eliminato al più una volta. Quando un punto è eliminato, esso non viene più riconsiderato. Inoltre, ogni volta che è eseguito il corpo del **while** è eliminato un punto. Pertanto, la complessità del ciclo **for** è $\Theta(n)$. La complessità di tutto l'algoritmo è invece $O(n \log n)$, perché dominata dalla fase di ordinamento dei punti.

Realizzando l'insieme di punti con un vettore $p$ di $n$ elementi di tipo "punto", l'algoritmo di Graham può essere scritto in Pascal con la seguente funzione dove la pila è simulata direttamente con la porzione $p[1..j]$ del vettore, scambiando tra loro punti del vettore stesso.

```
function GRAHAM(var p: vettore; n: integer): integer;
  var i, j, min: integer; R: retta; T: punto;
  begin
    min := 1;
    for i := 2 to n do
      if p[i].Y < p[min].Y then min := i;
    T := p[1]; p[1] := p[min]; p[min] := T;
    ordina p[1], ..., p[n] con HEAPSORT in base all'angolo α formato con
    l'asse orizzontale quando sono connessi con p[1], eliminando poi gli
    eventuali punti "allineati" ed aggiornando se necessario n;
    if n ≥ 2 then j := 2 else j := 1;
    for i := 3 to n do begin
      R.P1 := p[j]; R.P2 := p[j − 1];
      while not STESSAPARTE(R, p[1], p[i]) do begin
        j := j − 1;
        R.P1 := p[j];
        R.P2 := p[j − 1]
      end;
      j := j + 1;
      T := p[j];
      p[j] := p[i]; p[i] := T
    end;
    GRAHAM := j
  end;
```

La testa della pila è individuata dall'indice $j$. La cancellazione di $p[j]$ è effettuata semplicemente decrementando l'indice $j$ all'interno del **while**. L'indice $j$ è incrementato all'uscita per effettuare il successivo inserimento di $p[i]$, che avviene scambiando gli elementi $p[j]$ e $p[i]$, dato che $j \leq i$ e che $p[j]$ non è (più) un vertice dell'inviluppo "corrente" qualora $j < i$. La funzione restituisce come risultato l'indice $j$ tale che i punti $p[1], \ldots, p[j]$ sono i vertici dell'inviluppo convesso, nell'ordine che si incontrano secondo il verso antiorario. L'ordinamento iniziale può essere fatto anch'esso direttamente sul vettore $p$ con la procedura HEAPSORT, così che non sono necessarie strutture ausiliarie né per ordinare i dati di ingresso né per mantenere la soluzione, tranne ovviamente un campo aggiuntivo nel record di ciascun punto per contenere l'angolo $\alpha$. I dettagli riguardanti le modifiche da apportare alla procedura HEAPSORT per ordinare il vettore di punti in base ad $\alpha$ e l'eliminazione degli eventuali punti "allineati" sono lasciati per esercizio.



## 12.2 String matching

Il problema dello "STRING MATCHING" consiste nel trovare un'occorrenza di una sequenza $P$ di $m$ caratteri, detta pattern, all'interno di un'altra sequenza $T$ di $n$ caratteri, detta testo. In altri termini, si chiede se esiste un indice $k$ con $1 \leq k \leq n - m + 1$, tale che il $j$-esimo carattere di $P$ sia uguale al $(k + j - 1)$-esimo carattere di $T$, per $j = 1, 2, \ldots, m$. Le due sequenze (o stringhe) $P$ e $T$ sono formate da caratteri tratti dallo stesso insieme $A$, di cardinalità finita, detto "alfabeto", e sono tali che $m$ non supera $n$.

