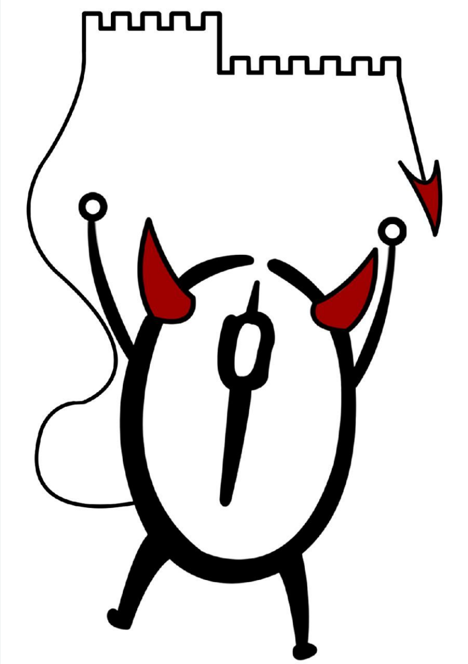
Olimpiadi Italiane di Informatica

OII 2018 - Campobasso
ITST G. Marconi
13 - 15 settembre 2018

Analisi di gara e spiegazione delle soluzioni



Un po' di statistiche

numero di partecipanti	90
soluzioni inviate	2354
soluzioni in C	69
soluzioni in C++	1461
soluzioni con punteggio pieno	2
punteggio medio	78,7
soluzioni inviate con submit	961
caffè bevuti dallo staff	+∞

Keywords

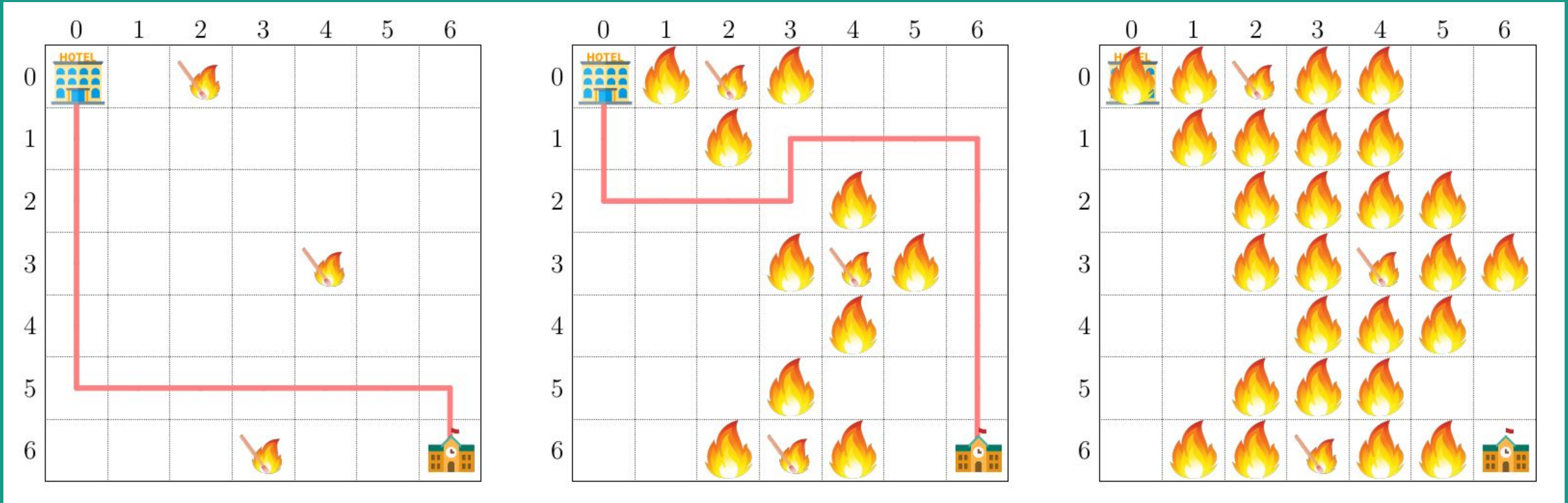
#1	[	1042309
#2	]	1042298
#3	(	37900
#4	)	37884
#5	{	21689
#6	}	21664
#7	int	18653
#8	if	12229
#9	else	2043
#32767	OII2018	0

Problemi in gara

- Incendio
- Cena
- Circuiti

Incendio

Punteggio medio	18.34/100
Miglior risultato	Michael Chelli (3h43min)
Primo score	Andrea Lavarone (18min)
Risolto da	2/90



