```
SommaMassima1( a ):        ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
  max = 0;
  FOR (i = 0; i < n; i = i+1) {
    FOR (j = i; j < n; j = j+1) {
      somma = 0;
      FOR (k = i; k <= j; k = k+1)
        somma = somma + a[k];
      IF (somma > max) max = somma;
    }
  }
  RETURN max;
```

Codice 2.4 Prima soluzione per il segmento di somma massima.

```
SommaMassima2( a ):        ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
  max = 0;
  FOR (i = 0; i < n; i = i+1) {
    somma = 0;
    FOR (j = i; j < n; j = j+1) {
      somma = somma + a[j];
      IF (somma > max) max = somma;
    }
  }
  RETURN max;
```

Codice 2.5 Seconda soluzione per il segmento di somma massima.

```
SommaMassima3( a ):        ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
  max = 0;
  somma = max;
  FOR (j = 0; j < n; j = j+1) {
    IF (somma > 0) {
      somma = somma + a[j];
    } ELSE {
      somma = a[j];
    }
    IF (somma > max) max = somma;
  }
  RETURN max;
```

Codice 2.6 Terza soluzione per il segmento di somma massima.

TEMPI	10	10^2	10^3	10^4
quadr	0	0.03	38	—
quadr	0	0.01	0.5	46
lineare	0	0	0	0.01

```
RicercaSequenziale( a, k ):                    pre: la lunghezza di a
  trovato = FALSE;
  indice = -1;
  FOR (i = 0; (i<n) && (!trovato); i = i+1) {
    IF (a[i] == k) {
      trovato = TRUE;
      indice = i;
    }
  }
  RETURN indice;
```

Codice 2.7 Ricerca sequenziale di una chiave k in un array a.

```
RicercaBinariaIterativa( a, k ):          (pre: la lunghezza di a è n)
  sinistra = 0;                                  ordinato
  destra = n-1;
  trovato = FALSE;
  indice = -1;
  WHILE ((sinistra <= destra) && (!trovato)) {
    centro = (sinistra+destra)/2;
    IF (a[centro] > k) {
      destra = centro-1;
    } ELSE IF (a[centro] < k) {
      sinistra = centro+1;
    } ELSE {
      indice = centro;
      trovato = TRUE;
    }
  }
  RETURN indice;
```

Codice 2.8 Ricerca binaria di una chiave k in un array a.

```
RicercaBinariaRicorsiva( a, k, sinistra, destra ):
                                  (pre: 0 ≤ sinistra ≤ destra ≤ n - 1
                                   a vettore ordinato
  IF (sinistra == destra) {
    IF (k == a[sinistra]) {
      RETURN sinistra;
    } ELSE {
      RETURN -1;
    }
  }
  centro = (sinistra+destra)/2;
  IF (k <= a[centro]) {
    RETURN RicercaBinariaRicorsiva( a, k, sinistra, centro );
  } ELSE {
    RETURN RicercaBinariaRicorsiva( a, k, centro+1, destra );
  }
```

Codice 2.9 Ricerca binaria di una chiave k in un array a con il paradigma del divide et impe

```
InsertionSort( a ):                                    ⟨pre: la lunghezza di a è n⟩
  FOR (i = 0; i < n; i = i+1) {
    prossimo = a[i];
    j = i;
    WHILE ((j > 0) && (a[j-1] > prossimo)) {
      a[j] = a[j-1];
      j = j-1;
    }
    a[j] = prossimo;
  }
```

Codice 2.3 Ordinamento per inserimento di un array a.

```
SelectionSort( a ):                                    ⟨pre: la lunghezza di a è n⟩
  FOR (i = 0; i < n; i = i+1) {
    minimo = a[i];
    indiceMinimo = i;
    FOR (j = i+1; j < n; j = j+1) {
      IF (a[j] < minimo) {
        minimo = a[j];
        indiceMinimo = j;
      }
    }
    a[indiceMinimo] = a[i];
    a[i] = minimo;
  }
```

Codice 2.2 Ordinamento per selezione di un array a.

```
MergeSort( a, sinistra, destra ):                    pre: 0 ≤ sinistra.destra ≤ n −
  IF (sinistra < destra) {
    centro = (sinistra+destra)/2;
    MergeSort( a, sinistra, centro );
    MergeSort( a, centro+1, destra );
    Fusione( a, sinistra, centro, destra );
  }
```

