Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Strutture dati dinamiche in C

Liste II

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **24**, 24 maggio 2021

Lezione 24a:

Programmi su Liste: rovesciamento

Rovesciamento di Liste

Cominciamo sempre con la specifica funzionale di una funzione che chiameremo `reverse`:

```
reverse    []    = []
reverse (x:xs) = addTail x (reverse xs)
```

Al solito, modulo sintassi, si possono tradurre queste equazioni ricorsive in **C** ottenendo la versione `Fun` di `reverse`:

```
lista reverseFun(lista L){
   if (!L) return NULL;
   return addTailRec(L->val, reverseFun(L->next));
}
```

Osserviamo due questioni interessanti:

- Perché usiamo `addTailRec`?
- La funzione fa un numero lineare di chiamate ricorsive, ma poi `addTailRec` è a sua volta lineare: quindi $\mathcal{O}(n^2)$.

Rovesciamento di Liste iterativo

D'altra parte, parlando della funzione `twiceL`, abbiamo visto che in una scansione iterativa, usando `cons` produce una lista in ordine rovesciato. Quindi il seguente programma iterativo:

```
int reverseFunIt(lista L){
   lista R = NULL;
   while (L){
         /* L0 = L ++ reverse(R) */
      R = cons(L->val, R);
      L = L->next;
   }
   return R;
}
```

rovescia con successo una lista iterativamente. La questione è che il programma ricorsivo **deve aggiungere al rientro delle chiamate ricorsive**, mentre il programma iterativo **scansiona una lista solo "in avanti"**.

Rovesciamento di Liste ricorsivo

Il trucco per ottenere una funzione ricorsiva efficiente è **costruire la lista risultato all'andata**. Tuttavia occorre memorizzare i risultati parziali della computazione in un **parametro ausiliario** e quindi definire una funzione ausiliaria a due parametri:

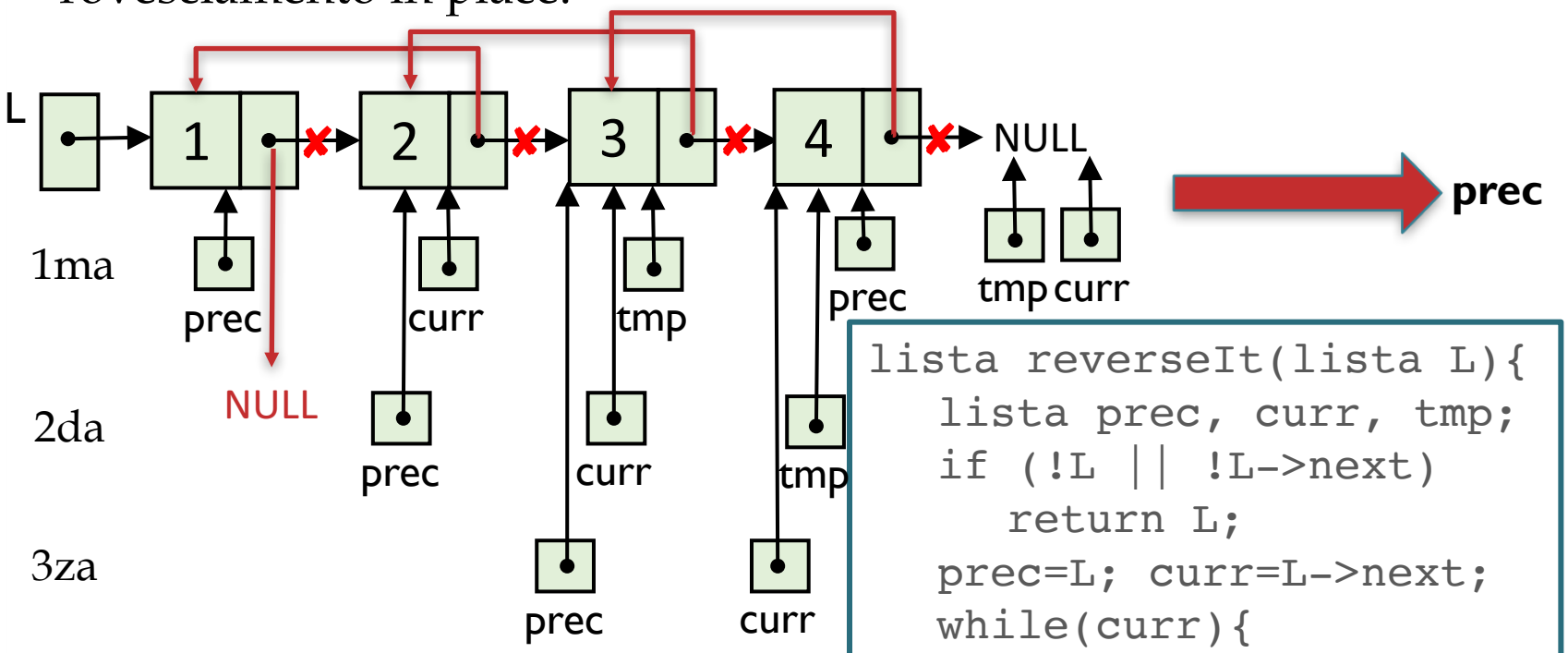
```
int reverseFun(lista L){
   return reverseFunAux(L, NULL);
}

int reverseFunAux(lista L, lista R){
   if (!L) return R;
   return reverseFunAux(L->next, cons(L->val, R));
}
```