Simulazione naïve (20/100)

Tempo: $O(N^3)$
Memoria: $O(N^2)$

Si simula l'evoluzione degli incendi in una griglia **NxN**:

Con una ricerca in profondità si controlla se esiste un percorso sicuro da **(0,0)** a **(N-1,N-1)** in tempo $O(N^2)$

Si espandono i fuochi nella griglia ripetendo il procedimento finchè possibile (al massimo **N** volte)

```
bool griglia[N][N];

for(int i=0; i<M; i++)
  griglia[X[i]][Y[i]] = true;

int sol = 0;

while(bfs(griglia, N)) {
  avanza(griglia, N);
  sol++;
}

return sol;
```

Simulazione ricerca binaria (42/100)

Tempo: $O(N^2 lg(N))$
Memoria: $O(N^2)$

È possibile sapere l'istante in cui ogni cella prenderà fuoco facendo partire una ricerca in profondità dagli incendi iniziali in tempo $O(N^2)$

Sapendo quando le celle prenderanno fuoco è facile sapere se si può arrivare alla sede dopo massimo **t** minuti

Si può trovare quindi il valore esatto con una ricerca binaria sul tempo

```
int tempi[N][N];

calcolaTempi(tempi, N, M, X, Y);

int lb = 0, ub = N+1;

while(lb+1 < ub) {
  int t = (lb+ub)/2;
  if(bfs(tempi, N, t)) lb = t;
  else ub = t;
}

return lb;
```

Un’osservazione cruciale

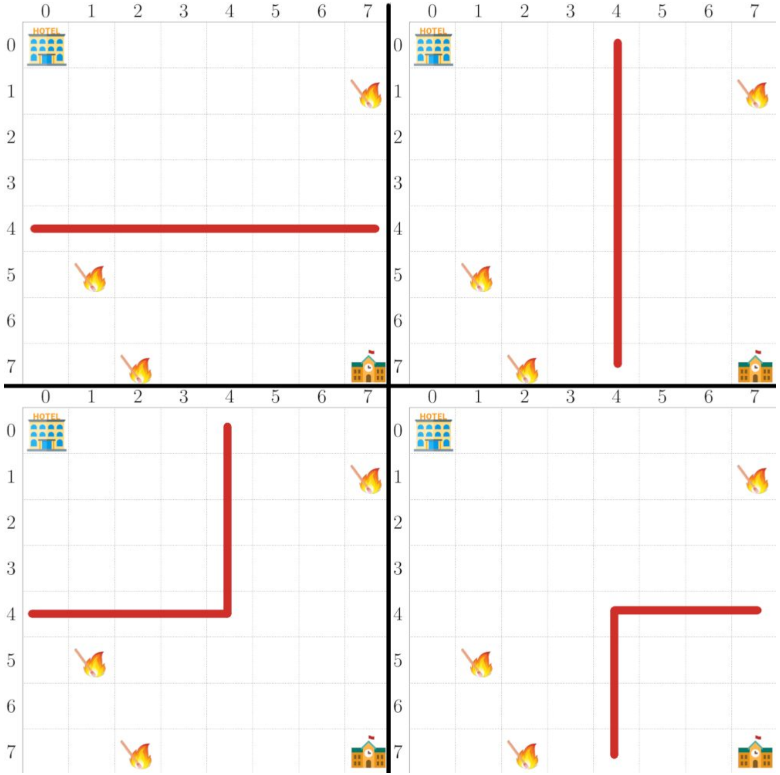
Quando non è più possibile arrivare alla sede di gara?

Finchè non esiste un percorso di celle incendiate che collega

uno dei muri tra **ovest e sud**

con

uno dei muri tra **est e nord**



L'altra osservazione importante

Quando due incendi entrano in contatto?

Consideriamo le differenze sui singoli assi:

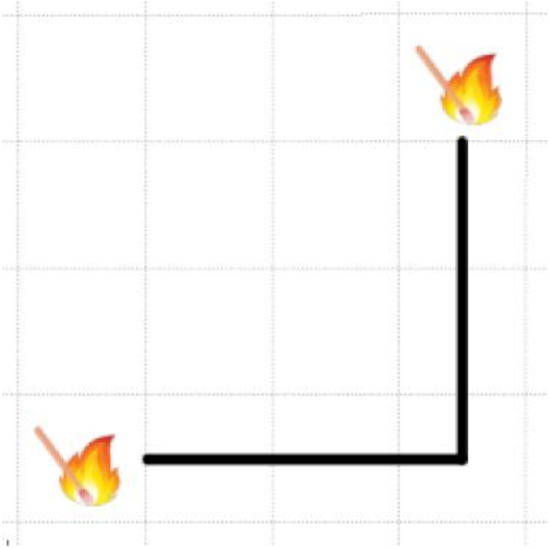
$$d_x = abs(x_1 - x_2)$$
$$d_y = abs(y_1 - y_2)$$

