Complessità Computazionale di un Algoritmo

Dato un problema, è possibile determinare un insieme, generalmente finito, di algoritmi in grado di risolverlo. La scelta di un algoritmo piuttosto che un altro può essere basata su differenti fattori. Ad esempio il tempo di esecuzione, le risorse necessarie alla sua esecuzione, etc. Tra essi, quello che gioca un ruolo essenziale è la complessità computazionale, che può essere misurata come tempo necessario all'esecuzione dell'algoritmo su un computer di architettura tradizionale.

La valutazione della complessità computazionale deve soddisfare due requisiti:

1) Non deve dipendere da una particolare macchina o da un particolare compilatore. Ciò è richiesto in quanto:

- Il tempo di esecuzione di ogni istruzione in un qualunque linguaggio (ad esempio in linguaggio macchina), non è uguale per tutti i calcolatori ma varia in dipendenza ad esempio della frequenza di lavoro della CPU (clock), dall'architettura del calcolatore o dal set di istruzioni di cui la macchina dispone (ad esempio la decodifica di ogni singola istruzione prima della sua esecuzione, è più rapida in una macchina che usa un set limitato di istruzioni, ossia in una macchina RISC).
- Il numero di istruzioni in linguaggio macchina che vengono generate in corrispondenza ad un programma scritto in un linguaggio ad alto livello, dipende dal particolare compilatore. È possibile dunque che uno stesso programma venga tradotto in programmi in linguaggio macchina di lunghezza (ossia numero di istruzione) differenti.

2)Deve essere espressa in funzione dei dati in ingresso. Dato che un algoritmo elabora dei dati in ingresso per fornirne altri che rappresentano la soluzione del problema, quello che interessa sapere è come il tempo di esecuzione dell'algoritmo varia al variare dei dati in ingresso. In particolare, la dipendenza del tempo di esecuzione dai dati in ingresso al problema può essere relativa sia alla dimensione dei dati in ingresso che al loro valore.

- **Dipendenza dalla Dimensione dei Dati.** La complessità computazionale di un algoritmo dipende sempre dal numero dei dati che devono essere elaborati dall'algoritmo, ossia dai dati in ingresso. Tale numero viene definito dimensione dei dati in ingresso. Ad esempio, nel caso di un algoritmo di ordinamento di un file o di un array, la dimensione coincide con la lunghezza del file o dell'array. Nel seguito verrà indicata con ***n*** la dimensione dei dati in ingresso.
- **Dipendenza dai Valori dei Dati.** In molti casi la complessità computazionale di un algoritmo dipende non solo dalla dimensione dei dati in ingresso ma anche dai particolari valori assunti dai dati in ingresso. In particolare, può accadere che i tempi di calcolo di un algoritmo in corrispondenza di una determinata dimensione dei dati in ingresso, varino in funzione dei valori

assunti dagli stessi dati in ingresso. Al fine di tener conto dell'influenza dei valori assunti dai dati in ingresso sulla complessità computazionale, in genere vengono considerati due possibili scenari: un tempo di calcolo massimo e uno medio. Nel primo caso, il legame tra tempo di esecuzione e parametro ***n*** viene fissato considerando i valori dei dati in ingresso che comportano il massimo tempo di esecuzione. Nel secondo caso viene considerato il tempo medio di esecuzione su dati di dimensione n, ottenuto considerando tutti i possibili valori dei dati in ingresso, per ciascuno di essi determinando la complessità computazionale e, infine, eseguendo la media dei valori di complessità computazionale ottenuti. Nella pratica ha interesse la valutazione del tempo massimo, ossia del caso peggiore. Tra l'altro, tale parametro è il più facile da misurare. Infatti se volessimo calcolare il tempo medio di esecuzione, dovremmo considerare uno scenario piuttosto significativo di dati in ingresso di dimensione n, e mediare sui relativi tempi di calcolo.

Per quanto detto, lo studio della complessità computazionale di un algoritmo consiste nell'individuare una relazione tra il tempo di esecuzione, la dimensione dei dati, n, e la dipendenza dal particolare valore del dato o dei dati in ingresso. Vista la complessità di quest'ultimo legame, molto spesso si preferisce semplificare la relazione da individuare, considerando solo il legame tra il tempo di esecuzione e la dimensione dei dati, n, in ingresso. Tra tutti gli scenari legati ai valori dei dati in ingresso, si considera quello caratterizzato dal caso peggiore. In ogni caso, la relazione che si vuole individuare deve prescindere sia dalla particolare macchina che dal compilatore utilizzato.

Nel seguito indicheremo con T(n) il tempo di esecuzione di un algoritmo, in funzione di n, considerando il caso peggiore, nel caso in cui tale tempo dipenda dai valori dei dati ingresso.

Il calcolo della complessità computazionale consiste dunque nell'individuare l'espressione della funzione T(n).

1. Confronto tra tempi di esecuzione diversi

In base al tempo di esecuzione T(n) è possibile confrontare algoritmi differenti ed individuare quello che presenta prestazioni migliori. In particolare dal confronto tra i tempi di esecuzione è possibile trarre le seguenti informazioni:

- fissata una stessa dimensione di dati in ingresso, è possibile conoscere i tempi di esecuzione ad essa relativi.
- È possibile che al variare della dimensione dei dati, il risultato del confronto possa essere diverso. Ad esempio se si confronta un andamento quadratico con uno lineare, per bassi valori

di n, il comportamento quadratico può offrire più bassi tempi di calcolo, mentre al crescere di n, il comportamento lineare è di gran lunga preferibile in quanto presenta tempi di esecuzione più bassi. In sostanza, dunque, è possibile valutare la convenienza di un algoritmo rispetto ad un altro in funzione della dimensione n.

- fissato un limite massimo di tempo di esecuzione, è possibile valutare quale è la dimensione massima di n che garantisce l'esecuzione dell'algoritmo entro il limite temporale. In altri termini, a parità di tempo di esecuzione è possibile individuare l'algoritmo che permette di risolvere un problema caratterizzato dal più alto valore di n.

2.Determinazione di T(n): Notazione O e Ω

È stato detto che il calcolo di T(n) deve essere effettuato soddisfacendo due esigenze: indipendenza dalla macchina e dal compilatore, e relazione con la dimensione dei dati in ingresso.

Per raggiungere tali obiettivi si ricorre a particolari notazioni che esplicitano il legame tra la funzione T e n. Tali notazioni sono O (si legge "o" grande) e Ω.