**Esempio 12.4** (String matching). Siano $P =$ `10110110` e $T =$ `10110010101-101011011011`, dove $A = \{0, 1\}$, $n = 23$ ed $m = 8$. È facile vedere che c'è un'occorrenza del pattern $P$ a partire dalla posizione $k = 14$ del testo $T$: `1011001-0101101011011011`. □

Un algoritmo ovvio per risolvere il problema consiste nel cercare di riconoscere il pattern a partire dalla prima posizione del testo, confrontando il primo carattere di $P$ col primo di $T$, il secondo di $P$ col secondo di $T$, ecc. Se il pattern non è interamente riconosciuto, si ripete il procedimento a partire però dalla seconda posizione nel testo, confrontando stavolta il primo carattere di $P$ col secondo di $T$, il secondo di $P$ col terzo di $T$, ecc. Se anche stavolta il pattern non è completamente riconosciuto, si riprova a partire dalla terza posizione del testo, poi dalla quarta, e così via, continuando fino al riconoscimento completo del pattern o all'esaurimento del testo. Realizzando le stringhe $P$ e $T$ con due vettori di caratteri, si ottiene la seguente funzione Pascal RICERCABRUTA, che utilizza tre indici: $k$ (per avanzare nel testo), $j$ (per scandire $P[1..m]$), ed $i$ (per scandire la porzione $T[k..k + m - 1]$ quando cerca di riconoscere se $P[1..m] = T[k..k + m - 1]$). La funzione restituisce la posizione $k$ di $T$ a partire dalla quale si trova la prima occorrenza di $P$, se c'è una copia di $P$ in $T$, oppure $i = n + 1$, se non c'è alcuna copia di $P$ in $T$.

```
function RICERCABRUTA(var P, T: vettore; n, m: integer): integer;
  var i, j, k: integer;
  begin
    i := 1; j := 1; k := 1;
    while (i ≤ n) and (j ≤ m) do
      if T[i] = P[j]
        then begin i := i + 1; j := j + 1 end
        else begin k := k + 1; i := k; j := 1 end;
    if j > m then
      RICERCABRUTA := k
    else
      RICERCABRUTA := i;
  end;
```

L'algoritmo può essere interpretato graficamente come se la configurazione di caratteri individuata dal pattern fosse sovrapposta a quella del testo e si cerchi per quale traslazione del pattern sul testo tutti i caratteri del pattern siano uguali a tutti i caratteri del testo.

**Esempio 12.5** (Sovrapposizione e traslazione). Eseguendo la procedura RICERCABRUTA sulle stringhe $P$ e $T$ dell'Esempio 12.4, si ottiene:

```
T = 10110010101101011011011    i = 1, j = 1, k = 1,
P = 10110110                   i = 6, j = 6,

T = 10110010101101011011011    i = 2, j = 1, k = 2,
P =  10110110                  i = 2, j = 1,

T = 10110010101101011011011    i = 3, j = 1, k = 3,
P =   10110110                 i = 4, j = 2,
    ⋮
T = 10110010101101011011011    i = 14, j = 1, k = 14,
P =              10110110      i = 22, j = 8, EUREKA!
```

□

Se il pattern non è completamente riconosciuto a partire dalla posizione $k$ del testo, la funzione RICERCABRUTA effettua un "backtrack" su entrambi gli indici $i$ e $j$. Nel caso pessimo, l'indice $k$ può assumere $n - m + 1$ valori e, per ogni valore di $k$, gli indici $i$ e $j$ possono assumerne $m$. Essendo in generale $m$ più piccolo di $n$, la quantità $n - m + 1$ è $O(n)$, e quindi nel caso pessimo sono eseguiti $O(mn)$ confronti tra caratteri di $P$ e $T$.

**Esempio 12.6** (Caso pessimo RICERCABRUTA). Se $A = \{0, 1\}$, il caso pessimo si incontra quando sia $P$ che $T$ sono formate da tutti 0 seguiti da un unico 1 finale. □

Vediamo come un "backtrack" meno brutale permetta di progettare un algoritmo di complessità $O(m + n)$.