Questo corrisponde all'analoga funzione Haskell efficiente.

Rovesciamento in-place iterativa

Per anni ho pensato fosse necessario questo virtuosismo per il rovesciamento in place.



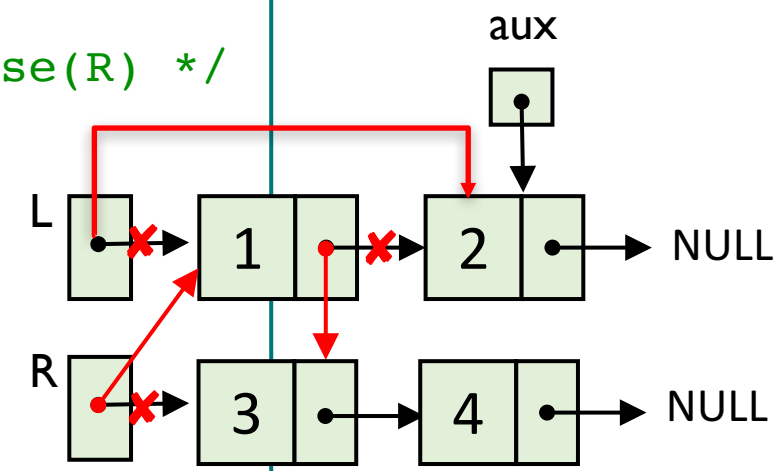

```
lista reverseIt(lista L){
   lista prec, curr, tmp;
   if (!L || !L->next)
      return L;
   prec=L; curr=L->next;
   while(curr){
      tmp=curr->next;
      curr->next=prec;
      prec=curr; curr=tmp;
   }
   L->next=NULL;
   return prec;
}
```

Rovesciamento di Liste iterativo

Ma in realtà è sufficiente adattare il rovesciamento `Fun`, evitando di fare `cons`, e spostando i pointer ricostruendo una nuova lista.

Il trucco consiste nell'avere una lista (cioè un pointer) su cui "appoggiare" i record del risultato…

```
int reversePlaceIt(lista L){
   lista R = NULL;
   lista aux;
   while (L){
         /* L0 = L ++ reverse(R) */
      aux = L->next;
      L->next = R;
      R = L;
      L = aux;
   }
   return R;
}
```



Lezione 24b:

Rimozione di elementi e deallocazione

Rimozione di tutte le occorrenze

Finora abbiamo scritto funzioni che modificano liste, ma sempre aggiungendo elementi o spostando puntatori.

Cominciamo con la funzione che rimuove tutte le occorrenze di un certo intero da una lista. Al solito cominciamo con le equazioni ricorsive:

```
remove    []    y = []
remove (x:xs) y = if x == y then remove xs y
                            else x:remove xs y
```

da cui al solito si ricava immediatamente del codice C in modo standard (ovviamente in versione Fun):

```
lista removeFun(lista L, int x){
   if (!L) return NULL;
   if (L->val == x) return removeFun(L->next, x);
   return cons(L->val, removeFun(L->next, x));
}
```

Dovendo creare una nuova lista, questa funzione **si limita a non ricopiare** gli elementi che non appartengono al risultato.

