Strategie algoritmiche

Luca Tagliavini

April 14-22, 2021

Contents

1 Divide et Impera

1.1 Torri di Hanoi

Le torri di Hanoi sono un problema matematico che consiste nello spostare dei dischi da un piolo all'altro, con alcune regole fisse:

- Si hanno tre pioli, i dischi sono inizialmente posti tutti sul piolo di sinistra
- n dischi, tutti di dimensioni diverse
- All'inizio tutti i dischi sono inpilati in ordine decrescente di diametro

Lo scopo del gioco e':

- Impilare in ordine decrescente i dischi sul piolo di destra
- Mai impilare un piolo piu' grande su uno piu' piccolo
- Muovere solo un disco alla volta
- Se serve usare anche il disco centrale

1.1.1 Soluzione divide et impera

```
algorithm hanoi(Stack p1, Stack p2, Stack p3, int n) -> void
  if n == 1 then
    // abbiamo solo un elemento nella prima stack, lo spostiamo nell'ultima
    // gioco risolto
    p3.push(p1.pop())
  else
    // sposto n-1 elementi da p1 a p2 appoggiandomi su p3
    hanoi(p1, p3, p2, n-1)
    // l'1 disco che rimane (sara' il piu' grande di tutti)
    // lo sposto da p1 a p3, dove sara' sicuramente nella
    // giusta posizione finale che ci aspettiamo
    p3.push(p1.pop())
    // sposto n-1 gli elementi da p2 a p3 appoggiandomi su p1
    // _vera_ chiamata ricorsiva che punta a risolvere il problema finale
    hanoi(p2, p1, p3, n-1)
  endif
endalgorithm
```

Divide: $n-1$ dischi da p_1 a p_2, 1 disco da p_1 a p_3, $n-1$ dischi da p_2 a p_3
Impera: applico l'algoritmo ricorsivamente sui sottoproblemi

Costo computazionale: $T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(n-1)+1 & \text{se } n > 1 \end{cases}$ Come risolvere questa equazione di ricorrenza?

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(2(Tn-2)+1)+1 \\
&= 2(2(2(Tn-2)+1)+1)+1 \\
&\dots \\
&= 2\cdot 2\cdot \ldots \cdot 2+1+\ldots+1 \\
&= 2^n+n
\end{aligned}
$$

1.2 Moltiplicazione di interi arbitrari in colonna

Consideriamo due interi di n cifre decimali, X, Y tali che:

$$
\begin{aligned}
X = x_{n-1}x_{n-2}\ldots x_1x_0 &= \sum_{i=0}^{n-1} x_i \cdot 10^i \\
Y = y_{n-1}y_{n-2}\ldots y_1x_0 &= \sum_{i=0}^{n-1} y_i \cdot 10^i
\end{aligned}
$$

Un approccio bruteforce applicando strettamente la "tecnica delle elementari" avrebbe un costo di $O(n^2)$, con il *divide et impera* possiamo invecere raggiungere un costo attorno a $O(n^\beta, \beta = 1.6$.

L'idea per l'implementazione divide et impera e' quella di dividere i due numeri di lunghezza n in altri numeri di lunghezza $n/2$ di questo tipo:

$$
\begin{aligned}
X &= X_1 \cdot 10^{n/2} + X_0 \\
Y &= Y_1 \cdot 10^{n/2} + Y_0
\end{aligned}
$$

Ad esempio 175123 diventera' $175 \cdot 10^3 + 123$. Il risultato che vogliamo ottenere sara' poi ottenibile tramite la formula:

$$
\begin{aligned}
X \cdot Y &= (X_1 \cdot 10^{n/2} + X_0) \cdot (Y_1 \cdot 10^{n/2} + Y_0) \\
&= (X_1Y_1) \cdot 10^n + (X_1Y_0 + X_0Y_1) \cdot 10^{n/2} + X_0Y_0
\end{aligned}
$$

