

## Esercizi di Bottigli

i) Scrivere una funzione iterativa che restituisce la somma pesata per posizione dei numeri della lista in posizione multipli di  $x$

int somma\_pesata (list L1, int x)

{ int risultato = 0; posizione = 1;

while (L1 != NULL)

{ if (posizione % x == 0)  
 risultato = risultato + (L1->d \* posizione);

posizione ++;

L1 = L1->next;

}

return risultato;

}

- Complessità in spazio: si tratta di una funzione iterativa, quindi il numero massimo di record di attivazione contemporaneamente presenti sullo stack è 1, e meglio, è costante e non è dipendente dalla dimensione dell'input: che la lista sia di 1, 10 o 100 nodi la funzione richiederebbe sempre lo stesso numero di record di attivazione. Dunque la complessità in spazio è costante  $O(1)$

- Complessità in tempo: per valutare la complessità in tempo dobbiamo stabilire il costo delle varie sezioni della funzione e calcolare quale prevale nella determinazione della complessità.

Inizialmente abbiamo dichiarazione ed assegnamento di variabili ed al termine uno return: sono istruzioni atomiche, il cui costo è costante. Il ciclo while è la porzione che determina la complessità: bisogna verificare quante volte viene iterato, da quali parametri in input ciò dipende e quanto costa una singola iterazione. Il ciclo è costituito da sole operazioni atomiche, che vengono eseguite indipendentemente dall'input

(la IF, che dipende dal valore di  $x$ , ma il caso peggiore, cioè che venga eseguito il sum then - il suo else non è presente - comporta comunque l'esecuzione di un'istruzione

ne atomico, quindi > costo costante). Quindi il costo di una singola iterazione è costante:  $O(1)$ .  
 Nel corpo del ciclo si scorre la lista un nodo alla volta (" $L1 = L1 \rightarrow next$ ;") fino alla fine della lista stessa (la condizione del ciclo è "while ( $L1 \neq NULL$ )": quindi viene ripetuto tante volte quant'è la lunghezza della lista  $L1$ ; l'altro pernmetto che la funzione riceve, int  $x$ , viene usato solo per stabilire se il valore del nodo corrente multiplo per la posizione debba o meno essere aggiunto alla somma persistente, comunque non incide sulla complessità.

La complessità in tempo quindi è data dal costo di una singola esecuzione per il numero di ripetizioni del ciclo, ossia  $O(1) \cdot O(\text{lunghezza}(L1))$ , cioè  $O(\text{lunghezza}(L1))$ .

2) Funzione che restituisce, dato una lista  $L1$  in input, la differenza fra la somma dei nodi nelle prime  $n$  posizioni e quelli da posizione  $n+1$  a posizione  $m$  (con  $m > n$ ) della lista

int somma\_e\_sottrai (list L1, int n, int m)

{ int risultato = 0, somma1 = 0, somma2 = 0, posizione = 1;

while ( $L1 \neq NULL \&\& posizione \leq n$ )

{

    somma1 = somma1 + L1  $\rightarrow$  d;

    posizione ++;

    L1 = L1  $\rightarrow$  next;

}

while ( $L1 \neq NULL \&\& posizione \geq m$ )

{

    somma2 = somma2 + L1  $\rightarrow$  d;

    posizione ++;

    L1 = L1  $\rightarrow$  next;

}

    risultato = somma1 - somma2;

return risultato;

}

• Complessità in spazio: iterativa, quindi costante  $O(1)$ ;

• Complessità in Tempo: il caso peggiore è che la lista debba essere

Scorsa, un node alla volta, fino alla fine (nel solo primo ciclo, nel caso in cui n sia uguale al numero dei nodi della lista; nel solo secondo ciclo, nel caso in cui n=0 e m = numero dei nodi della lista; in parte nel primo ed in parte nel secondo ciclo nel caso in cui m=numero dei nodi della lista, indipendentemente dal valore di n - che appunto comunque essere  $n \leq m$ ); i due cicli sono praticamente identici, quindi non importa in quale dei due la lista viene scorsa; inoltre il costo di una singola escursione del ciclo (come pure delle istruzioni si di fuori dai cicli, che sono atomiche) è costante. Il numero di ripetizioni del ciclo è comunque dipendente dalla lunghezza di L1, quindi la complessità in tempo è  $O(\text{lunghezza}(L1))$ .

