# Preparazione alle Olimpiadi di Informatica

UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

## Programmazione dinamica

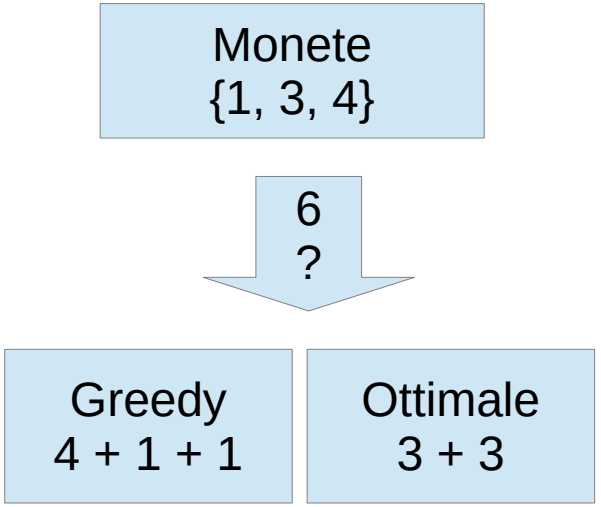
Mario Alviano

# Proprietà

- Il problema viene diviso in sottoproblemi
- Le soluzioni dei sottoproblemi vengono combinate
- Casi d’uso
  - Trovare una soluzione ottima
  - Contare il numero di soluzioni
- Vediamo un paio di esempi

# Problema delle monete (1)

Dato un insieme di (tagli di) monete (che include 1), qual è il numero minimo di monente per ottenere una data somma?




**Soluzione in 2 passi**

Formuliamo il problema
in termini ricorsivi

Usiamo **memoization**
per non risolvere più volte
lo stesso sottoproblema

# Problema delle monete (2)

Monete
{1, 3, 4}

| x | solve(x) | solution(x) |
|---|---|---|
| 0 | 0 | |
| 1 | 1 | 1 |
| 2 | 2 | 1 + 1 |
| 3 | 1 | 3 |
| 4 | 1 | 4 |
| 5 | 2 | 4 + 1 |
| 6 | 2 | 3 + 3 |

Come calcolare solve(x) ricorsivamente?

solve(x) := numero minimo di monete per ottenere x

# Problema delle monete (3)

- La scelta della prima moneta impone un sottoproblema
  - Se la prima moneta è 1, otteniamo x con 1 + solve(x-1) monete
  - Se la prima moneta è 3, otteniamo x con 1 + solve(x-3) monete
  - Se la prima moneta è 4, otteniamo x con 1 + solve(x-4) monete
- Prendiamo il minimo numero di monete
  - solve(x) = 1 + min(solve(x-1), solve(x-3), solve(x-4)), se x > 0
  - solve(0) = 0
  - solve(x) = ∞, se x < 0

# Problema delle monete (4)

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

**Inefficienza**

solve(10) richiede solve(7) e solve(6)

solve(7) richiede solve(6)

Come evitare di ricalcolare solve(6)?

# Memoization (1)

- Una tecnica generale per ottimizzare funzioni ricorsive
- I risultati della funzione ricorsiva vengono memorizzati

```
bool ready[N];
int value[N];
```

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}
```

# Memoization (2)

- Nel nostro esempio, l'array di int è sufficiente
  - inizializzare a -1 per indicare che il valore non è noto
- Se il dominio della funzione è ampio e sparso (o per semplicità), si può usare una mappa
  - unordered_map<int, int>

```
init({1, 3, 4});
cout << solve(6) << endl;
```

```
unordered_map<int, int> res;
vector<int> coins;

void init(const vector<int>& coins) {
    ::coins = coins;
    res.clear();
}

int solve(int x) {
    if(x < 0) return INT_MAX;
    if(x == 0) return 0;
    if(res.find(x) == res.end()) {
        DEBUG("Computing result for " << x);
        int best = INT_MAX;
        for(auto c : coins) best = min(best, solve(x - c));
        res[x] = 1 + best;
    }
    return res[x];
}
```