Se i due punti iniziali **sono allineati**:

$$dist_{1,2} = (d_x + d_y)/2$$

Se i due punti iniziali **non sono allineati**:

$$dist_{1,2} = (d_x + d_y - 1)/2$$



Widest path con dijkstra (83/100)

Tempo: $O(M^2 lg(M))$
Memoria: $O(M^2)$

Possiamo considerare gli incendi come un grafo completo:

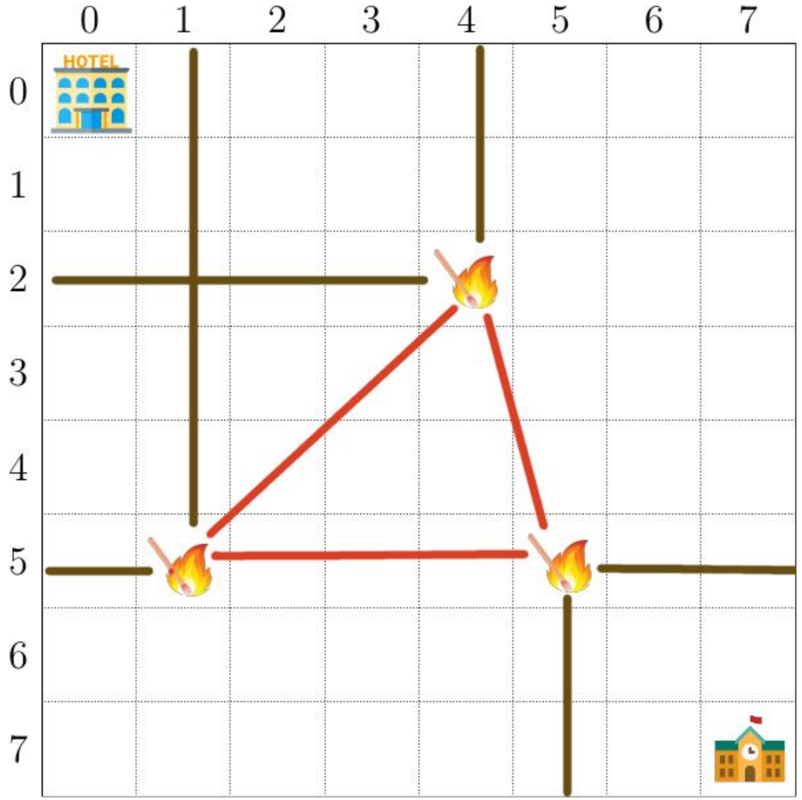
Ogni coppia di incendi `(i, j)` è collegata con arco di peso (distanza) **dist**$_{i,j}$

Non dimentichiamoci dei muri!

Ogni incendio è collegato col muro ovest/sud più vicino e con il muro est/nord più vicino

Soluzione:

Il percorso più *"largo"* dai muri ovest/sud fino ai muro est/nord



Togliere il lg(M) (92/100)

Tempo: $O(M^2)$
Memoria: $O(M^2)$

Implementando Dijkstra con una coda di priorità abbiamo una complessità totale di $O(E \lg(V))$

Nel nostro caso con grafo completo:

$$V = M \text{ e } E = M^2 \dashrightarrow O(M^2 \lg(M))$$

Per ridurre la complessità di dijkstra basta...

... non usare la coda di priorità

Ridurre la memoria (100/100)

Tempo: $O(M^2)$
Memoria: $O(M)$

Creando esplicitamente il grafo si ha un'occupazione di memoria $O(M^2)$:

