Pelstra di algoritmi

Marini Mattia

20 ottobre 2025

Indice

1 Introduzione **3**
- 1.1 Basi cpp . . . 3
  - 1.1.1 Input metodo 1 (consigliato) . . . 3
  - 1.1.2 Input metodo 2 . . . 3
  - 1.1.3 Ultra fast io . . . 4
- 1.2 Complessità . . . 5
  - 1.2.1 Esempio 1 . . . 5
  - 1.2.2 Esempio 2 . . . 6
- 1.3 Struttura problemi . . . 7

2 Programmazione dinamica **8**
- 2.1 Donimo . . . 8
- 2.2 Hateville . . . 8
- 2.3 Zaino . . . 9
- 2.4 Zaino umbound . . . 9
- 2.5 LCS . . . 9
- 2.6 Occorrenza k approssimata . . . 10
- 2.7 Prodotto di catena di matrici . . . 11
- 2.8 Intervalli pesati . . . 13

3 Esercizi dp **13**

4 Feature importanti di cpp **26**
- 4.1 Type inference - auto . . . 26
- 4.2 Passaggio per riferimento . . . 26
- 4.3 Range-based for (for-each loop) . . . 27
- 4.4 Overloading degli operatori . . . 27
  - 4.4.1 Opzioni per eseguire l'overloading . . . 27
  - 4.4.2 Funzioni da saper "overloadare" . . . 28
- 4.5 Structs . . . 28
- 4.6 Inizializzatori di liste . . . 29

5 Funzioni utili della std library **29**
5.1 Sorting - `std::sort` . . . 30
5.2 Pitov - `std::nth_element` . . . 30
5.3 Binary search 1 - `std::lower_bound`, `std::upper_bound` . . . 30
5.4 Binary search 2 - `std::binary_search` . . . 31
5.5 Range elementi uguali - `std::equal_range` . . . 31
5.6 Minimo e massimo in vettori - `std::min_element`, `std::max_element` . . . 31
5.7 Somma e moltiplica - `std::accumulate` . . . 32
5.8 Minimo e massimo - `std::minmax_element` . . . 32
5.9 Conta elementi - `std::count` . . . 32
5.10 Trova elementi - `std::find` . . . 32

6 Strutture dati di cpp **33**
6.1 Vector . . . 33
6.2 Stack . . . 33
6.3 Queue . . . 34
6.4 Priority Queue . . . 34
6.5 Set . . . 34
6.6 Map . . . 35
6.7 Unordered Map . . . 35

7 Problemi sito oii consigliati **36**

8 Soluzioni problemi sito OII **37**
8.1 Figonacci (`figonacci`) . . . 37
8.2 Discesa massima (`discesa`) . . . 37
8.3 Police 3 (`police3`) . . . 38
8.4 Piano degli studi (`pianostudi`) . . . 38
8.5 Police 4 (`police4`) . . . 39
8.6 Spiedini di frutta (`spiedini`) . . . 39
8.7 K-step ancestor (`treeancestor`) . . . 40
8.8 Taglialegna (`taglialegna`) . . . 40
8.9 Ma chi è quel mona (`porte`) . . . 46
8.10 Truffa al sushi (`sushi`) . . . 47
8.10.1 Idea di base del knapsack . . . 48
8.10.2 Knapsack e ricerca lineare . . . 48

1 Introduzione

Qui di seguito sono raccolte nozioni di base per affrontare ogni probrema relativo alle *OII*

1.1 Basi cpp

In ogni problema è necessario effettuare input/output su file[1]. Ci sono diversi modi per eseguire ciò.

1.1.1 Input metodo 1 (consigliato)

Vedi file input1.cpp

L'idea è di creare un oggeto `ifstream` e `ofstream` che poi potremmo utilizzare in maniera totalmente analoga a, rispettivamente, `cin` e `cout`

```
std::ifstream in("input.txt");
in >> a >> b;

std::ifstream out("output.txt");
out << a << b
```

Esiste un trucco per velocizzare notevolmente la velocità di input/output utilizzando questo metodo. In particolare, è sufficiente appendere le seguenti righe prima di scrivere o leggere su files:

```
ios_base::sync_with_stdio(false);
cin.tie(NULL);
```

Tuttavia se il problema sfora i limiti di tempo, con ogni probabilità è la soluzione a non essere corretta, non le operazioni di input/output. Queste righe possono essere utili per scalare la classifica sui siti di allenamento, non per altro

1.1.2 Input metodo 2

Vedi file input2.cpp

Questo metodo è più "vecchio" e meno consigliato. L'idea è di utilizzare le funzioni `freopen` per reindirizzare lo standard input/output su file:

```
FILE *in = fopen("input.txt", "r");
fscanf(in, "%d %d", &a, &b);

FILE *out = fopen("output.txt", "w");
fprintf(out, "%d %d\n", a, b);
```

dove le funzioni `fprintf` e `fscanf` prendono come argomenti:

- Il puntatore ad un file `FILE *`
- Una stringa `format`, contenente una serie di specificatori, preceduti da "`%`"

[1] In realtà a volte è sufficiente implementare il body di una funzione oppure la parte relativa all'output viene fornita

- `d`: decimal, numero intero
- `f`: float
- `s`: stringa c-style, in particolare `char *`

- Una serie variabili che corrispondono a quanto indicato in `format`. Nel caso di `scanf` è richiesto l'indirizzo di memoria di queset

1.1.3 Ultra fast io

Ci sono infine alcuni metodi per velocizzare l'input al massimo, utili per sprepere la perfomance al massimo, per arrivare nei primi in classifica. In particolare, questi metodi si basano sull'uso delle funzioni `getchat_unlocked()` e `putchar_unlocked()`

```
inline static int scanInt(FILE *file = stdin) {
  int n = 0;
  int neg = 1;
  char c = getc_unlocked(file);
  if (c == '-')
    neg = -1;
  while (c < '0' || c > '9') {
    c = getc_unlocked(file);
    if (c == '-')
      neg = -1;
  }
  while (c >= '0' && c <= '9') {
    n = (n << 3) + (n << 1) + c - '0';
    c = getc_unlocked(file);
  }
  return n * neg;
}

inline static void writeInt(int v, FILE *file = stdout) {
  static char buf[14];
  int p = 0;
  if (v == 0) {
    putc_unlocked('0', file);
    return;
  }
  if (v < 0) {
    putc_unlocked('-', file);
    v = -v;
  }
  while (v) {
    buf[p++] = v % 10;
    v /= 10;
  }
  while (p--) {
    putc_unlocked(buf[p] + '0', file);
  }
}
```

```
inline static int getString(char *buf, FILE *file = stdin) {
  std::string s;
  int c = getc_unlocked(file);

  // Skip leading whitespace
  while (c != EOF && (c == ' ' || c == '\n' || c == '\t' || c == '\r'))
    c = getc_unlocked(file);

  // Read until next whitespace or EOF
  int index = 0;
  while (c != EOF && c != ' ' && c != '\n' && c != '\t' && c != '\r') {
    buf[index++] = static_cast<char>(c);
    c = getc_unlocked(file);
  }

  return index;
}

inline static void putString(const std::string &s, FILE *file = stdout) {
  for (size_t i = 0; i < s.size(); i++)
    putc_unlocked(s[i], file);
}
```

Nota che le funzioni `putc_unlocked` e `getc_unlocked` sono disponibili solo in sistemi operativi unix(MacOs e Linux). Si possono usare in tranquillità dato che i server che testano il nostro codice sono tutti linux, ma il codice potrebbe non compilare in locale

1.2 Complessità

Il punto focale delle olimpiadi di informatica è non solo quello di scrivere algoritmi funzionanti, bensì efficienti. Per questa ragione è importante fornire critesi secondo i quali valutare la velocità d'esecuzione degli algoritmi

La logica di base sta nel relazionare il *numero di iterazioni* che un algoritmo deve eseguire alla *dimensione dell'input*.

1.2.1 Esempio 1

Supponiamo di avere un algoritmo per trovare il massimo in un vettore di n elementi. L'algoritmo fa quanto segue:

- Inizializza una variabile `max` al primo elemento del vettore
- Per ogni elemento del vettore controlla se è maggiore di `max`. In caso affermativo aggiorna `max` all'elemento corrente
- Ritorna `max`

Algoritmo: *Massimo vettore*

```
int max(int v[]):
    max ← v[0];
    for i = 0 to v.size − 1 do
        if v[i] > max then
            max ← v[i];
    return max;
```

In questo caso notiamo come siano necessarie n *iterazioni* perchè l'algoritmo termini (dove n è la dimensione del vettore v). Abbiamo quindi rapportato la dimensione dell'input alla complessità temporale dell'algoritmo

In questo caso, si dice che la complessità dell'algoritmo è $\Theta(n)$

1.2.2 Esempio 2

Supponiamo di avere un algorimo che debba eseguire una moltiplicazione applicando la proprietà distributiva:

$$(a+b+c)\cdot(d+e+f)$$

secondo la proprietà distributiva questo diventa:

$$\underbrace{(ad+ae+af)}_{A}+\underbrace{(bd+be+bf)}_{B}+\underbrace{(cd+ce+cf)}_{C}$$

ritornare un vettore che contenga i coefficienti (A, B, C)

Algoritmo: *Moltiplicazione distributiva*

```
int mul(int v1[], int v2[]):
    int rv = int[0...v1.size];
    for i = 0 to v1.size − 1 do
        rv[i] = 0;
        for j = 0 to v2.size − 1 do
            rv+ = v1[i] · v2[j];
    return rv;
```

Siccome per ogni elemento di v_1 devo scorrere interamente v_2, dovro ripetere $v_2 * v_1$ volte il body del ciclo.

In questo caso, se i due vettori hanno dimensione n, si dice che la complessità dell'algoritmo è $\Theta(n^2)$

1.2.2 Notazione Ω, Θ, O

In generale, per valutare la complessità di un algoritmo siamo interessati a più scenari:

- Nel peggiore dei casi, l'algoritmo che complessità ha? $\rightarrow$ notazione O

- Nel migliore dei casi, l'algoritmo che complessità ha?→ notazione Ω
- Nel "caso medio", l'algoritmo che complessità ha?→ notazione Θ

Nota bene: nella maggio parte dei casi siamo interessati alla coplessità nel caso pessimo O in quanto non possiamo escludere che questo si presenti nel dataset.

Per capire meglio la differenza fra caso ottimo e caso pessimo prendiamo in analisi l'algoritmo di *insertion sort*:

Algoritmo: *Insertion Sort*

```
int insertionSort(int v[]):
    for i = 1 to v.size − 1 do
        int key = v[i];
        int j = i − 1;
        while j ≥ 0 and v[j] > key do
            v[j + 1] = v[j];
            j = j − 1;
        v[j + 1] = key;
    return v;
```

In questo caso, dato un vettore lungo n, abbiamo due casi estremi:

- Il vettore è ordinato in modo crescente
- Il vettore è ordinato in modo decrescente

Nel primo caso l'algoritmo non entrerà mai nel ciclo while e dunque scorrerà il vettore una singola volta, originando una compessità di $\Omega(n)$.

Nel secondo caso l'algoritmo dovrà per ogni elemento del vettore scorrere (quasi) tutto il vettore stesso, originando una complessità di $O(n^2)$

1.3 Struttura problemi

Ogni problema delle OII e delle OIS ha una struttura simile e si compone come segue:

- Descrizione problema
- Descrizione dati di input
- Descrizione formato output
- Esempi
- Testcase

In particolare, il punteggio viene assegnato in base ai testcase che il nostro codice passa. Dobbiamo quindi scrivere un codice che risolva un dato problema stampando in output la soluzione. La correzione funziona come segue:

- I testcase sono raggruppati in un dato numero di *gruppi*
- Ad ogni gruppo di *testcase* è assegnato un punteggio e delle assunzioni. Ad esempio, ci può essere detto che i dati in input, in un dato gruppo non superano una certa dimensione o sono strutturati in un modo particolare
- Se all'interno di un gruppo i testcase sono tutti passati (output corretto), allora vengono assegnati i punti, altrimenti no

Si noti che per passare un testcase non è sufficiente che l'output sia corretto, ma il tempo di esecuzione e la memoria utilizzata devono essere entro i limiti previsti, specificati nel testo del problema

2 Programmazione dinamica

2.1 Donimo

Quanti modi ho di disporre tasselle di domino in una scacchiera $2 \times n$?

Soluzione

- Salvo in $dp[i]$ il numero di combinazioni che ci sono per un rettangolo $2 \times i$
- Ho due opzioni:
  - Metto 2 tessere in orizzontale, allora $dp[i] = dp[i-2]$
  - Metto 1 tessera in verticale, allora $dp[i] = dp[i-1]$
- Quindi $dp[i] = dp[i-1] + dp[i-2]$
- La soluzione è $\text{Fib}(n)$

2.2 Hateville

Ho un vettore di prezzi. Se prendo un prezzo $v[i]$ non posso prendere $v[i-1]$ e $v[i+1]$. Trova prezzo massimo

Soluzione

- Salvo in $dp[i]$ il prezzo massimo che posso ottenere con i vicini $\leq i$
- Ho due opzioni:
  - Non prendo $v[i]$, allora il prezzo è $dp[i-1]$
  - Prendo $v[i]$, allora il prezzo è $dp[i-1] + v[i]$

2.3 Zaino

Zaino ha capacità C, ho n pezzi di peso $w[i]$ e profitto $p[i]$. Trova profitto massimo

Soluzione

- Crea matrice $n \times C$ in cui si salva $dp[i][j]$ il profitto massimo che si può ottenere con i pezzi $\leq i$ e capacità $\leq j$
- Ho due opzioni:
  - Prendo pezzo (i, j), allora il prezzo migliore è $dp[i-1][j-w[i]] + p[i]$
  - Non lo prendo, allora il prezzo è $dp[i-1][j]$
- Posso ottimizzare lo spazio tenendo salvato solo due righe della matrice, la i e la $i-1$

2.4 Zaino umbound

Vedi zaino, solo che non c'è limite al numero di oggetti che uno puo prendere

Soluzione

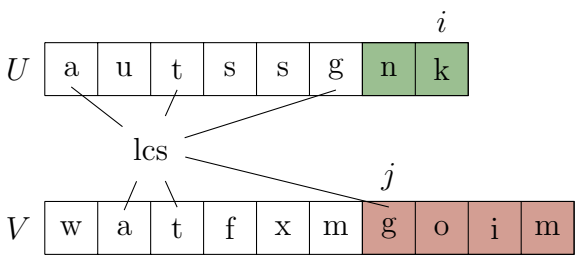
- Vettore dp in cui salvo in i il profitto massimo per uno zaino grande i
- Per ogni peso item x, il profitto massimo è $p[x] + dp[i - w[x]]$
- $dp[i]$ è il massimo fra tutti i valori trovati al punto 2

2.5 LCS

Date due stringhe U e T, trova la sottosequenza massimale. Una sottosequenza è una stringa che si ottiene da un'altra selezionandone solo alcuni caratteri (non necessariamente contigui, ma mantenendone l'ordine).