Codice 2.11 Ordinamento per fusione di un array a.

```
Fusione( a, sx, cx, dx ):                    pre: 0 ≤ sx ≤ cx ≤ dx ≤ n −
  i = sx;
  j = cx+1;
  k = 0;
  WHILE ((i <= cx) && (j <= dx)) {
    IF (a[i] <= a[j]) {
      b[k] = a[i];
      i = i+1;
    } ELSE {
      b[k] = a[j];
      j = j+1;
    }
    k = k+1;
  }
  FOR ( ; i <= cx; i = i+1, k = k+1)
    b[k] = a[i];
  FOR ( ; j <= dx; j = j+1, k = k+1)
    b[k] = a[j];
  FOR (i = sx; i <= dx; i = i+1)
    a[i] = b[i-sx];
```

Codice 2.12 Fusione di due segmenti adiacenti ordinati.

```
QuickSort( a, sinistra, destra ):
                                        (pre: 0 ≤ sinistra, destra ≤ n − 1)
  IF (sinistra < destra) {
    scegli pivot nell'intervallo [sinistra...destra];
    perno = Distribuzione( a, sinistra, pivot, destra );
    QuickSort( a, sinistra, perno-1 );
    QuickSort( a, perno+1, destra );
  }
```

Codice 2.13 Ordinamento per distribuzione di un array a.

```
Distribuzione( a, sx, px, dx ):          (pre: 0 ≤ sx ≤ px ≤ dx ≤ n −
  IF (px != dx) Scambia( px, dx );
  i = sx;
  j = dx-1;
  WHILE (i <= j) {
    WHILE ((i <= j) && (A[i] <= A[dx]))
      i = i+1;
    WHILE ((i <= j) && (A[j] => A[dx]))
      j = j-1;
    IF (i < j) Scambia( i, j );
  }
  IF (i != dx) Scambia( i, dx );
  RETURN i;

Scambia( i, j ):                                    (pre: sx ≤ i, j ≤
  temp = a[j]; a[j] = a[i]; a[i] = temp;
```

Codice 2.14 Distribuzione in loco degli elementi di un segmento a[sx. dx] in base alla pc px scelta per il pivot.

```
QuickSelect( a, sinistra, r, destra ):
                              (pre: 0 ≤ sinistra ≤ r − 1 ≤ destra ≤ n −
  IF (sinistra == destra) {
    RETURN a[sinistra];
  } ELSE {
    scegli pivot nell'intervallo [sinistra...destra];
    perno = Distribuzione( a, sinistra, pivot, destra );
    IF (r-1 <= perno) {
      QuickSelect( a, sinistra, r, perno-1 );
    } ELSE {
      QuickSelect( a, perno+1, r, destra );
    }
  }
```

Codice 2.15 Selezione dell'elemento di rango r per distribuzione in un array a.

```
// Calcola il numero massimo di cifre negli interi di A
function maxLength(A)
{
      var numCifre = 0;
      for(var i=0; i < A.length; i++){
            var s = Math.round(A[i]).toString();
            if (s.length > numCifre)
                  numCifre = s.length;
            }
      return numCifre;
}


// Restituisce la cifra i del numero n, contando da destra
function PrendiCifra(n,i)
{
      var s = n.toString();
      if (i >= s.length) return 0;
      return s.charAt(s.length - 1 - i);
}


// Ordina stabilmente in accordo alla cifra di posizione "posCifra"
function passata(A, posCifra)
 {
      var counter = new Array(10);
      var temp = new Array();
      var N = A.length;
      var i, d;

      for( d = 0; d <= 9; d++ ) counter[d] = 0;
      for( i = 0; i < N; i++ ) counter[ PrendiCifra(A[i],posCifra) ]
++;
      for( d = 1; d <= 9; d++ ) counter[d] += counter[d-1];

      // Decremento di uno il counter[] perché vettore parte da zero
      for( i = N-1; i >= 0; i-- )
            temp[ --counter[ PrendiCifra(A[i],posCifra) ] ] = A[i];

      // Ricopia in A il risultato dell'ordinamento sulla cifra
"posCifra"
      for( i = 0; i < N; i++ ) A[i] = temp[i];
}

// Procedura principale che ordina partendo dalla cifra meno
significativa
function RadixSort(A)
 {
      var maxLen = maxLength(A);
      for(var p=0; p < maxLen; p++ )
            passata(A, p);
 }
```

```
TorriHanoi( n, primo, secondo, terzo ):
  IF (n = 1) {
    PRINT primo ⟶ terzo;
  } ELSE {
    TorriHanoi( n - 1, primo, terzo, secondo );
    PRINT primo ⟶ terzo;
    TorriHanoi( n - 1, secondo, primo, terzo );
  }
```

```
TorriHanoiGen( n, k ):                    ⟨pre: n > 0 multiplo di k − 2⟩
  FOR (i = 1; i <= k-2; i = i+1)
    TorriHanoi(n/(k-2), 0, k-1, i);
  FOR (i = k-2; i >= 1; i = i-1)
    TorriHanoi(n/(k-2), i, 0, k-1);
```

