Dynamic Programming

Sommelier

Paolo, per festeggiare il suo quarantesimo compleanno, si è iscritto a un corso per sommelier, dove impara a distinguere ed apprezzare le diverse tipologie di vini. Si è accorto però che, nonostante prenda solo un assaggio di ogni tipo di vino, per lui vale la regola fondamentale delle bevande alcoliche: quando le bevi, mai scendere di gradazione. Infatti, se per esempio Paolo assaggia un vino da 9 gradi e poi uno da 7, il giorno dopo si sveglierà con un grosso mal di testa indipendentemente dalle quantità.
Per fortuna, in ogni serata del corso è disponibile l'elenco dei vini che verranno portati uno dopo l'altro, e di ogni vino viene riportata la gradazione alcolica.

Non è ammesso mettere da parte un vino per berlo in seguito: ogni volta che gli viene passato un vino Paolo può decidere se assaggiarlo o meno, versandone un poco nel suo Tastevin.
Inoltre, dal momento che dopo aver assaggiato un vino Paolo deve pulire accuratamente il suo Tastevin con un panno, questa operazione in pratica gli impedisce di assaggiare due vini consecutivi (nell'immagine qui a fianco potete vedere il Tastevin).
Paolo desidera assaggiare il maggior numero di vini possibile.

Fonte: https://training.olinfo.it/task/sommelier/

Assunzioni

- $2 \leq N \leq 99$.
- I vini hanno una gradazione alcolica compresa tra 1 e 99

Esempi

Input	Output
9	4
11 13 10 16 12 12 13 11 13	

Input	Output
12	5
11 13 11 10 11 12 16 12 12 11 10 14	

Soluzione && Codice

```
int vini[MAX_VINI], memo[MAX_VINI]={0};
int N;

int bevi(int i, int last_gradaz){
    if(i>=N)
    //means that we'll take 0 alcool because there aren't anymore
    //so I take the maximum, which is zero
        return 0;

    int non_prendo = bevi(i + 1, last_gradaz);//in case I don't drink the current one
    //so last_gradaz doesn't change
    if(vini[i]<last_gradaz)//if I'm NOT able to drink, then...
        return non_prendo;//...return the fact that I can't drink

    int prendo = bevi(i + 2, vini[i]) + 1;//if here means that I can drink it, so i+2 for the constraits
    //and update last_gradaz with vini[i], which is the chosen one
    return max(prendo, non_prendo);//use max() because this is a maximization problem
}
```

Complessità?

```
//...raga...è devastante ma...O(2^N) !!!
```

costo finale : O(2^N)

Difficile fare peggio di così.

Ma come abbiamo dedotto tale complessità da brrrividi?
Abbiamo usato il Teorema delle Ricorrenze Lineari di Ordine Costante!!! ;3
Velia...aiuto che stai dicendo?
Guardate https://www.youtube.com/watch?v=GLmDmJBpFrc

Codice commentato

Il ***modus operandi*** è sempre lo stesso per qualsiasi problema chieda di ottimizzare il risultato, ovvero di MASSIMIZZARE o MINIMIZZARE:

```
fun(int indx, int memo[]/[], item altro)
    if (esco dal range) --> return numero
    int prendo = fun(i+1, memo, altro``)+qualcosa
    int NonPrendo = fun(i+1, memo, altro)
    memo[]/[]=max/min(prendo, NonPrendo)
    return memo[]/[]
```

Ma sto' **memo[]/[]**? Ch'è?

memo[]/[]

ovvero Memoization

(uso la notazione memo[]/[] per indicare che memo può essere un array [] oppure una matrice [][])

Letteralmente vuol dire **"appena calcolo una funzione ricorsiva, la memorizzo la soluzione da qualche parte, ed in caso dovessi ritornare alla stessa funzione, allora prima calcolarla vedo se avevo memorizzato il suo valore prima. In caso restituisco il valore precedentemente calcolato"**

Memoization

> **Ma quindi cosa è Dynamic Programming?**
> **E' usare la ricorsione + memoization**

Ovvero richiamare su se stessa la stessa funzione e salvo in un array/matrix i valori del risultato (parziale) che ottengo per evitare forse in futuro di ricalcolare.