Soluzione

- Tabella dp con U su un lato e T sull'altro. In $dp[i][j]$ salvo la lunghezza della LCS fra la sottostringa $U[0, i]$ e $T[0, j]$
- Ho due opzioni:
  - $U[i] = T[j]$, allora $dp[i][j] = dp[i-1][j-1] + 1$ (aggiungo un carattere alla LCS più corta di 1)
  - $U[i] \neq T[j]$ allora $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$. Vedi immagine



Per migliorare la soluzione, se i caratteri sono diversi, devo aggiungere un carattere che sia nell'insieme dei caratteri dopo l'ultimo carattere comune. Quindi ho che

- A T, devo aggiungere un carattere che appartiene all'insieme rosso
- A U, devo aggiungere un carattere che appartiene all'insieme verde

Chiaramente la cosa è asimmetrica, per questo devo controllare $dp\,[i-1]\,[j]$ e $dp\,[i]\,[j-1]$

Dimostrazione formale : dobbiamo dimostrare che date due parole $U\,(u_1,\ldots,u_i)$ e $V\,(v_1,\ldots v_j)$ e $X\,(x_1,\ldots x_k)$ allora

- Se $u_i = v_j$ allora

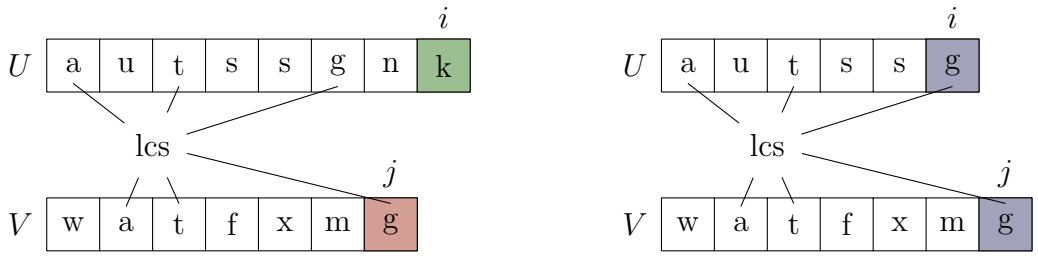
$$u_i = v_j = x_k$$
$$X\,(K-1) \in \mathcal{LCS}\,(U\,(i-1)\,, V\,(j-1))$$

- Se $u_i \neq v_j$ e $x_k \neq u_i$ allora

$$X \in \mathcal{LCS}\,(U\,(i-1)\,, V)$$

- Se $u_i \neq v_j$ e $x_k \neq v_j$ allora

$$X \in \mathcal{LCS}\,(U, V\,(j-1))$$



2.6 Occorrenza k approssimata

Data una stringa t e una p, diciamo che la distanza k di p da t è il numero minimo di *inserimenti, eliminazioni e scambi* che dobbiamo fare in t per far si che $t == p$.

$$t = \text{ "scempio" }, \quad p = \text{ esempio } \rightarrow k = 2$$

ad esempio, scambiando la "s" e "c" di *scempio* in "e" ed "s" rispettivamente

Il problma sta nel trovare in un testo t, la distanza minima di un pattern p da una sua qualsiasi sottostringa.

Ciò equivale a trovare quanti inserimenti, rimozioni e scambi devo fare nel testo per far si che il pattern diventi una sua sottostringa

Soluzione

- Inizializza matrice che ha p in verticale e t in orizzontale

- In $dp\,[i]\,[j]$ si salva *il minor valore di k per far si che* $p\,[0, i]$ *sia sottostringa di* $t\,[0, j]$ *che finisca in* j
- Se $p\,[i] == t\,[j]$ allora non serviranno altre mosse per riportare la soluzione di $dp\,[i-1]\,[j-1]$ alla soluzione corrente
- Se $p\,[i] \neq t\,[j]$ allora posso fare 3 cose:

	a	b	a	b	a	g
b						
a						
b		→	?			

+1
Fai coincidere bab con ab e poi elimina a

	a	b	a	b	a	g
b						
a			↓			
b			?			

+1
Fai coincidere ba con aba e poi aggiungi b

	a	b	a	b	a	g
b						
a		↘				
b			?			

+1
Fai coincidere ba con ab e poi cambia a in b

La soluzione migliore è data dal minimo valore nell'ultima riga della tabella

Nota che la prima riga e la prima colonna vanno riempite rispettivamente con $[0, \ldots, 0]$ e $[1, 2, \ldots, n-1, n]$. Questo ha senso in quanto:

- Per far si che il pattern vuoto sia sottostringa di t non serve alsona mossa ($[0, \ldots, 0]$)
- Per far sic che un pattern di lunghezza k sia sottostringa del testo vuoto è necessario aggiungere i k caratteri del pattern ($[1, 2, \ldots, n-1, n]$)

2.7 Prodotto di catena di matrici

Si vuole fare il prodotto matriciale tra $[A_1, A_2, \ldots, A_{n-1}, A_n]$. Il prodotto matriciale gode di proprietà associativa. Si trovi la parentizzazione che riduce al minimo il numero di moltiplicazioni scalari totali da compiere

Ad esempio, avendo $[A, B, C, D]$, posso parentizzare come segue:

$$[(A \cdot B) \cdot (C \cdot D)]\,, \quad [A \cdot (B \cdot C) \cdot D]\,, \quad [A \cdot (B \cdot (C \cdot D))]$$

e cosi via. Questo funziona in quanto per moltiplicare delle matrici bisogna assicurarsi che queste siano compatibili. Il numero di colonne della prima deve essere uguale al numero di righe della seconda. Ad esempio, indicando con [righe, colonne] una matrice, una serie che può essere moltiplicata è la seguente:

$$[4, 5] \cdot [5, 2] \cdot [2, 10] \cdot [10, 7] \rightarrow [4, 5, 2, 10, 7]$$

Nota che la dimensione di ogni matrice può essere salvata in un vettore c in cui c_i contiene il numero di colonne della matrice i, che corrisponde al numero di righe della matrice $i+1$. Quindi il numero di moltiplicazioni necessarie per eseguire $A_i \times A_j$ sarà:

$$c_i \cdot (\cdot c_{i-1} \cdot c_j)$$

- c_i: numero di moltiplicazioni per calcolare una cella

- $(\cdot c_{i-1} \cdot c_j)$: dimensione della matrice risultante

Soluzione

- Creo matrice `dp` come seguen:

	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

- In `dp[i][j]` salvo il minor numero di moltiplicazioni necessarie per moltiplicare le matrici fra `i` e `j`
- Costruisco matrice scorrento in diagonale a partire dalla diagonale più vicina alla diagonale principale. Il numero minore è dato dal numero minore date due parentizzazioni, ad esempio se ho

$$[A_3, A_4, A_5, A_6]$$

  dovro tentare con

$$[(A_3) \cdot (A_4, A_5, A_6)], \quad [(A_3, A_4) \cdot (A_5, A_6)], \quad [(A_3, A_4, A_5) \cdot (A_6)]$$

- Il risultato finale si trova in `dp[1][n]`, dove `n` è il numero di matrici
- Per ricostruire la parentizzazione, posso salvarmi in una tabella `last[i][j]` l'indice a cui ho "spezzato la parentizzazione". Poi posso ricostruirla ricorsivamente come segue:

Algoritmo 1: *Find minimum parenthesization*

```
printPar(int last[][], int i, int j):
    if i == j then
        print "A"; print i; print "]";
    else
        print "(";
        printPar(last, i, last[i][j]);
        print ".";
        printPar(last, last[i][j] + 1, j);
        print ")";
```

Algorithm 1: Print optimal parenthesization

2.8 Intervalli pesati

Vengono dati n intervalli aperti $[a_1, b_1[\,, [a_2, b_2[\,, \ldots [a_n, b_n[$. Ogni intervalli ha un valore w_i. Trovare il valore massimo che si può ottenere selezionando intervalli non sovrapposti.

Soluzione

- Ordina intervalli per tempo di fine
- Definisco la funzione `pred(i)`, che ritorna il *predecessore* di un intervallo, ossia il primo intervallo che ha tempo di fine minore del tempo di inizio di i
- Creo vettore `dp` che salva in `i` il valore massimo ottenibile con gli intervalli fino ad `i` compreso
- Itero su intervalli. Per ciascun intervallo `i` posso:
  - Selezionarlo: in questo il valore massimo ottenibile è dato da `dp[pred(i)] + w_i` a
  - Non selezionarlo: in questo caso il valore massimo è uguale al precedente `dp[i-1]`

Complessità: $O(n \log n)$

3 Esercizi dp

Esercizio 1: *Sottosequenza massima (Kadane's problem) (link)*

Dato in input un vettore `v`, contenente interi (anche negativi), si trovi la [a]sottosequenza che abbia somma degli elementi massima. Si ritorni quest'ultima

Input

La dimensione n del vettore e sulla nuova riga gli elementi del vettore separati da uno spazio

Output

La somma degli elementi della sottosequenza con somma massima

Input	Output	Discussione
5 1 2 3 4 5	15	La sottosequenza è data dall'intero vettore
8 -2 -3 4 -1 -2 1 5 -3	7	La sottosequenza è data dall'intervallo $[2, 6]$
4 -2 -3 -1 -11	0	Si assuma che la sottosequenza nulla abbia somma 0

Complessità ottimale: $O(v.size)$

[a]Una sottosequenza è un insieme di elementi adiacenti all'interno del vettore

Un approccio naif sarebbe quello di generare tutte le sottosequenze possibili e confrontarne la somma, stampando quella massima. Questo approccio tuttavia sarebbe davvero inefficiente, tuttavia è molto semplice da implementare:

```
public static int subsequenceIneff(int v[]) {

int max = 0;

for (int i = 0; i < v.length; i++) {

  int currSum = 0;
  for (int j = i; j >= 0; j--) {
    currSum += v[j];
    if (currSum > max)
      max = currSum;
  }

}

return max;

}
```

Complessità: $O\left(n^2\right)$

Un approccio più efficiente può essere implementato tramite programmazione dinamica. L'idea è la seguente:

- Salvo la sottosequenza con somma maggiore che finisce in posizione i-esima
- Calcolo la sottosequenza con somma maggiore che finisce in pos $i+1$ utilizzando la sottosequenza con somma maggiore che finisce in pos i. Chiamiamo questo vettore `dp`

Immaginiamo di salvarci i risultati intermedi in un vettore: questo vettore avrà dimensione n e conterrà in posizione i il sottovettore di somma massima che finisce in posizione i. Notiamo innanzitutto che calcolare `dp[0]` è scontato:

- Se `v[0] > 0` allora il sottovettore è costituito da un singolo elemento, ovvero `v[0]`
- Se `v[0] <= 0` allora il sottovettore è il sottovettore nullo, il quale ha sempre somma 0

Per calcolare invece `dp[i]`, ragiono nel seguente modo:

- Se `dp[i-1] + v[i] > 0` allora `dp[i] = dp[i-1] + v[i]` (mi conviene prendere la miglior sottosequenza che termina nella posizione prima e sommarci `v[i]`, anche se questo è negativo)
- Se `dp[i-1] + v[i] <=0` allora `dp[i] = 0` (conviene "ripartire a formare il vettore", dato che concatenando la subsequence con somma maggiore che termina in $i-1$ aggiungerei solo una quantità negativa)

Esercizio 2: *Cuttinig rod*

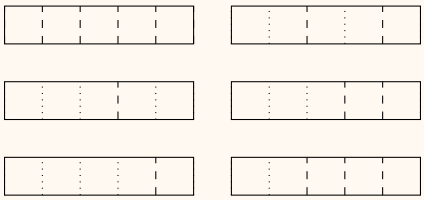
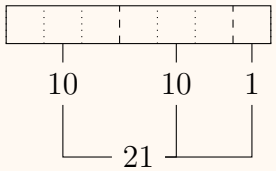
Dato un cilindro di lunghezza `n` e un vettore `v` di dimensione `n`, che in `v[i]` contenga il prezzo di un cilindro lungo `i+1`, si stampi il prezzo massimo che posso ottenere tagliando il cilindro in quante parti voglio

Input

La dimensione n(la lunghezza del cilindro) e sulla nuova riga gli n interi positivi che costituiscono gli elementi di v(ossia i prezzi di ogni taglio di cilindro)

Output

Il prezzo massimo che posso ottenere suddividendo il cilindro

Input	Output	Discussione
5 1 2 3 4 5	5	Posso tagliare il cilindro in molti modi per ottenere il valore 5: 
7 1 4 10 8 5 10 13	21	In questo caso ciò che conviene fare è spezzare il cilindro in 2 pezzi di lunghezza 3 e 1 di lunghezza 1: 

Complessità ottimale: $O(n \cdot v.size())$

L'idea di base per risolvere il problema è la seguente:

- Creo un vettore `dp` all'interno del quale salvo nella *i-esima* cella il valore massimo che posso ottenere suddividendo un cilindro di lunghezza $i+1$
- Costruisco il vettore `dp` partendo dal caso base: `dp[0] = prezzi [0]`, in quanto ho un solo modo di suddividere un cilindro lungo 1
- Contanto sul fatto che il vettore `dp` contenga il il prezzo maggiore che posso ottenere suddividendo il cilindro in un dato modo, calcolo `dp[i+1]` sfruttando i dati contenuti nelle celle precedenti del vettore `dp`.In particolare, supponendo di dover calcolare la posizione i del vettore `dp`, devo:

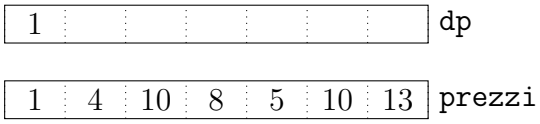
- Calcolare il prezzo che otterrei mettendo in posizione i un pezzo di ogni lunghezza, da 1 a i
- Confrontare i prezzi ottenuti
- `dp[i]` è il valore massimo fra questi prezzi

Vediamo un esempio grafico. Supponiamo di avere in input il vettore `prezzi`, con valori:

1 4 10 8 5 10 13

Ripercorriamo gli step appena descritti.