Rimozione di tutte le occorrenze

Più interessante la funzione `removeRec`: dovendo modificare la lista, oltre a spostare i puntatori è opportuno **deallocare** i nodi rimossi dalla lista (che altrimenti rimarrebbero irrangiugibili, quindi garbage e produrrebbero **memory leak**).

Occorre usare la funzione di libreria **`free(void*)`** (contenuta nella libreria `stdlib.h` come `malloc/calloc`). Occorre stare attenti, perché eliminando un nodo occorre farlo solo **dopo** che abbiamo modo di continuare a scorrere la lista.

```
int removeRec(lista L, int x){
   lista M;
   if (!L) return NULL;
   if (L->val == x){
         M = L->next;
         free(L);
         return removeRec(M, x);
   }
   L->next = removeRec(L->next
   return L;
}
```

Prima di liberare L, *salvo il pointer* L->next

È sempre utile la tecnica di "riassegnare" L->next

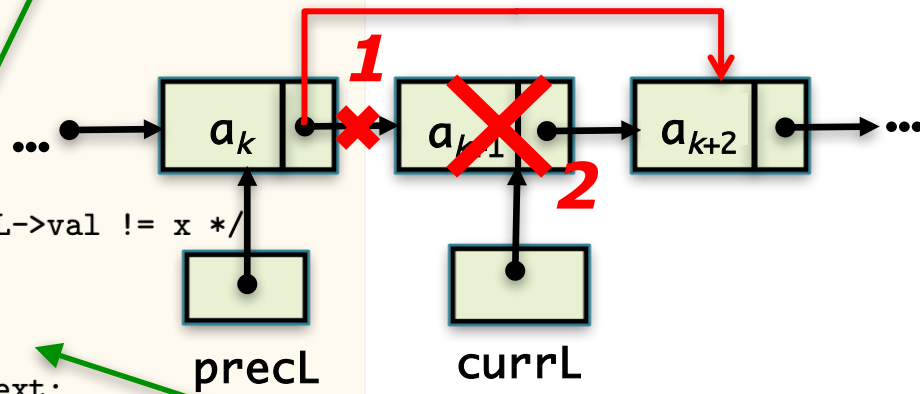
Rimozione iterativa

Al solito, più impegnativa la versione iterativa, un po' come la reverse: non avendo l'informazione disseminata nei record di attivazione e i valori di ritorno, occorre stare attenti!

```
lista removeIt(lista L, int x){
   lista inizioL = L;
   lista currL, precL, tmp;

      /* rimuove tutte le occorrenze iniziali di x. */
   while (inizioL && inizioL->val == x){
      tmp = inizioL->next;
      free(inizioL);
      inizioL = tmp;
    }
    precL = inizioL;
      /* alla prima iterazione precL->val != x */
    while (precL){
       currL = precL->next;
       if (currL && currL->val==x){
           1 precL->next = currL->next;
           2 free(currL);
          }
        precL = precL->next;
    }
    return inizioL;
}
```

Il primo ciclo serve a rimuovere eventuali sequenze iniziali di x e determinare il risultato



Come nel caso della reverse, occorrono due pointer consecutivi

Differenza tra liste

Poniamoci il problema di scrivere una funzione che rimuove da una lista M tutti gli elementi appartenenti a un'altra lista L. Possiamo ovviamente sfruttare remove:

```
diff xs    []    = xs
diff xs (y:ys) = diff (remove xs y) ys
```

da cui al solito si ricava immediatamente del codice **C** in modo standard. In questo caso la versione **`Fun`** e la versione **`Rec`** differiscono **solo nel fatto** che la prima chiama **`removeFun`**, mentre la seconda chiama **`removeRec`**:

```
lista diffRec(lista L, lista M){
   if (!M) return L;
   return diffRec(removeRec(L, M->val), M->next);
}
```

Attenzione: L'analoga versione **`Fun`** che chiama `removeFun` purtroppo **genera una gran quantità di liste che vanno perdute**!

