Lezioni del 7-05-14 e 09-05-14.

Ricorsione

Algoritmo: *serie di step con durata finita che permettono di raggiungere un certo obiettivo dato un input. La ricorsione è una tecnica usata per avere algoritmi più efficienti.*

Def. *Una funzione f si dice ricorsiva quando all'interno della sua definizione richiama se stessa, oppure se la funzione f richiama una funzione g che direttamente o indirettamente richiama la funzione f.*

Per cui un algoritmo è ricorsivo se all'interno dell'algoritmo c'è una funzione ricorsiva.

Esempio: se devo risolvere n! usando la ricorsione. Dalla definizione di fattoriale so che: n!=n (n-1)!

```
double fact(double N) {
      double sub;
      if(N==0) return 1.0;    // condizione d'uscita
      sub=fact(N-1);

      return N*sub;
}
```

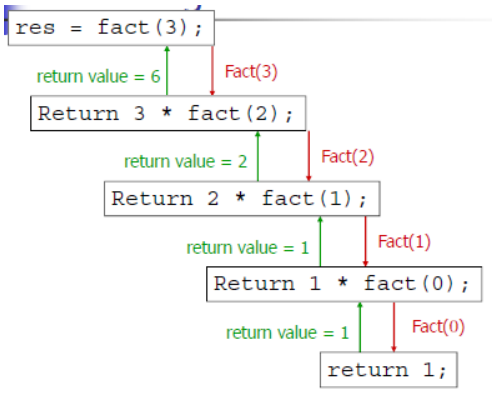
Questo algoritmo così snello va bene per qualunque N, se N=1, scorro la funzione una prima volta e a un certo punto mi auto-richiamo e torno dall'inizio, prima vado sempre avanti senza mai arrivare alla fine della funzione e quando incontro la condizione d'uscita inizio a tornare indietro fino alla funzione "madre" che l'ha richiamata.

Se voglio trovare fact(3) il flow è:

`res=fact(3) -> return 3*fact(2) -> return 2*fact(1) -> return 1*fact(0) -> return 1;`

scendo giù e nel frattempo la cardinalità dell'insieme su cui l'algoritmo lavora scende proporzionalmente man mano che scendo nella catena. A questo punto torno indietro:

`return 1 -> return 1 -> return 2 -> return 6;`



E' una tecnica molto furba, intelligente ed efficiente, serve a risolvere problemi la cui struttura si presta bene alla ricorsione. Se con un approccio top-down mi accorgo che tanti step sono in comune e sembra stupido fare n funzioni separate, faccio un'unica funzione che mi risolva quasi o tutto il grande algoritmo. Ma la ricorsione è qualcosa più sofisticato, si usa quando mi accorgo che lo stesso corpo d'istruzioni se applicato a un problema via via più piccolo risolve il problema iniziale. Nel caso del fattoriale mi accorgo che già la definizione è ricorsiva, c'è il fattoriale sia nel membro di destra che in quello di sinistra. L'unico problema che veramente risolvo è calcolare il fattoriale di 0, è il problema più piccolo, in tutti gli altri step io richiamo solo una funzione.

Gli errori principali sono:

- Sbagliare la condizione finale e quindi non uscire più.

- Mancando un controllo sulla memoria allocata, si rischia di saturarla. Quando chiamo chiamo chiamo ma non esco mai dalle funzioni, aggiungo sempre delle cose allo stack, inizio a deallocare solo quando torno indietro, però non ho il controllo di quanto possa scendere.

Motivazioni per usare la ricorsione: molti problemi in generale si prestano bene ad essere risolti con la ricorsione, perché la soluzione è spessa basata sul principio del "Divide et Impera". Ovvero suddividere un problema in piccoli problemi più semplici e poi combinare le singole piccole soluzioni per ottenere la soluzione del problemone. Anche dal punto di vista della computazione, **più è più piccolo il problema più è facile da risolvere.** Se analizziamo il paradigma del divide et impera:

Divide et impera

- Prendo a sotto-problemi dove ogni sotto-problema è b volte più piccolo di quello originale.
- Risolvo ogni sotto-problema lo risolvo. (subsolution=solve(subproblem)).
- Combino le subsolution per ottenere la soluzione del problema padre. Solution=combine(subsolutions1,2,3…).