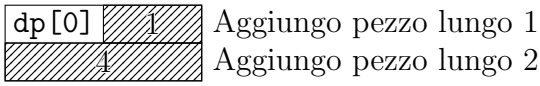
- Creo un vettore `dp` all'interno del quale salvo nella *i-esima* cella il valore massimo che posso ottenere suddividendo un cilindro di lunghezza $i + 1$
- Costruisco il vettore `dp` partendo dal caso base: `dp[0] = prezzi [0]`, in quanto ho un solo modo di suddividere un cilindro lungo 1. Quindi nel nostro caso:



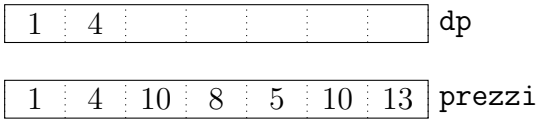
  Chiaramente, un cilindro id lunghezza 1 non può essere suddiviso, quindi l'unico prezzo possibile è il prezzo del cilindro lungo 1, ossia 1 nel nostro caso

- Contanto sul fatto che il vettore `dp` contenga il il prezzo maggiore che posso ottenere suddividendo il cilindro in un dato modo, calcolo `dp[i+1]` sfruttando i dati contenuti nelle celle precedenti del vettore `dp`.In particolare, supponendo di dover calcolare la posizione i del vettore `dp`, devo:
  - Calcolare il prezzo che otterrei mettendo in posizione i un pezzo di ogni lunghezza, da 1 a i
  - Confrontare i prezzi ottenuti
  - `dp[i]` è il valore massimo fra questi prezzi

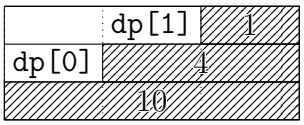
  Iniziamo quindi calcolando `v[1]`. So di avere già calcolato il prezzo migliore per suddividere un cilindro di lunghezza 1. Quindi per arrivare ad avere un cilindro di lunghezza 2 ho due alternative:

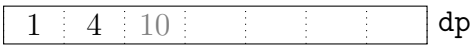


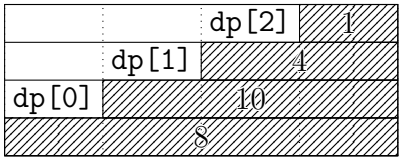
  Ora devo decidere se mi convenga prendere un solo cilindro da 2 oppure suddividerlo in due pezzi da 1. Mi basta però vedere quale delle opzioni ha prezzo maggiore. In questo caso conviene prendere un pezzo da 2 con costo 4. Dp diventa:

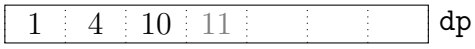


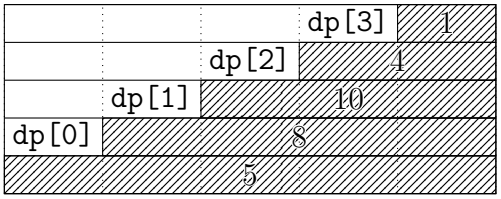
Ripetiamo il passaggio per $i = 3$

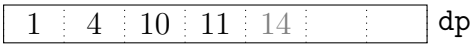


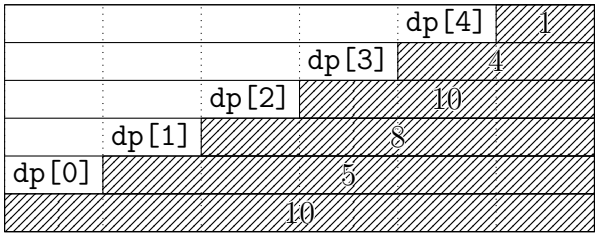


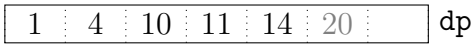


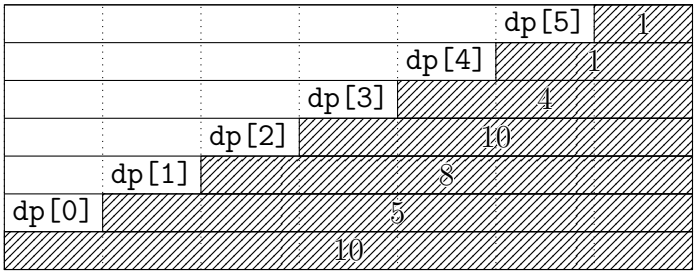


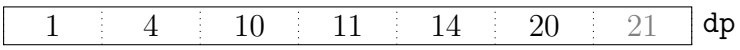












Esercizio 3: *Somma e media (link)*

Dati in input un numero N, e successivamente N interi, calcolare la media aritmetica e la somma di questi ultimi

Input:

Sulla prima riga l'intero N, sulla seconda riga N interi separati da uno spazio

Output:

Due interi: rispettivamente la somma degli N numeri e la loro media aritmetica

Input	Output	Discussione
1 12	12 12	Somma e media coincidono e hanno valore 12
7 1 2 34 -56 33 23 89	126 18	Somma e media coincidono e hanno valore 12

Complessità ottimale: $O(N)$

Esercizio 4: *Majority element (link)*

Dato un array *nums* di dimentione n, ritornare il *majority element*. Il *majority element* è l'elemento che appare di più di $\frac{n}{2}$ volte. Si può assumere che l'elemento esista sempre nell'array

Input:

Sulla prima riga l'intero n, sulla seconda riga n, ossia gli elementi di *nums*

Output:

Un intero, il majority element

Input	Output	Discussione
3 3 2 3	3	3 appare più di $3/2 = 1$ volta
7 2 2 1 1 1 2 2	2	2 appare più di $7/2 = 3$ volte

Complessità ottimale: $O(n)$

Esercizio 5: *Longest Common Subsequence (link)*

Date in input due stringhe *S1* ed *S2*, si ritorni la lunghezza della *longhest common subsequence*, ossia della [a]sottosequenza più lunga comunque ad esntrambe le stringhe.

Input:

Due stringhe, una per riga, composte da caratteri maiuscoli compresi fra A e Z

Output:

Un intero, la lunghezza della più lunga sottosequenza comune ad entrambe le stringhe

Input	Output	Discussione
AGGTAB GXTXAYB	4	La sottosequenza comune con lunghezza maggiore è *"GTAB"*, ed ha lunghezza pari a 4
AABBCCD AABBD	5	La sottosequenza comune con lunghezza maggiore è *"AABBD", ed ha lunghezza pari a 5*

Complessità ottimale: $O\left(s_1.lenght \cdot s_2.lenght\right)$

[a]Con sottoseuquenza si intende la una stringa che si può ottenere da un'altra eliminando determinati caratteri: *bedbreakfast* è una sottosequenza di *bedandbreakfast*. A differenza di ciò che accade in un sottovettore, i caratteri non devono essere necessariamente contigui: *bedbreakfast* è sottosequenza ma non sottovettore; *breakfast* è sottosequenza e sottovettore

Per risolvere questo problema dobbiamo utilizzare una matrice di supporto, che salverà i valori intermedi e ci permetterà di utilizzare la programmazione dinamica. Prendiamo come esempio il le stringhe *"AGGTAB"* e *"GXTXAYB"*. La matrice di supporto deve avere la seguente forma:

		G	X	T	X	A	Y	B	s1, j
	0	0	0	0	0	0	0	0	
A	0								
G	0								
G	0								
T	0								
A	0								
B	0								
s2, i									

Quindi la struttura della matrice è la seguente:

- Un orentamento rappresenta una stringa, l'altro l'altra (nota che le stringhe non sono salvate nella matrice, sono riportate in figura solo per rendere il procedimento più chiaro)
- La prima colonna e la prima riga sono riempite di zeri. Questo serve perché ci permette di evitare di incappare in indici negativi quando eseguiremo l'algoritmo

- Siano `s1` e `s2` le stringhe, nella cella di indice (i, j), salveremo la lunghezza della longest common subsequence per `s1.substring(0,i)` e `s2.substring(0,j)`

		G	X	T	X	A	Y	B	s1, j
	0	0	0	0	0	0	0	0	
A	0								
G	0						(1,6)		
G	0								
T	0								
A	0								
B	0								
s2, i									

Ad esempio, nella cella evidenziata, con indice $(1, 6)$, va salvata la lunghezza della LCS delle stringhe "GXTXAY" e "AG"

Detto questo, possiamo riempire la tabella secondo i seguenti criteri:

- Se `s1[i-1] == s2[j-1]` ciò significa che la soluzione ottimale per quel sottoproblema è data dalla soluzione ottimale per il sottoproblema con le medesime stringhe senza però questultimo carattere. La soluzione di questo problema si trova nella cella $(i - 2, j - 2)$
- Se `s1[i-1] != s2[j-1]` allora non ho modo di migliorare la lunghezza della LCS aggiungendo un elemento alle sottostringhe dei problemi precedenti. La soluzione ottimale è quindi da calcolare confrontando le soluzioni ottimali precedenti, in particolare sia `dp` la matrice:

  ```
  dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1])
  ```

  Volendo ad esempio calcolare il problema per le sottostringhe "GXT" e "AGGTA" posso partire dalle soluzioni dei sottoproblemi per le stringhe "GX", "AGGTA" e "GXT", "AGGT". Mi basta prendere la maggiore di queste due.

  Nota anche che non mi serve controllare la soluzione del sottoproblema "GX", "AGGT" in quanto colonne e righe sono tutte ordinate in maniera crescente, quindi è impossibile che la cella $(i - 1, j - 1)$ abbia un valore maggiore della cella $(i, j - 1)$ o $(i - 1, j)$

Esercizio 6: *Minimum coin (link)*

Vengono dati in input un array contenente un array `coins` (il quale rappresenta monete di diverso taglio) e un numero intero `ammount` (il quale rappresenta il quantitativo totale di monete). Calcolare il numero minimo di monete che si possono utilizzare per arrivare alla somma `ammount` . Si assuma di avere un numero infinito di monete per ogni taglio.

Input:

Due righe. Sulla prima gli interi `ammount` e `n` (la dimensione di `coins`), mentre sulla seconda gli elementi di `coins` separati da uno spazio. Nota che gli elementi di `coins` non vengono necessariamente dati in ordine crescente

Output:

Un intero, il numero minimo necessario di monete per arrivare ad `ammount`. Se la combinazione non fosse presente ritornare -1

Input	Output	Discussione
11 3 1 2 5	3	Per ottenere la somma 11 posso utilizzare 2 monete da 5 e 1 da 1
4 1 3	-1	Non è possibile ottenere una somma di 4 con sole monete da 3

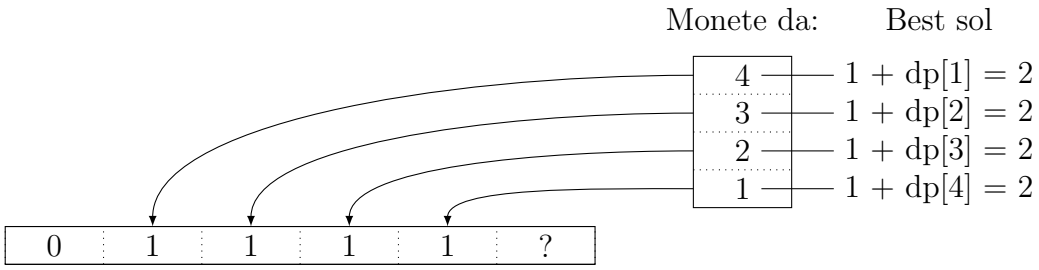
Complessità ottimale: $O(coins.size \cdot amount)$

Approccio intuitivo (sbagliato): cerco di riempire la somma con monete quanto più grandi possibile. Ad esempio con questo input, tuttavia, non funziona: somma: 20, monete: 15 13 7 1

L'approccio corretto è molto simile al problema *cutting rod*. Di fatto è come se dovessimo "riempire" un cilindro lungo `ammount` con pezzi di dimensioni contenute in `coins`.

