Appunti per il corso di

Algoritmi e Strutture Dati

1° anno - Laurea e Diploma in Informatica

Università di Genova

a.a. 1998/99

(Autore: prof. Gerardo COSTA)
(Docente: prof. Enrico PUPPO)

Versione Maggio 99

3ª Puntata

Complessità

Tecniche di programmazione

Per approfondimenti:

[CLR] Cormen, Leiserson, Rivest: Introduzione agli algoritmi, Vol 1, Jackson Libri

si prega di segnalare errori, punti oscuri,....

Indice

1 . Introduzione....3
1.1 . Calcolabilità e complessità....3
1.2 . A che serve lo studio della complessità di algoritmi/programmi ?....3
2 . Misure (o criteri) di complessità per algoritmi e programmi....5
3 . Funzioni complessità....7
3.1 . Dimensione dell'input (o dimensione del problema)....8
3.2 . Funzioni complessità - definizioni generali....9
3.3 . Complessità spazio....10
3.4 . Ulteriori osservazioni....10
4 . Ordini di grandezza: O, Ω e Θ....11
4.1 . O, Ω e Θ per funzioni di una sola variabile....11
4.2 . O, Ω e Θ per funzioni di piú variabili....15
4.3 . Significato delle stime con ordini di grandezza....16
5 . Un esempio di calcolo di complessità: insertion sort....18
6 . Regole empiriche per il calcolo di complessità....24
7 . Complessità della visita non ricorsiva di un albero....32
8 . Operazioni dominanti....35
9 . Complessità (intrinseca) dei problemi - algoritmi ottimi....37
10 . Ordini di grandezza e buon senso....40
11 . Complessità delle procedure e funzioni ricorsive....41
11.1 . Primo esempio....41
11.2 . Ricerca binaria....43
11.2.1 . Definizione induttiva di T(n) per l'algoritmo ricbin....46
11.3 . Un esempio con due chiamate ricorsive....47
11.3.1 Somme di potenze....49
11.4 Merge sort....50
11.5 . DFS ricorsiva su un albero....52
11.6 . Ancora un esempio di algoritmo su alberi....54
11.7 . L'esempio del fattoriale (e problemi di "dimensione")....55
11.8 . Numeri di Fibonacci....57
11.9 . Approccio bottom-up a Fibonacci....58
11.10 . Definizioni induttive e ricorsione....61
12 . Tecniche di programmazione....62
12.1 . Approccio collegato alla definizione induttiva dei dati....62
12.2 . Approccio ispirato all'induzione a passi....62
12.3 . Approccio ispirato all'induzione generalizzata - Divide et impera....63
12.4 . Tecnica della programmazione dinamica (dynamic programming)....67
12.4.1 . Coefficienti binomiali....67
12.4.2 . Il problema dello zaino (knapsack problem)....71
13 . Libri su algoritmi, strutture dati, complessità, ...75

1

. Introduzione

1.1 . Calcolabilità e complessità.

La (teoria della) ***calcolabilità***, che non vediamo, ha come problema di base:

> si può fare, si può risolvere, tramite un procedimento automatico, cioè con un algoritmo/programma ??

Il confine tra possibile ed impossibile corrisponde a quello tra finito ed infinito: è possibile quello che si può fare in un tempo finito, usando una quantità finita di "memoria", ... cioè usando una quantità finita di "risorse".

Quindi obiettivo primario è

> precisare i concetti di: algoritmo, funzione calcolabile con un algoritmo, problema risolubile con un algoritmo.

La (teoria della) ***complessità*** ha come problema di base:

> sapendo che si può fare, quanto è difficile/complesso, quanto costa farlo ??

Quindi obiettivo primario è precisare i concetti di:

1) complessità/costo di un algoritmo o programma
2) difficoltà intrinseca (o costo intrinseco) del calcolo di una funzione o della risoluzione di un problema.

Cominciamo dal punto 1); il 2) lo vedremo in seguito. Comunque, non vedremo la teoria della complessità, ma soltanto quella che si chiama ***complessità computazionale concreta***.

Nota. A seconda del contesto, programma ed algoritmo sono sinonimi, oppure indicano cose diverse.

Nei discorsi che faremo sulla complessità:

- *programma* = programma in un linguaggio di programmazione "vero";
- *algoritmo* = descrizione in linguaggio abbastanza comodo ed abbastanza preciso della soluzione del problema.

Quindi, dato l'algoritmo, arrivare ad un programma che risolve il problema dovrebbe essere solo una questione di pura programmazione.

1.2 . A che serve lo studio della complessità di algoritmi/programmi ?

In genere un problema ammette diverse soluzioni (qui intendiamo che la "soluzione" del problema sia, alla fine, un programma); come scegliere tra queste?

Ci sono almeno tre criteri:

1. semplicità, chiarezza, ...
2. generalità, riusabilità,....
3. efficienza: tempo di esecuzione, quantità di memoria necessaria,....

D’altra parte, ci sono diverse esigenze, che possono anche essere in contrasto: minimizzare i tempi di sviluppo del programma; minimizzare i costi dell’eventuale modifica, o aggiornamento, del programma; minimizzare i costi di esecuzione del programma.

Se dobbiamo scrivere un programma che verrà usato poche volte, su dati di dimensione ragionevole, e poi buttato via, la cosa da fare è ***minimizzare il costo dello sviluppo del programma***, scegliendo un algoritmo semplice, facile da capire e programmare. Notiamo che il costo dello sviluppo non comprende solo il tempo di pensare l’algoritmo e tradurlo in un linguaggio di programmazione, ma anche quello della verifica, del testing,.... In tutte queste fasi è un vantaggio avere un algoritmo semplice.

Se dobbiamo scrivere un programma che verrà piú tardi ripreso (magari da altri), modificato, ampliato,... è anche importante che sia scritto chiaramente e sia ben documentato; questo riduce i ***tempi di aggiornamento***.

Il discorso ***generalità***, ***riusabilità*** interseca i due precedenti; vediamo due esempi molto elementari.

Per risolvere il nostro problema dobbiamo ordinare un array; tanto vale scrivere bene una procedura di ordinamento; ci servirà ora, ma potremo riutilizzarla anche in altre situazioni (e allora guadagneremo tempo nella fase di sviluppo).

Per risolvere il nostro problema ci serve uno stack di caratteri; sappiamo anche che avrà sempre una lunghezza minore di un certo K. Possiamo implementare il nostro stack con un array (+ lunghezza) e lavorarci sopra, usando, nel programma, il fatto di sapere che è un array. Quando abbiamo finito tutto, il “cliente” arriva e ci dice che le cose sono cambiate un po’ .... come conseguenza non c’è più il limite di K. Allora l’implementazione ad array non va più bene, dobbiamo usare delle liste. Poichè il programma era stato scritto usando il fatto che lo stack era un array, bisogna ricontrollarlo tutto, per modificarlo. Se avessimo separato l’implementazione dello stack dal resto del programma (lavorando quindi sullo stack soltanto attraverso le procedure di base: empty, is-empty, pop, top, push) basterebbe modificare la parte di implementazione dello stack. Con un vantaggio ulteriore (e simile a quello che portava a scrivere una procedura di ordinamento): l’implementazione degli stack è un “pacchetto” a parte che possiamo riutilizzare anche in un altro momento.

Tutto quanto detto sopra riguarda i costi di sviluppo e manutenzione del nostro programma.

Se però il nostro programma dovrà essere usato moltissime volte (ad esempio, perchè è un foglio elettronico, o un text editor, o un compilatore, o un sistema operativo, o il sistema di gestione delle prenotazioni di una compagnia aerea) allora è necessario tener conto anche del ***costo di esecuzione*** del nostro programma (in termini di tempo macchina, di quantità di memoria necessaria, di quantità di traffico generato su una rete,.....)

Lo studio della complessità degli algoritmi e dei programmi fornisce un primo strumento per valutare il *costo di esecuzione*.

Non è l’unico; in particolare si utilizzano anche tecniche di *benchmarking* (banco di prova, cioè serie di test su input significativi) e di analisi del *profilo*, che però qui non discutiamo.

2

. Misure (o criteri) di complessità per algoritmi e programmi.

È piú semplice, per cominciare, riferirsi ai programmi.

- *Misure statiche o strutturali*; ad esempio: la lunghezza del programma, in termini di righe di codice, oppure il massimo livello di annidamento di cicli, procedure,.... . Si cerca quindi di valutare la complessità (nel senso intuitivo della parola) della struttura del programma. Queste misure hanno un interesse teorico; non le consideriamo e ci limitiamo a quelle dinamiche.
- *Misure dinamiche o computazionali*: si cerca di misurare la quantità di risorse utilizzate dal programma quando è in esecuzione.

***Risorse**:* come detto prima si possono considerare diversi tipi di risorse; in effetti ci si concentra su due: *tempo* e *spazio* (= memoria); anzi, nella maggioranza dei casi ci si limita al solo tempo; qui ci limiteremo, appunto a considerare il tempo.

Come misurare il tempo di esecuzione di un programma.

Vediamo un esempio: un programma ***P*** per ordinare (diciamo in modo crescente) una successione (finita) di interi.

Il tempo di esecuzione di ***P*** sull'input $i_1, ..., i_n$ dipende, ragionevolmente, almeno dai fattori seguenti:

1. n (ordinare 1000 elementi richiede piú tempo che ordinarne 10);
2. la successione $i_1, ..., i_n$ (alcuni metodi di ordinamento, es. insertion sort, hanno prestazioni migliori su input ordinati, o quasi ordinati, rispetto ad input molto disordinati);
3. l'*algoritmo astratto* sottostante (cioè il metodo di ordinamento scelto), sia ***A*** : un programma basato su un'idea furba è piú veloce (in genere) di uno basato su un'idea stupida;
4. il linguaggio di programmazione usato (collegato a bontà del compilatore - vedi sotto);
5. il sistema su cui si esegue il programma; sia dal punto di vista hardware (processore, bus, ...) che da quello software (sistema operativo, compilatore,...).

Nota: l'informazione data da 1) è contenuta in 2); vedremo presto perchè è utile citarli entrambi.

Il programmatore può agire solo sugli ultimi tre, anzi spesso solo sul 3) e, limitatamente, sul 4).

Rispetto a 4) e 5), il fattore 3) è "a monte": lo stesso ***A*** può essere implementato con diversi linguaggi e poi il programma fatto girare su macchine diverse. Vedremo che una scelta particolarmente infelice di ***A*** può non essere rimediabile anche con una felicissima di linguaggio e macchina.

In conclusione:

Molto spesso l'analisi si ferma a considerare solo 1), 2) e 3), o meglio, come vedremo: solo il fattore 3), in funzione di 1) e 2); in ogni caso, la *prima* analisi è di questo tipo; sulla base di questa analisi si può concludere, ad es, che un metodo di ordinamento è migliore/peggiore di un altro.

In situazioni particolarmente critiche (esempio: sistemi real-time, dove il tempo di risposta del programma è parte integrante della sua correttezza) si passa ad una successiva analisi, che tiene conto di 4) e 5); ad esempio, una volta codificato l'algoritmo in un programma, si valutano le

prestazioni reali (cioè in termini di secondi) di quest'ultimo, tramite test su una serie di input significativi (benchmarking).

Per altre considerazioni vedere oltre (Osservazioni, in fondo alla Sezione).

Qui ci limiteremo ad analisi che considerano solo 3) in funzione di 1) e 2).

In altri termini, riferendoci all'esempio, invece di considerare

(*)il tempo di esecuzione di ***P*** sull'input $i_1, ..., i_n$

considereremo

(**) il "tempo" di esecuzione di ***A*** sull'input $i_1, ..., i_n$

Subito c'è un problema: il *tempo* citato in (*) può essere un tempo fisico, mentre quello in (**) no; detto in un altro modo, in (**) non si misura il tempo, ma qualcos'altro. Anche per questo motivo, spesso si parla di *costo*, invece che di tempo.

In effetti, (**) diviene:

(***) il *numero di passi* necessari per eseguire ***A*** sull'input $i_1, ..., i_n$

Però ora bisogna capire cosa sono i passi !!

Un modo di far bene le cose è quello di fissare un "modello di calcolo" semplice, ad es. le RAM (Random Access Machines, un modello molto semplificato di macchina di von Neumann); in questi modelli è chiara la nozione di passo elementare. Allora si scrive ***A*** nel linguaggio corrispondente (nel modello RAM, ***A*** diventa un programma in un linguaggio di tipo assemblativo) e si contano i passi elementari .....

Non è assolutamente comodo seguire questa strada ed infatti nella maggior parte dei libri su algoritmi se ne segue un'altra (eccezione notevole: Knuth: The art of computer programming - 3 volumi, Addison Wesley 1973, testo sacro per l'argomento algoritmi e strutture dati).

In pratica, si considerano algoritmi formulati in linguaggio evoluto e si cerca di fare una *stima approssimata* (vedere oltre: *ordini di grandezza*) di quanti passi di calcolo sono necessari all'esecuzione, dove per *passo di calcolo* si intende l'equivalente dell'esecuzione di un istruzione elementare tipo LOAD, STORE, INCR, JUMP,.....

In effetti sappiamo che il tempo (fisico, questa volta) per eseguire un LOAD non è molto diverso da quello necessario ad eseguire INCR o JUMP; quindi, poichè si vogliono fare stime approssimate, è ragionevole *assumere per tutte un tempo uguale e costante*.

Quindi, per capire il costo di un algoritmo ***A*** (relativamente ad un dato input **i**), cioè quanti passi di calcolo sono necessari all'esecuzione di ***A*** su **i** , si utilizzano le nostre conoscenze su una possibile implementazione di ciascuna delle istruzioni del linguaggio evoluto usato per ***A***.

Tenuto conto del tipo di approssimazioni che faremo:

- alle istruzioni evolute che corrispondono a poche istruzioni macchina, ad esempio,

  se x e z sono variabili intere o reali e A è un array unidimensionale:

  x ← 77; z ← x∗x ; A[x-1] ← z - x ; scambia A[x] e A[x+1]

  e piú in generale ad istruzioni evolute che, *indipendentemente dal particolare input* considerato, corrispondono ad un numero *costante* di istruzioni macchina, ad esempio

per j = 1, 2, ..., 1000 : x ← x+1

verrà attribuito un *costo costante*;

- per le altre istruzioni, vedere oltre: *Regole empiriche per calcolare il costo delle istruzioni* (ad esempio per l'istruzione while cond do body si cercherà di capire quante volte si esegue il ciclo e quanto costa ciascuna esecuzione del body).

Vedremo poi che in molti casi si può seguire una strada ancora piú semplice, utilizzando la nozione di *operazione dominante* .

3 . Funzioni complessità

Limitiamoci sempre al tempo e ritorniamo all'esempio di ordinare una successione finita di numeri.
Volevamo calcolare il tempo di esecuzione di ***P*** sull'input $i_1, ..., i_n$ e siamo passati a calcolare il "tempo" (il costo) di esecuzione di ***A*** sull'input $i_1, ..., i_n$ espresso come "numero di passi di calcolo".
Al variare dell'input (nell'insieme INPUT_per_***A***), si ha una funzione (totale):

$\mathbf{Ti}_A$: INPUT_per_***A*** → **N**.

(totale perchè, ragionevolmente, l'algoritmo termina su tutti gli input).

Questa funzione non è facile da calcolare, anche in modo approssimato; il problema è che dati n numeri, e supponendoli distinti, ci sono n! possibili input composti da questi numeri e su ciascuno di essi il costo di ***A*** potrebbe essere diverso, anche a livello di approssimazioni (ad esempio, se ***A*** è basato sull'idea di inserimento diretto (insertion sort) allora il costo su $i_1, ..., i_n$ è proporzionale ad n se la sucessione $i_1, ..., i_n$ è già ordinata o quasi ordinata, è proporzionale ad n^2 se $i_1, ..., i_n$ è molto disordinata).

Inoltre non è molto interessante sapere quanto costa ordinare una particolare scelta di $i_1, ..., i_n$;
quello che interessa è sapere quanto costa ordinare n numeri (qualunque).
In altri termini ci interessa di piú una funzione (anche'essa totale) $\mathbf{T}_A$: **N** → **N**
t.c. $\mathbf{T}_A$ (n) = costo di ordinare una successione di n numeri.

A questo punto si ricade nel problema di prima: come si può definire $\mathbf{T}_A$ (n) se il costo dipende in realtà dall'input (e può andare da un costo proporzionale ad n ad uno proporzionale ad n^2) ?

Quello che si può fare è, se TEMPI(n) = {$\mathbf{Ti}_A$ **(i)** | **i** successione di n numeri} :

- considerare il *caso migliore*: $\mathbf{T}_A$ (n) = min TEMPI(n)
- considerare il *caso peggiore*: $\mathbf{T}_A$ (n) = max TEMPI(n)
- considerare il *caso medio*: $\mathbf{T}_A$ (n) = media TEMPI(n) - vedi sotto

Note.

- A rigori si dovrebbe scrivere $\mathbf{T}_{A}^{\,m}$, $\mathbf{T}_{A}^{\,p}$, $\mathbf{T}_{A}^{\,med}$ per distinguerle.
- L'insieme degli input è infinito, se prendiamo gli input in tutto **Z**, è finito se prendiamo gli input nell'insieme dei valori rappresentabili con 2/4/6/... bytes; comunque, nel caso che stiamo considerando, TEMPI(n) è un è un sottinsieme finito di **N** e quindi max e min esistono.
- Nell'ultimo caso si intende media probabilistica: se p_i è la probabilità che, tra tutti quelli di lunghezza n, l'input sia proprio **i**, allora $\mathbf{T}_A(n) = \sum_{\mathbf{i}} \{ p_i * \mathbf{Ti}_A(\mathbf{i}) \}$.

L'approccio del caso peggiore è quello piú seguito: per un algoritmo ***A*** si calcola sempre la funzione $\mathbf{T}_A$ nel caso peggiore; in alcuni casi (ad es. per provare a distinguere algoritmi che hanno un comportamento simile nel caso peggiore) si valuta anche la $\mathbf{T}_A$ nel caso migliore e/o, se è nota la distribuzione probabilistica degli input, la $\mathbf{T}_A$ nel caso medio.

Ritornando all'esempio, come si può stimare la $\mathbf{T}_A$ se non si conosce la $\mathbf{Ti}_A$?

Nel caso peggiore (e discorso analogo per il caso migliore), fissato n:

- spesso si può a ragionare assumendo un generico input di dimensione n; molto spesso si può almeno iniziare in questo modo;
- se non funziona (oppure, appena diventa necessario) si cerca di individuare un input **i** di lunghezza n, che sia *pessimo* (*ottimo*, se ci interessa il caso migliore), nel senso che il costo di ***A*** su **i** è proprio il massimo di TEMPI(n); allora $\mathbf{T}_A(n) = \mathbf{Ti}_A(\mathbf{i})$.

Tutto questo è facilitato dal fatto che <u>non si cerca di calcolare esattamente</u> $\mathbf{T}_A$, <u>ma solo di valutarne l'ordine di grandezza</u>.

Per poter passare alle definizioni di $\mathbf{T}_A$ per un generico algoritmo ***A***, bisogna introdurre la nozione di dimensione dell'input.

3.1 . Dimensione dell'input (o dimensione del problema)

Dato un generico algoritmo ***A*** (per risolvere un problema) bisogna trovare uno o piú parametri sufficienti a caratterizzare/individuare la dimensione degli input di ***A***.

(In effetti, la cosa non dipende tanto dall'algoritmo, quanto dal problema.)

Se ***A*** fosse un vero programma, il vero input sarebbe una stringa e allora la dimensione sarebbe, ovviamente, la lunghezza.

Il punto è che vogliamo rimanere a livello di algoritmo (piú o meno astratto) e non vogliamo complicarci la vita a capire come codificare i nostri input (anche questi astratti: es, una successione di numeri); inoltre quello che ci interessa è capire come vanno le cose in funzione di parametri piú sintetici, piú significativi, piú vicini al problema (ad es. il numero di elementi da ordinare, piuttosto che la lunghezza della stringa che li codifica (e che dipende dalla base in cui si scrivono i numeri, tiene conto anche dei separatori,....)).

Conseguenza: non abbiamo una definizione di dimensione dell'input e per ogni problema dobbiamo decidere quali parametri scegliere per caratterizzarla. Di solito, per fortuna, è abbastanza chiaro quali parametri scegliere.

Nota: i parametri che consideriamo sono sempre in **N**.

Esempi (seguendo lo schema problema ||→ elenco dei parametri):

1) ricerca di x in una successione di n numeri ||→ n
2) ricerca di x in una successione di n stringhe ||→ n, lng

   dove lng = lunghezza di x (infatti il costo del confronto tra due stringhe dipende dalla loro lunghezza; per la precisione, dalla lunghezza della piú corta; ovviamente dobbiamo supporre che ci siano stringhe di lunghezza maggiore od uguale ad x, altrimenti sarebbe inutile cercare .....)
3) prodotto di matrici quadrate di dimensione n ||→ n
4) prodotto di matrici $n \times p$ e $p \times q$ ||→ n, p, q
5) problema che riguarda un albero con n nodi e altezza h ||→ n, h

In 1) x viene ignorato, perchè la sua dimensione è trascurabile rispetto a quella della successione. In 2) si può ignorare lng se è piccola rispetto ad n, ma non se può essere arbitrariamente lunga. In 4) ci si può ridurre (a scapito della precisione) ad un solo parametro $m = \max\{n, p, q\}$. In 5) a volte basta considerare uno dei due parametri (anche perchè, come vedremo, non sono indipendenti).

Un altro aspetto è l'individuazione dell'input !

Spesso infatti si presenta l'algoritmo ***A*** sotto forma di procedura o funzione. In questi casi l'input corrisponde ai parametri attuali, o ad alcuni di essi, ma anche a delle variabili globali. Riferendosi a esempi che si vedranno in seguito:

- l'algoritmo di merge sort di solito è presentato come una procedura MS con tre parametri: A di tipo array [1..n] of integer, inf e sup di tipo integer. Per ordinare A si chiama MS(A, 1, n); quindi l'input corrisponde ad A;
- l'algoritmo di insertion sort di solito è presentato come una procedura con un parametro A di tipo array e di nuovo l'input corrisponde ad esso (o meglio al parametro attuale della chiamata); ma si può anche presentare come procedura senza parametri, riferendosi ad un array A globale; in questo caso l'input corrisponde ad una variabile globale.

Nei casi dubbi, il modo migliore per capire qual'è l'input è di ignorare il modo in cui viene presentato ***A*** e riferirsi direttamente al problema.

3.2 . Funzioni complessità - definizioni generali

Senza tanti discorsi applichiamo la schema visto sull'esempio dell'ordinamento al caso di un generico algoritmo ***A*** con insieme di input INPUT_per_***A***, supponendo di aver individuato k parametri per la dimensione. In quello che segue, quindi: $\mathbf{n} = (n_1, ..., n_k)$

Per tener conto di eventuale non-terminazione, la funzione $\mathbf{Ti}_A$ può anche assumere valore +8; quindi $\mathbf{Ti}_A$: INPUT_per_***A*** $\rightarrow \mathbf{N} \cup \{+8\}$.

Inoltre, nel caso generale, $\text{TEMPI}(\mathbf{n}) = \{\mathbf{Ti}_A(\mathbf{i}) \mid \mathbf{i}$ input di dimensione $\mathbf{n}\}$ potrebbe essere un insieme infinito, quindi dobbiamo usare inf/sup al posto di min/max.

Funzione (o meglio: funzioni) di *complessità-tempo* $\mathbf{T}_A : \mathbf{N}^k \rightarrow \mathbf{N} \cup \{+8\}$

- nel *caso migliore*: $\mathbf{T}_A(\mathbf{n}) = \inf \text{TEMPI}(\mathbf{n})$
- nel *caso peggiore*: $\mathbf{T}_A(\mathbf{n}) = \sup \text{TEMPI}(\mathbf{n})$

- nel *caso medio*: $\mathbf{T}_A(\mathbf{n})$ = media TEMPI(**n**)

  dove TEMPI(**n**) = {$\mathbf{Ti}_A(\mathbf{i})$ | **i** input di dimensione **n**} .

  Per fortuna, considereremo **solo algoritmi che terminano su tutti gli input** (eventualmente restringendo l'insieme INPUT_per_***A*** ai soli input accettabili) e TEMPI(**n**) sarà un insieme finito, quindi il tutto si semplifica a:
- $\mathbf{Ti}_A$: INPUT_per_***A*** → **N**
- $\mathbf{T}_A : \mathbf{N}^k \rightarrow \mathbf{N}$
  - nel *caso migliore*: $\mathbf{T}_A(\mathbf{n})$ = min TEMPI(**n**)
  - nel *caso peggiore*: $\mathbf{T}_A(\mathbf{n})$ = max TEMPI(**n**)
  - nel *caso medio*: $\mathbf{T}_A(\mathbf{n})$ = media TEMPI(**n**)

3.3 . Complessità spazio

Applicando un procedimento analogo alla risorsa "memoria" (cioè individuando una unità (piú o meno astratta) di memoria,....) si ottiene la

funzione (o meglio: le funzioni) di *complessità-spazio* $\mathbf{S}_A : \mathbf{N}^k \rightarrow \mathbf{N}$

Nel seguito: ci interesseremo esclusivamente alla complessità-tempo e, salvo che si precisi diversamente, ci riferiremo al caso peggiore.

