TPFI 2021/22
Hw 5: A taste of C

assegnato: 26 maggio 2022, consegna 9 giugno 2022

Esercizio 1 (ALTAMENTE COMPOSTI) Usare la logica del crivello di Eratostene (vedi *slide Lezione* **22**, pagina **20**) per scrivere una funzione (vedi riquadro) che *alloca* un vettore di k elementi (che chiameremo d) e lo restituisce in output caricato mettendo in $d[i]$ il numero di divisori (non necessariamente primi) di i (questa è la sequenza A000005 dell'*On-line Encyclopedia of Integer Sequecens*). La funzione inoltre carica in ac il primo indice del massimo del vettore, cioè il più grande numero altamente composto minore di k.

```
int* maxAltamenteComposto(int k, int* ac)
   /* PREC: k>0, POST: torna un vettore d lungo k
    * tale che d[i]=numero divisori di i>0 e carica in *ac
    * il maggiore numero altamente composto < k.
    */
```

ESEMPIO: Se k fosse 31, il vettore risultante sarebbe:

$$\#, 1, 2, 2, 3, 2, 4, 2, 4, 3, 4, 2, 6, 2, 4, 4, 5, 2, 6, 2, 6, 4, 4, 2, 8, 3, 4, 4, 6, 2, 8$$

e la funzione caricherebbe in `*ac` il numero 24 che è il primo numero con 8 divisori (# significa che non è rilevante il valore del vettore in posizione 0 ☺).

Esercizio 2 (SUCCESSIONE DI ULAM) Consideriamo la successione di Ulam (vedi Slides & Homework precedenti). Scrivere una funzione C di prototipo:

```
int ulam(int n)
   /* PREC: n>=0, POST: torna l'n-esimo numero di Ulam */
```

che preso in input un numero naturale n ritorna u_n. Ad esempio, se l'input fosse 4, la funzione torna 6. Se l'input fosse 11, la funzione torna 28.

OSSERVAZIONI: Potete seguire molte strade. Può essere utile allocare un vettore `u` con $n + 1$ posizioni (indicizzate da 0 a n), caricate i valori iniziali in `u[0]` e `u[1]` e calcolatevi iterativamente tutta la successione partendo da `u[2]` fino a `u[n]`. Per facilitare la ricerca del $k+1$-esimo elemento u_{k+1} (una volta noti $u_0, u_1, \ldots, u_k$) osservate che necessariamente si ha che $u_k + 1 \leq u_{k+1} \leq u_{k-1} + u_k$. Infatti, se ci fosse qualche numero minore di u_k che si scrive in modo unico come somma di due precedenti, sarebbe già stato inserito nella successione. Inoltre, essendo la successione strettamente crescente, è ovvio che $u_{k-1} + u_k$ si scrive in modo unico come somma di due precedenti (in quanto questa somma è strettamente maggiore di ogni altra somma di due precedenti) e ciò, tra l'altro, dimostra che la successione è infinita (e la ricerca di u_{k+1} termina sempre).

Esercizio 3(Ulam Reloaded) Riconsideriamo la successione di Ulam definita nell'Esercizio precedente. Stavolta dovete scrivere una funzione C di prototipo:

```
int nextU(listDCFirstLast U){
     /* PREC: U contiene ordinatamente
        i primi k>=2 elementi della successione u */
```

che, sotto la precondizione che `U` contenga *ordinatamente* i primi $k \geq 2$ elementi della successione u, calcola il $k+1$-esimo, lo restituisce come risultato e lo aggiunge in coda a `U`. Il tipo `listDCFirstLast` è il tipo *lista doppiamente concatenata* in cui c'è un descrittore della struttura dati contenente un puntatore al primo e all'ultimo elemento della lista. Inoltre, ogni nodo contiene un pointer al successivo e un pointer al precedente. Dare anche le definizioni di tipo.

Per esemplificare ulteriormente cosa devono fare le funzioni richieste, confido che risulti eloquente la Figura 1.