La notazione O fornisce una delimitazione superiore al tempo di esecuzione di un algoritmo, cioè fornisce una valutazione approssimata per eccesso. La definizione della notazione O è:

Sia f(n) una funzione definita sugli interi non negativi n. Diciamo che T(n) è O(f(n)), se T(n) è al più una costante moltiplicativa per f(n), fatta al più eccezione per valori piccoli di n. Più formalmente diremo che T(n) è O(f(n)) se, *nel caso peggiore relativamente ai valori di ingresso*, esiste un numero n_0 e una costante c, tale che $T(n) \leq c \cdot f(n)$ per ogni $n \geq n_0$, ossia se:

$$\exists n_0, c : T(n) \leq c \cdot f(n) \quad \forall n \geq n_0$$

Alcuni valori di O(n) possono essere:

O(1)	costante
O(n)	lineare
$O(n^2)$	quadratica
O(n·logn)	n logaritmo di n

La notazione Ω fornisce una delimitazione inferiore al costo di esecuzione di un algoritmo. La definizione della notazione Ω è:

Sia g(n) una funzione definita sugli interi non negativi n. Diciamo che T(n) è Ω (g(n)), se, *nel caso peggiore relativamente ai valori di ingresso*, esiste una costante c, tale che T(n)≥c·g(n) per un numero infinito di valori di n, ossia:

$$\exists c : T(n) \geq c \cdot g(n) \quad \forall n$$

Chiaramente la delimitazione inferiore al costo di un algoritmo non può essere rappresentata da una funzione che, al crescere delle dimensioni dell'input, cresce più velocemente della delimitazione superiore della complessità. Se le due delimitazioni coincidono allora si ha una valutazione esatta della complessità di un algoritmo.

Generalmente, dato un algoritmo, si preferisce valutare la complessità computazionale in termini di O, poiché essa fornisce una stima del caso peggiore. Nel seguito, il calcolo della complessità computazionale farà riferimento solo a tale fattore.

3.Complessità Computazionale delle più note Istruzioni in C

Visto che in genere il tempo di esecuzione di una istruzione in un linguaggio ad alto livello dipende dal tempo di esecuzione delle istruzioni corrispondenti in linguaggio macchina, la relazione tra il tempo T(n) e la dimensione n, viene ricavata dall'analisi del numero di istruzioni corrispondenti in linguaggio macchina. Il conteggio del numero di istruzioni non viene fatto al fine di ottenere un valore preciso, ma solo per determinare la relazione di dipendenza del numero di istruzioni dalla dimensione, n, del problema (costante, lineare, quadratica etc.). In tal modo viene individuata una relazione tra il numero di istruzioni e la dimensione del problema. Per quanto detto, la relazione tra numero di istruzioni e dimensione n del problema può essere assunta valida anche per il tempo di esecuzione.

Nel seguito verrà valutata la complessità computazionale di alcune istruzioni note.

Operazioni di Assegnazione

Si consideri la seguente assegnazione:

```
x=5
```

Il numero di istruzioni a livello di codice macchina/assembly per eseguire la singola assegnazione è indipendente dalla dimensione dei dati in ingresso, n; anzi tale numero è sempre lo stesso al variare di n. Se ad esempio l'input del programma è rappresentato da una lista contenente n elementi o da un vettore di dimensione n, il numero di istruzioni che assegnano un valore ad una generica variabile è sempre pari a quello mostrato precedentemente. Dunque la relazione tra numero di istruzioni e la dimensione n è una funzione costante. Per quanto detto precedentemente tale

relazione può essere assunta anche per il tempo di esecuzione T(n), che dunque sarà O(1). Per dimostrare ciò basti pensare che esisterà sempre un valore n_0 di n e una costante c per cui $T(n) \leq c \cdot 1$. Ad esempio c può essere preso pari ad una valore più grande del massimo tempo di esecuzione delle istruzioni di assegnamento viste precedentemente.

Operazioni Aritmetiche

Si consideri la seguente operazione aritmetica:

$$x=x+y \cdot k$$

La sequenza di istruzioni in linguaggio assembly/macchina corrispondenti a tale operazione non dipende ovviamente dalla dimensione dei dati in ingresso. Ad esempio se si suppone che il programma gestisce un vettore di n elementi, l'operazione $x=x+y \cdot k$ richiederà sempre lo stesso numero di istruzioni in linguaggio macchina per qualunque valore di n. Anche in questo caso, dunque, la relazione tra numero di istruzioni e n è una costante. Per quanto riguarda T(n) possiamo concludere che la sua relazione con n è anch'essa una costante, ossia O(1).

Operazioni Logiche

Si consideri la seguente operazione di confronto:

```
(a<b)
```

Anche in tal caso si possono fare le stesse considerazioni esposte per le operazioni aritmetiche, perché la relazione tra numero di istruzione in linguaggio macchina/assembly necessarie ad effettuare un confronto e la dimensione n è costante, dunque T(n) è O(1). In generale, una qualunque operazione logica è caratterizzata da complessità O(1).

Istruzioni Condizionali:if else

Si consideri il seguente frammento di programma in linguaggio C:

```
if (a<b && c>d)
        istruzione1;
else istruzione2;
```

Il blocco di istruzioni in linguaggio macchina necessarie ad eseguire l'espressione booleana associata al comando if è costante al variare di n (valgono le stesse considerazioni esposte per gli operatori aritmetici/logici). Il numero delle istruzioni che compongono il blocco1 e blocco2 ovviamente dipende dal tipo di problema affrontato. In ogni caso la loro esecuzione è mutuamente esclusiva e dipende dall'esito dell'espressione booleana associata al comando if.

Considerando il caso peggiore, il numero di istruzioni eseguite è pari a quello necessario per implementare le operazioni atte a computare l'espressione booleana associata al comando if, più il massimo numero di istruzioni tra quelle del blocco1 e quelle del blocco2. Siano TB1(n) e TB2(n) i tempi di esecuzione dei due blocchi, e si supponga che sia O(fB1(n)) e O(fB2(n)) la loro complessità computazionale, rispettivamente. Il tempo T(n) per eseguire l'istruzione condizionale if considerata, risulterà dunque pari a O(1)+max(O(fB1(n)),O(fB2(n))), dove O(1) è relativo all'esecuzione dell'espressione booleana. Più avanti verrà dimostrato come questa espressione possa essere semplificata.

Si noti che il calcolo della complessità computazionale dell'istruzione if potrebbe essere differente da quello appena trattato. Ciò accade se l'espressione booleana dipende dalla dimensione n e si verifica, ad esempio, quando l'espressione contiene una chiamata ad una funzione la cui complessità è dipendente da n. Nei paragrafi successivi verrà considerato tale caso.

Istruzioni Iterative:for

Si consideri il seguente ciclo for

```
for (i=0; i<g(n); i++)
        istruzione;
```

dove si è supposto di esprimere il numero di cicli come una funzione g(n) della dimensione dei dati in ingresso. Ad esempio si consideri un algoritmo che deve riempire un vettore di dimensione n; in tal caso g(n)=n. Vi possono essere dei casi in cui il numero di iterazioni del ciclo è indipendente da n ed in tal caso basta porre g(n)=k, dove k è una costante non dipendente da n e legata al particolare problema da risolvere.

Il numero di istruzioni complessive per eseguire il ciclo for, dipende dalle istruzioni eseguite per ogni ciclo. Le istruzioni presenti nel ciclo for, sono relative alla fase di inizializzazione della variabile i, alla fase di confronto di fine ciclo, all'esecuzione del blocco di istruzioni, all'incremento unitario della variabile di controllo di fine ciclo e al ritorno all'inizio del ciclo (in linguaggio macchina esiste un'istruzione che forza la CPU a tornare all'inizio del ciclo). Sia TI(n) il tempo di esecuzione delle istruzioni, e si supponga che sia O(f(n)) la relativa complessità computazionale. Il tempo T(n) complessivo sarà:

O(1) (relativo all'inizializzazione della variabile i) +

Σ (O(1)(relativo al controllo di fine ciclo)

+ O(f(n)) (relativo all'esecuzione del blocco di istruzioni)

+O(1)(relativo all'incremento della variabile i)

+O(1)(ritorno ad inizio ciclo)

), dove la sommatoria è estesa al numero di iterazioni, che è g(n), ossia una funzione di n. Ossia, la complessità del ciclo for sarà data da:

$$O(1) + \sum_{g(n)} (O(1) + O(f(n)) + O(1) + O(1))$$

Istruzioni Iterative:while e do while

Si consideri il seguente ciclo while

```
while (a<b)
        istruzioni;
```

Come nel caso precedente si supponga che il numero di iterazioni realizzate sia una funzione g(n) della dimensione dei dati, n. Come si vede l'espressione booleana associata al ciclo while è relativa ad un confronto. Nella realtà potrebbe essere più complessa, ma la trattazione mostrata nel seguito può essere sempre valida.

