▼ **10.0 - Tecniche algoritmiche**

Esistono diverse **tecniche algoritmiche** ognuna delle quali permette di risolvere problemi di simile fattura. Noi vedremo le seguenti 3 tecniche algoritmiche:

- **Divide-et-impera**

  Un problema viene suddiviso in sotto-problemi che vengono risolti ricorsivamente, approccio top-down.

- **Algoritmi greedy**

  Ad ogni passo si fa sempre la scelta che al momento sembra ottima.

- **Programmazione dinamica**

  La soluzione viene costruita a partire dalle soluzioni di un insieme di sotto problemi, approccio bottom-up.

▼ **10.1 - Divide-et-impera**

Struttura di un algoritmo divide-et-impera

Gli algoritmi **divide-et-impera** sono in genere formati dalle seguenti **3 fasi**:

- **Divide**: viene suddiviso il problema di partenza in sotto-problemi di dimensione minore.
- **Impera**: vengono risolti i sotto-problemi in maniera ricorsiva.
- **Combina**: vengono unite le soluzioni dei sottoproblemi per costruire la soluzione del problema di partenza.

Esempio: torre di Hanoi

Il problema della **torre di Hanoi** è un gioco matematico che consiste nell'avere 3 pioli sistemati da sinistra verso destra. Nel piolo di destra sono presenti n dischi di dimensione diversi impilati in ordine decrescente con il grande in basso e il più piccolo in alto. Lo scopo del gioco è quello di impilare tutti gli n dischi in maniera decrescente nel piolo di sinistra potendo spostare un solo disco alla volta e senza mai impilare un disco più grande sopra un disco più piccolo, utilizzando se serve anche il piolo centrale.

Lo **pseudocodice** dell'algoritmo divide-et-impera di tale problema è il seguente:

```
void Hanoi(Stack p1, Stack p2, Stack p3, integer n) {
  if (n = 1) p3.push(p1.pop())
  else {
    hanoi(p1, p3, p2, n - 1)
    p3.push(p1.pop())
    hanoi(p2, p1, p3, n - 1)
```

```
  }
}
```

Le **fasi del divide-et-impera** di questo algoritmo sono le seguenti:

- **Divide**
  1. $n-1$ dischi da p1 a p2.
  2. 1 disco da p1 a p3.
  3. $n-1$ dischi da p2 a p3.
- **Impera**

  Esecuzione ricorsiva degli spostamenti.

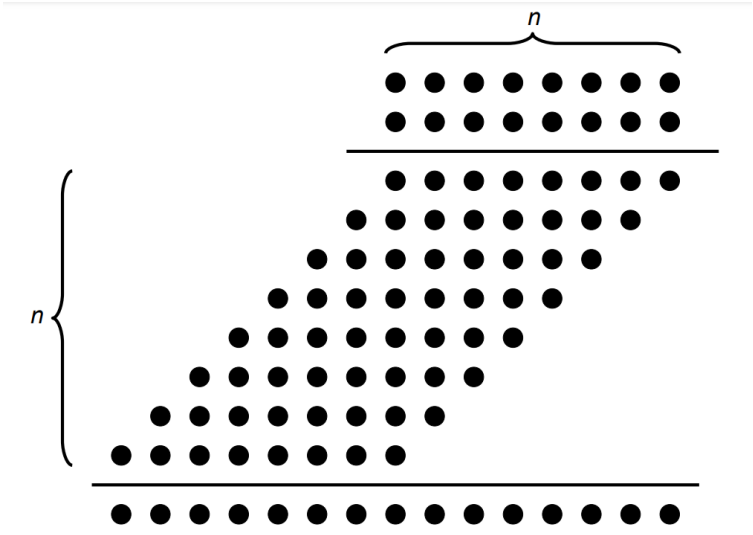
Per calcolare il **costo computazionale** occorre risolvere l'equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot T(n-1) + 1 & n > 1 \end{cases}$$

È possibile dimostrare tale soluzione al problema della torre di Hanoi ha un costo computazionale **esponenziale**, dunque se venisse effettuata un operazione al secondo, per trasferire 64 dischi dal piolo di sinistra a quello di destra servirebbe un tempo pari a circa 127 volte l'età del nostro sole.

Esempio: moltiplicazione di interi

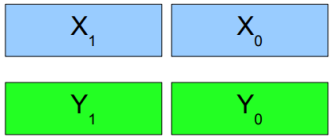
Consideriamo due interi ad n cifre X e Y. Notiamo che l'algoritmo di "moltiplicazione in colonna" ha costo $O(n^2)$ in quanto ogni cifra di Y deve essere moltiplicata per ogni cifra di X.



Metodo della "moltiplicazione in colonna".

Ci chiediamo dunque se è possibile fare di meglio con un algoritmo più efficiente.

Dividiamo i due numeri X e Y in due parti uguali, ottenendo dunque $X = X_1 \cdot 10^{n/2} + X_0$ e $Y = Y_1 \cdot 10^{n/2} + Y_0$.