3.4 . Ulteriori osservazioni

Alcuni dei passaggi fatti (da ***P*** passare ad ***A***, dall'input passare alla dimensione, considerare solo il tempo, il caso peggiore ...) lasciano un po' perplessi, almeno a prima vista. Alle motivazioni viste, possiamo aggiungere:

- le stime di complessità le useremo soprattutto per *confrontare* diversi algoritmi per il medesimo problema (piú che per fare considerazioni *in assoluto* su un algoritmo) o per confrontare la complessità di un algoritmo con la complessità intrinseca del problema (che definiremo);
- è importante fare delle stime su algoritmi astratti, proprio per poter scartare quelli che hanno complessità elevata, non giustificata da quella del problema; sarebbe seccante dover arrivare fino al codice Pascal o C per poi accorgersi che il programma è da buttare (del tutto analogo alla questione correttezza: si deve cercare di verificare la correttezza dell'idea di soluzione a livello astratto e poi controllare che la correttezza venga mantenuta nei vari passi che portano fino al programma concreto).

4 . Ordini di grandezza: O, Ω e Θ

Gli *ordini di grandezza* ci servono per fare una stima della complessità degli algoritmi (e poi anche dei problemi). Le definizioni che diamo si riferiscono a generiche funzioni, anche se poi le applicheremo a funzioni di complessità.

4.1 . O, Ω e Θ per funzioni di una sola variabile

Le funzioni che consideriamo, f,g,..., sono da **N** in **R** (l'insieme dei reali) e sono definite e non-negative da un certo punto in poi; chiariremo più avanti il motivo di usare **R**.

O(g) $=^{\text{def}}$ { f | esistono c ∈ **R**, c > 0, ed n_0 ∈ **N**, t.c. per ogni n = n_0 : f(n) = c g(n) }

Notare che: c ed n_0 dipendono da f; possiamo sempre prendere c razionale; inoltre si poteva anche scrivere: c f(n) = g(n) (ovviamente scegliendo un altro c .....); nella definizione data, c è tipicamente = 1; mettendolo a sinistra, invece, è tipicamente = 1.

Ω(g) $=^{\text{def}}$ { f | esistono c ∈ **R**, c > 0, ed n_0 ∈ **N**, t.c. per ogni n = n_0 : g(n) = c f(n) }

Θ(g) $=^{\text{def}}$ O(g) ∩ Ω(g).

Esempi: vedere piú avanti.

Si legge, si dice....

- il simbolo Ω si legge *omega*, Θ si legge *teta (o theta).*
- se f ∈ O(g) si dice che *f cresce al piú come g*, oppure che *g è un limite superiore per f* ;
  f ∈ O(g) generalizza f = g [cioè : f(n) = g(n), per ogni n]
- se f ∈ Ω(g) si dice che *f cresce almeno come g*, oppure che *g è un limite inferiore per f*;
  f ∈ Ω(g) generalizza f = g [cioè : f(n) = g(n), per ogni n]
- se f ∈ Θ(g) si dice che *f cresce come g*, oppure che *f è dello stesso ordine di grandezza di g*;
  f ∈ Θ(g) generalizza f = g

Osservazioni.

Useremo O( ), Ω( ) e Θ( ) a proposito di funzioni complessità.

Ad esempio: se ***A*** è l'algoritmo di ricerca binaria (in un array ordinato di n elementi) e $\mathbf{T}_A$ è la sua funzione complessità, $\mathbf{T}_A$: **N** → **N**, vedremo che $\mathbf{T}_A(n)$ = a * parte_intera($\log_2 n$) + b.

Calcolare esattamente le costanti a e b non è facile e non lo faremo; ci basterà capire che a > 0 e quindi il termine con il logaritmo compare davvero. Se non sappiamo che valore hanno, è inutile tirarsele dietro, come pure è seccante scrivere "parte_intera". Scriveremo quindi, semplicemente: $\mathbf{T}_A$ ∈ Θ(g) , dove g è tale che g(n) = $\log_2$ (n).

Il risultato è che $\mathbf{T}_A$ è caratterizzato da una espressione semplice, $\log_2 n$, che però ci fornisce una informazione sufficientemente precisa su come cresce $\mathbf{T}_A(n)$ al crescere di n. Ad esempio, ci permette di distinguere la complessità della ricerca binaria da quella dell'algoritmo ovvio, sia ***S***, di ricerca sequenziale; questo infatti ha una complessità che cresce linearmente al crescere di n: $\mathbf{T}_S(n)$ = c n + d , con c e d costanti, c >0.

Abbiamo detto che si usano le notazioni O( ), Ω( ) e Θ( ) per **semplificare**; quindi, con $\mathbf{T}_A$ come sopra, non ha senso scrivere : $\mathbf{T}_A$ ∈ Θ(h) , dove h è tale che h(n) = 17.35 $\log_2$ (n) - 48 , anche se, in base alle definizioni date, questo è corretto.

Le usiamo anche per fornire **informazioni significative e precise** su come cresce la complessità al crescere della dimensione dell'input; quindi, sempre con $\mathbf{T}_A$ come sopra, non ha senso scrivere:

$\mathbf{T}_A \in \Omega(h)$, dove $h(n) = 1$, oppure $\mathbf{T}_A \in O(h)$, dove $h(n) = n^{10}$; è giusto, in base alle deifnizioni, ma troppo impreciso.

Concludendo, nell'usare le notazioni O( ), Ω( ) e Θ( ) dobbiamo farci guidare da due principi: semplicità e precisione.

Sempre in base al criterio della precisione, cercheremo di dare delle stime in Θ( ), ripiegando su O( ) ed Ω( ) quando non ci riusciamo.

L'esempio della ricerca binaria serve anche a chiarire alcuni aspetti delle definizioni di O( ), Ω( ) e Θ( ):

- abbiamo usato codominio **R** e non **N** per non dover sempre passare alla "parte intera";
- il ruolo di n_0 è duplice: permette di trascurare i casi in cui funzioni come il logaritmo sono indefinite (per il logaritmo: n=0) e di trascurare i casi iniziali (input di piccole dimensioni);
- le costanti, c, d,..., ci permettono le semplificazioni discusse sopra, permettendoci di trascurare costanti moltiplicative e "termini di grado inferiore".

Proprietà di base (dimostrabili facilmente dalle def.ni; alcune analoghe a quelle di =, =, =)**:**

1. $\Theta(g) = \{ f \mid$ esistono c, d > 0 ed n_0 t.c. per ogni $n = n_0$: $c\, g(n) = f(n) = d\, g(n) \}$

2. $f \in O(f)$ e analogamente per Ω e Θ (possiamo chiamarla *riflessività*)

3. se $f \in O(g)$ e $g \in O(h)$ allora $f \in O(h)$; analogamente per Ω e Θ (possiamo chiamarla *transitività*)

4. $f \in O(g)$ sse $g \in \Omega(f)$

5. se $f = g$ allora $O(f) \subseteq O(g)$ e $\Omega(g) \subseteq \Omega(f)$

vale anche se $f(n) = g(n)$ da un certo punto in poi (per ogni $n = n_0$ per qualche n_0)

6. $f \in \Theta(g)$ sse $g \in \Theta(f)$ sse $\Theta(g) = \Theta(f)$

Collegamento con il concetto di limite.

Poichè consideriamo funzioni con dominio in **N**, il collegamento è quello con i limiti delle successioni (infatti: le funzioni da **N** in un insieme A coincidono con le successioni di elementi di A), ma sfruttando i teoremi che legano limiti di successioni e limiti di funzioni, si può anche ragionare sui limiti di funzioni da **R** in **R**.

È facile verificare (in base alla definizione) che

se $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = c$ (cioè, se il limite esiste ed è uguale ad una costante c)

allora: se c? 0 si ha $f \in \Theta(g)$ e quindi $g \in \Theta(f)$

se $c = 0$ si ha $f \in O(g)$ e quindi $g \in \Omega(f)$ ma $f \notin \Theta(g)$

(Notare che , essendo f e g non negative da un certo punto in poi, c = 0.)

Non vale il viceversa, in particolare perchè il limite può non esistere. Come (contro)esempio, basta considerare:

f(n) = 2n se n è pari e g(n) = 3n se n è pari
3n se n è dispari 2n se n è dispari

Chiaramente $f \in \Theta(g)$ ma il limite del rapporto non esiste.

Il legame con il concetto di limite, comunque fornisce uno strumento utile per ragionare in alcuni casi.

Proprietà della somma e "del massimo"

Indichiamo con $f+g$ e $\max\{f,g\}$ le funzioni tali che:

$f+g(n) = f(n)+g(n)$ $\max\{f,g\}(n) = \max\{f(n), g(n)\}$ allora

[somma] se $f_j \in O(g_j)$ per $j=1,2$ allora $f_1+f_2 \in O(g_1 + g_2)$

se $f_j \in \Omega(g_j)$ per $j=1,2$ allora $f_1+f_2 \in \Omega(g_1 + g_2)$

[max] $f+g \in \Theta(\max\{f,g\})$

Dimostrazione.

1) Supponiamo che $f_j \in O(g_j)$ per $j=1,2$. Allora esistono c_1, c_2, n_1, n_2 tali che
   per ogni $n = n_j$: $f_j(n) = c_j\, g_j(n)$ $j = 1,2$ e quindi
   per ogni $n = n_0$: $f_1(n) + f_2(n) = c\,(g_1(n) + g_2(n))$ se $n_0 = \max\{n_1, n_2\}$ e $c = \max\{c_1, c_2\}$.
2) Supponiamo che $f_j \in \Omega(g_j)$ per $j=1,2$. Allora esistono c_1, c_2, n_1, n_2 tali che
   per ogni $n = n_j$: $c_j\, f_j(n) = g_j(n)$ $j = 1,2$ e quindi
   per ogni $n = n_0$: $c\,(f_1(n) + f_2(n)) = g_1(n) + g_2(n)$ se $n_0 = \max\{n_1, n_2\}$ e $c = \max\{c_1, c_2\}$.
3) Chiaramente, per ogni n: $f(n) + g(n) = 2 \max\{f,g\}(n)$ quindi: $f+g \in O(\max\{f,g\})$.
   Ricordando che le funzioni sono non negative da un certo punto in poi, sia n_0 tale che
   per ogni $n = n_0$: $f(n) = 0$ e $g(n) = 0$ allora:
   per ogni $n = n_0$: $\max\{f, g\}(n) = f(n) + g(n)$ quindi $f+g \in \Omega(\max(f, g\})$.

Notazione semplificata

Invece di scrivere: $f \in \Theta(g)$, dove g è tale che $g(n) = exp$ in genere si scrive

$f(n) \in \Theta(exp)$ [alcuni scrivono addirittura: $f(n) = \Theta(exp)$]

ad esempio, invece di $\mathbf{T}_A \in \Theta(g)$, dove g è t.c. $g(n) = 2^n$

in genere si scrive $\mathbf{T}_A(n) \in \Theta(2^n)$

Nel caso dei **logaritmi** c'è un ulteriore semplificazione: si scrive $\mathbf{T}_A(n) \in \Theta(\log n)$, senza specificare la base; infatti quando si usa Θ le costanti "non contano" e se a, b, x sono tali che i logaritmi sono definiti, si ha: $\log_a x = c \log_b x$, dove $c = \log_a b$.

Analogamente per O e Ω.

Esempi (usando la notazione semplificata).

Negli esempi supponiamo sempre che i coefficienti, le basi dei logaritmi,.... siano tali da rispettare le proprietà di definitezza e non-negatività. Inoltre; k, a, b, ... sono costanti, k è intero e pk(n) è un generico polinomio di grado k a coefficienti reali: $pk(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0$

- $\Theta(1) = \Theta(a)$ per ogni a strettamente positiva
- $pk(n) \in O(n^k)$ [vedere oltre per dimostrazione]
- $pk(n) \in \Omega(n^k)$ se $a_k > 0$ [vedere oltre per dimostrazione]

  quindi: $pk(n) \in \Theta(n^k)$ se $a_k > 0$
- $(\log n)^k \in O(n)$ per ogni k = 0 [vedere oltre per la dimostrazione]
- $(\log n)^a \in O(n)$ per ogni a = 0 [vedere oltre per la dimostrazione]
- $\log n^k \in \Theta(\log n)$ [infatti $\log n^k = k \log n$]
- $(\log n)^5 + n^3 \in \Theta(n^3)$ [usando la proprietà [**max**]]
- $\log(n^5 a^n) = \log n^5 + \log a^n = 5 \log n + n \log a \in \Theta(n)$
- $pk(n) \in O(a^n)$, per ogni k ed ogni $a > 1$
- $a^n \in O(b^n)$ e $a^n \notin \Omega(b^n)$ se $1 < a < b$
- $n! \in O(n^n)$ e $n! \in \Omega((n/2)^{(n/2)})$
- $\log (n!) \in \Theta(n \log n)$ [usando la riga precedente e le proprietà dei logaritmi]
- se $f(n) = n^2$ quando n è pari ed n^3 quando n è dispari allora $f(n) \in O(n^3) \cap \Omega(n^2)$ e non si può precisare meglio.

Alcune dimostrazioni

Sia $pk(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0$

$pk(n) \in O(n^k)$

Dobbiamo dimostrare che esistono $c > 0$ ed n_0 tali che:$pk(n) = c\, n^k$ per ogni $n = n_0$.

Se i coefficienti sono tutti nulli, allora il risultato è banale, altrimenti

sia $a = \max \{ | a_i | , i = 0, ..., k \}$ dove | | indica il valore assoluto; notare che $a > 0$;

ovviamente: $pk(n) = a\, n^k + a\, n^{k-1} + ... + a\, n + a$ e, se $n > 0$

$= a\, n^k + a\, n^k + ... + a\, n^k + a\, n^k = (k+1)\, a\, n^k$

quindi, per ogni $n > 0$: $pk(n) = (k+1)\, a\, n^k$

allora: $c = (k+1)\, a$ e $n_0 = 1$

se $a^k > 0$, allora $pk(n) \in \Omega(n^k)$

Dobbiamo dimostrare che esistono $d > 0$ ed n_0 tali che, per ogni $n = n_0$.

[1] $pk(n) = d\, n^k$ cioè:

[2] $(a_k - d)n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 = 0$

Se $k = 0$ allora la tesi è banale, con $d = a_k/2$ e per ogni n.

Altrimenti, chiamiamo $q(n)$ il polinomio a sinistra di = in [2]

sia $b = \max \{ | a_i | , i = 0, ..., k-1 \}$ notare che non si considera il coefficiente a_k

allora $q(n) = (a_k - d)n^k - b\, n^{k-1} - ... - b\, n - b$ e, se $n = 0$

$> (a_k - d)n^k - b\, n^{k-1} - ... - b\, n^{k-1} - b\, n^{k-1} = (a_k - d)n^k - k\, b\, n^{k-1}$

Per concludere, allora, basta vedere che esistono $d > 0$ ed n_0 tali che:

[3] $(a_k - d)n^k - k\, b\, n^{k-1} = 0$

Scegliamo $d = a_k / 2$; allora la [3] diventa

[4] $(1/2)\, a_k n^k - k\, b\, n^{k-1} = 0$

se $n > 0$ si può dividere tutto per n^{k-1} e moltiplicare tutto per 2, ottenendo

[5] $a_k\, n - 2kb = 0$

che è vera per ogni $n = 2kb/a_k$

$(\log n)^k \in O(n)$ se $k = 0$

Qui conviene ricorrere ai limiti delle corrispondenti funzioni su **R** :

$$\lim_{x \to +\infty} \frac{(\log x)^k}{x} = \text{(usando il teorema dell' Hopital)}$$

$$\lim_{x \to +\infty} \frac{(\log x)^{k-1}}{x^2} = \text{(iterando)}$$

$$\lim_{x \to +\infty} \frac{1}{x^{k+1}} = 0$$

Per essere precisi, quanto sopra vale se il logaritmo è in base *e* ; tuttavia $\log_a x = c \log_e x$, dove c è una costante.

$(\log n)^a \in O(n)$ se $a = 0$

Sia $k = 1 + \text{parte_intera}(a)$; allora $(\log n)^a < (\log n)^k$; poichè $(\log n)^k \in O(n)$ questo basta per concludere.

Classi notevoli di funzioni

- le funzioni in $O(1)$ sono dette f*unzioni costanti*; notare che $O(1)$? $O(0)$!
- le funzioni in $O(n)$ sono dette *funzioni (al piú) lineari*
- le funzioni in $O(n^k)$, per qualche k, sono dette *funzioni (al piú) polinomiali*
- le altre sono dette *sovrapolinomiali* o anche, sbrigativamente, *esponenziali*

4.2 . O, Ω e Θ per funzioni di piú variabili.

Diamo solo le definizioni per O( ), le altre sono analoghe. Le funzioni che consideriamo, f,g,..., sono da $\mathbf{N}^k$ in **R** e sono definite e non-negative da un certo punto in poi.

$O(g) =^{def} \{ f \mid \exists\, c \in \mathbf{R},\, c > 0, \text{ ed } \exists\, n_0 \in \mathbf{N} \text{ t.c. } \forall\, n_1, ..., n_k = n_0 :\ f(n_1, ..., n_k) = c\, g(n_1, ..., n_k) \}$

Esempi (sempre in notazione semplificata, ed usando 2 variabili: n e p):

- $17n^2p - 31np + 4np^2$ è in $\Omega(np)$, $\Omega(n^2p)$, $\Omega(np^2)$, $\Omega(n^2p + np^2)$
  ma anche in $O(n^2p + np^2)$ e quindi è in $\Theta(n^2p + np^2)$
- $n \log p$ è in $\Omega(n)$, $O(np)$ e ovviamente in $\Theta(n \log p)$

Estensioni della notazione

Alle volte si usano O, Ω e Θ anche quando le funzioni non sono esplicite (e con significato abbastanza ovvio).

Ad es. se n è il numero dei nodi ed h è l'altezza di un albero binario, allora si dice che

n è in $O(2^h)$ e h è in $\Omega(\log n)$;

se l'albero è completo, allora: n è in $\Theta(2^h)$ e h è in $\Theta(\log n)$.

4.3 . Significato delle stime con ordini di grandezza.

Supponiamo di avere un problema **PR** e tre algoritmi per risolverlo: A_1, A_2 ed A_3. Inoltre, per i=1,2,3, sia T_i la funzione (di un solo argomento: sia n) complessità-tempo (caso peggiore) associata ad A_i

Supponiamo che: $T_1(n) \in \Theta(n)$; $T_2(n) \in \Theta(n^5)$; $T_3(n) \in \Theta(2^n)$.

- Per n piccolo, le differenze non sono rilevanti (ad es. per n=3: n^5 è maggiore di 2^n); se ci interessano solo input con dimensione molto piccola, le indicazioni in ordine di grandezza non sono significative; bisogna fare analisi piú fini (ad es contare davvero il numero di passi).
- Per n sempre piú grande (cioè guardando il *comportamento asintotico*) le differenze si fanno sempre piú sensibili, annullando il peso delle costanti e dei termini di ordine inferiore nascoste dalle notazione Θ. Il primo algoritmo è migliore degli altri (nel caso peggiore; nel caso medio le cose potrebbero andare diversamente).

Per avere un'idea della diversità di comportamento di questi algoritmi, supponiamo che sia proprio: $T_1(n) = n$; $T_2(n) = n^5$; $T_3(n) = 2^n$; supponiamo inoltre che il tempo effettivo per eseguire "un passo" sia 1μs (cioè 10^{-6} secondi).

Allora abbiamo la seguente tabellina:

	costo	tempo su input di dim = 10	tempo su input di dim = 60
A_1	n	= 10^{-5} secondi	= $6 * 10^{-5}$ secondi
A_2	n^5	= 0.1 secondi	˜ 13 minuti
A_3	2^n	˜ 10^{-3} secondi	˜ 366 secoli

Per chiarire il fatto che l'algoritmo astratto può essere il fattore dominante per le prestazioni di un programma e pesare molto piú, ad esempio, che la velocità della macchina, supponiamo di comprare una macchina 1000 volte piú veloce della precedente; allora con l'algorimo A_3 e con input di dimensione 60, il tempo necessario diventa circa 36 anni. E` senza dubbio un bel progresso rispetto ai 366 secoli che servivano prima, ma ci vuole ancora molta pazienza ....

Vedendo le cose da un altro punto di vista, consideriamo la tabella:

	costo	vecchio computer	nuovo computer
A_1	n	N_1	$N'_1 = 1000\, N_1$

A2	n^5	N_2	$N'_2 \sim 4 N_2$
A3	2^n	N_3	$N'_3 \sim N_3 + 10$

Nella 3a colonna, N_1, N_2 ed N_3 rappresentano la dimensione massima dell'input che si riesce a gestire in un tempo di calcolo fissato, sia **t**, e usando il vecchio computer (non interessa qui conoscere i valori di N_i); nell'ultima colonna vediamo le dimensioni massime gestibili, sempre nel tempo **t**, avendo a disposizione la nuova macchina, 1000 volta piú veloce della precedente.

Per l'ultima riga il ragionamento è, se tp è il tempo di esecuzione di 1 passo sulla vecchia macchina:

$$tp * 2^{N_3} = \mathbf{t} = (tp / 1000) * 2^{N'_3} \sim (tp / 2^{10}) * 2^{N'_3} = tp * 2^{(N'_3 - 10)}$$

da cui si ottiene che $N'_3 \sim N_3 + 10$.

5

. Un esempio di calcolo di complessità: insertion sort

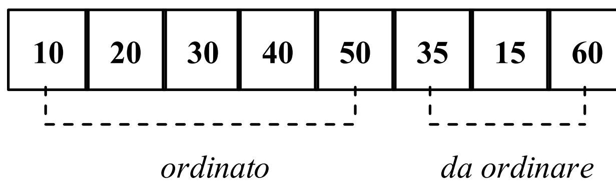
Vediamo su un esempio come si può calcolare la complessità di un algoritmo; successivamente daremo delle regole empiriche, ma abbastanza generali, che servono da guida in questi calcoli e permettono di semplificarli.

L'esempio è un metodo semplice per ordinare arrays: insertion sort, o inserimento diretto (bisogna però dire che tra i metodi semplici il migliore, dal punto di vista dell'efficienza, è quello che abbiamo visto per primo: selection sort).

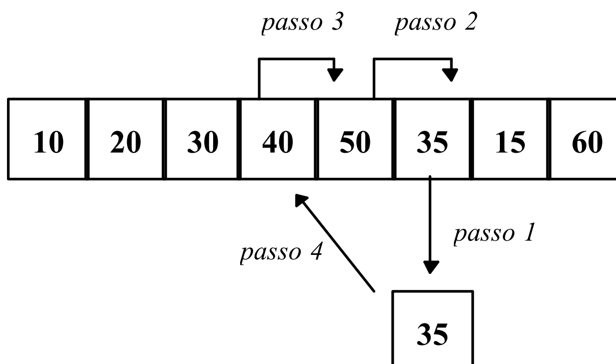
L'idea di base, per l'inserimento diretto, è illustrata in Figura 1.

Figura 1.

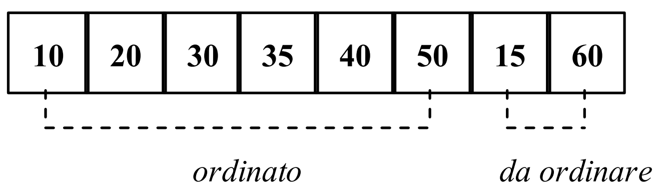
Supponiamo di aver già ordinato una parte dell'array:



Ora sistemiamo il primo degli elementi "fuori posto", cioè 35
(la casella in basso rappresenta una variabile ausiliaria):



Quindi, l'array diventa:



All'inizio, la parte ordinata è costituita da un solo elemento: il primo.

Insertion sort sotto forma di “pezzo di programma”.

Supponiamo di avere un programma in cui ci siamo definiti ed abbiamo “riempito” un array:

a : array [1 ... n] di interi.   Ora vogliamo ordinarlo; per farlo usiamo:

- j   per indicare l’elemento da “sistemare”
- i   per scorrere l’array da j-1 a 1
- temp   per salvare il valore dell’elemento da “sistemare”

Lo pseudo-codice per il pezzo di programma che ordina l’array a è:

```
per j = 2, 3, ..., n :    { temp ← a[j]
                            /* ora inseriamo a[j] al suo posto */
                            i ← j - 1
                            while i > 0 e a[i] > temp do {  a[i+1] ← a[i]
                                                            i ← i-1
                                                         }
                            a[i+1] ← temp
                          }
```

Nota: nella condizione del while, e è un and stile C : se (i > 0) è falso, non si passa a valutare la seconda parte della condizione (infatti a[i] non sarebbe definito).

Costo dell’algoritmo

Per valutare il costo dell’algoritmo, traduciamolo in un “diagramma di flusso”, traducendo il “per” ed il while; vedere Fig. 2.