Esempio di Memoization con Fibonacci

$$fibonacci(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & \text{altrimenti} \end{cases}$$

Ovvero se voglio calcolare fibonacci(5), considerando che noi siamo una macchiana e non sappiamo nulla, se sapessimo già quanto fa fibonacci(4) e fibonacci(3) allora sommerei tali valori, perché **per definizione fibonacci(di un num)=fibonacci(num prec)+fibonacci(num prec prec)**.

Ma noi come macchine non sappiamo nulla, quindi mi chiamo ricorsivamente fino ad un caso base.

Potete vedere che potevamo calcolare solo il "cammino" più sinistro. Ma abbiamo calcolato fib(3) due volte, fib(2) 3 volte, fib(1) 5 volte inutilmente. **Bastava calcolarlo una volta e salvarlo da una parte**.

Tale tecnica prende il nome di Memoization, **ovvero memorizzare ogni soluzione di una chiamata ricorsiva e prima di calcolare una nuova chiamata, controllare se tale nuova chiamata è stata già calcolata guardando nella nostra memoria**/array/matrice.
Quindi passare dal codice inefficiente di:

```
if(n==0 || n==1) return n;
else return fibonacci(n-1)+fibonacci(n-2);
//costo? --> O(2^N) ... TERRIFICANTE E DA INCUBOO, ma O(1) per la memoria
```

Ma come abbiamo dedotto tale complessità da brrrividi?
Abbiamo usato il Teorema delle Ricorrenze Lineari di Ordine Costante!!! ;3
Guardate https://www.youtube.com/watch?v=GLmDmJBpFrc

```
if(n==0 || n==1) return n;
if(memo[i] != empty_value) return memo[i];//di solito empty_value è -1 o 0
else
    memo[i] = fibonacci(n-1)+fibonacci(n-2);
    return memo[i]
```

Facendo così ci assicuriamo che ogni valore venga computato una singola volta al massimo, quindi **O(n) per tempo e memoria ED ERAVAMO PRIMA ad O(2^N) !!!!!!!!!!!!** Tutto aggiungendo una riga di codice, che però ciela una grande filosofia.

N.B.: "empty_value" è il valore con cui inizializziamo memo[]/[] e se il valore non è cambiato, vuol dire che non abbiamo già visto quel calcolo e quindi lo calcolo.

Poldo

Il dottore ordina a Poldo di seguire una dieta. Ad ogni pasto non può mai mangiare un panino che abbia un peso maggiore o uguale a quello appena mangiato. Quando Poldo passeggia per la via del suo paese, da ogni ristorante esce un cameriere proponendo il menù del giorno. Ciascun menù è composto da una serie di panini, che verranno serviti in un ordine ben definito, e dal peso di ciascun panino. Poldo, per non violare la regola della sua dieta, una volta scelto un menù, può decidere di mangiare o rifiutare un panino; se lo rifiuta il cameriere gli servirà il successivo e quello rifiutato non gli sarà più servito.

Si deve scrivere un programma che permetta a Poldo, leggendo un menù, di capire qual è il numero massimo di panini che può mangiare per quel menù senza violare la regola della sua dieta.
Riassumendo, Poldo può mangiare un panino se e solo se soddisfa una delle due condizioni:

- Il panino è il primo che mangia in un determinato pasto;
- Il panino non ha un peso maggiore o uguale all'ultimo panino che ha mangiato in un determinato
pasto

ATTENZIONE che NON siamo obbligatu a prendere sempre il primo

Fonte: https://training.olinfo.it/task/poldo

Assunzioni

- $1 \leq N \leq 10\,000$
- $0 \leq p < 10\,000$

Esempi

Input	Output
5	3
3	
6	
7	
5	
3	

Soluzione && Codice

```
int N;
uint16_t memo[MAXN][MAXN]={0};//using uint16 for reducing memory
//by using this matrix 10k*10k we will be immensively generous for memory
int panino[MAXN];

int D(int i, int last){
    if (i >= N)
    //if I'm here it means that i want to access (i) outside from the maximum range [0;N-1]
    //so what if we want/consider unexisting panini? How many can we eat? ... zero of course
        return 0;
```