Calcoliamo a questo punto il prodotto di X e Y:

$$X \cdot Y = (X_1 \cdot 10^{n/2} + X_0) \cdot (Y_1 \cdot 10^{n/2} + Y_0)$$
$$= (X_1 Y_1) \cdot 10^n + (X_1 Y_0 + X_0 Y_1) \cdot 10^{n/2} + (X_0 Y_0) \cdot 10^n$$

Sapendo che la moltiplicazione per 10^n ha un costo computazionale $O(n)$ in quanto equivale ad uno shift a sinistra di n posizioni e notando che l'operazione contiene 4 prodotti di numeri a $n/2$ cifre, otteniamo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n = 1 \\ 4 \cdot T(n/2) + n & n > 1 \end{cases}$$

Tale equazione è risolvibile utilizzando il Master Theorem, con il quale si ottiene un costo computazionale $O(n^2)$, equivalente a quello del metodo della "moltiplicazione in colonna".

Notiamo però che se poniamo P_1, P_2 e P_3 come segue:

$$P_1 = (X_1 + X_0) \cdot (Y_1 + Y_0) = X_1 Y_0 + Y_1 X_0 + X_1 Y_1 + X_0 Y_0$$
$$P_2 = X_1 Y_1$$
$$P_3 = X_0 Y_0$$

possiamo porre $X \cdot Y$ nel seguente modo:

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

Osserviamo quindi che il calcolo di $X \cdot Y$ richiede in tutto solo 3 prodotti (P_1, P_2 e P_3). Riscriviamo quindi l'equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n = 1 \\ 3 \cdot T(n/2) + n & n > 1 \end{cases}$$

Otteniamo dunque, utilizzando il Master Theorem, un costo computazionale equivalente a $O(n^{\log_2 3}) \approx O(n^{1.59})$.

Esempio: sottovettore di valore massimo

Il problema consiste nel, dato in input un vettore v di lunghezza n contenente n valori arbitrari, individuare il sottovettore non vuoto di v la **somma dei cui elementi sia massima** e restituire tale somma.

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

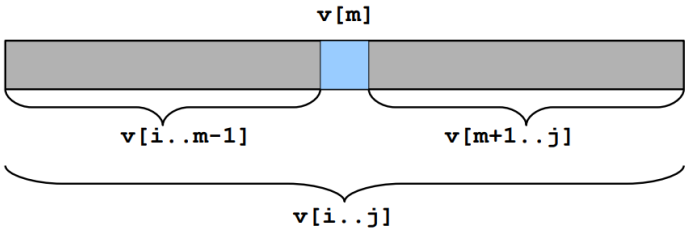
Una prima soluzione di tale problema può essere quella di utilizzare un sistema di **forza bruta**, il quale analizza tutti i possibili sotto vettori del vettore di partenza ed individua quello con valore massimo. Lo **pseudocodice** di un tale algoritmo è il seguente:

```
double SommaMax(double v[1 ... n]) {
  double smax = v[1]
  for (int i = 1; i < n; i++) {
    double s = 0
    for integer (int j = i; j < n; j++) {
      s = s + v[j]
      if (s > smax) smax = s
    }
  }
  return smax
}
```

Il **costo computazionale** di questo algoritmo è $O(n^2)$.

È possibile però migliorare questo costo utilizzando una tecnica divide-et-impera, la quale consiste nel dividere l'array in due parti, separate dall'elemento centrale $v[m]$, e ricadendo in 3 casistiche:

- Il sottoarray con valore massimo si trova nell'array di sinistra $v[i \dots m-1]$.
- Il sottoarray con valore massimo si trova nell'array di destra $v[m+1 \dots j]$.
- Il sottoarray con valore massimo si trova a cavallo tra i due array.



Occorre dunque calcolare il valore della somma degli elementi di questi 3 sottoarray e confrontarli. Per i primi 2 è possibile farlo richiamando lo stesso algoritmo in maniera ricorsiva, mentre per il terzo il calcolo è più complicato. Notiamo che tale sotto-array può contenere una parte prima di $v[m]$ e una parte dopo di $v[m]$. Occorre dunque calcolare il sottoarray sa con valore massimo, incluso quello vuoto, che abbia come ultimo elemento $v[m-1]$ e quello sb con valore massimo, incluso quello vuoto, che abbia come primo elemento $v[m+1]$. Otteniamo dunque che il sottoarray con valore massimo a cavallo tra i due sottoarray è equivalente a $sa + v[m] + sb$.

Lo **pseudocodice** che sfrutta tale algoritmo è dunque il seguente:

```
double sommaMax(double v[1 ... n], int i, int j) {
  if (i > j) return 0
  else if (i == j) return V[i]
  else {
    m = Math.floor((i + j) / 2)

    double l = sommaMax(v, i, m - 1)
    double r = sommaMax(v, m + 1, j)

    double sa = 0, sb = 0, s = 0
    for (int k = m - 1; k >= i; k--) {
     s = s + V[k]
     if (s > sa) sa = s
    }
    s = 0;
    for (int k = m + 1; k <= j; k++) {
     s = s + V[k]
     if (s > sb) sb = s
    }

    return max(l, r, sa + v[m] + sb)
```

```
  }
}
```

Il **costo computazionale** di tale algoritmo, dimostrabile tramite l'utilizzo del Master Theorem, è $O(n \log n)$.

▼ 10.2 - Greedy

> La tecnica algoritmica **greedy** consiste nel fare ad ogni passo la scelta che sembra ottima.

