MANUALE DEL BUON ASDINO

A cura di: Panzera Valerio

INDICE

Per la ricerca rapida

CTRL + F

caso base

Discorso finale

Introduzione

Questo piccolo manuale si pone l'obbiettivo di far comprendere al lettore quali siano i meccanismi che devono essere applicati negli esercizi ricorsivi-iterativi. Gli esercizi proposti sono stati portati a ricevimento, quindi questo dovrebbe darvi la garanzia (oltre al fatto che ho superato lo scritto) che ciò che leggete sia affidabile, ovviamente non al 100%, poiché sono ancora uno studente della triennale, ma non ancora per molto.

Il motivo che mi porta a scrivere questo breve manuale di facile lettura, è solo uno: stufo di vedere colleghi bocciati o che gli venga rallentata la carriera universitaria solamente perché il professore non si degna di svelarci a lezione tutti i concetti necessari per la traduzione. Infatti tutti quelli appresi da me e da altri tre ragazzi che come me hanno superato l'esame, derivano da circa tre mesi di ricevimento. Non c'è altro modo per apprenderli. Alla fine della lettura noterete che alla fine non era nulla di difficile, ma bastava solamente aver presente tutte le strategie che si celano dietro gli esercizi.

Nel manuale adotto un linguaggio mio per fini didattici, ma che non alcuna valenza al di fuori del manuale, quindi non utlizzatelo!!

Ricorsivo-iterativo

Le regole:

Una volta conosciute le “regole” che si devono applicare per la costruzione di un algoritmo iterativo, non sarà molto difficile fare un eserczio di traduzione.

Non a caso utilizzo la parola “regola”, in quanto sono meccanismi che **DEVONO** essere eseguiti per giungere ad una corretta conclusione dell’esercizio. Una volta imparate le regole, non avrete problemi nello svolgere qualsivoglia esercizio ricorsivo-iterativo; certo, non è mica così meccanico lo svolgimento, ma sicuramente non si possono violare le regole.

Regola n°1:
lo scheletro

Per ogni esercizio svolto che andremo a vedere, noterete che la struttura di base, chiamiamolo **scheletro**, è sempre la stessa. Un **while** che cicla e simula il ritorno dalla chiamate, un **if-else** che distingue le nuove chiamate dalle vecchie, e così via, come in figura qui sotto.

```
1  ALGO_IT(parametri){
2      /*INIZIALIZZAZIONE*/
3
4      while(){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8
9          }else{ //vecchia chiamata
10             /*RECUPERO CONTESTO*/
11         }
12     }//fine while
13
14     return
15 }//fine algoritmo
```

Oh neofiti della materia, non spaventatevi di fronte a tale struttura, essa è facile ed adesso andremo a vedere il perché .

Come detto precedentemente,lo scheletro non può non esserci! Infatti, a tal proposito, vi consiglio, prima d'iniziare ogni esercizio ricorsivo-iterativo, d'impostarlo e poi di riempire i corpi.

While: è esso a simulare le chiamate dell'algoritmo ricorsivo;

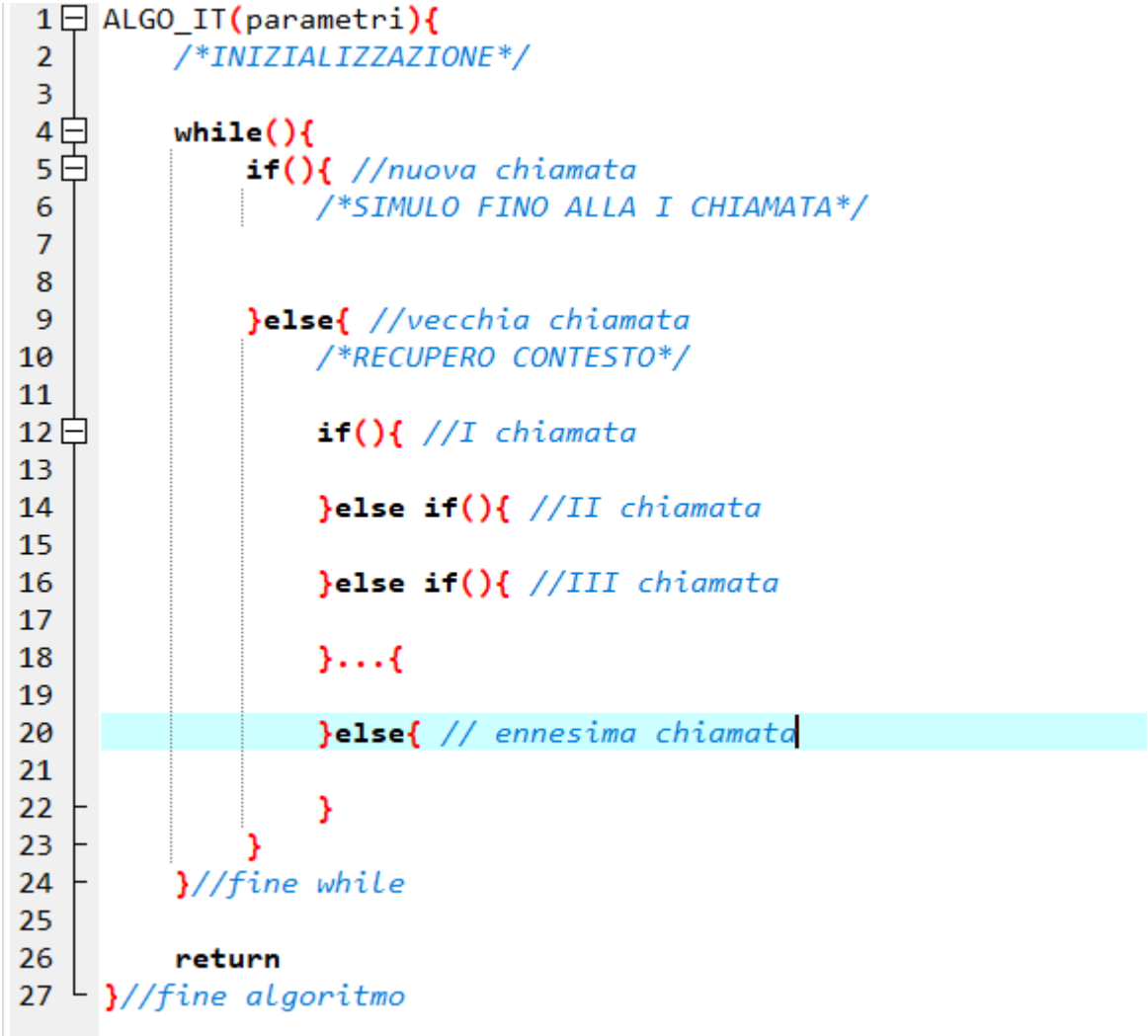
if-else: come avrete già notato, la riga 5 prevede un if che continua con un else alla riga 9. Acuti osservatori, una volta notato questo, vi sarete anche accorti che accanto seguono due commenti "nuova chiamata" e "vecchia chiamata". La nuova chiamata simulerà tutte le istruzioni fino alla prima chiamata ricorsiva. Invece, la parte di vecchia chiamata, simulerà il ritorno da, appunto, una qualsiasi chiamata.

Ultima attenzione va al commento "recupero contesto" a riga 10: subito dopo essere tornati da una chiamata ricorsiva è necessario recuperare tutte le variabili che ci servono, vedremo in seguito come recuperarle e quali sono queste variabili.