Anche in questo caso il numero di istruzioni complessive dipende dalle istruzioni eseguite per ogni ciclo. Le istruzioni presenti per ogni ciclo sono relative alla fase di valutazione di fine ciclo, all'eventuale salto alla linea di fine ciclo, alla computazione delle istruzioni associate al ciclo while e all'eventuale salto ad un nuovo ciclo. Il numero di istruzioni relativo a: all'eventuale salto alla linea di fine ciclo e all'eventuale salto ad un nuovo ciclo è indipendente dalla dimensione dei dati, n. Il tempo di calcolo relativo alla computazione delle istruzioni associate al ciclo while può, invece, dipendere da n. Sia TI(n) il tempo di esecuzione delle istruzioni, e sia O(f(n)) la relativa complessità computazionale. Il tempo T(n) complessivo sarà:

Σ (

O(1)(relativo al controllo di fine ciclo)

+O(f(n))

+ O(1)(ritorno ad inizio ciclo)

), dove la sommatoria è estesa al numero di iterazioni, g(n).

La complessità del ciclo while sarà, dunque:

$$\sum_{g(n)} (O(1) + O(f(n)) + O(1))$$

Le stesse considerazioni e conclusioni sono valide per il ciclo do-while.

4.Semplificazioni delle Espressioni O

Le espressioni di T(n) viste precedentemente possono essere piuttosto complicate. Allo scopo di semplificarle esistono alcune tecniche basate su alcune proprietà, che verranno illustrate nel seguito.

4.1.Legge Transitiva

Ipotesi: Sia f(n) di O(g(n)) e sia g(n) di O(h(n)).

Tesi: segue che f(n) è O(h(n)).

Dimostrazione. Per ipotesi esiste un n_1 e un c_1 tali che per qualunque $n \geq n_1$ risulta $f(n) \leq c_1 \cdot g(n)$. Sempre in base alle ipotesi, risulta che esiste un n_2 e un c_2 tali che per qualunque $n \geq n_2$ risulta $g(n) \leq c_2 \cdot h(n)$. Sia ora n_0 il più grande tra n_1 e n_2, e sia $c = c_1 \cdot c_2$. Per qualunque $n \geq n_0$ risultano valide entrambe le relazioni ossia, per la proprietà transitiva della relazione ≤, risulta $f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$, ossia f(n) è O(h(n)).

Questa proprietà permette di semplificare alcune espressioni O.

Ad esempio si supponga che T(n) sia $O(n^2-1)$. La funzione n^2-1 è $O(n^2)$. Infatti è possibile sempre trovare un n_0 e una costante c, per cui $n^2-1 \leq c \cdot n^2$, $\forall n \geq n_0$. Ad esempio basta scegliere $n_0=0$ e c=1. In base alla legge transitiva, risulta dunque che T(n) è $O(n^2)$, che risulta molto più semplice.

Si supponga che T(n) sia O(n-1). La funzione n-1 è O(n). Infatti è possibile sempre trovare un n_0 e una costante c, per cui $n-1 \leq c \cdot n$, $\forall n \geq n_0$. Ad esempio basta scegliere $n_0=0$ e c=1. In base alla legge transitiva, risulta dunque che T(n) è O(n).

Come altro esempio, si supponga che T(n) sia $O(n^2-n)$. La funzione n^2-n è $O(n^2)$. Infatti è possibile sempre trovare un n_0 e una costante c, per cui $n^2-n \leq c \cdot n^2$, $\forall n \geq n_0$. Ad esempio basta scegliere $n_0=0$ e c=1. In base alla legge transitiva, risulta dunque che T(n) è $O(n^2)$.

Come ultimo esempio, si supponga che T(n) sia O(log(n-1)). La funzione log(n-1) è O(log n). Infatti è possibile sempre trovare un n_0 e una costante c, per cui $\log(n-1) \leq c \cdot \log n$, $\forall n \geq n_0$. Ad esempio basta scegliere $n_0=2$ e c=1. In base alla legge transitiva, risulta dunque che T(n) è O(log n).

Per la legge transitiva vista precedentemente, se si riuscisse a dimostrare che T(n) è O(n), si potrebbe anche dire che T(n) è $O(n^2)$, $O(n^3)$, etc. Al fine di valutare il tempo di esecuzione di un algoritmo è chiaro che è nostro interesse determinare una espressione O che sia la più precisa possibile, ossia vogliamo determinare una funzione f(n) tale che T(n) sia O(f(n)) e non esista nessun altra funzione g(n) che cresca al crescere di T(n) ma più lentamente di f(n).

4.2.Regola della Somma

Ipotesi: Si supponga di avere un programma, composto dalla sequenza di due blocchi di istruzioni caratterizzati da tempi di esecuzione T1(n) e T2(n). Si supponga che T1(n) sia O(f1(n)) e T2(n) sia O(f2(n)). Per quanto detto fino ad ora, il tempo T(n) di esecuzione dei due blocchi di istruzioni è dato dalla somma dei loro tempi di esecuzione, ossia T(n) è O(f1(n))+O(f2(n)). Si supponga adesso che f2(n) sia O(f1(n)).

Tesi: T(n) sarà O(f1(n)).

Dimostrazione: Per ipotesi esisteranno n_1, n_2, n_3 e c_1, c_2, c_3 per cui risulta:

1. $\forall\ n \geq n_1$, $T1(n) \leq c_1 \cdot f_1(n)$
2. $\forall\ n \geq n_2$, $T2(n) \leq c_2 \cdot f_2(n)$
3. $\forall\ n \geq n_3$, $f2(n) \leq c_3 \cdot f_1(n)$

Sia n_0 il massimo tra n_1, n_2, n_3. Per qualunque $n \geq n_0$, risulta:

$T1(n)+T2(n) \leq c_1 \cdot f1(n)+c_2 \cdot f2(n)$

ma per la 3) possiamo scrivere:

$T1(n)+T2(n) \leq c_1 \cdot f1(n)+c_2 \cdot f2(n) \leq c_1 \cdot f1(n)+c_2 \cdot c_3 \cdot f1(n)$

Dunque, possiamo scrivere:

$T1(n)+T2(n) \leq c_1 \cdot f1(n)+c_2 \cdot c_3 \cdot f1(n)=c \cdot f1(n)$, dove $c=c_1+c_2 \cdot c_3$.

Abbiamo dimostrato che T1(n)+T2(n) è O(f1(n)).