Procediamo così:

- Creo vettore `dp` di dimensione `ammount`, all'interno del quale salvo nella cella i il numero minore di monete per creare una somma pari ad i
- `dp[0]=0`, ossia ho modo di creare una somma pari a zero con zero monete
- Per calcolare la cella i-esima del vettore `dp`, devo ragionare nel seuguente modo:
  - Per ogni moneta che abbbia valore inferiore a i, calcoliamo il minor numero di monete che possiamo usare utilizzando il vettore `dp`, in analogia con il problema *cutting rod*. Supponiamo di avere `amount=5, coins=[1, 2, 3, 4]`



La miglior souzione per una somma pari a 5 è quindi 2. Possiamo usare 2 monete in modi diversi ((3,2), (4,1)) per ottenere la somma 5

- Mettendo in ordine il concetto intuitivo dobbiamo creare `dp[i]` mettendo nel seguente modo:
  - Per ogni elemento di `coins` che sia minore di `i` calcolo il numero minimo di monete che è necessario per arrivare ad una somma di `i` utilizzando `dp`. Supponendo di dover includere una moneta di valore `value`, allora il minor numero di monete per arrivare a `i` è dato da

    ```
    dp[i-value] + 1
    ```

  - Il valore minimo di monete per arrivare alla somma `i` è il valore minimo fra tutti quelli calcolari al punto precedente
  - Occhio ai casi nei quali non è possibile ottenere una somma specifica tramite le monete a disposizione. In questi casi metteremo il valore -1 nel vettore `dp`

Esercizio 7: *Unique paths (link)*

Un robot si muove su di una griglia `n x m`. Il robot inizialmente è posizionato sulla cella $[0]\,[0]$ e deve arrivare alla cella $[m-1]\,[n-1]$. Il robot può muoversi solamente verso il basso e verso destra. Dati due interi `n, m` che indicano la dimensione della griglia, calcolare il numero di percosi possibili

Input

Gli interi `m` e `n`

Output

Il numero di percorsi possibili

Input	Output	Discussione
2 2	2	I percorsi possibili sono $[(R \to D), (D \to R)]$
3 2	3	I percorsi possibili sono $(R \to D \to D)$, $(D \to D \to R)$, $(D \to R \to D)]$

Complessità ottimale: $O(n \cdot m)$

L'idea di base è la seguente:

- Creo matrice `dp` che salva nella generica cella $[i]\,[j]$ il numero di percorsi tramite i quali posso arrivare in $[i]\,[j]$
- Inizializzo la prima colonna e la prima riga della matrice a 1: per raggiungere le celle della prima riga e colonna ho un solo modo, ossia rispettivamente spostarmi a destra o spostarmi in basso (non posso tornare indietro)
- Per ogni cella che avanza calcolo il valore come la somma della cella a sinistra e della cella a sopra: questo perche per arrivare nella cella $[i]\,[j]$ posso passare per la cella $[i-1]\,[j]$ oppure per la cella $[i]\,[j-1]$. La somma dei modi che ho per arrivare nelle suddette celle è il numero di modi che ho per arrivare nella cella corrente
- Una volta generata l'intera tabella, nella ultima cella in basso a destra avro il risultato al problema

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & / & / & / & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & / & / & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & / & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & / & / & / & / & / \\ 1 & 3 & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & / & / & / & / \\ 1 & 3 & 6 & / & / & / & / \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & / & / & / \\ 1 & 3 & 6 & 10 & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & / & / \\ 1 & 3 & 6 & 10 & 15 & / & / \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & / \\ 1 & 3 & 6 & 10 & 15 & 21 & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 3 & 6 & 10 & 15 & 21 & 28 \end{bmatrix}$$

Esercizio 8: *Unique paths II (link)*

Un robot si muove su di una griglia `n x m`. Il robot inizialmente è posizionato sulla cella $[0]\,[0]$ e deve arrivare alla cella $[m-1]\,[n-1]$. Il robot può muoversi solamente verso il basso e verso destra. Sulla griglia possono essere presenti degli ostacoli attraverso i quali il robot non può passare. Data una matrice `obstacles`, nella quale le celle con valore 1 indicano le celle con ostacoli, trovare il numero di percorsi possibili per arrivare nell'ultima cella in fondo a destra

Input

Gli interi `m` e `n` e nelle `m` righe successiva gli elementi della matrice `obstacles`

Output

Il numero di percorsi possibili

Input	Output	Discussione
3 3 0 0 0 0 1 0 0 0 0	2	I percorsi possibili sono $[(R \to R \to D \to D), (D \to D \to R \to R)]$

Complessità ottimale: $O(n \cdot m)$

L'idea è molto simile al problema Unique paths, con l'unica differenza che dobbiamo tenere in considerazione i casi in cui una rotta è preclusa da un ostacolo. Inoltre, anzichè creare una nuova matrice `dp`, possiamo usare direttamente la matrice `obstacles` che ci viene data.

- Riempio la prima colonna e riga della matrice `obstacles` con valore 1 fino al primo ostacolo. Dal primo ostacolo in poi avrò solo celle irraggiungibili. Setto le celle irraggiungibili con valore -1, in quanto usando 1 rischierei di confondere le celle irraggiungibili con le celle raggiungibili da 1 solo cammino
- Calcolo le celle rimanenti con la stessa logica usata in Unique paths, tenendo conto però che:
  - Nel caso ci sia un ostacolo nella cella a sinistra o in alto a quella corrente non posso arrivare da quella direzione. Se non rimane nemmeno una rotta possibile posso impostare il valore della della a -1, in quanto non riesco a ragiungerla in nessun modo
  - Nel caso ci sia un ostacolo nella cella corrente (`obstacles[i][j] == 1`) cambio il valore e metto -1, onde evitare ambiguità come spiegato precedentemente
- Ancora una volta, nella cella $[m-1, n-1]$ ci sarà la soluzione del problema

$$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix} \xrightarrow{\text{riempio prima riga e colonna}} \begin{bmatrix}1&1&1\\1&1&0\\1&0&0\end{bmatrix} \xrightarrow{\text{cambio ostacolo in -1}} \begin{bmatrix}1&1&1\\1&-1&0\\1&0&0\end{bmatrix}$$

$$\begin{bmatrix}1&1&1\\1&-1&1\\1&0&0\end{bmatrix} \begin{bmatrix}1&1&1\\1&-1&0\\1&1&0\end{bmatrix} \begin{bmatrix}1&1&1\\1&-1&1\\1&1&2\end{bmatrix}$$

4 Feature importanti di cpp

Di seguito vediamo una scarrellata di feature un filo più avanzate di c++ e della standard library che possono tornare molto utili in programmazione competitiva.

4.1 Type inference - auto

Spesso con l'aumentare della complessità delle strutture dati, il tipo di una variabile può diventare molto lungo e difficile da scrivere. Per ovviare a questo problema, C++ fornisce la keyword `auto`, che permette al compilatore di dedurre automaticamente il tipo della variabile in fase di compilazione. Ad esempio, per inizializzare un iteratore, anzichè scrivere:

```
std::vector<int>::iterator it = vec.begin();
```

possiamo scrivere:

```
auto it = vec.begin();
```

Questo meccanisco si chiama *type inference*. Di fatto affidiamo al compilatore il compito di dedurre il tipo della variabile in base al contesto

4.2 Passaggio per riferimento

Solitamente in C++ le variabili vengono passate per valore, ossia viene creata una copia della variabile originale. In alcuni casi, però, può essere utile passare la variabile per riferimento. Questo meccanismo è molto simile all'uso di un puntatore. Questo può essere desiderabile per due motivi principali:

- Per far si che eventuali modifiche alla variabile all'interno della funzione si riflettono anche sulla variabile originale
- Per evitare il costo computazionale di copiare variabili di grandi dimensione

Per fare ciò, basta aggiungere il simbolo `&` al tipo della variabile nel prototipo della funzione. Ad esempio:

```
void by_ref(int& x){
  x += 10; // Modifica la variabile originale
  }
void by_val(int x){
  x += 10; // Modifica solo la copia locale nella funzione
  }
```

Tendenzialmente, questo viene particolarmente utile per passare strutture dati quali `vector`, i quali possono essere molto grandi:

```
void by_ref(std::vector<int>& vec){ } // Veloce
void by_val(std::vector<int> vec) { } // Lento
```

4.3 Range-based for (for-each loop)

Possiamo utilizzare dello zucchero sintattico per iterare su collezioni come array e vettori. Invece di scrivere:

```
for (int i = 0; i < vec.size(); ++i) {
  process(vec[i]);
}
```

possiamo scrivere:

```
for (auto &element : vec) {
  process(element);
}
```

Nota come qui si faccia uso del passaggio per riferimento (sezione 4.2) e dell'inferenza di tipo (sezione 4.1)

4.4 Overloading degli operatori

In C++, è possibile sovraccaricare gli operatori per definire comportamenti personalizzati per le classi (o struct) definite dall'utente. Questo permette di utilizzare operatori standard (come +, -, , ecc.) con oggetti delle proprie classi in modo intuitivo. Fra questi, quelli che a noi tornano più utili sono:

4.4.1 Opzioni per eseguire l'overloading

Saper eseguire l'overloading dell'operatore di confronto minore (<) è particolarmente utile quando si lavora con strutture dati che richiedono un ordinamento, come `std::set` (sezione 6.5) o `std::priority_queue` (sezione 6.4). Definendo questo operatore, possiamo specificare come gli oggetti della nostra classe devono essere confrontati tra loro.

Supponendo di avere la seguente struct:

```
struct Point {
  int x, y;
  Point(int x, int y) : x(x), y(y) {  }
};
```

Abbiamo 2 modi per eseguire l'overloading dell'operatore

Metodo 1: Definire l'operatore all'interno della classe

```
struct Point {
    // Contenuto della struct
      bool operator<(const Point& other) const {
            return x < other.x;
        }
   };
```

Metodo 2: Definire l'operatore come funzione esterna

```
// Funzione esterna alla struct Point
bool operator<(const Point& a, const Point& b) {
   return a.x < b.x;
}
```

4.4.2 Funzioni da saper "overloadare"

Le funzioni più comuni che possono essere utili da overloadare sono:

- Operatore minore (<): Utile per poter inserire oggetti in strutture che richiedano un ordinamento interno, come as esempio (sezione 6.5) o `std::priority_queue` (sezione 6.4)

  ```
  // All'interno della struct
  bool operator<(const Point& other) const {}
  // Oppure come funzione esterna
  bool operator<(const Point& a, const Point& b) {}
  ```

  La funzione deve restituire `true` se l'oggetto corrente/a sinistra è minore di `other` (o `b` nella versione esterna)

- Operatore di output (<<): Permette di stampare oggetti personalizzati tramite flussi di output come `std::cout`

  ```
  // Come funzione esterna
  std::ostream& operator<<(std::ostream& os, const Point& p) {
      return os << "(" << p.x << ", " << p.y << ")";
  }
  ```

  La funzione deve restituire un riferimento a `std::ostream` e permette di utilizzare `cout << punto;`

4.5 Structs

Talvolta risulta utile immagazzinare i dati in una struct o classe. In programmazione competitiva non è mai rischiesto l'utilizzo di funzioni avanzate, ma è bene sapere almeno la sintassi di base. Definiamo ad esempio una struttura `Point`:

```
struct Point {
  int x;
  int y;
  // Costruttore "standard"
  Point (int x, int y) {
    this->x = x;
    this->y = y;
  }
  // Costruttore con initializer list
  Point(int x, int y) : x(x), y(y) {}
};
```

Una feature specifica di cpp è l'initializer list, che fornisce una strategia efficace per inizializzare i campi di una struttura o classe. Invece di assegnare i valori ai membri all'interno del corpo del costruttore, possiamo utilizzare l'initializer list per inizializzare i membri direttamente al momento della creazione dell'oggetto. Questo può essere più efficiente, specialmente per tipi di dati complessi o costosi da copiare.

4.6 Inizializzatori di liste

In cpp si possono inizializzare liste e vettori in maniera efficace usando le graffe {}. Ad esempio:

```
// Inizializza un array di interi
int arr[] = {1, 2, 3, 4, 5};

// Inizializza un vettore di interi
vector<int> vec = {10, 20, 30};

// Inizializza una matrice 3x3
vector<vector<int>> matrix = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

Una sintassi molto simile può essere usata per inizializzare le struct:

```
struct Point {
  int x;
  int y;
};

point p = {10, 20};        // Inizializza un oggetto Point con x=10 e y=20
point q = {.x=30, .y=40};  // Equivalente a quella sopra
```

Alcuni esempio più complessi possono essere:

```
vector<Point> points = { {1, 2}, {3, 4}, {5, 6} };
unordered_map<int, Point> pointMap = {
  {1, {10, 20}},
  {2, {30, 40}}
};
```

5 Funzioni utili della std library

La standard library di C++ offre una vasta gamma di funzioni e algoritmi che possono semplificare notevolmente il codice e migliorare le prestazioni. Di seguito sono elencate alcune delle funzioni più utili e comuni che possono essere impiegate in programmazione competitiva.

5.1 Sorting - std::sort

Indubbiamente, fra le funzioni più comuni e utili della standard library di C++ c'è la funzione `std::sort`, che permette di ordinare array e vettori in maniera molto efficiente. La funzione utilizza l'algoritmo di ordinamento Introsort, che combina Quicksort, Heapsort e Insertion Sort per garantire prestazioni ottimali nella maggior parte dei casi ($O(n \log n)$).

```
std::sort(begin_iterator, end_iterator);
```

ad esempio, per ordinare un vettore di interi:

```
// Con vector
std::vector<int> vec = {4, 2, 5, 1, 3};
std::sort(vec.begin(), vec.end());

// Con array
int v[] = {4, 2, 5, 1, 3};
std::sort(v, v + 5); // v + 5 punta alla fine dell'array
```

inoltre, è possibile specificare un criterio di ordinamento personalizzato passando un *functor*. In particolare possiamo specificare `std::less<type>()` per ordinare in ordine crescente e `std::greater<type>()` per ordinare in ordine decrescente:

```
std::vector<int> vec = {4, 2, 5, 1, 3};

// Ordina in ordine crescente, default
std::sort(vec.begin(), vec.end(), std::less<int>());
// Ordina in ordine decrescente
std::sort(vec.begin(), vec.end(), std::greater<int>());
```

5.2 Pitov - std::nth_element

La funzione `std::nth_element` riordina parzialmente un intervallo: dopo la chiamata, l'elemento in posizione `nth` conterrà il valore che avrebbe dopo un ordinamento completo, tutti gli elementi prima saranno minori o uguali e quelli dopo maggiori o uguali (ma la partizione non è ordinata). Tempo atteso $O(n)$.