La struttura dello scheletro segue due varianti principali, che vedremo adesso: **scheletro array** e **scheletro alberi.**

Scheletro array

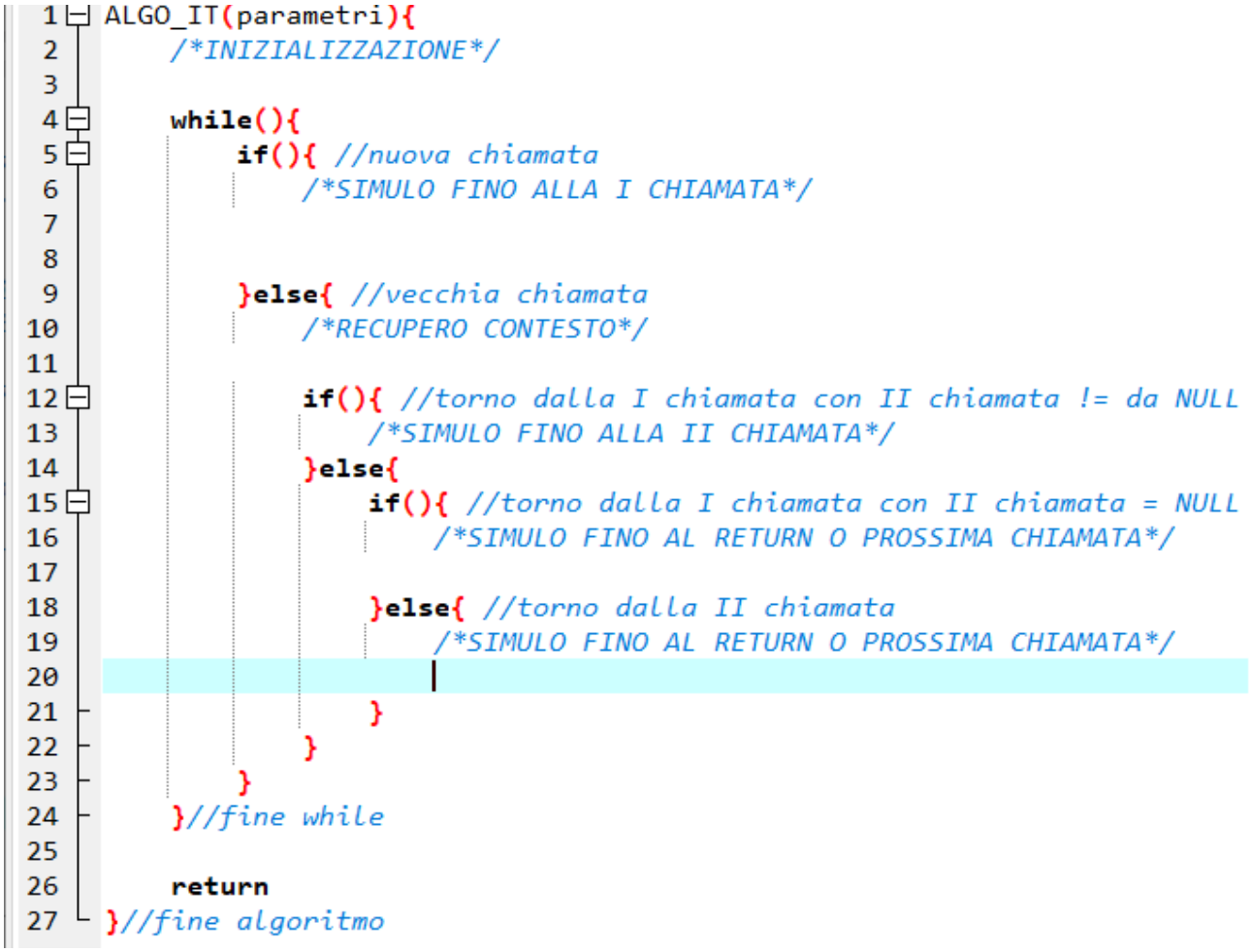
Come mostrato nella figura qui sotto, nel while di riga 9 sono presenti tanti if-else a seconda di quante chiamate ricorsive dobbiamo simulare. Come vedremo in seguito, ci sono delle condizioni vanno messe nell'if per simulare tali chiamate. Per adesso ci basta sapere che le chiamate possono essere distinte.



```
1  ALGO_IT(parametri){
2      /*INIZIALIZZAZIONE*/
3
4      while(){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8
9          }else{ //vecchia chiamata
10             /*RECUPERO CONTESTO*/
11
12             if(){ //I chiamata
13
14             }else if(){ //II chiamata
15
16             }else if(){ //III chiamata
17
18             }...{
19
20             }else{ // ennesima chiamata
21
22             }
23         }
24     }//fine while
25
26     return
27 }//fine algoritmo
```

Scheletro alberi

Lo scheletro degli alberi risulta meno intutivo, ma una volta capito non c'è molto da dire.



```
1  ALGO_IT(parametri){
2      /*INIZIALIZZAZIONE*/
3
4      while(){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8
9          }else{ //vecchia chiamata
10             /*RECUPERO CONTESTO*/
11
12             if(){ //torno dalla I chiamata con II chiamata != da NULL
13                 /*SIMULO FINO ALLA II CHIAMATA*/
14             }else{
15                 if(){ //torno dalla I chiamata con II chiamata = NULL
16                     /*SIMULO FINO AL RETURN O PROSSIMA CHIAMATA*/
17
18                 }else{ //torno dalla II chiamata
19                     /*SIMULO FINO AL RETURN O PROSSIMA CHIAMATA*/
20
21                 }
22             }
23         }
24     }//fine while
25
26     return
27 }//fine algoritmo
```

Come avrete già notato, l'else contenente le vecchie chiamate (riga 9), contiene degli if-else (riga 12) diversi da quelli presentati dallo scheletro degli array. L'if a riga a 12 simula il ritorno dalla I chiamata fino ad arrivare alla seconda, con la seconda chiamata diversa da NULL. Mentre l'if a riga 15 viene eseguita se si torna dalla I chiamata ma la seconda è uguale a NULL, in questo caso la chiamata non

va simulata ma calcolata (nella parte dedita agli esercizi vedremo come). Infine l'else a riga 18 simula il ritorno dalla seconda chiamata.

Le righe 16 e 19 commentate come "SIMULO FINO AL RETURN O PROSSIMA CHIAMATA" sono molto importanti, perché a seconda del numero di chiamate mi comporterò di conseguenza. Infatti la figura sopra, è la traduzione di un algoritmo con esattamente due chiamate ricorsive su alberi.

Mentre nella figura qui sotto possiamo notare come vanno gestite i casi in cui le chiamate ricorsive di una struttura ad albero siano più di due.

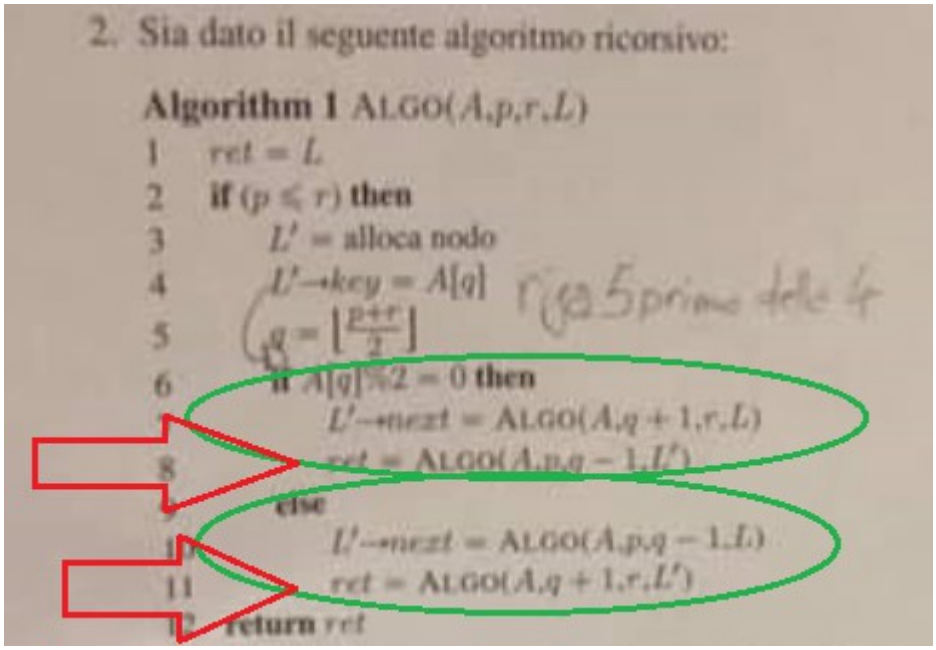
```
1  ALGO_IT(parametri){
2      /*INIZIALIZZAZIONE*/
3
4      while(){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8
9          }else{ //vecchia chiamata
10             /*RECUPERO CONTESTO*/
11
12             if(){ //torno dalla I chiamata con II chiamata != da NULL
13                 /*SIMULO FINO ALLA II CHIAMATA*/
14             }else{
15                 if(){ //torno dalla I chiamata con II chiamata = NULL
16                     /*SIMULO FINO AL RETURN O PROSSIMA CHIAMATA*/
17
18                 }else{ //torno dalla II chiamata
19                     /*SIMULO FINO AL RETURN O PROSSIMA CHIAMATA*/
20
21                     if(){ //torno dalla II chiamata con III chiamata != NULL
22                         /*SIMULO FINO AL RETURN O PROSSIMA CHIAMATA*/
23
24                     }else{
25                         if(){ //torno dalla II chiamata con II chiamata = NULL
26                             /*SIMULO FINO AL RETURN O PROSSIMA CHIAMATA*/
27
28                         }else{ //torno dalla III chiamata
29                             /*SIMULO FINO AL RETURN O PROSSIMA CHIAMATA*/
30                             ...
31
32                         }
33                     }
34                 }
35             }
36         }
37     }//fine while
38
39     return
40 }//fine algoritmo
```