Poiché il corpo del ciclo `WHILE` è vuoto, per ogni programma A, osserviamo che `Paradosso(A)` termina se e solo se la guardia `Termina(A, A)` restituisce il valore FALSE, ovvero se e solo se il programma A non termina quando viene eseguito sui dati d'ingresso A. Possiamo quindi concludere che `Paradosso(Paradosso)` termina se e solo se la guardia `Termina(Paradosso,Paradosso)` restituisce il valore FALSE, ovvero se e solo se il programma `Paradosso` non termina quando viene eseguito sui dati d'ingresso `Paradosso`. In breve, `Paradosso(Paradosso)` termina se e solo se `Paradosso(Paradosso)` non termina!

Questa contraddizione deriva dall'aver assunto l'esistenza di `Termina`, l'unico anello debole del filo logico tessuto nell'argomentazione precedente. Quindi, un tale programma non può esistere e, pertanto, diciamo che il problema della fermata è **indecidibile**. Purtroppo esso non è l'unico: per esempio, stabilire se due programmi A e B sono equivalenti, ovvero producono sempre i medesimi risultati a parità di dati in ingresso, è anch'esso un problema indecidibile. Notiamo che l'uso di uno specifico linguaggio non influisce su tali risultati di indecidibilità, i quali valgono per qualunque modello di calcolo che possa formalizzare il comportamento di un calcolatore (più precisamente di una macchina di Turing).

1.2 Trattabilità di problemi computazionali

L'esistenza di problemi indecidibili restringe la possibilità di progettare algoritmi e programmi ai soli problemi decidibili. In questo ambito, non tutti i problemi risultano risolvibili in tempo ragionevole, come testimoniato dal noto problema delle **Torri di Hanoi**, un gioco del XIX secolo inventato da un matematico francese, Edouard Lucas, legandolo alla seguente leggenda indiana (probabilmente falsa) sulla divinità Brahma e sulla fine del mondo. In un tempio induista dedicato alla divinità, vi sono tre pioli di cui il primo contiene $n = 64$ dischi d'oro impilati in ordine di diametro decrescente, con il disco più ampio in basso e quello più stretto in alto (gli altri due pioli sono vuoti). Dei monaci *sannyasin* spostano i dischi dal primo al terzo piolo usando il secondo come appoggio, con la regola di non poter spostare più di un disco alla volta e con quella di non porre mai un disco di diametro maggiore sopra un disco di diametro inferiore. Quando i monaci avranno terminato di spostare tutti i dischi nel terzo piolo, avverrà la fine del mondo.

La soluzione di questo gioco è semplice da descrivere usando la ricorsione. Supponiamo di avere spostato ricorsivamente i primi $n - 1$ dischi sul secondo piolo, usando il terzo come appoggio. Possiamo ora spostare il disco più grande dal primo al terzo piolo, e quindi ricorsivamente spostare gli $n - 1$ dischi dal secondo al terzo piolo usando il primo come appoggio.

