interni* (cioè tutti i nodi escluse le foglie) è necessario effettuare esattamente una somma; quindi, il numero totale di somme è uguale al numero dei nodi interni. Poiché il numero delle foglie è , in base a una delle proprietà degli alberi binari (completi), si ha che il numero dei nodi interni è . In conclusione, la complessità di tempo dell’algoritmo di somma degli elementi di un array ricorsivo, con approccio divide et impera, è:

somme

Si noti che si tratta di una *complessità assoluta*, cioè indipendente dal valore degli elementi dell’array, e che tale complessità è identica a quella del classico algoritmo non ricorsivo di somma degli elementi di un array basato sull’approccio incrementale.

* ***Complessità di spazio***

, ovvero il size dell’array, dato che tale algoritmo opera *in place*.

* ***Complessità asintotica***

La complessità asintotica è: .

### Massimo degli elementi di un array ricorsivo – approccio incrementale

*Algoritmo:*

int max(int x, int y) {

if(x>=y)

return x;

else

return y;

}

int max\_ricorsivo(int a[], int n) {

if(n==1) //caso base

return a[0];

else

/\*auto-attivazione\*/

return max(a[n-1], max\_ricorsivo(a, n-1));

}

Analizziamo le complessità dell’algoritmo del massimo degli elementi di un array ricorsivo, con approccio incrementale.

* ***Complessità di tempo***

La complessità di tempo è:

.

* ***Complessità di spazio***

, ovvero il *size* dell’array, dato che questo algoritmo opera *in place*.

* ***Complessità asintotica***

La complessità asintotica è: .

### Massimo degli elementi di un array ricorsivo – approccio divide et impera

*Algoritmo:*

int max(int x, int y) {

if(x>=y)

return x;

else

return y;

}

int max\_ricorsivo(int a[], int n) {

int mediano;

if(n==1) //caso base

return a[0];

else {

mediano=(n-1)/2

/\*auto-attivazione\*/

return max(max\_ricorsivo(a, mediano+1), max\_ricorsivo(a+mediano+1, n-mediano-1));

}

La complessità di tempo dell’algoritmo del massimo degli elementi di un array è:

;

la complessità asintotica è:

.

### Algoritmo di Euclide – Massimo comun divisore (MCD)

*Algoritmo:*

int mcd(int a, int b) {

int r=a%b;

if(r==0)

return b;

else

return mcd(b, r);

}

La complessità di tempo dell’algoritmo di Euclide è: ;

la complessità asintotica è: .

### Algoritmo di Fibonacci – complessità esponenziale

*Algoritmo:*

int fibonacci(int n) {

if(n==1 || n==0)

return n;

else

return fibonacci(n-1)+fibonacci(n-2);

}

La complessità di tempo dell’algoritmo di Fibonacci è: , dove

la complessità asintotica è: .

### Algoritmo di Fibonacci – complessità lineare – programmazione dinamica

Per questo algoritmo con questa programmazione viene definito un array globale:

double fibogia[100];

e lo carichiamo in questo modo:

for(i=0;i<100;i++)

fibogia[i]=0.0;

*Algoritmo:*

double fibonacci(int n) {

if(n<=1)

return (double)n;

if(fibogia[n]!=0)

return fibogia[n];

else {

fibogia[n]=fibonacci(n-1)+fibonacci(n-2);

return fibogia[n];

}

}

Questo algoritmo evita di ricalcolare più volte lo stesso numero di fibonacci.

La complessità di tempo è: , che è la medesima complessità dell’algoritmo di Fibonacci iterativo.

### Algoritmo per il calcolo della lunghezza di una stringa ricorsivo

*Algoritmo:*

int strlen\_ricorsivo(char \*str) {

if(\*str==’\0’)

return 0;

else

return strlen\_ricorsivo(str+1);

}

### Algoritmo per contare i numeri pari di un array ricorsivo

Per questo algoritmo definiamo una function booleana che indica il resto della divisione:

boolean e\_pari(int x) {

return !(x%2);

}

*Algoritmo:*

int num\_pari\_ricorsivo(int a[], int n) {

if(n==0)

return 0;

else if(e\_pari(a[0]))

return 1+num\_pari\_ricorsivo(a+1, n-1);

else

return num\_pari\_ricorsivo(a+1, n-1);