Se il concetto non vi è chiaro da subito, non vi preoccupate, è presente un esercizio nella sezione esercizi, per aiutarvi nella comprensione. La figura simula il ritorno da tre chiamate ricorsive, e vi faccio notare, che rispetto alla figura precedente non è cambiato molto, ho solo reiterato gli if-else, annidandoli l'uno dentro l'altro. I puntini di sospensione "..." indicano che potrei continuare ad annidare all'infinito.

Insiemi e falsi insiemi

Questo è un concetto tanto astratto quanto importante per la costruzione della struttura. Ricordatevi che se non sapete fare la struttura, non passate l'esame, mettetevelo bene in testa.

Cosa intendiamo per **"insiemi e falsi insiemi"**? In pratica, come vedremo nelle regole successive esistono modi per discriminare le chiamate ricorsive tramite parametri, ma esistono modi per discriminarli anche tramite strutture (od insiemi). Prendete in considerazione la seguente traccia:

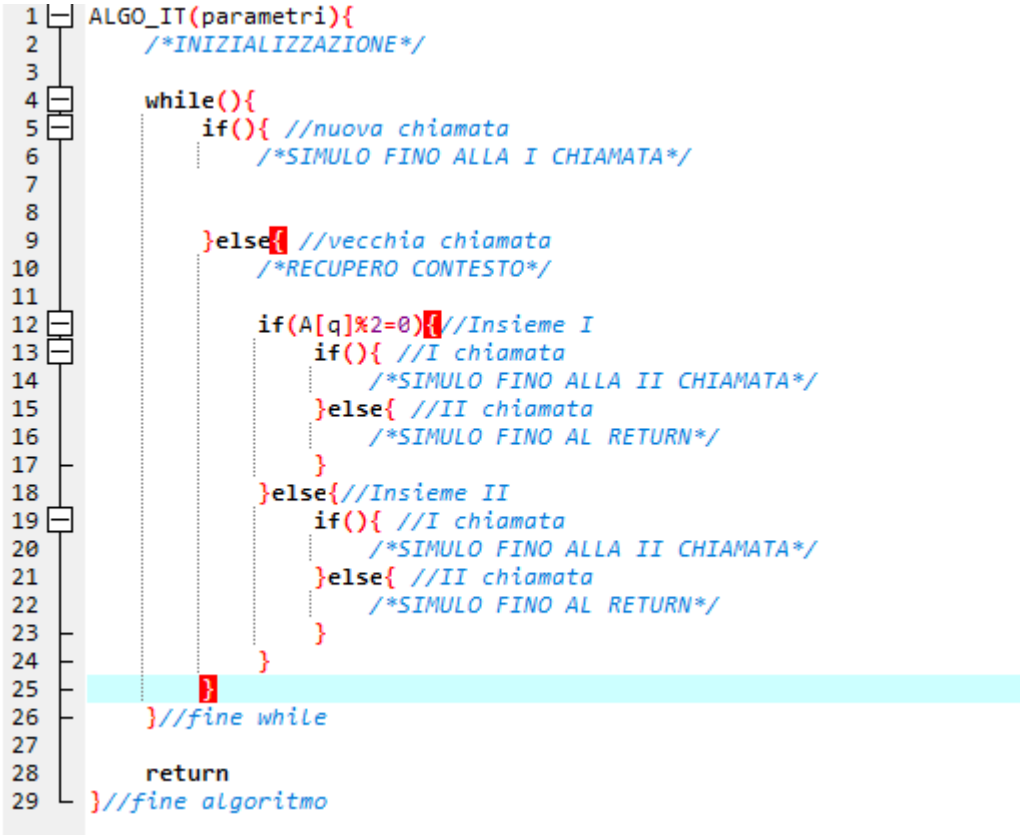

2. Sia dato il seguente algoritmo ricorsivo:

Algorithm 1 ALGO(A,p,r,L)

```
1   ret = L
2   if (p ≤ r) then
3       L' = alloca nodo
4       L'→key = A[q]
5       q = ⌊(p+r)/2⌋
6       if A[q]%2 = 0 then
7           L'→next = ALGO(A,q+1,r,L)
8           ret = ALGO(A,p,q−1,L')
9       else
10          L'→next = ALGO(A,p,q−1,L)
11          ret = ALGO(A,q+1,r,L')
12      return ret
```

Come noterete, le chiamate ricorsive (freccia rossa) sono distinte da due insiemi (verde). Secondo il caro Benny, se possiamo distinguere le chiamate tramite quelli che io definisco come insiemi, **DOBBIAMO** farlo!! In sintesi, per definizione, gli insiemi sono strutture if-else che contengono chiamate ricorsive al loro interno, e se questi sono presenti nella traccia allora le condizioni dell'if-else andranno ricopiate nell'else che gestisce le vecchie chiamate.

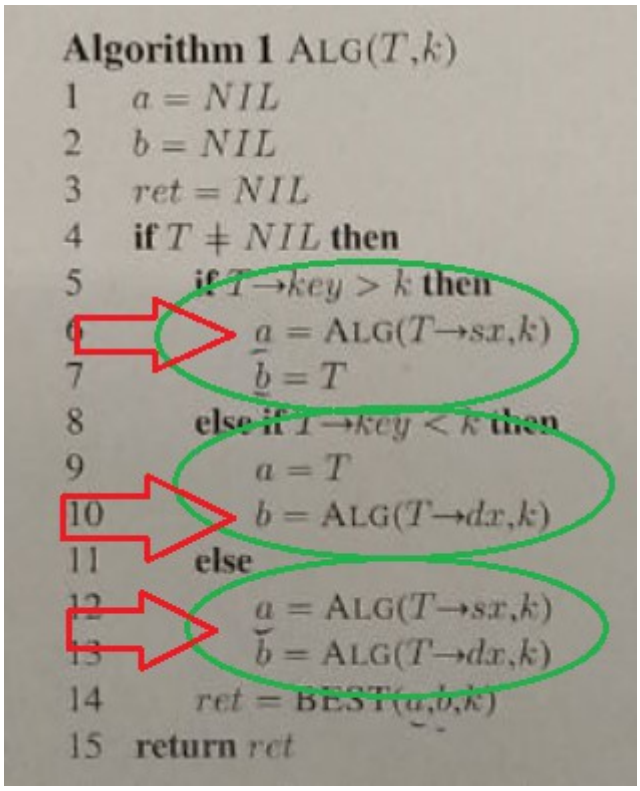
Ecco un esempio di come verrebbe gestita la traccia in alto nella traduzione:



```
1  ALGO_IT(parametri){
2      /*INIZIALIZZAZIONE*/
3
4      while(){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8
9          }else{ //vecchia chiamata
10             /*RECUPERO CONTESTO*/
11
12             if(A[q]%2=0){//Insieme I
13                 if(){ //I chiamata
14                     /*SIMULO FINO ALLA II CHIAMATA*/
15                 }else{ //II chiamata
16                     /*SIMULO FINO AL RETURN*/
17                 }
18             }else{//Insieme II
19                 if(){ //I chiamata
20                     /*SIMULO FINO ALLA II CHIAMATA*/
21                 }else{ //II chiamata
22                     /*SIMULO FINO AL RETURN*/
23                 }
24             }
25         }
26     }//fine while
27
28     return
29 }//fine algoritmo
```

Come noterete dalla figura, la riga 12 (insieme I) simula il primo insieme e distingue la prima coppia di chiamate ricorsive, mentre la riga 18 (insieme II) distingue l'altra coppia di chiamate ricorsive. Nota bene: ho utilizzato il concetto d'insiemi per discriminare le coppie di chiamate, ed il concetto di struttura array visto precedentemente per discriminare le chiamate.

