

Algoritmi e Strutture Dati

Backtracking

Alberto Montresor

Università di Trento

2024/08/11

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

- 1 Introduzione
- 2 Enumerazione
 - Sottoinsiemi e permutazioni
 - Subset sum
 - Giochi
- 3 Backtracking iterativo
 - Inviluppo convesso

Esercizio

A mia figlia (prima elementare) è stato chiesto di disegnare tutte le possibili sequenze composte da tre pallini rossi e due pallini gialli.

Voi siete all'università, quindi provate a disegnare tutte le sequenze composte da quattro pallini rossi e tre gialli.

Soluzioni ammissibili e soluzioni ottime

Soluzione ammissibile

Dato un problema, una **soluzione ammissibile** è una soluzione che **soddisfa un insieme di criteri**

- **Zaino**: sottoinsieme di oggetti di peso inferiore alla capacità
- **Sottosequenza comune**: una stringa che è sottosequenza di entrambe le stringhe in input

Valutare le soluzioni ammissibili

Nei problemi di ottimizzazione, viene definita una **funzione di costo/guadagno** definita sull'insieme delle soluzioni ammissibili

- **Zaino**: somma dei guadagni degli oggetti selezionati
- **Sottosequenza comune massimale**: lunghezza della stringa

Brute-force

Esplorazione

In alcuni problemi è richiesto o necessario **esplorare l'intero spazio delle soluzioni ammissibili**.

Enumerazione

Elencare **tutte** le soluzioni ammissibili

Esempio: Elencare tutte le permutazioni di un insieme

Soluzione: Algoritmi di enumerazione

[1, 2, 3]

[1, 3, 2]

[2, 1, 3]

[2, 3, 1]

[3, 2, 1]

[3, 1, 2]

Brute-force

Esplorazione

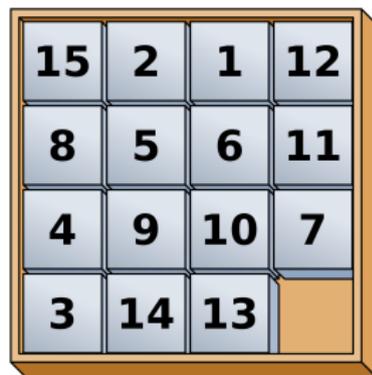
In alcuni problemi è richiesto o necessario **esplorare l'intero spazio delle soluzioni ammissibili**.

Ricerca

Trovare **una** soluzione ammissibile in uno spazio delle soluzioni molto grande

Esempio: Trovare una sequenza di mosse per il gioco del 15

Soluzione: Algoritmi di enumerazione, fermandosi alla prima soluzione trovata



https://it.wikipedia.org/wiki/Gioco_del_quindici#/media/File:15-puzzle.svg

Brute-force

Esplorazione

In alcuni problemi è richiesto o necessario **esplorare l'intero spazio delle soluzioni ammissibili**.

Conteggio

Contare tutte le soluzioni ammissibili

Esempio: contare il numero di modi in cui è possibile esprimere un valore n come somma di k numeri primi.

Soluzione: Se non è possibile contare in modo analitico, bisogna enumerare tutte le soluzioni ammissibili e contarle.

Input: $n = 23, k = 3$

Output: 2

$$\begin{aligned}23 &= 3 + 7 + 13 \\ &= 5 + 7 + 11\end{aligned}$$

Brute-force

Esplorazione

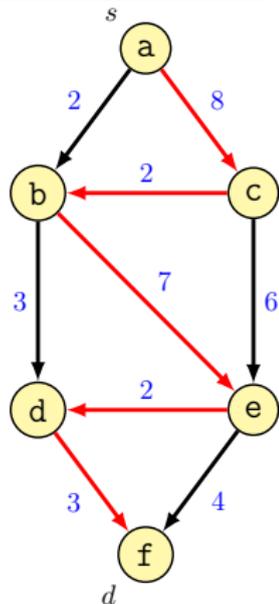
In alcuni problemi è richiesto o necessario **esplorare l'intero spazio delle soluzioni ammissibili**.

Ottimizzazione

Trovare una delle soluzioni ammissibili migliori (**ottimizzazione**) rispetto ad un certo criterio di valutazione

Esempio: trovare il cammino di peso massimo da s a d in un grafo pesato

Soluzione: Enumerare tutti i cammini da s a d , restituire quello di peso massimo



Costruire tutte le soluzioni è costoso

- Lo spazio delle possibili soluzioni può avere dimensione superpolinomiale
- A volte è l'unica strada possibile
- La potenza dei computer moderni rende "affrontabili" problemi di dimensioni medio-piccole
 - $10! = 3.63 \cdot 10^6$ (permutazione di 10 elementi)
 - $2^{20} = 1.05 \cdot 10^6$ (sottoinsieme di 20 elementi)
- A nostro vantaggio, a volte lo spazio delle soluzioni non deve essere analizzato interamente