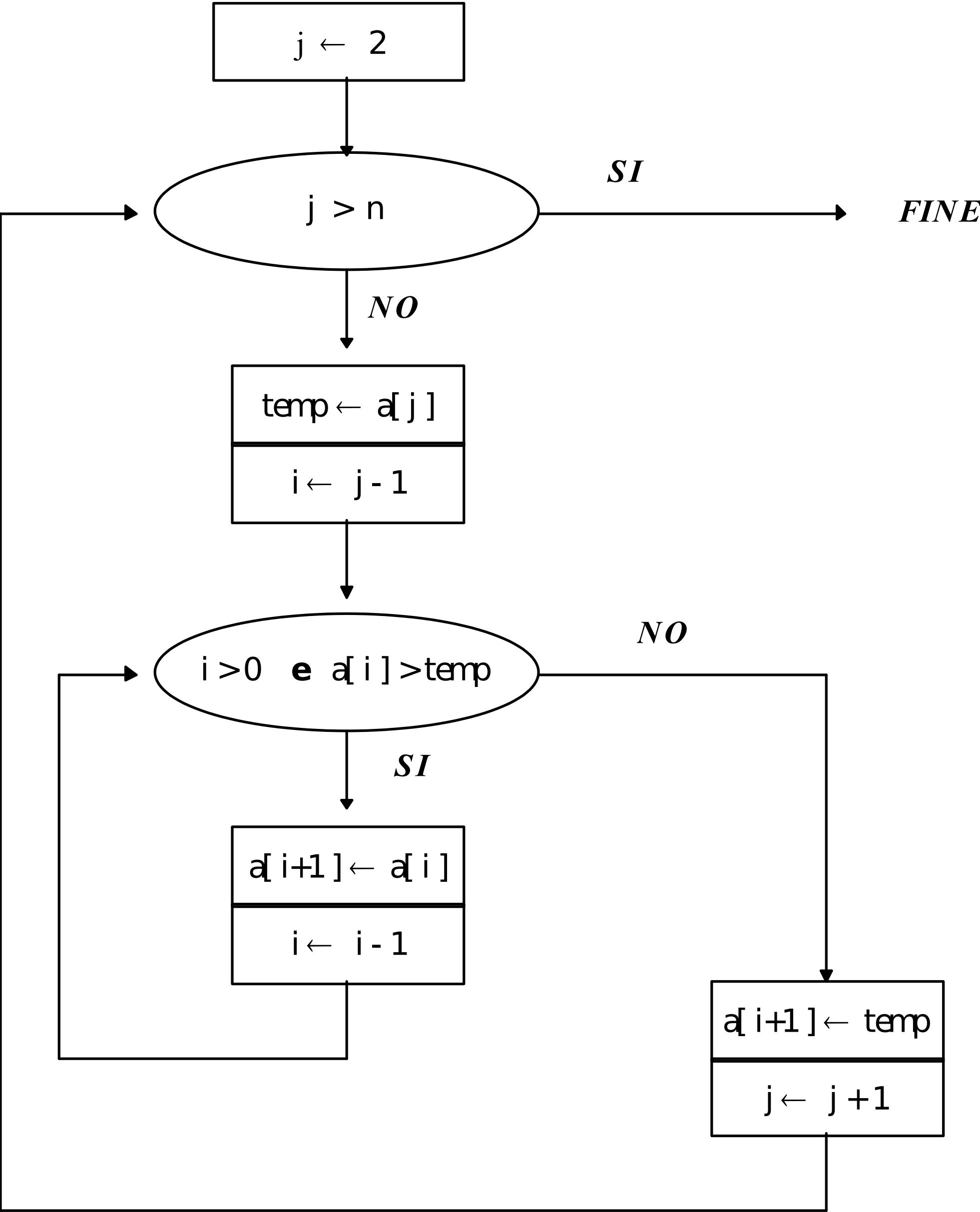
Ci interessa valutare il costo di tutto l’algoritmo in funzione di n (cioè il numero di elementi nell’array), che è un buon paramentro per caratterizzare la dimensione dell’input dell’algoritmo di ordinamento (che è appunto l’array).

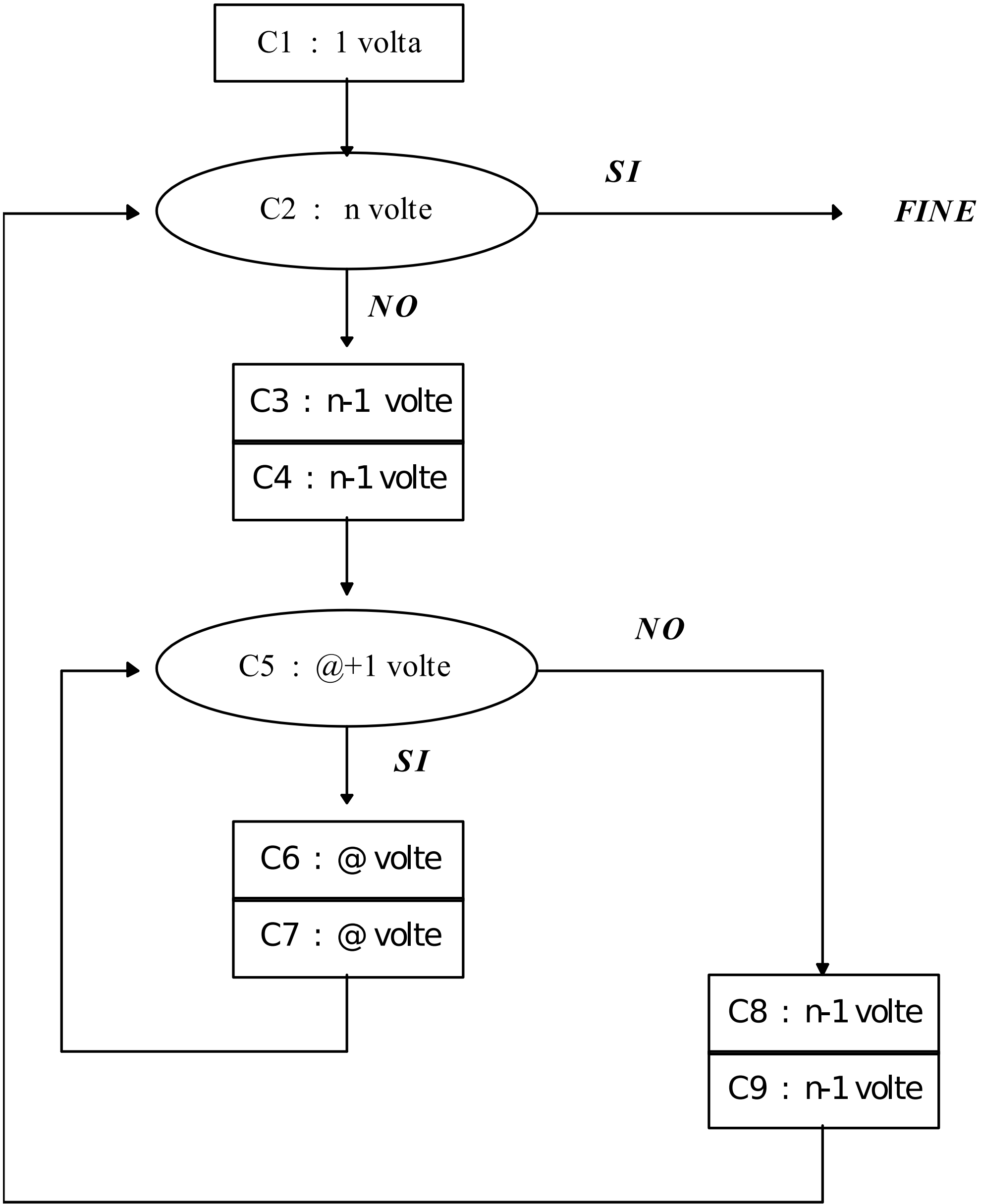
Il costo di eseguire ciascuna delle istruzioni contenute nelle “scatole” di Fig. 2, come pure il costo dei test contenuti negli “ovali” di Fig. 2 è costante: non dipende da n (inoltre è “piccolo”, in quanto ogni istruzione/test corrisponde a poche istruzioni macchina).

Questo è visualizzato in Fig 3, dove, al posto delle istruzioni ed dei test abbiamo indicato, per ciascuna: il costo (usando delle costanti C1, C2,...) ed il numero di volte che viene eseguita nel caso peggiore.

Qual’è il caso peggiore ? È quello in cui gli elementi sono in ordine rovesciato (ad esempio, con n=5: 50, 40, 30, 20, 10); ad ogni passo, l’elemento j-mo va confrontato con tutti i precedenti e questi vengono tutti traslati di un posto.

Figura 2 - Diagramma dell'algoritmo





In **Fig. 3** dovrebbe essere tutto chiaro, a parte il valore di **@**.

Quello che abbiamo detto a proposito del caso peggiore, però, fa capire che

al passo j-mo, il corpo del while (cioè le due istruzioni a[i+1] ← a[i] ; i ← i-1)

si esegue j-1 volte, perchè bisogna traslare tutti gli elementi da 1 a j-1.

Allora $@ = \Sigma_{(j=2...n)} (j-1) = \Sigma_{(k=1...n-1)} k$

Poichè questo genere di sommatorie si incontra spesso, vediamo come si ottiene il risultato.

$@ = \Sigma_{(k=1...n-1)}\ k$ Allora:

$$
\begin{array}{lclclclcl}
@ & = & 1 & + & 2 & + & ..... & + & n-1 \\
@ & = & n-1 & + & n-2 & + & .... & + & 1 \\
\hline
2@ & = & n & + & n & + & ..... & + & n \quad (n-1\ \text{addendi})
\end{array}
$$

Quindi $@ = n\,(n-1)/2$.

Adesso possiamo sommare tutti i costi; se indichiamo con T_{IS} la funzione complessità tempo dell'algoritmo, abbiamo:

$T_{IS}(n) = C1 + n\,C2 + ... + (n(n-1)/2 + 1)\,C5 + ... = a\,n^2 + b\,n + c$

con a, b, c costanti opportune; $a > 0$.

Quindi $T_{IS}(n) \in \Theta(n^2)$

Come alternativa: prima di calcolare il valore di @ facciamo un po' di somme e semplifichiamo.

$T_{IS}(n) = C1 + n\,C2 + ... + (@ + 1)\,C5 + ... =$

$a\,n + b@ + c$ con a, b, c opportune costanti intere e positive

Poichè $@ = 1 + 2 + ...+ (n-1)$ si ha subito: $p^2 < @ < n^2$ dove $p = (n-1)\ \text{div}2$

Quindi @ è in $\Theta(n^2)$ e domina sul resto; dunque: $T_{IS}(n) \in \Theta(n^2)$

Insertion sort come procedura

Vedere l'algoritmo come "pezzo di programma" non è molto naturale; la cosa piú ovvia è scriverlo come procedura:

procedura insert_sort (aa array [1 .. n] of integer) con aa parametro IN-OUT, quindi per riferimento

dichiarazioni di j, i, temp

pseudo-codice come sopra, cambiando a con aa.

I conti di complessità, allora si riferiscono ad una generica chiamata

insert_sort(a) con a array [1 .. n] of integer

I conti non cambiano, salvo che per una cosa: bisogna valutare il costo del "passaggio dei parametri". In questo caso, tale costo è costante, perchè il passaggio per riferimento equivale a "passare alla procedura" un puntatore (in C poi, per i parametri array si passa sempre e comunque solo il puntatore al 1° elemento). Ritorneremo sulle procedure più avanti.

Se gli elementi dell'array non sono interi ....

Nei conti che abbiamo fatto, abbiamo valutato costante il costo di confrontare 2 elementi dell'array (istruzione: a[i] > temp) e quello di copiarli (istruzione: a[i+1] ← a[i]). Questo è corretto se gli elementi sono interi, reali, caratteri, record di piccola dimensione,....

Se invece abbiamo un array di stringhe di lunghezza "arbitraria" (quindi un array di arrays o di liste), allora dobbiamo calcolare anche il costo dei confronti e degli spostamenti.

Per la precisione, T_{IS} deve avere 2 parametri: n (numero di elementi) ed lng (che caratterizza la lunghezza delle stringhe nell'array). Il problema è come scegliere lng, visto che le stringhe hanno lunghezza arbitraria. Nell'ottica del caso peggiore, in genere si sceglie lng = lunghezza massima delle stringhe nell'array. A questo punto, il caso peggiore è quello in cui le stringhe hanno (quasi) tutte lunghezza (vicina alla) massima.

In questo caso, il costo dei confronti è lineare in lng (cioè, della forma: a lng + b, con a >0).

Per il costo di copiatura, le cose sono diverse: un array di stringhe di lunghezza variabile è un array di puntatori a liste o array dinamici che contengono le stringhe; allora, l'istruzione a[i+1] ← a[i], copia semplicemente un puntatore e quindi è a costo costante.

In conclusione abbiamo: T_{IS} (n, lng) $\in \Theta(\text{lng } n^2)$.

Si possono fare i conti direttamente sulla Fig. 1

Per calcolare la complessità di insertion sort non è necessario scrivere il codice; è sufficiente ragionare sulla Fig. 1. Poniamo:

Cf_j = numero di confronti tra elementi dell'array al passo j-mo

M_j = numero di "spostamenti" (copiature, assegnazioni) di elementi al passo j-mo.

Si vede che, per ogni j : $M_j = Cf_j + 1$ sempre

$Cf_j = 1$ nel caso migliore

$Cf_j = j-1$ nel caso peggiore

Poichè j varia da 2 ad n, si ottiene (calcolando una sommatoria analoga a quella precedente) che, per tutto l'algoritmo:

- nel caso migliore, si fanno $\Theta(n)$ confronti e $\Theta(n)$ spostamenti
- nel caso peggiore, si fanno $\Theta(n^2)$ confronti e $\Theta(n^2)$ spostamenti.

Si capisce, inoltre, che un programma (una procedura) non scema per insertion sort avrà un costo che dipende solo dal numero di confronti (ed il loro costo) e dal numero di spostamenti (ed il loro costo). Noti il numero ed il costo di confronti e spostamenti, si ha subito la complessità dell'algoritmo.

Si dice che confronti e spostamenti sono *operazioni dominanti* nell'algoritmo (vedere la sezione su Operazioni dominanti).

Per esercizio:

quello che abbiamo fatto su insertion sort farlo su selection sort, nelle due versioni: pezzo di programma e procedura; ignorare eventuali limiti messi sul numero di elementi (tipo n < MAX).

6

. Regole empiriche per il calcolo di complessità

Vediamo delle regole empiriche (e semplici) per valutare il tempo di esecuzione delle istruzioni (e il tempo necessario a valutare le espressioni) che usiamo per scrivere algoritmi, ad eccezione delle (chiamate a) procedure e funzioni ricorsive, che vedremo successivamente.

L'idea di base è la seguente:

- alcune delle espressioni ed istruzioni che utilizziamo sono "elementari"; per queste abbiamo anche un'idea di come sono (o possono essere) implementate a livello di linguaggio assemblativo;
- le espressioni e istruzioni non elementari, incluse le chiamate a procedure o funzioni non ricorsive, si possono tradurre in una successione di istruzioni elementari.

Scriviamo:

tempo(exp) per abbreviare "tempo necessario a valutare l'espressione exp"

tempo(istr) per abbreviare "tempo necessario ad eseguire l' istruzione istr"

Inoltre, ricordiamo che stiamo considerando la complessità rispetto al tempo, quindi "tempo" e "costo" sono equivalenti.

1) Espressioni

tempo(exp) = costante

eccetto: exp con chiamate a funzioni

exp con operazioni su blocchi di dati (es. aa+bb con aa, bb arrays)

tempo(exp con chiamate a funzioni) = Σ tempi(chiamate delle funzioni)

vedremo più avanti come calcolare i tempi delle chiamate di funzioni;

tempo(exp con operazioni su blocchi di dati) = Σ tempi(operazioni "semplici")

esempio: tempo(aa + bb) = $\Sigma_{(i = inf,..., sup)}$ tempo(aa[i] + bb[i])

se aa, bb : array [inf .. sup] of ....

2) Assegnazione, return, espressioni-istruzioni stile C

tempo(x ← exp) = tempo(exp)

tempo(aa[exp'] ← exp) = tempo(exp) + tempo(exp') *qui aa è un array*

tempo(rr.campo) ← exp) = tempo(exp) *qui rr è un record*

e analogamente per altri tipi di assegnazione "semplice"

ma: tempo(aa ← bb) = $\Sigma_{(i = inf,..., sup)}$ tempo(aa[i] ← bb[i])

se aa, bb : array [inf .. sup] of ....

ed analogamente per altre assegnazioni "non semplici".

Infine: tempo(return(exp)) = tempo (exp)

tempo(exp;) = tempo (exp) *questo per lo stile C*

3) Input/output; allocazione/deallocazione di memoria

Qui le cose si complicano. A voler essere precisi:

tempo(scrivi(exp)) = tempo(exp) + ???

tempo (leggi (x)) = ???

tempo (new(p)) = ??? analogo per malloc e calloc

tempo (dispose(p)) = tempo (free (p)) = ???

Spiegazione dei ???.

Il fatto è che le istruzioni di input/output e di allocazione e deallocazione della memoria possono causare delle chiamate al sistema operativo.

New, malloc, calloc generano una richiesta di "locazioni libere"; in alcuni casi (o con alcune implementazioni), la ricerca di queste locazioni è fatta all'interno di uno spazio attribuito al programma al momento dell'esecuzione e quindi può essere abbastanza rapida; in altri casi, la ricerca è fatta dal sistema operativo, dopo aver sospeso l'esecuzione del programma; comunque, non è chiaro quanto tempo sia necessario per soddisfare la richiesta; o meglio: dipende dall' implementazione del linguaggio, dal sistema operativo e dalle circostanze (tanta/poca memoria a disposizione); in ogni caso non si capisce guardando l'algoritmo!

Per la lettura, pensiamo all'esecuzione di per j = 1, 2 ..., MAX : leggi(aa[j])

supponendo che i dati siano in un file su disco ed aa sia un array.

La prima istruzione (leggi (aa[1]) genera un trasferimento di un blocco di **k** dati da disco ad un buffer in memoria centrale (operazione molto lenta rispetto alla velocità con cui il processore esegue le istruzioni); la seconda operazione di lettura (leggi (aa[2]) preleva il dati direttamente dal buffer (quindi è molto più veloce; si può dire: a costo costante) e così via... Se MAX = 10***k** avremo 10 letture costose e tutte le altre a costo costante. Il fatto è che il valore di **k** dipende dal sistema e non è noto quando discutiamo l'algoritmo. (Anche qui ci sono varianti, ottimizzazioni,....)

Le operazioni di scrittura e di deallocazione delle memoria (dispose o free) richiedono anch'esse un certo lavoro, ma non necessariamente subito; anzi, si può dire che in genere vengono "accantonate" ed eseguite "in differita", in modo da migliorare l'efficienza (questo spiega perchè alle volte si riesce ad accedere al record puntato da un puntatore p, anche *dopo* un `free(p)`).

Per semplificarci la vita, **nei conti useremo**:

tempo(scrivi(exp)) = tempo(exp) e quindi

tempo (scrivi (exp_1, ..., exp_k)) = tempo(exp_1) + ... + tempo(exp_k)

tempo (leggi (x)) = costante e quindi

tempo (leggi (x_1, ..., x_k)) = costante (infatti k è costante)

tempo (new(p)) = costante analogo per malloc e calloc

tempo (dispose(p)) = tempo (free (p)) = costante

Per rimediare a questa approssimazione brutale (??? = costante), **nel progettare** gli algoritmi:

- si cerca di allocare la memoria “a blocchi”, invece che un elemento alla volta; ad esempio si evitano le liste se si possono usare array dinamici;
- si evita di liberare la memoria se non c’è pericolo di rimanere senza.

C’è poco da fare, invece, riguardo all’input/output: non possiamo certo leggere solo una parte dei dati o stampare solo una parte dei risultati per fare più in fretta !

4) Successione di istruzioni

tempo(istr_1; .... ; istr_k) = tempo(istr_1) + ... + tempo(istr_k)

Usando gli ordini di grandezza e la proprietà [max], vista in Sezione , possiamo limitarci al massimo dei tempi(istr_i); infatti:

χ (t_1 + ...+ t_k)) = χ(**max** { t_1,..., t_k }) per χ = O, Ω, Θ.

5) Istruzioni condizionali

Vogliamo calcolare: **T** = tempo(**if** cond **then** blocco_1 **else** blocco_2)

Siano: Tc = tempo(cond)

T_i = tempo(blocco_i) per i=1, 2

allora: Tc + min { T_1, T_2 } = **T** = Tc + max{ T_1, T_2 }

Con la filosofia del caso peggiore:

se per ogni dimensione d, esiste un input di dimensione d che fa scegliere il ramo più costoso, allora **T** = Tc + max{ T_1, T_2 }.

Ragionamento analogo per **if-then** (si può anche ricavare da sopra mettendo a zero il costo del ramo else che manca).

6) Istruzione while

Per calcolare il tempo di **while** cond **do** blocco ci sono due problemi: bisogna capire quante volte si esegue il blocco (e questo è l’aspetto più difficile) e bisogna valutare il costo di esecuzione del blocco ad ogni iterazione: non è detto che sia sempre uguale; qualche volta, anche il costo di valutare la condizione varia ad ogni iterazione.

Vediamo un esempio: scrivere tutti i prefissi non vuoti di una stringa data.

Se la stringa è `pesce`

i prefissi non vuoti sono: `p, pe, pes, pesc, pesce`

Un possibile algoritmo è il seguente:

Input: n, st (stringa di lunghezza n)

Output: i prefissi di st in ordine crescente di lunghezza, uno per riga

Usa un array di caratteri ast di lunghezza n per immagazzinare st

(è un array dinamico, stile C, quindi in effetti ast è un puntatore a carattere)

e altre variabili intere ovvie

Istruzioni

(1)leggi(n)

```
(2)alloca spazio per  ast  [ ad  esempio:   ast = calloc(n, sizeof(char)) ]
    /*      gli indici in  ast  vanno da  0   a   n-1       */
(3)k ← 0
(4)while   k < n  do
      {      (4.1)     j ← 0
             (4.2)     while  j = k  do       {   scrivi( ast[j] )  ;      j++    }
             (4.3)     vai a capo sull'output  [ =  scrivi  ('a_capo')  ]
             (4.4)     k ++
      }
```

L'esempio è un po' artificiale, perchè in un caso come questo non si userebbero dei "while", ma dei "for", comunque serve ai nostri scopi.

Nota. La convenzione che usiamo qui nel numerare le istruzioni rispecchia la struttura delle istruzioni stesse: (1), (2), (3), (4) sono istruzioni "separate"; (4.1), (4,2), (4,3), (4.4) sono "tra di loro separate", ma fanno parte di (4); quindi, ad esempio, i costi di (1) e di (3) e di (4.2) e (4.3) si possono calcolare separatamente, mentre il costo di (4) include quello di (4.1),..., (4.4).

In base a quello che abbiamo visto:

- le istruzioni (1), (2), (3) hanno costi costanti; diciamo che la loro somma è d
- le istruzioni (4.1), (4.3), (4.4) hanno costi costanti; diciamo che la loro somma è c

quindi resta solo da calcolare il costo di (4.2) e poi di (4).

Per il costo di (4.2):

- il corpo, cioè { scrivi(ast[j]); j++ } ha costo costante sia cc1;
- anche il test j = k ha costo costante, sia ct1
- il corpo si esegue k+1 volte
- il test viene valutato k+2 volte (all'ultima produce "falso" e dopo non si esegue più il corpo)

Allora il costo di (4.2) è: $(k+1)\, cc1 + (k+2)\, ct1 =$

$= a\,k + b$ (con a e b costanti opportune, $a > 1$).

Per il costo di (4):

- il test $(k < n)$ ha costo costante, chiamiamolo ct2, e si valuta n+1 volte; quindi il suo contributo al costo è : $(n+1)\, ct2$;
- il corpo { (4.1), ...(4.4) } ha un costo che è dato da : $ak + b + c = ak + b'$

Il punto è che il costo del corpo non è costante, come nel caso (4.2), ma dipende dal valore della variabile di controllo, cioè k; quindi per calcolare il costo di (4) non ce la possiamo cavare con una moltiplicazione, ma dobbiamo usare una sommatoria:

costo di (4) $= (n+1)\, ct2 + \Sigma_{(k = 0 \ldots n-1)} (ak+b')$ e sviluppando e semplificando:

$= a_0 + a_1 n + a \Sigma_{(k = 0 \ldots n-1)} k$ e ricordando la sommatoria di insertion sort:

$= b_0 + b_1 n + b_2 n^2$ con b_0, b_1, b_2 costanti opportune e $b_2 > 0$

Per ottenere il costo di tutto l'algoritmo, basta ora sommare il costo di (4) a quello di (1), (2), (3); quindi la complessità dell'algoritmo è in $\Theta(n^2)$.

Ritornando al caso generale:

tempo(while cond do blocco) = Tc_last + $\Sigma_{(i = 1...max)}$ (Tc_i + Tb_i)

dove: max = il numero di volte che si ripete il ciclo

Tb_i = tempo di esecuzione del blocco alla i-ma iterazione

Tc_i = tempo di valutazione della condizione alla i-ma iterazione

Tc_last = tempo di valutazione della condizione l'ultima volta (prima di uscire)

Come già detto, il problema è stimare i costi Tb_i e Tc_i e capire quante volte si esegue il ciclo; spesso non si riesce a farlo con precisione, ma si approssima tutto usando gli ordini di grandezza.

In particolare, nella maggior parte dei casi:

il tempo di valutare la condizione è dominato dal resto e quindi si ignora.

Ritornando all'esempio di sopra: una volta capito come stanno le cose, e volendo ottenere subito la stima in ordini di grandezza, il conto si fa in modo molto più sbrigativo:

- tutti i costi sono costanti, a parte quello dei due while;
- il costo di tutto l'algoritmo è, in ordini di grandezza, determinato dal costo di (4);
- il while (4.2) ha un costo della forma $ak+b$, con $a > 0$;
- in (4), il costo del test è dominato dal costo del corpo, che è della forma $ak+b'$, con $a > 0$;
- il costo di (4) è determinato dalla sommatoria $\Sigma_{(k = 0...n-1)} (ak+b')$
- $\Sigma_{(k = 0...n-1)} (ak+b') = b_0 + b_1 n + b_2 n^2$ con b_0, b_1, b_2 costanti opportune e $b_2 > 0$
- quindi l'algoritmo è in $\Theta(n^2)$.