```
Computing result for 6
Computing result for 5
Computing result for 4
Computing result for 3
Computing result for 2
Computing result for 1
2
```

# Soluzione iterativa

- Approccio bottom-up
  - Si parte dai casi base
  - Si ottiene la soluzione per x dalle soluzioni precedenti

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

- La versione iterativa in genere è più efficiente di quella ricorsiva
- La versione ricorsiva in genere è più semplice (ma non sempre)

# Costruzione di una soluzione (1)

Monete
{1, 3, 4}

| x | solve(x) | solution(x) |
|---|---|---|
| 0 | 0 | |
| 1 | 1 | 1 |
| 2 | 2 | 1 + 1 |
| 3 | 1 | 3 |
| 4 | 1 | 4 |
| 5 | 2 | 4 + 1 |
| 6 | 2 | 3 + 3 |

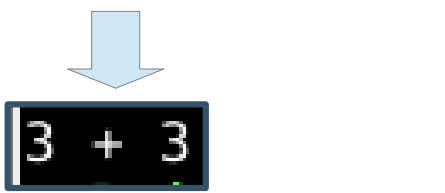
Come calcolare solution(x)?

solve(x) := numero minimo di monete per ottenere x

# Costruzione di una soluzione (2)

- Ogni soluzione aggiunge qualcosa alle soluzioni dei sottoproblemi
- Memorizziamo il delta
- Per il nostro esempio, memorizziamo la prima moneta

```
solution({1, 3, 4}, 6);
```




```
void solution(const vector<int>& coins, int x) {
    int res[x+1] = {0};
    int first_coin[x+1];
    for(int i = 1; i <= x; i++) {
        res[i] = INT_MAX;
        for(auto c : coins) {
            if(i - c >= 0 && 1 + res[i - c] < res[i]) {
                res[i] = 1 + res[i - c];
                first_coin[i] = c;
            }
        }
    }

    // print the solution
    cout << first_coin[x];
    for(;;) {
        x -= first_coin[x];
        if(x <= 0) break;
        cout << " + " << first_coin[x];
    }
    cout << endl;
}
```

# Contare le soluzioni (1)

- In quanti modi si può ottenere x = 5 da {1, 3, 4}?
  - 1 + 1 + 1 + 1 + 1
  - 1 + 1 + 3
  - 1 + 3 + 1
  - 3 + 1 + 1
  - 1 + 4
  - 4 + 1

# Contare le soluzioni (2)

Come calcolare count(x)?

Monete
{1, 3, 4}

| x | count(x) |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 4 |
| 5 | 6 |
| 6 | 9 |

**Soluzione**

Si sommano le soluzioni dei sottoproblemi

(Se i sottoproblemi non sono indipendenti, allora non è così semplice.)

count(x) := numero di modi per ottenere x

# Contare le soluzioni (3)

```
int count_solutions(const vector<int>& coins, int x) {
    int res[x+1] = {1};
    for(int i = 1; i <= x; i++) {
        for(auto c : coins) {
            if(i - c >= 0) res[i] += res[i - c];
        }
    }
    return res[x];
}
```

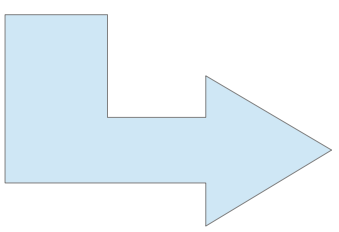
Se il numero di soluzioni è grande, in genere viene chiesto di fornire il risultato modulo M, ad esempio per M = $10^9+7$. In questi casi, aggiungere res[i] %= M; dopo ogni operazione.

# Longest Increasing Subsequence (1)

Dato un array, trovare una sequenza di elementi (da sinistra verso destra) di valore crescente e lunghezza massima.

Input

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 6 | 2 | 5 | 1 | 7 | 4 | 8 | 3 |

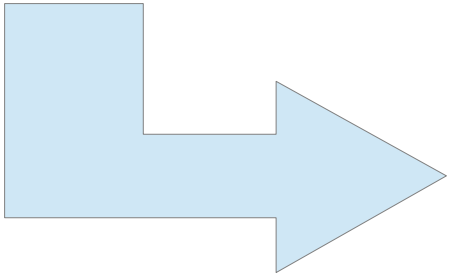


LIS (4 elementi)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 6 | 2 | 5 | 1 | 7 | 4 | 8 | 3 |

# Longest Increasing Subsequence (2)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 6 | 2 | 5 | 1 | 7 | 4 | 8 | 3 |



Funzione desiderata

$$
\begin{aligned}
\texttt{length}(0) &= 1 \\
\texttt{length}(1) &= 1 \\
\texttt{length}(2) &= 2 \\
\texttt{length}(3) &= 1 \\
\texttt{length}(4) &= 3 \\
\texttt{length}(5) &= 2 \\
\texttt{length}(6) &= 4 \\
\texttt{length}(7) &= 2
\end{aligned}
$$

length(k) := lunghezza della LIS che termina con l'elemento in posizione k

# Longest Increasing Subsequence (3)

- Per ogni k, proviamo a estendere le soluzioni ottime dei sottoproblemi

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k],length[i]+1);
        }
    }
}
```