L'algoritmo risolvera' dunque in modo ricorsivo la seguente equazione:

$$
X \cdot Y = (X_1Y_1) \cdot 10^n + (X_1Y_0 + X_0Y_1) \cdot 10^{n/2} + X_0Y_0
$$

- La moltiplicazione per 10^n richiede tempo $O(n)$ poiche' esegue uno shift di n bit

- Abbiamo poi 4 prodotti di numeri a $n/2$ cifre
- L'equazione di ricorrenza e' la seguente:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 4T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

  dove c_1 e c_2 sono costanti per il costo di ogni moltiplicazione
- Con il Master Theorem otteniamo un costo di $O(n^2)$, non abbiamo ancora migliorato l'algoritmo originale

1.2.1 Miglioramento

Poiche' svolgiamo la moltiplicazione 4 vole l'algoritmo *divide et impera* non e' conveniente. Tuttavia possiamo semplificare le operazioni necessarie:

Ponendo:

$$\begin{aligned} P_1 &= (X_1 + X_0) \cdot (Y_1 + Y_0) \\ P_2 &= (X_1 Y_1) \\ P_3 &= (X_0 Y_0) \end{aligned}$$

possiamo scrivere:

$$X \cdot Y = P_2 10^n + (P_1 - P_2 - P_3) \cdot 10^{n/2} + P_3$$

Ora otteniamo una relazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 3T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

E ora per il Master Theorem avremo un costo di $O(n^{1.6})$.

1.3 Sottovettore di valore massimo

Dato un vettore $v[1..n]$ si ha il seguente problema:

- Individuare un *sottovettore non vuoto* di v la cui somma degli elementi e' la massima possibile
- I vettori saranno

$$\begin{aligned} 1 &\text{ di lunghezza } n \\ 2 &\text{ di lunghezza } n-1 \\ 3 &\text{ di lunghezza } n-2 \\ &\vdots \\ k &\text{ di lunghezza } n-k+1 \\ &\vdots \\ n &\text{ di lunghezza } 1 \end{aligned}$$

per cui avremo un numero di vettori pari a $O(n(n-1)/2) = O(n^2)$

- per svolgere la somma degli elementi di un sottovettore, nel caso pessimo in cui il sottovettore e' quello grande n avremo un costo $O(n)$. Moltiplicandolo questo costo per il numero di vettori dara' un costo $O(n^3)$

1.3.1 Approccio brute force

```
algorithm max_sum(double v[1..n]) -> double
  double[] max <- v[1];
  for int i := 0 to n do
    for int j := 1 to n do
      double s <- 0
      for int k := i to j do
        s <- s + v[k]
      endfor
      if s > max
        max = s
    endfor
  endfor
  return max
endalgorithm
```

Costo: $O(n^3)$ poiche' si hanno 3 cicli for annidati.

1.3.2 Approccio brute force

Si puo' notare pero' che il calcolo della somma nella soluzione brute force e' superfluo in quanto possiamo tenerne traccia all'interno del secondo loop sommando a ogni iterazione il vallore dell'elemento $v[j]$

```
algorithm max_sum(double v[1..n]) -> double
  double[] max <- v[1];
  for int i := 0 to n do
    double s <- 0
    for int j := 1 to n do
      s <- s + v[j]
      if s > max
        max = s
    endfor
  endfor
  return max
endalgorithm
```

Costo: $O(n^2)$

1.3.3 Approccio divide et impera

Proviamo con la tecnica divide et impera iniziando dividendo il vettore in due sottovettori grandi $n/2$ e separati dall'elemento in posizione $mn/2$.