7) Altre istruzioni iterative.

Repeat: del tutto analogo a while.

Istruzioni "per", "for".

Purtroppo ci sono diverse varianti, abbastanza diverse tra loro. Il modo più sicuro, fino a quando non si è acquistata abbastanza pratica, è di tradurle usando un while, o un diagramma (disegno con le scatole,....).

Vediamo i due tipi più frequenti.

7.1) **per** k **=** e_inf, e_inf+1, ..., e_sup **:** blocco *nostra versione, equivalente a:*

for k **:=** e_inf **to** e_sup **do** blocco *stile Pascal*

dove e_sup ed e_inf sono espressioni

È equivalente a:

```
{  sup ← e_sup
   k ← e_inf
   while k = sup do {  blocco ;  k++ }
}
```

Qui sup è una nuova variabile; serve a modellare il fatto che e_sup viene valutato una volta sola, come del resto e_inf. Quindi il costo di tutta l'istruzione for/per, sia Tf è:

Tf = t_inf + t_sup — tempo per valutare e_inf ed e_sup
+ num (t_test + t_incr) — vedi (*) sotto
+ t_test — ultimo test, quando k =sup+1
+ $\Sigma_{(i = inf ... sup)}$ (t_blocco_i) — vedi (**) sotto

(*) qui num = sup - inf +1 = numero di volte che si esegue il ciclo
t_test è il costo del test
t_incr = costante è il costo di incrementare k

(**) t_blocco_i è il costo di eseguire il blocco quando k=i

Per un esempio, vedere insertion sort. In genere l'ultimo termine domina sul resto, quindi è sufficiente calcolare $\Sigma_{(i = inf ... sup)}$ (t_blocco_i)

7.2) for(exp_1; exp_2; exp_3) <blocco> *stile C*

È equivalente a:

```
{ exp_1 ;
  while exp_2 do { <blocco>
                   exp_3 ; }
}
```

8) Procedure non ricorsive

Il costo da valutare è <u>quello delle chiamate</u> a procedura.
Data una dichiarazione: procedure P (parf_1, ..., parf_k) { blocco }
consideriamo una chiamata P (para_1, ..., para_k);
Qui: i parf_i sono i parametri formali e i para_i sono i parametri attuali, che possono anche essere espressioni. Quindi:

tempo (P (para_1, ..., para_k))= tempo (passaggio dei parametri)
+ tempo(blocco)

Il **tempo(blocco)** si calcola applicando le regole che stiamo descrivendo; se contiene delle chiamate a procedura/funzione, il loro costo si calcola a parte, separatamente,.... e poi si somma tutto. Poichè abbiamo detto che non c'è ricorsione, prima o poi riusciamo a calcolare il costo di tutte le chiamate a procedura/funzione.

Per valutare il **tempo (passaggio dei parametri)** bisogna distinguere tra parametri IN, OUT e IN-OUT e riferirsi al modo in cui il meccanismo è realizzato a livello di implementazione.

- Se parf_i è un parametro OUT o IN-OUT, allora para_i deve essere una variabile; bisogna calcolare l'indirizzo e "passarlo alla procedura". In genere tutto questo è a tempo costante, a meno che per calcolare l'indirizzo non serva calcolare delle espressioni indice, come nel caso para_i = aa[exp, exp'], con aa array; comunque è simile a quanto visto per l'assegnazione.
- Se parf_i è un parametro IN, allora para_i è una espressione; il costo del passaggio è lo stesso dell'assegnazione parf_i ← para_i cioè quello di valutare para_i e copiarlo in parf_i .

<u>In conclusione, il costo del passaggio dei parametri è costante, eccetto :</u>

- quando bisogna calcolare espressioni indice complesse (magari con chiamate a funzioni);
- quando bisogna copiare in parf_i un valore strutturato: ad esempio un record (in C e Pascal) o un array (solo in Pascal; in C gli array non sono mai passati per valore, ma si passa - per valore- solo il puntatore al primo elemento).

Per un esempio: vedere sezione successiva (complessità della visita di un albero).

9) Funzioni non ricorsive

I conti sono del tutto analoghi a quelli per le chiamate a procedura.

Data una dichiarazione: function F (parf_1, ..., parf_k) { blocco }

ed una chiamata F (para_1, ..., para_k) [che è una espressione].

tempo (F (para_1, ..., para_k))= tempo (passaggio parametri) + tempo(blocco).

Procedure e funzioni ricorsive: vedere oltre.

10) Altre istruzioni

Si procede in analogia a quanto visto; ad esempio:

- istruzione **switch/case**: si traduce in una cascata di if then else, oppure si ragiona direttamente su quella che è la semantica dello switch, o del case:

  si valuta questo, poi si confronta, poi ......
- **break**: deve solo realizzare un JUMP, quindi è a costo costante;
- **"istruzioni ad alto livello"** tipo:

  "ordina l'array A in modo crescente"

  "scorri la lista L a partire dalla testa fino al primo record con il campo info = x"

bisogna capire come si tradurrebbero usando istruzioni standard e fare i conti sulla traduzione.

11) Dichiarazioni

Il costo di processare le dichiarazioni viene di solito ignorato, per due motivi:

- è costante, anche nel caso della dichiarazione di un array molto grande, a meno che l'implementazione non preveda una inizializzazione automatica;
- in quasi tutti gli algoritmi sensati, il costo delle istruzioni è dominante.

12) Compilazione, linking,....

Anche i costi legati a queste fasi vengono ignorati. Il motivo è sono costi che dipendono dal sistema e dall'ambiente di programmazione usato e che comunque precedono l'esecuzione.

Dobbiamo ricordare che, nella realtà, si fanno i conti di complessità solo su algoritmi che poi verranno tradotti in programmi che verranno usati molto; inoltre, il conto di complessità (quello relativo al "tempo") ha lo scopo di stimare l'ordine di grandezza del **tempo di esecuzione del codice oggetto**; niente di più.

Del resto, c'è un altro tipo di costo, molto rilevante anche dal punto di vista economico, che è ignorato: il costo del lavoro del programmatore!

Esercizi:

fare i conti di complessità per il programma e la procedura sui numeri primi, e per il programma che genera tutti i sottinsiemi di {1,...,n} e per la procedura di ricerca binaria, nella versione iterativa.

In tutti questi esempi, ignorare eventuali limiti messi sul numero di elementi (tipo $n < MAX$).

7

. Complessità della visita non ricorsiva di un albero

Usiamo questo esempio, per chiarire alcune questioni, in particolare: *genericità* dei parametri attuali, dimensioni dell'input (qui, il parametro).

Per semplicità consideriamo alberi binari; il ragionamento si estende facilmente al caso di alberi di altro tipo. Quindi, a livello di implementazione:

un albero è un puntatore_a_nodo

un nodo è un record a tre campi: info di tipo .... *non interessa conoscerlo*

sin, des di tipo albero *figlio sinistro e destro*

Consideriamo una visita in ampiezza o profondità; lo schema è identico, realizzato come procedura non ricorsiva, solo che: se la visita è in ampiezza, usa una coda di alberi (cioè di puntatori); se è in profondità, usa uno stack di alberi e si inverte l'ordine dei figli.

```
visita ( tt : albero     --  parametro IN)
usa  aux  che  è uno stack / una coda
{   if    tt    non è vuoto
    then   {   inizializza aux a vuoto
               metti  tt   in  aux      -- in testa se è stack, in coda se è coda
               while    aux  non  è vuoto  do {
                       preleva l'elemento in testa ad aux, sia   tx
                       "visita la radice di tx", chiamiamola r
                       se  r  ha figli  mettili in aux  (in testa / coda)
                    }    chiude il while
           }  chiude l'if-then
}
```

Come visto in precedenza, ci interessa la complessità di una chiamata

visita(t) , dove t è un albero binario.

Per essere più precisi, quello che ci interessa è una *chiamata generica*: non interessa calcolare il costo di visita(t) , per un *particolare* t, ma per *qualunque* t (puchè del tipo giusto: albero binario). Capito questo, non ha molto senso distinguere il (nome del) parametro formale e il (nome del) parametro attuale; per indicare un generico albero binario possiamo benissimo usare il nome "tt", quindi vediamo di calcolare il costo

della generica chiamata visita(tt) , dove tt è un generico albero binario.

Nota. Seguiremo questa convenzione (non distinguere tra parametro formale a parametro attuale, nel fare i conti di complessità) anche nel seguito. Quando affronteremo le procedure ricorsive, vedremo che questa scelta ci semplifica le cose.

Bisogna ora capire quale parametri usare per caratterizzare la dimensione dell'albero. In genere, si usa: il numero dei nodi, oppure l'altezza, oppure entrambe. Se l'albero è (quasi) perfettamente bilanciato, allora c'è un legame tra altezza e numero di nodi e quindi basta conoscere uno dei due per avere l'altro; in generale non è così.

Supponiamo, per cominciare, di conoscere il numero di nodi di tt e chiamiamolo nn. (In effetti, poichè stiamo visitando un albero e la visita di un albero non è altro che la visita di tutti i nodi, è chiaro che nn è il parametro più comodo.)

Poichè ci interessa capire cosa succede al crescere di nn possiamo ignorare il caso nn=0; quindi il test "tt non è vuoto" sarà vero e si eseguirà il ramo else.

Costi delle varie parti:

- passaggio dei parametri: costante
- i vari test (tt non è vuoto, aux non è vuoto, r ha figli...): costante

  per la precisione: si tratta di costanti diverse (e diverse da quelle di sotto, o di sopra) però per una stima in ordini di grandezza, non interessa distinguerle.
- operazioni (inizializza aux a vuoto, metti in aux, preleva elemento di testa): costante, se le operazioni su stack e code sono implementate bene
- "visita della radice di tx": dipende da cosa facciamo; per ora supponiamo che sia costante;
- tempo(if ... then ....) = costante + tempo(while .....)

  notare che la costante di sopra mette assieme il costo di "tt non è vuoto" e di "metti tt in aux"
- tempo (while ....) = costante + num * costante

  dove num è il numero di volte che si esegue il ciclo

Si tratta solo di capire quanto vale num. Se ci si limita a guardare il codice, non è facile: prima aux è vuoto, poi si inseriscono elementi, poi si tolgono,..... ; in effetti, guardando solo il codice non è nemmeno ovvio che il while termini. Un modo di capire che termina e di determinare num è il seguente: ad ogni "giro" si visita un nodo, ogni volta è un nodo diverso, si visitano tutti i nodi dell'albero; quindi num = nn.

Allora il costo della chiamata, in funzione di nn è dato dalla funzione T_nodi:

$$T_nodi(nn) = c + nn * c \ (\text{con } c > 0) \qquad \text{quindi} \qquad T_nodi(nn) \in \Theta(nn)$$

Questo è esattamente quello che ci si doveva aspettare; con un po' di esperienza si vede subito, ad occhio, che $T_nodi(nn) \in \Theta(nn)$.

Però non conviene fare le cose ad occhio se l'esperienza non c'è! Vedere oltre.

Supponiamo ora di conoscere solo l'altezza di tt, sia h.

Vogliamo quindi la complessità in funzione di h; sia T_alt la funzione relativa.

Guardando l'algoritmo, si capisce, come già detto, che invece è comodo fare i conti in funzione del numero di nodi, sia nn. Bene, lo facciamo e concludiamo, come sopra, con un costo in $\Theta(nn)$.

A questo punto, si tratta di riportare il risultato in funzione dell'altezza.

Se l'albero fosse (quasi) perfettamente bilanciato, sarebbe facile: $nn \approx 2^{h+1}$ quindi avremmo:

$T_alt(h) \in \Theta(2^h)$.

Nel caso generale, non resta che ricordarsi che ci interessa, come al solito, il caso peggiore.

Rispetto ad un'altezza fissata h, il caso peggiore (nota: per questo algoritmo; non sempre!) è quello in cui il numero dei nodi è massimo (per alberi di quell'altezza), ma questo è proprio il caso di sopra: a parità di altezza, l'albero binario con massimo numero di nodi è quello perfettamente bilanciato.

Quindi abbiamo ancora: $T_alt(h) \in \Theta(2^h)$.

Ritorniamo al costo di "visitare un nodo".

I casi sono sostanzialmente due: o è costante come supposto sopra, oppure dipende dalla "dimensione del nodo (o meglio dell'etichetta)". Per capire quest'ultimo caso, basta pensare a nodi con etichette che sono stringhe di lunghezza arbitraria e visita = stampa etichetta.

In questo caso, però, ci serve un parametro aggiuntivo, la dimensione massima dei nodi.

Quindi dobbiamo conoscere, o poter stimare: mdn = max { dim(etichetta di x), $x \in$ Nodi(tt) }.

Inoltre dobbiamo calcolare il costo della visita di un nodo in funzione della sua dimensione d; supponiamo si dato dalla funzione V(d).

Allora:

la complessità in funzione di nn e mdn è: $T'(nn, mdn) \in \Theta(nn * V(mdn))$

quella in funzione dell'altezza e di mdn è: $T''(h, mdn) \in \Theta(2^h * V(mdn))$.

Complessità spazio.

Vediamo, tanto per cambiare, quanto costa la visita in termini di spazio, cioè di memoria occupata. Indichiamo sempre con nn il numero di nodi e con h l'altezza dell'albero.

Per prima cosa c'è la memoria occupata dall'albero; questa dipende dalla dimensione dei nodi e dalla implementazione degli alberi. Nel caso di nodi con etichette che sono interi e la solita implementazione a record e puntatori, abbiamo una occupazione pari a 3 nn celle per interi.

Quello che distingue i due tipi di visita è però lo spazio richiesto dalle strutture ausiliarie: cioè la coda o lo stack.

Nella visita in profondità, lo stack contiene al più h elementi e ci sono dei momenti che ne contiene effettivamente h; quindi per eseguire l'algoritmo serve uno spazio ausiliario pari all'altezza (o meglio proporzionale all'altezza; se lo stack è a lista, ci sono anche i puntatori della lista, quindi in totale 2h puntatori circa).

Notare che la dimensione dello stack varia, ma quello che dobbiamo calcolare è il valore massimo che può raggiungere; infatti se questo valore supera le possibilità della nostra macchina dobbiamo rinunziare alla visita.

Ragionando in maniera analoga, si vede che nella visita in ampiezza, lo spazio che serve per la coda è proporzionale al massimo numero di nodi allo stesso livello.

Tenendo fisso il numero di nodi, tutto dipende dalla forma dell'albero; due i casi estremi:

- albero "alto e magro", cioè con pochi nodi per livello e, quindi, una altezza che è dello stesso ordine di grandezza di n, cioè in $\Theta(nn)$; allora per lo stack servono $\Theta(nn)$ celle e per la coda ne bastano poche;
- albero "basso e grasso", cioè (quasi) completo; allora l'altezza è in $\Theta(\log nn)$ e per lo stack servono $\Theta(\log nn)$ celle, mentre per la coda ne servono $\Theta(nn)$, infatti, verso la fine, la coda contiene circa nn/2 nodi (le foglie).

Conti “ad occhio”

Come detto sopra, con un po’ di esperienza, si può stimare la complessità dell’algoritmo di visita, senza fare tanti conti. Il problema è che l’esperienza bisogna acquistarla. Un modo di procedere che permette di evitare gli errori che nascono da conti troppo superficiali, ma anche da quelli troppo dettagliati, è il seguente:

1. fare un conto ad occhio
2. fare un conto più preciso, ma senza perdersi troppo in dettagli
3. confrontare il risultato col precedente; se non quadra capire dove si è sbagliato (in 1 o in 2 ?) e correggere
4. fare un conto ancora più preciso
5. confrontare il risultato col precedente; se non quadra capire dove si è sbagliato e correggere
6. eccetera ....

Questo modo di fare le cose si può usare assieme all’altro di iniziare con stime grossolane che permettono solo di concludere in O(...), oppure in Ω(...), raffinandole poco alla volta fino a concludere, se possibile, con una stima in Θ(...).

8 . Operazioni dominanti

Nella moltiplicazione tra matrici, il costo dell’algortimo standard (quello che segue la def. data a Geometria) è determinato dal numero di moltiplicazioni tra elementi, nel senso che ora precisiamo.

Se ho matrici $n \times n$: si fanno n^3 moltiplicazioni e (circa) n^3 somme (e inoltre le somme sono in effetti meno costose delle moltiplicazioni).

Si verifica facilmente che il costo dell’algoritmo, in funzione di n, è:

$T(n) = a n^3 + b n^2 + c n + d$ dove a, b, c, d sono opportune costanti, con $a > 0$.

D’altra parte, il costo totale delle sole moltiplicazioni è

$CM(n) = (costo_per)\, n^3$ dove costo_per è il costo di una moltiplicazione, che è costante.

Passando agli ordini di grandezza: sia T(n) che CM(n) sono in $\Theta(n^3)$; quindi per stimare la complessità dell’algoritmo, basta stimare quante sono le moltiplicazioni e valutare il loro costo.

Negli algoritmi di selection sort ed insertion sort e nella procedura merge (usata per merge sort), il costo di tutto è determinato da:

- numero di confronti tra elementi
- numero di copiature di elementi (alcuni dicono; spostamenti di elementi)

Per insertion sort, questo è stato già discusso alla fine della sezione relativa; per selection sort e per merge il discorso è analogo.

Per l’algoritmo di vista degli alberi binari discusso nella sezione precedente, il costo totale, in ordini di grandezza, coincide con il costo di visitare tutti i nodi.

Operazioni come la moltiplicazione (nel prodotto di matrici), i confronti e le copiature (negli algoritmi di ordinamento), la visita di un nodo (nella visita di un albero) si dicono *operazioni dominanti* (per l'algoritmo).

Questa situazione è molto frequente, per cui proviamo a dare qualche definizione.

A, è un algoritmo; op, op1, op2 sono operazioni che compaiono in ***A***

Si dice che op è dominante in ***A*** (o per ***A***) se

$\mathbf{T}_{\boldsymbol{A}}(\mathbf{n}) \in \Theta(\text{Num}(\mathbf{n}) * \text{costo_op})$

dove: **n** caratterizza la dimensione dell'input ad ***A*** (può anche essere una tupla)

Num(**n**) è il numero di volte che si esegue op (nel caso peggiore) per input di dimensione **n**

costo_op è il costo di una singola op

Si dice che op1, op2 sono dominanti in ***A*** (o per ***A***) se

$\mathbf{T}_{\boldsymbol{A}}(\mathbf{n}) \in \Theta(\text{Num1}(\mathbf{n}) * \text{costo_op1} + \text{Num2}(\mathbf{n}) * \text{costo_op2})$

dove: **n** caratterizza la dimensione dell'input ad ***A***

Num1(**n**) è il numero di volte che si esegue op1 (nel caso peggiore) per input di dimensione **n**

costo_op1 è il costo di una singola op1

Num2 ed op2 sono analoghe

La definizione si estende facilmente a k operazioni dominanti.

Uso delle operazioni dominanti.

Invece di stimare il costo di ***A***, usando le regolette viste precedemente, si cerca di trovare una, o più, operazioni dominanti, di stimare il loro costo (in genere è facile) e di stimare il numero di volte che vengono eseguite (non è necessario calcolare il numero preciso di volte; basta l'ordine di grandezza). Di qui si ottiene direttamente la complessità dell'algoritmo.

9

. Complessità (intrinseca) dei problemi - algoritmi ottimi

Sia **PR** un problema.

- Se esiste un algoritmo *A* che risolve **PR** e t.c. $\mathbf{T}_A$ è in O(f), allora si dice che:
  -- **PR** ha una complessità (intrinseca) in O(f) oppure che
  -- f è un limite superiore alla complessità intrinseca di **PR** oppure che
  -- **PR** è in O(f)
- Se per ogni algoritmo *A* che risolve **PR** si ha $\mathbf{T}_A$ è in Ω(f), allora si dice che:
  -- **PR** ha una complessità (intrinseca) in Ω(f) oppure che
  -- f è un limite inferiore alla complessità intrinseca di **PR** oppure che
  -- **PR** è in Ω(f)
- Se **PR** è in O(f) ed è anche in Ω(f) allora si dice che:
  -- **PR** ha una complessità (intrinseca) in Θ(f) oppure che
  -- la complessità intrinseca di **PR** è dell'ordine di f oppure che
  -- **PR** è in Θ(f)

Note:

- anche qui stiamo considerando il caso peggiore.
- non abbiamo definito la complessità di un problema, abbiamo detto solo come si stima.
-

Trovare un limite superiore è "semplice": basta trovare un algoritmo che risolve **PR** e calcolarne la complessità.

Per i limiti inferiori le cose sono piú complicate: non è possibile controllare tutti i possibili algoritmi per **PR**, anche perchè tra questi ci sono anche quelli che ancora non sono stati scoperti ....

Quello che può fare è: dato **PR** capire quale funzione si deve scegliere come f e poi dimostrare che ogni algoritmo per **PR** ha una complessità in Ω(f). Queste dimostrazioni non sono facili, in genere; ci accontenteremo quindi dei cosí detti "limiti inferiori banali", cioè ottenuti con ragionamenti molto semplici (vedere esempi, più avanti).

Algoritmi ottimi

Se **PR** è in Ω(f) e *A* è un algoritmo per **PR** con complessità in O(f) [e quindi in base alle def. sono entrambi in Θ(f)] allora si dice che *A* è un algoritmo ottimo (o ottimale).

In altre parole, un algoritmo è ottimo quando non è possibile fare di meglio, almeno in ordini di grandezza. È importante non dimenticare che l'ottimalità di cui parliamo è solo in ordini di grandezza. Dati due algoritmo ottimi per **PR** è possibile che uno sia, in realtà, migliore dell'altro.

Chiamiamo A_1 ed A_2 gli algoritmi e siano $\mathbf{T}_1(n)$ e $\mathbf{T}_2(n)$ le rispettive funzioni complessità; allora A_1 è in effetti il migliore dei due se, per esempio

$\mathbf{T}_1(n) = 5\ n^2 - 30\ n + 1000$ e $\mathbf{T}_2(n) = 10\ n^2 + \log_2 n + 300$

oppure se, nel caso medio, $\mathbf{T}_1(n)$ è in Θ(n), mentre $\mathbf{T}_2(n)$ è in $\Theta(n^2)$.

Esempi.

2. **PR** = ricerca di un elemento x in un array aa di n interi (o reali, o caratteri,...)

- **PR** è in O(n), perchè l'algoritmo ovvio di ricerca sequenziale è in O(n);
- **PR** è in Ω(n), infatti un algoritmo corretto deve dare risposta NO quando x non è in aa; per fare questo, deve almeno controllare tutti gli elementi di aa, quindi deve fare almeno n passi;
- dunque la ricerca sequenziale è un algoritmo ottimo e problema ed algoritmo sono in Θ(n).

> **Attenzione.**
> Con Ω(...) si rischia di fare confusione , sia ragionando a livello di algoritmo che a livello di problema.
> Infatti c'è la tentazione di dire:
>
> se x è il primo elemento di aa, allora il costo per trovarlo è costante, dunque il problema (o l'algoritmo) è in Ω(1).
>
> Bisogna ricordare che **anche le stime in Ω(...) si fanno relativamente al caso peggiore** (a meno che non stiamo esplicitamente considerando il caso migliore o medio); chiaramente, il caso in cui x si trova, e subito, non è il caso peggiore!

2. Del tutto analogo se invece di un array consideriamo una lista.

3. Se l'array aa è ordinato, il ragionamento di sopra non vale; del resto la ricerca binaria risolve il problema in Θ(log n) [vedere complessità delle procedure ricorsive]; in effetti il limite inferiore per il problema è proprio log n (e quindi il problema è in Θ(log n) e la ricerca binaria è ottima), ma la dimostrazione non è immediata.

4. **PR** = ordinamento di un array aa di n interi (o reali, o caratteri,...)
   - **PR** è in O(n log n), perchè ci sono algoritmi di ordinamento, come merge sort, in Θ(n log n) [vedere complessità delle procedure ricorsive];
   - si dimostra (ma non è immediato) che **PR** è in Ω(n log n)

     (per essere piú precisi: si dimostra che qualunque algoritmo basato su confronti tra elementi deve fare, nel caso peggiore, almeno Ω(n log n) confronti);
   - dunque merge sort è ottimo e problema ed algoritmo sono in Θ(n log n);
   - esistono algoritmi di ordinamento di array con complessità in Θ(n), cioè lineare (bin sorting, radix sorting), ma si applicano solo a casi particolari, mentre qui discutiamo il caso generale.

1. **PR** = ordinamento di una lista (o un file) di n interi (o reali, o caratteri,...)
   - **PR** è in O(n log n), perchè ci sono algoritmi di ordinamento, ad es. quelli derivati da merge sort, in Θ(n log n);
   - si dimostra (ma non è immediato) che **PR** è in Ω(n log n)

     (per essere piú precisi: si dimostra che qualunque algoritmo basato su confronti tra elementi deve fare, nel caso peggiore, almeno Ω(n log n) confronti);
   - dunque gli algoritmi tipo merge sort sono ottimi ; problema ed algoritmi sono in Θ(n log n);
   - anche qui esistono algoritmi di ordinamento con complessità in Θ(n) (sempre bin sorting e radix sorting), ma si applicano solo a casi particolari.