$O(n^2)$

# Longest Increasing Subsequence (4)

- Manteniamo una lista di LIS di lunghezze diverse e con ultimo elemento minimo
  - Iniziamo con la LIS formata da v[0]
  - Se l'elemento v[i] è più grande di tutti gli ultimi elementi, cloniamo la LIS più grande e aggiungiamo v[i]
  - Altrimenti troviamo il più grande ultimo elemento minore di v[i], ed estendiamo questa LIS con v[i].

https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

# Longest Increasing Subsequence (5)

{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}.

```
A[0] = 0. Case 1. There are no active lists, create one.
0.
-----------------------------------------------------------
A[1] = 8. Case 2. Clone and extend.
0.
0, 8.
-----------------------------------------------------------
A[2] = 4. Case 3. Clone, extend and discard.
0.
0, 4.
0, 8. Discarded
-----------------------------------------------------------
A[3] = 12. Case 2. Clone and extend.
0.
0, 4.
0, 4, 12.
-----------------------------------------------------------
A[4] = 2. Case 3. Clone, extend and discard.
0.
0, 2.
0, 4. Discarded.
0, 4, 12.
```

```
A[5] = 10. Case 3. Clone, extend and discard.
0.
0, 2.
0, 2, 10.
0, 4, 12. Discarded.
-----------------------------------------------------------
A[6] = 6. Case 3. Clone, extend and discard.
0.
0, 2.
0, 2, 6.
0, 2, 10. Discarded.
-----------------------------------------------------------
A[7] = 14. Case 2. Clone and extend.
0.
0, 2.
0, 2, 6.
0, 2, 6, 14.
-----------------------------------------------------------
A[8] = 1. Case 3. Clone, extend and discard.
0.
0, 1.
0, 2. Discarded.
0, 2, 6.
0, 2, 6, 14.
```

```
A[9] = 9. Case 3. Clone, extend and discard.
0.
0, 1.
0, 2, 6.
0, 2, 6, 9.
0, 2, 6, 14. Discarded.
-----------------------------------------------------------
A[10] = 5. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 5.
0, 2, 6. Discarded.
0, 2, 6, 9.
-----------------------------------------------------------
A[11] = 13. Case 2. Clone and extend.
0.
0, 1.
0, 1, 5.
0, 2, 6, 9.
0, 2, 6, 9, 13.
-----------------------------------------------------------
A[12] = 3. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 1, 5. Discarded.
0, 2, 6, 9.
0, 2, 6, 9, 13.
```

```
A[13] = 11. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 2, 6, 9.
0, 2, 6, 9, 11.
0, 2, 6, 9, 13. Discarded.
-----------------------------------------------------------
A[14] = 7. Case 3. Clone, extend and discard.
0.
0, 1.
0, 1, 3.
0, 1, 3, 7.
0, 2, 6, 9. Discarded.
0, 2, 6, 9, 11.
-----------------------------------------------------------
A[15] = 15. Case 2. Clone and extend.
0.
0, 1.
0, 1, 3.
0, 1, 3, 7.
0, 2, 6, 9, 11.
0, 2, 6, 9, 11, 15. <-- LIS List
```

https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

# Longest Increasing Subsequence (6)

```
int lis(const vector<int>& v) {
    if(v.size() == 0) return 0;

    vector<int> tail;       // last element of each LIS
    tail.push_back(v[0]);   // start with the first element
    for(int i = 1; i < v.size(); i++) {
        if(v[i] > tail.back()) tail.push_back(v[i]);                // clone and extend
        else *lower_bound(tail.begin(), tail.end(), v[i]) = v[i];   // clone, extend and discard
    }

    return tail.size();
}
```