Nel caso della ricorsione, **la soluzione dei singoli sotto-problemi viene fatta richiamando il problema principale**.

Quindi se solution=solve(problem), la funzione solve() è la stessa a essere applicata in ogni subproblem.

Poiché la ricorsione non può essere infinita, devo trovare delle condizioni di ritorno. Mi fermo quando la soluzione del sotto-problema è talmente banale e semplice che riesco a risolverla con un costo computazionale minimo oppure quando non c'è più bisogno del metodo ricorsivo.

È obbligatorio avere una terminazione e devo essere sicuro che i sotto-problemi siano man mano più semplici di quello originario. Il sotto-problema figlio deve essere più semplice del sotto-problema padre, qualunque sia il padre.

Esempio: serie di Fibonacci.

Problema: trovare l'ennesimo numero di Fibonacci.

Definizioni $FIB_{n+1}=FIB_n+FIB_{n-1}$ per n>0; $FIB_0=0$ e $FIB_1=1$;

Un algoritmo ricorsivo che posso utilizzare:

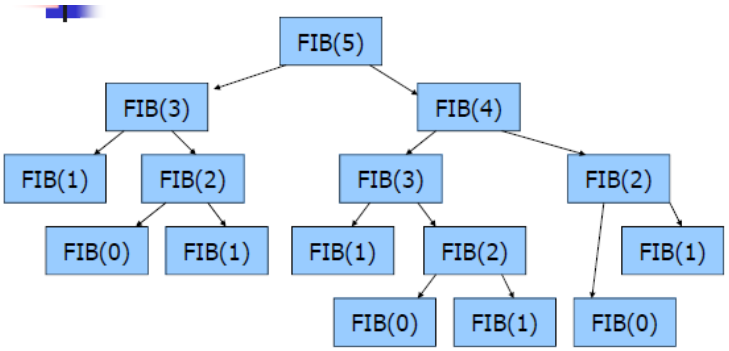
```
long fib(int n) {
long f1,f2;

       if(N==0) return 0;
       if(N==1) return 1;
       f1=fib(N-1);
       f2=fib(N-2);

       return f1+f2;
}
```

Questo algoritmo è molto semplice, non ho bisogno di strumenti complicati, va bene qualunque sia N (non consideriamo eventuali problemi di memoria). Qui risolvo il sotto-problema in due sotto-problemi (e non in uno solo come per il fattoriale), devo calcolare due numeri di Fibonacci per trovare il numero di Fibonacci successivo.

Semplicemente prendiamo la definizione ricorsiva e la riscrivo in termini "informatici", non sto pensando a una soluzione al problema perché ce l'ho già. Se voglio trovare fib(5) questo è l'albero che si viene a creare: (la cardinalità del problema figlio è sempre minore di quella del padre):



Fonte: testgroup.it

Esempio: ricerca binaria

In un array ordinato voglio trovare il valore x . Divido l'array in due parti, una non serve perché è ordinato, e nell'altra riapplico la ricerca.

Qui su un array da 10 elementi riesco con 3-4 iterate a ricercare l'elemento, basta che la lista sia ordinata (ogni volta che devo aggiungere un elemento devo solo spendere un po' più di tempo a capire dove inserirlo). Man mano che scendo la dimensione del problema diminuisce perché ogni volta si dimezza la dimensione del vettore.

```
int search(int v[], int a, int b, int x) {
int c;

	if(b-a==0) { // se ho un solo elemento
		if(v[a]==x) return a;
		else return -1;
	}

	c=(a+b)/2;
	if(v[c]>=x) return search(v,a,c,x);
	else return serach(v, c+1, b, x);
}
```

Anche qui ho all'inizio della funzione dei controlli per capire se sono alla fine, qui non combino la soluzione dei sotto-problemi ma una volta trovata devo solo passarla alle chiamate precedenti.