Alternative per la versione Fun

Ovviamente possiamo sempre battere altre strade e costruire direttamente la lista risultato e determinare piuttosto gli elementi che appartengono alla lista risultato:

```
diff []     ys = []
diff (x:xs) ys = if belongsTo x ys
                 then diff xs ys
                 else x:diff xs vs
```

```
int belongsTo(lista L, int x){
   if (isEmpty(L)) return 0;
   if (head(L)==x) return 1;
   return belongsTo(tail(L));
}

lista diffFun(lista L, lista M){
   if (isEmpty(L)) return NULL;
   if (belongsTo(M, head(L)))
       return diffFun(tail(L), M);
   return cons(diffFun(tail(L), M), head(L));
}
```

Però osservate che i Linguaggi Funzionali **devono** sistematicamente **evitare l'eccessiva allocazione di memoria**!

Alternative per la versione Fun

Come sempre c'è un trucco per ottenere versioni Fun quando lo schema ricorsivo sembra un po' complicato oppure **quando si ottiene qualcosa che genera troppa memoria**: creare una copia della lista e poi chiamare una versione Rec sulla copia.

```
lista diffFun(lista L, lista M){
   return diffRec(copy(L), M);
}
```

Se invece volessi fare la differenza simmetrica?

Esercizio: Trovare due liste *L* ed *M*, per cui: il programma:

```
lista diffSimm(lista L, lista M){
   return append(diffRec(L, M),
      diffRec(M, L));
}
```

Algebricamente corretto.
Computazionalmente no.

dà risultati diversi se chiamato con `diffSimm(L, M)` rispetto a `diffSimm(M, L)`.

Elimina Duplicati

Facciamo un ultimo esempio: scrivere una funzione che elimina i duplicati da una lista. Anche qui, possiamo scrivere facilmente delle equazioni ricorsive usando remove:

```
elimDupl   []    = []
elimDupl (x:xs) = x:elimDupl(remove x xs)
```

Ancora una volta, **attenzione all'implementazione C**! Nelle dispense concludo affrettatamente che, per evitare generazione incontrollata di liste durante il calcolo, meglio usare `removeRec`.

```
lista elimDuplFun(lista L){
   if (!L) return NULL;
   return cons(L->val,
      elimDuplFun(removeRec(L, L->val)));
}
```

Ma è questa una vera versione Fun? Cosa fa di sbagliato? Come vedete, **anch'io finisco facilmente in trappola…**

Deallocazione di una lista

Chiudiamo con le liste con una funzione che si limita a deallocare una lista dalla memoria.

Non scrivo le equazioni ricorsive perché chiaramente **questo problema non ha senso in un mondo di valori** senza memoria.

```
void deallocaRec(lista L){
   if (L){
      deallocaRec(L->next);
      free(L);
   }
}
```

```
void deallocaIt(lista L){
   lista prec=L;
   lista curr;
   while(prec){
      curr = prec->next;
      free(prec);
      prec = curr;
   }
}
```

Lezione 24c:

Liste ordinate e ordinamenti

Inserimento ordinato in lista ordinata

Queste funzioni sono del tutto analoghe all'inserimento in coda. Occorre solo prestare attenzione a **dove posizionarsi** per effettuare l'inserimento.

E a **ricopiare la coda** per ottenere un vero comportamento Fun!

```
lista insertRec(lista L, int x){
   if (!L || L->val>=x) return cons(L, x);
   L->next = insertRec(L->next, x);
   return L;
}
```

```
lista insertFun(lista L, int x){
   if (!L) return cons(L, x);
   if (L->val>=x) return cons(copy(L), x);
   return cons(insertFun(L->next,x) L->val);
}
```

Insertion Sort

Avendo l'inserimento ordinato, possiamo facilmente scrivere l'algoritmo Insertion Sort: quali vantaggi ha rispetto il suo corrispettivo naturale sugli array?

```
lista insertSortIt(lista L){
   lista O = NULL;
   lista aux;
   while (L){
      aux = L->next;
      O = insertRec'(O, L);
      L = aux;
   }
   return O;
}
```

*Non volendo allocare memoria, occorre **inserire il nodo** invece che il valore*

Osservate che grazie ai parametri, non è necessario avere variabili di appoggio

```
lista insertSortRec(lista L){
   if (!L) return NULL;
   insertRec'(L, insertSortRec(L->next));
}
```

Un grande classico: merge

Vediamo l'implementazione C nella versione Rec:

```
lista merge(lista L, lista M){
   if (!M) return L;
   if (!L) return M;
   if (L->val <= M->val){
      L->next = merge(L->next, M);
      return L;
      }
   M->next = merge(L, M->next);
   return M;
}
```

Cosa notate rispetto all'analoga funzione sui vettori ordinati?