Ecco un secondo esempio:



Algorithm 1 ALG(T,k)

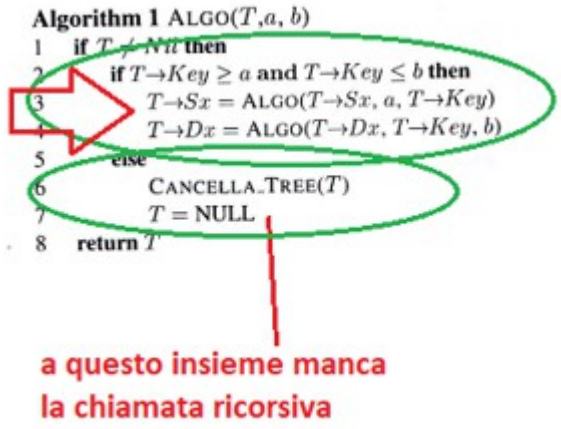
```
1   a = NIL
2   b = NIL
3   ret = NIL
4   if T ≠ NIL then
5       if T→key > k then
6           a = ALG(T→sx,k)
7           b = T
8       else if T→key < k then
9           a = T
10          b = ALG(T→dx,k)
11      else
12          a = ALG(T→sx,k)
13          b = ALG(T→dx,k)
14      ret = BEST(a,b,k)
15  return ret
```

Come noterete in questa traccia sono presentei tre insiemi (verdi) e quattro chiamate ricorsive (freccia rossa). Ecco come andrebbe gestita la traduzione:

```
1  ALGO_IT(parametri){
2      /*INIZIALIZZAZIONE*/
3
4      while(){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8
9          }else{ //vecchia chiamata
10             /*RECUPERO CONTESTO*/
11
12             if(T->key > k){//Insieme I
13
14             }else if(T->key < k){//Insieme II
15
16             }else{ //Insieme III
17                 if(){ //Torno dalla I chiamata con la II chiamata != da NULL
18
19                 }else{
20                     if(){ //Torno dalla I chiamata con la II chiamata = NULL
21
22                     }else{ //Torno dalla seconda chiamata
23
24                     }
25                 }
26             }
27         }
28     }//fine while
29
30     return
31 }//fine algoritmo
```

Da notare che la riga 12 (insieme I) e la riga 14 (insieme II) gestiscono le prime due chiamate ricorsive, mentre la riga 16 (insieme III) le altre due. Per l'insieme I ed l'insieme II non ho bisogno di applicare la struttura ad albero, studiate precedentemente, perché non ho bisogno di discriminare alcuna chiamata dall'altra in quanto sono presenti singole chiamate ricorsive, la condizione dell'insieme stesso mi assicura di tornare da quella chiamata. Mentre per il caso dell'insieme III è stata applicata la struttura albero, perché esso contiene più di una chiamata, quindi ho bisogno di discriminarle in qualche modo. Nota bene: ho utilizzato la struttura ad albero perché la traccia proponeva un algoritmo ricorsivo su un albero.

I falsi insiemi sono quegli insiemi che non contengono chiamate ricorsive, ma che ci potrebbero trarre in inganno, perché simili agli insiemi. Eccovene degli esempi di insiemi e falsi insiemi per convincervi:



La figura in alto è chiaramente un falso insieme perché ad un insieme manca la chiamata ricorsiva, prerequisito

indispensabile secondo la definizione d'insieme data precedentemente; quindi non ha senso riprodurre le condizioni if-else. Di seguito, rispettivamente, l'**unica** struttura possibile implementabile:

```
ALGO_IT(parametri){
    /*INIZIALIZZAZIONE*/


    while(){
        if(){ //nuova chiamata
            /*SIMULO FINO ALLA I CHIAMATA*/

        }else{ //vecchia chiamata
            /*RECUPERO CONTESTO*/

            if(){ //ritorno dalla I chiamata con la II chiamata != da NULL
                /*SIMULO FINO LA II CHIAMATA*/
            }else{
                if(){ //ritorno dalla I chiamata con la II chiamata = NULL
                    /*SIMULO FINO LA II CHIAMATA*/

                }else{ //ritorno dalla II chiamata
                    /*SIMULO FINO AL RETURN*/

                }
            }
        }

    }//fine while

    return
} //fine algoritmo
```

Regola n°2:
gestione degli stack

Se, come me ed il 99% degli studenti, vi siete lasciati come ultima materia ASD allora sicuramente non c'è bisogno che vi spieghi cos'è uno stack, mentre se siete secchioni del 2° anno con tutti gli esami in regola, beh, siete secchioni non c'è comunque bisogno che vi spieghi cosa sia.

La gestione degli stack è uno quegli argomenti che mi ha lasciato perplesso finché il buon Benny non mi ha trasmesso la **"conoscenza"** (che bella parola, non vi riempe la bocca?), a ricevimento. Dopo tanti bla, bla, bla tipici di un professore per farmi capire quando una variabile va inserita in uno stack, io vi riassumo questo concetto con due semplici domande che vanno applicate ad ogni variabile:

1. **L'ho modficata prima della chiamata?**
2. **La dovrò leggere dopo la chiamata e prima della prossima chiamata o del return?**

Semplice no? Per ogni variabile basta porsi queste domande e se la risposta è per entrambe "si", allora

bisogna pusharla per poi essere ripresa nel ritorno da una vecchia chiamata. Queste domande sono applicabili sia in caso siamo nel contesto di nuova chiamata che di vecchi chiamata. Eccezion fatta per le variabili “discriminatorie”, ovvero quelle variabili che mi servono per discriminare le chiamate ricrosive l’una dall’altra, che vedremo in seguito nella regola n°4. Esse vanno pushate solamente nel caso di nuova chiamata, NON di vecchia.

Regola n°3:

salvare il contesto prima di una chiamata

Il solo titolo potrebbe essere sufficiente per la comprensione del capitolo, ma visto che oggi è un martedì pomeriggio, e da piccolo sognavo di fare lo scrittore, mi sembra giusto mettere nero su bianco qualche parola.

Scherzi a parte, come vedremo meglio negli esercizi, prima di ogni chiamata ricorsiva è necessario salvare il contesto, ovvero salvare quelle variabili che rispondono positivamente alle domande della regola n°2. Perché farlo? Nel ricorsivo, ad ogni chiamata, non si perde traccia delle variabili utilizzate (fonti LP1), mentre nell'iterativo non avviene la stessa cosa. Quindi è fondamentale, prima di entrare in una chiamata, salvare **SEMPRE** il contesto, inserendolo nei famosi stack e poi recuperarlo nell'else, prima di iniziare a simulare il ritorno di una chiamata.

Regola n°4:
scegliere un discriminatore

Prima di cimentarsi nella costruzione dell'algorimo iterativo, è fondamentale scegliere, quello che a me piace chiamare **"discriminatore",** ovvero una variabile che mi permetta di distinguere una chiamata da un'altra. Su che base si deve scegliere tale discriminatore? Beh, semplicemente basta guardare i parametri delle chiamate ricorsive e vedere ad occhio, quali cambiano. Esso svolge una funzione importante, in quanto è il protagonista delle condizioni degli if-else che distinguono le chiamate; in seguito, nella sezione apposita per gli esercizi, vedremo come.

Insieme al discriminatore dobbiamo anche scegliere una seconda variabile, che chiameremo **"padre del discriminatore"** che tiene traccia dell'ultimo valore del discriminatore, ovvero il valore che ha assunto prima di entrare in una chiamata. Spesso viene indicato come Last.