La regola della somma è estremamente importante in quanto comporta delle semplificazioni notevoli nell'espressione di O. Si considerino, per esempio le espressioni O viste per alcune istruzioni elementari:

4.2.1.Istruzioni Condizionali:if else

È stato visto che, detti TB1(n) e TB2(n) i tempi di esecuzione dei due blocchi condizionali, e supposto che tali tempi siano O(fB1(n)) e O(fB2(n)), rispettivamente, il tempo T(n) per eseguire l'istruzione condizionale if else, è pari a:

$$O(1)+\max(O(fB1(n)),O(fB2(n)))$$

Nel caso in cui O(fB1(n)) sia il massimo, la complessità risulta:

O(1)+O(fB1(n))

È possibile dimostrare che 1 è O(fB1(n)). Infatti è sempre possibile trovare un valore n_0 e una costante c, per cui $1 \leq c \cdot f(n)$, $\forall n \geq n_0$. Per tale motivo, in base alla regola della somma, l'espressione:

O(1)+O(fB1(n)) diventa O(fB1(n)).

Nel caso in cui O(fB2(n)) sia il massimo, la complessità risulta:

O(1)+O(fB2(n)) che diviene O(fB2(n)).

Dunque, la complessità dell'istruzione if else, **nel caso in cui la condizione del comando if è composta da una espressione booleana**, è:

$$\max(O(fB1(n)),O(fB2(n)))$$

4.2.2.Istruzioni Iterative:for

È stato dimostrato che il tempo T(n) complessivo è dato dalla somma:

$$O(1) + \sum_{g(n)} (O(1) + O(f(n)) + O(1) + O(1))$$

Semplifichiamo l'espressione dentro la sommatoria. Le tre espressioni O(1) possono essere semplificate in una sola. Infatti si consideri la somma O(1)+O(1). In tal caso f1(n)=1 e f2(n)=1. Siccome f2(n) è O(f1(n)), è possibile applicare la regola della somma, ossia semplificare O(1)+O(1) in O(1). Nel nostro caso abbiamo O(1)+O(1)+O(1). Applicando la regola della somma alla prima coppia e poi alla coppia rimanente si ottiene, O(1)+O(1)+O(1)=O(1).

La sommatoria diviene:

$$O(1) + \sum_{g(n)} (O(1) + O(f(n))$$

Si consideri per il momento solo la sommatoria:

$$\sum_{g(n)} (O(1) + O(f(n))$$

Si consideri l'espressione O(1)+O(f(n)), dove f(n) è incognita. È stato detto prima, che 1 è O(f(n)), qualunque f(n). Dunque, applicando la regola della somma, la sommatoria diventa:

$$\sum_{g(n)} (O(1) + O(f(n)) = \sum_{g(n)} O(f(n) = O(f(n)) + O(f(n)) + O(f(n)) + ... + O(f(n)) = g(n) \cdot O(f(n))$$

E' ovvio che, sebbene ci sia una sommatoria di termini O(f(n)), la regola della somma non poteva essere applicata visto che il numero di termini dipende da g(n).

Dire che T(n) ha complessità g(n)·O(f(n)), significa che esiste un n_0 tale che per qualunque $n \geq n_0$ si ha $T(n) \leq c \cdot g(n) \cdot f(n)$, ossia T(n) è O(g(n) ·f(n)). Dunque risulta:

$$\sum_{g(n)} O(f(n) = O(g(n) \cdot f(n))$$

La complessità dell'istruzione for diviene, dunque:

O(1)+O(g(n)·f(n)). Per la regola della somma si può semplificare in O(g(n)·f(n)). Nel caso molto frequente in cui g(n)=n, risulta O(n·f(n)).

4.2.3.Istruzioni Iterative:while e do while

È stato visto che il tempo T(n) complessivo è:

$$\sum_{g(n)} (O(1) + O(f(n)) + O(1))$$

Semplifichiamo l'espressione dentro la sommatoria. Le due espressioni O(1) possono essere semplificate in una sola. La sommatoria diviene:

$$\sum_{g(n)} (O(1) + O(f(n)))$$

Per quanto detto prima, applicando la regola della somma, la sommatoria diviene:

$$\sum_{g(n)} (O(1) + O(f(n))) == O(f(n)) + O(f(n)) + O(f(n)) + O(f(n)) + \ldots.. + O(f(n)) = g(n) \cdot O(f(n))$$

Si può scrivere allora:

$$\sum_{g(n)} (O(1) + O(f(n))) = O(g(n) \cdot f(n))$$

La complessità dell'istruzione while diviene, O(g(n)·f(n)).

5.Procedura Generale per il Calcolo della Complessità Computazionale in Assenza di Procedure

In base a quanto detto in precedenza, è possibile ricavare il tempo di esecuzione di un qualunque algoritmo tramite l'applicazione di uno o più di uno dei seguenti calcoli.

1. Ogni comando di assegnazione è caratterizzato da tempo di esecuzione O(1).
2. Ogni operazione matematica/logica ha complessità O(1).
3. Ogni comando di tipo while o do while è caratterizzato da complessità O(g(n)·f(n)), dove O(f(n)) è la complessità del corpo del ciclo while o do while e g(n) è il limite superiore del numero di iterazioni del ciclo. **Attenzione si ricorda che tale valore di complessità è corretto nel caso in cui non compaia alcuna chiamata a funzione/procedura dentro la condizione del ciclo while e dentro il corpo di programma del ciclo while. Nel caso in cui sia presente una funzione/procedura nella condizione e/o nel corpo del ciclo while bisogna applicare le procedure di calcolo descritte nel capitolo 6 e 7.**
4. Ogni comando di tipo for è caratterizzato da complessità O(g(n)·f(n)), dove O(f(n)) è la complessità del corpo del ciclo for e g(n) è il limite superiore del numero di iterazioni. **Attenzione si ricorda che tale valore di complessità è corretto nel caso in cui non compaia alcuna chiamata a funzione/procedura dentro la condizione di fine ciclo e dentro il corpo di programma del ciclo for. Nel caso in cui sia presente una funzione/procedura nella condizione e/o nel corpo del ciclo for, bisogna applicare le procedure di calcolo descritte nel capitolo 6 e 7.**
5. Ogni comando condizionale if è caratterizzato da O(max(f1(n),f2(n)), dove O(f1(n)) e O(f2(n)) sono le complessità computazionali dei blocchi relativi alla condizione dell'if e dell'else, rispettivamente. **Attenzione si ricorda che tale valore di complessità è corretto nel caso in cui non compaia alcuna chiamata a funzione/procedura dentro la condizione del comando if. Nel caso in cui sia presente una funzione/procedura nella condizione del comando if, bisogna applicare le procedure di calcolo descritte nel capitolo 6 e 7.**
6. Ogni comando di ingresso/uscita è caratterizzato da tempo di esecuzione O(1). Ciò non è stato dimostrato in precedenza, ma la dimostrazione è riconducibile all'istruzione di assegnazione. Ad esempio il comando **scanf("%d",&x)**, può essere visto come una operazione di assegnazione, avente, dunque, complessità O(1).
7. Ogni blocco di istruzioni è caratterizzato da un tempo di esecuzione dato dalla somma O(f1(n))+O(f2(n))+..+O(fm(n)) dove O(fj(n)) (j=1..m) è la complessità computazionale di

ciascuna istruzione che compone il blocco. Ovviamente la complessità computazionale effettiva verrà calcolata applicando la regola della somma.