Tale tecnica può essere utilizzata nel caso in cui è possibile dimostrare che tra le scelte disponibili ne esiste una semplice che porta alla soluzione ottima, e che fatta tale scelta resta un sottoproblema della stessa struttura del problema principale, risolvibile a sua volta tramite una scelta greedy.

Esempio: problema del resto

Dato un intero R che rappresenta un importo in centesimi da erogare, scegliere un numero minimo di monete necessarie per erogare tale importo utilizzando solo i seguenti tagli: 50c, 20c, 10c, 5c, 2c, 1c.

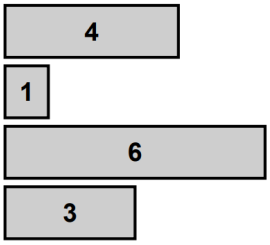
Per un sistema monetario di questo tipo è possibile adottare un algoritmo di tipo greedy, il quale consiste nel prendere di volta in volta la moneta più grande contenuta nella cifra da erogare. I sistemi monetari che consentono un approccio di tipo greedy sono detti **sistemi monetari canonici**, ma esistono anche altri tipo di sistemi monetari per i quali l'approccio greedy non è quello ottimale, ad esempio per erogare 6 con monete da 4, 3 e 1 (greedy: 4 + 1 + 1, ottimo: 3 + 3).

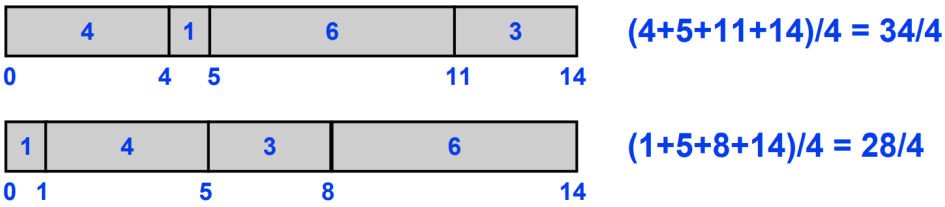
Lo **pseudocodice** dell'algoritmo di tipo greedy per questa tipologia di problemi è il seguente:

```
// R = resto da erogare
// T[1 ... n] = gli n tagli di monete a disposizione
// output = numero totale di monete da erogare
int restoGreedy(int R, int T[1 ... n]) {
  int coinNum = 0
  int i = 1

  while (R > 0 && i <= n) {
    if (R >= T[i]) {
      R -= T[i]
      coinNum++
    } else i++
  }

  if (R > 0) errore: resto non erogabile // es. il taglio minore è > 1c
  else return coinNum
}
```

Esempio: problema di scheduling

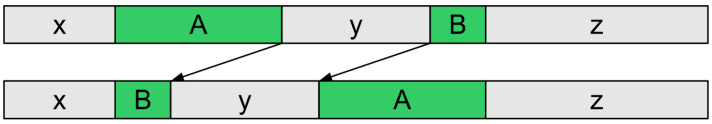
Un problema risolvibile tramite un algoritmo di tipo greedy consiste nel **minimizzare il tempo medio di completamento** di un insieme di operazioni. Ad esempio questo problema è rilevante nello scheduling del job di un processore. Molto spesso infatti un singolo processore si ritrova a dover gestire un numero n di job ognuno dei quali ha un proprio tempo di esecuzione. Per massimizzare l'efficienza generale dell'elaboratore e sfruttare al massimo le capacità del processore è utile minimizzare il tempo medio di completamento, il quale si calcola tramite la somma dei tempi di completamento dei singoli job diviso il numero di job completati.





L'algoritmo greedy per questo tipo di problemi consiste nell'eseguire n passi, in ognuno dei quali si manda in esecuzione il job, tra quelli che rimangono, che richiede meno tempo.

Possiamo dimostrare la correttezza di questo ragionamento tramite un esempio:



Osserviamo che i tempi di completamento dei job x e z rimangono uguali, e il tempo di completamento di A nella seconda soluzione è uguale al tempo di completamento di B nella prima. Notiamo però che il tempo di completamento di B nella seconda soluzione è minore rispetto al tempo di completamento di A nella prima soluzione, e che il tempo di completamento di y è diminuito. Possiamo quindi concludere che il tempo medio di completamento è diminuito.

Esempio: problema della compressione

Il **problema della compressione** consiste nel dover rappresentare un insieme di caratteri tramite una certa codifica (ad ogni carattere deve essere assegnato un certo codice binario) in modo tale da minimizzare la lunghezza di un insiemi di caratteri codificati.

Nel fare ciò dobbiamo dunque tenere conto della probabilità con cui viene utilizzato un certo caratteri, in modo tale da assegnare ad un carattere molto frequente una codifica con meno bit.

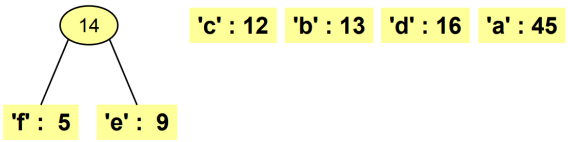
Inoltre, visto che utilizziamo una **codifica a lunghezza variabile** per il motivo appena accennato, dobbiamo tenere in mente che nessun codice può essere un **prefisso** di un altro codice. Ad esempio possiamo notare che la codifica f(a) = 1, f(b)=10, f(c)=101 risulta ambigua in quanto il codice 101 può essere decodificato sia come "c" che come "ba", in quanto la codifica di "b" è un prefisso della codifica di "c".