```
std::nth_element(begin, nth, end);
```

```
std::vector<int> v = {4, 2, 8, 1, 5};
std::nth_element(v.begin(), v.begin() + 2, v.end());
// v[2] ora contiene il terzo elemento più piccolo
```

5.3 Binary search 1 - std::lower_bound, std::upper_bound

`std::lower_bound` restituisce l'iteratore al primo elemento $\geq$ di un valore; `std::upper_bound` restituisce l'iteratore al primo elemento $>$ di un valore. Entrambe richiedono che l'intervallo sia ordinato e lavorano in $O(\log n)$.

```
std::lower_bound(begin, end, val);
```

```
std::upper_bound(begin, end, val);
```

```
std::vector<int> v = {1, 2, 3, 5, 5, 8};
// Primo >= 5
auto it1 = std::lower_bound(v.begin(), v.end(), 5); // punta a v[3]
// Primo > 5
auto it2 = std::upper_bound(v.begin(), v.end(), 5); // punta a v[5]
```

5.4 Binary search 2 - std::binary_search

Controlla se un elemento è presente in un intervallo ordinato usando ricerca binaria, ritorna `true` o `false`. Complessità $O(\log n)$.

```
std::binary_search(begin, end, val);
```

```
std::vector<int> v = {1, 2, 3, 5, 8};
if (std::binary_search(v.begin(), v.end(), 3)) {
    // 3 è presente
}
```

5.5 Range elementi uguali - std::equal_range

Restituisce una coppia di iteratori: primo elemento $\geq val$, primo $> val$; utile per trovare il range di un certo valore in uno spazio ordinato.

```
auto range = std::equal_range(begin, end, val);
```

```
std::vector<int> v = {1, 2, 5, 5, 5, 8};
auto range = std::equal_range(v.begin(), v.end(), 5);
// range.first punta a v[2], range.second a v[5]
```

5.6 Minimo e massimo in vettori - std::min_element, std::max_element

Trovano l'iteratore all'elemento minimo o massimo in un intervallo.

```
std::min_element(begin, end);
```

```
std::max_element(begin, end);
```

```
std::vector<int> v = {4, 7, 2, 9, 1};
auto it_min = std::min_element(v.begin(), v.end()); // v[4] = 1
auto it_max = std::max_element(v.begin(), v.end()); // v[3] = 9
```

5.7 Somma e moltiplica - std::accumulate

La funzione `std::accumulate` calcola la somma (o un'operazione generica) degli elementi nell'intervallo. Necessita di `#include <numeric>`.

```
std::accumulate(begin, end, init);
```

```
std::vector<int> v = {1, 2, 3, 4};
int somma = std::accumulate(v.begin(), v.end(), 0); // somma = 10

// anche con operazione generica (ad esempio moltiplicazione)
int prodotto = std::accumulate(v.begin(), v.end(), 1,
    std::multiplies<int>());
```

5.8 Minimo e massimo - std::minmax_element

Restituisce una coppia di iteratori: primo al minimo, secondo al massimo. Unico ciclo su tutto l'intervallo.

```
auto [it_min, it_max] = std::minmax_element(begin, end);
```

```
std::vector<int> v = {4, 7, 2, 9, 1};
auto res = std::minmax_element(v.begin(), v.end()); // res.first verso il
    minimo, res.second verso il massimo
```

5.9 Conta elementi - std::count

Conta quante volte un dato valore appare nell'intervallo.

```
std::count(begin, end, val);
```

```
std::vector<int> v = {1, 5, 2, 5, 5, 8};
int howMany = std::count(v.begin(), v.end(), 5); // howMany = 3
```

5.10 Trova elementi - std::find

Restituisce un iteratore al primo elemento uguale a quello cercato, oppure `end` se non trovato.

```
std::find(begin, end, val);
```

```
std::vector<int> v = {1, 3, 4, 6};
auto it = std::find(v.begin(), v.end(), 4); // it punta a v[2]
if (it == v.end()) {
    // valore non trovato
}
```

6 Strutture dati di cpp

Nella standard library di C++ sono presenti diverse strutture dati che possono essere utili per implementare algoritmi di programmazione dinamica. Di seguito una breve panoramica delle più comuni.

6.1 Vector

Il vector fornisce un array dinamico, che può quindi cambiare la sua dimensione dinamicamente in maniera efficiente. Le operazioni principali che ci fornisce consistono nell'inserimento di elementi in fondo e accesso tramite indice in tempo costante

```
std::vector<type> v(size, initial_value);
```

Funzione	Complessità	Descrizione
`v.size()`	$O(1)$	Restituisce il numero di elementi contenuti
`v.empty()`	$O(1)$	Ritorna `true` se il vettore è vuoto
`v.push_back(x)`	$O(1)^*$	Aggiunge un elemento in fondo; può riallocare
`v.pop_back()`	$O(1)$	Rimuove l'ultimo elemento
`v[i]`	$O(1)$	Accesso diretto senza controlli
`v.at(i)`	$O(1)$	Accesso controllato con verifica dell'indice
`v.front()`	$O(1)$	Ritorna riferimento al primo elemento
`v.back()`	$O(1)$	Ritorna riferimento all'ultimo elemento
`v.begin(), v.end()`	$O(1)$	Ritorna iteratori di inizio e fine
`v.insert(pos, x)`	$O(n)$	Inserisce un elemento nell'indice specificato
`v.erase(pos)`	$O(n)$	Rimuove un elemento all'indice specificato
`v.clear()`	$O(n)$	Rimuove tutti gli elementi
`v.resize(n)`	$O(n)$	Cambia la dimensione logica del vettore

Con * si indica la complessità ammortizzata. Detta "stringi stringi", significa che l'operazione singola può avere complessità superiore a quella indicata, ma su una successione di operazioni possiamo essere sicuri che il costo medio sia quello indicato

6.2 Stack

Lo `stack` implementa una struttura LIFO (Last In, First Out), in cui l'ultimo elemento inserito è il primo a essere rimosso. È molto efficiente e tipicamente utilizzato per gestire operazioni annidate, backtracking e funzioni ricorsive.

```
std::stack<type> s;
```

Funzione	Complessità	Descrizione
`s.size()`	$O(1)$	Restituisce il numero di elementi
`s.empty()`	$O(1)$	Ritorna `true` se lo stack è vuoto
`s.push(x)`	$O(1)$	Inserisce un elemento in cima
`s.pop()`	$O(1)$	Rimuove l'elemento in cima
`s.top()`	$O(1)$	Ritorna riferimento all'elemento in cima

6.3 Queue

La `queue` implementa una struttura FIFO (First In, First Out), in cui il primo elemento inserito è il primo a uscire. È ideale per modellare code, buffer circolari o processi di scheduling.

```
std::queue<type> q;
```

Funzione	Complessità	Descrizione
`q.size()`	$O(1)$	Restituisce il numero di elementi
`q.empty()`	$O(1)$	Ritorna `true` se la queue è vuota
`q.push(x)`	$O(1)$	Inserisce un elemento in fondo
`q.pop()`	$O(1)$	Rimuove l'elemento frontale
`q.front()`	$O(1)$	Riferimento al primo elemento
`q.back()`	$O(1)$	Riferimento all'ultimo elemento

6.4 Priority Queue

La `priority_queue` è una coda con priorità implementata solitamente tramite *heap*. Permette di estrarre sempre l'elemento con valore massimo (o minimo, se configurata come min-heap) in maniera efficiente (tempo costante).

```
std::priority_queue<type> pq;
```

Funzione	Complessità	Descrizione
`pq.size()`	$O(1)$	Numero di elementi
`pq.empty()`	$O(1)$	`true` se vuota
`pq.top()`	$O(1)$	Ritorna l'elemento con priorità più alta
`pq.push(x)`	$O(\log n)$	Inserisce un elemento preservando l'heap
`pq.pop()`	$O(\log n)$	Rimuove l'elemento con priorità più alta

Questa struttura dati, a differenza di vettori, stack, queue e altre ha bisogno che il tipo di elemento contenuto implementi l'operatore `<` (dal punto di vista logico risulta impossibile creare questa struttura dati se non esiste un criterio secondo il quale possiamo ordinare gli elementi)

6.5 Set

Il `set` è una struttura che contiene elementi unici, ordinati secondo un comparatore (`std::less<>` di default). Utile quando abbiamo bisogno di tenere una collezione di dati che risulta ordinata ad ogni momento. Internamente è implementato come un albero bilanciato Red-Black, che garantisce prestazioni logaritmiche.

```
std::set<type> s;
```

Funzione	Complessità	Descrizione
`s.size()`	$O(1)$	Numero di elementi
`s.empty()`	$O(1)$	`true` se vuoto
`s.insert(x)`	$O(\log n)$	Inserisce un elemento (se non presente)
`s.erase(x)`	$O(\log n)$	Rimuove un elemento
`s.find(x)`	$O(\log n)$	Ritorna iteratore all'elemento, se presente
`s.count(x)`	$O(\log n)$	Ritorna 1 se presente, 0 altrimenti
`s.begin(), s.end()`	$O(1)$	Iteratori inizio/fine
`s.lower_bound(x)`	$O(\log n)$	Primo elemento $\geq$ x

Nota che a differenza della priority queue (sezione 6.4), *ogni* elemento del set è ordinato, mentre la priority queue garantisce l'accesso solo al maggiore/minore

6.6 Map

La `map` è una struttura associativa che conserva coppie chiave–valore, ordinate secondo un comparatore sulla chiave. Anche qui l'implementazione è basata su un albero Red-Black.

```
std::map<key_type, value_type> m;
```

Funzione	Complessità	Descrizione
`m.size()`	$O(1)$	Numero di elementi
`m.empty()`	$O(1)$	True se vuota
`m.insert(kv)`	$O(\log n)$	Inserisce una coppia chiave–valore
`m.erase(k)`	$O(\log n)$	Rimuove la chiave
`m.find(k)`	$O(\log n)$	Trova la chiave
`m.count(k)`	$O(\log n)$	1 se presente, 0 altrimenti
`m[k]`	$O(\log n)$	Accesso/creazione valore associato alla chiave
`m.at(k)`	$O(\log n)$	Accesso con controllo (senza creare)

6.7 Unordered Map

La `unordered_map` implementa una tabella hash, che permette operazioni di accesso e inserimento in tempo medio costante. L'ordine degli elementi non è definito.

```
std::unordered_map<key_type, value_type> um;
```

Funzione	Complessità	Descrizione
`um.size()`	$O(1)$	Numero di elementi
`um.empty()`	$O(1)$	True se vuota
`um.insert(k, v)`	Medio $O(1)$ / Worst $O(n)$	Inserisce o aggiorna un elemento
`um.erase(k)`	Medio $O(1)$ / Worst $O(n)$	Rimuove la chiave
`um.find(k)`	Medio $O(1)$ / Worst $O(n)$	Trova la chiave
`um.count(k)`	Medio $O(1)$ / Worst $O(n)$	True se presente
`um[k]`	Medio $O(1)$ / Worst $O(n)$	Accesso/creazione valore
`um.at(k)`	$O(1)$	Accesso con controllo

Nota come dal punto di vista della performance, la `unordered_map` sia preferibile alla `map`. Quindi va preferita la textttmap solo nel caso in cui sia necessario avere le chiavi *sempre* ordinate

7 Problemi sito oii consigliati

Di seguito una raccolta di problemi provenienti dal sito degli allenamenti della OII. Sono proposte delle soluzioni in sezione sezione 8

Problema	Tecniche	Difficoltà	Soluzione
Figonacci (`figonacci`)	dp	★ ☆ ☆ ☆ ☆	sezione 8.1
Discesa massima (`discesa`)	dp	★ ☆ ☆ ☆ ☆	sezione 8.2
Police 3 (`police3`)	dp	★ ☆ ☆ ☆ ☆	sezione 8.3
Piano degli studi (`pianostudi`)	dp, binary search	★ ★ ★ ☆ ☆	sezione 8.4
Police 4 (`police4`)	dp	★ ★ ☆ ☆ ☆	sezione 8.5
Spiedini di frutta (`spiedini`)	dp	★ ★ ☆ ☆ ☆	sezione 8.6
K-step ancestor (`treeancestor`)	dp, graph	★ ★ ☆ ☆ ☆	sezione 8.7
Taglialegna (`taglialegna`)	dp, amortized analysis	★ ★ ★ ★ ★	sezione 8.8
Ma chi è quel mona (`porte`)	dp	★ ★ ☆ ☆ ☆	sezione 8.9
Truffa al sushi (`sushi`)	dp, binary search, math	★ ★ ★ ★ ★	sezione 8.10

8 Soluzioni problemi sito OII

Di seguito una raccolta di soluzioni per i problemi proposti in sezione 7. Per ogni problema è riportato il link alla pagina del problema e il link alla soluzione proposta in questa dispensa.

8.1 Figonacci (figonacci)

Problema sito OII	https://training.olinfo.it/task/ois_figonacci
Soluzione proposta	files/esercizi/figonacci

L'idea qui è di "srotolare" la sommatoria, rendendosi conto che

$$\begin{aligned} G_n &= \sum_{i=0}^{i=n-2} G_{n-1} - G_i \\ &= (G_{n-1} - G_{n-2}) + (G_{n-1} - G_{n-3}) + \ldots + (G_{n-1} - G_1) + (G_{n-1} - G_0) \\ &= (n-1)\, G_{n-1} - \sum_{i=0}^{i=n-2} G_i \\ &= (n-1)\, G_{n-1} + G_{n-1} - (n-2)\, G_{n-2} - G_{n-2} \\ &= (n-1)\, G_{n-1} - (n-1)\, f\,(n-2) + G_{n-1} \\ &= (n-1)\,(G_{n-1} - G_{n-2}) + G_{n-1} \end{aligned}$$

abbiamo così ottenuto a tutti gli effetti una formula che possiamo applicare direttamente in forma ricorsiva(con *memoization*) o iterativa:

```
dp[i] = (i-1)(dp[i-1]*dp[i-2]) + dp[i-1]
```

8.2 Discesa massima (discesa)

Problema sito OII	https://training.olinfo.it/task/discesa
Soluzione proposta	files/esercizi/discesa

L'idea in questo caso è di creare un albero di discesa in cui in ogni posizione `[i][j]` salviamo *il maggior peso di una discesa che termina nella posizione* `[i][j]`.

Ad esempio, qui sotto è riportato a destra l'albero `dp` per l'albero in input di sinistra

$$\begin{array}{c} 1 \\ 2\quad 9 \\ 3\quad 7\quad 5 \\ 8\quad 4\quad 11\quad 6 \end{array} \quad \rightarrow \quad \begin{array}{c} 1 \\ 3\quad 10 \\ 6\quad 17\quad 15 \\ 14\quad 21\quad 28\quad 21 \end{array}$$

rimane da capire come calcolare i valori nella file i esima basandosi sui precedenti. In particolare abbiamo 3 casi:

- *Primo elemento*: posso arrivare *solo da destra*, quindi:

```
dp[i][0]=dp[i-1][0] + w[i][0]
```

- *Ultimo elemento*: posso arrivare *solo da sinistra*, quindi

```
dp[i][size]=dp[i-1][size-1] + w[i][size]
```

- *Elemento in mezzo*: posso arrivare sia da sinistra che da destra, quindi dovro prendere la strada che fra le due ha peso maggiore:

```
dp[i][j]=w[i][j] + max(dp[i][j], dp[i][j+1])
```

Chiaramente il caso base è quello riguardante il primo elemento, per cui il persorso di peso massimo è dato dal peso dell'elemento stesso.

Occorre ragionare un filo meglio sugli indici ma la logica rimane inalterata. Nota bene: *è veramente necessario salvare l'intera matrice dp?*

8.3 Police 3 (police3)

Problema sito OII	https://training.olinfo.it/task/ois_police3
Soluzione proposta	files/esercizi/police3

L'idea qui è, per quanto banale possa semprare, ad ogni semaforo abbiamo due opzioni: fermarsi o meno. Per questa ragione possiamo salvare informazioni in due vettori `dp`, per ciascuno dei due casi:

- `ts[i]`: salva in i il minor tempo per superare il semaforo i fermandosi a quest'ultimo
- `tr[i]`: salva in i il minor tempo per superare il semaforo i, scappando

possiamo costruire i valori dei due vettori basandoci sui precedenti secondo la seguente logica

- Se *mi fermo* al semaforo i-esimo, il tempo ideale sarà dato dall'attesa al semaforo + il tempo migliore ottenuto fermandosi o meno a quello precedente (posso scegliere di fermarmi oppure no, siccome non violo nessun vincolo in ogni caso)

```
ts[i] = T[i] + min(tr[i - 1], ts[i - 1]);
```

- Se *non mi fermo* al semaforo i-esimo, il tempo ideale sarà dato dal tempo ideale per arrivare al semaforo $i - 1$, fermandocisi (in questo caso devo assicurarmi di fermarmi al semaforo precedente per non violare alcun vincolo)

```
tr[i] = ts[i - 1];
```

Anche in questo caso, è davvero necessario salvare l'interi vettori in memoria?

8.4 Piano degli studi (pianostudi)

Problema sito OII	https://training.olinfo.it/task/ois_pianostudi
Soluzione proposta	files/esercizi/pianostudi

8.5 Police 4 (police4)

Problema sito OII	https://training.olinfo.it/task/ois_police4
Soluzione proposta	files/esercizi/police4

L'idea in questo caso è di salvare di volta il tempo minimo per arrivare al semaforo `i` avendone saltati al più `j`. Questa informazione può essere mantenuta in una matrice $n \times r$, dove n è il numero di *semafori* e r il numero massimo di semafori che *possono essere saltati*.

La i-esima colonna/riga della matrice può essere calcolata usando la precedente. In particolare, supponiamo di voler calcolare il tempo minore per arrivare al semaforo i saltandone al più j. Allora ho due opzioni:

- Arrivo al semaforo $i-1$ saltandone al massimo $j-1$. In questo caso il tempo ideale è quello ottenuto saltando il semaforo $i-1$
- Arrivo al semaforo $i-1$ saltandone al più j. In questo caso devo controllare se devo aspettare o meno. In caso affermativo devo aggiungere il tempo di attesa alla soluzione

La soluzione ottimale è data dalla migliore delle due. Formarlmente dati T e X, rispettivamente l'intervalli dei semafori e il vettore delle posizioni, posso calcolare `dp` come segue:

```
t1 = dp[i-1][j-1];
t2 = dp[i-1][j] + (can_pass ? 0 : T-dp[i-1][j] % T);
dp[i][j] = min(t1, t2) + X[i] - X[i-1];
```

8.6 Spiedini di frutta (spiedini)

Problema sito OII	https://training.olinfo.it/task/oii_spiedini
Soluzione proposta	files/esercizi/spiedini

L'idea è che una volta che uno spiedino è stato mangiato fino ad un certo punto da una parte, allora la soluzione ideale è prefissata e consiste nel mangiarlo dalla parte opposta finchè è possibile. La chiave sta tuttavia nel comprendere che non è necessario calcolare tutta la somma da capo ad ogni iterazione, ma possiamo seguire questa logica:

- Mangio lo spiedino *da sinistra* fino all'ultima fragola che permette di *non* superare la soglia. Tengo sempre traccia della soglia corrente
- Partendo *da destra* mangio lo spiedino fino a quando la soglia lo permette
- Tengo traccia del numero di fragole mangiate i

Dopodichè, ripeto questi step fino a quanto non raggiungo l'estremità sinistra dello spiedino:

- Da sinistra, scorro fino alla fragola precedente, abbassando la soglia
- Da destra scorro fino all'ultima fragola che riesco ad includere, aggiornando soglia e quantità di fragole mangiate

La soluzione migliore ottenuta in ogni step dell'iterazione precedente è la soluzione ottimale al problema

8.7 K-step ancestor (treeancestor)

Problema sito OII	https://training.olinfo.it/task/ois_treeancestor
Soluzione proposta	files/esercizi/treeancestor

La soluzione sta nell'esplorare l'albero tenendo traccia della profondità corrente e del tragitto che ha portato fino al nodo che stiamo attualmente visitato. Possiamo fare ciò tramite 2 variabili `depth` e `path[n]`

Dato che il grado in questo caso è semplicemente un albero, possiamo sfruttare anche una semplice DFS, dato che possiamo essere sicuri che il path che ci porta a ciascun nodo è unico.

Per ciascun nodo, controlliamo se la depth è maggiore o minore di K. Nel primo caso sfruttiamo l'array contenente il path per assegnare il valore del k-ancestor, nel secondo semplicemente assegnamo -1. Formalmente:

```
if (depth >= K) {
  ancestor[node] = path[depth - K];
} else {
  ancestor[node] = -1;
}
```

nella DFS chiamiamo ricorsivamente la funzione con un valore di `depth` incrementato di 1 ogni volta

8.8 Taglialegna (taglialegna)

Problema sito OII	https://training.olinfo.it/task/oii_taglialegna
Soluzione proposta	files/esercizi/taglialegna

Il problema è piuttosto avanzato, per questo è necessario fare delle osservazioni preliminari che torneranno utili dopo.

1. In una soluzione ottima ho 3 opzioni:
   (a) L'ultimo albero a destra viene tagliato e fatto cadere a *sinistra*
   (b) L'ultimo albero a destra viene tagliato e fatto cadere a *destra*
   (c) L'ultimo albero a destra *non* viene tagliato ma fatto cadere a *destra* per "effetto domino"

   L'idea importante è che in una soluzione ottima l'ultimo albero a destra, se cade a sinistra è perchè è stato tagliato e fatto cadere a sinistra. Per quanto sia ovvio è molto importante per capire gli step successivi

2. Gli unici punti in cui si può interrompere l'effetto domino sono i punti in cui vi sono alberi di altezza 1

3. Per quanto detto al punto precedente, possiamo vedere il problema come una serie di intervalli che errano fatti cadere in una delle due direzioni. Agli estremi di questi intervalli ci sono alberi di altezza 1

4. Quando avviene un effetto domino, non tutti gli alberi partecipano nella catena della caduta. In particolare un albero può essere abbattuto da un albero che non è quello precedente nella catena. Chiameremo questo albero `ff`(first-falling) di `i`. Possiamo quindi considerare l'effetto domino solo per gli alberi abbattitori, nel modo seguente:
$$i \to \text{ff}\,[i] \to \text{ff}\,[\text{ff}\,[i]] \to \dots$$

Detto questo, individuiamo il sotto-problema di programmazione dinamica e il caso base. In particolare sappiamo che:

- Il sotto-problema consiste nel calcolare il *il numero minore di tagli* per abbattere gli alberi da 0 a i

- Il caso base è quanto $n = 1$, ossia quando ho un solo albero. In questo caso la soluzione è fare un unico taglio

Rimane ora capire come utilizzare le soluzioni ottime da 0 a $i-1$ per calcolare la i-esima soluzione ideale. Possiamo ragionare così per calcolare la soluzione ottimale i-esima:

- Se l'albero viene *tagliato a sinistra*, allora simulo l'effetto domino e verifico dove questo si interrompe. Supponendo che la catena abbia fatto cadere k alberi, allora

```
solution[i] = 1 + solution[i-k-1]
```

- Se l'albero viene *tagliato a destra*, allora non rimane che pulire in maniera ottimale gli alberi da 0 a $i-1$, quindi:

```
solution[i] = 1 + solution[i-1]
```

- Se l'albero viene *fatto cadere a destra*, allora all'interno della *catena dei first-fall* devo capire quale albero è più conveniente tagliare per innescare l'effetto domino. Chiamando questo albero `opt`, allora la soluzione ottimale è data da

```
solution[i] = 1 + solution[opt-1]
```

Andiamo ora ad elencare le strutture dati che ci aiuteranno nel calcolo progressivo del vettore `solution`

- `lb[i]`: *left-bound*, contiene nella posizione i l'indice fino al quali gli alberi cadono se l'albero i viene tagliato a *sinistra*

- `rb[i]`:*right-bound*, contiene nella posizione i l'indice fino al quali gli alberi cadono se l'albero i viene tagliato a *destra*

- `ff[i]`: *first-fall*, come descritto precedentemente, il primo albero alla sinistra di i che in un effetto a catena colpisce i

Come vedremo, possiamo calcolare tutti questi vettori in tempo $O(n)$

- `solution[i]`: il vettore *dp* principale. Salva la soluzione ottimale per il range $[0, i]$
- `dir[i]`: il vettore che contiene la direzione in cui va tagliato il primo albero per ottenere la soluzione ottimale
- `opt_split[i]`: *optimal-split*, salva la posizione ottimale per dare inizio all'effetto domino nella soluzione migliore in cui l'ultimo albero cade a destra
- `first_cut[i]`: salva l'indice del primo albero da tagliare nella i-esima sotto soluzione ottimale

Parte 1: calcolo `lb`, `rb`

Partiamo dal calcolo dei vettori `lb` e `rb`. L'idea è proprio quella di simulare a tutti gli effetti l'effetto domino. In particolare, pensiamo di avere una serie di intervalli che cadono per effetto domino e un albero che viene fatto cadere a sinistra:

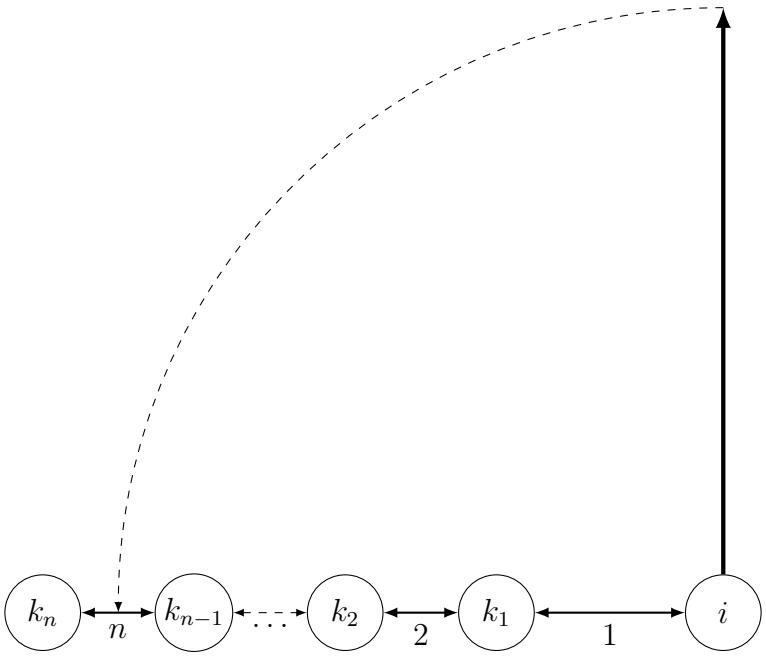


Figura 1: Calcolo left-bound

In questa figura, sappaimo già che gli intervalli da 1 a n cadono a sinistra per effetto domino, interrompendosi nei punti $k_2, \ldots, k_n$. `lb[i]` sarà quindi dato da valore più basso fra i `lb` degli intervalli che l'albero abbatte sicuramente, ossia tutti gli intervalli il cui albero più a destra viene abbattuto dall'albero i. In figura, questi sono precisamente gli intervalli $1, \ldots, n$. Più formalmente

Algoritmo: *Calcolo left-bound*