Sia da un punto di vista di efficienza che di programmazione?

Liste ordinate: efficienza

Altre funzioni binarie, come la differenza da **quadratiche diventano lineari**. Più precisamente dette m ed n le lunghezze delle due liste, si passa da una complessità $m \times n$ a $m+n$.

L'idea è sempre quella di **scorrere parallelamente** le due liste come nella funzione merge.

```
minus [] xs = []
minus xs [] = xs
minus (x:xs) (y:ys) |
   x < y    = x : minus xs (y:ys)
   x > y    = minus (x:xs) ys
 otherwise = minus xs (y:ys)
```

La complessità è $\mathcal{O}(m+n)$ perché ad ogni chiamata ricorsiva decresce la lunghezza o del primo o del secondo parametro.

Liste ordinate: efficienza

Vediamo l'implementazione **C** nella versione **Rec** che necessita dell'attenzione di deallocare i nodi quando necessario.

```
lista diffOrd(lista L, lista M){
   if (!M) return L;
   if (!L) return NULL;
   if (L->val == M->val){
         lista N = L->next;
         free(L);
         L = diffOrd(N, M);
   } else if (L->val < M->val)
         L->next = diffOrd(L->next, M);
   else diffOrd(L, M->next);
   return L;
}
```

Liste ordinate: efficienza

Le **liste ordinate** ovviamente possono rendere alcuni programmi **più efficienti**. Tutta una serie di funzioni come `remove`, `belongsTo`, etc. possono evitare di andare in fondo alla lista, rimanendo comunque lineari.

```
lista removeRecOrd(lista L, int x){
   lista M;
   if (!L || L->val > x) return L;
   if (x == L->val){
      M = L->next;
      free(L);
      return removeRecOrd(M, x);
   }
   L->next = removeRecOrd(L->next, x);
   return L;
}
```

```
lista belongsToOrd(lista L, int x){
   if (!L || L->val > x) return 0;
   if (x == L->val) return 1;
   return belongsToOrd(L->next);
}
```

Divisione di una lista in due

1.1 Divisione di Liste (21 settembre 2016)

Considerare il problema di dividere una lista di interi in due liste, una contenente tutti gli elementi che occorrono in posizione pari e una contenente tutti gli elementi in posizione dispari. Scrivere due funzioni in linguaggio C*:*

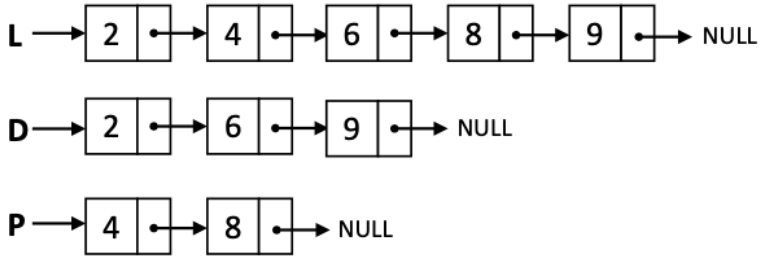
1. *La prima,* `void dividiFun(lista L, lista* D, lista *P)`, alloca *nuova memoria e copia gli elementi di posto dispari nella lista* `D` *e gli elementi di posto pari nella lista* `P`.