I **codici di Huffman** sono delle tipologie di codifiche realizzabili utilizzando un algoritmo greedy. Tali codifiche si realizzano seguendo i seguenti step:

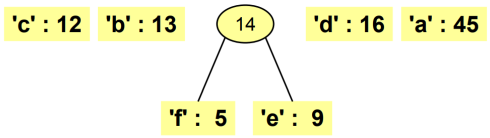
1. Costruire una lista di nodi, in cui ogni nodo contiene un carattere e la sua frequenza (es. numero di volte in cui compare all'interno di un pdf), ordinata in base alla frequenza.

'f' : 5 **'e' : 9** **'c' : 12** **'b' : 13** **'d' : 16** **'a' : 45**

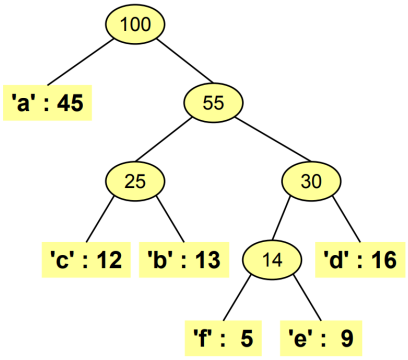
2. Rimuovere dalla lista i due nodi con la frequenza minore e collegarli ad un nodo padre etichettato con la somma delle frequenze dei suoi due figli.



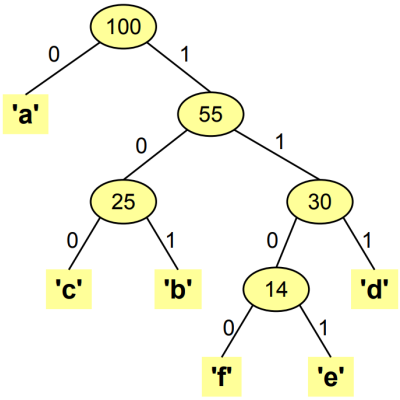
3. Aggiungere l'albero appena creato alla lista in posizione adatta in modo da mantenere una lista ordinata.



4. Ripetere i passi 2 e 3 fino a quando non rimane un solo albero.



5. Etichettiamo gli archi sinistri dell'albero con uno 0 e gli alberi destri con un 1.



Una volta costruito l'albero possiamo associare ogni carattere al suo codice unendo i numeri presente negli archi necessari per passare dalla radice al nodo contenente il carattere. Ad esempio nell'albero appena costruito il carattere 'f' viene codificato con 1100.

Lo **pseudocodice** per costruire un tale albero è il seguente:

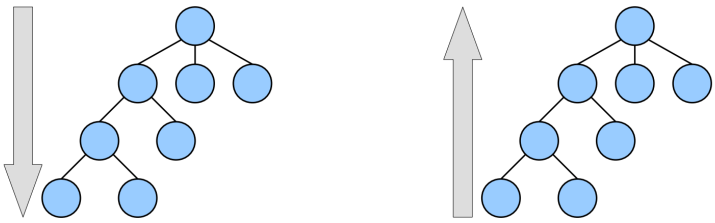
```
Tree huffman(float f[1 ... n], char c[1 ... n]) {
  Q = new MinPriorityQueue()

  // insert single nodes into the queue
  for (int i = 1; i < n; i++) {
    z = new TreeNode(f[i], c[i])
    Q.insert(f[i], z)
  }

  // create the tree
  for (int i = 1; i < n - 1; i++) {
    z1 = Q.findMin()
    Q.deleteMin()
    z2 = Q.findMin()
    Q.deleteMin()

    z = new TreeNode(z1.f + z2.f, '')
    z.left = z1
    z.right = z2

    Q.insert(z1.f + z2.f, z)
  }

  // return the tree
  return Q.findMin()
}
```

▼ **10.3 - Programmazione dinamica**

Programmazione dinamica vs divide-et-impera

> La **programmazione dinamica** consiste nel risolvere un problema combinando in maniera appropriata le soluzioni dei suoi sottoproblemi.

A differenza degli algoritmi divide-et-impera però la programmazione dinamica è una **tecnica iterativa** invece che ricorsiva e utilizza un approccio **bottom-up** piuttosto che top-down. Questa è vantaggiosa quando ci sono **sottoproblemi ripetuti**, mentre il divide-et-impera è utile quando i sottoproblemi sono indipendenti.



Approccio divide-et-impera (destra) vs programmazione dinamica (sinistra).

Esempio: algoritmo di Fibonacci

Divide-et-impera

Un primo approccio per calcolare un numero di Fibonacci è quello di utilizzare la tecnica del **divide-et-impera**, utilizzando la definizione di numero di Fibonacci e quindi calcolando in maniera ricorsiva i due numeri di Fibonacci precedenti.