### 12.2.1 Algoritmo di Knuth, Morris e Pratt

L'idea di base dell'algoritmo proposto da Knuth, Morris e Pratt nel 1977 è la seguente: dopo aver riconosciuto $j - 1$ caratteri del pattern a partire da una certa posizione nel testo ed aver fallito al $j$-esimo, perché tornare indietro di $j - 2$ posizioni nel testo? In effetti, i $j - 1$ caratteri già riconosciuti fanno parte del pattern stesso e sono noti addirittura prima di iniziare la ricerca nel testo! Perché non trarre vantaggio da questa informazione nota in anticipo?

**Esempio 12.7** (Backtrack inutile). Se $T = 10110010101101011011011$ e $P = 10110110$, i primi 5 caratteri di $P$ sono uguali ai primi 5 caratteri di $T$, ma il sesto è diverso. Al momento del primo backtrack, $i = j = 6$. Ripartire con $i = 2$ e $j = 1$ corrisponde a traslare a destra di una posizione il pattern rispetto al testo. Ma ciò corrisponde anche a traslare a destra di una posizione rispetto a se stessa la sottosequenza 10110 di $P$ che è stata riconosciuta nel testo! Nella sottosequenza 10110, i primi 4 caratteri non coincidono con gli utimi 4 ed è pertanto inutile ripartire con $i = 2$, perché sicuramente non potrà esserci una copia di $P$ a partire da $T[2]$. Inoltre, neanche i primi 3 caratteri di 10110 coincidono con gli ultimi 3, ed è altrettanto inutile ripartire con $i = 3$, perché non potrà esserci una copia di $P$ neanche a partire da $T[3]$. I primi 2 caratteri, invece, coincidono con gli ultimi 2 (10110 e 10110) e pertanto conviene ripartire direttamente con $i = 4$. Ma poiché i successivi due caratteri di $P$ (cioè 10) coincidono sicuramente con quelli di $T$, tanto vale ripartire con $i = 6$ e $j = 3$ (anziché 1). Poiché 6 è proprio il vecchio valore di $i$ al momento del backtrack, è inutile effettuare il backtrack sull'indice $i$, e basta effettuarlo soltanto sull'indice $j$, portandolo da 6 a 3. □



Si considerino due copie dei primi $j - 1$ caratteri del pattern, e si immagini di disporle orizzontalmente una sotto all'altra, in modo che il primo carattere della copia inferiore stia "sotto" il secondo carattere della copia superiore. Se tutti i caratteri sovrapposti nelle due copie non sono uguali, si traslino di una posizione a destra tutti i caratteri della copia inferiore. Si arresti tale procedimento di traslazione non appena tutti i caratteri sovrapposti nelle due copie siano identici, oppure quando non ci siano più caratteri sovrapposti. Il nuovo valore di "backtrack" da assegnare all'indice $j$, che indichiamo con "$back[j]$", è proprio uguale al numero di caratteri sovrapposti più uno (se non ci sono caratteri sovrapposti, $back[j]$ risulta ovviamente uguale ad 1, come nella RICERCABRUTA). Per $2 \leq j \leq m$, si ha:

$$back[j] = max\{h:\ h \leq j - 2 \text{ e } P[1..h - 1] = P[j - h + 1..j - 1]\}.$$

Per $j = 1$ è conveniente porre $back[1] = 0$, come sarà chiarito fra breve.

**Esempio 12.8** (Valori di backtrack). I valori di $back[j]$ per il pattern $P = 10110110$ sono i seguenti, accanto ai quali sono evidenziate per chiarezza anche le massime sovrapposizioni tra le due copie dei primi $j - 1$ caratteri di $P$ che portano ai valori ottenuti:

```
back[1] = 0                    back[2] = 1   1
                                               1

back[3] = 1   10               back[4] = 2   101
                 10                              101

back[5] = 2   1011             back[6] = 3   10110
                 1011                            10110

back[7] = 4   101101           back[8] = 5   1011011
                 101101                          1011011
```