Il sottovettore di somma massima potrebbe torvarsi:

- interamente nella prima meta' $v[1..m-1]$
- interamente nella seconda meta' $v[m+1..n]$
- a cavallo tra le due meta'

La ricerca piu' difficile e' sicuramente quella a cavallo tra le due meta' e dunque quella che contiene anche $v[m]$. Una strategia puo' essere quella di leggere una serie di prefissi (elementi in $S_a = sum\{v[0..m-1]\}$) e suffissi (elementi in $S_b = sum\{v[m+1..n]\}$) di $v[m]$. A questo punto il massimo sara' dato da $v[m] + S_a + S_b$. Vediamone lo pseudocodice:

```
algorithm max_sum(double v[1..n], int i, n) -> double
  if i > n return 0
  elif i == n return v[i]
  else
    m <- floor((i+n)/2)
    dobule l = max_sum(v, i, m-1)
    dobule r = max_sum(v, m+1, n)
    dobule sa <- 0, sb <- 0, tmp_s <- 0;
    for k := m-1 to i do
      tmp_s <- tmp_s + v[k]
      if(tmp_s > sa) sa <- tmp_s
    endfor
    tmp_s <- 0
    for k := m+1 to n do
      tmp_s <- tmp_s + v[k]
      if(tmp_s > sb) sb <- tmp_s
    endfor
    return max(l, r, v[m] + sa + sb)
  endif
endalgorithm
```

L'equazione di ricorrenza e': $T(n) = 2T(n/2) + n$.
Tramite il master theorem otteniamo un costo di $O(n \log_2 n)$.

2 Algoritmi *greedy*

Quando tra le molte scelte se ne puo' identificare una migliore che sicuramente ci portera' alla soluzione ottimale e quando l'algoritmo ha una struttura ottima e dunque siamo sicuri che fatta tale scelta la struttura del problema non varia, possiamo usare la tecnica greedy, per rendere l'algoritmo piu' veloce. Non tutti gli algoritmi offrono scelte greedy e non sempre questa soluzione e' conveniente, ma ci sono problemi interessanti da analizzare.

2.1 Problema del resto

Riceviamo in input un numero intero r che rappresenta (in centesimi) il resto da dare in monete. Abbiamo a disposizioni solo i tagli di monete diponibili nella currency dell'euro. La soluzione piu' intuitiva che e' anche quella greedy e' quella di iniziare a dare i pezzi di moneta piu' grandi possibili, in tal modo infatti le chiamate ricorsive sul resto ancora da dare saranno sempre fatte con *la piu' piccola quantita' di resto ancora da dare*. Tuttavia questa tecnica greedy e' vantaggiosa solo su *sistemi monetari canonici*. Ecco alcuni controesempi:

- erogare 6 con monete: $[4, 3, 1]$ (greedy: $4, 1, 1$, ottimale: $3, 3$)
- erogare 6 con monete: $[5, 2]$ (greedy: 5 e poi fallisce, ottimale (e unica) $2, 2, 2$)

2.1.1 Soluzione greedy-intuitiva

```
algorithm change(int r, int t[1..n]) -> integer
  sort(t) // decreasing
  // nm = number of coins, i = index of the max
  // coin we're looking to give
  int nm <- 0, i <- 1
  while r > 0 and i < n
    if r >= t[i] then
      r <- r - t[i]
      nm <- nm + 1
    else
      i <- i + 1
    endif
  endwhile

  if r > 0 then
    // cannot give change with the given pieces
    return err
  else
    return nm
  endif
endalgorithm
```

2.2 Problema di scheduling

Abbiamo una serie di job da eseguire, ognuno con la sua durata, ed abbiamo un esecutore che puo' eseguirli in un ordine arbitrario. Il tempo impiegato e' invariabile in quanto dato dalla sommatoria di tutti i tempi, tuttavia in base all'ordine di esecuzione possiamo miglioreare il *tempo medio di completamento*. La cosa e' desiderabile assumendo che ogni lavoro compiuto ci dia un premio e vogliamo raggiungere piu' premi possibile nel minor tempo.

La scleta *greedy* e' anche quella piu' vantaggiosa, ossia: svolgere subito tutti i lavori che richiedono meno tempo e procedere mano a mano con quelli che impiegano sempre piu' tempo.