Lo **pseudocodice** è il seguente:

```
int fibRic(int n) {
 if ((n = 1) || (n = 2)) return 1
 else return fibRic(n - 1) + fibRic(n - 2)
}
```

Il **costo computazionale** è $O(2^n)$.

È possibile migliorare questo risultato notando che alcuni numeri vengono calcolati più volte, dunque utilizzando una tecnica di **memorizzazione** si evita di ricalcolarsi.

Lo **pseudocodice** è il seguente:

```
int cache[n]

int fibMemorization(int n) {
 for (i = 0; i < n; i++) cache[i] = -1
 fibMem(n)
}

int fibMem(int n) {
  if (cache[n] != -1) return cache[n]

  if ((n = 1) || (n = 2)) cache[n] = 1
  else cache[n] = fibMem(n - 1) + fibMem(n - 2)

  return cache[n]
}
```

Il **costo computazionale** è $O(n)$.

Programmazione dinamica

Un secondo approccio consiste nell'utilizzare la **programmazione dinamica**, risolvendo tramite un approccio bottom-up prima tutti i numeri di Fibonacci precedenti per poi calcolare il numero dato in input seguendo la definizione.

Lo **pseudocodice** è il seguente:

```
int fibIter(integer n) {
  if (n <= 2) return 1
  else {
    int f[1 ... n]
    f[1] = 1;
    f[2] = 1;

    for (int i = 3; i < n; i++)
      f[i] = f[i - 1] + f[i - 2]

    return f[n]
  }
}
```

Il **costo computazionale** è:

- **Tempo**: $O(n)$.
- **Spazio**: $O(n)$.

Visto che per calcolare ogni numero di Fibonacci ci occorrono solamente i due numeri di Fibonacci precedenti possiamo ottimizzare l'utilizzo della memoria facendo diventare il costo computazionale in termini di spazio costante.

Lo **pseudocodice** è il seguente:

```
int fib(int n) {
  if (n < 2) return 1
  else {
    int f[0 ... 2]
    f[1] = 1
    f[2] = 1

    for (int i = 2; i < n; i++) {
      f[0] = f[1]
      f[1] = f[2]
      f[2] = f[1] + f[0]
    }

    return f[2]
  }
}
```

Il **costo computazionale** è:

- **Tempo**: $O(n)$.
- **Spazio**: $O(1)$.

Esempio: sottovettore di valore massimo

Abbiamo già analizzato questo problema nella sezione degli algoritmi divide-et-impera. Gli algoritmi che abbiamo utilizzato sono stati 2, uno che utilizzava la tecnica della **forza bruta** e aveva un costo computazionale $O(n^2)$, e l'altro che utilizzava la tecnica del **divide-et-impera** e il quale costo computazionale era $O(n \log n)$.

Possiamo trovare un'ulteriore soluzione a questo problema utilizzando la **programmazione dinamica**. Sia $V[1 \dots n]$ il vettore in input, consideriamo il sottoproblema $P(i)$ il quale calcola il valore massimo tra i sottovettori di V che abbiano come ultimo elemento $V[i]$. L'idea sarebbe quella di costruire un vettore $S[1 \dots n]$ che in posizione i presenta la soluzione del sottoproblema $P(i)$. La soluzione al problema originario è dunque il valore massimo contenuto nel vettore S.

A questo punto dobbiamo risolvere il sottoproblema $P(i)$:

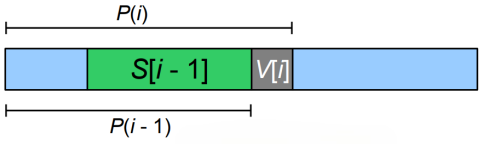
- Caso $i = 1$

  $S[1] = V[1]$, in quanto è l'unico sottovettore vuoto possibile.

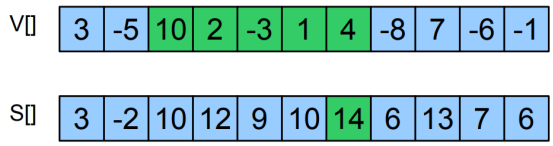
- Caso $i > 1$

  Supponiamo di aver già risolto il sottoproblema $P(i-1)$. Siccome nel risultato dobbiamo per forza includere $V[i]$, ci resta solo da valutare se aggiungere a $V[i]$ anche $S[i-1]$ o meno.

  Quindi $S[i] = max(V[i], V[i] + S[i-1])$.



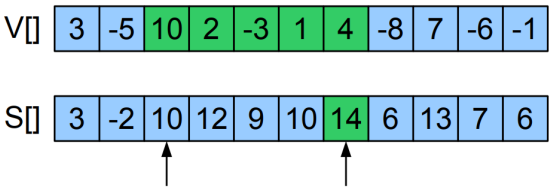
A questo punto abbiamo costruito un vettore del seguente tipo:



Lo **pseudocodice** per fare costruire tale vettore e restituire il valore massimo è il seguente:

```
double sottovettoreMax(double V[1 ... n]) {
  double S[1 ... n]
  S[1] = V[1]

  int imax = 1
  for (int i = 2; i < n; i++) {
    if (S[i - 1] + V[i] >= V[i])
      S[i] = S[i - 1] + V[i]
    else S[i] = V[i]

    if (S[i] > S[imax]) imax = i
  }

  return S[imax]
}
```

Nel caso in cui volessimo conoscere quale sia il sottovettore con somma massima possiamo farlo osservando il vettore S. L'indice contenente l'elemento con valore massimo è l'indice di fine del sottoarray, mentre l'indice di inizio corrisponde a quello del primo elemento, partendo dall'indice di fine e andando verso sinistra, il quale valore corrisponde al valore presente nel vettore V.