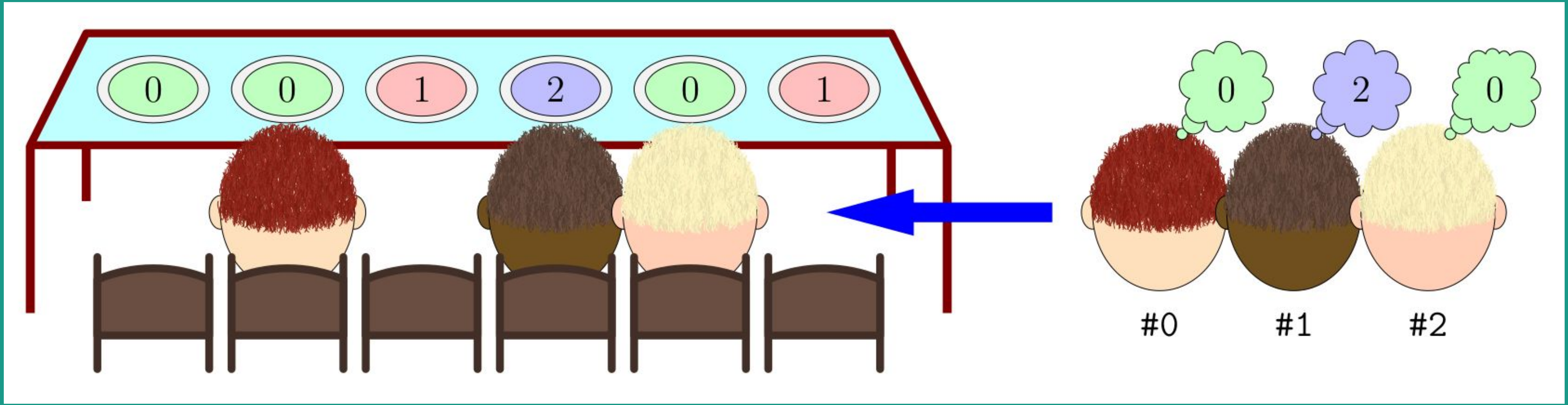
> Con M=10.000 la matrice di adiacenza occupa ~380MiB,
> Con M=12.000 si arriva fino a ~550Mib (andando in MLE)

È davvero necessario memorizzare esplicitamente?

No, si può calcolare in O(1) il peso di ogni arco quando mi serve (comunque massimo 2 volte per arco)

Cena

Punteggio medio	30.97/100
Primo punteggio massimo	Lorenzo Rossi 2 (1h11min)
Primo score	Alessandro Girardi (15min)
Risolto da	0/90



Una prima osservazione

Un problema più semplice, Mojito e Chupito non fanno nulla.

Come sapere in modo efficiente se è possibile disporre gli studenti?

È sufficiente mettere gli studenti più a sinistra possibile!

```
auto check = [&](int l, int r) {
    int j = 0;
    for (int i = l; i <= r && j < P; i++)
        if (s[i] == p[j])
            j++;
    return j == P;
};
```

Dobbiamo contare gli intervalli di una sequenza che contengono una determinata sottosequenza!

Soluzione naïve (13/100)

Tempo: $O(S^3)$
Memoria: $O(1)$

Per ogni possibile intervallo `(l, r)` provo a vedere se gli studenti si possono sedere usando l'algoritmo appena descritto.

```
long long sol = 0;
for (int l = 0; l < S; l++)
    for (int r = l; r < S; r++)
        if (check(l, r))
            sol++;
```

Un’ulteriore osservazione (63/100)

Tempo: O(S²)
Memoria: O(1)

Dato un intervallo `(l, r)` tale che `r` è il più piccolo per cui tutti gli studenti possano ancora sedersi, anche `(l, r+1)`, `(l, r+2)`, ..., `(l, S-1)` sono intervalli in cui ci si può sedere.

Il numero di questi intervalli è `S-r`.

Per ogni possibile inizio (O(S)) cerco di far sedere gli studenti più a sinistra possibile per trovare il corretto valore di `r` (O(S)).

Una soluzione quasi ottima

Tempo: O(SP)
Memoria: O(SP)

È possibile velocizzare la funzione cerca precomputando l'array **`next[C][S]`**

Dove **`next[c][i] = k`** indica che **k** è il primo indice che contiene **c** in **`s[i..S]`**

Con questo array è possibile cercare P in S con complessità O(P)

In questo modo miglioriamo la soluzione $O(S^2)$ abbassandola a O(SP)

```
// next[c][i] = posizione del prossimo c
// a partire dal carattere i
int next[100][S+1];
for(int c=0; c<100; c++){
  next[c][S] = S+1;
  for(int i=S-1; i >= 0; i--)
    if(s[i] == c)
      next[c][i] = i;
    else
      next[c][i] = next[c][i+1];
}

// Esiste p in s[l..S] ?
auto check = [&](int l){
  int i=0;
  while(k < S+1 && i < P)
    l = next[p[i++]][l]+1;
  return (P == i);
}
```

Programmazione dinamica

Tempo: O(SP)
Memoria: O(SP)

Restringiamoci a un suffisso **S[i..]** dei piatti e **P[j..]** dei partecipanti: **ce ne sono SP**.