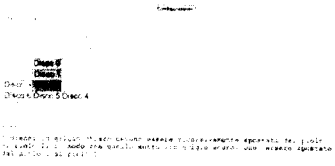
```
TorriHanoi( n, primo, secondo, terzo ):
  IF (n = 1) {
    PRINT primo ⟶ terzo;
  } ELSE {
    TorriHanoi( n - 1, primo, terzo, secondo );
    PRINT primo ⟶ terzo;
    TorriHanoi( n - 1, secondo, primo, terzo );
  }
```

Dimostriamo, per induzione sul numero n di dischi, che il numero di mosse effettuate eseguendo tale programma e stampate come "origine ⟶ destinazione", è pari a $2^n - 1$: il caso base $n = 1$ è immediato; nel caso $n > 1$ occorrono $2^{n-1} - 1$ mosse per ciascuna delle due chiamate ricorsive per ipotesi induttiva, a cui aggiungiamo la mossa nella riga 6, per un totale di $2 \times (2^{n-1} - 1) + 1 = 2^n - 1$ mosse. Purtroppo, non c'è speranza di trovare un programma che effettui un numero di mosse inferiore a tale quantità, in quanto è stato dimostrato che le $2^n - 1$ mosse sono necessarie e non è possibile impiegarne di meno.

Nel problema originale con $n = 64$ dischi, supponendo che ogni mossa richieda un secondo, occorrono $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ secondi, che equivalgono a circa 584 942 417 355 anni, ovvero quasi 585 miliardi di anni: per confronto, la teoria del *big bang* asserisce che l'Universo è stato creato da un'esplosione cosmica in un periodo che risale a circa 10–20 miliardi di anni fa.

ALVIE: problema delle Torri di Hanoi

Osserva, sperimenta e verifica
HanoiTower



Le Torri di Hanoi mostrano dunque che, anche se un problema è decidibile ovvero è risolubile mediante un algoritmo, non è detto che l'algoritmo stesso possa sempre risolverlo in tempi ragionevoli: ciò è dovuto al fatto che il numero di mosse e quindi il tempo di esecuzione del programma, è **esponenziale** nel numero n di dischi (n appare all'esponente di $2^n - 1$). Il tempo necessario per spostare i dischi diventa dunque rapidamente insostenibile, anche per un numero limitato di dischi, come illustrato nella seguente tabella, in cui il tempo di esecuzione è espresso in secondi (s), minuti (m), ore (h), giorni (g) e anni (a).

n	5	10	15	20	25	30	35	40	45
tempo	32 s	17 m	9 h	12 g	1 a	34 a	1089 a	34865 a	1115689 a

L'esponenzialità del tempo di esecuzione rende anche limitato l'effetto di eventuali miglioramenti nella velocità di esecuzione dei singoli passi, perché, in tal caso, basta aumentare di poco il numero n di dischi per vanificare ogni miglioramento. Supponiamo infatti di poter eseguire $m = 2^s$ operazioni in un secondo, invece di una singola operazione al secondo: in tal caso, anziché circa 2^n secondi, ne occorrono $2^n/m = 2^{n-s}$ per spostare gli n dischi. L'effetto di tale miglioramento viene però neutralizzato molto rapidamente al crescere del numero di dischi in quanto è sufficiente portare tale numero a $n + s$ (dove $s = \log m$) per ottenere lo stesso tempo complessivo di esecuzione. In altre parole, un miglioramento delle prestazioni per un fattore *moltiplicativo* si traduce in un aumento solo *additivo* del numero di dischi trattabili. La tabella seguente esemplifica questo comportamento nel caso $n = 64$, mostrando il numero di dischi gestibili in un tempo pari a 18 446 744 073 709 551 615 secondi, al variare della velocità di esecuzione: come possiamo vedere, miglioramenti anche molto importanti di quest'ultima si traducono in piccoli incrementi del numero di dischi che il programma è in grado di gestire.

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	67	70	73	77	80	83	93

Di diversa natura è invece l'andamento **polinomiale**, come possiamo mostrare se consideriamo la generalizzazione del problema delle Torri di Hanoi al caso in cui siano disponibili $k > 3$ pioli. A tale scopo, supponiamo che i pioli siano numerati da 0 a $k-1$ e che il problema consista nello spostare i dischi dal piolo 0 al piolo $k-1$ (rispettando le regole sopra descritte). In tal caso, possiamo usare il codice `TorriHanoi` come sottoprogramma all'interno della seguente soluzione al problema generalizzato (per semplicità di esposizione, supponiamo che $n > 0$ sia un multiplo di $k-2$).

```
TorriHanoiGen( n, k ):                    ⟨pre: n > 0 multiplo di k − 2⟩
  FOR (i = 1; i <= k-2; i = i+1)
    TorriHanoi(n/(k-2), 0, k-1, i);
  FOR (i = k-2; i >= 1; i = i-1)
    TorriHanoi(n/(k-2), i, 0, k-1);
```