Esempio: problema dello zaino

Il problema dello zaino consiste nell'avere un insieme $X = \{1, \dots, n\}$ di n oggetti, ognuno dei quali ha un peso $p[i]$ e $v[i]$. Disponiamo di un contenitore/zaino in grado di trasportare fino a un peso massimo P e dobbiamo determinare un sottoinsieme di X tale che il peso complessivo dei suoi oggetti sia minore di P, mentre il valore complessivo di essi sia il massimo possibile.

Greedy 1

La prima soluzione consiste nello scegliere ad ogni passo l'oggetto con il **valore massimo** tra quelli rimanenti nell'insieme X e tali per cui il loro peso sia minore o uguale alla capacità residua nello contenitore.

Questo algoritmo però non fornisce sempre la soluzione ottima.

Greedy 2

La seconda soluzione consiste nello scegliere ad ogni passo l'oggetto con il **valore specifico massimo** tra quelli rimanenti nell'insieme X e tali per cui il loro peso sia minore o uguale alla capacità residua nello contenitore.

Il valore specifico di un oggetto consiste nel rapporto tra il suo valore e il suo peso.

Anche questo algoritmo, pur essendo più preciso rispetto al primo, non fornisce sempre la soluzione ottima.

Programmazione dinamica

È possibile trovare una soluzione ottima per questo problema utilizzando la programmazione dinamica. Questo approccio consiste nel creare un vettore V di due dimensioni in cui per ogni elemento $V[i,j]$ inserire il valore del problema risolto per i primi i elementi di X e per la capacità j del contenitore.

Per riempire tale array ci ritroviamo i seguenti casi:

- $i = 1$

  Se $p[1] > j$, allora $V[1,j] = 0$.

  Se $p[1] \leq j$, allora $V[1,j] = v[1]$.

- $i > 1$

  Supponiamo di avere già il valore $V[i-1,j]$, dobbiamo scegliere se inserire il nuovo elemento i oppure no.

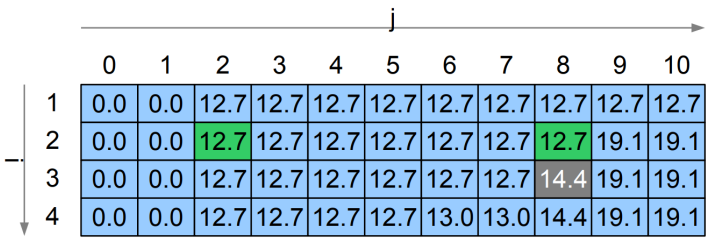
  Se $p[i] > j$, allora non possiamo inserirlo, quindi $V[i,j] = V[i-1,j]$.

  Se $p[i] \leq j$ allora possiamo decidere se inserirlo, e in quel caso $V[i,j] = V[i-1, j - p[i]] + v[i]$, oppure non inserirlo, quindi $V[i,j] = V[i-1,j]$. Dobbiamo dunque scegliere il valore massimo tra le due opzioni.

  Riassumendo abbiamo che:

$$V[i,j] = \begin{cases} V[i,j] = V[i-1,j] & p[i] > j \\ max(V[i-1, j-p[i]] + v[i], V[i-1,j]) & p[i] \leq j \end{cases}$$

p =[2, 7, 6, 4]
v =[12.7, 6.4, 1.7, 0.3]



$V[3,8] = max(V[2,6] + 1.7, V[2,8])$.

Il valore della soluzione del problema di partenza è dunque $V[n,P]$.

Per conoscere anche l'indice degli oggetti che vengono inseriti nel contenitore utilizziamo una tabella ausiliaria K della stessa grandezza della tabella V che riempiamo durante il riempimento di quest'ultima tramite un true se l'oggetto i viene inserito nel contenitore quando la capienza è j, altrimenti false.

	0	1	2	3	4	5	6	7	8	9	10
1	F	F	T	T	T	T	T	T	T	T	T
2	F	F	F	F	F	F	F	F	F	T	T
3	F	F	F	F	F	F	F	F	T	F	F
4	F	F	F	F	F	F	T	T	F	F	F

Successivamente utilizziamo il seguente algoritmo che fa uso della tabella K:

```
int j = P
int i = n
while (i > 0) {
  if (K[i, j]) {
    print(“Seleziono oggetto “, i)
    j -= p[i]
  }
  i--
}
```

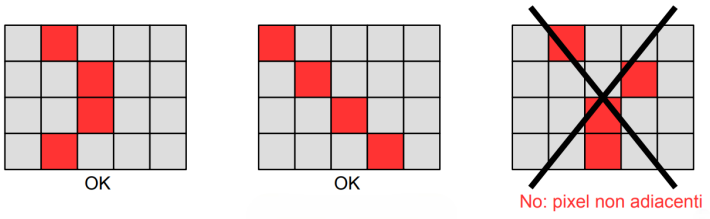
Esempio: Seam Carving

Il **Seam Carving** è un algoritmo di ridimensionamento delle immagini che utilizza l’importanza dei pixel e la programmazione dinamica per eliminare le parti meno rilevanti della figura.