Backtracking

Filosofia

"Prova a fare qualcosa; se non va bene, disfalo e prova qualcos'altro"
"Ritenta, e sarai più fortunato"

Ricorsione

Un metodo **sistematico** per esplorare uno spazio di ricerca, utilizzando la ricorsione per memorizzare le scelte fatte finora

Iterazione

Utilizza un approccio greedy, eventualmente tornando sui propri passi

- **Inviluppo convesso**
- String matching

Una tecnica algoritmica che, come altre, deve essere personalizzata per ogni applicazione individuale

Organizzazione generale

Organizzazione generale

- Una soluzione viene rappresentata come un **vettore di scelte** $S[1 \dots n]$
- Il contenuto degli elementi $S[i]$ è preso da un **insieme di scelte** C dipendente dal problema

Esempi

- C elementi di un insieme, possibili soluzioni **sottoinsiemi** di C
- C elementi di un insieme, possibili soluzioni **permutazioni** di C
- C mosse di gioco, possibili soluzioni **sequenze di mosse**
- C archi di un grafo, possibili soluzioni **percorsi sul grafo**

Esercizio

Sottoinsiemi

Prendete carta e penna, e elencate tutti i sottoinsiemi dell'insieme $\{A, B, C, D\}$

Permutazioni

Prendete carta e penna, e elencate tutte le permutazioni dell'insieme $\{A, B, C, D\}$

Enumerazione – Schema parziale

```

enumeration( $\langle \text{dati problema} \rangle$ , ITEM[]  $S$ , int  $i$ ,  $\langle \text{dati parziali} \rangle$ )
% Verifica se  $S[1 \dots i - 1]$  contiene una soluzione ammissibile
if accept( $\langle \text{dati problema} \rangle$ ,  $S$ ,  $i$ ,  $\langle \text{dati parziali} \rangle$ ) then
    % "Processa" la soluzione (stampa, conta, etc.)
    processSolution( $\langle \text{dati problema} \rangle$ ,  $S$ ,  $i$ ,  $\langle \text{dati parziali} \rangle$ )
else
    % Calcola l'insieme delle scelte in funzione di  $S[1 \dots i - 1]$ 
    SET  $C = \text{choices}(\langle \text{dati problema} \rangle$ ,  $S$ ,  $i$ ,  $\langle \text{dati parziali} \rangle$ )
    % Itera sull'insieme delle scelte
    foreach  $c \in C$  do
         $S[i] = c$ 
        % Chiamata ricorsiva
        enumeration( $\langle \text{dati problema} \rangle$ ,  $S$ ,  $i + 1$ ,  $\langle \text{dati parziali} \rangle$ )

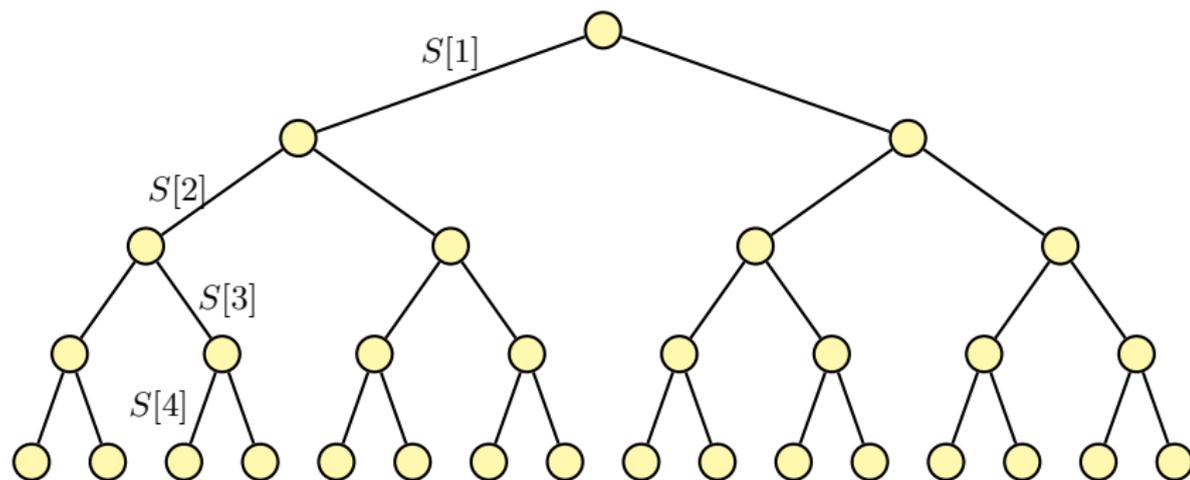
```