1. Se prendiamo i problemi da 1 a 5 e cambiamo il tipo degli elementi, i costi possono cambiare. Infatti, i costi sono stati calcolati assumendo costanti il costo dei confrontare due elementi e quello di copiare un elemento. Se gli elementi, ad esempio, fossero stringhe di lunghezza arbitraria, allora dovrebbe entrare in gioco un altro parametro: lmax = lunghezza massima delle stringhe presenti nella lista/array; infatti il costo di confrontare due stringhe è proporzionale alla loro lunghezza (per la copiatura: non è detto si debba copiare, se le stringhe sono liste, o array dinamici stile C, basta "spostare i puntatori di testa"). Allora tutti i costi degli algoritmi andrebbero moltitplicati per lmax.

2. **PR** = moltiplicazione di due matrici quadrate (di interi o real) di dimensione n
   - **PR** è in $O(n^3)$, perchè l'algoritmo che si deriva immediatamente dalla definizione di prodotto di matrici è in $\Theta(n^3)$;
   - **PR** è banalmente in $\Omega(n^2)$, perchè qualunque algoritmo corretto deve almeno "guardare" tutti gli elementi delle 2 matrici;
   - sono stati studiati tanti algoritmi (alcuni estremamente complicati) fino ad arrivare ad algoritmi in $O(n^{2.4})$, però non conosciamo algoritmi in $O(n^2)$.

5. **PR** = visita (in ampiezza, o in profondità) di un albero con n nodi.
   - Come abbiamo visto, il costo dipende da quello delle operazioni che si fanno sui singoli nodi, al momento della visita; <u>supponiamo che abbiano costo costante</u> ; allora:
   - **PR** è in $O(n)$, perchè l'algoritmo standard è in $\Theta(n)$ *vedere fogli precedenti*;
   - **PR** è banalmente in $\Omega(n)$, perchè qualunque algoritmo corretto deve almeno "guardare" tutti i nodi;
   - dunque ....

5. **PR** = generare (e stampare) tutti i sottinsiemi di {1,...,n}
   - **PR** è banalmente in $\Omega(2^n)$, perchè i sottinsiemi sono 2^n
   - è anche in $\Omega(n\ 2^n)$ se consideriamo il costo di stamparli, infatti la metà dei sottinsiemi ha almeno n/2 elementi (per ogni sottinsieme di k elementi, il complementare ne ha n-k);
   - guardando l'algoritmo nel Fascicolo 1 (togliendo la limitazione sul valore di n) e ragionando come sopra, si vede che questo è in $\Theta(n\ 2^n)$; <u>da questo si conclude solo che</u> **PR** è in $O(n\ 2^n)$; ma usando il punto precedente si ottiene che **PR** è in $\Theta(n\ 2^n)$
   - se poi andiamo a considerare la dimensione "vera" dell'input, $ln = \log_b n$, dove b è la base della rappresentazione di n come stringa, e facciamo le stime di complessità in funzione di ln, le cose peggiorano di molto, infatti quello che succede è che in $\Theta(n\ 2^n)$ dobbiamo sostituire gli n con b^{ln} (questo punto verrà ripreso in seguito, parlando di algoritmi numerici: fattoriale, numeri di Fibonacci,...)

10. Ordini di grandezza e buon senso

Prima di passare alle preocedure ricorsive, un appello al buon senso.

Tutti questi discorsi a base di ordini di grandezza non devono farci trascurare le cose ovvie.

In particolare: quello che è inutile resta inutile e quello che è stupido resta stupido, anche se "in ordini di grandezza non fa differenza".

Prendiamo il solito esempio di cercare un elemento in un array di n interi. L'algoritmo solito risolve il problema, nel caso peggiore, in n passi, perchè controlla ogni elemento una volta sola; un algoritmo stupido potrebbe controllare ogni elemento 3 volte, compiendo quindi $3n$ passi. Tutti e due gli algoritmi sono in $\Theta(n)$ e, per quello che abbiamo visto, sono ottimi; questo non cambia il fatto che il secondo è un algoritmo stupido.

In altre parole: le stime in O( ), Ω( ) e Θ( ) non possono fornire una giustificazione per rigiri e contorsioni inutili.

11

. Complessità delle procedure e funzioni ricorsive

Procediamo come al solito per esempi, usando algoritmi già visti.

11.1 . Primo esempio.

Problema: dato un array di n caratteri, cambiare tutte le 'a' in 'A'

Algoritmo ricorsivo:

```
        procedura  a_to_A (   aa: array [1 .. n] of char,        parametro IN-OUT
                              k : int                            parametro IN        )
         if k = n  then   {   if aa[k] ='a'  then  aa[k] ← 'A'
                              a_to_A (aa, k+1)
                          }
```

Dato un generico array di n caratteri, chiamiamolo sempre aa, per modificarlo completamente la chiamata è a_to_A(aa, 1).

Ci interessa quindi <u>la complessità della chiamata</u> a_to_A(aa, 1).

Sembra ragionevole usare come parametro n , cioè la dimensione dell'array.

<u>Attenzione:</u> come al solito abbiamo usato una notazione in cui n viene trattato come se fosse un parametro della procedura benché non sia dichiarato come tale (cfr Seconda Puntata, Sez.2.3). In questo caso n non deve mai essere pensato come una costante, anzi è proprio il parametro che stabilisce la dimensione dell'input.

Il modo più semplice ed elementare per capire il costo della chiamata consiste nel simulare l'esecuzione, seguendo quindi la catena di chiamate ricorsive, supponendo n abbastanza grande in modo da garantire un po' di passi.

Esecuzione di **a_to_A(aa, 1)**:

1. passaggio dei parametri
2. esecuzione di if 1 = n then {

   if aa[1] = 'a' then aa[1] ← 'A'

   a_to_A (aa, 2)

   }

Qui è tutto chiaro, tranne che per la chiamata ricorsiva; vediamola.

Esecuzione di **a_to_A (aa, 2)**:

1. passaggio dei parametri
2. esecuzione di if 2 = n then {

   if aa[2] = 'a' then aa[2] ← 'A'

   a_to_A (aa, 3)

   }

Eccetera, fino all'ultima chiamata.

Esecuzione dell’ultima chiamata **a_to_A (aa, n+1)**:

1. passaggio dei parametri
2. esecuzione di if n+1 = n then .... *qui si esegue solo il test e si esce.*

Venendo ai costi:

- costo della 1a chiamata **a_to_A (aa, 1)** = costi che sappiamo calcolare + costo della 2a chiamata
- costo della 2a chiamata **a_to_A (aa, 2)** = costi che sappiamo calcolare + costo della 3a chiamata
- costo della 3a chiamata **a_to_A (aa, 3)** = costi che sappiamo calcolare + costo della 4a chiamata
- eccetera.
- costo dell’ultima chiamata **a_to_A (aa, n+1)** = costi che sappiamo calcolare.

I costi che sappiamo calcolare, alla k-ma chiamata, con k < n+1:

- passaggio dei parametri: costo costante
- test k = n : costo costante
- if aa[k] = 'a' then aa[k] ← 'A' : costo costante
  (sia che si esegua il ramo then, che in caso contrario)

All’ultima chiamata, infine abbiamo un costo costante.

Quindi, ad ogni chiamata, tranne l’ultima, abbiamo (in questo algoritmo):

un costo costante + il costo della chiamata successiva.

Schematicamente:

successione di chiamate:	costo
a_to_A (aa, 1)	**c + (tutto quello che c’è sotto)**
↓	
a_to_A (aa, 2)	**c + (tutto quello che c’è sotto)**
↓	
....	**..**
a_to_A (aa, n)	**c + (tutto quello che c’è sotto)**
↓	
a_to_A (aa, n+1)	**d**

Sommando tutto si ha : n * c + d (con c, d costanti intere maggiori di 1).

Quindi se chiamiamo T la funzione complessità dell’algoritmo a_to_A : $\mathbf{T(n) \in \Theta(n)}$.

Riepilogando, il metodo consiste di:

- capire come si sviluppa la ricorsione; in questo caso abbiamo una semplice catena di chiamate; in casi più complicati avremo un "albero di chiamate";
- per una chiamata generica: calcolare il costo di tutto (passaggio dei parametri + esecuzione del corpo) tranne le chiamate ricorsive interne;
- capire i costi di eventuali chiamate "speciali" (qui è solo l'ultima);
- sommare tutto.

Una volta capito il metodo, nei casi semplici i conti si fanno in modo molto sbrigativo; nel nostro esempio:

- tutto costante, tranne le chiamate ricorsive;
- ci sono circa n chiamate;
- quindi: $T(n) \in \Theta(n)$.

11.2 . Ricerca binaria

Vediamo ora un esempio più interessante: ricerca binaria.

Ricordiamo che il problema è il seguente: dati un array ordinato ed un elemento, si vuole risposta vero/falso, a seconda che l'elemento appartenga / non appartenga all'array.

Per ulteriori dettagli vedere la parte di dispense dedicata alle procedure ricorsive.

Algoritmo ricorsivo (quello che avevamo chiamato ricbin_1):

```
function ricbin ( xx: float,  aa : array [1 ..n] of float, inf, sup : integer ) : boolean
var   med : integer
{if  inf = sup  then  return ( aa[inf] = xx )
    else   if  inf = sup-1  then  return ( (aa[inf] = xx)  or  (aa[sup] = xx) )
           else   {  med ← (sup + inf) div 2
                     if  aa[med] = xx  then  return (true)
                     else      if  aa[med] > xx
                               then      return ( ricbin (xx, aa, inf, med-1) )
                               else      return ( ricbin (xx, aa, med+1, sup) )
                  }
}
```

Quello che vogliamo calcolare è il costo della generica chiamata ricbin(xx, aa, 1, n).

(Poichè abbiamo una funzione, questa chiamata sarà all'interno di una espressione, ma dal punto di vista dei conti non ha importanza.)

Ma per farlo, a causa della ricorsione, dovremo capire il costo di ricbin(xx, aa, inf, sup) con $1 = inf = sup = n$.

Il costo della chiamata, dipende da xx ed aa :

- se $xx = aa[k]$ con k t.c. $n = 2k$ ottengo subito il risultato (vero)

- se xx non compare in aa allora “devo arrivare fino in fondo per scoprirlo”

  (lo stesso succede se xx c’è, ma è all’indice 1 oppure n).

  Però ci interessa il caso peggiore; supponiamo quindi che xx non appartenga ad aa (non è l’unico caso peggiore, ma è il più semplice).

  Anche qui è ragionevole considerare i costi in funzione della dimensione dell’array, cioè di n.

Costo di ricbin(xx, aa, 1, n) = costo del passaggio dei parametri
+ costo dell’esecuzione del corpo della funzione

Il costo del passaggio dei parametri è costante, se l’array è passato per riferimento (come avviene in C).

Per capire il costo del corpo, ricordiamo che l’ipotesi è che xx non compare nell’array, quindi i test di uguaglianza produrranno sempre “falso”; inoltre, come al solito, prendiamo n abbastanza grande.

C’è però il problema che, a seconda del valore di xx e dell’elemento “di mezzo”, si va a destra, oppure a sinistra. Si vede subito, però, che dal punto di vista dei costi non fa differenza.

Inoltre, ad ogni passo si scarta un elemento (quello con indice med) e poi si divide in due e la divisione non è sempre esatta (questo succede quando rimane un numero dispari 2k+1 di elementi; se ne prendono k a sinistra e k+1 a destra).

Per non perdersi nei dettagli, conviene prima ragionare sull’idea intuitiva dell’algoritmo, sempre nell’ipotesi che xx non sia nell’array, trascurando il fatto che si toglie un elemento e supponendo che l’array si divida bene.

Schematicamente:

Successione delle chiamate:	costo di ciascuna chiamata senza le chiamate interne	
ricbin su tutto aa	costante	(1 - vedere oltre)
ricbin su mezzo aa (circa)	costante	(1)
ricbin su un quarto di aa (circa)	costante	(1)
....	....	
ricbin su uno o due elementi di aa	costante	(2)

(1)Costo costante perchè si tratta del costo

- del passaggio dei parametri (che abbiamo visto essere costante)
- dell’esecuzione del corpo, senza le chiamate interne; per la precisione delle seguenti istruzioni (dove inf e sup hanno i valori opportuni, con sup - inf > 2):

```
if  inf = sup  then ....          questo ramo then non si esegue
else      if  inf = sup-1  then  ....     anche  questo  ramo then non si esegue
          else   {  med ← (sup + inf) div 2
                    if  aa[med] = xx   then .... questo  then  non si esegue, per ipotesi su
  xx
                    else      if  aa[med] > xx
```

qui c'è l'esecuzione di una (ed una sola) delle due chiamate ricorsive; il cui costo però NON viene contato qui.

```
                 }
```

(2)Costo costante perchè si tratta del costo

- del passaggio dei parametri (che abbiamo visto essere costante)
- dell'ultima esecuzione del corpo; per la precisione delle seguenti istruzioni (dove inf e sup sono tali che   inf = sup, oppure inf = sup-1):

```
if  inf = sup  then  return ( aa[inf] = xx )
else      if  inf = sup-1  then  return ( (aa[inf] = xx)  or  (aa[sup] = xx) )
          else   .....     non si esegue
```

Resta solo da capire quante sono le chiamate. Vediamo prima un conto approssimativo:

Ad ogni passo: si prende la metà (circa) dgli elementi dell'array; quindi:

1a chiamata: su n elementi
2a chiamata su n/2 elementi (circa)
3a chiamata su n/4 elementi (circa)
....
i-ma chiamata su n/p elementi (circa) con $p = 2^{i-1}$
....
k-ma chiamata su 1 o 2 elementi

Nell'ultima chiamata, la k-ma, in base a quanto visto per il generico passo i-mo, dobbiamo avere:

$1 = n/p$ oppure $2 = n/p$ con $p = 2^{k-1}$ quindi $k \approx \log_2 n$.

Il costo totale è dunque $c*k + d$ con: c, d costanti intere positive, $k \approx \log_2 n$.

Quindi, se T è la funzione complessità: $T(n) \in \Theta(\log n)$.

Ricordiamo che quando si usano gli ordini di grandezza, la base del logaritmo non è rilevante.

Siamo arrivati al risultato attraverso tutta una serie di "circa". Si possono fare le cose con un po' più di precisione. Vediamolo e contemporaneamente vediamo un'altro modo di impostare le cose, definendo T(n) per induzione (si dice anche: usando una *ricorrenza*, oppure usando una *relazione di ricorrenza* o *equazioni (sistemi) alle ricorrenze*).

11.2.1 . Definizione induttiva di T(n) per l'algoritmo ricbin.

Sempre nell'ipotesi che l'elemento xx non sia nell'array, in base all'analisi precedente (sul passaggio dei parametri e l'esecuzione del corpo di ricbin), abbiamo per T il seguente sistema, con a', a , b costanti opportune (intere e positive):

$$T(1) = a$$
$$T(2) = a'$$
$$T(n) = T((n-1)/2) + b \qquad \text{per } n > 2$$

L'ultima riga è esatta se (n-1)/2 è intero. Poichè poi dobbiamo iterare, e prendere, ad esempio: ((n-1)/2 - 1) / 2 , supponiamo che "n abbia la forma giusta" per poterlo fare.

Si vede che va bene n della forma 2^k-1 (infatti: togliendo 1 e dividendo per 2 ritroviamo un numero della stessa forma).

Con questa ipotesi su n, il sistema è esatto. Possiamo risolverlo, per sostituzioni successive.

$$T(n) = T\left(\frac{n-1}{2}\right) + b =$$

$$= T\left(\frac{\frac{n-1}{2} - 1}{2}\right) + b + b = \text{(semplificando)} \quad = T\left(\frac{n-3}{4}\right) + 2b =$$

$$= T\left(\frac{\frac{n-3}{4} - 1}{2}\right) + b + 2b = \text{(semplificando)} \quad = T\left(\frac{n-7}{8}\right) + 3b =$$

....

$$= T\left(\frac{n-(2^i-1)}{2^i}\right) + ib \qquad \text{(all' i-mo passo)}$$

....

$$= T\left(\frac{n-(2^{k-1}-1)}{2^{k-1}}\right) + (k-1)b \quad = \text{(semplificando, poichè : } n-(2^{k-1}-1) = 2^{k-1} \text{)}$$

$$= T(1) + (k-1)b =$$

$$= a + (k-1)b$$

Poichè $k \approx \log_2 n$ si conclude: $T(n) \in \Theta(\log n)$.

Tutto questo con l'ipotesi: $n = 2^k-1$.

Per un n generico:

- esiste sempre k tale che $2^k-1 = n = 2^{k+1}-1$; quindi: $k \approx \log_2 n$ e $\Theta(k) = \Theta(\log n)$;

  la funzione T è crescente (questo non lo dimostriamo, ma è "nella natura delle cose": nel caso di ricerca binaria la complessità non può diminuire al crescere di n); dunque:

  $T(2^k-1) = T(n) = T(2^{k+1}-1)$;
- poichè $T(2^k-1) \in \Theta(\log(2^k-1)) = \Theta(k)$ e $T(2^k-1) = T(n)$ otteniamo: $T(n) \in \Omega(k)$
- poichè $T(2^{k+1}-1) \in \Theta(\log(2^{k+1}-1)) = \Theta(k)$ e $T(n) = T(2^{k+1}-1)$ otteniamo: $T(n) \in O(k)$
- dunque: $T(n) \in \Theta(k) = \Theta(\log n)$.

Questo conclude la discussione sull’algoritmo di ricerca binaria.

> **Nota.** Il “trucco” di considerare un n particolare, che ci aiuta a fare i conti in modo comodo, si ritrova in altri casi. La tecnica per estendere il risultato ottenuto ad un n generico è sempre del tipo di quello visto sopra. A questo punto, in pratica, quest’ultimo passaggio si omette: si ragiona su un n della forma comoda ed il risultato lo si prende come risultato generale.

11.3 . Un esempio con due chiamate ricorsive

Questo è un esempio artificiale, costruito apposta per avere due chiamate ricorsive, ma tutti gli altri costi costanti, in modo che ci si possa concentrare sull’effetto della ricorsione; non è quindi un esempio di programmazione sensata.

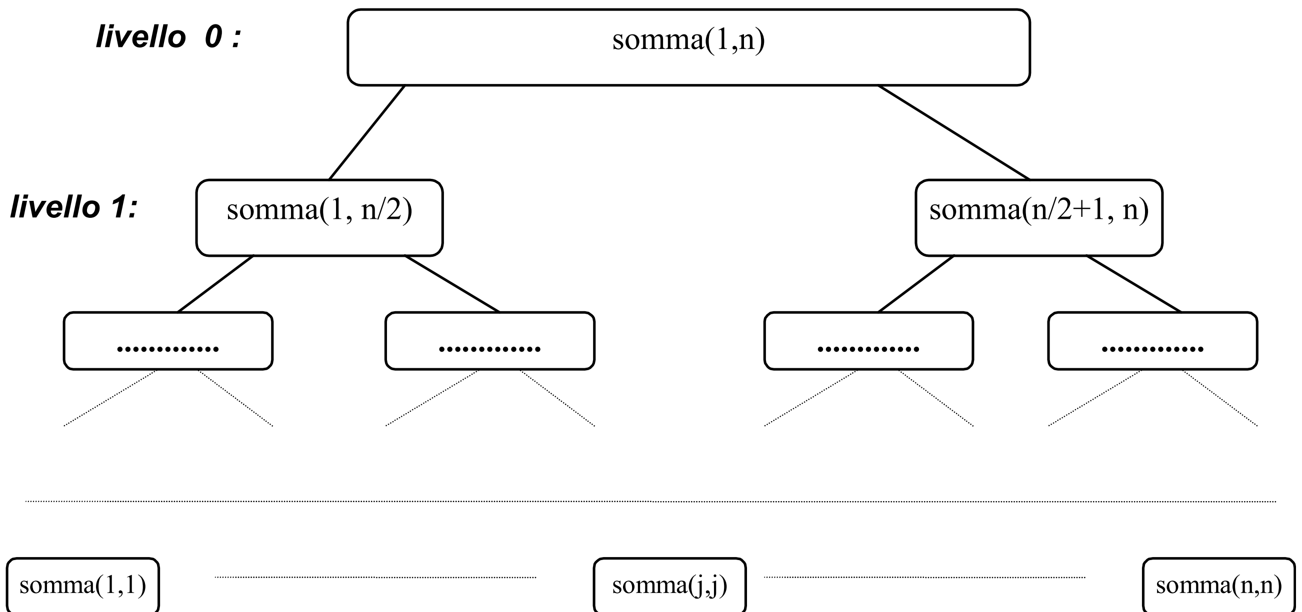
Si tratta di calcolare la somma dei valori contenuti in un array, seguendo lo schema “divide et impera” già visto per merge sort.

```
function somma (  aa : array [1 .. n] of integer,     inf, sup : integer  ) :  integer
        qui:  1 = inf = sup = n      e, per motivi di efficienza, l’array è passato per riferimento
var   med, s1, s2: integer
{  if   inf = sup    then  return (aa[inf])
   else   {  med ← (inf+sup) div 2
             s1 ← somma(aa, inf, med)
             s1 ← somma(aa, med+1, sup)
             return (s1 + s2)
          }
}
```

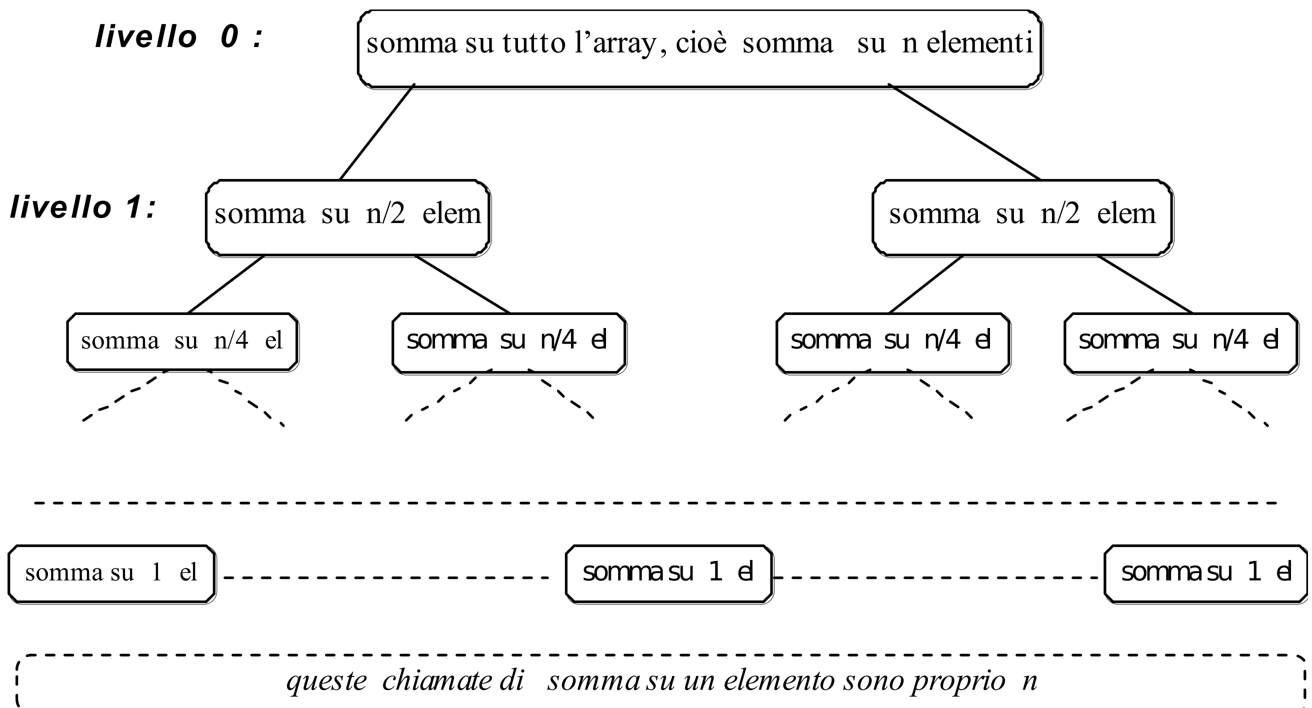
Vogliamo il costo di somma(aa, 1, n).

Per prima cosa, costruiamo “l’albero delle chiamate”. Per semplificarci la vita, supponiamo che “l’array si divida sempre bene”, cioè che n sia una potenza di 2: $n = 2^k$, per qualche k.

Allora abbiamo il seguente albero delle chiamate :



In effetti, possiamo semplificare l'albero, indicando solo, per ogni chiamata, su quanti elementi si lavora:



Notiamo che al generico livello i-mo, la procedura lavora su $n/2^i$ elementi; all'ultimo livello, dobbiamo avere $n/2^i = 1$ e quindi $i = k$. Dunque abbiamo un albero binario completo di altezza k; il numero totale dei nodi è $2^{k+1} - 1$; i nodi interni sono 2^k-1 e le foglie sono 2^k (cioè n).

Per ogni chiamata, cioè per ogni nodo dell'albero, calcoliamo i costi, trascurando le due sottochiamate ricorsive. Poichè l'array è passato per riferimento, il costo del passaggio dei parametri è costante. Pure costante è il costo delle altre istruzioni. In conclusione:

- costo delle chiamate di livello minore di k: b (costante intera > 1)
- costo delle chiamate di livello k: c (altra costante intera > 1)

A questo punto, per avere il costo totale basta fare un po' di somme e moltiplicazioni e si ottiene:

$b(2^k - 1) + c\,2^k$ Poichè $2^k = n$, il tutto è in $\Theta(n)$.