2.3 Problema della compressione (codifica di Huffman)

Vogliamo trovare la quantita' di bit necessaria per rappresentare una sequenza binaria in modo che essa sia la minore possibile, ossia che i file sia il piu' compresso possibile. Avremo tuttavia alcuni vincoli:

- Useremo una *funzione di codifica* che preso un carattere dell'alfabeto e ci restituisce una sequenza di caratteri per rappresentare tale carattere.
- Presa una sequenza di caratteri $c_1, c_2, \ldots, c_n$ verra' codificata come $f(c_1), f(c_2), \ldots, f(c_n)$.
- Una volta che una sequenza e' stata codificata dev'essere *sempre possibile decodificarla* tramite una lettura *sequenziale* (bit-after-bit) di tale codifica.

Data la sequenza $c_1, c_2, \ldots c_n$ potremo definire la nostra funzione di codifica in grado di **minimizzare** la lunghezza della nostra codifica.

Supponiamo di avere un alfabeto ristretto composto dalle sole lettere:

a,	b,c,	d,e,	f

che appaiono con le seguenti probabilita':

45%,	13%,12%,	16%,16%,	9%

2.3.1 Codici a lunghezza fissa

Le codifiche a lunghezza fissa sono quelle usate dagli standard come ASCII, UTF, etc.

Codifica tramite ASCII: usiamo 8 bit per ogni carattere, che ci da 2^8 diversi caratteri disponibili dove ogni carattere viene convertito in una sequenza di bit che rappresenta un numero.
Per rappresentare n caratteri serviranno $8 \cdot n$ bit.

Codifica basata sull'alfabeto: 3 bit per carattere, che ci da $2^3 = 8$ caratteri disponibili che bastano per i nostri 6 caratteri.

a,	b,c,	d,e,	f
000,	001,010,	011,100,	101

Per rappresentare n caratteri serviranno $3 \cdot n$ bit.

2.3.2 Codici a lunghezza variabile

Possiamo usare invece una codifica a lunghezza variabile in modo da rappresentare i caratteri piu' usati con codici univochi piu' corti (nel nostro esempio

la a) e codici meno usati con codifiche piu' lunghe. Il costo della codifica sara' dato da:

$$C(n) = \sum_{j=1}^{6} P(c_j) \cdot L(c_j) \cdot n = n \cdot \sum_{j=1}^{6} P(c_j) \cdot L(c_j)$$

Dove $P(c_j)$ e' la probabilita' che c_j appaia nella sequenza, $L(c_j)$ e' la lunghezza della codifica del carattere c_j, e c_j e' il j-esimo carattere del nostro alfabeto.

Nella nostra codifica d'esempio abbiamo un peso pari a $2.24n$.

Poiche' questa codifica (vedi slide) e' una cosiddetta *codifica senza prefissi* e' stata strutturata in modo che nessun codice possa diventare un prefisso di altre sequenze. Esempi:

- $a = 0, c = 1, b = 01$ si nota che la sequenza 01 puo' essere interpretata come ac che come b. Questa codifica e' ambigua e non puo' essere decodificata, proprio poiche' non vale la *proprieta' della codifica senza prefissi.*
- $a = 0, b = 10, c = 101$, si nota che 101 e' solo c e 10 e' solo b.

2.4 Codici di Huffman

Ha sviluppato un algoritmo e una relativa soluzione al problema di identificare la codifica piu' conveniente per un dato alfabeto con la relativa percentuale di apparizione. L'idea e' di rappresentare tutti i possibili caratteri **come foglie di un albero binario**. In questo modo tutti i padri delle foglie (che sono i nostri caratteri) avranno un valore binario $0 - 1$ in modo che ogni foglia abbia un percorso univoco per salire alla radice che ne determina anche la codifica finale.

Algoritmo di decodifica: Leggi dalla radice un percorso che va a destra se il bit e' 1 e a sinistra se il bit e' 0 fino a raggiungere una foglia che rappresenta il risultato della nostra decodifica.