Sia `Valid[i][j]` il minimo **k** tale per cui **P[j..]** è una sottosequenza di **S[i..j]**, oppure **S** se non c'è.

La soluzione del nostro problema è la somma di `S-Valid[i][0]` per ogni **i = 0..S-1**.

Calcoliamo ricorsivamente!

S[i..] vuoto: `Valid[S][j] = S` (niente)

P[j..] vuoto: `Valid[i][P] = i` (tutto)

Se `S[i] == P[j]`: `Valid[i][j] = Valid[i+1][j+1]`

Se `S[i] != P[j]`: `Valid[i][j] = Valid[i+1][j]`

Sfruttare le basse ripetizioni

Tempo: O(SA)
Memoria: O(SA)

Calcoliamo `Valid[i][j]` solo quando `S[i] = P[j]`.

Sfruttando `next[c][i]` calcolato precedentemente, `Valid[i][j]==Valid[k][j]` con `k=next[i][S[j]]` quando `S[i] != P[j]`.

Ricorsivamente: `Valid[i][j] = Valid[k][j+1]` dove `k = next[i][S[j+1]]`

Comprimiamo tali `Valid[i][j]` in `Vzip[i][n]`, dove **j** è l'**n**-esima occorrenza di **S[i]** in **P**.

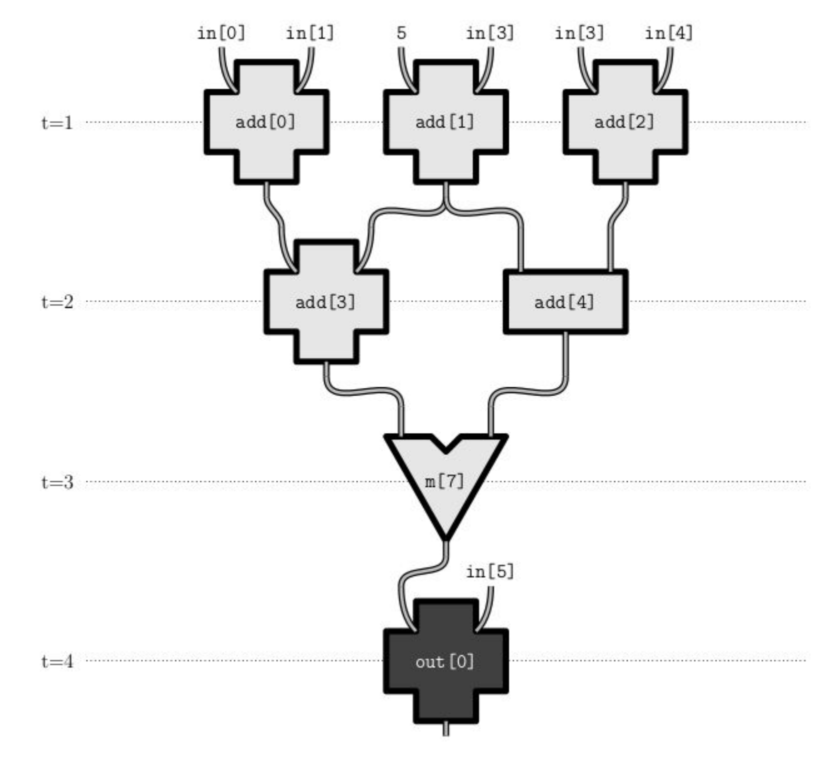
Calcoliamo anche `occ[c][i]` = **i**-esimo **c** in **P** e `index[i]` = numero di **P[i]** prima di **i** in **P**.

`Vzip[i][n] = Vzip[k][h]` dove:

`h = index[j+1]` e `j = occ[S[i]][n]`

Circuiti

Punteggio medio	29.46/100
Primo punteggio massimo	Federico Stazi (4h59min)
Primo score	Marco Rudelli (16min)
Risolto da	0/90



Introduzione

Abbiamo 3 problemi...