Regola n°5:

simulare fino alla chiamata

Ogni qualvolta si torna da una chiamata si simulano tutte le istruzioni che precedono la chiamata ricorsiva seguente, facendo sempre attenzione a rispettare tutte le altre regole.

Regola n°6:
simulare fino il return

Ogni volta vorrete simulare il return di un algoritmo, vi basterà **“forzare la risalita”,** come piace dire al professore. Forzare la risalita non è affatto difficile, essa avviene in quattro passaggi:

1. **Poppare tutti gli stack;**
2. **Assegnare il valore del discriminatore al padre;**
3. **Assegnare il valore che termina il while**
4. **Assegnare il valore alla variabile ret**

Il passaggio n°1 e 3 hanno entrambe lo scopo di terminare il while, tranne che la prima serve anche a svuotare gli stack per fare nuovo spazio per le chiamate sucessive.

Il passaggio n°2 è importante per, appunto distinguere una chiamata dall’altra, infatti all’iterazione successiva è necessario sapere da che chiamata si è tornati per procedere, e lo facciamo tenendo traccia del discriminatore nel padre.

Il passaggio n°4, semplicemente, imita il ritorno **(return)** dell’algoritmo.

Ripeto, che questi passaggi sono indispensabili quando si forza la risalita.

Applicazione

Introduzione

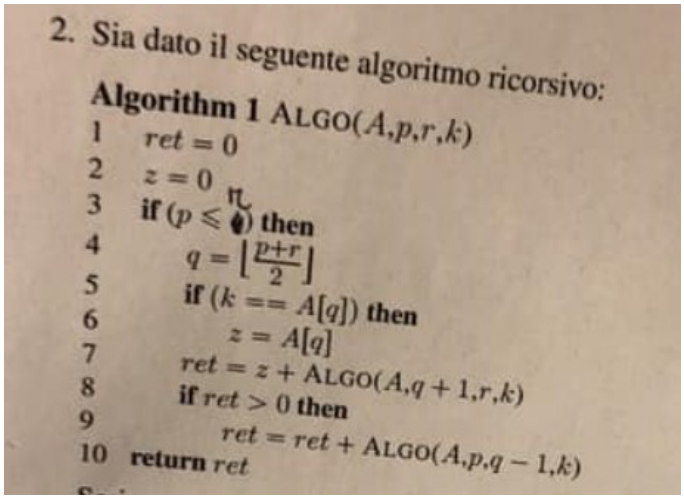
Per una maggiore comprensione della teoria, in questa parte del manuale, procederemo nel vedere il ragionamento che si cela dietro ogni esercizio, inoltre come e quando applicare le regole precedentemente viste.

Ripeto, gli esercizi sono stati portati a ricevimento, tra un insulto e l'altro sono stato capace di estrapolare concetti essenziali che mi hanno permesso di superare lo scritto, ma non è detto che siano "perfetti"; **quindi questo manuale non sostituisce in alcun modo un eventuale ricevimento**, ma spera di porre le basi per uno svolgimento più consapevole da parte dello studente. In sintesi, spero che una volta letto questo manuale non diciate "ma che casso devo fare??!" di fronte ad una traccia di traduzione.

Esercizio:

29/03/19 B

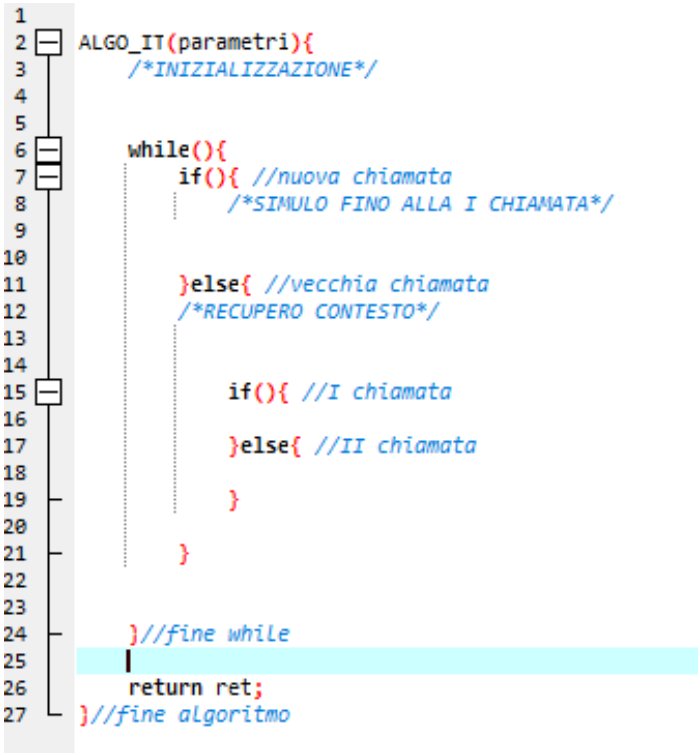
Bando alle ciance e vediamo la costruzione del seguente esercizio:



2. Sia dato il seguente algoritmo ricorsivo:

Algorithm 1 ALGO(A,p,r,k)

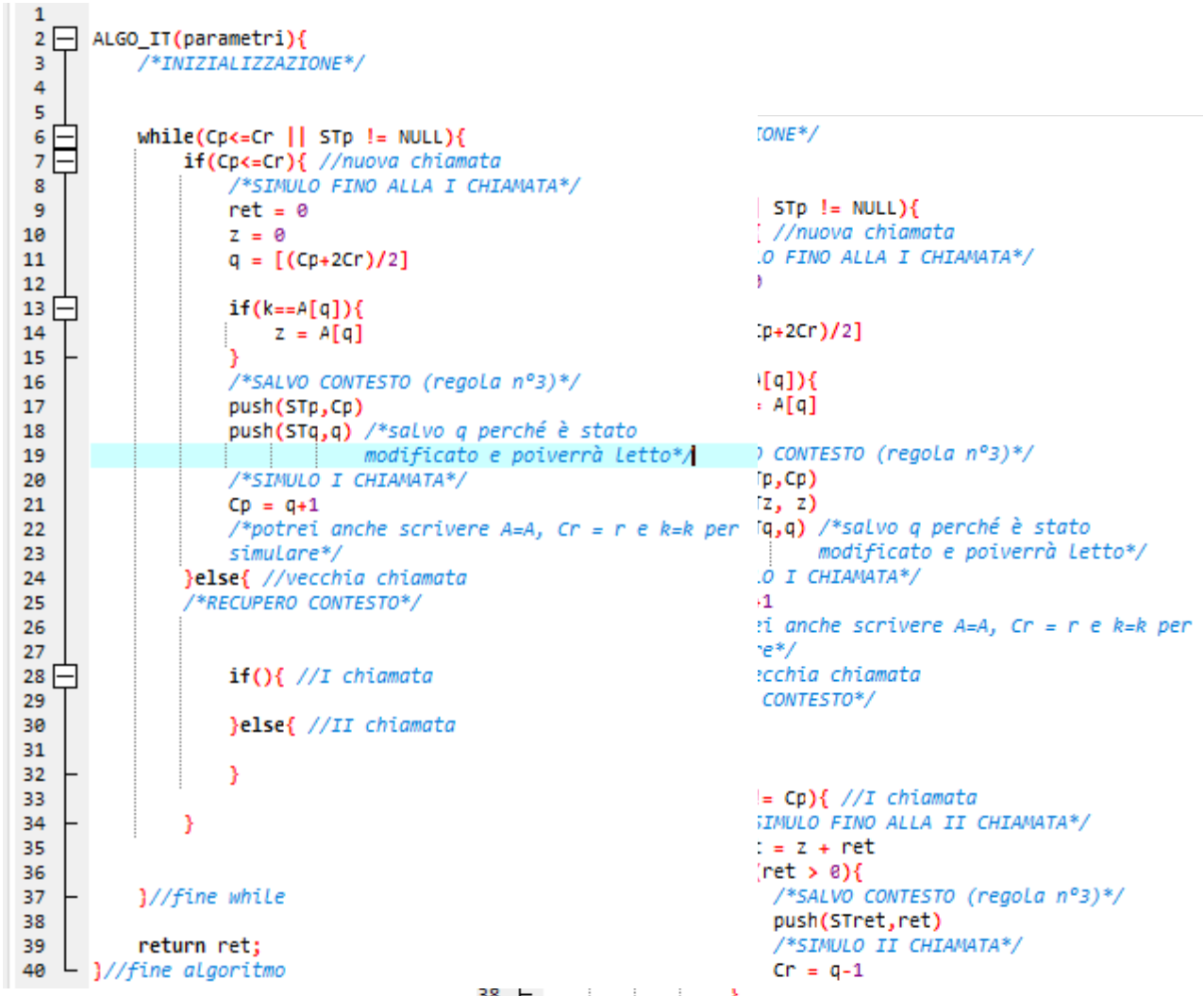
```
1  ret = 0
2  z = 0
3  if (p ≤ r) then
4      q = ⌊(p+r)/2⌋
5      if (k == A[q]) then
6          z = A[q]
7      ret = z + ALGO(A,q + 1,r,k)
8      if ret > 0 then
9          ret = ret + ALGO(A,p,q − 1,k)
10 return ret
```



```
1
2  ALGO_IT(parametri){
3      /*INIZIALIZZAZIONE*/
4
5
6      while(){
7          if(){ //nuova chiamata
8              /*SIMULO FINO ALLA I CHIAMATA*/
9
10
11         }else{ //vecchia chiamata
12         /*RECUPERO CONTESTO*/
13
14
15             if(){ //I chiamata
16
17             }else{ //II chiamata
18
19             }
20
21         }
22
23
24     }//fine while
25
26     return ret;
27 }//fine algoritmo
```