Intuitivamente, il codice precedente divide gli n dischi in $k-2$ gruppi di $\frac{n}{k-2}$ dischi ciascuno. Il primo ciclo sposta, per ogni i, l'i-esimo gruppo dal disco 0 al disco i, usando il disco $k-1$ come appoggio e invocando `TorriHanoi`, mentre il secondo ciclo sposta tale gruppo dal disco i al disco $k-1$ usando il disco 0 come appoggio: notiamo che, per rispettare la regola di sovrapposizione dei dischi, il secondo ciclo scorre i gruppi in ordine inverso rispetto al primo. Il numero di mosse è dunque pari a $2 \times (k-2)$ volte il numero di mosse richiesto per spostare $\frac{n}{k-2}$ dischi usando tre pioli. Non è difficile estendere il suddetto codice in modo che funzioni per tutti i valori di n (anche quando

n non è un multiplo di k − 2), osservando che il numero totale di mosse è pari a

$$M(n,k) = 2 \times (k-2) \times \left(2^{\frac{n}{k-2}} - 1\right)$$

Ponendo $k^* = \left\lfloor \frac{n}{\log n} \right\rfloor$ per $n \geq 5$, possiamo verificare che $M(n,k^*) \leq n^2$. In tal caso, il problema delle Torri di Hanoi può quindi essere risolto con un numero di mosse quadratico nel numero dei dischi (notiamo che è in generale un problema aperto stabilire il numero minimo di mosse per ogni n e per ogni $k > 3$). Per esempio, volendo spostare gli $n = 64$ dischi del problema originale usando $k^* = 10$ pioli, sono sufficienti soltanto $64^2 = 4096$ secondi contro i $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ secondi necessari nel caso di tre pioli (supponendo di poter effettuare una mossa al secondo).

Il problema delle Torri di Hanoi con tre o più pioli illustra come una soluzione che richiede un numero esponenziale di passi risulti irragionevole se confrontata con una che ne richiede un numero polinomiale. In generale, il passaggio da un andamento esponenziale a uno polinomiale ha due importanti conseguenze. In primo luogo, un polinomio cresce molto più lentamente di una qualunque funzione esponenziale, come mostrato nella seguente tabella relativa al polinomio n^2 e analoga a quella vista nel caso della funzione 2^n.

n	5	10	15	20	25	30	35	40	45
tempo	25 s	100 s	225 s	7 m	11 m	15 m	21 m	27 m	34 m

In secondo luogo, la polinomialità del tempo di esecuzione rende molto più efficaci gli eventuali miglioramenti nella velocità di esecuzione dei singoli passi. Ad esempio, nel caso del problema generalizzato delle Torri di Hanoi con n dischi e k^* pioli, potendo eseguire m operazioni in un secondo, invece di una singola operazione al secondo, occorrerebbero $\frac{n^2}{m} = (n/\sqrt{m})^2$ secondi per spostare gli n dischi. L'effetto di tale miglioramento permane a lungo in quanto è necessario portare il numero di dischi a $n \times \sqrt{m}$ per ottenere lo stesso tempo complessivo di esecuzione. In altre parole, un miglioramento di un fattore *moltiplicativo* nelle prestazioni si traduce in un aumento anch'esso *moltiplicativo* del numero di dischi trattabili. La tabella seguente (analoga a quella vista nel caso della funzione 2^n) esemplifica questo comportamento nel caso $n = 64$, mostrando il numero di dischi gestibili (con k^* pioli) in un tempo pari a 4096 secondi, al variare della velocità di esecuzione: come possiamo vedere, miglioramenti di quest'ultima si traducono in incrementi significativi del numero di dischi che il programma è in grado di gestire.