### Algoritmo di ricerca sequenziale ricorsiva

*Algoritmo:*

boolean ricerca\_sequenziale(int a[], int n, int chiave) {

if(n==0)

return false;

else if(a[n-1]==chiave)

return true;

else

return ricerca\_sequenziale(a, n-1, chiave);

}

### Algoritmo di uguaglianza tra due stringhe ricorsivo

*Algoritmo:*

int uguaglianza\_stringhe(char \*str1, char \*str2) {

if((\*str1==’\0’ && \*str2!=’\0’) || (\*str1!=’\0’ && \*str2!=’\0’))

return 0; //caso base

if(\*str1==’\0’ && \*str2==’\0’)

return 1; //caso base

else if(\*str1==\*str2)

return uguaglianza\_stringhe(str1+1, str2+1);

else

return 0; //caso base

}

### Algoritmo di string matching ricorsivo

*Algoritmo:*

int string\_matching(char \*testo, char \*chiave) {

int ris;

if(\*(testo+strlen(chiave)-1)==’\0’)

return 0;

else {

ris=strnlen(testo, chiave, strlen(chiave));

if(ris==0)

return 1+string\_matching(testo+1, chiave);

else

return string\_matching(testo+1, chiave);

}

}

### Algoritmo di ordinamento per inserimento ricorsivo (insertion sort ricorsivo)

*Algoritmo:*

void insertion\_sort(int a[], int n) {

int el\_da\_ins, j;

if(n<=1)

return 0;

insertion\_sort(a, n-1);

el\_da\_ins=a[n-1];

j=n-2;

while(j>=0 && el\_da\_ins<a[j]) {

a[j++]=a[j];

j--;

}

a[j++]= el\_da\_ins;

}

## Struct

### Record

Il *record* è un tipo strutturato e viene realizzato attraverso le struct; tale meccanismo permette di dichiarare specifici tipi derivati.

### Dichiarazione di un record

Una *struct* si dichiara:

struct <etichetta> {

<tipo> <variabile>;

<tipo> <variabile>;

…

<tipo> <variabile>;

};

Esempio:

struct punto2D { 🡪 punto2D è il nome del tipo derivato

double ascissa;

double ordinata;

};

La dichiarazione di una struct non alloca memoria. Per allocare memoria bisogna dichiarare una variabile di tipo struct:

struct punto2D vertice, estremo\_sx;

in questo modo viene allocata memoria per le variabili vertice ed estremo\_sx e queste saranno di tipo struct punto 2D.

Una variabile di tipo struct può essere dichiarata anche in questi modi:

1. struct data {

int giorno;

char mese[10];

int anno;

} dataNascita, partenza;

in questo modo le variabili sono dichiarate insieme alla struttura struct data.

1. struct {

int giorno;

char mese[10];

int anno;

} dataNascita, partenza;

in questo modo le variabili sono dichiarate insieme alla struttura che, non avendo etichetta, non può essere riutilizzata per dichiarare altre variabili.

1. typedef struct data Data;

qui data è un nuovo tipo di dato.

È possibile avere anche delle struct annidate. Esempio:

struct indirizzo {

struct {

char \*toponimo;

int nCivico;

} sede;

int CAP;

char \*comune;

char \*provincia;

};

### Accesso ai campi di una struct

Per accedere ai campi di una struct si utilizza la *dott notation*. Prendiamo in considerazione struct dataNascita:

dataNascita.giorno=20;

dataNascita.mese=”aprile”;

dataNascita.anno=2002;

Per poter assegnare i campi di una struct a un’altra si procede in questo modo:

<nome struct 1>=<nome struct 2>;

quindi ogni campo di struct 1 avrà lo stesso valore di struct 2.

## Numeri casuali e pseudocasuali

### Fenomeno casuale

Un fenomeno viene definito *casuale* se non è prevedibile; esso viene indicato come fenomeno *random*.

### Fenomeno pseudocasuale

Un fenomeno viene definito *pseudocasuale* se sembra casuale, ma non lo è; esso viene indicato come fenomeno *pseudorandom*.

### Successione di numeri casuale

Una successione di numeri viene definita *casuale* se l’n-simo numero della successione non dipende dai precedenti, ovvero non è prevedibile dagli n- numeri.

### Successione di numeri pseudocasuale

Una successione di numeri viene definita *pseudocasuale* se non esiste un algoritmo a complessità polinomiale che è in grado di stabilire se essa è diversa da una successione casuale. (esempio: è una sequenza pseudocasuale).