8. La complessità computazionale deve essere sempre computata nel caso peggiore. Ciò vuol dire che se si scopre che la complessità computazionale dipenda, oltre che da n, anche da altre variabili o da particolari valori dei dati in ingresso, bisogna considerare, per tali variabili e valori, i valori che rendono massima la complessità computazionale.

5.1.Esempio 1

Sia dato il programma di ordinamento BubbleSort in linguaggio C:

```
(1)          for (j=0; j<n-1 && !ordinato; j++) {
(2)                 ordinato=1;
(3)                 for (i=n-1; i>j; i--)
(4)                       if (v[i]<v[i-1]) {
(5)                             temp=v[i];
(6)                             v[i]=v[i-1];
(7)                             v[i-1]=temp;
(8)                                                ordinato=0;
(9)                                   }
(10)         }
```

La complessità del programma coincide con quella del ciclo for alla linea (1), visto che il corpo del ciclo for si estende fino alla linea (10). Per calcolare la complessità computazionale del ciclo for alla riga (1) applichiamo la regola 4) del capitolo 5, ottenendo che tale complessità è data da:

$O(g1(n)\cdot f1(n))$.

In tal caso $g1(n)=(n-1)$, perché dobbiamo metterci nel caso peggiore e dobbiamo supporre che la variabile ***ordinato*** venga sempre messa a 0 alla riga (8).

$O(f1(n))$ è la complessità computazionale del blocco che si estende dalla riga (2) alla riga (9), che dobbiamo calcolare. Essa è data da:

$O(f1(n))=O(1)$ (assegnazione linea (2)) + $O(f2(n))$ (complessità del secondo ciclo for).

Dobbiamo dunque calcolare $O(f2(n))$, che è la complessità computazionale del secondo ciclo for alla riga (3). Dobbiamo applicare ancora la 4) del capitolo 5. Risulta allora:

$O(f2(n))=O(g2(n)\cdot f3(n))$,

$g2(n)=n-1-j$; visto che j varia da 0 a n-2, dobbiamo scegliere il caso peggiore, ovvero il valore di j che rende massima la g2(n), ossia j=0, per cui risulta $g2(n)=n-1$.

f3(n) è relativa all'istruzione if alla linea (4), per cui si applica la regola 5) vista precedentemente. Visto che l'operazione fatta nel caso in cui la condizione if vale, è una sequenza di 4 assegnamenti (linee (5-8)), si ha che:

$O(f3(n))=O(1)+ O(1)+ O(1)+ O(1)=O(1)$

ottenuta applicando la regola della somma.

Da ciò si ricava che:

$O(f2(n))=O((n-1)\cdot 1)=O(n-1)$

Applicando la regola transitiva, si ottiene:

$O(f2(n))=O(n)$

Possiamo a questo punto calcolare la:

$O(f1(n))=O(1)+O(n)=O(n)$, per la regola della somma.

Dunque la complessità computazionale dell'intero algoritmo sarà, $O((n-1)\cdot n)=O(n^2-n)$. Applicando la regola transitiva si ottiene una complessità $O(n^2)$.

5.2.Esempio 2

Sia dato il programma in C:

```
(1)     for (i=2; i<n; i++)     {
(2)           j=i;
(3)           printf("%d", j);
(4)           while (!(j%2)) {
(5)                 j/=2;
(6)                 printf("%d", j);
(7)           }
        }
```

Alla riga (1) vi è un ciclo for, per cui applichiamo la regola 4) del capitolo 5. Come prima anche in tal caso il ciclo for ha un corpo che si conclude alla fine del programma. La complessità computazionale del ciclo for, e dunque dell'intero programma, è data da:

$O(g1(n)\cdot f1(n))$.

In tal caso g1(n)=n-2, mentre O(f1(n)) è la complessità computazionale del blocco da (2) a (7), che dobbiamo calcolare. Essa è data da:

O(f1(n))=O(1) (assegnazione linea (2)) + O(1) (istruzione printf alla linea (3))+O(f2(n)) (complessità comando while alla linea (4)).

Dobbiamo dunque calcolare O(f2(n)), che è la complessità computazionale del comando while alla linea (4), che ha un corpo comprendente le linee (5) e (6). Dobbiamo applicare in questo caso la regola 3 vista precedentemente. Risulta allora:

O(f2(n))=O(g2(n)·f3(n)),

g2(n) specifica il limite superiore del numero di iterazioni, mentre f3(n) è la complessità del corpo del ciclo while.

Come si vede dal programma, il limite sul numero di iterazioni dipende da j(che è posto uguale a i prima del ciclo while). Per ciascun valore di j, il limite superiore di iterazioni è ottenibile nel caso in cui il numero j sia una potenza di 2, ossia 2^k. In tal caso infatti è necessario dividere j per 2 fino a quando si riduce a 1. Sia k il numero di iterazioni in tal caso, ossia k rappresenta il numero di iterazioni massimo per ciascun valore di j. Come detto, tale valore massimo si ha quando j è una potenza di 2, ossia $j=i=2^k$; dunque si ha k=log j= log i. Risulta allora g2(n)=log i.

Per quanto riguarda f3(n), si ha

O(f3(n))=O(1) (relativa all'assegnazione alla linea (5))+O(1) (relativa al comando printf alla linea (6))=O(1)

Da ciò si ricava che:

O(f2(n))=O((log i) ·1)=O(log i)

che dipende dal valore di i. Visto che a noi interessa un limite superiore della complessità computazionale, tale limite si ottiene quando i=n-1, per cui:

O(f2(n))=O(log (n-1))

Poichè log (n-1) è O(logn), per la legge transitiva, risulta una complessità di:

O(f2(n))=O(log n)

Da ciò si può ricavare:

O(f1(n))=O(1)+O(1)+O(log n)=O(log n), per la regola della somma

e dunque la complessità computazionale dell'intero algoritmo sarà, O((n-2)·log n). Applicando la legge transitiva, si ottiene una complessità: O(n·log n).

6.Procedura Generale per il Calcolo della Complessità Computazionale in Presenza di Procedure non Ricorsive

Fino ad ora sono stati considerati programmi non contenenti procedure o funzioni. Adesso considereremo tale caso, limitandoci alle procedure o funzioni non ricorsive.

La procedura generale che deve essere seguita per il calcolo della complessità computazionale, consiste nei seguenti passi:

1. si analizza ciascuna procedura o funzione fino ad individuare quella o quelle che non presentano chiamate ad altre procedure o funzioni al loro interno.
2. per ciascuna procedura o funzione che non presentano chiamate ad altre procedure o funzioni, devono essere applicate le regole viste nella sezione 5. In tal modo viene determinata la complessità computazionale delle procedure o funzioni considerate.
3. devono, a questo punto, essere considerate tutte le altre procedure e funzioni che chiamano direttamente le procedure o le funzioni di cui è stata determinata la complessità, come detto al passo precedente. Per ciascuna di tali procedure e funzioni viene calcolata la complessità computazionale, sostituendo a ciascuna chiamata di procedura o funzione la relativa complessità computazionale.
4. il procedimento espresso al passo 3, deve essere applicato a tutte le procedure e funzioni contenenti procedure e funzioni, di cui si è calcolata già la complessità computazionale. Tale procedimento termina quando è stata calcolata la complessità di tutte le procedure e funzioni presenti nel programma principale.
5. la complessità computazionale del programma main(), viene calcolata sostituendo a ciascuna chiamata di procedura o funzione la relativa complessità computazionale, già calcolata ai passi precedenti.