Alternativamente, si possono sommare i costi per livello (qui non è comodo, ma è utile vederlo come allenamento per casi più complicati, vedere ad esempio merge sort, paragrafo successivo):

livello	**numero di chiamate**	**costo totale del livello** (cioè senza contare le chiamate ricorsive interne)
0	1	b
1	2	2b
2	4	4b
¦	¦	¦
i	2^i	2^i b
¦	¦	¦
k-1	2^{k-1}	2^{k-1} b
k	2^k (= n)	$2^k d = n d$

Sommando tutto, si ottiene:

$T(n) = (b + 2b + 4b + ... + 2^i b + ... + 2^{k-1} b) + d\,n$

Il termine

$B = b + 2b + 4b + ... + 2^i b + ... + 2^{k-1} b = b(1 + 2 + ... + 2^i + ... + 2^{k-1})$

si può semplificare, ricorrendo alle formule per le somme di potenze (vedi sotto), oppure ricollegando tutto al numero dei nodi interni (vedi sopra).

In conclusione, $T(n) \in \Theta(n)$.

11.3.1 Somme di potenze.

In effetti sono già state viste, ma tanto vale riprenderle.

Vogliamo calcolare: $s_k = a^0 + a^1 + a^2 + ... + a^i + ... + a^k$ con a reale positivo, a ? 1.

(Se a = 1 la somma è banale.)

Si vede subito che $s_{k+1} = s_k + a^{k+1}$ ma anche $s_{k+1} = 1 + a\,s_k$

Quindi $s_k + a^{k+1} = 1 + a\,s_k$ da cui, essendo a ? 1, si ha

[*] $\mathbf{s_k = (a^{k+1} - 1) / (a - 1)}$

Nei conti di complessità, la situazione che incontreremo più spesso è quella in cui la base, a, è una costante, mentre k è il parametro (o uno dei parametri, o qualcosa di legato ad uno dei parametri). Quindi è utile semplificare la formula passando agli ordini di grandezza e mettendo in rilievo il termine dominante a^k. La [*] si può riscrivere cosí:

$s_k = c\,a^k + d$ dove: $c = a / (a-1)$ e $d = 1 / (a-1)$

essendo c, d costanti, poichè lo è a, si conclude:

[**] $\mathbf{s_k \in \Theta(a^k)}$

11.4 Merge sort

Per una discussione sull'algoritmo, vedere la parte di dispense sulle procedure ricorsive.

```
procedura ms (    aa : array [1 .. n] of integer      parametro IN-OUT
                  inf, sup : integer     parametri IN che variano tra 1 ed n, con inf = sup
   )
var  med: integer

{  if  inf < sup
   then  {  med ← (inf+sup) div 2
            ms(aa, inf, med)
            ms(aa, med+1, sup)
            merge (aa, inf, med, sup)
         }
}
```

Come al solito, ci interessa il costo della chiamata ms(aa, 1, n), espresso in funzione di n; chiamiamo T la funzione. Per calcolarlo, ci serve quello della procedura merge.

Costo di merge.

Quello che serve è il costo della generica chiamata: merge (aa, inf, med, sup).

Andando a guardare il codice di merge si vede facilmente che, se indichiamo con k il numero di elementi che si considerano, cioè $k = sup - inf$ (a rigori sarebbe: $k = sup - inf + 1$, ma non fa differenza, per k grande) e con M la funzione complessità della merge abbiamo:

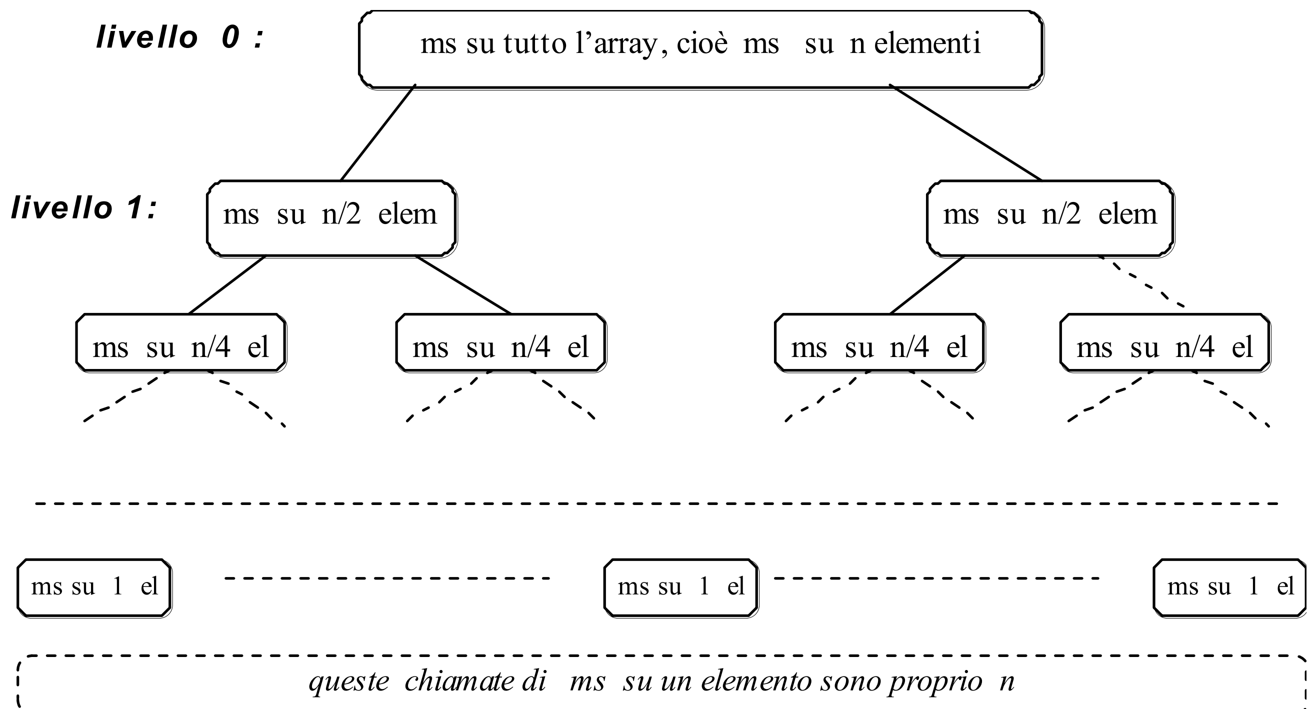
$M(k) = pk + q \in \Theta(k)$ (p e q sono costanti intere e positive).

A questo risultato si arriva anche considerando le operazioni dominanti, che sono: copiatura di elementi e confronto tra elementi, e ragionando a "livello di disegno". Infatti, nel caso peggiore:

- tutti (o quasi tutti) gli elementi vengono copiati da aa nell'array di supporto, sia bb;
- per ogni copiatura, c'è stato precedentemente un confronto tra elementi;
- poi si ricopia da bb in aa.

Quindi, nel caso peggiore, si hanno, circa: 2k copiature e k confronti.

Ritornando alla chiamata ms(aa, 1, n), abbiamo il seguente "albero di chiamate (ricorsive)":



Notare che nell'albero compaiono solo le chiamate alla procedura ms, non quelle a merge, perchè il costo di merge l'abbiamo già calcolato.

Notare anche che il disegno è esatto se n ha al forma 2^k; per ora supponiamo sia così.

Guardando il codice, si vede che il costo della generica chiamata ms(aa, inf, sup)

con $1 = inf < sup = n$,

senza considerare le due chiamate ricorsive interne,

dipende solo dal costo di merge (tutto il resto è costante; in particolare il passaggio dei parametri).

Quindi, ponendo $k = sup-inf$, si ha un costo lineare in k, cioè della forma: $ak + b$, con a e b costanti intere e positive. Allora, riferendosi all'albero di sopra, abbiamo la situazione seguente:

livello	**numero di chiamate**	**costo totale del livello** (cioè senza contare le chiamate ricorsive interne)
0	1	$a\,n + b$
1	2	$2 (a\,n/2 + b) = a\,n + 2b$
2	4	$4 (a\,n/4 + b) = a\,n + 4b$
|	|	|
i	2^i	$2^i (a\,n/2^i + b) = a\,n + 2^i\,b$
|	|	|
k-1	2^{k-1}	$a\,n + 2^{k-1}\,b$
k	2^k (= n)	$n\,d$

d costante intera > 0

Sommando tutto, si ottiene:

$T(n) = k\,a\,n + (b + 2b + 4b + ... + 2^i b + ... + 2^{k-1}b) + d\,n$

Il termine

$B = b + 2b + 4b + ... + 2^i b + ... + 2^{k-1}b$

si può semplificare ricorrendo alle formule per le somme di potenze, oppure collegandolo al numero di nodi interni nell'albero (vedere sezioni precedenti) e si vede che $B \in \Theta(2^k) = \Theta(n)$.

In conclusione, il termine dominante in $T(n)$ è $k\,a\,n = (\log n)\,a\,n$

Dunque: $T(n) \in \Theta(n \log n)$.

Questo, se $n = 2^k$; per il caso generale si può procedere come per ricerca binaria, quindi

la complessità dell'algoritmo di merge sort è $T(n) \in \Theta(n \log n)$.

In alternativa all'analisi fatta sull'albero delle chiamate, si può usare una definizione induttiva della funzione complessità:

$T(1) = d$

$T(n) = 2\,T(n/2) + an + b$

Però, provando a risolvere per sostituzione, il modo più semplice di capire come vanno le cose è schematizzare la situazione usando un albero; si ricade quindi in quello che abbiamo visto.

11.5 . DFS ricorsiva su un albero.

Vediamo la complessità dell'algoritmo ricorsivo di visita DFS, nel caso di alberi ternari. Il ragionamento si estende facilmente al caso di alberi di altro tipo.

(Ricordiamo che abbiamo già discusso le visite non ricorsive.)

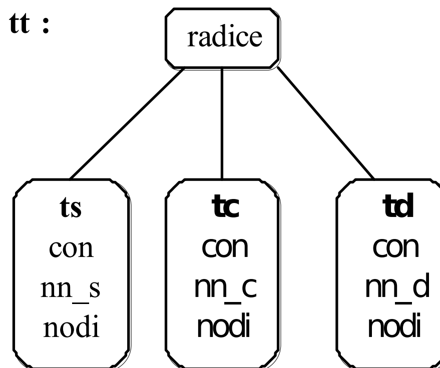
L' algoritmo ricorsivo è dato dalla procedura dfs (che nelle pagine dedicate alle visite avevamo chiamato dfs_2):

```
procedura dfs ( tt : albero_ternario  --  parametro IN)
    istruzioni:
    {  if   tt  non è vuoto
       then  {   visita la radice di tt, sia root(tx)
                 dfs ( ts )        dove  ts  è il sottoalbero di sinistra
                 dfs ( tc )        dove  tc  è il sottoalbero di centro
                 dfs ( td )    dove  td  è il sottoalbero di destra
             }
    }
```

Vediamo la complessità della (generica) chiamata dfs(tt).

Come per l'algoritmo non ricorsivo, supponiamo, per cominciare, di conoscere il numero di nodi di tt e chiamiamolo nn e supponiamo inoltre che il costo di "visitare un nodo" sia costante.

albero tt :



Tutto quello che sappiamo è chenn = 1 + nn_s + nn_c + nn_d.

In particolare uno dei tre sottoalberi potrebbe essere molto più grande degli altri; quindi l'albero delle chiamate generato dalla chiamata dfs(tt) non è, in genere bilanciato come quelli che abbiamo visto nel caso di ricerca binaria o merge sort (ed è proprio per questo che l'esempio è interessante).

Però, esaminando l'algoritmo, si vede che il costo del passaggio dei parametri è costante, quindi:

- il costo di dfs(tt) è: costo della visita di un nodo + costo delle 3 chiamate ricorsive;
- abbiamo supposto che il costo di visitare un nodo sia costante;
- per ogni nodo dell'albero, c'è una chiamata; quindi l'albero delle chiamate ricorsive generate dalla chiamata dfs(tt) ha la stessa forma di tt.

Quindi, per calcolare il costo totale, basta pensare di avere un costo costante per ogni nodo di tt e sommare il tutto.

Risultato: la complessità T_nodi(nn) è in $\Theta(nn)$, come nel caso non ricorsivo.

Supponiamo ora di <u>conoscere solo l'altezza</u> di tt, sia h.

Ragionando esattamente come nel caso non ricorsivo, si ottiene: $T_alt(h) \in \Theta(3^h)$.

La situazione: **costo di visitare un nodo non costante**, si affronta come nel caso non ricorsivo.

11.6

. Ancora un esempio di algoritmo su alberi

Consideriamo le seguenti dichiarazioni

```
typedef   struct nodo * albero;
struct  nodo { int info;
                albero left, right;   /* left = sinistra,  right = destra */
         };


void  Twist(albero tt) {
                 if (tt != NULL)  {  Twist(tt -> left);

                              { /*ora scambia i due figli: */
                                albero   aux;
                                aux = tt ->left;
                                tt ->left  = tt ->right;
                                tt -> right = aux;
                               }
                    }   /* chiude if-then */
   } /* chiude Twist */
```

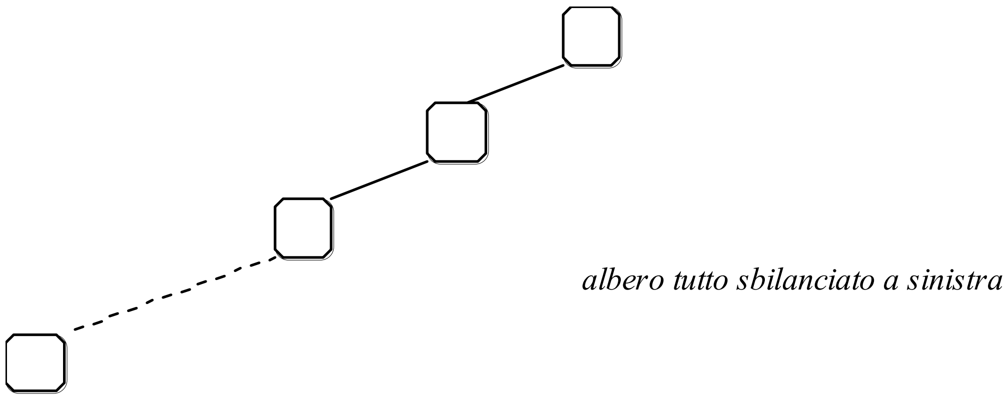
Vediamo solo il costo di Twist (tt) in funzione del numero di nodi, sia n, di tt. Abbiamo:

- passaggio del parametro: costo costante
- corpo della procedura: costo costante, se trascuriamo la chiamata ricorsiva

Si tratta quindi solo di capire quante sono le chiamate

Chiamata su albero:	numero di nodi dell'albero:	
tt	n	
tt -> left	n_1	con $n_1 < n$
(tt -> left) -> left	n_2	con $n_2 < n_1$
eccetera		

Non conosciamo il valore di $n_1, n_2, \ldots$, però, nel caso peggiore la situazione è la seguente:



albero tutto sbilanciato a sinistra

Allora le chiamate ricorsive sono n e la complessità nel caso peggiore è in $\Theta(n)$.

11.7

. L’esempio del fattoriale (e problemi di “dimensione”)

Vediamo l’esempio del fattoriale, non perchè sia interessante il calcolo di complessità (che è banale), ma per capire meglio la questione “dimensione dell’input” o “dimensione dei dati” o “dimensione del problema”.

Ricordiamo che: se n è intero positivo : $n! = n (n-1) (n-2)....1$ e inoltre: $0! = 1$.

Per gestire il caso di input errato (cioè < 0), usiamo il valore -1 come “valore speciale” che segnala errore.

Algoritmo ricorsivo:

```
function fatt_ric( n:  integer) : integer
   {     if  n < 0  then  return (-1)          messaggio di errore e si esce
         else   if   n = 0   then    return(1)     else    return( n * fatt_ric(n-1) )
   }
```

Algoritmo iterativo:

```
function fatt_it (n:  integer) : integer
   {     var   integer  res, k
         if  n < 0  then  return (-1)
         else        {      res ← 1
                            per  k = 2,...,  n :   res ← k*res          si esegue solo per n >1
                            return( res )
                     }
   }
```

Complessità dell’algoritmo iterativo:

è facile vedere che il costo della chiamata fatt_it(n) in funzione di n è:

T_it(n) =	a	se $n < 0$	a costante intera > 0
T_it(n) =	b	se $n = 0, 1$	b costante intera > 0
T_it(n) =	$c(n-1) + d$	se $n > 1$	c, d costanti intere > 0

Complessità dell’algoritmo ricorsivo:

usando T_ric(n) per indicare la complessità della chiamata fatt_ric(n) in funzione di n, si ha subito la seguente definizione induttiva (con p, q, r costanti):

T_ric(n) =	p	se $n < 0$	p costante intera > 0
T_ric(n) =	q	se $n = 0$	q costante intera > 0
T_ric(n) =	r + T_ric(n-1)	se $n > 0$	r costante intera > 0

Risolvendo, per sostituzioni successive, l’ultima riga diventa:

T_ric(n) = $r + r + ... + r + q = n r + q$

Quindi:

T_ric(n) = p   se $n < 0$ p costante intera > 0

r n + q   se n = 0   q, r costanti intere > 0

Sostanzialmente, la complessità, sia nella versione iterativa che in quella ricorsiva, è lineare in n per n positivo o nullo, mentre è costante per n negativo (o nullo).

Il punto però è un altro: è corretto misurare la complessità in funzione di n ?

In effetti, facendo così stiamo misurando la complessità in funzione dell'argomento (= input), cioè n, invece che della dimensione dell'input: la lunghezza della stringa di cifre che rappresenta n.

Sia lg la lunghezza della stringa di cifre che rappresenta n. Sappiamo che:

- $lg \approx 1 + \log_2 |n|$ se usiamo la rappresentazione binaria
- $lg \approx 1 + \log_{10} |n|$ se usiamo la rappresentazione decimale.

  dove |n| è il valore assoluto di n e "1" tiene conto del segno.

Sia   TL_it(lg)   il costo dell'algoritmo iterativo per il fattoriale, in funzione della lunghezza dell'argomento. In base alla definizione (caso peggiore):

TL_it(lg) = max { T_it(n) | n tale che lg = lunghezza della stringa per n }

= T_it(max_n_lg)

dove max_n_lg = massimo numero positivo rappresentabile con una stringa di lunghezza lg.

Del tutto analogo il ragionamento nel caso ricorsivo. Allora:

- TL_it(lg) = T_it(2^{lg-1} - 1) $\in \Theta(2^{lg})$ nel caso binario
- TL_it(lg) = T_it(10^{lg-1} - 1) $\in \Theta(10^{lg})$ nel caso decimale
- TL_rec(lg) = T_rec(2^{lg-1} - 1) $\in \Theta(2^{lg})$ nel caso binario
- TL_rec(lg) = T_rec(10^{lg-1} - 1) $\in \Theta(10^{lg})$ nel caso decimale

Abbiamo quindi che, rispetto alla dimensione dell'input, il fattoriale ha un costo esponenziale!

Osservazioni.

1. E` importante non fare confusioni: i conti fatti per dare una stima della funzione T(n) vanno benissimo; il calcolo della complessità in funzione di n è utile e fornisce alcune informazioni; però, la valutazione della complessità in funzione della dimensione dell'input ci fornisce una stima di complessità più adeguata.
2. Tutto questo discorso porta a riguardare gli esempi visti in precedenza e chiedersi se abbiamo fatto qualche "errore". In effetti, la maggior parte degli esempi visti riguarda problemi su array o alberi. La dimensione di un array è data dal numero di elementi e dalla dimensione dei singoli elementi, analogamente per gli alberi. Nel caso di elementi con dimensione fissata (caratteri, numeri [per i quali la dimensione può essere arbitraria a livello di problema, ma è fissata nella realtà dei programmi], record di piccole dimensioni) la dimensione dei singoli elementi può essere trascurata; nel caso in cui la dimensione dei singoli elementi non è fissata, va considerata come ulteriore parametro; vedere a proposito la discussione fatta relativamante ad insertion sort, nel caso gli elementi siano stringhe.

11.8 . Numeri di Fibonacci

Questo è un esempio in cui è facile ottenere la valutazione in O ed in Ω, mentre quella in Θ è difficile (è per questo che lo vediamo e per le analogie con il problema di calcolare i coefficienti binomiali).

Consideriamo la seguente definizione:

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$ (per $n > 2$)

Questa definizione definisce una successione di numeri, detti numeri di Fibonacci:

$F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, ..., F_{15} = 610, F_{16} = 1597,....$

È una successione a crescita molto rapida; per una spiegazione, legata al ritmo di riproduzione dei conigli, vedere: Fabrizio Luccio, La struttura degli algoritmi, Boringhieri, pag 74 (libro presente in Biblioteca). Volendo, si può dare una definizione alternativa partendo da $F_0 = 0$.

Per arrivare in modo semplice ad un algoritmo per calcolare questi numeri (o meglio per calcolare l'ennesimo) conviene trasformare le definizione precedente in una definizione di funzione. Usando $\mathbf{N}^+$ per indicare gli interi positivi:

Fib : $\mathbf{N}^+ \to \mathbf{N}^+$ è data (induttivamente) da

Fib(1) = 1

Fib(2) = 1

Fib(n) = Fib(n-1) + Fib(n-2) (per $n > 2$)

A questo punto, è immediato scrivere un algoritmo ricorsivo per Fib. Per non complicarci la vita con il caso che l'argomento sia errato (vedi caso del fattoriale), supponiamo di chiamare la funzione sempre con input positivo.

```
function  fib (n : integer) : integer        si assume  n > 0
      {  if   ( n = 1  or  n = 2)    then  return (1)
         else  return (fib(n-1) + fib(n-2))                 }
```

Per prima cosa calcoliamo la complessità in funzione dell'argomento n (anche se, dopo quanto abbiamo visto per il fattoriale, dovremo poi passare alla lunghezza di n).

Dal codice si ha subito la seguente definizione induttiva per la funzione T:

$T(1) = T(2) = a$ con a e b costanti intere > 0

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

Non è facile risolvere esattamente questo sistema e trovare una stima in $\Theta(...)$.

In mancanza di meglio, si può cercare di determinare un limite inferiore ed uno superiore. Possiamo ragionare così:

T_inf(n) = T(n) = T_sup(n) dove:

T_sup(1) = T_sup(2) = a

T_sup(n) = 2 T_sup(**n-1**) + b

e

T_inf(1) = T_inf(2) = a

$T_inf(n) = 2\ T_inf(\mathbf{n-2}) + b$

Quindi: la base è sempre la stessa, ma in T_inf supponiamo di avere due chiamate su n-2 elementi, mentre in T_sup supponiamo due chiamate su n-1 elementi.

Risolvendo i due sistemi per sostituzione, si ottiene (per n > 2):

$$\begin{aligned}
T_sup(n) &= 2\ T_sup(n-1) + b \\
&= 2\ (\ 2\ T_sup(n-2) + b) + b = 4\ T_sup(n-2) + 3b \\
&= 4\ (\ 2\ T_sup(n-3) + b\ ) + 3b = 8\ T_sup(n-3) + 7\ b = 2^3\ T_sup(n-3) + (2^3-1)b \\
&= \ldots\ldots\ldots\ldots \\
&= 2^{n-2}\ T_sup(2) + (2^{n-2}-1)b = 2^{n-2}\ a + (2^{n-2}-1)b
\end{aligned}$$

Quindi: $T_sup(n) \in \Theta(2^n)$

$$\begin{aligned}
T_inf(n) &= 2\ T_inf(n-2) + b \\
&= 4\ T_inf(n-4) + 3b \\
&= 8\ T_inf(n-6) + 7\ b \\
&= \ldots\ldots\ldots\ldots \\
&= 2^k\ T_inf(n-2k) + (2^k-1)\ b \\
&= \ldots\ldots\ldots\ldots \\
&= 2^p\ T_inf(z) + (2^p-1)\ b \qquad \text{dove: } p \approx n/2 \text{ e } z \text{ è } 1\text{, oppure } 2
\end{aligned}$$

Quindi: $T_inf(n) \in \Theta(2^{n/2}) = \Theta(\ (\sqrt{2})^n\ )$.

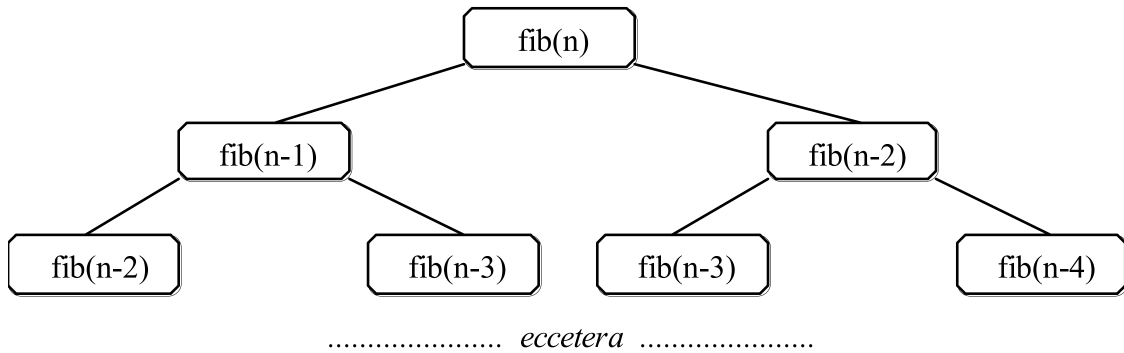
Allora : $T(n) \in \Omega(\ (\sqrt{2})^n\ ) \cap O(2^n)$.