3) Funzione che, data una lista L1 ed un numero x, restituisce in output il numero di volte in cui x compare tre volte nel filo

int conta-tre-di-filo (list L1, int x)

```

{
  int risultato = 0;
  if (L1 == NULL)
    if (L1->next != NULL)
      while (L1->next->next != NULL)
        {
          if (L1->d == x && L1->next->d == x && L1->next->next->d == x)
            risultato++;
          L1 = L1->next;
        }
  return risultato;
}
  
```

- Complessità in spazio: funzione iterativa,  $O(1)$

- Complessità in tempo: tutte istruzioni atomiche, costo costante; il ciclo si esegue finché si arriva al penultimo nodo, quindi fino a lunghezza di L1-2; dato però che la valutazione della complessità è asintotica, per la dimensione di L1 che tende ad infinito -2 è trascurabile, quindi la complessità in tempo è  $O(\text{lunghezza}(L1))$

4) Funzione che, dato una lista L1, un intero x ed una posizione n, inserisce in L1 una duplicazione di tutti i nodi contenuti x che sono in posizione multipla di n (in L1)

void duplica-nodo-conditionato (list<L1>, int x, int n)

{ int posizione = 1; list<L1> pi;

while (L1 != NULL)

{ if (posizione % n == 0 && L1->d == x)

{ p = newnode();

p->d = L1->d;

p->next = L1->next;

L1->next = p;

L1 = L1->next;

}

posizione++;

L1 = L1->next;

}

- complessità in spazio:  $O(1)$

- complessità in tempo:  $O(\text{lunghezza}(L1))$

Solti ragionamenti: il fatto di servirsi della funzione d'appoggio newnode non cambia nulla: la complessità in spazio è comunque costante (è vero che quando si richiede newnode ci sarà il suo record di istruzione sullo stack, ma tale numero di record è costante e non dipende dalla dimensione dell'input, quindi la complessità in spazio è comunque costante)

Il ciclo while, che contiene solo operazioni atomiche, viene ripetuto tante volte quanto è la lunghezza della lista originale: se si aggiunge un nodo viene subito fatto scorrere e non si incrementa il contatore, in modo che scorrimento e conteggio delle posizioni si riferiscono sempre alla lista originale. Prima della scorrere del ciclo viene fatto scorrere al nodo successivo e si incrementa il contatore di posizione: in questo modo si passa al nodo effettivamente successivo della lista originaria: in pratica se si aggiunge il nodo, si scorre due volte e si incrementa il contatore di posi-

Funzione di 1 (il nodo aggiunto non va contato, dato che il conteggio di posizione si riferisce alla lista originale); se non si inserisce il nodo, si scorre la lista di un solo nodo e si incrementa il contatore di posizione di 1. Quindi il ciclo while viene eseguito un numero di volte pari alla lunghezza di L1.

Il costo di una singola esecuzione è costante, indipendentemente dal fatto che le istruzioni del sono then dell' if vengano eseguite o meno (sono tutte istruzioni atomiche), quindi la complessità in tempo della funzione è pari a  $O(\text{lunghezza}(L1))$ .

5) Funzione che, data una lista L1, restituisce in output una nuova lista contenente la duplicazione dei nodi di L1 che sono somma dei loro due predecessori immediati.

```

list crea-lista-con-somma (list L1)
{
    list head = NULL, tail, p;
    :
    if (L1 != NULL)
        if (L1->next != NULL)
            while (L1->next->next != NULL)
            {
                if (L1->next->next->d = L1->d + L1->next->d)
                {
                    p = newnode();
                    p->d = L1->next->next->d;
                    p->next = NULL;
                    if (head == NULL)
                    {
                        head = p;
                        tail = p;
                    }
                    else
                    {
                        tail->next = p;
                        tail = p;
                    }
                    L1 = L1->next;
                }
            }
            return head;
}

```

- complessità in spazio: iterativa, quindi costante  $O(1)$
- complessità in tempo: istruzioni atomiche  $\geq 1$  di fuori del ciclo; istruzioni atomiche anche nel ciclo (quindi anche una singola esecuzione del ciclo ha costo costante); il ciclo viene ripetuto un numero di volte pari alla lunghezza della lista -2, ma dato che la valutazione è assintotica il -2 si ignora, quindi la complessità in tempo è  $O(\text{lunghezza (list)})$

- 6) Funzione che date due liste di interi L1 ed L2 e due interi n ed m,  $0 \leq n \leq m$ , restituisce in output una nuova lista costruita come segue:
- I) fino a posizione n sono duplicati i nodi di L1;
  - II) da posizione n+1 a posizione m escluso i nodi sono la somma dei nodi di L1 ed L2 in posizione corrispondente (se fra n ed m, se una lista diventa vuota, la nuova lista assume i valori della lista non vuota);
  - III) da posizione m in poi sono duplicati i nodi di L2