6.1.Esempio 1

Si consideri il seguente programma in linguaggio C:

```
#include<stdio.h>
#include<malloc.h>
#include "lista.h"

list l;
position p;
tipobaseList x;

main()
{
(1)  p=First(l);
(2)  while (p!=End(l)) {
(3)           x=Retrieve(l,p);
(4)           printf("%d",x);
(5)           p=Next(l,p);
     }
}
```

dove si suppone che **tipobaseList** sia implementato tramite un **int** e dove list.h contiene l'implementazione della lista tramite puntatori e di tutte le sue funzioni primitive.

Il programma main() chiama le procedure/funzioni First, End, Retrieve, Next appartenenti alla libreria list.h. Prima di poter calcolare la complessità del programma main() è necessario, dunque, calcolare al complessità di tali procedure. **Ovviamente tale complessità dipende dall'implementazione della lista;** supponiamo che tale implementazione sia realizzata con puntatori (infatti nel listato abbiamo inserito #include<malloc.h>).

Data la lista l, è ovvio che in tal caso la complessità computazionale deve essere calcolata in funzione della dimensione della lista, ossia del numero di elementi in essa contenuti, che indicheremo con **n**.

E' facile verificare (si lascia come esercizio tale verifica) che:

- La procedura **First** ha complessità O(1).
- La procedura **End** ha complessità O(n), dove n è la dimensione della lista, ossia il numero di elementi che la compone, come già detto.
- La procedura **Retrieve** ha complessità O(1)
- La procedura **Next** ha complessità O(1).

La complessità computazionale del main() si calcola sostituendo alla chiamata di ciascuna procedura, la rispettiva complessità. Ne segue che la complessità del main() è:

Riga 1. O(1)+O(1), perché c'è una assegnazione e la chiamata alla funzione First

Riga 2. La complessità del ciclo while è $\sum_{g(n)}(O(f1(n)+O(f2(n))+O(1))$, dove O(f1(n)) è la complessità della condizione di fine ciclo (p!=End(l)), O(f2(n)) è la complessità del corpo del ciclo e O(1) è la complessità del ritorno a inizio ciclo; invece g(n) è il numero di volte che viene eseguito il ciclo. E' chiaro che O(f1(n)) è pari a O(1)+O(n), perché è la complessità dell'istruzione p!=End(l), e dunque vi è un confronto e la chiamata alla funzione End. O(f2(n)) è pari alla somma delle complessità delle istruzioni alla riga (3), pari a O(1)+O(1) (chiamata a Retrieve e assegnazione), alla riga (4), pari a O(1), e alla riga (5), pari a O(1)+O(1) (chiamata a Next e assegnazione). Applicando la regola della somma, la complessità del corpo del ciclo è O(1). Dunque si ottiene che la complessità del ciclo while è $\sum_{g(n)}(O(n)+O(1)+O(1)) = \sum_{g(n)}(O(n))$, applicando la regola della somma. Nel caso peggiore g(n) è n, dunque la complessità diviene $O(n^2)$.

Applicando la regola della somma alle righe 1 e 2, si ottiene che la complessità computazionale del main() è $O(n^2)$.

Si consideri adesso lo stesso programma in linguaggio C, ottenuto facendo una semplicissima modifica alle linee (1), (2) che adesso diventano (1), (2) e (3). Il programma è equivalente a quello precedente, ossia compie le stesse funzioni; l'unica differenza è che il valore della funzione End(l) viene calcolato una sola volta (tanto non cambia mai !) e viene assegnato alla variabile u, di tipo position.

```
#include<stdio.h>
#include<malloc.h>
#include "lista.h"

list l;
position p,u;
tipobaseList x;

main()
{
(1)  p=First(l);
(2)  u=End(l);
(3)  while (p!=u) {
(4)           x=Retrieve(l,p);
(5)           printf("%d",x);
(6)           p=Next(l,p);
     }
}
```

Se si ripetono gli stessi passi illustrati prima, si può verificare che adesso la complessità computazionale è scesa a O(n). Questo dimostra che il calcolo della complessità computazionale è

importantissimo, in quanto permette di individuare delle soluzioni algoritmiche che possono ridurre di molto i tempi di calcolo. Nell'esempio presentato il calcolo della funzione End(l) veniva ripetuto ad ogni ciclo while; ma ciò era inutile in quanto il valore fornito dalla funzione End(l) non cambia ad ogni ciclo, in quanto la lista non viene modificata ma solo visitata. Dunque in tal caso si era commesso un errore logico, una ingenuità di programmazione che però aveva comportato una elevatissima complessità computazionale ($O(n^2)$), che è stata ridotta a O(n).

6.2.Esempio 2

Si consideri il seguente programma in linguaggio C:

```
#include<stdio.h>
#define n 10
int vettore[n];

void riempi(int v[], int dim)
{
        int i;

        for (i=0; i<dim; i++) {
              printf("Inserisci l'elemento di indice %d ",i);
              scanf("%d",v+i);
              fflush(stdin);
 }
}

main()
{
          int i;
(1)       riempi(vettore,n);
(2)       for (i=0; i<n; i++)
(3)           printf ("Elemento di indice %d ", i, " = %d ", vettore[i]\n");
}
```

Il programma main() chiama la procedura riempi(int v[], int dim). Prima di poter calcolare la complessità del programma main() è necessario, dunque, calcolare al complessità della procedura riempi().

La procedura riempi(), non contiene alcuna chiamata ad altre procedure o funzioni. Si noti che al fine del calcolo della complessità computazionale della procedura riempi(), è importante mettere in evidenza che essa realizza il riempimento di un generico vettore di dimensione dim; a tale parametro formale, dim, viene fatto corrispondere il valore n, nella chiamata del main() alla riga (1). Dunque la complessità computazionale della procedura riempi() può essere calcolata direttamente in funzione di n, visto che dim coincide con n come già detto. Applicando le regole descritte nella sezione 5, si ottiene che la complessità computazionale di tale procedura è O(n).

La complessità computazionale del main() si calcola sostituendo alla chiamata della procedura riempi() alla riga (1) la complessità O(n). Ne segue che la complessità del main() è:

O(n)+O(n), che sono le complessità delle istruzioni alle righe 1,2 rispettivamente. Applicando la regola della somma, si ottiene che la complessità computazionale del main() è O(n).

7.Procedura Generale per il Calcolo della Complessità Computazionale in Presenza di Procedure Ricorsive

Al fine di determinare il tempo di esecuzione delle procedure ricorsive si applica una strategia che si articola in due passi:

1. definizione di una **Relazione di Ricorrenza**;
2. risoluzione della relazione di ricorrenza.

7.1.Definizione della Relazione di Ricorrenza.