Per prima cosa notiamo che faremo una traduzione ad un ricorsivo array, quindi possiamo tranquillamente applicare la **regola n°1,** utilizzando lo scheletro array. Aggiungiamo tanti if-else, all'interno dell'else che discerne la nuova dalla vecchia chiamata, quante sono le chiamate ricorsive, in questo caso due.

Applico la **regola n°5,** riscrivo tutte le istruzioni finché non arrivo alla prima chiamata ricorsiva (da riga 8 a 15). Arrivati alla prima chiamata, devo applicare la **regola n°3,** ovvero salvare il contesto (da riga 16 a 20). Per decidere quale variabile salvare, faccio affidamento alla **regola n°2,** quindi guardo quali sono le variabili che sono modificate prima della chiamata e lette dopo la chiamata e prima della prossima chiamata, in questo caso Cp ,q e z. Nota bene: controllo dalla traccia quali sono quelle modificate prima e lette dopo la chiamata.



```
ALGO_IT(parametri){
    /*INIZIALIZZAZIONE*/


    while(Cp<=Cr || STp != NULL){
        if(Cp<=Cr){ //nuova chiamata
            /*SIMULO FINO ALLA I CHIAMATA*/
            ret = 0
            z = 0
            q = [(Cp+2Cr)/2]

            if(k==A[q]){
                z = A[q]
            }
            /*SALVO CONTESTO (regola n°3)*/
            push(STp,Cp)
            push(STq,q) /*salvo q perché è stato
                        modificato e poiverrà letto*/
            /*SIMULO I CHIAMATA*/
            Cp = q+1
            /*potrei anche scrivere A=A, Cr = r e k=k per
            simulare*/
        }else{ //vecchia chiamata
        /*RECUPERO CONTESTO*/


            if(){ //I chiamata

            }else{ //II chiamata

            }

        }


    }//fine while

    return ret;
}//fine algoritmo
```

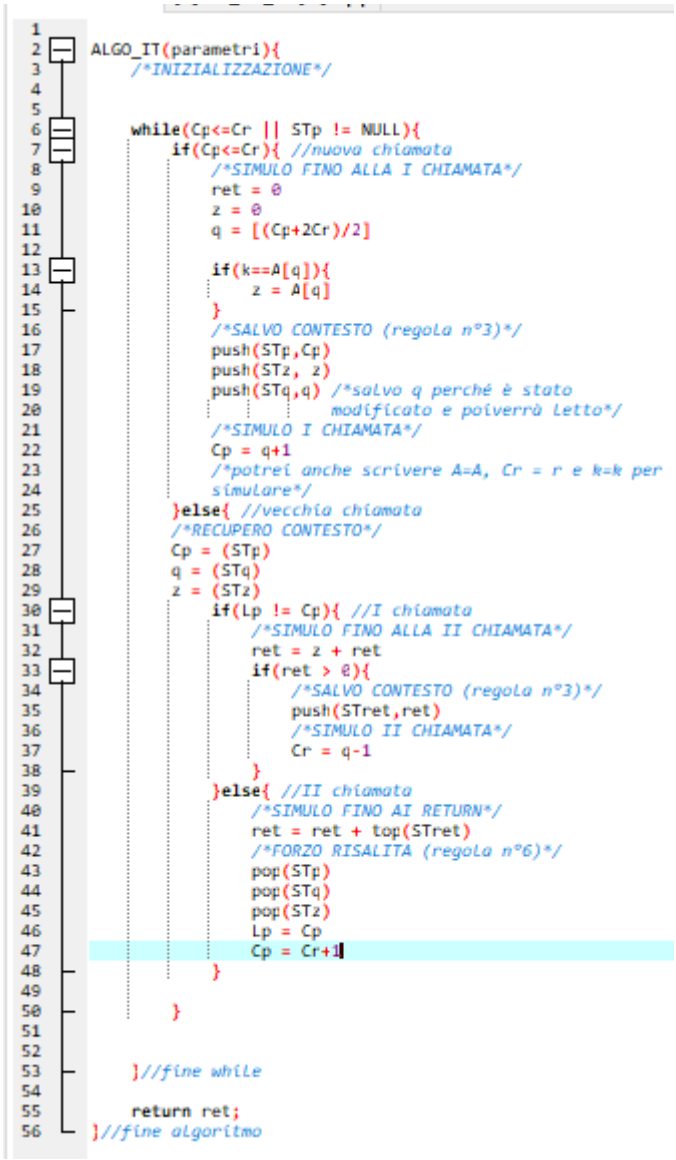
```
        }
    }else{ //II chiamata
        /*SIMULO FINO AI RETURN*/
        ret = ret + top(STret)

    }

}
```

Procedendo entriamo nell'else. Qui, come richiede lo scheletro, recuperiamo tutte le variabili che andranno lette.

Secondo la **regola n°4** scegliamo un discriminatore, in questo caso p è il nostro discriminatore, in quanto, come si nota dalla stessa traccia, i parametri che cambiano sono 2: p ed r. Fra i due ho scelto, senza nessun criterio particolare, di confrontare p, quindi utilizzerò la variabile Lp (padre del discriminatore) che tiene traccia del valore

del discriminatore e lo confronta una volta tornati da una chiamata, per capire da quale chiamata sono tornato.

Ritornati dalla prima chiamata simulo tutto ciò che viene dopo la prima chiamata fino ad arrivare alla seconda(da riga 31 a 33). Pusho la variabile ret nello stack, poiché viene modificata ed in seguito letta, come richiesto dalla **regola n°2** (riga35).