list lista-due-liste (list L1, list L2, int n, int m)

```
{ list head=NULL, tail, p;
```

```
int posizione = 1;
```

```
while (L1 != NULL && L2 != NULL)
```

```
{ if (posizione <= n)
```

```
    p = newnode();
```

```
    p->d = L1->d;
```

```
    p->next = NULL;
```

```
    if (head == NULL)
```

```
        { head = p; tail = p; }
```

```
    else
```

```
        { tail->next = p; tail = p; }
```

```
}
```

```
if (posizione > n && posizione < m)
```

```
{
```

```
    p = newnode();
```

```
    p->d = L1->d + L2->d;
```

```
    p->next = NULL;
```

```

if (head == NULL)
{ head = p; tail = p; }
else
{ tail->next = p; tail = p; }
}

if (posizione >= m)
{
    p = newnode();
    p->d = L2->d;
    p->next = NULL;
    if (head == NULL)
    { head = p; tail = p; }
    else
    { tail->next = p; tail = p; }
}

positione++;
L1 = L1->next; L2 = L2->next;
}

while (L1 != NULL && L2 == NULL)
{
    if (posizione < m),
    {
        p = newnode();
        p->d = L1->d;
        p->next = NULL;
        if (head == NULL)
        { head = p; tail = p; }
        else
        { tail->next = p; tail = p; }

        positione++;
        L1 = L1->next;
    }

    while (L1 == NULL && L2 != NULL)
    {
        if (posizione > n)
        {
            p = newnode();
            p->d = L2->d;
        }
    }
}

```

```

p->next = NULL;
if (head == NULL)
{ head = p; tail = p; }
else
{ tail->next = p; tail = p; }
}
positione +=;
L2 = L2->next;
}
return head;
}

```

- complessità in spazio: funzione iterativa, quindi  $O(1)$
- complessità in tempo: sia le istruzioni fuori dai cicli che quelle nei cicli sono atomiche, quindi  $\geq$  complessità costante; i cicli vengono iterati, finché è possibile scorrere un nodo alla volta le due liste in parallelo, quando una delle due obbliga a finire, quella con successivo da valutare fino al termine; quindi si ha un numero totale di iterazioni pari alla lunghezza della più lunga delle due liste, dato che è equivalente anche alla complessità in tempo:  $O(\max(\text{lunghezza } L1, \text{lunghezza } L2))$

- 7) Funzione che, date due liste di interi  $L1$  ed  $L2$  e un intero  $n > 0$  elimini dalla prima lista i nodi per cui la somma del contenuto di  $L1$  ed  $L2$  in posizione corrispondente (rispetto alle posizioni originali) sia multiplo di  $n$ . Se  $L2$  termina si consideri invece della somma solo il contenuto di  $L1$

```

list eliminate_multipli (list L1, list L2, int n)
{
    list head = L1, prev = NULL;
    while (L1 != NULL && L2 != NULL)
    {
        if (((L1->d + L2->d) % n == 0)
            if (prev == NULL)
            { head = L1->next; free(L1); L1 = head; L2 = L2->next; }
            else

```

```

{ prev->next = L1->next; free (L1); L1=prev->next; L2=L2->next; }

else
{ prev=L1; L1=L1->next; L2=L2->next; }
}

while (L1 != NULL && L2 == NULL)
{
    if (L1->d % n == 0)
        if (prev == NULL)
            { head = L1->next; free (L1); L1=head; }
        else
            { prev->next = L1->next; free (L1); L1=prev->next; }

    else
        { prev=L1; L1=L1->next; }
}
return head;
}

```

- complessità in spazio:  $O(1)$
- complessità in tempo:  $O(\text{lunghezza}(L1))$  si scorre solo fino alla fine di L1 un modo alla volta, quindi complessivamente si fanno tanti cicli quanto è la lunghezza di L1; tutte le istruzioni sono atomiche, quindi la complessità in tempo è pari al numero dei cicli.

8) Funzione che, data una lista di interi L1, calcoli e restituisca la lunghezza della massima sequenza di valori uguali

```

int valori_uguali (list L1)
{
    int seq=0, seqmax=0, lastval;
    while (L1 != NULL)
    {
        if (seq == 0)
            { lastval=L1->d; seq++; }

        else
            {
                if (L1->d == lastval)
                    seq++;
                else
                    {
                        lastval=L1->d;
                        seq=1;
                    }
            }
        if (seq > seqmax)
            seqmax=seq;
    }
    return seqmax;
}

```

```

if (seq > seqmax)
    seqmax = seq;
}
seq = 1;
}
L1 = L1->next;
}
return seqmax;
}

```

- complessità in spazio:  $O(1)$
- complessità in tempo:  $O(\text{lunghezza}(L1))$
- Non cambia nulla rispetto a casi precedenti: tutte istruzioni atomiche, cioè che viene ripetuto tante volte quanti sono i nodi della lista.