- Somma di un array
- Somme prefisse
- Massimo sottoarray

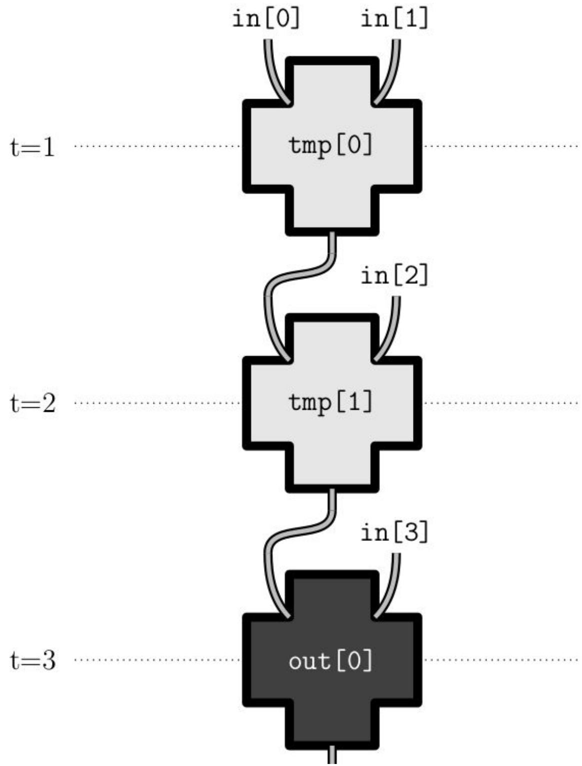
... analizziamoli separatamente!

tempo di calcolo: N
componenti: N

Somma di un array

Soluzione ovvia: da sinistra verso destra

```
tmp[0] = in[0] + in[1]
tmp[1] = tmp[0] + in[2]
...
out[0] = tmp[N-3] + in[N-1]
```



Somma di un array

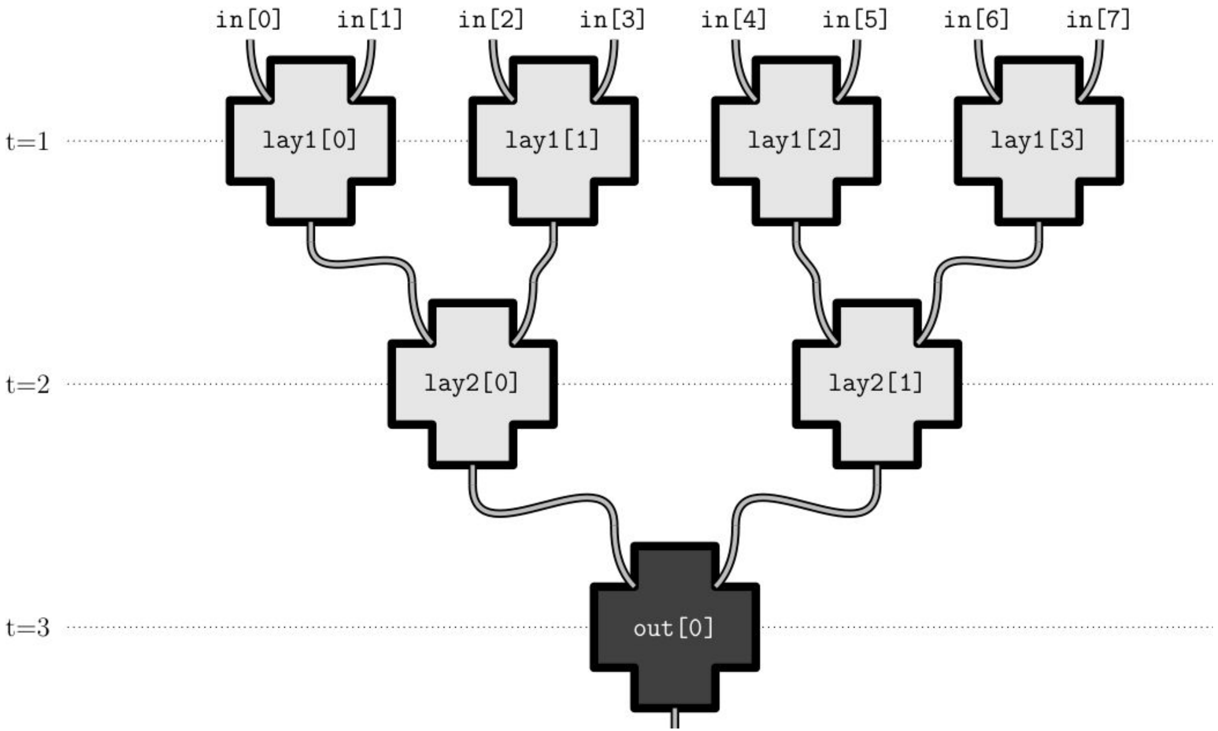
tempo di calcolo: log N
componenti: N

Sommando ad albero, si riduce il tempo di calcolo.