```
int [] compute_left_bound(n, h):
    int lb ← new int [0 . . . n − 1];
    for i ← 0 to n − 1 do
        lb[i] ← i;
        while lb[i] > 0 ∧ lb[i] > i − h[i] + 1 do
            lb[i] ← lb[ lb[i] − 1 ];
    return lb;
```

Il calcolo del vettore `rb` è speculare, ma segue la medesima logica.

Ora è fondamentale discurete la complessità di questa funzione. Potremmo in un primo momento pensare *erroneamente* che la funzione abbia complessità di $O(n^2)$, dato che il ciclo esterno scorre l'interno vettore e, nel caso peggiore, anche quello interno scorre tutti gli elementi da i a 0. Tuttavia, il ragionamento è un filo più sottile.

Diciamo che un albero viene *saltato da i* se all'interno del ciclo while che calcola `lb[i]` viene utilizzato per aggiornare il valore di *lower-bound*

$$lb[i] \leftarrow lb[\underbrace{lb[i] - 1}_{\text{saltato}}]$$

Aiutandoci con la figura 1, possiamo intuire che gli alberi saltati, nel calcolo di `lb[i]`, sono dati da $k_1, k_2, \ldots k_{n-1}$. La chiave sta nel capire che ogni albero può essere saltato una volta sola. Questo avviene in quanto, supponendo che gli alberi $k_1, \ldots k_{n-1}$ vengano saltati dall'albero i, allora nelle iterazioni successive questi non possono più essere saltati, in quanto al più verrà saltato prima l'albero i, il quale avrà un *left-bound* minore di k_{n-1}, quindi tutto il range in cui stanno gli alberi $k_1, \ldots k_{n-1}$ non verrà considerato nella computazione.

Quindi, ancora una volta, siccome ogni albero viene saltato una e una sola volta, la riga $lb[i] \leftarrow lb[lb[i] - 1]$ del ciclo while viene eseguita al più n volte, portando ad una complessità totale di $O(n)$

Parte 2: calcolo `ff`

Il calcolo di `ff` può implementato in maniera piuttosto semplice con piccolissime modifiche alla funzione che calcola `rb`. Sempre riferendoci a fig. 1, possiamo notare che il primo albero ad abbattere gli alberi $k_1, \ldots, k_{n-1}$ in una catena di abbattitori è i. Per questo motivo ogni volta che calcoliamo `rb[i]` possiamo anche aggiornare il valore per `ff` degli alberi $k_1, \ldots k_{n-1}$. Formalmente:

Algoritmo: *Calcolo right-bound e first-fall*

```
int // compute_rb_ff(n, h):
    int rb ← new int [0 . . . n − 1];
    int ff ← new int [0 . . . n − 1];
    for i ← n − 1 downto 0 do
        rb[i] ← i;
        ff[i] ← −1;
        while rb[i] < n − 1 and rb[i] < i + h[i] − 1 do
            ff[rb[i] + 1] ← i;
            rb[i] ← rb[rb[i] + 1];
    return rb, ff;
```

La complessità rimane sempre $O(n)$

Parte 3: calcolo `solution`

Innanzitutto, ci appoggeremo alle strutture dati `dp` descritte qui. Per calcolare il vettore `solution`, gli step sono, dal punto di vista logico, quelli descritti qui. Descriviamo più nel dettaglio le 2 casistiche:

- Taglio a sinistra:

```
solution[i] = 1 + (lb[i] > 0 ? solution[lb[i] - 1] : 0);
opt_cut[i] = i;
dir[i] = 0;
```

- Albero cade a destra:
  - Taglio a destra:

```
solution[i] = 1 + (idx2 > 0 ? solution[i - 1] : 0);
```

  - Albero abbattuto a destra per effetto domino:

```
split_index = opt_split[ff[i]]; // Optimal index to cut the tree
solution[i] = 1 + (idx1 > 0 ? solution[split_index - 1] : 0);
```

Ora ci rimane solo da aggiornare il vettore `opt_cut`, così da rendere l'intero algoritmo efficiente. Anche qui la programmazione dinamica viene in nostro soccorso. Più in particolare, so che supponendo che la catena di k first fall che abbattono i sia composta dagli alberi $1, 2, \ldots, k-1, i$. Allora so già che fra i primi $k-1$ il migliore è dato da `opt_split[k-1]`, non rimane che testare l'ultimo, ossia il caso in cui i è abbattuto a destra. Nel caso la soluzione migliore si ottenesse con la prima opzione, allora `opt_split[i]=opt_split[ff[i]]`, altrimenti `opt_split[i] = i`

Chiaramente, scorrendo ogni elemento del vettore una sola volta, la funzione ha complessità $O(n)$

Algoritmo: *Calcolo soluzione*

```
compute_solution(n, lb, rb, ff):
    solution[0] ← 1;
    dir[0] ← 1;
    opt_split[0] ← 0;
    first_cut[0] ← 0;

    for i ← 1 to n do

        ▷ Case 1:  last tree is cut to the left
        sol_1 ← 1+if lb[i] > 0 then solution[lb[i] − 1] else 0;
        cut_1 ← i;
        dir_1 ← false;

        ▷ Case 2:  last tree falls to the right
        if ff[i] ≠ −1 then

            ▷ Case 2.1:  tree falls by domino effect
            idx_1 ← opt_split[ff[i]];
            sol_21 ← 1+if idx_1 > 0 then solution[idx_1 − 1] else 0;

            ▷ Case 2.2:  tree is cut to the right
            idx_2 ← i;
            sol_22 ← 1+if idx_2 > 0 then solution[idx_2 − 1] else 0;

            opt_split[i] ←if sol_21 < sol_22 then idx_1 else idx_2;
        else
            opt_split[i] ← i;

        sol_2 ← 1+if opt_split[i] > 0 then solution[opt_split[i] − 1] else 0;
        cut_2 ← opt_split[i];
        dir_2 ← true;

        if sol_1 < sol_2 then
            solution[i] ← sol_1;
            first_cut[i] ← cut_1;
            dir[i] ← dir_1;
        else
            solution[i] ← sol_2;
            first_cut[i] ← cut_2;
            dir[i] ← dir_2;
    return solution, first_cut, dir;
```

Parte 4: ricostruire la soluzione

Una volta calcolati i vettori `solution`, `first_cut` e `dir` è piuttosto immediato ricostruire la soluzione "risalendo" al contrario i tagli che sono stati fatti:

Algoritmo 2: *Ricostruzione soluzione*

```
reconstruct_solution(n, first_cut, dir):
    pos ← n − 1;
    while pos ≥ 0 do
        abbatti(first_cut[pos], dir[pos]);
        if dir[pos] = false then
            pos ← lb[pos] − 1;
        else
            pos ← first_cut[pos] − 1;
```

Complessivamente, abbiamo che

Step	task	complessità
Step 1	calcolo `lb`, `rb`	$O(n)$
Step 2	calcolo `ff`	$O(n)$
Step 3	calcolo `solution`, `dir`, `best_cut`	$O(n)$
Step 4	ricostruisco soluzione	$O(n)$

Complessità totale: $O(n)$

8.9 Ma chi è quel mona (porte)

Problema sito OII	https://training.olinfo.it/task/roiti_porte
Soluzione proposta	files/esercizi/porte

L'idea chiave qui sta in un'assunzione implicita del testo del problema:

"Gaetano inizia sbattendo la porta 0 e *deve terminare sbattendo l'ultima, la* $N-1$"

quindi ci viene detto che per forza l'ultima porta della sequenza verrà sbattuta.

Possiamo procedere definendo la struttura `dp`: `dp[i]` conterrà il numero di sequenze armoniche distinte possibili utilizzando le prime `i` porte

Ora sappiamo che se sbattiamo la porta i-esima, allora possiamo sbattere le porte $i-1$, $i-2$, $i-4$, $i-8$ e così via, più in generale tutte le porte $i-2^j$. Dobbiamo quindi considerare ciascuna delle seguenti opzioni e sommare il numero di combinazioni possibili. Per ciascuna di queste combinazioni, tuttavia, abbiamo già salvato il numero di sequenze armoniche possibile nel vettore $dp[i-2^j]$. E' necessario tuttavia considerare che il numero di sequenze armoniche va incrementato solo se vale la condizione specificata nel testo del problema, ossia che *la somma delle loro frequenze deve essere divisibile per la loro distanza*, formalmente:

$$\text{Incremento} \Leftrightarrow (A[i] + A[i-2^j]) \mod 2^j == 0$$

Il caso base è dato da `dp[0] = 1`, in quanto c'è una sola sequenza armonica possibile con una sola porta

Algoritmo 3: *Porte*

```
int compute_sequences(N, A):
    dp ← new int [0...N − 1] = [0, ..., 0] ;          ▷ dp[i]=0
    dp[0] ← 1;
    foreach i in 0...N − 1 do
        foreach j such that i − 2^j ≥ 0 do
            if (A[i] + A[i − 2^j]) mod 2^j = 0 then
                dp[i] ← dp[i] + dp[i − 2^j];
    return dp[N − 1] mod MOD
```

Detto questo, è possibile implementare l'algoritmo con qualche accortezza riguardante i moduli, applicandolo ogni volta che incrementiamo `dp[i]`, e salvandoci la potenza di 2 di volta in volta, evidanto di doverla ricalcolare ad ogni iterazione:

Algoritmo 4: *Porte 2*

```
int compute_sequences_2(N, A):
    dp ← new int [0...N − 1] = [0, ..., 0] ;          ▷ dp[i]=0
    dp[0] ← 1;
    for i ← 1 to N − 1 do
        pow ← 1;
        while pow ≤ i do
            if (A[i] + A[i − pow]) mod pow = 0 then
                dp[i] ← dp[i] + dp[i − pow];
                dp[i] ← dp[i] mod MOD;
            pow ← pow · 2;
    return dp[N − 1] mod MOD
```

Ad ogni iterazione del ciclo esterno, il ciclo interno viene eseguito al più $O(\log N)$ volte.

Complessità totale: $O(N \log N)$

8.10 Truffa al sushi (sushi)

Problema sito OII	https://training.olinfo.it/task/preoii_sushi
Soluzione proposta	files/esercizi/sushi

Il problema è chiaramente di programmazione dinamica e presenta delle soluzioni naiv che tuttavia risultano troppo pesanti dal punti di vista computazionale. Per vedere come possiamo arrivare ad una soluzione efficiente partiamo esplorando le più naive. Nel nostro caso partiremo da una soluzione "ad hoc" piuttosto semplice dal punti di vista intuitivo che comunque porta ad un discreto punteggio ed esploreremo poi diverse ottimizzazioni

Soluzione	Punteggio	Descrizione
files/esercizi/sushi/v2	70	Knapsack e ricerca lineare
files/esercizi/sushi/v1	84	Soluzione "ad hoc" naive
files/esercizi/sushi/v3	99	Knapsack e ricerca esponenziale/binaria
files/esercizi/sushi/v4	100	Knapsack, ricerca esponenziale/binaria e bitset
files/esercizi/sushi/v5	100	Knapsack, ricerca esponenziale/binaria e bitset custom

Eviteremo di affrontare la soluzione "ad hoc" in quanto non permette di essere riadattada per ottenere full score. Ci concentreremo invece sulle soluzioni basate su knapsack e ricerca esponenziale/binaria.

8.10.1 Idea di base del knapsack

Questo problema è interamente riconducibile al problema noto del Knapsack 0-1. Per riassumere rapidamente, ci vengono dati

- Uno zaino di capienza B
- N oggetti di peso $A[i]$

IL problema del *knapsack 0-1* consiste nel capire se è possibile riempire lo zaino con gli oggetti dati in modo tale che il peso totale sia esattamente B. Ciascun oggetto può essere selezionato 0 o 1 volte

8.10.2 Knapsack e ricerca lineare

Nel nostro caso possiamo definire il $k-$esimo sotto-problema come il seguente:

- Dato un budget B
- Dato un numero massimo di portate per ogni piatto R
- Data una lista di prezzi dei diversi tipi di sushi A

Verificare se sia possibile spendere esattamente B mangiando al più R portate di ogni piatto

Supponiamo di avere $A = [2, 3, 8]$ e $B = 14$. Inizialmente chiediamoci se sia possibile raggiungere il budget con $R = 1$, ossia massimo un pezzo di sushi per ciascun tipo

$$\begin{bmatrix} & 0 & 1 & 2 & \dots & 13 & 14 \\ -1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 2 & 0 & 0 & 1 & \dots & 0 & 0 \\ 3 & 0 & 0 & 1 & \dots & 0 & 0 \\ 8 & 0 & 0 & 1 & \dots & 1 & 0 \end{bmatrix}$$

nell'ultima cella in basso a destra troveremo 1 nel caso in cui fosse possibile raggiungere il budget B con massimo R piatti per ogni tipo, 0 altrimenti. L'idea è di procedere aumentando R di uno alla volta fino a quando non troviamo un valore di R per cui è

possibile ottenere un budget B. Se non troviamo un valore di R minore di B allora significa che non è possibile ottenere il budget in alcun modo.

Procediamo ora con il calcolo per $R = 2$. L'idea è la stessa, però aggiungiamo ogni tipo di piatto 2 volte; con $A = [2, 3, 8]$ e $B = 14$ avremmo gli oggetti $[2, 2, 3, 3, 8, 8]$. Possiamo risolvere con una matrice `dp` in modo analogo:

$$\begin{bmatrix} & 0 & 1 & 2 & \dots & 13 & 14 \\ -1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 2 & 0 & 0 & 1 & \dots & 0 & 0 \\ 3 & 0 & 0 & 1 & \dots & 0 & 0 \\ 8 & 0 & 0 & 1 & \dots & 1 & 0 \\ 2 & 0 & 0 & 1 & \dots & 1 & 0 \\ 3 & 0 & 0 & 1 & \dots & 1 & 1 \\ 8 & 0 & 0 & 1 & \dots & 1 & 1 \end{bmatrix}$$

In questo caso la cella in basso a destra contiene 1, possiamo quindi terminare. Abbiamo ottenuto il risultato: $R = 2$.

Nota bene! Per ogni nuovo valore di R non è necessario ricalcolare l'intera tabella `dp`, ma è sufficiente aggiungere una portata per ciascun piatto all'ultima tabella `dp` e riprendere la computazione dall'ultima riga:

$$\begin{bmatrix} & 0 & 1 & 2 & \dots & 13 & 14 \\ -1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 2 & 0 & 0 & 1 & \dots & 0 & 0 \\ 3 & 0 & 0 & 1 & \dots & 0 & 0 \\ 8 & 0 & 0 & 1 & \dots & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} & 0 & 1 & 2 & \dots & 13 & 14 \\ -1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 2 & 0 & 0 & 1 & \dots & 0 & 0 \\ 3 & 0 & 0 & 1 & \dots & 0 & 0 \\ 8 & 0 & 0 & 1 & \dots & 1 & 0 \\ 2 & 0 & 0 & 1 & \dots & 1 & 0 \\ 3 & 0 & 0 & 1 & \dots & 1 & 1 \\ 8 & 0 & 0 & 1 & \dots & 1 & 1 \end{bmatrix}$$

Le prime righe rimangono invariate!

Per la computazione della matrice `dp` è opportuno inoltre usare la tecnica per ridurre lo spazio: ogni volta teniamo salvate solo le ultime 2 righe della matrice, in quanto per calcolare la riga i-esima bi bastano i valori della precedente

Complessità totale: $O(N \cdot R \cdot B) = O(N \cdot B^2)$, siccome $R = O(B)$

8.10.2 Knapsack e ricerca esponenziale/binaria

Sfortunatamente l'approccio precedente non è sufficientemente efficiente per ottenere il full score. Per questo motivo possiamo procedere con una ricerca esponenziale/binaria sul valore di R. Partiamo elencando un piccolo teorema matematico, chiave per la soluzione efficiente:

Teorema 1: *Somma potenze di due*

Dato R, k, t tali che:

$$R = 1 + 2 + 4 + \ldots + 2^k + t \quad 0 \leq t < 2^{k+1}$$

per ogni $0 \leq x \leq R$ esiste $S \subseteq \{1, 2, 4, \ldots, 2^k, t\}$ tale che $\text{sum}(S) = x$.

L'idea è che posso raggruppare gli oggetti del knapsack in gruppi dati dalle potenze di 2 più un fattore di "resto" t. Questo perché ogni peso che potrei ottenere usando R oggetti separati, è ottenibile tramite diverse combinazioni di oggetti raggruppati per potenze di due

Formalmente questa proprietà si può dimostrare per induzione. Per semplicità dimostriamo il caso in cui $t = 0$ e poi estendiamo ad un generico t

- Caso base: $R = 2^0$: posso ottenere ogni numero $\in [0, R]$
  - $0 = \text{sum}(\emptyset)$
  - $1 = \text{sum}(\{2^0\})$
- Supponendo per ipotesi induttiva che la proprietà sia vera per ogni $x < k$ allora possiamo dimostrare facilmente che il teorema vale anche per k. Dato

  $$R = \underbrace{1 + 2 + 4 + \ldots + 2^{k-1}}_{} + 2^k$$

  con le potenze $2^0, \ldots, 2^{k-1}$ posso, per ipotesi induttiva ottenere tutti i numeri fino a $\sum_{k=0}^{k-1} 2^k$. Questa è una nota *successione geometrica*:

  $$\sum_{k=0}^{k-1} 2^k = 2^k - 1$$

  Dunque posso ottenere i numeri in $\left[0, 2^k - 1\right]$ con le prime $k - 1$ potenze, mentre i numeri nel range $\left[2^k, R\right]$ aggiungendo la potenza 2^k
- Per questo motivo la proprietà vale per ogni $k \in \mathbb{N}$

Per quanto riguarda la costante t si può ragionare in maniera analoga a quanto fatto nello step induttivo. Sappiamo che per $R = 2^0 + \ldots + 2^k$ possiamo ottenere ogni somma in $[0, R]$ ossia in $\left[0, 2^{k+1} - 1\right]$. Quindi a patto che $t \leq 2^{k+1} - 1$ possiamo ottenere ogni somma in $[0, R + t]$ se combinando gli elementi in $\left\{2^0, \ldots, 2^k\right\}$ con t. La condizione sul range di t serve in quanto se $t > 2^k - 1$ non riuscirei ad ottenere i nueri in $\left[2^{k+1}, t - 1\right]$ (si creerebbe un "buco" in quanto t sarebbe troppo grande e i numeri ottenuti combinando $\left\{2^0, \ldots, 2^k\right\}$ con t partirebbero troppo dopo quelli ottenuti esclusivamente da $\left\{2^0, \ldots, 2^k\right\}$)

Per il nostro problema questo teorema è fondamentale in quanto ci permette di inserire ogni volta invece che una singola portata per ciascun tipo di sushi, un gruppo di portate dato dalle potenze di 2 più un eventuale "resto" t. Questo possiamo farlo proprio perchè abbiamo la certezza per il teorema 8.10.2 che ogni somma ottenibile con R portate è ottenibile anche con questo raggruppamento. Nel pratico inseriamo oggetti di peso $2^k \cdot A[i]$

ad ogni iterazione del knapsack. Supponendo di avere $A = [2, 3, 6]$ allora gli oggetti inseriti daranno:

$$[\underbrace{2, 3, 4,}_{R=1} \underbrace{4, 6, 8,}_{R=2} \underbrace{6, 12, 16,}_{R=4} , \underbrace{16, 24, 32}_{R=8}]$$

Così facendo possiamo individuare il range entro il quale R giace in maniera efficiente. In particolare, per individuare il range sono sufficienti $O(\log R)$ iterazioni di knapsack. Una volta ottenuto il range di valori di R entro il quale giace la soluzione possiamo eseguire una ricerca binaria. Questo secondo step è reso possibile dalla costante t indicata nel teorema. Andremo ad inserire oggetti di peso $t \cdot A[i]$. Ancora una volta siamo sicuri di poter ottenere tutte le possibili somme che potremmo ottenere inserendo ogni piatto singolarmente

Algoritmo 5: *Sushi exponential/binary search*