Funzione iterativa: non c'è la ricorsione, c'è un insieme di istruzioni con for e while, **tutti i problemi ricorsivi possono essere implementati con funzioni iterativi.** Per termini di chiarezza, se il programma deve andare ad altri è importante che sia chiaro, spesso utilizzando la ricorsione ottengo un codice più snello e chiaro. Se invece lo faccio iterativo, non vedo la soluzione ricorsiva subito nell'algoritmo ma faccio dei casini allucinanti che rischio di capire solo io.

Fattoriale con soluzione iterativa:

```
double fact (double N) {

	double tot=1.0;
	int i;
```



```
	for(i=2; i<=N; ++i) {
		tot=tot*I;

	return tot;

	}
```

La soluzione iterativa permette comunque di risolvere il problema e forse per noi è anche più immediate, ma è più complicata da leggere per una persona esterna.

Invece la Fibonacci a livello iterativo è qualcosa del tipo:

```
long fib(int N)
{    long f1p=1, f2p=0, f ;
     int i;
     if(N == 0) return 0 ;
     if(N == 1) return 1 ;

     f = f1p + f2p ; /* N==2 */
     for(i=3; i<= N; ++i)
     {    f2p = f1p ;
          f1p = f ;
          f = f1p+f2p ;
     }
     return f ;
}
```

Fonte: testgroup.it

Problemi suggeriti:

- Scrivere la versione iterativa per il calcolo del coefficiente binomiale.
- Versione iterativa del calcolo di un determinante.

Esempio: the Knight tour

Problema matematico, coinvolge il pezzo del cavallo nella scacchiera, parte da una posizione qualunque e deve visitare tutte la caselle che può solo una volta. Non è così chiaro che si debba usare la ricorsione. Voglio massimizzare il numero di caselle che visita. Supponiamo che la scacchiera sia una 4x4, faccio tutte le mosse fin quando arrivo a un punto in cui facendo la mossa successiva (mossa 13) mi bloccherei. A questo punto torno alla mossa 12 e provo a prendere un'altra strada. E' un divide et impera perché man mano che scendo ho sempre meno caselle in cui posso andare.

Da ogni nodo posso fare al massimo 8 mosse, il numero di step per ricoprire la scacchierà è N^2, l'albero di ricerca ha al massimo 8^{N^2} nodi, nel caso peggiore la foglia è quella più a destra e avrei quindi $O(8^{N^2})$ chiamate ricorsive.

Esempio: X values expansions

X indica un don't care, la funzione OR torna 1 se in ingresso ha 1X o X1. Vogliamo implementare un programma che riceva una stringa binaria con 1-0-X e torni tutte le possibili combinazioni con solo 0 e 1. Anche qui posso esplorare tutto l'albero dove in numero di foglie (ovvero di combinazioni) è 2^N con N=numero di X e la profondità dell'albero è al massimo N+1.

Esempio: puzzle delle 8 regine

Mettere 8 regine in una scacchiera 8X8 in modo che nessuna di queste otto possa mangiare nessun altra. #todo da vedere a casa.

Esempio:domino

#todo da vedere da casa.

Analisi della complessità computazionale

Chiunque può scrivere algoritmi, il problema è scrivere algoritmi efficienti, in termini di memoria e di tempo impiegato. Devo riuscire a fare delle valutazioni assolute, che non dipendano dall'hardware usato per farlo girare.

Algoritmo: procedimento di calcolo. Una sequenza finita d'istruzioni ciascuna delle quali ha un significato ben preciso che può essere risolta in un tempo ben preciso. In genere non c'è mai una sola soluzione a un dato problema, è importante fare l'analisi degli algoritmi per comparare due o più algoritmi che risolvano lo stesso problema. Per esempio capire come si comportano i diversi algoritmi nel così detto worst case? Ordinare un vettore da 5 elementi è semplice per tutti gli algoritmi ma se deve ordinare 2000000 elementi? È lì che si fanno i confronti. **In genere si sceglie l'algoritmo migliore nel caso peggiore**. Spesso appena supero una determinata barriera l'algoritmo diventa inutilizzabile. Inoltre facendo l'analisi di un algoritmo riesco a capire dove posso migliorare l'efficienza, capire quali blocchi "pesano" di più e quindi come migliorarlo (anche qui scelto un algoritmo non cerco di minimizzare tutto l'algoritmo ma solo il suo sotto-blocco più pesante). Devo riuscire a capire quale pezzo del mio algoritmo pesa di più.