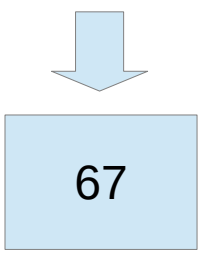
```
cout << lis({0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}) << endl;
```

https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

# Percorsi in una griglia (1)

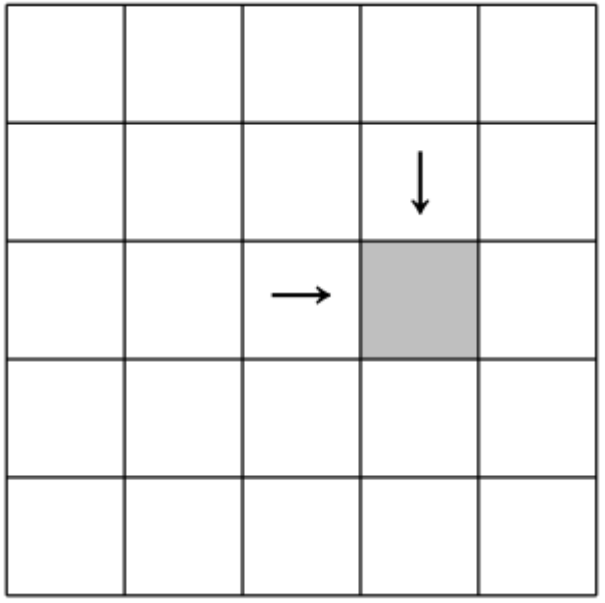
- Iniziamo dall’angolo in alto a sinistra
- Possiamo andare a destra e in basso
- Dobbiamo raggiungere l’angolo in basso a destra
- Vogliamo massimizzare la somma delle celle visitate

| | | | | |
|---|---|---|---|---|
| 3 | 7 | 9 | 2 | 7 |
| 9 | 8 | 3 | 5 | 5 |
| 1 | 7 | 9 | 8 | 5 |
| 3 | 8 | 6 | 4 | 10 |
| 6 | 3 | 9 | 7 | 8 |



# Percorsi in una griglia (2)

- Ogni cella può essere raggiunta o dall'alto o da sinistra



sum(y,x) := massima somma fino alla cella di riga y e colonna x

$$\mathtt{sum}(y,x) = \max(\mathtt{sum}(y,x-1), \mathtt{sum}(y-1,x)) + \mathtt{value}[y][x]$$

# Percorsi in una griglia (3)

- Iniziamo con una matrice di zeri
- Calcoliamo i valori di ogni sottoproblema
  - da sinistra a destra
  - dall'alto in basso

```
int sum[N][N];
```

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
    }
}
```

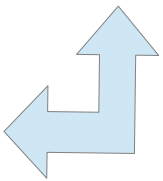
# Percorsi in una griglia (4)

- Stessa soluzione, ma con mappe

```
// store results in
//      int => (int => int)
unordered_map<int, unordered_map<int, int>> res;
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        res[i][j] = grid[i][j] + max(res[i-1][j], res[i][j-1]);
    }
}
cout << res[n-1][n-1] << endl;
```

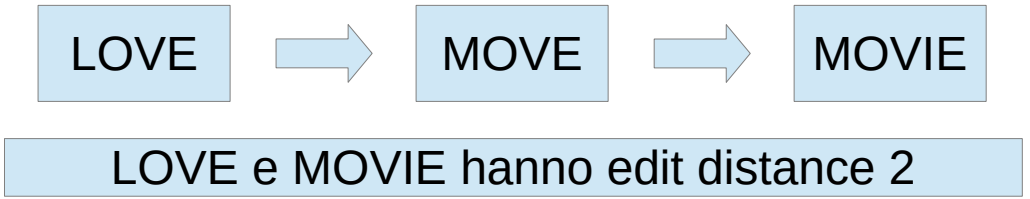
```
struct pair_hasher {
    size_t operator()(const pair<int, int>& x) const {
        return x.first * 119 + x.second;
    }
};
```