Algoritmo di codifica: Partiamo dalla foglia che rappresenta il carattere di nostro interesse e saliamo fino alla radice leggendo i valori dei nodi su cui passiamo.

> Una proprieta' che ci torna molto comoda degli alberi, in quanto vogliamo avere sequenze di bit minimali, e' che se un nodo ha un solo figlio (destro o sinistro) quel nodo puo' essere rimosso in quanto e' superfluo e non rompe la proprieta' della codifica senza prefissi.

L'algoritmo di Huffman usa pesantemente questa idea della rappresentazione arborescente, e svolge alcune operazioni per creare l'albero necessario:

- si legge un file d'esempio e si legge la *frequenza* di tutti i caratteri
- si costruisce il codice sotto forma di albero
- si rappresenza il codice tramite la codifica

- lo si puo' leggere (decodificare) usando lo stesso albero

Per la costruzione dell'albero useremo una tecnica bottom-up, partendo dalle foglie e costruendo man mano il percorso fino alla radice.

1. il primo step sara' ordinare tutti i caratteri in una lista ordinata (crescente)
2. poiche' tutti i nodi hanno due figli, iniziero' prendendo i due caratteri che hanno la frequenza piu' bassa, in quanto saranno quelli che vorro' avere in assoluto piu' in basso. Una cosa interessante da notare e' che la frequenza del nodo creato viene data dalla frequenza delle sue due foglie (e in questo modo cosi' via ricorsivamente)
3. inserisco l'albero nella lista ordinata e la ordino. Cosi' potro' mantere la caratteristica *greedy* di questo algoritmo che prende alla cieca i primi due elementi della lista e li unisce in un nodo.
4. al secondo passaggio avremo gli $k-2$ caratteri restanti dallo step precedente e l'albero creato prima, che ha assunto la frequenza data da $c_1 + c_2$ come descritto al punto 2. Adesso si riesegue lo step 2.
5. procedendo cosi' in modo ricorsivo si ottiene l'albero desiderato.

Una volta ottenuto l'albero di ricorrenza possiamo dare valori binari $0-1$ a ogni nodo in modo da ottenere l'albero pronto affiche' sia usato dall'algoritmo di codifica.

```
struct TreeNode
  real f
  char c
  TreeNode l, r

algorithm hoffman(float f[], char c[]) -> Tree
  Q <- new MinPriorityQueue()
  for i := 1 to n do
    z <- new TreeNode(f[i], c[i])
    Q.insert(f[i], z)
  endfor
  for i := 1 to n-1 do
    z1 <- Q.pop();
    z2 <- Q.pop();
    z <- new TreeNode(z1.f + z2.f, '')
    z.left = z1;
    z.right = z2;
    Q.insert(z1.f + z2.f, z)
  endfor
  return Q.findMax()
endalgorithm
```

Il primo ciclo ha costo $n \log_2 n$, mentre il secondo $n \log_2 n$.
Costo: $\Theta(n \log_2 n + n \log_2 3n) = O(n \log_2 n)$

3 Programmazione dinamica

La programmazione dinamica e' una tecnica usata per trovare la soluzione di problemi che richiedono la *soluzione ottima* (vedi sottovettore massimo). Si dividera' il problema primario in sottoproblemi piu' semplici e si puntera' a risolverli tramite le tecniche migliori per ogni sottoproblema.

3.1 Differenze tra divide et impera e programmazione dimanica

Divide et impera utilizza una *tecnica ricorsiva* che mira a risolvere dei sottoproblemi **indipendenti** con un approccio *top-down*, partendo dunque dal problema iniziale e dividendolo in sottoproblemi piu' semplici, fino ad arrivare alle foglie (caso base) che risultano problemi banali dove non viene piu' usata la chiamata ricorsiva.