La bontà di un algoritmo si valuta o in modo:

- Soggettivo.
    - Semplicità.
    - Chiarezza.
    - Idoneità alla risoluzione di un determinato problema (non usare un Caterpillar per far fuori una mosca).
- Oggettivi:
    - Analisi computazionale.

Analisi computazionale

Def. Efficiency *(efficienza): abilità di risolvere il problema utilizzando le più basse risorse computazionali possibili.*

I due fattori fondamentali caratterizzanti l'efficienza sono i :

- ***Costi spaziali*****:** memoria richiesta (non lo guardiamo perché tanto si fa sommando nel worst case le dimensioni di tutte le allocazioni statiche/dinamiche).
- ***Costo temporale***.

Un algoritmo A è migliore di un algoritmo B se a parità di hardware l'algoritmo A ha bisogno di meno memoria/tempo rispetto a B.

La complessità computazionale dal punto di vista temporale viene valutata considerando:

- Grandezza del problema (n): numero di "simboli" utilizzati per codificare il problema, in generale potrà essere il numero di byte, bit, record, caratteri, celle di un vettore etc. (dimensione dell'input per farla breve). Nel paradigma del divide et impera la grandezza del problema scende passando dal problema al sotto-problema.
- Tempo di esecuzione (T): *numero di operazioni elementari necessarie per risolvere il problema*. **Il tempo di esecuzione è strettamente legato alla grandezza del problema.** (una cosa è calcolare 1! Un'altra 10000!).

Quindi il tempo di esecuzione viene espresso solo come funzione della grandezza dei dati in ingresso, è solo un T=f(n) e non dipende dai dati in sé sul quale l'algoritmo opera ma solo sulla loro dimensione (intesa come cardinalità). Così riusciamo a svincolarci dall'hardware sul quale stiamo testando l'algoritmo. Ogni operazione ha un determinato costo, c'interessa capire come esplode il numero di operazioni totali da fare per far girare l'algoritmo all'aumentare di n (inoltre così svincoliamo dalla bontà del compilatore, non interessa come il compilatore stravolge il codice. Il compilatore garantisce solamente che i passi logici saranno gli stessi, il problema è capire come lo fa; potrebbe rendere l'algoritmo meno efficiente di quello che è veramente).

Warning: Non sempre la memoria che andiamo a consumare e il tempo d'esecuzione sono gli unici fattori da considerare per valutare la bontà di un algoritmo.

Esempio:
Se t_* è il tempo per fare un prodotto, t_+ il tempo per fare una somma e t_s il tempo per fare un incremento unitario (in genere le ALU implementano questo tipo di operazioni in modo più efficiente, non costano quanto una somma), allora:

```c
int main(){ /*A1*/
  int m;
  m = 10 * 10;
  printf("%d\n", m);
}
```

```c
int main(){ /*A2*/
  int i,m; m=0;
  for(i=1; i<=10; i++)
    m = m + 10;
  printf("%d\n", m);
}
```

```c
int main(){ /*A3*/
  int i,j,m; m=0;
  for(i=1; i<=10; i++)
    for(j=1; j<=10; j++)
      m++;
  printf("%d\n", m);
}
```

Fonte: testgroup.it

T1=t_* (ho una sola moltiplicazione);
T2=$10t_+$(10 operazioni di somma);
T3=$100t_s$ (due cicli annidati);

Quale è il migliore? Dovrei sapere quanto costano t_*, t_+ e t_s in termini di millisecondi. Se li faccio girare su 4 computer con caratteristiche diverse dove le operazioni hanno un tempo diverso, mi accorgo che non c'è un algoritmo migliore, a seconda della macchina ho uno o più algoritmi più efficiente. Il problema è che non posso fare un'analisi in base al tempo (anche perché fra 5 anni magari ho delle macchine molto più potenti sulle quali posso farlo girare). Non è una valutazione equa, una buona caratterizzazione dei costi dovrebbe

essere indipendente dal computer e dalla dimensione dell'istanza del processo (ovvero mi devo svincolare anche da n, adesso magari n=2Meg è tanto ma magari fra 5 anni è piccolo).