2. *La seconda,* `void dividi(lista L, lista* D, lista *P)` *modifica la lista originaria* `L` *modificando adeguatamente i puntatori restituendo nella variabile* `P` *un puntatore alla testa della lista degli elementi di posto pari e nella variabile* `D` *un puntatore alla testa della lista degli elementi di posto dispari.*

Esempio

ESEMPIO: *Entrambe le funzioni, se la lista di ingresso L fosse $\langle 2,4,6,8,9\rangle$, dovrebbero costruire le liste $D = \langle 2,6,9\rangle$ e $P = \langle 4,8\rangle$. Tuttavia, i loro effetti in memoria saranno molto diversi, come esemplificato in Fig. 1.*

a) risultato in memoria dopo l'esecuzione di `divideFun(L, &D, &P)`



b) risultato in memoria dopo l'esecuzione di `divide(L, &D, &P)`

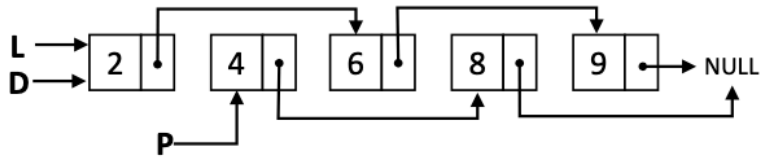


Figura 1: Effetti in memoria delle due funzioni.

Soluzione, versione Fun

Un'elegante soluzione (tra le tante) fa uso di **mutua ricorsione**: dividiFun sistema sulla lista dei dispari, mentre dividi2Fun sistema sulla lista dei pari e si chiamano l'un l'altra.

```
  /* prototipo */
void dividi2Fun(lista L, lista *D, lista* P);

void dividiFun(lista L, lista *D, lista* P){
   if (L){
      dividi2Fun(L->next, D, P);
      *D = cons(L->val, *D);
   } else {*D=NULL; *P=NULL;}
}

void dividi2Fun(lista L, lista *D, lista* P){
   if (L){
      dividiFun(L->next, D, P);
      *P = cons(L->val, *P);
   } else {*D=NULL; *P=NULL;}
}
```

Soluzione, versione Fun reloaded

In realtà `dividiFun` e `dividi2Fun` fanno esattamente la stessa cosa e **si limitano a scambiare i ruoli dei due paramentri, d P e D**. Tanto vale scrivere un'unica funzione 😉

Tenete tuttavia presente la possibilità della mutua ricorsione.

```
void dividiSwapFun(lista L, lista *D, lista* P){
   if (L){
      dividiSwapFun(L->next, P, D);
      *D = cons(L->val, *D);
   } else {*D=NULL; *P=NULL;}
}
```

Soluzione, versione Rec

Come quasi sempre, decisamente più impegnativa la versione che non alloca nuova memoria. Occorre stare molto attenti **a non cadere in un sentiero che si sta sbriciolando sotto ai piedi**.

Avendo la **fortuna** di scrivere una **funzione ricorsiva**, conviene fare quest'operazione **sulla via del ritorno**.

Si può partire dalle soluzioni precedenti, ma usiamo un'altra idea, **scansionare la lista a due a due**.

```
void dividInPlace (lista L, lista *D, lista* P){
   if (L && L->next){
      dividInPlace (L->next->next, D, P);
      L->next->next = *P;
      *P = L->next;
      L->next = *D;
      *D = L;
   } else {*D=L; *P=NULL;}
}
```

Devo avere 2 casi base

P e D vengono "attaccati" e poi assegnati a due pointer consecutivi

Funziona anche se **L==NULL**

Et voilà, MergeSort!

Visto che abbiamo parlato di Selection Sort, ma anche di merge, e di divisione di una lista in due metà, non resta che salutarci su una deliziosa funzioncina che fa **Merge Sort sulle liste**:

```
lista mergeSort(lista L){
   if (!L) return NULL;
   lista D, P;
   dividInPlace(L, &D, &P);
   return merge(mergesort(D), mergeSort(P));
}
```

Quali vantaggi ha questa funzione rispetto a quella che ordina array? E la complessità asintotica?

Lezione 24d:

Quello che in Haskell non si può fare

Determinare la palindromìa

Esercizio 2 (10 punti) Considerare il problema di verificare se una lista di caratteri è palindroma, cioè se letta da sinistra a destra o da destra a sinistra dà la stessa sequenza di caratteri (come le parole 'non', 'otto', 'radar', 'ingegni', 'onorarono', 'ossesso', ...).