D'altro canto la programmazione dinamica risolve i problemi in ordine *bottom-up*, risolvendo prima i problemi banali, memorizzando le soluzioni trovate per i sottoproblemi semplici (in modo da non svolgere due volte lo stesso lavoro). Solitamente vengono scritti in forma *iterativa* e sfruttano sottostrutture ausiliarie per ricordarsi i risultati precedenti.
NOTA: la programmazione dinamica e' vantaggiosa solo quando sappiamo che un sottoproblema dovra' essere risolto piu' volte, e quindi conviene farne un *memoize*. In tal modo, quando andiamo a risolvere un sottoproblema riusciamo a farlo in modo efficiente poiche' ci basiamo sui risultati precedenti degli altri sottoproblemi.

3.1.1 Rivediamo Fibonacci

Scrivere un fibonacci in maniera ricorsiva ci da un costo computazionale di $O(2^n)$, poiche' molte chiamate base come $fib(1), fib(2), \ldots$ appaiono svariate volte, e ovviamente restituiscono sempre lo stesso valore. Converrebbe dunque mantenere questi risultati in memoria per evitare di ricomputarli.

Eccone una soluzione con il *memoizing* della serie di fibonacci:

```
algoritm fib(int n) -> int
  if n < 2 then
    return 1
  else
    int f[1..n]
    f[1] <- 1
    f[2] <- 1
    for int i <- 3 to n do
      f[i] = f[i-1] + f[i-2]
    endfor
    return f[n]
endalgorithm
```

Costo: $O(n)$
Occupazione di memoria: $O(n)$

3.2 Problema del sottovalore massimo

Identifichiamo il sottoproblema $P(i)$ che consiste nell'identificare il massimo della somma degli elementi dei sottovettori non vuoti del vettore $V[1..i]$ **che hanno $V[i]$ come ultimo elemento**.

La soluzione al problema del sottovettore massimo sara':

$$sol = \max\{P(1), P(2), \ldots, P(n)\}$$

La soluzione al sottoproblema base (sottovettore $V[1..1]$) e' elementare.
La soluzione al sottoproblema ricorsivo (sottovettore $V[1..i]$) consiste nel combinare la solzuione $P(i-1)$ e tenerla se essa e' positiva sommandola a $V[i]$, altrimenti scartarla e tenere solo $V[i]$. Possiamo esprimerlo maticamente tramite:

$$P(i) = \max\{V[i] + P(i-1), V[i]\}$$

```
algorithm max_subset(real V[1..n]) -> int
  real S[1..n]
  S <- V[1]
  int imax = 1
  for integer i <- 2 to n do
    if S[i-1] + V[i] then
      S[i] <- S[i-1] + V[i]
    else
      S[i] <- V[i]
    endif

    if S[i] > S[imax] then
      imax = i
    endif
  endfor
  return S[imax]
endalgorithm
```

Costo: $\Theta(n)$
Costo spazio: $\Theta(n)$ ma possiamo renderlo costante usando variabili e non un array

```
algorithm max_subset(real V[1..n]) -> int
  curr_max <- V[1], last_max <- V[1]
  for integer i <- 2 to n do
    if last_max + V[i] then
```

```
      curr_max <- curr_max + V[i]
    else
      curr_max <- V[i]
    endif

    if curr_max > last_max then
      last_max = curr_max
    endif
  endfor
  return last_max
endalgorithm
```

3.3 Problema dello zaino

Abbiamo un insieme $X[1..n]$ di oggetti, a cui viene associato un peso $P[1..n] \in \mathbb{N}^n$ come interi. Disponiamo poi un contenitore, in grado di trasportare al massimo un peso $P \in \mathbb{N}$. Il risultato sara' $Y \subseteq X$ tale che

- il peso di Y sara' $\leq$ del peso di P

$$\sum_{i=1}^{n} P[i] \leq P$$

- il valore complessivo degli oggetti in Y sia il massimo possibile entro il peso dato

3.3.1 Approccio greedy #1

La prima soluzione greedy che ci viene in mente consiste nel prendere subito l'elemento di valore maggiore (essendo greedy), ma potremo lasciare una serie di oggetti piu' piccoli che valgono molto di piu' messi assieme poiche' facciamo una scelta ingorda.