□

In altri termini, quando si verifica se c'è una copia di $P$ a partire da $T[k]$ e risulta $P[1..j - 1] = T[k..i - 1]$ ma $P[j] \neq T[i]$, con $k \leq i \leq k + m - 1$, allora la prossima posizione di $T$ per tentare di riconoscere $P$ è $i - back[j] + 1$. Ma poiché, per definizione di $back[j]$, i primi $back[j] - 1$ caratteri di $P[1..j - 1]$ coincidono con gli ultimi, si ha che $P[1..back[j] - 1] = T[i - back[j] + 1..i - 1]$. Pertanto, $i$ non viene modificato e si va a verificare se c'è una copia di $P[back[j]..m]$ a partire da $T[i]$.

L'algoritmo di Knuth, Morris e Pratt, sotto forma di funzione Pascal KMP, è ottenuto da RICERCABRUTA apportando alcune modifiche. Innanzitutto è introdotto in fase di inizializzazione il calcolo del vettore $back$, effettuato con una opportuna procedura CALCOLABACK. È poi eliminato l'indice $k$, divenuto un inutile doppione dell'indice $i$ dato che non è effettuato più alcun "bactrack" su $i$. Il "backtrack" sull'indice $j$ viene effettuato con l'assegnamento "$j := back[j]$", che sostituisce "$j := 1$", tranne in un caso particolare. Infatti, se $j$ è uguale ad 1 al momento del "backtrack", allora significa che $P[1] \neq T[i]$ e che all'iterazione successiva occorre nuovamente considerare $P[1]$, ma confrontandolo con $T[i + 1]$. Siccome $j$ è divenuto uguale a 0 con l'assegnamento "$j := back[j]$", essendo $back[1] = 0$ per definizione, allora $j$ è subito reimpostato ad 1 ed $i$ è incrementato. Questo è il motivo per il quale è stato definito $back[1] = 0$. La funzione KMP restituisce un intero con lo stesso significato di RICERCABRUTA; in particolare, se il pattern è stato interamente riconosciuto (cioè se $j > m$) allora la posizione di $T$ a partire dalla quale si trova l'occorrenza di $P$ è $i - m$.

```
function KMP(var P, T: vettore; n, m: integer): integer;
  var i, j: integer; back: vettore;
  begin
    CALCOLABACK(P, back, m);
    i := 1; j := 1;
    while (i ≤ n) and (j ≤ m) do
      if (j = 0) or (T[i] = P[j])[1]
        then begin
          i := i + 1;
          j := j + 1
        end else j := back[j];
    if j > m then KMP := i − m
      else KMP := i;
  end;
```

Per inizializzare il vettore *back* si riutilizza una lieve variante dello stesso algoritmo, in cui però si confronta il pattern $P$ con se stesso. I valori di $back[j]$ sono computati per $j = 1, 2, \ldots, m$ in accordo alla definizione $max\{h: h \leq j-2$ e $P[1..h-1] = P[j-h+1..j-1]\}$. $back[1]$ e $h$ sono inizializzati a 0 prima del ciclo, mentre $back[1] = 1$, poiché alla prima iterazione $h = 0$. Se $P[j] \neq P[h]$, allora si tenta con un valore più piccolo di $h$, il cui valore di "backtrack" $back[h]$ è stato quindi già computato. Quando invece $j$ ed $h$ sono entrambi incrementati, si ha che $P[1..h-1] = P[j-h-1..j-1]$ e quindi $back[j] = h$.

```
procedure CALCOLABACK(var P, back: vettore; m: integer);
  var j, h: integer;
  begin
    back[1] := 0;
    j := 1;
    h := 0;
    while j ≤ m do
      if (h = 0) or (P[j] = P[h])
        then begin
          j := j + 1;
          h := h + 1;
          back[j] := h
        end else h := back[h];
  end;
```

**Esempio 12.9** (Esecuzione CALCOLABACK). Eseguendo la procedura CALCOLABACK sulla stringa $P$ dell'Esempio 12.4, si ha:

[1] Nel valutare un'espressione $c_1$ **or** $c_1$ **or** $\cdots$ **or** $c_N$, la maggior parte dei compilatori procede da sinistra verso destra, arrestando la valutazione al primo $c_k$ = **true**. Pertanto, nella condizione "$(j = 0)$ **or** $(T[i] = P[j])$" la valutazione di $P[j]$ è effettuata solo se $j \neq 0$ e non dovrebbe essere riportato un errore di indirizzamento fuori dal vettore $P$. Comunque, è possibile definire $P$ nell'intervallo $0..m$, utilizzando $P[0]$ come sentinella in modo da evitare eventuali errori di compilazione dovuti ad indirizzamenti fuori dal vettore.



| | | | |
|---|---|---|---|
| $P =$ | 10110110 | $j = 1, h = 0,$ | |
| | | $j = 2, h = 1,$ | $back[2] = 1,$ |
| $P =$ | 10110110 | $j = 2, h = 1,$ | |
| | 1 | $j = 2, h = back[1] = 0,$ | |
| | | $j = 3, h = 1,$ | $back[3] = 1,$ |
| $P =$ | 10110110 | $j = 3, h = 1,$ | |
| | 1 | $j = 4, h = 2,$ | $back[4] = 2,$ |
| | 10 | $j = 4, h = back[2] = 1,$ | |
| $P =$ | 10110110 | $j = 4, h = 1,$ | |
| | 1 | $j = 5, h = 2,$ | $back[5] = 2,$ |
| | 10 | $j = 6, h = 3,$ | $back[6] = 3,$ |
| | 101 | $j = 7, h = 4,$ | $back[7] = 4,$ |
| | 1011 | $j = 8, h = 5,$ | $back[8] = 5.$ |

□

La procedura CALCOLABACK può essere ulteriormente migliorata perché computa $back[j]$ senza tener conto dell'esito del confronto tra $P[j]$ e $P[h]$. Infatti, se la funzione KMP effettua un "backtrack" sull'indice $j$ per un certo valore dell'indice $i$, allora deve essere $T[i] \neq P[j]$. Ma in tal caso, se $P[j] = P[h]$ allora anche $T[i] \neq P[h]$. Pertanto, poiché $back[j] = h$, la funzione KMP effettuerà un secondo backtrack su $j$ senza modificare $i$. Ciò può essere evitato assegnando direttamente $back[h]$ a $back[j]$ qualora $P[j] = P[h]$. A tal fine, è sufficiente modificare la CALCOLABACK sostituendo l'assegnamento "$back[j] := h$" con l'istruzione:

**if** $P[j] = P[h]$ **then** $back[j] := back[h]$ **else** $back[j] := h$

Infatti, poiché il vettore *back* è riempito da sinistra verso destra ed $h < j$, il valore $back[h]$ è stato già computato e può essere utilizzato per calcolare $back[j]$. Questa semplice modifica garantisce che per ciascun valore dell'indice $i$ ci sia al più un solo backtrack sull'indice $j$.

**Esempio 12.10** (CALCOLABACK modificata). Si riconsideri il vettore *back* per $P =$ `10110110`. Applicando la precedente modifica si ottiene:

| | |
|---|---|
| $back[1] = 0,$ | |
| $back[2] = 1,$ | perché $0 = P[2] \neq P[1] = 1,$ |
| $back[3] = back[1] = 0,$ | perché $1 = P[3] = P[1] = 1,$ |
| $back[4] = 2,$ | perché $1 = P[4] \neq P[2] = 0,$ |
| $back[5] = back[2] = 1,$ | perché $0 = P[5] = P[2] = 0,$ |
| $back[6] = back[3] = 0,$ | perché $1 = P[6] = P[3] = 1,$ |
| $back[7] = back[4] = 2,$ | perché $1 = P[7] = P[4] = 1,$ |
| $back[8] = back[5] = 1,$ | perché $0 = P[8] = P[5] = 0.$ |