Notare: $T_inf(n) = T(n)$ e $T_inf(n) \in \Theta(\ (\sqrt{2})^n\ )$ implica $T(n) \in \Omega(\ (\sqrt{2})^n\ )$ e
$T(n) = T_sup(n)$ e $T_sup(n) \in \Theta(2^n)$ implica $T(n) \in O(2^n)$
e, da quanto sopra, non si può concludere di più (cioè non si può concludere una stima in $\Theta(....)$ per T).

11.9 . Approccio bottom-up a Fibonacci

Ma è veramente così costoso calcolare i numeri di Fibonacci ? NO.

Guardando l'albero delle chiamate si vede subito che si calcolano più volte i medesimi numeri.



.... *eccetera* ....

Il fatto è che, ad esempio, le due chiamate fib(n-2) che si vedono nel disegno, vengono eseguite indipendentemente; quindi per calcolare F_n, i numeri F_k, con k < n, si calcolano tante volte (anzi, tantissime volte!), con grandissimo spreco di tempo. Per avere un'idea:

F_{45} = 1 134 903 170 e, nel calcolo, F_k viene calcolato num_k di volte dove, ad esempio:

k	num_k
1	433 494 437
2	701 408 733
3	433 494 437
4	267 914 296
5	165 580 141
6	102 334 155
7	63 245 986
8	39 088 169
9	24 157 817
10	14 930 352
11	9 227 465
12	5 702 887
13	3 524 578
14	2 178 309
15	1 346 269
16	832 040

Per esercizio: scrivere un programma che, preso n, stampa in output una tabellina come quella di sopra; però completa: con k = 1,...., n. (Serve una piccola modifica alla funzione fib ....)

C'è un modo più efficiente di fare le cose, seguendo un approccio "dal basso verso l'alto", in inglese *bottom-up*. L'algoritmo ricorsivo procede in modo *top-down* (dall'alto verso il basso): si parte da quello che si vuole calcolare, il numero F_n, e seguendo la definizione induttiva si "scende" a calcolare i numeri di indice inferiore che servono.

Nell'approccio bottom-up, si parte calcolando F_1, F_2, ... , F_k, ... fino ad arrivare a F_n.

Il punto chiave è che questi numeri si calcolano una volta sola! Questo è un bel risparmio, perchè i numeri da F_1 a F_n sono (ovviamente) solo n. Vedere sotto per un algoritmo.

Se tutto questo servisse solo per i numeri di Fibonacci non sarebbe molto interessante; invece ci sono altri casi significativi in cui si ha un fenomeno analogo. Riprenderemo questo argomento perlando di "tecniche di programmazione" e, in particolare, della "programmazione dinamica".

Algoritmo bottom-up (esempio elementare di tecnica della programmazione dinamica).

L'idea è di partire dai primi numeri, calcolarli, memorizzarli, ...

Usiamo due variabili intere: dispari che serve a memorizzare, successivamente, F_1, F_3, ...
pari che serve a memorizzare, successivamente, F_2, F_4, ...

Facciamo un ciclo; ad ogni esecuzione del ciclo calcoliamo una coppia di numeri fino ad ottenere quello che vogliamo (magari anche il successivo).

(Ovviamente, questo non è l'unico modo di fare il calcolo partendo dal basso.)

Se k è la variabile di controllo del ciclo, lo schema è illustrato dalla tabella seguente:

	k	dispari	pari	
inizio:		$F_1 = 1$	$F_2 = 1$	
ciclo:	1	$F_3 = 2$	$F_4 = 3$	*valori **dopo** l'esecuzione del ciclo con k =1*
	2	$F_5 = 5$	$F_6 = 8$	*valori **dopo** l'esecuzione del ciclo con k =2*
	3	$F_7 = 13$	$F_8 = 21$	*valori **dopo** l'esecuzione del ciclo con k =3*
	4	$F_9 = 34$	$F_{10} = 55$	*valori **dopo** l'esecuzione del ciclo con k =4*
	5	$F_{11} = ...$	$F_{12} = ...$	*valori **dopo** l'esecuzione del ciclo con k =5*

Resta da capire come legare l'indice k all'indice della coppia di numeri che si calcolano; dalla tabellina si vede che al passo k = i si calcolano i numeri F_{2i+1} ed F_{2i+2}

Quindi possiamo scrivere l'algoritmo nella forma seguente:

```
function  fib_bu (n : integer) : integer
      {  var   dispari, pari, k, sup : integer
         if   n < 1  then  return(-1)          è un messaggio di errore
         else
               if   ( n = 1  or  n = 2)    then  return (1)
               else   {  pari ← 1  ;      dispari ← 1
                         if  (n è pari)  then  sup ← (n-2) div 2   else   sup ← (n-1) div 2
                         per  k: 1, 2, ..., sup  :    {  dispari ← dispari + pari
                                                         pari ← pari + dispari        }
                         if  (n è pari)   then  return (pari)  else return(dispari)
                      }
      }          notare che il test  (n è pari) si può scrivere come:  (n mod 2 = 0)
```

È facile vedere che la complessità del nuovo algoritmo è lineare in n, cioè

il costo di fib_bu(n) è in $\Theta(n)$.

Usando un ragionamento analogo a quello visto per il fattoriale, si vede, però, che il costo è esponenziale rispetto alla dimensione di n.

L'algoritmo ricorsivo, misurato in funzione della dimensione di n, ha invece un costo doppiamente esponenziale: in $\Omega((\sqrt{2})^p)$, con $p = 2^{lg}$, oppure $p = 10^{lg}$, se lg = dimensione di n.

11.10. Definizioni induttive e ricorsione.

L'esempio di Fibonacci mette in luce il collegamento e la distinzione tra definizione induttiva di una funzione e algoritmi ricorsivi per il calcolo della funzione medesima.

La definizione induttiva della funzione Fib, sia dal punto di vista matematico che da quello della programmazione, si può utilizzare in due modi.

1. **Top-down.** Volendo calcolare Fib(n), calcoliamo Fib(n-1) e Fib(n-2); per calcolare Fib(n-1), calcoliamo Fib(n-2) e Fib(n-3),....... fino ad arrivare ai casi di base: Fib(1) e Fib(2). Nel fare questo, possiamo essere abbastanza furbi da non calcolare due volte la stessa cosa. Questo modo di usare la definizione induttive è quello che corrisponde all'algoritmo ricorsivo che si deriva banalmente dalla definizione stessa. Solo che nell'algoritmo ricorsivo non c'è modo di essere furbi.

2. **Bottom-up.** Volendo calcolare Fib(n), partiamo dalla base e, successivamente, calcoliamo tutti i valori Fib(k) fino ad arrivare a Fib(n). Pensandoci, questo corrisponde alla nostra formalizzazione delle definizioni induttive di insiemi (usando le successioni di insiemi). Questo approccio si può tradurre in un algoritmo iterativo (cioè che usa cicli while o for, invece della ricorsione).

Questo discorso non è limitato al caso dei numeri di Fibonacci, ma vale in generale. Una definizione induttiva di funzione si traduce naturalmente in una algoritmo ricorsivo per il calcolo, top-down, della funzione. Dal punto di vista della correttezza questo è del tutto soddisfacente; dal punto di vista dell'efficienza non è detto, bisogna controllare. In genere è possibile ricavare anche un algoritmo iterativo; solo è un po' più difficile (alle volte un po' tanto). Dal punto di vista dell'efficienza, però, ci sono spesso dei vantaggi (non sempre, basta guardare il caso del fattoriale dove le complessità sono identiche).

In conclusione, volendo un algoritmo per una funzione definita induttivamente:

- per prima cosa si considera l'algoritmo ricorsivo e si stima la sua complessità; se è ragionevole, si usa;
- altrimenti si cerca di ricavare quello iterativo bottom-up, facendo attenzione a scriverlo correttamente (perchè è, in genere, più difficile da ricavare); si stima la sua complessità; se è migliore di quello ricorsivo allora si utilizza, altrimenti .... si va a simpatia (infatti, a meno di non aver fatto pasticci, la complessità non aumenta rispetto a quello ricorsivo).

Nelle sezioni seguenti, parlando di tecniche di programmazione, vedremo che una situazione analoga si può avere nel risovere un problema.

12

. Tecniche di programmazione

Ci limitiamo a vedere alcune varianti di "approccio induttivo", sempre su esempi. Alcuni sono stati visti, altri sono nuovi; in parte si tratta solo di catalogare e precisare tecniche già usate.

Qui ci interessa l'induzione come metodo per risolvere problemi, quindi non citiamo, tra gli esempi, gli algoritmi per calcolare funzioni definite induttivamente. In questi casi, infatti, non c'è nessun problema da risolvere; si deve solo tradurre la definizione induttiva in un algoritmo ricorsivo (approccio top-down), o in un algoritmo iterativo (approccio bottom-up), come visto in precedenza parlando dei numeri di Fibonacci.

Il vantaggio principale dell'approccio induttivo è legato alla *correttezza*:

- Se i dati sono definiti in maniera induttiva, un algoritmo che segua tale definizione garantirà di trattare tutti i vari casi che si possono presentare;
- Spesso si riesce a utilizzare la struttura induttiva del'algoritmo per dimostrarne la correttezza con un metodo di diostrazione induttiva (a passi o generalizzata).

12.1 . Approccio collegato alla definizione induttiva dei dati.

Lo abbiamo visto soprattutto nel caso degli alberi: algoritmi ricorsivi di visita, ricerca, ...

Non è detto che sia sempre praticabile (ad esempio, non si applica alla visita per livelli), ma è uno dei primi tentativi da fare.

Nota. Nel problema di cercare un certo valore x tra le etichette dei nodi di un albero t, gli input sono due: x e t; alternativamente, possiamo dire che l'input ha due componenti: x e t . Nell'algoritmo, si sfrutta solo la struttura induttiva di t. Questo succede spesso; anche in quello che segue, l'idea induttiva non si applica necessariamente a tutte le componenti dell'input, ma solo ad alcune.

12.2 . Approccio ispirato all'induzione a passi.

Dato un problema P con input di dimensione n (oppure: una componente dell'input ha dimensione n; oppure: una delle dimensioni che caratterizzano l'imput è n) proviamo a vedere se:

- sappiamo risolvere P con input piccolo (es: $n = 0$, $n = 1$, ...);
- supponendo di aver risolto P per $n = k$, riusciamo a risolvere P per $n = k+1$ (utilizzando la soluzione per il caso $n = k$).

Esempio: permutazioni di n elementi.

P è: generare e stampare tutte le permutazioni di n elementi distinti : $e_1, e_2, ..., e_n$

(essendo gli n elementi distinti, le permutazioni sono esattamente $n!$).

Se $n = 1$ è facile, c'è una sola permutazione: $< e_1 >$.

Se abbiamo generato tutte le permutazioni dei primi k elementi e le abbiamo memorizzate, ad esempio in un insieme Per_k , possiamo generare quelle dei primi k+1 elementi come segue:

da ogni $< x_1, x_2, ..., x_k >$ in Per_ k otteniamo k+1 permutazioni di $e_1, e_2, ..., e_{k+1}$:

$< e_{k+1}, x_1, x_2, ..., x_k >$

$< x_1, e_{k+1}, x_2, ..., x_k >$

....

$< x_1, x_2, ..., x_k, e_{k+1}>$

Quindi l'algoritmo consiste nel generare, successivamente, gli insiemi Per_1, Per_2,.... fino a Per_n e poi stampare gli elementi di quest'ultimo.

Correttezza dell'algoritmo.

E` immediato vedere che generiamo solo permutazioni corrette. Meno immediato è capire che le otteniamo tutte. La dimostrazione è (ovviamente) per induzione aritmetica a passi. Il predicato è:

Q(n) = il nostro metodo genera tutte le permutazioni di $e_1, e_2, ..., e_n$

base: per n = 1 il predicato è ovviamente vero;

passo: supponiamo Q(n-1) vero.

Questo vuol dire che Per_n-1 contiene tutte le permutazioni di $e_1, e_2, ..., e_{n-1}$.

Sia $p = < y_1, y_2, ..., y_n >$ una qualunque permutazione di $e_1, e_2, ..., e_n$

Supponiamo che e_n compaia al posto j-mo, cioè $e_n = y_j$.

Allora $q = < y_1, ..., y_{j-1}, y_{j+1}, ..., y_n >$ è una permutazione di $e_1, e_2, ..., e_{n-1}$.

Per ipotesi induttiva q è in Per_n-1; dunque, usando il nostro metodo prima o poi la consideriamo e a partire da essa generiamo n permutazioni inserendo e_n al 1° posto, poi al 2°, ... Quando inseriamo e_n al j-mo posto, otteniamo proprio p.

Complessità dell'algoritmo.

L'algoritmo è corretto; però molto costoso, sia in termini di spazio, che in termini di tempo.

Come **tempo**: il numero di permutazioni generate, per k = 1, ..., n è: 1 + 2! + 3! + ... + n!

inoltre, generare una permutazione di lunghezza k ha un costo lineare in k.

Quindi, il costo totale è proporzionale a 1 + 2*2! + 3*3! + ... + n*n!

Si vede subito che l'algoritmo è in $\Omega(n * n!) \cap O(n^2 * n!)$. Con un po' di fatica si riesce a dimostrare che è in $\Theta(n * n!)$. D'altra parte, il problema chiede di stampare n! permutazioni (ciascuna di lunghezza n); quindi il problema è in $\Omega(n * n!)$. In conclusione, il nostro algoritmo è ottimale.

Per quanto riguarda lo **spazio**: alla fine del passo k-mo, abbiamo costruito e memorizzato, in qualche modo, l'insieme Per_k che ha k! elementi di lunghezza k; quindi ......

Altri esempi di questo tipo di approccio induttivo:

- algoritmo per lo "zaino decisionale", che vedremo;
- insertion sort, anche se non è stato presentato in quest'ottica (**per esercizio**, provare a riformulare la spiegazione dell'algoritmo in modo da esplicitare l'aspetto induttivo).

Per ulteriori esempi, vedere libro di Manber (citato in Sezione).

12.3 . Approccio ispirato all'induzione generalizzata - Divide et impera.

Generalizza quello collegato all'induzione a passi (che quindi è un caso particolare).

La base è la stessa. Poi, per risolvere il caso n = k+1 , supponiamo di aver risolto il problema per n = k, k-1, k-2, ..., 0 (non si va sotto zero, perchè una dimensione dell'input negativa non ha senso) e vediamo se possiamo sfruttare (alcune di) queste soluzioni per risolvere il caso n = k+1.

Gli esempi visti sono ricerca binaria e merge-sort.

Però, in genere, questi vengono citati come esempio della tecnica **divide et impera** (= dividi e domina). Il fatto è che la tecnica divide et impera è solo una diversa formulazione dell'approccio ispirato all'induzione generalizzata; questa formulazione è forse piú comoda quando si cerca di progettare algoritmi, quindi la seguiamo.

Nella ricerca binaria, dati un array aa di n elementi ed un elemento z, per vedere se z compare in aa:

se aa ha 1 o 2 elementi, risolviamo direttamente il problema;

altrimenti:

- dividiamo aa in due parti aa_1, aa_2; nel dividere controlliamo il "punto di mezzo"; se troviamo z abbiamo concluso, altrimenti:
- scegliamo la metà ci interessa, sia aa', e continuiamo su aa' usando la medesima tecnica;
- quando abbiamo il risultato della ricerca su aa', lo restituiamo come risultato della ricerca su tutto aa.

Nel merge sort, dato un array aa di n elementi, per ordinarlo:

se aa ha 1 elemento, risolviamo direttamente il problema (senza fare nulla);

altrimenti:

- dividiamo aa in due parti, siano aa_1 e aa_2 (qui non c'è nessuna possibilità di conclusione immediata);
- applichiamo la medesima tecnica ad aa_1 ed aa_2 (qui non c'è scelta);
- quando abbiamo "i risultati" che provengono da aa_1 e aa_2, li mettiamo assieme (fondendoli, tramite la procedura merge) per ottenere il risultato relativo a tutto aa (qui il "risultato" è l'array stesso modificato).

Lo schema generale si fomula meglio in termini di "risultato" più che di "soluzione".

Dato un problema P con input X e indicando con |X| la dimensione di X, lo schema si può formulare sotto forma di procedura ricorsiva:

```
divimp_P( X )
if      |X| = d           (con  d  che dipende da P)
then    risolvi direttamente  P su  X    e, chiamando  Y  il "risultato", return ( Y )
else  {  dividi      (d1)    da  X  ricava  X1, ...., Xm  (con |Xj| < |X|,  j = 1,...,m)
                     (d2)    if     dalla divisione si ottiene un "risultato"  Y
                             then  return ( Y )        e quindi non si esegue il resto
                     (d3)    scegli  gli Xj che interessano;
                             per non complicare la notazione, supponiamo che siano i primi  q
          impera       per  j = 1, ...., q :  divimp_P( Xj )  e sia  Yj  il "risultato"
        combina     Y ← opportuna combinazione di  Y1, ... ,Yq
                    return ( Y )
      }
```

Ritornando ai nostri esempi e usando le notazioni di sopra:

- merge sort: X = aa ; m = 2; non si può concludere in *(d2)*; q = 2 ; X1 = aa_1, X2 = aa_2 ; Y1 e Y2 sono X1 e X2 ordinati; la combinazione è la fusione;
- ricerca binaria: X = (aa, z) ; m = 2; X1 = (aa_1, z), X2 = (aa_2, z); è possibile concludere in *(d2)*; q = 1 (nel senso che si sceglie uno solo tra X1 e X2); la parte di combinazione è elementare: Y = Y1.

Ci sono esempi più complicati dove gli Xj non sono semplici "pezzi" di X; inoltre non sempre gli Xj corrispondono a "pezzi disgiunti" di X, potrebbe esserci una parziale sovrapposizione. Però non vediamo esempi di questo fenomeno (un esempio è la moltiplicazione rapida, vedere libro di Luccio, sezione 5.2.2.).

Quello che è importante sottolineare è che, comunque,

la dimensione degli Xj deve essere strettamente minore di quella di X,

altrimenti non non si "divide" e quindi non si "impera".

Complessità di divimp_P(X)

Se $n = |X|$ e $n_j = |Xj|$ abbiamo la seguente definizione induttiva della funzione complessità:

$T(n)$ = costante per $n = d$

$T(n) = D(n) + C(n) + T(n_1) + ... + T(n_q)$

dove $D(n)$ è il costo della fase "divisione" e $C(n)$ è il costo della fase "combinazione".

Studiando questo sistema di equazioni, si può vedere che $T(n)$ dipende da due fattori:

1. costi di divisione e combinazione; questo è ovvio, come è ovvio che più bassi sono meglio è;

2. bontà della divisione; questo è più delicato ed ha due aspetti:

2.1. dimensioni, cioè confronto tra n e $n_1 + ... + n_q$

2.2. bilanciamento tra le dimensioni $n_1, ... , n_q$

Il difficile è capire come queste tre cose interagiscono; non lo vediamo (per una discussione dettagliata, ed anche dei risultati precisi, vedere Cormen-Leiserson-Rivest, cap 4) e ci limitiamo a considerarle separatamente.

Per quanto riguarda le dimensioni:

- la situazione migliore si ha quando $n >> (n_1 + ... + n_q)$, cioè quando "si butta via una parte dei dati"; un esempio è ricerca binaria dove si butta via metà dei dati; vedere anche nota sotto;
- la situazione "neutra" si ha quando $n \approx (n_1 + ... + n_q)$, cioè si considerano (quasi) tutti i dati; un esempio è merge sort;
- la situazione peggiore si ha quando $n << (n_1 + ... + n_q)$, cioè c'è una sovrapposizione, anche parziale, tra gli Xj e quindi "i dati (alcuni dati) vengono considerati più volte"; di questa situazione non abbiamo fornito esempi (forse il più semplice degli esempi non artificiali è la "moltiplicazione rapida" già citata).

Nota: quando si butta via una parte dei dati, non è la stessa cosa buttare via "una frazione fissa dei dati", oppure buttare via "un numero fisso di dati".

Non è necessario fare esempi di algoritmi, basta considerare due possibili semplici sistemi, corrispondenti ad una sola chiamata e a costi di divisione e di composizione costanti:

T1(0) = a T2(0) = c

T1(n) = T1(n/k) + b T2(n) = T2(n-p) + b k e p >1

Risolvendo per sostituzione:

in T2 si arriva alla base, 0 elementi, in n/p passi (circa): n, n - p, n - 2p, ...

in T1 si arriva alla base in $\log_k n$ passi (circa): n, n/k, n/k^2, ...

Quindi: $T1(n) \in \Theta(\log n)$, mentre $T2(n) \in \Theta(n)$.

L'**effetto del bilanciamento** di un algoritmo divide-et-impera riguarda appunto la distinzione tra **dividere** e **sottrarre**. Vediamo con un esempio.

In una situazione dove m = q, consideriamo due algoritmi: il solito merge sort (in cui si divide, perché ad ogni chiamata ricorsiva si considerano metà (circa) dei dati della procedura chiamante) ed una variante di merge sort in cui, invece, si sottrae: se gli elementi sono k, se ne prendono k-1 da una parte ed 1 solo dall'altra (varrebbe la stessa cosa se se ne prendessero 2, o 10, o un qualunque numero indipendente da k).

```
procedura ms_sbil (     aa : array [1 .. n] of integer,  inf, sup : integer   )

{  if   inf < sup                          quindi nell'array  ci sono almeno 2 elementi
   then  {  ms(aa, inf, inf)               prendo  solo il primo elemento
            ms(aa, inf+1, sup)             prendo tutti gli altri
            merge (aa, inf, inf+1, sup)    è  la procedura  merge  di mergesort, che
funziona                                          anche in questo caso
         }
}
```

Per questa nuova procedura la complessità è (con a, b, a', b' c costanti intere > 0):

T(1) = c

T(n) = T(n-1) + T(1) + M(n) = T(n-1) + a n + b

dove M(n) = complessità del merge, che è lineare in n, cioè della forma a'n + b'

Risolvendo per sostituzione si ottiene:

$$T(n) = (an + b) + (a(n-1) + b) + ... + (a (n-j) + b) + ... + (a (2) + b) + c$$
$$= a (n + (n-1) + ... + 2) + (n-1) b + c$$
$$\in \Theta(n^2)$$

Quindi, otteniamo una complessità proporzionale a n^2, mentre quella del merge sort è proporzionale a n log n. Per confronto, riscriviamo il sistema per T(...) di merge sort:

T(1) = c

T(n) = 2T(n/2) + M(n) = 2T(n/2) + a n + b

Tutto questo si vede anche bene considerando gli alberi di chiamata per merge sort e per le procedura modificata e facendo i conti lì sopra.

12.4 . Tecnica della programmazione dinamica (dynamic programming)

Abbiamo già visto un esempio parlando di numeri di Fibonacci.

Questa tecnica può essere utile quando, dopo aver impostato la soluzione del problema in modo induttivo, o secondo un approccio divide et impera, o comunque in modo top-down (cioè scomponendo il problema in sottoproblemi.....) si vede che alcuni sottoproblemi vengono riesaminati più volte, troppe volte. Quando questo succede, la complessità può divenire esponenziale; quindi bisogna trovare un approccio alternativo.

Si prova allora un approccio bottom-up: partendo dai problemi elementari (i più piccoli), si risolvono successivamente, ed usando le soluzioni già ottenute, problemi “più grossi” fino ad arrivare al problema che era l’obiettivo.

Per evitare di ricalcolare: si memorizzano le soluzioni dei problemi già risolti (tutte o solo quelle che servono). Grosso modo si costruisce una tabella di risultati intermedi.

Da questo punto di vista, l’esempio dei numeri di Fibonacci è semplice, perchè basta ricordare gli ultimi due risultati calcolati. Per il calcolo dei coefficienti binomiali, che vedremo, basta tenere l’ultima riga della tabella. Per problemi più complessi, serve la tabella completa.

12.4.1 . Coefficienti binomiali

Riprendiamo per prima cosa le definizioni.

Per ragioni “tipografiche” usiamo la notazione [n : k] invece di quella più familiare “n su k”.

[n : k] si definisce per n e k (interi) t.c. 0 = k = n.

Una definizione utilizza il fattoriale: [n : k] = n! / (k! ∗ (n-k)!).

Abbiamo già visto che ci sono dei problemi nell’utilizzare questa definizione per scrivere un algoritmo che calcola [n : k]; infatti, anche quando il valore di [n : k] è abbastanza piccolo, i termini fattoriali possono avere un valore molto grande, che eccede le dimensioni massime per gli interi. L’ostacolo si può aggirare usando i “float”, ma si perde in precisione, mentre il valore di un coefficiente binomiale è sempre intero (come abbiamo visto parlando di induzione).

Quindi sarebbe necessario scrivere un algoritmo che cerca di semplificare i termini al numeratore e al denominatore, prima di eseguire moltiplicazioni e divisioni.

Non approfondiamo questo punto e passiamo all’altra definizione per i coefficienti binomiali.