3.3.2 Approccio greedy #2

Una scelta piu' intelligente sarebbe valutare gli oggetti in base al valore che essi assumono relativo al peso(i.e. un oggetto che pesa 2 kg ma vale 25 danari deve avere precedenza rispetto a uno che pesa 4kg ma ne vale 30) che denomineremo **valore specifico**. Facendo una scelta greedy su questo coefficiente (che assumiamo di avere gia' generato, ma avrebbe comunque costo $\Theta(n)$) possiamo scrivere un algoritmo greedy che da risultati piu' itelligenti, ma non sempre ottimali.

Tuttavia anche questa soluzione non da sempre un risultato ottimale poiche' a volte si riempie lo zaino con elementi che hanno certamente un buon valore specifico, ma hanno un peso tale che impediscono l'aggiunta di altri elementi nel futuro, i quali avrebbero prodotto un miglior risultato.

3.3.3 Approccio dinamico

Dividiamo il problema principale in tanti sottoproblemi che consistono nel risolvere $P(i, j)$, che calcola il miglior sottoinsieme di elementi fino a i per riempire lo zaino di capienza j. Le soluzionie $V[i, j]$ conterranno il massimo valore ottenibile da un sottoinsieme degli oggetti $\{1, 2, \ldots, i\}$ in uno zaino di capacita' j (con $i = 1, 2, \ldots, n$ e $j = 0, 1, 2, \ldots P$).

Partiremo riempiendo la prima colonna del vettore riesultati $V[j \times i]$ in modo da avere tutti zero, poiche' si possono mettere solo 0 elementi dentro a un vettore di capienza 0. Riempiremo poi la prima riga, dove metteremo 0 fino a quando il valore di j non sara' abbastanza grande da contenere $P[1]$, al che inseriremo $X[1]$. Una volta inizializzata la prima colonna e la prima riga sara' facile calcolare il valore per le altre celle. I sottoproblemi $P(i, j)$ risolvono il problema di quantificare il valore del bottino, non il sottoinsieme che genera tale bottino. Per porre rimedio a cio' potremo usare dei valori di ritorno piu' complessi che contengono anche queste variabili. Ecco una definizione matematica di come inizializare le celle:

$$P(i,j) = \begin{cases} 0 & \text{se } j = 0 \vee (i = 1 \wedge P[1] > j) \\ v[1] & \text{se } i = 1 \wedge P[1] \leq j \\ P(i-1, j) & \text{se } P[i] < j \\ \max\{P(i-1, j), v[i] + P(i-1, j - P[i])\} & \text{se } P[i] \geq j \end{cases}$$

Ove le chiamate ricorsive a $P(i, j)$ vengno rimpiazzate dall'accesso alla matrice dei risultati precedenti V in modo da evitare le molte chiamate duplicate, in pieno stile di programmazione dinamica.

Si possono poi capire *quali* oggetti sono stati scelti per la soluzione migliore tenendo traccia in una matrice analoga (tramite valori booleani $true \vee false$) quando un elemento vienep reso indicando la cella con un $true$.

3.3.4 Problema del Seam Carving

Il Seam Carving e' una tecnica intelligente per il resizing dell'immagine. La tecnica consiste nell'assegnare a ogni pixel di una immagine una *"energia"* che e' un numero decimale compreso tra 0 e 1. In questo modo possiamo tracciare *linee* (cuciture) che passano in verticale tutta l'immagine e cercano di evitare i pixel con energia alta, i quali indicano i pixel importanti per il contenuto dell'immagine. In gergo algoritmico cerco ogni iterazione una cucitura verticale di peso minimo, dopo di che ricalcolo l'energia, il peso, di ogni pixel e riapplico l'algoritmo da capo. Rimuovendo poi queste linee identificate sara' possibile ridimensionare l'immagine senza perdere troppi contenuti, i quali sono preservati dal velore energetico dei pixel.