1. (**3 punti**) Considerate il seguente programma (dove la funzione `eqList` verifica se due liste sono uguali e `reverse` restituisce il puntatore alla testa di una lista che contiene gli elementi di `L` in ordine rovesciato):

```
int palindroma(charList L){
    if (eqList(L, reverse(L)) return 1;
       else return 0;
   }
```

   Sotto quali ipotesi sulla funzione `reverse` la funzione dà risultati corretti?

Si intascavano comodamente **3** punti semplicemente dicendo che `reverse` **deve lasciare** `L` **immutata**. Cioè deve essere come reverseFun.

Assolutamente **imprevedibili i risultati** se L viene modificata.

Palindroma: idea semplice

2. (**6 punti**) Scrivere una funzione C *ricorsiva* `int palindroma(charList L)` che restituisce 1 solo se la lista L in ingresso è palindroma *senza* usare strutture dati ausiliarie.

Vediamo due soluzioni di questo punto. La prima alla portata di tutti: la vediamo prima in versione iterativa e poi ricorsiva.

Idea: determinare la lunghezza n della lista. Poi si scorre `L` (ricordando la posizione k del nodo corrente) confrontando il nodo corrente con quello in posizione $n - k$ (se si comincia a contare da 0). Si esce non appena si trovano due elementi diversi oppure quando k diventa maggiore o uguale di $n - k$.

Molti seguirono questa strada in modo più o meno convincente, spesso annidando cicli che scorrevano la lista…

Ma **la soluzione poteva essere resa semplice definendo una funzione** `int nth(lista, int)` che estrae l'ennesimo elemento da una lista… Vediamola.

Palindroma iterativa

```
int nth(lista L, int n){
/* PREC: length(L)<=n */
   if (!n) return L->val;
   return nth(L->next, n-1);
}

int palindromaIt(lista L){
   int n = length(L);
   lista lAux = L;
   while(n>0){
      if (lAux->val != nth(lAux, n))
         return 0;
      lAux = lAux->next;
      n = n-2;
   }
   return 1;
}
```

Bisogna calcolare il fatto che vado avanti su L!

Palindroma ricorsiva

```
int palindromAux(lista L, int n){
/* PREC: length(L)<=n */
   if (n<=0) return 1;
   if (L->val != nth(L, n)) return 0;
   return palindromAux(L->next, n-2);
}

int palindromaRec(lista L){
   return palindromAux(L, length(L));
}
```

Spero lo studente apprezzi la semplicità del codice finale. La perdita di qualche punto a causa dell'**irritazione del professore per abuso di annidamento di cicli** non è una punizione ingiusta! Le **soluzioni semplici permettono di risolvere problemi complessi**. Vedere soluzioni semplici, ovviamente è complicato.

Comunque, il programma scritto sopra è chiaramente **quadratico** nella lunghezza della lista. Si può fare meglio?

Palindroma ricorsiva efficiente

Occorre un'idea diabolica: la lista viene attraversata all'indietro al rientro delle chiamate ricorsive (idea già vista spesso): **rientrando posso avanzare sulla lista inseguendo i pointer `next` memorizzandoli in una variabile passata per indirizzo**.

```
int palindromAux(lista L, lista* lAux){
   if (!L) return 1;
   if (!palindromAux(L->next, lAux)
         || L->val != (*lAux)->val)
      return 0;
   *lAux = (*lAux)->next;
   return 1;
}

int palindromaRec(lista L){
   return palindromAux(L, &L);
}
```

I perfezionisti potevano evitare di fare controlli inutili dopo metà lista, aiutandosi con dei parametri interi (**Esercizio**).

Ultimo punto

3. (**1 punto**) Quale struttura dati ausiliaria viene in realtà allocata *implicitamente* durante l'esecuzione della vostra funzione?

Qui era sufficiente dire che **la versione ricorsiva alloca 'implictamente' uno stack**, in cui vengono memorizzati di fatto tutti i puntatori a tutti i nodi della lista.

Esercizio: scrivete una versione **iterativa lineare** che non usa la funzione `nth`, ma usa uno stack (o semplicemente una lista supplettiva usata come uno stack). A fine computazione, tale lista va deallocata (oppure deallocata mentre si estraggono gli elementi).

Lezione 24

That's all Folks!

Grazie per l'attenzione…

…Domande?