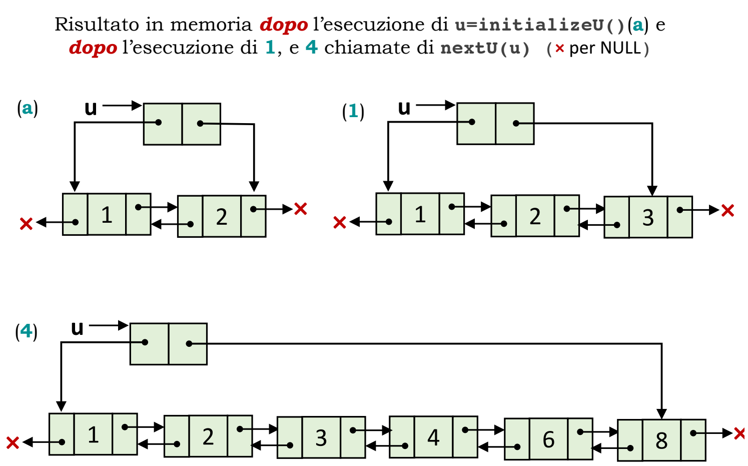

Figura 1: Esempio di esecuzione della funzione `nextU` (Esercizio **3**).

Esercizio 5 (QUELLO CHE IN HASKELL NON SI PUÒ FARE) Scrivere una funzione C che implementi il *crivello di Eulero*. Non è ovvio "saltare" in modo efficiente i numeri già cancellati per trarne vantaggio nelle successive cancellazioni. La soluzione che vi propongo di implementare consiste nell'usare un vettore di coppie di naturali, *succ* e *prec*, come una *lista doppiamente concatenata* in cui nella posizione i, se i non è stato cancellato, *succ* è il numero di posizioni che occorre saltare per andare al prossimo numero non cancellato, mentre *prec* è il numero di posizioni che occorre saltare (all'indietro) per andare al precedente numero non cancellato.

Definiamo un tipo `Pair` che è una coppia di interi `succ` e `prec` e definiamo un vettore di `Pair` (vedi file `eulero.h`). Questo vettore va inizializzato con tutti 1 (che significa appunto che tutti i numeri sono ancora potenziali primi). Quindi lo stato del vettore, inizialmente è il seguente (dove # significa 'non rilevante'):

pos	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
succ	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
prec	#	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Dopo aver cancellato i multipli di due, il vettore avrà i seguenti valori:

pos	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
succ	1	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#
prec	#	1	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#

Ora, partendo da 3 posso facilmente saltare sui numeri non cancellati. Moltiplicando questi per 3 ottengo quelli da cancellare in questa iterazione, e cioè 9, 15, 21, ottenendo la seguente situazione:

pos	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
succ	1	2	#	2	#	4	#	#	#	2	#	4	#	#	#	2	#	4	#	#	#	2	#
prec	#	1	#	2	#	2	#	#	#	4	#	2	#	#	#	4	#	2	#	#	#	4	#

Nell'esempio, a questo punto ho finito, perché il prossimo numero non cancellato è il 5 e $5^2 > 24$. Partendo da 2 e scorrendo il vettore usando i puntatori *succ* posso stampare tutti i numeri non cancellati che sono a questo punto necessariamente primi (vedi funzione `printPrimes` nel main fornito).

Voi dovete scrivere una funzione: `Pair* eulerSieve(int n);` che restituisce un vettore di coppie da cui sia possibile ricostruire tutti i numeri primi da 2 a n.

OSSERVAZIONI: I puntatori *prev* servono essenzialmente per effettuare in modo efficiente le operazioni di cancellazione. Le cancellazioni sono 'problematiche' perché sono operazioni distruttive sulla struttura dati e potrebbero, se fatte con poca cura, rendere inconsistente lo stato del vettore.

SPERIMENTAZIONI: Verificare che questo programma risulta effettivamente più efficiente del crivello di Eratostene. Ovviamente, il guadagno asintotico ($\ln \ln n$) è modesto e fa operazioni più complicate. Occorrerà provarlo per un qualche n sufficientemente grande (ordine di migliaia o milioni…).