□

Durante l'esecuzione della funzione KMP, l'indice $j$ può essere decrementato al più una volta per ciascun valore dell'indice $i$. Poiché $i$ può assumere $O(n)$ valori diversi, il numero di confronti tra caratteri di $P$ e di $T$ eseguiti nel ciclo **while** è $O(n)$. D'altronde, per le stesse ragioni, la procedura CALCOLABACK modificata richiede $O(m)$ confronti tra caratteri. Pertanto, la complessità della funzione KMP è $O(m+n)$.

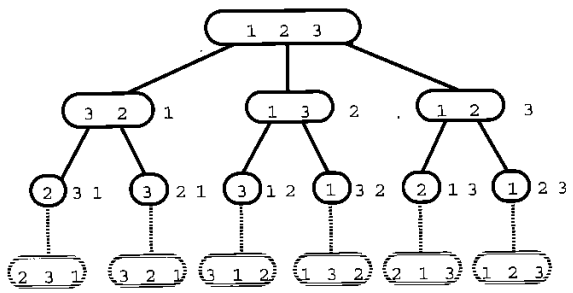


Figura 12.5: Generazione di tutte le permutazioni di $A$ con $A[1] = 1, A[2] = 2$, e $A[3] = 3$.

**Esempio 12.11** (Algoritmo di Knuth, Morris e Pratt). Eseguendo la funzione KMP sulle stringhe $P$ e $T$ dell'Esempio 12.4 ed utilizzando il vettore *back* dell'Esempio 12.10, si ottiene:

```
T =  10110010101101011011011   i = 1, j = 1,
P =  10110110                  i = 6, j = 6, back[6] = 0,

T =  10110010101101011011011   i = 7, j = 1,
P =        10110110            i = 10, j = 4, back[4] = 2,

T =  10110010101101011011011   i = 10, j = 2,
P =           10110110         i = 15, j = 7, back[7] = 2,

T =  10110010101101011011011   i = 15, j = 2,
P =                 10110110   i = 22, j = 8, EUREKA!
```

□

## 12.3 Esercizi

**Esercizio 12.1** (Angoli $\alpha$). Se $dx$ e $dy$ sono le differenze tra le ascisse e le ordinate dei punti $p$ $i$ e $p_1$, calcolare l'angolo $\alpha = \tan^{-1} dy/dx$ con la funzione di libreria Pascal per l'arcotangente richiede parecchio tempo. Dato che $\alpha$ serve solo nella fase di ordinamento dei punti dell'algoritmo di Graham, si può utilizzare una quantità diversa dall'angolo $\alpha$, più veloce da calcolare e che produca lo stesso ordinamento. Si dimostri che $dy/(dy + dx)$ è adatta allo scopo e si scriva una opportuna funzione Pascal per calcolarla in tempo O(1), tenendo conto dei casi in cui $dx$ e $dy$ possono essere positive, negative, o nulle.

**Esercizio 12.2** (Calcolo vettore *back*). Si calcoli il vettore *back* utilizzato dalla funzione KMP per il pattern $P =$ `abracadabra`.

**Esercizio 12.3** (Funzione KMP su lista). Si riscriva la funzione KMP assumendo che il pattern $P$ ed il testo $T$ siano memorizzati con liste realizzate con puntatori.

**Esercizio 12.4** (Tutte le permutazioni). Si scriva una procedura Pascal basata sulla tecnica "backtrack" per stampare tutte le permutazioni degli $n$ elementi contenuti in un vettore $A[1..n]$. (Suggerimento: si applichi la definizione ricorsiva secondo la quale tutte le permutazioni di $n$ elementi possono essere viste come tutte le permutazioni di $n - 1$ elementi seguite a turno dall'elemento mancante $A[i]$, per $i = 1, 2, \ldots, n$, come mostrato nella Fig. 12.5).