Mmmm ma perché usiamo una matrice e non vettore?
Tecnicamente possiamo farlo con due vettori, ovvero una matrice 2xN. Ma perché non con solo un vettore lungo N? **Provateci**

```
    if (memo[i][last]==0){
    //if here, means that it is zero like it was declared
    //--> so that coords was never calculated, so calculate it now
        int non_prendo = D(i + 1, last);//if I don't take it then I pass again last without changing it

        if (panino[i] >= last) {
        //means that the current panino "i" isn't acceptable,
        //because it weigths more then the last one

            memo[i][last] = non_prendo;//before returning non_prendo, memorize it ---> MEMOIZATION POWER
            return non_prendo;//return the only possible choice
        } else {
            int prendo = D(i + 1, panino[i]) + 1;//increment of 1 because I've chosen to eat ONE panino
            memo[i][last] = max(prendo, non_prendo);//use max() because the problem asked us the MAXIMUM
            return memo[i][last];
        }
    }
```

```
     else
     //if I'm here, means that memo[][][] isn't 0, so it was previously calculated
     //so we've just to return instead of recalculating it
         return memo[i][last];
}
```

Complessità

Similmente o quasi identico a fibonacci, anche qui grazie alla potenza semplice della memoization calcoliamo ogni valore al massimo una volta e nel caso pessimo guarderemo tutti i valori della nostra matrice che è grossa NxN, ovvero N^2.
Quindi facilmente intuiamo O(N^2) perché dovrei, in caso sempre pessimo, riempire N^2 caselle.

Costo TEMPORALE AND SPAZIALE finale: O(N^2) con memoizatioin

Nonna

Gabriele, stanco del duro lavoro di consulenza a cui è stato sottoposto recentemente, può finalmente rilassarsi a pranzo dalla nonna, che è molto premurosa e gli prepara sempre un succulento pranzetto composto di N portate. L'unica pecca è che la nonna si lascia sovente andare un po' la mano, e le ultime volte Gabriele è poi tornato a casa tutto dolorante dai postumi di una brutta indigestione. Questa volta ha quindi deciso di prendere precauzioni, e pianificare con cura cosa mangiare e cosa no.

Dalle sue precedenti esperienze, è riuscito a stimare quanti grammi di cibo K deve mangiare al minimo affinché la nonna si senta soddisfatta e non si offenda dell'inappetenza del nipotino. Inoltre, appena arrivato in casa, è riuscito a sbirciare il menù scoprendo quale peso Pi ha ciascuna portata. Aiuta
Gabriele a trovare l'insieme di portate con peso totale minimo possibile ma almeno K!

Assunzioni

- $1 \leq N \leq 5000$.
- $1 \leq K \leq 5000$.
- $1 \leq Pi \leq 1\,000\,000$ per ogni $i = 0 \ldots N - 1$.
- È sempre possibile mangiare K grammi di cibo (il peso totale di tutte le portate è almeno K).

Input	Output
3 500	720
230 260 230	

Input	Output
8 600	600
140 160 180 220 20 100 40 80	

Soluzione

```
int memo[MAXN][MAXN]={0};
int pesi[MAXN];

int D(int i, int k){//"k" is how much we can still eat
    if (k <= 0)//standard check to see if our stomach/how much we can eat is at least positive
        return 0;//if here, it means that my stomach is full, so can't eat anymore, so to eat ZERO

    if (i >= N)//if out-of-range then we put a high value, which won't be consederend by min()
        return MAXP * 2;

    if(memo[i][k]!=0)//here we do magic with memoization
        return memo[i][k];//if here then it means that we previously calculated it
        //so to avoid compute it again

    if (memo[i][k]==0){//if never computed
        int non_prendo = D(i + 1, k);
        int prendo = D(i + 1, k - pesi[i]) + pesi[i];// !!! we do k-p[i] so to update our future "k"
        //means that if I've chosen it, then I eat it, then my "k" has to decrease

        memo[i][k] = min(prendo, non_prendo);
        return memo[i][k];
    }
    return memo[i][k];
}
```

Complessità

Ditecela voi ;)

Tip: è mooolto simile a Poldo