(i) base: [n : 0] = [n : n] = 1

(ii) passo (con 0 < k < n): [n : k] = [n-1 : k-1] + [n-1 : k]

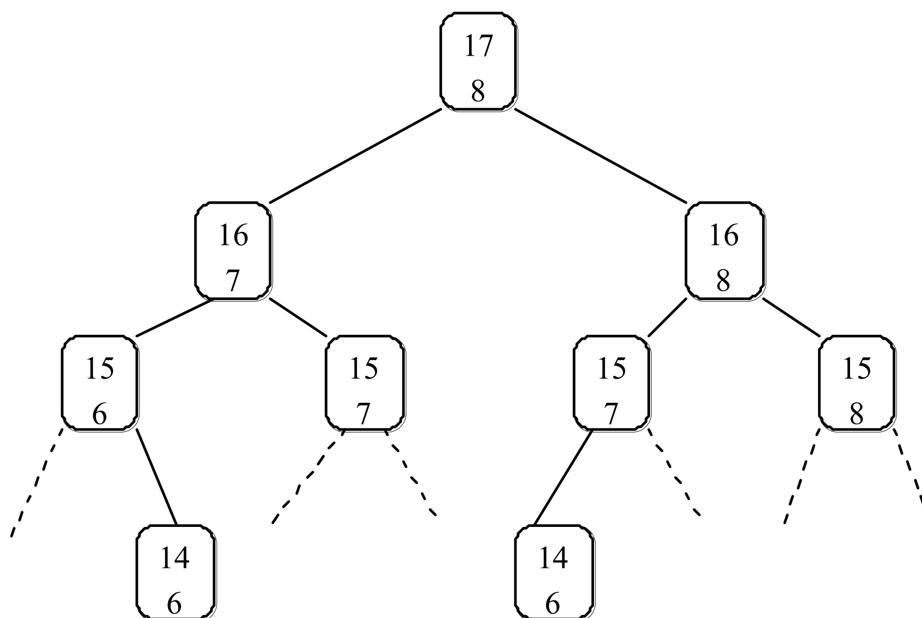
Questa definizione si traduce immediatamente nel seguente algoritmo ricorsivo (già visto):

```
function cobin(n, k : integer) : integer
        var   temp1, temp2 : integer

        if (0 = k) and (k = n)     then
                      if  (k=0)  or  (k=n)   then  return(1)
                      else   {   temp1 ← cobin(n-1, k-1)
                                 temp2 ← cobin(n-1, k)
                                 return (temp1 + temp2)     }
        else  return (-1)        come messaggio di errore
```

Qui si verifica un fenomeno analogo a quello già discusso per i numeri di Fibonacci: molti coefficienti vengono calcolati più volte (anzi, troppe volte; ad esempio, per calcolare [17 : 8] il coefficiente [2 : 1] viene calcolato 6435 volte!).

Questo (cioè che i coefficienti si calcolano più volte, ma non quante volte) si intuisce schizzando l'albero delle chiamate ricorsive per un qualche [n : k] che non si riduca immediatamente ai casi di base; scegliamo ad esempio [17 : 8]:



Non solo [15 : 7] viene calcolato due volte, ma vengono calcolati due volte tutti i coefficienti che compaiono nei due sottoalberi (uguali) con radice in [15 : 7]; [14 ; 6] è ancora peggio, e così via.

Basta questo per capire che le cose vanno male.

> **Per esercizio:** scrivere un programma che, presi n e k, stampa in output per ogni coefficiente binomiale "più piccolo di [n : k]" quante volte quel coefficiente viene calcolato se si usa la procedura di sopra per calcolare [n : k].

Invece non è facile capire la complessità dell'algoritmo ricorsivo. Infatti, in genere, i rami dall'albero delle chiamate sono di lunghezza diseguale.

Se calcoliamo, ad esempio, [n : n-2] i rami di destra sono corti; in particolare, quello più a destra è lungo 2. Se invece calcoliamo [n : 2] sono i rami di sinistra ad essere corti.

Si ottiene un albero abbastanza bilanciato con coefficienti [n : k] dove $k \approx n/2$. In questo caso i rami sono tutti di lunghezza almeno n/2 (circa); potando tutti i rami in modo che abbiano la stessa lunghezza otteniamo un albero binario bilanciato di altezza n/2 (circa) che ha $\Omega(2^{n/2})$ nodi. D'altra parte, i rami più lunghi hanno lunghezza al più n (circa) e quindi il numero totale dei nodi è in $O(2^n)$.

Con questo ragionamento si capisce che per certi valori di k il costo è in $\Omega(2^{n/2}) \cap O(2^n)$.

Comunque, come detto prima, basta guardare il disegno per capire che le cose vanno male.

Invece non dovrebbero:

per calcolare [n : k] basta conoscere il valore dei coefficienti "più piccoli"

se anche li prendiamo tutti, sono meno di (n+1)(k+1), quindi meno di $(n+1)^2$

(infatti: (n+1)(k+1) è il numero delle coppie del prodotto cartesiano $\{0, 1, 2, ..., n\} \times \{0,1, ..., k\}$ e non tutte ci danno un coefficiente valido).

Conviene allora tentare un approccio bottom-up; questo porta ad una soluzione ben nota: il triangolo di Pascal-Tartaglia.

Per arrivarci, prima disponiamo "a triangolo" i coefficienti binomiali: vedere Figura T1.

La prima riga è formata da tutti i coefficienti con "0 sopra", la seconda da tutti quelli con "1 sopra",...., la j-ma riga contiene tutti i coefficienti con "j-1 sopra"; all'interno di una riga, i coefficienti sono ordinati in base a "quello che sta sotto".

In ogni riga: il primo e l'ultimo coefficiente hanno valore noto = 1 (base della definizione induttiva); gli altri si ottengono sommando il valore dei due coefficienti che "stanno sopra".

Così si arriva la triangolo di Pascal-Tartaglia vero e proprio, che contiene solo i valori (i risultati); vedere Figura T2.

Figura T1

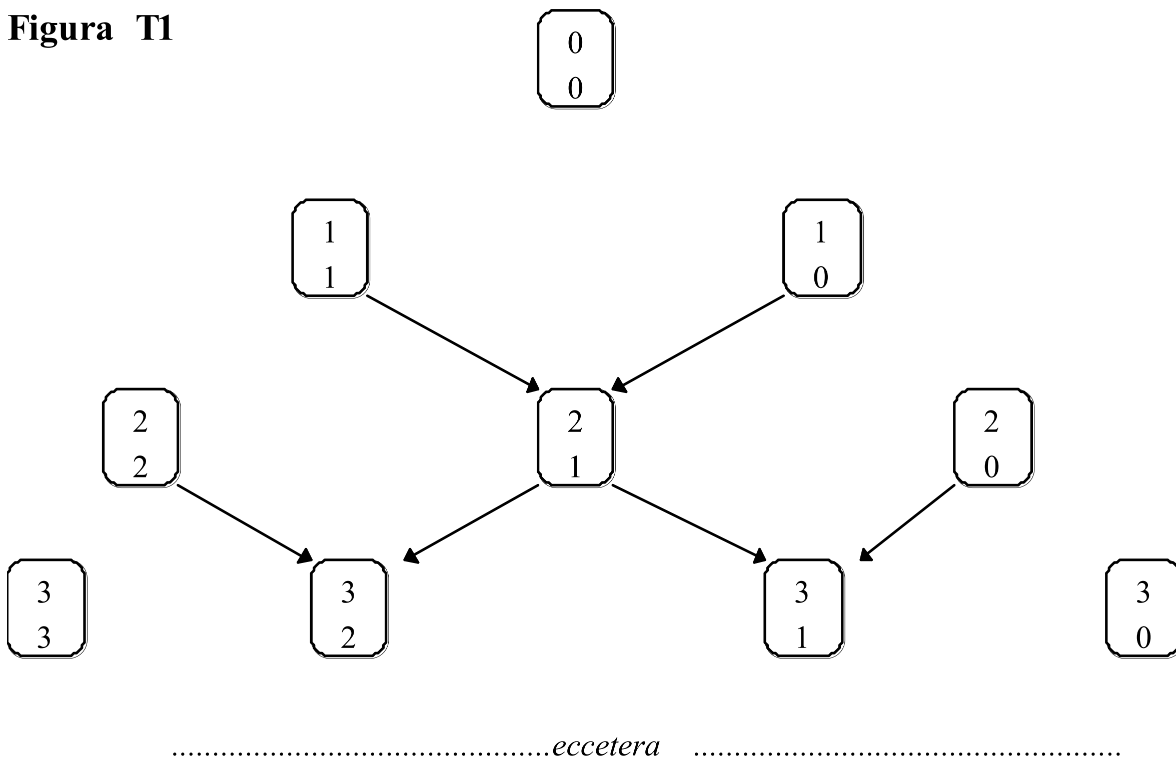
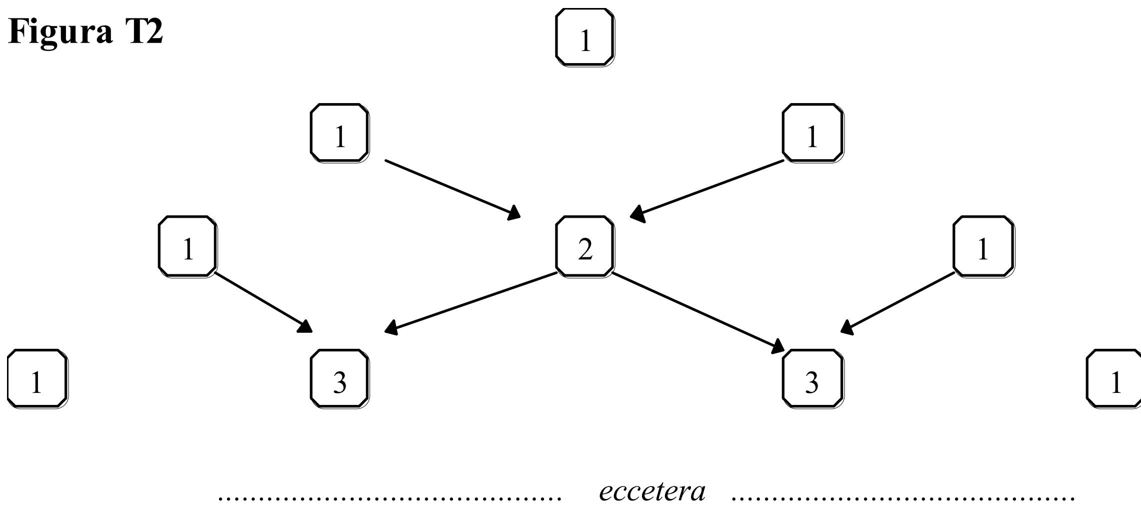


Figura T2



.... *eccetera* ....

In pratica, il triangolo si usa così: volendo, ad esempio, calcolare il coefficiente [17 : 8], si costruiscono una dopo l’altra le righe 1, 2, 3, ... fino alla riga 17 compresa (che contiene i valori dei coefficienti della forma [16 : k]), poi si trovano i valori che corrispondono ai due coefficienti [16 : 7] e [16 : 8] e si sommano.

Il triangolo è particolarmente vantaggioso volendo <u>calcolare tutti i coefficienti</u> della forma [n : k], per un dato n. In effetti, tanto vale calcolare sempre tutti i coefficienti, per un dato n, e nel seguito supponiamo di fare questo.

L’algoritmo basato su questa idea, usa solo due array (di lunghezza n): uno memorizza la riga precedente (già costruita), nell’altro si inseriscono i valori per la nuova riga; poi si scambiano i ruoli dei due array.

Per esercizio: scrivere l’algoritmo basato sul triangolo di Pascal-Tartaglia, che, dato n, calcola tutti i coefficienti della forma [n : k], 0 = k = n.

Il costo di calcolare, in questo modo, tutti i coefficienti della forma [n : k], 0 = k = n, dipende solo dal numero di elementi nelle prime n+1 righe del triangolo (cioè dal numero di coefficienti [p : h], con p = n), se non si fanno pasticci nell’implementare l’idea.
Poichè la riga i-ma contiene proprio i elementi, abbiamo una somma dalla forma familiare:

$$1 + 2 + 3 + ... + (n+1) \qquad \text{che è uguale a} \qquad (n+2)(n+1)/2.$$

Il costo è quindi in $\Theta(n^2)$. Notare che si tratta, comunque, di un costo esponenziale rispetto alla lunghezza dell’input (l’input è n).

12.4.2

. Il problema dello zaino (knapsack problem)

Ci sono tante varianti di questo problema. Una è la seguente:

> dato un contenitore (zaino) di "capacità" K ed n oggetti O_1, ..., O_n, ciascuno con una "dimensione" ed un "valore", si tratta di scegliere un sottinsieme degli oggetti, da mettere nel contenitore, in modo da massimizzare il "valore trasportato".

Un'altra variante non considera i valori e chiede di riempire il più possibile il contenitore.

Qui discutiamo una variante più semplice (detta ***decisionale***):

> dato un contenitore di "capacità" K ed n oggetti O_1, ..., O_n, ciascuno con una "dimensione", vogliamo solo decidere se è possibile scegliere un sottinsieme degli oggetti, da mettere nel contenitore, in modo da "riempirlo esattamente".

La prima cosa da fare è precisare il problema, precisando cosa sono la dimensione e la capacità e cosa vuol dire "riempire esattamente".

Per semplicità, scegliamo che capacità e dimensione siano numeri interi; allora, se l'oggetto O_j ha dimensione k_j :

- "scegliere un sottinsieme di oggetti" equivale a: scegliere un sottinsieme J degli indici, cioè dell'insieme {1,2, ..., n}
- "riempire esattamente" equivale a: $K = \Sigma_{j \in J} k_j$
- infine, ci si rende conto che per risolvere il problema gli oggetti non interessano, ma interessa solo la loro dimensione.

A questo punto, si può formulare il problema in maniera matematica.

Problema dello zaino, versione decisionale.

Input: $K , k_1 , ... , k_n$ interi positivi o nulli

Output: vero se esiste J sottinsieme di {1, ..., n} tale che $K = \Sigma_{j \in J} k_j$
falso altrimenti

Si può arrivare ad una soluzione di questo problema ragionando in modo induttivo:

fissati $K , k_1 , ... , k_n$ proviamo a risolvere il problema usando **solo i primi j oggetti**.

Per comodità, introduciamo un po' di notazione:

indichiamo con P(j) il problema con input $K, k_1 , ... , k_j$ (j = n).

Soluzione induttiva - Base.

P(1) è facile: la risposta è "vero" se $K = k_1$ (allora J = {1})
oppure K = 0 (allora J = ∅)
"falso" altrimenti

Soluzione induttiva - Passo.

Qui scriviamo: " P(j) = vero " per "l'output per P(j) è vero", analogamente per falso.

Dobbiamo risolvere P(j). Osserviamo che

- se P(j-1) = vero allora anche P(j) = vero
  (il problema si risolve scegliendo J tra i primi j-1 elementi, l'elemento j-mo non serve);
- se P(j-1) = falso allora la soluzione, se esiste, deve usare l'elemento j-mo quindi:
  P(j) = vero sse P(j-1) = vero, **togliendo k_j alla capacità K**

infatti: se [P(j-1) = vero, togliendo k_j alla capacità K] scegliendo un certo J,
allora P(j) = vero, scegliendo $J \cup \{j\}$.

A questo punto si capisce che non possiamo considerare solo i sottoproblemi che si ottengono tenendo fissa la capacità K e prendendo i primi j oggetti; dobbiamo anche far variare la capacità.

Inoltre, quando sottraiamo k_j dalla capacità dobbiamo verificare che il risultato sia ancora maggiore o uguale a zero: se lo è si continua, altrimenti non c'è speranza. Di qui si capisce che ha senso anche considerare una capacità uguale a zero; a questo punto si può anche far partire j da zero.

Quindi si arriva alla soluzione induttiva vera e propria.

Soluzione induttiva

Fissati K , k_1 , ... , k_n per ogni j ed H tali che 0 = j = n e 0 = H = K :
indichiamo con P(j, H) il problema con input H, k_1 , ... , k_j

In altri termini, P(j, H) è il problema dove consideriamo una capacità H (= K) e ci limitiamo a considerare i primi j oggetti (j = n); notare che P(n, K) è il problema originario.

Base.

I problemi P(0, H) sono facili: risposta "vero" sse H = 0 (e allora J = ∅).

Passo.

Scriviamo: " P(j, H) = vero " per "l'output per P(j, H) è vero", analogamente per falso.

Per risolvere P(j, H):

- se P(j-1, H) = vero allora anche P(j, H) = vero
- se P(j-1, H) = falso allora la soluzione, se esiste, deve usare l'elemento j-mo quindi:

  P(j, H) = vero sse P(j-1, H-k_j) = vero

  precisazione: prima di considerare P(j-1, H-k_j), bisogna controllare H-k_j : se è = 0 allora si continua; altrimenti l'elemento j-mo non può essere usato e quindi P(j, H) = falso.

Precisazione sullo schema induttivo

Lo schema induttivo che si è seguito è:

- induzione aritmetica a passi su j (quindi sul numero di elementi che consideriamo)
- obiettivo (se si vuole: predicato da dimostrare): sappiamo risolvere P(j, H) per ogni H : 0=H=K
- base (j = 0): sappiamo risolvere (direttamente) P(0, H) per ogni H : 0 = H = K
- passo (j > 0): supponendo di saper risolvere P(j-1, H) per ogni H : 0 = H = K
  risolviamo P(j, H) per ogni H : 0 = H = K

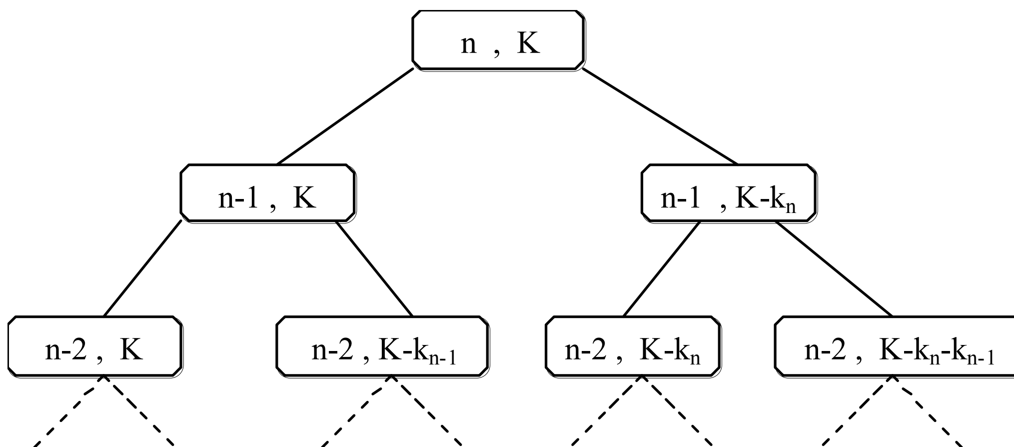
Precisato lo schema, non ci vuole molto a scrivere un algoritmo ricorsivo che risolve il problema dello zaino:

```
function zaino ( j: integer, H: integer, ak: array[1..n] of integer )
```

il parametro ak memorizza $k_1, ..., k_n$

corpo della funzione: per esercizio

Per capire la complessità di una chiamata zaino(n, K, ak), schematizziamo l'albero delle chiamate, indicando solo i primi due argomenti (il terzo, l'array, non cambia nel corso delle chiamate):



Il primo argomento diminuisce di 1 ad ogni livello e ci si ferma quando è arrivato a 0; però ci si ferma anche quando il secondo argomento diventa negativo. Anche qui è possibile che l'albero abbia rami di lunghezza diseguale: lunghi (circa) n verso sinistra e più corti verso destra. Se i k_j sono piccoli e K grande, anche quelli di destra saranno abbastanza lunghi.

Nel caso peggiore si fanno tutte le chiamate; infatti il caso peggiore corrisponde alla risposta "falso", che si ottiene solo dopo aver esplorato tutte le possibilità.

Se consideriamo n (cioè il numero di oggetti) come unico parametro, allora possiamo scegliere K ed i k_i in modo da ottenere un caso pessimo. Ad esempio, se $k_1 = k_2 = ... = k_n = 1$ e $K = n+1$ non c'è soluzione e l'albero delle chiamate è un albero binario completo di altezza n. Quindi ci sono $2^{n+1}-1$ chiamate e la complessità è in $\Omega(2^n)$

Però ci sono, al più, (n+1)(K+1) problemi da risolvere e, per K non enorme rispetto ad n, 2^n è molto maggiore di (n+1)(K+1). Quindi siamo nel caso tipico in cui si pensa ad una soluzione nello stile della programmazione dinamica.

Algoritmo nello stile della programmazione dinamica

Input: K, n, $k_1, ..., k_n$ come sopra.

Usa un array ak di n elementi per memorizzare i k_j

una matrice, sia MP, di dimensioni n+1 per K+1. Alla fine dell'algoritmo vogliamo:

MP [j, H] = vero/falso se P(j, H) = vero/falso

Il risultato (l'output) si troverà quindi in MP[n, K].

La matrice viene riempita per righe e partendo "dal basso", cioè dalla riga j = 0, seguendo quanto visto sopra:

```
leggi  K ed  n
leggi   k1, ..., kn   e mettili in  ak[1],...., ak[n]
MP[0,0] ← vero
per  H = 1, 2, ..., K :   MP[0, H] ← falso      questo conclude la "base"
per  j = 1, 2, ..., n :
      per  H =  0, 1, ..., K :
            {     if  MP[j-1, H]   then   MP[j, H] ← vero
                  else
                        if   H - ak[j] = 0   then   MP[j, H] ← MP[ j-1, H - ak[j] ]
                        else  MP[j, H] ← falso
            }
```

Si vede subito che, qualunque sia l'input, l'algoritmo fa sempre un numero di passi proporzionale alla dimensione della matrice, quindi la complessità dell'algoritmo è in $\Theta(n K)$.

Se volessimo anche conoscere l'insieme J

L'algoritmo visto produce solo output vero/falso. È facile modificarlo per ottenere (nel caso vero) anche un insieme di indici J t.c. $\Sigma \{ k_j \mid j \in J \} = K$.

Si usa una seconda matrice booleana Q, con dimensione $n \times K$ (gli indici 0 non si usano).

Si vuole, per j > 0 e H > 0:

se MP[j, H] = vero
allora Q[j, H] = vero se la soluzione trovata, al problema P(j, H), usa l'elemento j-mo
Q[j, H] = falso altrimenti (la soluzione trovata non usa l'elem. j-mo)

Nota: se MP[j, H] = falso, non ci interessa il valore di Q[j,H].

Posto di saper calcolare correttamente Q, vediamo che basta per ottenere quello che vogliamo:

se MP[n,K] = vero (cioè il problema originale ha soluzione)
allora se Q[n,K] = vero allora: $J = J' \cup \{n\}$ dove J' è quello per $P(n-1, K - k_n)$
altrimenti J è quello relativo a P(n-1,K)

Quindi, basta "procedere a ritroso".

Per esercizio : scrivere l'algoritmo per ricostruire J.

Per calcolare Q[j,H] basta modificare il corpo del "for su H" come segue:

```
{     if  MP[j-1, H]   then    {  MP[j, H] ← vero  ;   Q[j, H] ← falso   }
      else
         if   H - ak[j] = 0
         then       if   MP[ j-1, H - ak[j] ]  then  {  MP[j, H] ← vero  ;   Q[j, H] ← vero  }
                    else  MP[j, H] ← falso
         else  MP[j, H] ← falso
}
```

Notare infine che si può sostituire la matrice MP con due vettori: uno per la riga "precedente" ed uno per la riga che si sta costruendo (poi alla fine si scambiano i vettori ....), mentre serve mantenere tutta la matrice Q.

13. Libri su algoritmi, strutture dati, complessità, ...

(sono quasi tutti presenti in biblioteca)

Introduttivi:

Bertossi: Strutture Algoritmi Complessità (ECIG - Genova)

Lodi, Pacini: Introduzione alle strutture di dati, Boringhieri

Wirth: Algoritmi + Strutture dati = Programmi (soprattutto per ordinamenti)

Di media difficoltà

Aho, Hopcroft, Ullman: Data structures and algorithms (Addison Wesley, 1983)

Aho, Hopcroft, Ullman: The design and analysis of computer algorithms (Addison Wesley 1974)

Ausiello, Marchetti-Spaccamela, Protasi: Teoria e progetto di algoritmi fondamentali (Franco Angeli, 1988)

Horowitz, Sahni: Fundamentals of data structures in Pascal (Computer Science Press, 1990)
esiste anche versione con il C ed è stata tradotta in italiano (Strutture dati in C McGraw Hill - Italia)

Horowitz, Sahni: Fundamentals of computer algorithms (Computer Science Press, 1978)

Luccio: La struttura degli algoritmi (Boringhieri 1982)

Manber: Introduction to algorithms. A creative approach (Addison Wesley, 1989)

Sedgewick: Algoritmi in C, (Addison-Wesley Masson)

Più avanzati o specializzati :

Cormen, Leiserson, Rivest: Introduction to Algorithms (MIT Press 1990); è stato tradotto ed è edito dalla Jackson, spezzato in 3 volumi.

Melhorn: Data structures and Algorithms - 3 volumi (Springer Verlag 1984)

Knuth: The art of computer programming - 3 volumi (Addison Wesley 1973)

Even: Graph algorithms (Computer Science Press, 1979)

Reingold, Nievergelt, Deo: Combinatorial algorithms. Theory and practice (Prentice Hall, 1977)

Garey, Johnson: Computers and intractability. A guide to the theory of NP-completeness (Freeman, 1979)

Quasi per lettura amena:

Harel: Algorithmics, the spirit of computing (Addison-Wesley, 1987)

Fine della 3ª (ed ultima) puntata