Differenza tra le usuali tecniche di ridimensionamento di un’immagine e il Seam Carving.

Questa tecnica, per ogni immagine con M righe e N colonne da ridimensionare, utilizza un vettore E di due dimensioni $M \times N$ che per ogni pixel presenta un valore compreso tra 0 e 1 che ne rappresenta l’energia, ossia l’importanza calcolata solitamente in base al suo colore e al contrasto con i pixel adiacenti (0: poco importante, 1: molto importante).

L’obiettivo è quello di scegliere, per ogni riga/colonna da eliminare, la cucitura la cui somma delle energie dei pixel in essa contenuti sia minore, dove per cucitura si intende un cammino composto da pixel adiacenti.



Cuciture verticali.

Per risolvere il problema è dunque necessario creare un vettore W di due dimensioni, grande quanto il vettore E, in cui per ogni elemento $W[i,j]$ viene memorizzato il valore della cucitura con valore minimo che termini nel pixel $[i,j]$.

Per riempire tale vettore, prendendo in considerazione cuciture verticali, ci ritroviamo i seguenti casi:

- $i = 1$

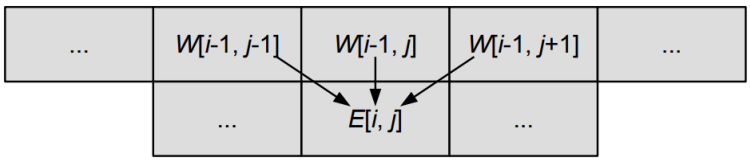
  L'ultimo pixel della cucitura si trova nella prima riga, dunque $W[1,j] = E[1,j]$.

- $i > 1$
  - $j = 1$

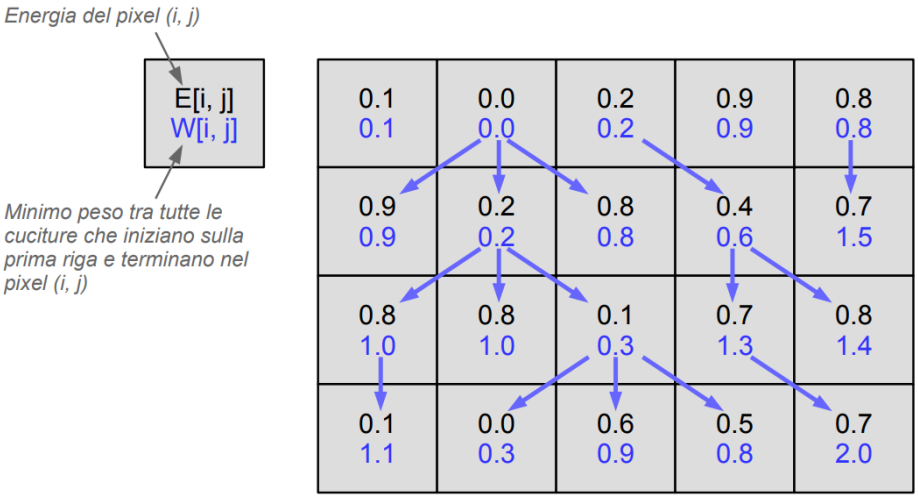
    $W[i,j] = min(W[i-1,j], W[i-1,j+1])$.
  - $1 < j < N$

    $W[i,j] = min(W[i-1,j-1], W[i-1,j], W[i-1,j+1])$.
  - $j = N$

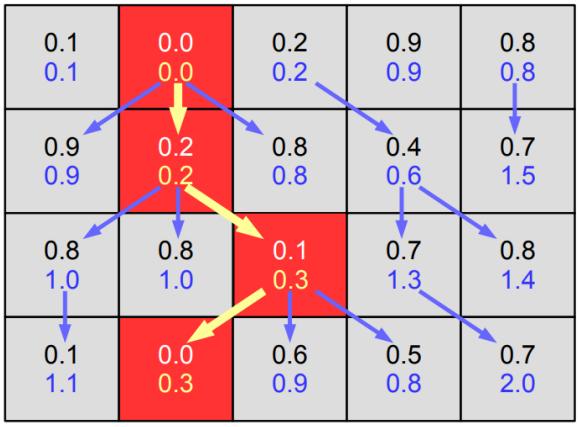
    $W[i,j] = min(W[i-1,j-1], W[i-1,j])$.



Ci ritroveremo con un vettore di questo tipo:

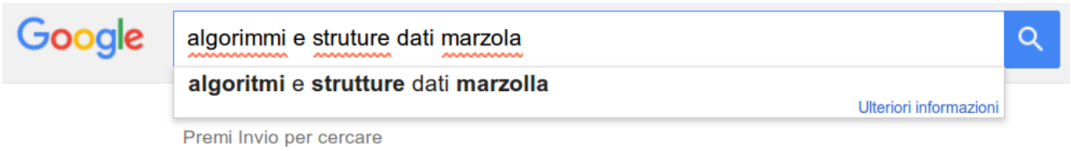


A questo punto per eliminare una cucitura dall'immagine scegliamo quella con il valore minimo tra tutte le cuciture che terminano nella riga M.