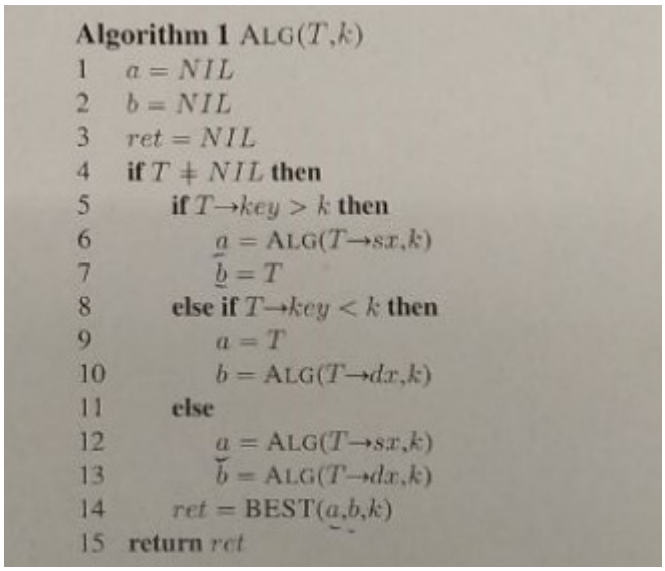


```
1
2   ALGO_IT(parametri){
3       /*INIZIALIZZAZIONE*/
4
5
6       while(Cp<=Cr || STp != NULL){
7           if(Cp<=Cr){ //nuova chiamata
8               /*SIMULO FINO ALLA I CHIAMATA*/
9               ret = 0
10              z = 0
11              q = [(Cp+2Cr)/2]
12
13              if(k==A[q]){
14                  z = A[q]
15              }
16              /*SALVO CONTESTO (regola n°3)*/
17              push(STp,Cp)
18              push(STz, z)
19              push(STq,q) /*salvo q perché è stato
20                            modificato e poiverrà letto*/
21              /*SIMULO I CHIAMATA*/
22              Cp = q+1
23              /*potrei anche scrivere A=A, Cr = r e k=k per
24              simulare*/
25          }else{ //vecchia chiamata
26          /*RECUPERO CONTESTO*/
27          Cp = (STp)
28          q = (STq)
29          z = (STz)
30              if(Lp != Cp){ //I chiamata
31                  /*SIMULO FINO ALLA II CHIAMATA*/
32                  ret = z + ret
33                  if(ret > 0){
34                      /*SALVO CONTESTO (regola n°3)*/
35                      push(STret,ret)
36                      /*SIMULO II CHIAMATA*/
37                      Cr = q-1
38                  }
39              }else{ //II chiamata
40                  /*SIMULO FINO AI RETURN*/
41                  ret = ret + top(STret)
42                  /*FORZO RISALITA (regola n°6)*/
43                  pop(STp)
44                  pop(STq)
45                  pop(STz)
46                  Lp = Cp
47                  Cp = Cr+1
48              }
49
50          }
51
52
53      }//fine while
54
55      return ret;
56  }//fine algoritmo
```

Mentre, se torno dalla seconda chiamata, dovrò simulare fino al return, quindi secondo la **regola n°6** popperò tutti gli stack così da dare spazio alle prossime variabili e terminare il while (riga da 43 a 45). Assegnerò il valore del discriminatore al padre, per tenerne traccia ai cicli successivi (riga 46). Forzo la terminazione del while (riga 47). Assegnerei pure ret = ret, per rimanere fedele ai passaggi della regola, ma non è necessario ed il prof lo considererebbe un errore; il prof considera errori tutti gli sprechi. Fine esercizio.

Esercizio:

24/01/19



```
Algorithm 1 ALG(T,k)
1   a = NIL
2   b = NIL
3   ret = NIL
4   if T ≠ NIL then
5       if T→key > k then
6           a = ALG(T→sx,k)
7           b = T
8       else if T→key < k then
9           a = T
10          b = ALG(T→dx,k)
11      else
12          a = ALG(T→sx,k)
13          b = ALG(T→dx,k)
14      ret = BEST(a,b,k)
15  return ret
```

Come vediamo, la traccia divide le chiamate ricorsive tramite tre insiemi (riga 5,8 ed 11), quindi, come spiegato nella **regola n°1** posso da subito replicare nella struttura le condizioni degli if-else degli insiemi. Nella figura a destra gli insiemi corrispondono rispettivamente a riga 11, 14 e 17. Da notare che per l'insieme I e l'insieme II non utilizzo la struttura albero vista nella **regola n°1**, poiché le chiamate all'interno dell'insieme sono singole, ciò significa che già quando entro in quella condizione mi riferisco a quella chiamata, **non ho bisogno di fare alcuna distinzione**. Discorso diverso per quanto rigaurda l'insieme III, di riga 17 nel codice: ho due chiamate ricorsive, quindi mi serve discriminare le due chiamate in qualche modo.

Come faccio? Applica la **regola n°4**, ovvero vedo per quali parametri si differenziano le chiamate.

```
1  ALGO_IT(){
2      /*INIZIALIZZAZIONE*/
3
4      while(CT != NULL || STt != NULL){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8          }else{ //vecchia chiamata
9              /*RECUPERO CONTESTO*/
10
11             if(CT->key > k){ //Insieme I
12                 /*SIMULO FINO AL RETURN*/
13
14             }else if(CT->key < k){ //Insieme II
15                 /*SIMULO FINO AL RETURN*/
16
17             }else{ //Insieme III
18                 if(){ //torno dalla I chiamata con la II != da NULL
19                     /*SIMULO FINO ALLA II CHIAMATA*/
20
21                 }else{
22                     if(){ //torno dalla I chiamata con la II = NULL
23                         /*SIMULO FINO AL RETURN*/
24
25                     }else{ //torno dalla II
```

Nella figura a sinistra notiamo che sono state modificate le righe

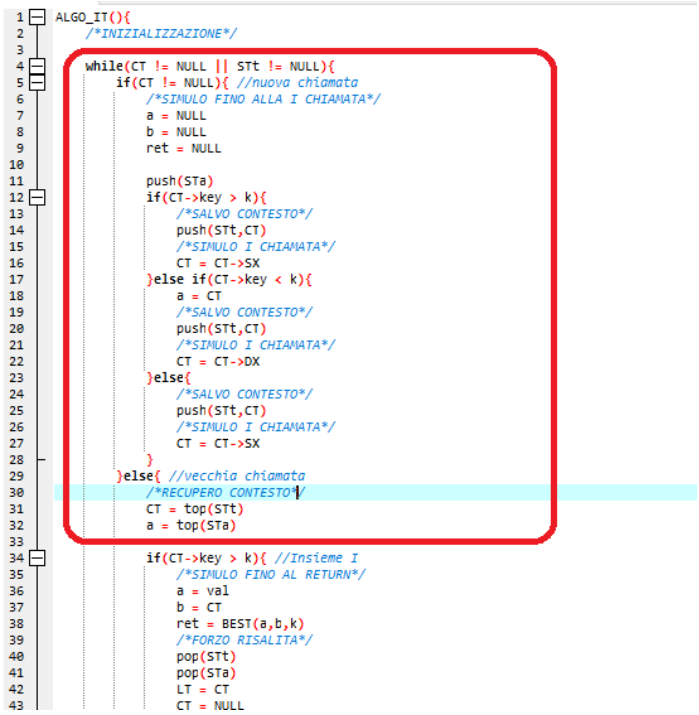
```
2      /*INIZIALIZZAZIONE*/
3
4      while(CT != NULL || STt != NULL){
5          if(){ //nuova chiamata
6              /*SIMULO FINO ALLA I CHIAMATA*/
7
8          }else{ //vecchia chiamata
9              /*RECUPERO CONTESTO*/
10
11             if(CT->key > k){ //Insieme I
12                 /*SIMULO FINO AL RETURN*/
13
14             }else if(CT->key < k){ //Insieme II
15                 /*SIMULO FINO AL RETURN*/
16
17             }else{ //Insieme III
18                 if(LT != CT->DX && CT->DX != NULL){ //torno dalla I chiamata con la II != da NULL
19                     /*SIMULO FINO ALLA II CHIAMATA*/
20
21                 }else{
22                     if(CT->DX = NULL){ //torno dalla I chiamata con la II = NULL
23                         /*SIMULO FINO AL RETURN*/
24
25                     }else{ //torno dalla II
26                         /*SIMULO FINO AL RETURN*/
27
28                     }
29                 }
30             }
31
32         }
33
34     }//fine while
35
36 }//fine algoritmo
```

18 e 22. **LT è il padre** che tiene traccia di CT.

Il riquadro rosso indica dove sono avvenute le modifiche.

Costruita la struttura posso andare ad inizio codice (riga 7) ed applicare la **regola n°5**, ovvero simulare fino all'arrivo della prima chiamata. Ho dovuto ricopiare la struttura if-elseif-else perché non so a quale chiamata entrerò. Potrei entrare nella prima condizione così come potrei entrare nella seconda o terza. In ogni caso, prima di entrare nella chiamata (o simularla) la **regola n°3** m'impone di salvare il contesto, e la **regola n°2** mi dice qualche variabile va

salvata. Le variabili salvate sono due: CT ed a. La variabile a viene modificata nel secondo insieme e letta dalla funzione BEST(). Negli altri due insiemi non viene letta, quindi potrei anche evitare di salvarla a riga 11 del codice e topparla nel secondo insieme della vecchia chiamata, ma anche così va bene, a patto che ci ricordiamo di popparla quando la usiamo od al forzamento della risalita.