3.3.5 Suddivisione del problema

Divideremo l'algoritmo nei seguenti sottoproblemi:

- $P(i,j)$ che trova la cucitura di peso minimo fino al punto (i,j)
- Salvo le soluzioni nella matrice $W[i \times j]$
- Il risultato sara' il $\min\{W[m,1],\ldots,W[m,n]\}$

Ecco una formula che descrive il calcolo di $P(i,j)$ (che vengono ovviamente salvati in $W[i,j]$.

$$P(i,j) = \begin{cases} E(1,j) & \text{se } j = 1 \\ E[i,j] + \min\{P(i-1,j-1), P(i-1,j), P(i-1,j+1)\} & \text{se } 1 < j < N \\ E[i,j] + \min\{P(i-1,j-1), P(i-1,j)\} & \text{se } j = N \end{cases}$$

Una volta trovata l'ultima riga (a parte facendo il minimo tra gli ultimi tre pixel) si risale in cima controllando quali valori di energia combaciano per capire le scelte che sono state fatte nel tempo.

3.4 Distanza di Levenshtein

E' un algoritmo usato ad esempio da uno spell checker che controlla ogni parola che non appartiene al dizionario con quelle ad essa smili per trovare quella piu' simile. Per calcolare la similitudine di due stringhe si puo' usare la distanza di Levenshtein. Scriveremo dunque un algoritmo $intlev(chars1[1..n], chars2[1..m])$ che ritorna un numero in base alla similituine di due stringhe (basso se sono simili, alto altrimenti).

La distanza di Levenshtein e' basata sul concetto di *edit distance* che consiste nel numero di operazioni di editing per trasformare la parola sbagliata in quella giusta. Gli editing ammessi sono:

1. lasciare immutato il carattere corrente (costo 0)
2. cancellare un carattere (costo 1)
3. inserire un carattere (costo 1)
4. sostituire il carattere corrente con uno diverso (costo 1)

NOTA: il cursore si puo' muovere solo in una direzione (nel nostro caso solo in avanti), partendo dal primo carattere e applicando un editing per poi spostarsi al successivo.

Si potrebbero svolgere questi editing in ordine vario o addirittura cancellare tutta la stringa e riscriverla, tutte operazioni che darebbero costi diversi. Percio' definiremo come output del nostro algoritmo *la editing con il valore minore.*

3.4.1 Divisione in sottoproblemi

Analizzeremo dei prefissi delle nsotre sottostringhe come sottoproblema base e vedremo che sara' facile identificarli una volta calcolati i primi in tempo costante. Indichiamo le sottostrighe delle stringhe $S[1..n]$ e $T[1..n]$ come $S[1..i]$ con $i \in \{0,\dots,n\}$ e $T[1..j]$ con $j \in \{0,\dots,m\}$. Usando $i = 0 \vee j = 0$ indichiamo le stringhe vuote.

Come per tutti gli altri algoritmi in programmazione dinamica useremo una matrice $L[i \times j]$ per salvare i risultati delle chiamate precedenti di P. Il risultato del problema sara' dato poi da $P(n,m)$, ovvero $L[n,m]$.

La prima colonna della matrice sara' riempita con i risultati di $P(i,0) = i$ con $i \in \{0,\dots,n\}$, poiche' passare da una stringa lunga i a una lunga 0 richiede i operazioni di delete. Analogamente per la prima riga iterando sulla seconda variabile j.

$$P(i,j) = \begin{cases} \max\{i,j\} & \text{se } j = 0 \vee j = 0 \text{ (0)} \\ P(i-1,j-1) & \text{se } S[i] = T[j] \text{ (1)} \\ 1 + \min\{P(i-1,j-1), P(i-1,j), P(i,j-1)\} & \text{se } S[i] \neq T[j] \text{ (4,2,3)} \end{cases}$$
$$\forall i \in \{0,\dots,n\}, j \in \{0,\dots,m\}$$