Esempio: distanza di Levenshtein

I **correttori ortografici** sono strumenti in grado di suggerire le parole corrette più simili rispetto a quelle che abbiamo digitato.



Alcuni di questi strumenti fanno utilizzo del concetto di "**edit distance**" al fine di comprendere la distanza tra due stringhe e suggerire le stringhe più simili a quella che sta venendo digitata. L'edit distance consiste nel numero di operazioni di editing necessarie per trasformare una stringa in un'altra, e le **operazioni di editing** disponibili sono le seguenti:

- Lasciare immutato un carattere, costo 0.
- Cancellare un carattere, costo 1.
- Inserire un carattere, costo 1.
- Sostituire un carattere con uno diverso, costo 1.

Si inizia dal primo carattere della stringa e ad ogni operazione ci si sposta sul carattere successivo. Possiamo notare tramite un esempio che è possibile avere più di una sequenza di operazioni possibili per trasformare una stringa in un'altra.

ALBERO	→	LBERO	cancellazione A
LBERO	→	BERO	cancellazione L
BERO	→	ERO	cancellazione B
ERO	→	RO	cancellazione E
RO	→	O	cancellazione R
O	→	_	cancellazione O
_	→	L_	inserimento L
L_	→	LI_	inserimento I
LI_	→	LIB_	inserimento B
LIB_	→	LIBR_	inserimento R
LIBR_	→	LIBRO_	inserimento O

ALBERO	→	LBERO	cancello A
LBERO	→	LBERO	lascio immutato
LBERO	→	LIBERO	inserisco I
LIBERO	→	LIBERO	lascio immutato
LIBERO	→	LIBRO	cancello E

Due sequenze di trasformazione della stringa “albero” in “libro”.

Osserviamo infatti che abbiamo utilizzato due sequenze differenti per trasformare la stringa “albero” in “libro”, le quali presentano rispettivamente costo 11 e 3.

La **distanza di Levenshtein** tra due stringhe $S[1 \dots n]$ e $T[1 \dots m]$ è il costo minimo tra tutte le sequenze di editing che trasformano la stringa S nella stringa T.

È possibile ricavare la distanza di Levenshtein tramite programmazione dinamica, costruendo un vettore L di due dimensioni in cui per ogni elementi $L[i, j]$ viene inserita la distanza di Levenshtein tra $S[1 \dots i]$ e $T[1 \dots j]$. Possiamo quindi riempire tale vettore nel seguente modo:

- $i = 0$ oppure $j = 0$

  In questo caso una delle due stringhe è vuota, dunque la distanza di Levenshtein corrisponde alla lunghezza della stringa non vuota.

  $L[i, j] = max(i, j)$.

- Altrimenti
  - $S[i] \neq T[j]$

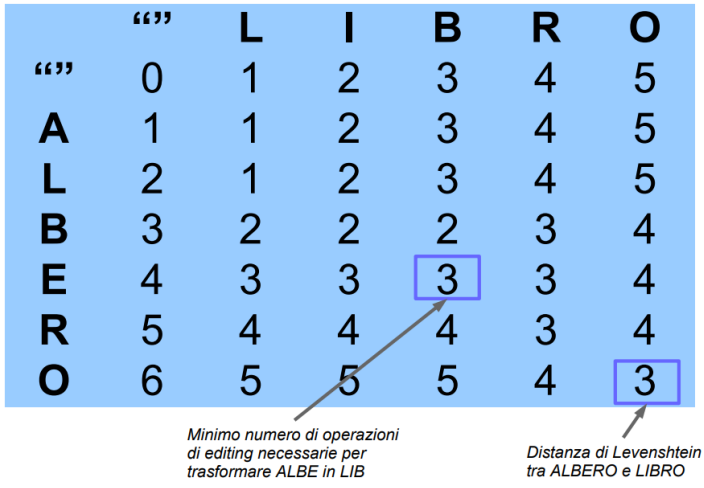
    La distanza di Levenshtein è il minimo tra il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j]$ + rimozione del carattere $S[i]$, il costo per trasformare $S[1 \dots i]$ in $T[1 \dots j - 1]$ + aggiunta del carattere $T[j]$ e il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j - 1]$ + modifica del carattere $S[i]$ in $T[j]$.

    $L[i, j] = max(L[i - 1, j] + 1, L[i, j - 1] + 1, L[i - 1, j - 1] + 1)$.

  - $S[i] = T[j]$

    La distanza di Levenshtein è il minimo tra il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j]$ + rimozione del carattere $S[i]$, il costo per trasformare $S[1 \dots i]$ in $T[1 \dots j - 1]$ + aggiunta del carattere $T[j]$ e il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j - 1]$ + lasciare l’ultimo carattere invariato.

    $L[i, j] = max(L[i - 1, j] + 1, L[i, j - 1] + 1, L[i - 1, j - 1])$.



Dopo aver costruito tale vettore troviamo il valore della distanza di Levenshtein tra $S[1 \dots n]$ e $T[1 \dots m]$ in $L[n, m]$.