Infine applico la **regola n°5** ove trovo scritto “SIMULO FINO ALLA n CHIAMATA”, proprio perché sto tornando dalla chiamata precedente; ed applico la **regola n°6** ove trovo scritto “FORZO RISALITA”.

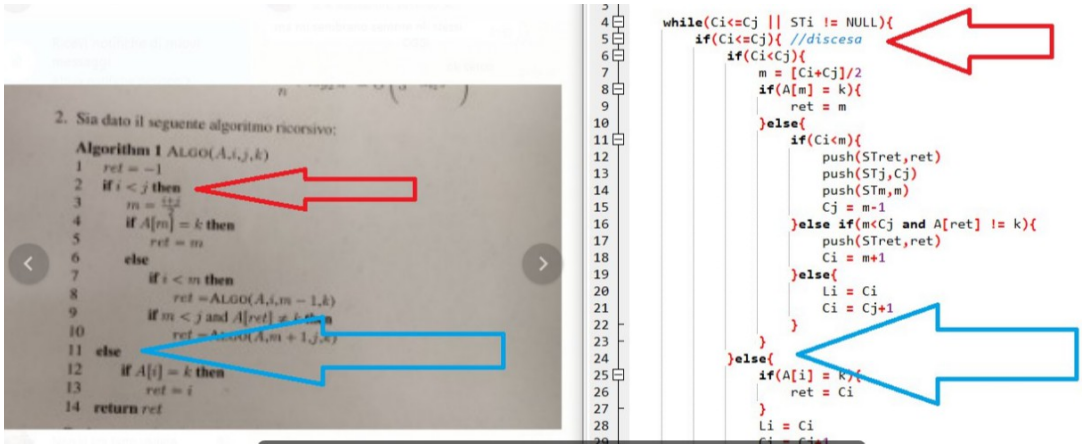
Maggiore attenzione però va dedicata alla riga 58 del codice, in quanto, la variabile a viene modificata prima della seconda chiamata ed in seguito letta dalla funzione BEST(). Infatti a riga 58 mi libero del vecchio valore di a poi pusho il nuovo; non farlo verrebbe considerato errore, perché lo stack si riempirebbe di valori inutili.

```
28              }
29          }else{ //vecchia chiamata
30              /*RECUPERO CONTESTO*/
31              CT = top(STt)
32              a = top(STz)
33
34              if(CT->key > k){ //Insieme I
35                  /*SIMULO FINO AL RETURN*/
36                  a = val
37                  b = CT
38                  ret = BEST(z,t,k)
39                  /*FORZO RISALITA*/
40                  pop(STt)
41                  pop(STz)
42                  LT = CT
43                  CT = NULL
44              }else if(CT->key < k){ //Insieme II
45                  /*SIMULO FINO AL RETURN*/
46                  b = val
47                  ret = BEST(z,t,k)
48                  /*FORZO RISALITA*/
49                  pop(STt)
50                  pop(STz)
51                  LT = CT
52                  CT = NULL
53              }else{ //Insieme III
54                  if(LT != CT->DX && CT->DX != NULL){ //torno dalla I chiamata con la II != da NULL
55                      /*SIMULO FINO ALLA II CHIAMATA*/
56                      a = val
57                      /*SALVO CONTESTO*/
58                      pop(STz) //poppo perché ho il valore vecchio dato a riga 31, e bisogna liberare lo stack
59                      push(STz)
60                      /*SIMULO II CHIAMATA*/
61                      CT = CT->DX
62                  }else{
63                      if(CT->DX = NULL){ //torno dalla I chiamata con la II = NULL
64                          /*SIMULO FINO AL RETURN*/
65                          a = val
66                          b = NULL //calcolo la seconda chiamata quando CT = NULL
67                          ret = BEST(z,t,k)
68                          /*FORZO RISALITA*/
69                          pop(STz)
70                          pop(STt)
71                          LT = CT
72                          CT = NULL
73                      }else{ //torno dalla II
74                          /*SIMULO FINO AL RETURN*/
75                          b = val
76                          ret = BEST(z,t,k)
77                          /*FORZO RISALITA*/
78                          pop(STz)
79                          pop(STt)
80                          LT = CT
81                          CT = NULL
82                      }
83                  }
84              }
85
86          }
87
88      }//fine while
89
90      return ret
91  }//fine algoritmo
```

Esercizio:

Caso base

Delle volte, c'è bisogno di gestire quello che viene chiamato caso base. Come facciamo ad accorgerci se dobbiamo essere noi a gestire il caso base o l'algoritmo lo fa già da solo? Già ad occhio nelle tracce si può notare che non c'è un unico if che gestisce tutto, ma anche un else. Se ci trovassimo in quella condizione, allora è necessario gestire il caso base, come già detto. Non farlo comporterebbe la bocciatura, perché ripetono nuovamente: se si sbaglia la struttura si è bocciati. Nelle figure in basso possiamo vedere come in due tracce (le uniche trovate in giro - 1) viene gestito.



Nella figura a destra, la freccia rossa indica l'if che, come detto, stranamente non contiene tutto l'algoritmo, anzi una parte è gestita dall'else indicato con la freccia blu. La soluzione sta nel gestire il caso base, ma qual è il caso base? i = j è il caso base. Nel codice (sostituite il commento "discesa" con

“nuova chiamata”), la nuova chiamata gestirà, sia il caso base che non. Fatta la struttura, il resto della traduzione può avvenire applicando le regole come abbiamo sempre fatto.

Anche in questo caso, nella figura a destra, abbiamo una “anomalia”: l’if di riga 1 nella traccia, non contiene tutto l’algoritmo, anzi viene spezzato dall’else di riga 12. Anche in questo caso, salta all’occhio che dobbiamo gestire il caso base. La traduzione per la gestione del caso base è data dalla figura accanto. Il caso base è j – i >= 0.

Traccia B 16/06/2021

Scrivere un algoritmo **iterativo** che **simuli precisamente** il comportamento ricorsivo dell'algoritmo sotto riportato, dove i parametri i e j sono valori interi, A un array di interi e x un valore Booleano.

```
1  ALGORITMO(A,i,j,x)
1    if j − i >= 1 then
2      y = Rand()%2
3      if (x = 1) then
4        ret = ALGORITMO(A, ⌊(i+j)/2⌋ + 1, j, y)
5        if ret%2 = 0 then
6          ret = ALGORITMO(A, i, ⌊(i+j)/2⌋, 1 − y)
7      else
8        ret = ALGORITMO(A, i, ⌊(i+j)/2⌋, y)
9        if ret%2 = 1 then
10         ret = ret+ALGORITMO(A, ⌊(i+j)/2⌋ + 1, j, 1 − y)
11   else
12     ret = A[i]
13   return ret
```

```
18  ALGO_IT(A,i,j,x){
19
20      /*INIZIALIZZAZIONE*/
21
22      while(Cj-Ci>=0 || ST != NULL){
23          if(Ci-Cj>=0){ //discesa
24              if(Ci-Cj>=1){
25                  y = Rand()%2
26                  if(x=1){
27                      push(STi,Ci)
28                      push(STy,y)
29                      push(STx,Cx)
30                      Ci = [(Ci+Cj)/2]+1
31                      Cx = y
32                  }else{
33                      push(STy,y)
34                      push(STj,Cj)
35                      push(Stx,Cx)
36                      Cx = 1-y
37                  }
38              }else{
39                  ret = A[i]
40                  val = ret
41                  Lj = Cj
42                  Cj = Ci-1
43              }
44          }else{ //ripresa
45              /*RECUPERO*/
46              Ci = top(STi)
47              Cj = top(STj)
```

Discorso finale

Spero che il manuale possa essere utile a qualcuno, anzi spero d'innalzare di molto la media dei promossi. In ogni caso il manuale non sostituisce il prof né un eventuale ricevimento, anzi questo va integrato.

Che la forza sia con voi, giovani studenti.