```
int sushi(int N, int B, int A[]):
    R_lower, R_upper = exponential_search(N, B, A);
    R = binary_search(N, B, A, R_lower R_upper);
    return R;
```

Vediamo ora uno pseudocodice che contiene un po' di informazioni in più riguardo all'implementazione dei due tipi di ricerca

Algoritmo 6: *Sushi exponential/binary search in depth*

```
int sushi(int N, int B, int A[]):
    prev_buf ← new bool [B + 1] = [0, ..., 0];
    curr_buf ← new bool [B + 1] = [0, ..., 0];
    latest_buf ← new bool [B + 1] = [0, ..., 0];

    R, latest = exponential_search(N, B, A);

    ▷ Lower and upper bounds for binary search of parameter t
    R_lower ← 1;
    R_upper ← R + 1;

    R = binary_search(N, B, A, R_lower, R_upper, latest);
    return R;
```

Seguono spiegazione e pseudocodice per le funzioni `exponential_search` e `binary_search`

Algoritmo 7: *Sushi exponential search*

```
(int, bool []) exponential_search(int N, int B, int A[]):
    prev_buf [0] ← 1;
    block_size ← 1 ;

    ▷ Ricerca esponenziale del range in cui giace R
    while true do
        latest ← prev_dp ;                                   ▷ Deep copy
        for i ← 0 to N − 1 do
            weight ← block_size · A[i];
            for j ← 1 to B do
                opt_1 ← prev_dp[j];
                opt_2 ← if weight > j then 0 else prev_dp[j − weight];
                curr_dp[j] ← opt_1 or opt_2;
            swap(prev_dp, curr_dp);

        ▷ Check if capacity B can be formed
        if prev_dp[B] then
            return (R, latest);
        else if 2 · block_size − 1 ≥ B then
            return  (−1, []);
        block_size ← 2 · block_size ;
```

Questa funzione ritorna $R = \sum_{i=0}^{k} 2^i$ dove k è il primo valore che risolve il problema. Intuitivamente aggiungiamo 2^i a R fino a quanto non riusciamo a risolvere il problema. Dunque otterremo un *upper_bound* e *lower_bound* della soluzione. Il vettore latest contiene i valori dell'ultima iterazione del knapsack per $R = 2^0 + \ldots + 2^{i-1}$, in maniera tale che non dobbiamo ricalsolarli per lo step di binary search.

Algoritmo 8: *Sushi binary search*

```
(int, bool []) binary_search(int N, int B, int A[], int R_lower, int R_upper,
bool latest []):
    while lower_bound < upper_bound do
        prev_dp ← latest ;                                   ▷ Deep copy
        mid ← ⌊(lower_bound+upper_bound)/2⌋;
        for i ← 0 to N − 1 do
            weight ← mid · A[i];
            for j ← 1 to B do
                opt_1 ← prev_dp[j];
                opt_2 ← if weight > j then 0 else prev_dp[j − weight];
                curr_dp[j] ← opt_1 or opt_2;
            swap(prev_dp, curr_dp);
        if prev_dp[B] then
            upper_bound ← mid;
        else
            lower_bound ← mid + 1;
    return block_size − 1 + lower_bound;
```

In questo caso procediamo secondo una ricerca binaria per valori di R fra *lower_bound* e *upper_bound*, ricercando, di fatto, il valore di t relativo al teorema 8.10.2. Questo è ancora una volta giustificato dal fatto che prendendo il set di potenze di 2 con t $\left\{2^0, \ldots, 2^k, t\right\}$ possiamo ottenere tutti i numeri naturali in $[0, R_{upper}]$, perciò non stiamo "trascurando" possibili combinazioni

Abbamo dunque migliorato significamente la complessità, in quanto ora è sufficiente applicare il knapsack $O(\log(n))$ volte. Applicando questo algoritmo abbiamo ottenuto la seguente complessità:

Step	Complessità
Exponential search	$O(BN\log(R)) = O(BN\log(B))$
Binary search	$O(BN\log(R)) = O(BN\log(B))$

Complessità: $O(BN\log(B))$

8.10.2 Knapsack e ricerca esponenziale/binaria con bitset

Sfortunatamente la soluzione descritta in 8.10.2 non è ancora sufficiente per ottenere un full score. Possiamo ulteriormente ottimizzare il calcolo di knapsack utilizzando un `bitset`. Questa struttura dati è di fatto un'`hash_set` di dimensione statica, le cui chiavi sono interi nel range $[0, n]$, dove n è la dimensione del bitset. Internamente viene memorizzato un array di `uint64_t`, dove ogni bit viene acceso singolarmente tramite operazione bitwise.

In particolare ci serve conoscere le operazioni di *bitwise or* e *shift logico*

Bitwise or

Supponiamo di avere un insieme di bit dato da

$$S_1 = [0, 0, 1, 0, 1, 0, 0, 0, 1]$$
$$S_2 = [1, 0, 0, 1, 0, 0, 1, 0, 0]$$
$$S_1|S_2 = [1, 0, 1, 1, 1, 0, 1, 0, 1]$$

L'idea è dunque quella di applicare l'or logico per ogni bit ("*bit a bit*")

Shift logico Lo shift logico è un'operazione che permette di spostare tutti i bit di un certo numero di posizioni a sinistra o a destra. Ad esempio

$$S_1 = [0, 0, 1, 0, 1, 0, 0, 0, 1]$$
$$S_1 << 2 = [1, 0, 1, 0, 0, 0, 1, 0, 0]$$
$$S_1 >> 3 = [0, 0, 0, 0, 0, 1, 0, 1, 0]$$

Quindi sposto i bit a sinistra/destra di un certo numero di posizioni, riempiendo con zeri le posizioni vuote

Ora analizziamo l'operazione fondamendale del knapsack, ossia quella in cui calcoliamo la $i-$esima fila della tabella:

$$curr\_dp\,[j] = prev\_dp\,[j]\;||\;prev\_dp\,[j - weight]$$

Questo può essere interpretato come segue: la cella $i-$esima del vettore dp è accesa se la cella i del vettore *prev_dp* è accesa oppure se è accessa la cella *weight* celle più avanti. Questo può essere tradotto con la seguente operazione bitwise

$$curr\_dp\,[j] = prev\_dp\,[j]\;||\;prev\_dp << weight$$

Dunque il calcolo del knapsack, sfruttando i bitset diventa il seguente:

$S2 = S1$;
for int $a : A$ **do**
  $weight \leftarrow a * R$;
  $S2| = S2 << weight$;

Dal punto di vista teorico, questa funzione ha la stessa complessità della versione senza `bitset`, tuttavia è 64 volte inferiore:

Con `bitset`	Senza `bitset`
$O\left(\frac{NB}{64}\right)$	$O\left(NB\right)$

Applicando questa ottimizzazione possiamo ottenere un full score! L'implementazione in `cpp` è disponibile in `files/esercizi/sushi/v4`
Possiamo entrare in classifica utilizzando un'implementazione custom del `bitset`. In particolare, l'operazione:

```
S2 |= S2 << (a * R)
```

risulta inefficiente per il fatto che `S2 << (a * R)` crea un nuovo `bitset`, eseguendo una deep copy. Tuttavia l'intera operazione può essere eseguita in place.

```
template <size_t N> class masked_bitset {
  static constexpr size_t num_words = size_t(N + 63) / 64;
  uint64_t buf[num_words]{};

public:
  // Set ith bit
  void set(size_t i) { buf[i / 64] |= (1ULL << (i % 64)); }

  // Unset ith bit
  void unset(size_t i) { buf[i / 64] &= ~(1ULL << (i % 64)); }

  // Test ith bit
  bool test(size_t i) const { return (buf[i / 64] & (1ULL << (i % 64)))
    != 0; }

  void offset_or(size_t offset) {
    // Basically does this: *this |= (*this << offset);
    if (offset >= N)
      return;
    size_t word_shift = offset / 64;
    size_t bit_shift = offset % 64;
    bool zero_shif_correction = bit_shift == 0;
    for (size_t i = num_words; i-- > word_shift + 1;) {
      // std::cout << i << std::endl;
      uint64_t high = buf[i - word_shift] << bit_shift;
      uint64_t low =
          buf[i - word_shift - 1 + zero_shif_correction] >> (64 -
   bit_shift);
      buf[i] |= (high | low);
    }

    buf[word_shift] |= buf[0] << bit_shift;
  }
```

In particolare l'operazione `S2 |= S2 << (a * R)` è implementata in place dal metodo `offset_or`. Questo piccolo accorgimento ci permette raggiungere la prima posizione in classifica (almeno per il momento). Questa soluzione è implementata in `files/esercizi/sushi/v5`