Questa operazione si chiama

REDUCE

Si calcola la somma degli intervalli `[i*2`^{k}`, (i+1)*2`^{k}`)` per tutti i valori di **i** e **k**

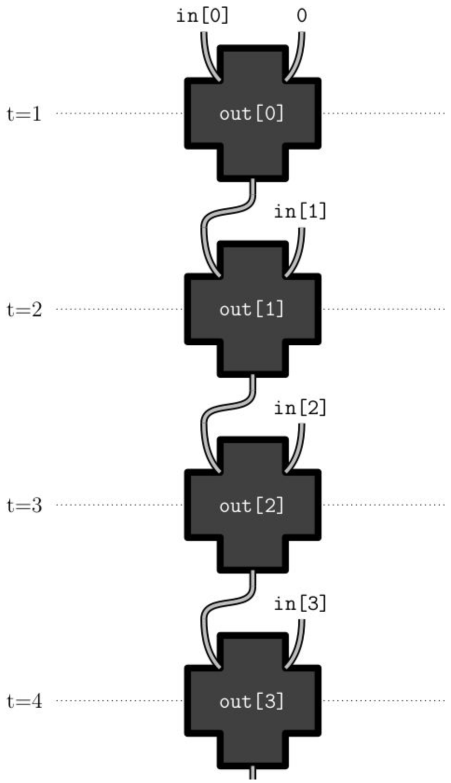


Somme prefisse

tempo di calcolo: N
componenti: N

Soluzione ovvia: da sinistra verso destra

```
out[0] = in[0] + 0
out[1] = in[1] + out[0]
out[2] = in[2] + out[1]
out[3] = in[3] + out[2]
...
```

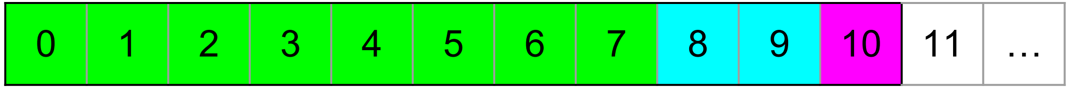


Somme prefisse

SCAN

Si calcola la somma degli intervalli `[0, i)` per ogni intero **i**.

Questo intervalli sono unione di pochi (log N) intervalli di tipo `[i*2^k, (i+1)*2^k)`



$$11_{10} = 1011_2$$

Esempio:

`[0,11)`

`= [0,8) U [8,10) U [10,11)`

$= [0*2^3,1*2^3)\ U\ [4*2^1,5*2^1)\ U\ [10*2^0,11*2^0)$

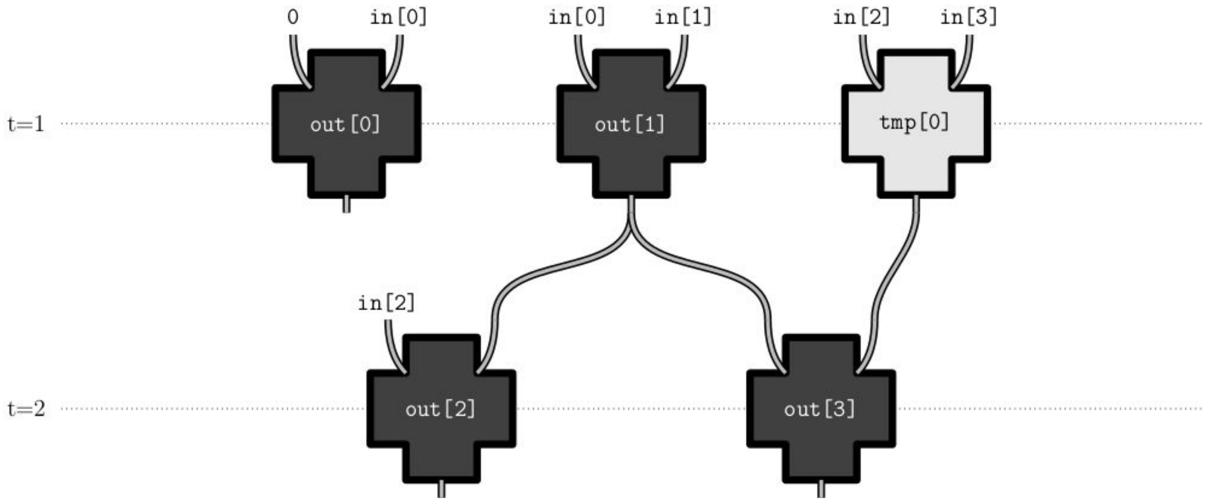
Somme prefisse

tempo di calcolo: log N
componenti: ?