Una relazione di ricorrenza è composta da:

- Base
- Induzione

Data una procedura P ricorsiva, il calcolo della Base e della Induzione viene realizzato formalizzando il tempo di esecuzione $T_P(n)$ della procedura, nella seguente maniera:

1. Si analizzano tutte le istruzioni che compongono la procedura P e si determinano i relativi tempi di calcolo. Per tutte le istruzioni per cui si può calcolare la complessità computazionale in termini di O(f(n)), i tempi di calcolo verranno espressi in funzione di tale notazione. Essendo la procedura P ricorsiva, una o più istruzioni che la compongono sarà costituita da una chiamata alla stessa procedura P; solo per tali casi, il tempo di calcolo di ciascuna chiamata a P viene momentaneamente lasciato irrisolto ed indicato con $T_P(k)$, dove k è la dimensione dei dati in ingresso alla attuale chiamata ricorsiva. La dimensione k risulta, in genere, inferiore a n, allo scopo di garantire la fine della ricorsione; si pensi, ad esempio, ad un algoritmo di ordinamento ricorsivo del tipo Quicksort, dove ad ogni ricorsione viene considerato un vettore da ordinare di dimensione sempre più piccola. In tal caso ciascuna ricorsione termina quando il vettore da ordinare ha dimensione 1.
2. Si individua il valore di n, indicato con n_0, per il quale la ricorsione ha termine. In corrispondenza di tale valore si determina il valore assunto da $T_P(n_0)$, sulla base dell'analisi di tutte le istruzioni che compongono la procedura P. Tale valore costituisce la **base** della relazione di ricorrenza. **E' necessario puntualizzare che molto spesso la ricerca della base, ossia del valore n_0 di n che determina la fine della ricorsione, viene realizzata in modo più**

approssimato, al fine di trovare un valore di n_0 che risulti più utile nella risoluzione della relazione di ricorrenza. Ad esempio, molto spesso viene scelto come n_0 non il valore esatto che determina la fine della ricorsione, ma un valore tale che dia la sicurezza che dopo un numero finito di ricorsioni, la ricorsione stessa termini. Ciò verrà spiegato in dettaglio negli esempi mostrati successivamente.

3. L'**induzione** della relazione di ricorrenza è ottenuta considerando il generico valore di n per cui vi è ricorsione, e calcolando quanto vale $T_P(n)$ in tal caso, utilizzando sempre l'analisi dei tempi di esecuzione di tutte le istruzioni che compongono la procedura, ottenuta al passo 1.
4. Nelle relazioni precedentemente individuate (Base ed Induzione della relazione di ricorrenza) si sostituisce la notazione O(f(n)) con una costante moltiplicata per f(n).

7.1.1. Esempio del Calcolo della Relazione di Ricorrenza: Visita di una Lista.

Si consideri la seguente procedura ricorsiva che effettua una visita in una lista, nell'arco di posizioni comprese tra p e u (ad esempio si supponga di passare i valori di First(l) e End(l) al posto di p e u; in tal caso si avrebbe la visita completa della lista).

```
void visita(list l, position p, position u)
{
  tipobaseList x;

(1)  if (p!=u) {
(2)        x=Retrieve(l,p);
(3)        printf("\nL'elemento e' %d ",x);
(4)        p=Next(l,p);
(5)        visita(l,p,u);
(6)  }
}
```

Sia $T_P(n)$ il tempo di esecuzione della procedura ricorsiva visita(), dove n è la dimensione dei dati in ingresso, ossia il numero di elementi contenuti nella lista e compresi tra le posizioni p e u (si ricordi che la visita è delimitata dalle posizioni p e u). Determiniamo la relazione di ricorrenza, attraverso i passi prima descritti:

1. La procedura visita() si compone di una istruzione if alla linea (1). Nel caso in cui la condizione (p!=u) è falsa, la ricorsione termina; nel caso contrario, viene eseguito il blocco di programma (2)-(5). Le linee (2), (3) e (4) hanno complessità O(1), mentre alla linea (5) si ha la chiamata alla stessa procedura, ma con la posizione p avanzata di una posizione (tramite l'istruzione (4)). Ciò significa che ciascuna ricorsione considera un vettore che ha un numero di elementi pari a quelli considerati alla ricorsione precedente meno 1; per tale motivo, la chiamata alla linea (5) può essere sostituita da $T_P(n-1)$.

2. **Base:** La ricorsione termina quando quando p==u; in tal caso la dimensione del vettore può essere assunta pari a 0, visto che non ci sono altri elementi da visitare. In tal caso, risulta $T_P(0)=O(1)$, che è la complessità relativa al calcolo dell'espressione dell'if alla riga (1).
3. **Induzione:** se n è tale che vi è ricorsione (ossia p!=u), l'espressione di $T_P(n)$ è data dalla complessità del comando if alla riga (1), ossia pari a: O(1) (riga 1)+ O(1) (riga 2)+O(1) (riga 3)+ O(1) (riga 4) +T_P (n-1), ossia O(1)+ T_P (n-1)
4. Sostituendo a O(1)=a nella base e O(1)=b nell'induzione, si ottiene:

   BASE : T_P (0)=a;

   INDUZIONE: T_P (n)=b+T_P (n-1)

7.1.2.Esempio del Calcolo della Relazione di Ricorrenza: Ricerca Binaria.

Si consideri il seguente algoritmo di ricerca binaria:

```
int RicercaBinaria (tipobase v[], int inf, int sup, tipobase x)
{
      int m;
(1)   if (inf<=sup) {
(2)         m=(inf+sup)/2;
(3)         if (v[m]==x) return(1);
(4)         if (v[m]>x) return(RicercaBinaria(v,inf,m-1,x));
(5)         else return(RicercaBinaria(v,m+1,sup,x));
(6)   } else return(0);
}
```

Sia $T_P(n)$ il tempo di esecuzione della funzione ricorsiva RicercaBinaria, dove n è la dimensione dei dati in ingresso, ossia la dimensione del vettore v, cioè n=sup-inf+1. Determiniamo la relazione di ricorrenza:

5. La funzione RicercaBinaria si compone di una istruzione if alla linea (1). Nel caso in cui la condizione (inf<=sup) è falsa, l'istruzione alla linea (6) impiega un tempo O(1). Nel caso contrario, viene eseguito il blocco di programma (2)-(5). Le linee (2) e (3) hanno complessità O(1), mentre alle linee (4) e (5) si hanno le chiamate alla stessa procedura, ma con parametro n/2, in quanto in ciascuna delle due chiamate viene considerato solo metà del vettore originario. Ciascuna delle due chiamate alle linee (4) e (5) può essere sostituita da $T_P(n/2)$.
6. **Base:** La ricorsione termina quando quando inf>sup; in tal caso la dimensione del vettore può essere assunta pari a 0, visto che i due estremi si sono invertiti (se i due estremi inf e sup fossero coincidenti, allora la dimensione del vettore delimitato da inf e sup, sarebbe stata 1). In tal caso, risulta $T_P(0)=O(1)$, che è la complessità relativa all'if alla riga (1) e al return alla riga (6).
7. **Induzione:** se n è tale che vi è ricorsione (ossia n è uguale o maggiore di 1), il tempo di esecuzione della procedura è dato dalla somma di: O(1) (espressione riga 1) + O(1) (riga

2)+O(1) (riga 3) + O(1) (espressione if riga 4) +max(T_P(n/2), T_P(n/2)). Dunque si ha che l'espressione di T_P(n) è O(1)+T_P (n/2)