```
// store results in
//      (int,int) => int
unordered_map<pair<int, int>, int, pair_hasher> res;
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        res[{i,j}] = grid[i][j] + max(res[{i-1,j}], res[{i,j-1}]);
    }
}
cout << res[{n-1,n-1}] << endl;
```



# Edit Distance (1)

- Anche nota come Levenshtein distance
- Numero minimo di operazioni per trasformare una stringa in un’altra
  - inserire un carattere: da ABC a ABAC
  - rimuovere un carattere: da ABC a AC
  - modificare un carattere: da ABC a ADC



# Edit Distance (2)

**Input**

x := stringa di lunghezza n
y := stringa di lunghezza m

**Funzione ricorsiva per i prefissi di x e y**

distance(a,b) := edit distance fra $x_0 \ldots x_a$ e $y_0 \ldots y_b$

carattere inserito alla fine di x

carattere rimosso alla fine di y

modifica l'ultimo carattere, se diverso

$$\begin{aligned} \mathtt{distance}(a,b) = \min(&\mathtt{distance}(a,b-1)+1, \\ &\mathtt{distance}(a-1,b)+1, \\ &\mathtt{distance}(a-1,b-1)+\mathtt{cost}(a,b)) \end{aligned}$$

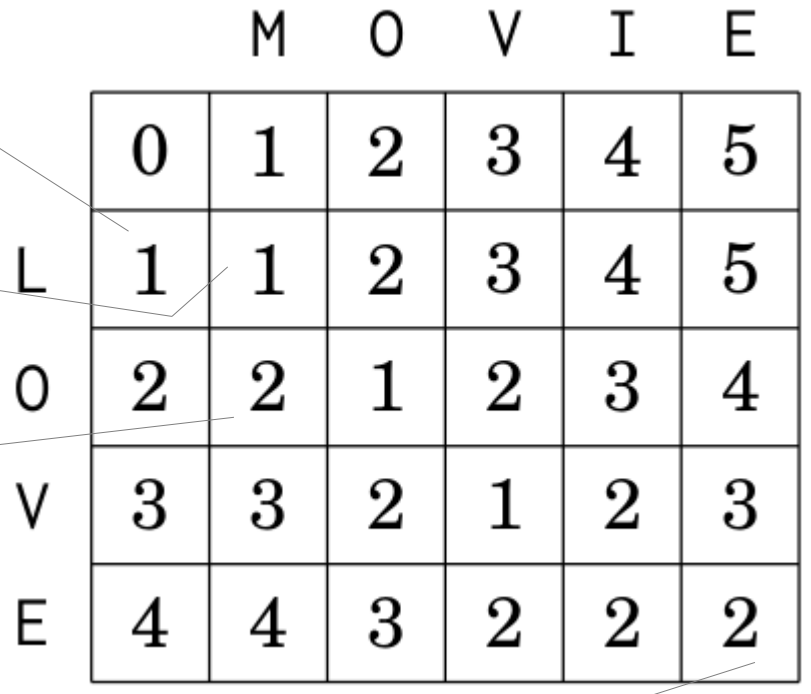
cost(a,b) := 1 se $x_a$ != $y_b$, 0 altrimenti

# Edit Distance (3)

L è a distanza 1 dalla stringa vuota

L è a distanza 1 da M

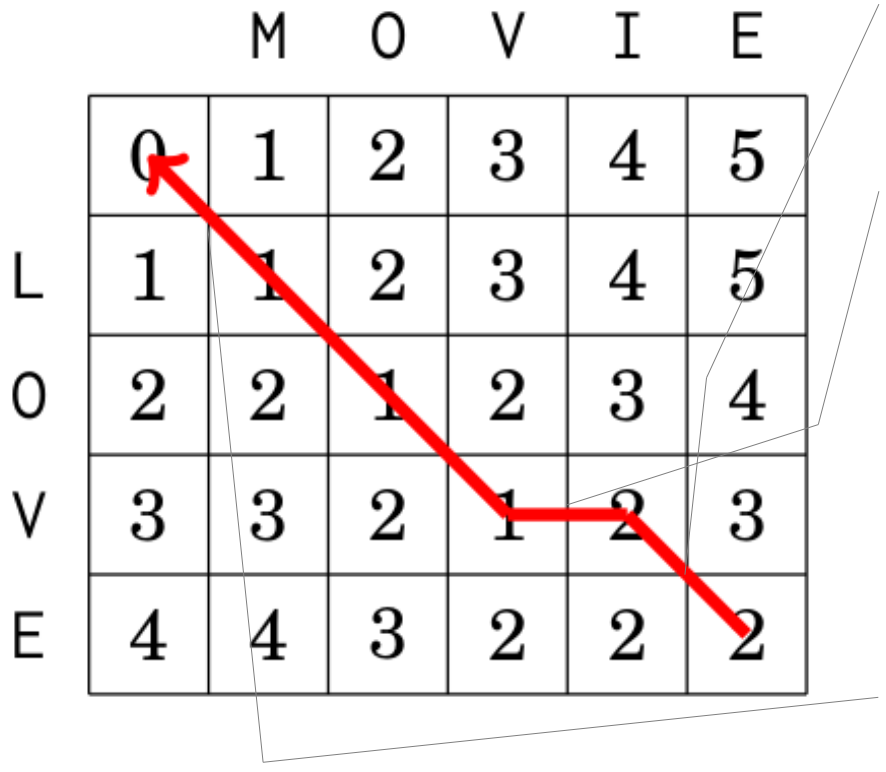
LO è a distanza 2 da M



| | | M | O | V | I | E |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| L | 1 | 1 | 2 | 3 | 4 | 5 |
| O | 2 | 2 | 1 | 2 | 3 | 4 |
| V | 3 | 3 | 2 | 1 | 2 | 3 |
| E | 4 | 4 | 3 | 2 | 2 | 2 |

LOVE è a distanza 2 da MOVIE

# Edit Distance (4)



| | | M | O | V | I | E |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| L | 1 | 1 | 2 | 3 | 4 | 5 |
| O | 2 | 2 | 1 | 2 | 3 | 4 |
| V | 3 | 3 | 2 | 1 | 2 | 3 |
| E | 4 | 4 | 3 | 2 | 2 | 2 |

LOVE e MOVIE hanno lo stesso terminatore
(nessuna operazione)

LOV e MOVI hanno edit distance uguale a
1 + edit distance di LOV e MOV
(aggiunta di I)

L e M hanno edit distance 1
(modifica di L in M)

# Edit Distance (5)