Tali tipo di successione può essere sempre riprodotta ed è sempre *periodica*.

I numeri di una successione pseudocasuale appartengono ad un intervallo prefissato: dove viene definito RAND\_MAX.

### Generatore di numeri pseudocasuali

Un *generatore di numeri pseudocasuali* consiste in un algoritmo con lo scopo di generare una successione di numeri pseudocasuali; essi sono basati su opportune *formule ricorrenti*. Alcuni esempi sono:

* *Generatore lineare a congruenza* . Questa è una formula ricorrente del primo ordine, quindi necessita di caso base, che viene definito *seed*: corrisponde al valore (o i valori) iniziali della formula che stiamo analizzando. In C, per generare un numero casuale, viene utilizzata la funzione rand(); per impostare il seed con la function rand() si usa la function srand(<numero seed>). (esempio: srand(10); il seed assume valore 10). La successione di numeri pseudocasuali che verrà generata dipende dal seed che viene indicato.
* *Generatore di Fibonacci ritardato* . Altre formule di questo tipo di generatore sono:

Dove: e .

### Cambio di intervallo

Supponiamo di avere una successione di numeri interi che ha intervallo e si vuole una successione di numeri interi, ma con intervallo . Per fare ciò esiste una formula: , che in linguaggio C corrisponde a rand()%(B+1). Con questo cambio di intervallo si va a modificare RAND\_MAX (da ).

Se invece, oltre al RAND\_MAX, si vuole modificare anche l’inizio dell’intervallo, si ricorre alla seguente formula: , che in linguaggio C corrisponde a 1+rand()%(A). Un esempio può essere il lancio del dado, che in linguaggio C corrisponde al codice 1+rand()%6. In alternativa a quest’ultima formula si può usare la formula: .

### Function rand

Per generare numeri casuali in linguaggio C si usa la function rand() che genera, precisamente, numeri interi che vanno dall’intervallo . Per usare questa function si usa la libreria <stdlib.h>.

In C il seed è fissato per default, quindi ad ogni esecuzione viene generata la stessa sequenza di numeri pseudocasuali. Per modificare il seed si usa la function srand(<nuovo seed>): in questo modo verrà generata una nuova sequenza di numeri pseudocasuali.

Per avere sempre sequenze diverse di numeri pseudocasuali si usa la function time(0), contenuta nella libreria <time.h>, che restituisce l’ora, i minuti e i secondi dell’istante in cui viene avviato il programma: in questo modo, ad ogni istante si avrà sempre una nuova sequenza di numeri pseudocasuali; la si scrive in questo modo:

srand((unsigned int)time(0));

### Generare un tipo reale – intervallo

Per generare numeri pseudocasuali di tipo reale si utilizza questa tecnica:

nc=(float)rand()/(float)RAND\_MAX;

### Generare un tipo reale – intervallo

Per generare numeri pseudocasuali di tipo reale si utilizza questa tecnica:

nc=a+(b-a)\*rand()/(float)RAND\_MAX;

## Memoria dinamica

### Memoria stack e heap

Con la memoria *stack* si indica la memoria statica, che contiene tutto ciò che è dichiarato nel codice.

Con memoria *heap* si indica la memoria dinamica, in cui la dimensione degli elementi può variare nella fuse di runtime del programma.

### Gestione della memoria dinamica

Per allocare memoria si utilizzano due funzioni:

* malloc(): in input ha solo un parametro, ovvero la dimensione dell’elemento da allocare;
* calloc(): in input ha due parametri, ovvero il numero di elementi da creare e la dimensione degli elementi che verranno allocati;

Esempi:

*Con malloc:*

int \*v;

v=(int \*)malloc(n\*sizeof(int));

*Con calloc:*

int \*v;

v=(int \*)calloc(n, sizeof(int));

In entrambi i casi si va a creare un array di n elementi di tipo int; unica differenza è che la calloc() setta a zero tutti gli elementi che andrà a creare

Per liberare memoria si utilizza la funzione:

* free();

Per, invece, modificale lo spazio allocato si utilizza la funzione:

* realloc();

### Libreria <malloc.h>

Per poter utilizzare le funzioni di gestione della memoria dinamica si usa la libreria <malloc.h>.