8. Sostituendo a O(1)=a nella base e O(1)=b nell'induzione, si ottiene:

   BASE : T_P (0)=a;

   INDUZIONE: T_P (n)=b+T_P (n/2)

7.2.Risoluzione della Relazione di Ricorrenza:Sostituzioni Successive

Una tecnica per risolvere la relazione di ricorrenza è per sostituzioni successive. Per spiegare tale metodo facciamo uso della relazione di ricorrenza:

- Base: T_P(1)=a
- Induzione: T_P (n)=b+T_P (n-1)

e consideriamo solo l'induzione:

$$T_P (n)=b+T_P (n-1) \qquad (1)$$

In tale relazione, sostituiamo a n il valore n-1, ottenendo:

$$T_P (n-1)=b+T_P (n-2)$$

Sostituiamo a n il valore n-2, nella relazione (1), ottenendo:

$$T_P (n-2)=b+T_P (n-3)$$

Sostituendo le espressioni ottenute nella relazione originale (1), si ha:

$$T_P (n)=b+b+b+T_P (n-3)$$

In generale, dunque, si ha che:

$$T_P (n)=i\cdot b+T_P (n-i), \text{ dove } 1\leq i<n$$

Nella relazione T_P (n)=i·b+T_P (n-i), si consideri i=n-1, ossia il valore di i per cui T_P (n-i) diviene pari al valore corrispondente alla base, T_P(1)=a. Dunque si ottiene, facendo le opportune sostituzioni:

$$T_P (n)=(n-1)\cdot b+T_P (1)=(n-1)\cdot b+a \text{ ossia } T_P (n) \text{ è } O(n)$$

7.2.1.Esempi di Risoluzione della Relazione di Ricorrenza:

Esempio 1

Si consideri la relazione di ricorrenza trovata per l'esempio 7.1.1:

- BASE : T_P (0)=a
- INDUZIONE:T_P (n)=b+T_P (n-1)

Consideriamo l'induzione:

$$T_P (n)=b+T_P (n-1) \qquad (2)$$

Sostituiamo a n il valore n-1, ottenendo:

$$T_P(n-1)=b+T_P(n-2)$$

Sostituiamo a n il valore n-2, nella relazione (2) ottenendo:

$$T_P(n-2)=b+T_P(n-3)$$

Sostituendo le espressioni ottenute nella relazione originale (2), si ha:

$$T_P(n)=b+b+b+T_P(n-3)=3\cdot b+T_P(n-3)$$

In generale, dunque, si ha che:

$$T_P(n)= i\cdot b+T_P(n-i), \text{ dove } 1\leq i\leq n$$

Come si ricorderà la base è relativa al valore n=0 ($T_P(0)=a$)); per poter sfruttare tale valore, si deve trovare un valore di i per cui n-i=0, ossia i deve essere uguale a n. Si consideri, allora, il valore i=n e lo si sostituisca nella:

$$T_P(n)= i\cdot b+T_P(n-i)$$

ottenendo:

$$T_P(n)= n\cdot b+T_P(0)= b\cdot n+a, \text{ ossia } T_P(n) \text{ è } O(n)$$

Esempio 2

Si consideri la relazione di ricorrenza trovata per l'esempio 7.1.2:

- BASE : $T_P(0)=a$
- INDUZIONE:$T_P(n)=b+T_P(n/2)$

Consideriamo l'induzione:

$$T_P(n)=b+T_P(n/2) \qquad (3)$$

Sostituiamo a n il valore n/2, ottenendo:

$$T_P(n/2)=b+T_P(n/4)$$

Sostituiamo a n il valore n/4, nella relazione (3) ottenendo:

$$T_P(n/4)=b+T_P(n/8)$$

Sostituendo le espressioni ottenute nella relazione originale (3), si ha:

$$T_P(n)=b+b+b+T_P(n/8)=3\cdot b+T_P(n/8)=3\cdot b+T_P(n/2^3)$$

In generale, dunque, si ha che:

$$T_P(n)= i\cdot b+T_P(n/2^i), \text{ dove } 1\leq i\leq \log_2 n$$

Come si ricorderà la base è relativa al valore n=0 ($T_P(0)=a$)); per poter sfruttare tale valore, si dovrebbe trovare un valore di i per cui $n/2^i=0$, ossia i dovrebbe essere tendente ad infinito, che è impossibile.

In casi come questi, la vera base non può essere utilizzata e devono essere considerati altri valori assimilabili a condizioni di base, e che comportano dei vantaggi nella risoluzione della relazione di ricorrenza. Ad esempio si consideri il valore n=1. La scelta della base in corrispondenza di n=1 è

corretta perché siamo sicuri che quando n=1, dopo un numero finito di ricorsioni, la ricorsione stessa termina. Infatti, il caso n=1 corrisponde a inf=sup. Si possono avere due casi. Il primo è che v[m], dove m=inf=sup, coincida con x. In tal caso la ricorsione termina. Nel caso in cui v[m]>x o v[m]<x, si ha l'ultima chiamata ricorsiva con n=0, alle linee (4) o (5). Dunque, in tal caso, la ricorsione terminerà. Per n=1 si ottiene che la BASE diviene $T_P(1)=O(1)$, ossia $T_P(1)=a$.

Si consideri il valore $i=\log_2 n$, tale che $n/2^i=1$ e lo si sostituisca nella:

$$T_P(n)= i\cdot b+T_P(n/2^i)$$

ottenendo:

$$T_P(n)= b\cdot\log_2 n +T_P(1)= b\cdot\log_2 n +a, \text{ ossia } T_P(n) \text{ è } O(\log n)$$

Si noti che il $\log_2$ è stato indicato come log n, visto che tutti i logaritmi sono equivalenti a meno di una costante di proporzionalità.

Esempio 3
Si consideri la relazione di ricorrenza:

- BASE : $T_P(1)=a$
- INDUZIONE:$T_P(n)=b\cdot n+2T_P(n/2)$

Si consideri sempre l'induzione:

$$T_P(n)=b\cdot n+2T_P(n/2) \qquad (4)$$

Sostituiamo a n il valore n/2, ottenendo:

$$T_P(n/2)=b\cdot n/2+2\cdot T_P(n/4)$$

Sostituiamo a n il valore n/4, nella relazione (4) ottenendo:

$$T_P(n/4)=b\cdot n/4+2\cdot T_P(n/8)$$

Sostituendo le espressioni ottenute nella relazione originale (4), si ha:

$$T_P(n)=b\cdot n+2(b\cdot n/2 +2\cdot (b\cdot n/4+2\cdot T_P(n/8)))=b\cdot n+b\cdot n+b\cdot n+8\cdot T_P(n/8)=3\cdot b\cdot n+2^3\cdot T_P(n/2^3)$$

In generale, dunque, si ha che:

$$T_P(n)= i\cdot b\cdot n+2^i\, T_P(n/2^i), \text{ dove } 1\le i\le\log_2 n$$

Si consideri il valore $i=\log_2 n$ e lo si sostituisca nella:

$$T_P(n)= i\cdot b\cdot n+2^i\cdot T_P(n/2^i)$$

ottenendo:

$$T_P(n)= b\cdot n\cdot\log_2 n +T_P(1)= b\cdot n\cdot\log_2 n +a, \text{ ossia } T_P(n) \text{ è } O(n\cdot\log n)$$