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	202	640	2023	6400	20238	64000	2023857

1.2.1 Rappresentazione e dimensione dei dati

Volendo generalizzare la discussione fatta nel caso delle Torri di Hanoi a un qualunque problema computazionale, è anzitutto necessaria una breve escursione nella rappresentazione e nella codifica dei dati elementari utilizzati dal calcolatore. Secondo quanto detto in riferimento alla teoria dell'informazione di Claude Shannon, il bit (*binary digit*) segnala la presenza (1) oppure l'assenza (0) di un segnale o di un evento con due possibilità equiprobabili.[4]

La stessa sequenza di bit può essere interpretata in molti modi, a seconda del significato che le vogliamo assegnare nel contesto in cui la usiamo: può essere del codice da eseguire oppure dei dati da elaborare, come abbiamo visto nel problema della fermata. In particolare, gli interi nell'insieme $\{0, 1, \ldots, 2^k - 1\}$ possono essere codificati con k bit $b_{k-1}b_{k-2}\cdots b_1 b_0$. La regola per trasformare tali bit in un numero intero è semplice: basta moltiplicare ciascuno dei bit per potenze crescenti di 2, a partire dal bit meno significativo b_0, ottenendo $\sum_{i=0}^{k-1} b_i \times 2^i$. Per esempio, la sequenza 0101 codifica il numero intero $5 = 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$. La regola inversa può essere data in vari modi, e l'idea è quella di sottrarre ogni volta la massima potenza del 2 fino a ottenere 0. Per rappresentare sia numeri positivi che negativi è sufficiente aggiungere un bit di segno.

I caratteri sono codificati come interi su $k = 8$ bit (ASCII) oppure su $k = 16$ bit (*Unicode*/UTF8). La codifica riflette l'ordine alfabetico, per cui la lettera 'A' viene codificata con un intero più piccolo della lettera 'Z' (bisogna porre attenzione al fatto che il carattere '7' non è la stessa cosa del numero 7). Le stringhe sono sequenze di caratteri alfanumerici che vengono perciò rappresentate come sequenze di numeri terminate da un carattere speciale oppure a cui vengono associate le rispettive lunghezze.

I numeri reali sono codificati con un numero limitato di bit a precisione finita di 32 o 64 bit nello standard IEEE754 (quindi sono piuttosto dei numeri razionali). Il primo bit è utilizzato per il segno; un certo numero dei bit successivi codifica l'esponente, mentre il resto dei bit serve per la mantissa. Per esempio, la codifica di $-0,275 \times 2^{18}$ è ottenuta codificando il segno meno, l'esponente 18, e quindi la mantissa 0,275 (ciascuno con il numero assegnato di bit).

Infine, in generale, un insieme finito è codificato come una sequenza di elementi separati da un carattere speciale per quell'insieme: questa codifica ci permetterà, se necessario, di codificare anche insiemi di insiemi, usando gli opportuni caratteri speciali di separazione.

Le regole di codifica discusse finora, ci consentono, per ogni dato, di ricavarne una rappresentazione binaria: nel definire la **dimensione del dato**, faremo riferimento alla lunghezza di tale rappresentazione o a una misura equivalente.

[4] Il bit viene usato come unità di misura: 1 *byte* = 8 bit, 1 *kilobyte* (KB) = 2^{10} byte = 1024 byte, 1 *megabyte* (MB) = 2^{10} KB = 1 048 576 byte, 1 *gigabyte* (GB) = 2^{10} MB = 1 073 741 824 byte, 1 *terabyte* (TB) = 2^{10} GB, 1 *petabyte* (PB) = 2^{10} TB e così via.

```
GeneraBinarie( A, b ):                    (pre: i primi b bit in A sono da generare)
  IF (b == 0) {
    Elabora( A );
  } ELSE {
    A[b-1] = 0;
    GeneraBinarie( A, b-1 );
    A[b-1] = 1;
    GeneraBinarie( A, b-1 );
  }
```

```
GeneraPermutazioni( A, p ):   (pre: i primi p elementi di A sono da permutare)
  IF (p == 0) {
    Elabora( A );
  } ELSE {
    FOR (i = p-1; i >= 0; i = i-1) {
      Scambia( i, p-1 );
      GeneraPermutazioni( A, p-1 );
      Scambia( i, p-1 );
    }
  }
```