Noi non vogliamo scrivere algoritmi una tantum, devono funzionare qualunque sia la dimensione dell'input (non devo risolvere 10^2 ma n^2), altrimenti lo risolvo in modo specifico una tantum. Quello che va fatto è porsi nel caso generale, nel caso di n^2 n è la grandezza del problema. **La dimensione di un problema è data in genere dalla dimensione dell'input.** Quindi ci deve essere un legame fra la dimensione del problema e il costo computazionale, e in genere si fa il limite per capire quello che succede quando n diventa molto grande. Se consideriamo un n molto grande possiamo inoltre applicare delle piccole approssimazioni che non ci spostano dal comportamento asintotico, l'errore relativo che compio diminuisce all'aumentare di n. Per cui i programmi **non devono dipendere dall'input della specifica istanza** (=caso particolare) né **dal tempo di esecuzione delle varie istruzioni**.

Comportamento asintotico

Il comportamento asintotico è il costo di esecuzione dell'algoritmo al crescere della grandezza del problema, ovvero:

$$\lim_{n\to\infty} T(n)$$

In tal modo si trascurano le costanti e i termini di ordine inferiore. Quando analizzo la complessità di un algoritmo *n* rimane una lettera. Non si ragiona sulla specifica istanza ma si vede quello che succede quando *n*->∞. Quindi l'analisi del comportamento asintotico prescinde dal compilatore e dalla macchina usata, non specifico il range di valori (per n fra 10 e 20 l'algoritmo A è migliore dell'algoritmo B mentre se n>50 allora vince l'algoritmo B). Inoltre spesso quando lo scrivo non so a priori qual è il massimo valore che il mio algoritmo può supportare. Tuttavia solo l'analisi del comportamento asintotico non è l'unica da farsi per valutare correttamente un algoritmo, bisognerebbe considerare anche ma la memoria, il numero di accessi di input/output (scrittura/lettura su file, a video o di accesso a internet). Inoltre se so già che n al massimo non va a 1Meg ma solo fino a 10, allora l'analisi asintotica non serve a molto ma devo fare analisi più accurata.

*Def. **step**: è l'esecuzione di un segmento di codice (blocco di istruzioni) il cui tempo di processing non dipende dalla dimensione dell'input (ovvero da n).*

Tornando al caso del calcolo di 10^2 e generalizzando a n, nel primo caso ho sempre una sola istruzione, nel secondo caso ho n operazioni e nel terzo caso ho sempre n^2 operazioni, indipendentemente dalla complessità della singola operazione. Quando si effettua l'analisi asintotica, tutte queste operazioni hanno lo stesso costo (ovvero non c'è distinzione fra t_*, t_+ e t_s), così facendo ci svincoliamo dal tempo di esecuzione dalla singola istruzione e dalla specifica istanza.
Inoltre in questo caso specifico il primo ha un comportamento asintotico costante (è il top del top) perché non dipende neanche da n.

Notazione O, Ω e Θ

- Un algoritmo ha una complessità O(f(n)) se esiste una funzione che lo delimita superiormente, ovvero se qualunque n, T(n) cresce al più come f(n), f(n) è una specie di worst case.

$$T(n) = O(f(n)) \leftrightarrow \exists c > 0 : \lim_{n\to\infty} \left|\frac{T(n)}{f(n)}\right| = c$$

- Ω(f(n)) invece rappresenta il limite inferiore, per quanto decresce n, io non scenderò mai al di sotto di una certa f(n).

$$T(n) = \Omega(f(n)) \leftrightarrow \exists c, n_0 : |T(n)| \geq c|f(n)| \; \forall \; n > \; n_0$$

- T(n)=Θ(f(n)) è limitata da c1 e c2 indipendentemente da come cresce.

Esempi: bubble sort

Si parte dal corpo di codice più interno perché in genere queste istruzioni non dipendono da n, il corpo interno costa sempre c1, se inglobo anche il primo for la complessità cresca a c2(n-i), se poi inglobo anche il for più esterno dove ho una sommaria del for interno, allora dopo "ovvi calcoli" trovo che la complessità è un $O(n^2)$.