SCAN

Si calcola la somma degli intervalli [0, i) per ogni intero **i**.

Questo intervalli sono unione di pochi (log N) intervalli di tipo [i*2^k, (i+1)*2^k)



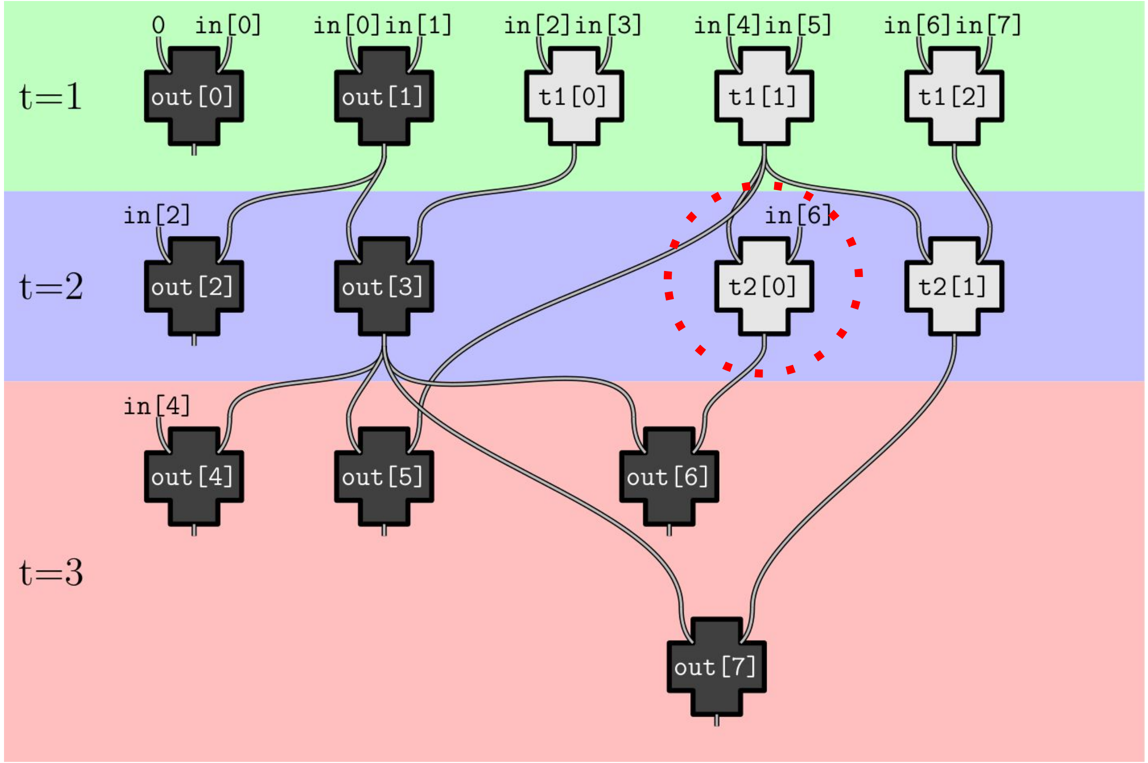
tempo di calcolo: log N
componenti: N log N

Somme prefisse

SCAN

Si calcola la somma degli intervalli `[0, i)` per ogni intero `i`.

Questo intervalli sono unione di pochi (log N) intervalli di tipo `[i*2^k, (i+1)*2^k)`



Somme prefisse

tempo di calcolo: log N
componenti: N

SCAN

Si calcola la somma degli intervalli `[0, i)` per ogni intero `i`.

Questo intervalli sono unione di pochi (log N) intervalli di tipo
$[i*2^k, (i+1)*2^k)$

Combinando gli intervallini da sinistra verso destra, si scende a O(N) componenti

Massimo sottoarray

Osservazioni:

1. `somma su [i,j) = somma su [0,j) - somma su [0,i)`

2. `max { somma su [i,j) | i < j } =`
   `max { somma su [0,j) - somma su [0,i) | i < j } =`
   `max { somma su [0,j) - min { somma su [0,i) | i < j } | j }`

Soluzione:

1. usare SCAN per calcolare `min { somma su [0,i) | i < j }` per ogni `j`
2. usare REDUCE per calcolare il **`max`**