```
int edit_distance(const char* x, const char* y) {
    unordered_map<int, unordered_map<int, int>> res;
    function<int(int, int)> fun = [&](int a, int b) -> int {
        if(a < 0 || b < 0) return INT_MAX - 1;
        if(a == 0) return b;
        if(b == 0) return a;
        if(res[a].find(b) == res[a].end()) {
            res[a][b] = min(min(fun(a, b-1) + 1, fun(a-1, b) + 1),
                fun(a-1,b-1) + (x[a-1] == y[b-1] ? 0 : 1));
            DEBUG(a << " " << b << " => " << res[a][b]);
        }
        return res[a][b];
    };
    return fun(strlen(x), strlen(y));
}
```

```
cout << edit_distance("LOVE", "MOVIE") << endl;
```

# Esercizi suggeriti (1)

- Numeri di Figonacci (figonacci)
  - Variante dei numeri di Fibonacci
- Esame di maturità (esame)
  - Formulare il problema in termini ricorsivi (60 punti)
  - Usare memoization (100 punti)
- Missioni segrete (missioni)
  - Formulare il problema in termini ricorsivi (40 punti)
  - Usare memoization (100 punti)
- Assenza di gravità (gravity)
  - Formulare il problema in termini ricorsivi (30 punti)
  - Usare memoization (70 punti)
  - Usare una formulazione iterativa (100 punti)

# Esercizi suggeriti (2)

- Pranzo dalla nonna (nonna)
  - Formulare il problema in termini ricorsivi (60 punti)
  - Usare memoization (80 punti)
  - Generare le somme possibili (100 punti)
- Interstellar Trasmissions (seti)
  - Formulare il problema in termini ricorsivi (35 punti)
  - Usare memoization o forma iterativa (100 punti)
- Torri (torri)
  - Formulare il problema in termini ricorsivi (35 punti)
  - Usare memoization (100 punti)
- Server Provisioning (server)
  - Formulare il problema in termini ricorsivi (70 punti)
  - Usare memoization con perfect hash (100 punti)

# Esercizi suggeriti (3)

- La dieta di Poldo (poldo)
  - Longest Increasing Subsequence
- Corso per Sommelier (sommelier)
  - Molto simile a Longest Increasing Subsequence
- A matter of size (kabbalah)
  - Partire dall'angolo in basso a destra e per ogni cella calcolare la massima soluzione muovendosi solo verso destra e verso il basso
- Il triangolo (triangolo)
  - Partire dall'ultima riga e combinare le soluzioni ai sottoproblemi per ottenere le righe superiori
- Wonderful Grove (copac)
  - Partire dalle foglie dell'albero: costo = #nodi – 1
  - Combinare le soluzioni dei figli: costo = max(max(costo figli), #nodi rimanenti)

# Fine della lezione