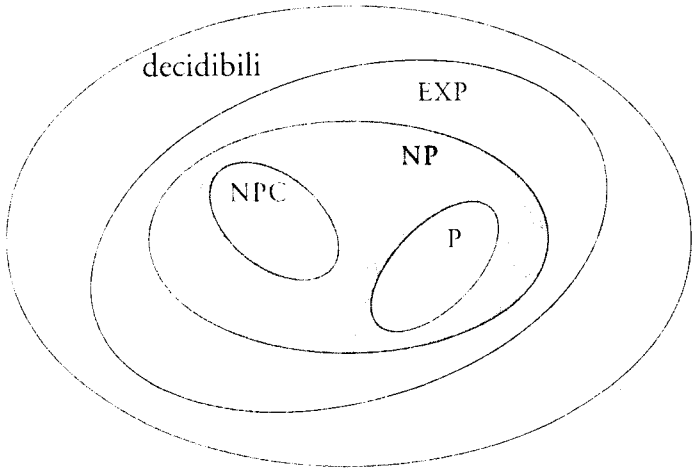


Figura 1.2 Una prima classificazione dei problemi computazionali decidibili.

1.2.2 Algoritmi polinomiali ed esponenziali

Abbiamo già osservato che, tranne che per piccole quantità di dati, un algoritmo che impiega un numero di passi esponenziale è impossibile da usare quanto un algoritmo che non termina! Nel seguito useremo il termine **algoritmo polinomiale** per indicare un algoritmo, per il quale esiste una costante $c > 0$, il cui numero di passi elementari sia al massimo pari a n^c per ogni dato in ingresso di dimensione n. Questa definizione ci porta a una prima classificazione dei problemi computazionali come riportato nella Figura 1.2 dove, oltre alla divisione in problemi indecidibili e decidibili, abbiamo l'ulteriore suddivisione di quest'ultimi in **problemi trattabili** (per i quali esiste un algoritmo risolutivo polinomiale) e **problemi intrattabili** (per i quali un tale algoritmo non esiste): facendo riferimento alla figura, tali classi di problemi corrispondono rispettivamente a P e EXP − P, dove EXP rappresenta la classe di problemi risolubili mediante un **algoritmo esponenziale**, ovvero un algoritmo il cui numero di passi è al più esponenziale nella dimensione del dato in ingresso.[5]

Talvolta gli algoritmi esponenziali sono utili per esaminare le caratteristiche di alcuni problemi combinatori sulla base della generazione esaustiva di tutte le istanze di piccola taglia.

Discutiamo un paio di casi, che rappresentano anche un ottimo esempio di uso della ricorsione nella risoluzione dei problemi computazionali. Nel primo esempio, vogliamo

[5] Volendo essere più precisi, i problemi intrattabili sono tutti i problemi decidibili che non sono inclusi in P: tra di essi, quindi, vi sono anche problemi che non sono contenuti in EXP. Nel resto di questo libro, tuttavia, non considereremo mai problemi che non ammettano un algoritmo esponenziale.

generare tutte le 2^n sequenze binarie di lunghezza n, che possiamo equivalentemente interpretare come tutti i possibili sottoinsiemi di un insieme di n elementi.

Per illustrare questa corrispondenza, numeriamo gli elementi da 0 a $n-1$ e associamo il bit in posizione b della sequenza binaria all'elemento b dell'insieme fornito (dove $0 \leqslant b \leqslant n-1$): se tale bit è pari a 1, l'elemento b è nel sottoinsieme così rappresentato; altrimenti, il bit è pari a 0 e l'elemento non appartiene a tale sottoinsieme.

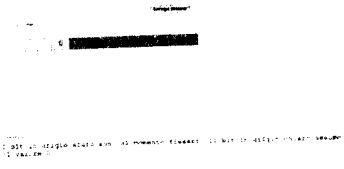
Durante la generazione delle 2^n sequenze binarie, memorizziamo ciascuna sequenza binaria A e utilizziamo la procedura `Elabora` per stampare A o per elaborare il corrispondente sottoinsieme. Notiamo che A viene riutilizzata ogni volta sovrascrivendone il contenuto ricorsivamente: il bit in posizione b, indicato con A[b − 1], deve valere prima 0 e, dopo aver generato tutte le sequenze con tale bit, deve valere 1, ripetendo la generazione.