STG Design Examples (part 3)

Esercizio-waveform generator

***Hint**: ricordarsi di disegnare la top level view.*

Il fatto di non avere ingressi vuol dire che da ciasun stato escono 2^0=1 archi; non so ancora quanti stati avrò ma di sicuro è una specie di catena, non ho diramazioni. Inoltre nei circuiti sequenziali con le macchine di Moore le uscite cambiano solo a uno specificato edge del clock, non importa se positivo o negativo, non posso far cambiare le uscite un po' su un fronte e un po' sull'altro. Quindi mi servono 6 colpi di clock e poi ho due alternative. Il periodo è 6 volte minore del periodo di clock, solo con 3 clock non sarei riuscita a farlo, questa è la massima frequenza possibile con il clock. Il diagramma a stato avrà sei stati e per ogni stato a ogni uscita assegno un determinato valore. Inoltre gli stati li codifico partendo da 000 e andando avanti (anche perché nello stato 000 deve arrivare il reset, in questo stato l'uscita è 101).

Esercizio 2_2-013: Server

***Hint**: cerchiamo di usare i don't care.*

Usano il protocollo a 4 fasi. Il richiedente alza REQ, e il server risponde dicendo "ho capito" alzando l'ACK, quando il richiedente non ne ha più bisogno abbassa REQ e il server risponde dicendo "ho capito" abbassando il ACK. Anche il server e la resource comunicano con REQ e ack (in questo caso l'ack viene mandato dalla resource al server). 2 client e una risorsa.
noi ci concentriamo sul controllo, non sui dati che transitano. Il server riceve una richiesta da client 1, ma posso mandare l'ack a 1 solo se il server riceve a sua volta l'ack dalla risorsa, solo a quel punto posso dare l'ack al client. Non c'è il problema di memorizzare eventuali richieste da parte di uno dei due client mentre la risorsa è già occupata perché il client che fa la richiesta terrà alto il REQ finché il server non gli risponde con un l'acknowledge.

Slinding window

Una FSM con 1 solo ingresso e una sola uscita è caratterizzata da comportamento a finestra scorrevole se e solo se i valori della sua uscita dipendono solamente dagli ultimi N valori del suo ingresso. Per risolvere questi problemi posso usare o approccio a forza bruta e quindi avere 2^N stati, vado di STT e se poi copro la mappa di Karnaugh scopro che Y(0)=y(1) e.. -> ciò che ottengo è uno shift register associato a una rete combinatoria indipendentemente da quello che fa il mio circuito.

Riprendendo il caso del pulse generator, partiamo dal grafo e andando avanti ottengo proprio uno shift register e una porta and. Avrei un fronte se gli ultimi due bit valgono 01.

Esercizio:Palindrome string detector.

E-voting detector

N persone possono votare e abbiamo 3 candidati. Visto che deve funzionare per N votanti, meglio non usare il diagramma a stati o si rischia un suicidio di massa. Devo avere un blocco che rilevi un fronte positivo e in quel caso generi un impulso alto per un colpo di clock, solo così posso mandarlo all'enable (il contatore incrementa il suo contenuto a ogni colpo di clock, non posso tenere alto quel segnale per sempre altrimenti continuerei ad incrementare).

16-bits up counter without TC

Abbiamo a disposizione un contatore da 4 bit, dall'esterno devo avere contatore da 16 bits con la stessa piedinatura. Me ne servono 4. Il clock va collegato in parallelo a tutti quanti perché deve essere perfettamente sincrono. Anche LD_CNT_n va in parallelo a tutti. Ok la porta and per capire quando un contatore è saturo, inoltre devo incrementare il contatore successivo se non solo quello precedente ma tutti quelli precedenti erano tutti uni, attenzione all'induzione, non basta vederlo in un caso ma bisogna controllare che funzioni anche negli altri casi. Inoltre il secondo e gli altri li devo attivare se tutti quelli precedenti sono saturi E se il circuito è abilitato. Rispetto al contatore standard, questo non ha il terminal counter, ma la logica con l'and e l'enable implementa esattamente il terminal count. Se adesso come blocchetto elementare un contatore con il terminal count posso collegarli in serie.