Il seguente codice ricorsivo permette di ottenere tutte le 2^n sequenze binarie di lunghezza n: inizialmente, dobbiamo invocare la funzione `GeneraBinarie` con input $b = n$.

```
GeneraBinarie( A, b ):                    ⟨pre: i primi b bit in A sono da generare⟩
  IF (b == 0) {
    Elabora( A );
  } ELSE {
    A[b-1] = 0;
    GeneraBinarie( A, b-1 );
    A[b-1] = 1;
    GeneraBinarie( A, b-1 );
  }
```

ALVIE: generazione ricorsiva delle sequenze binarie

Osserva, sperimenta e verifica
`BinaryStringGeneration`



Il secondo esempio riguarda la generazione delle permutazioni degli n elementi contenuti in una sequenza A. Ciascuno degli n elementi occupa, a turno, l'ultima posizione in A e i rimanenti $n-1$ elementi sono ricorsivamente permutati. Per esempio, volendo generare tutte le permutazioni di $n = 4$ elementi a, b, c, d in modo sistematico, possiamo generare prima quelle aventi d in ultima posizione (elencate nella prima colonna), poi quelle aventi c in ultima posizione (elencate nella seconda colonna) e così via:

a b c d	a b d c	a d c b	d b c a
b a c d	b a d c	d a c b	b d c a
a c b d	a d b c	a c d b	d c b a
c a b d	d a b c	c a d b	c d b a
c b a d	d b a c	c d a b	c b d a
b c a d	b d a c	d c a b	b c d a

Restringendoci alle permutazioni aventi d in ultima posizione (prima colonna), possiamo permutare i rimanenti elementi a, b, c in modo analogo usando la ricorsione su questi tre elementi. A tal fine, notiamo che le permutazioni generate per i primi $n - 1 = 3$ elementi, sono identiche a quelle delle altre tre colonne mostrate sopra. Per esempio, se ridenominiamo l'elemento c (nella prima colonna) con d (nella seconda colonna), otteniamo le *medesime* permutazioni di $n - 1 = 3$ elementi; analogamente, possiamo ridenominare gli elementi b e d (nella seconda colonna) con d e c (nella terza colonna), rispettivamente. In generale, le permutazioni di $n - 1$ elementi nelle colonne sopra possono essere messe in corrispondenza biunivoca e, pertanto, ciò che conta sono il numero di elementi da permutare come riportato nel codice seguente. Invocando tale codice con parametro d'ingresso $p = n$, possiamo ottenere tutte le $n!$ permutazioni degli elementi in A:

```
GeneraPermutazioni( A, p ):  ⟨pre: i primi p elementi di A sono da permutare⟩
  IF (p == 0) {
    Elabora( A );
  } ELSE {
    FOR (i = p-1; i >= 0; i = i-1) {
      Scambia( i, p-1 );
      GeneraPermutazioni( A, p-1 );
      Scambia( i, p-1 );
    }
  }
```

Notiamo l'utilizzo di una procedura `Scambia` prima e dopo la ricorsione così da mantenere l'invariante che gli elementi, dopo esser stati permutati, vengono riportati al loro ordine di partenza, come può essere verificato simulando l'algoritmo suddetto.

ALVIE: generazione ricorsiva delle permutazioni

Osserva, sperimenta e verifica
`PermutationGeneration`

```
Ricerca( tabella, k ):
  h = Hash(k);
  p = tabella[h].Ricerca( k );
  IF (p != null) RETURN p.dato ELSE RETURN null;

Inserisci( tabella, e ):
  IF (Ricerca( tabella, e.chiave ) == null) {
    h = Hash( e.chiave );
    tabella[h].InserisciFondo( e );
  }

Cancella( tabella, k ):
  IF (Ricerca( tabella, k ) != null) {
    h = Hash(k);
    tabella[h].Cancella( k );
  }
```

Codice 5.3 Dizionario realizzato mediante tabelle hash con liste di trabocco.

```
Ricerca( tabella, k ):
  FOR (i = 0; i < m; i = i+1) {
    h = Hash[i](k);
    IF (tabella[h] == null) RETURN -1;
    IF (tabella[h].chiave == k) RETURN tabella[h];
  }

Inserisci( tabella, e ):                    (pre: tabella contiene n < m chiavi)
  IF (Ricerca( tabella, e.chiave ) == null) {
    i = -1;
    DO {
      i = i+1;
      h = Hash[i]( e.chiave );
      IF (tabella[h] == null) tabella[h] = e;
    } WHILE (tabella[h] != e);
  }
```

Codice 5.4 Dizionario realizzato mediante tabelle hash con indirizzamento aperto.