Soluzioni parziali

- Il parametro i rappresenta l'indice della prossima decisione da prendere
- La soluzione parziale $S[1 \dots i - 1]$ contiene le decisioni prese finora
- **Caso base:**
 - Se $S[1 \dots i - 1]$ è una soluzione ammissibile, questa viene processata
 - Assumendo che le soluzioni ammissibili non possano essere estese, la ricorsione termina
- **Passo ricorsivo**
 - Altrimenti, calcoliamo l'insieme C delle scelte possibili
 - Per ogni elemento $c \in C$:
 - Scriviamo c sulla scelta $S[i]$
 - Chiamiamo ricorsivamente la funzione con indice $i + 1$

Albero delle decisioni

- **Albero di decisione** \equiv Spazio di ricerca
- **Radice** \equiv Soluzione parziale vuota
- **Nodi interni dell'albero di decisione** \equiv Soluzioni parziali
- **Foglie in un albero di decisione** \equiv Soluzioni ammissibili



Sottoinsiemi

Elencare tutti i sottoinsiemi dell'insieme $\{1, \dots, n\}$

```
subsetsRec(int n, int[] S, int i)
```

```
% S ammissibile dopo n scelte
```

```
if i > n then
```

```
    | processSolution(S, n)
```

```
else
```

```
    | % Non presente / presente
```

```
    | SET C = {0, 1}
```

```
    | foreach c ∈ C do
```

```
        | S[i] = c
```

```
        | subsetsRec(n, S, i + 1)
```

```
subsets(int n)
```

```
% Vettore delle scelte
```

```
int[] S = new int[1 ... n]
```

```
subsetsRec(n, S, 1)
```

```
processSolution(int[] S, int n)
```

```
print "{ "
```

```
for i = 1 to n do
```

```
    | if S[i] then
```

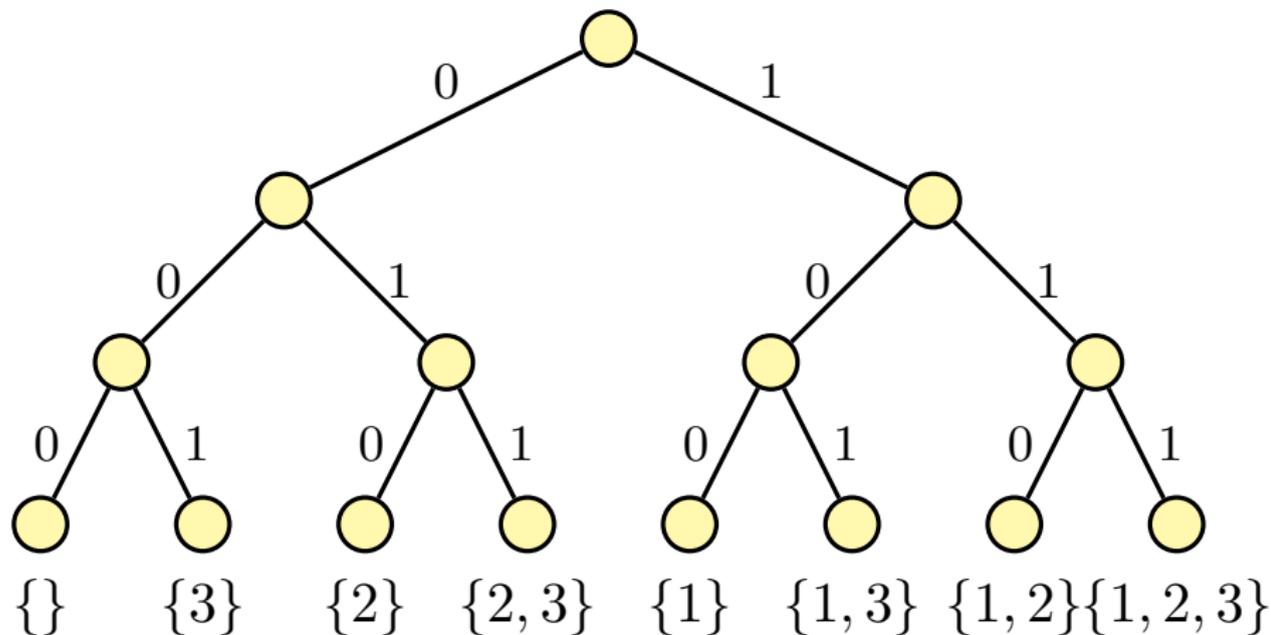
```
        | print i, " "
```

```
println "}"
```

Sottoinsiemi – Note

- Come richiesto dal problema, tutto lo spazio possibile viene esplorato
- Complessità $\Theta(n \cdot 2^n)$
- In che ordine vengono stampati gli insiemi?

Sottoinsiemi – Albero delle decisioni



Sottoinsiemi – Versione iterativa

È possibile risolvere questo problema in modo iterativo?

```
subsets(int n)
```

```
for j = 0 to  $2^n - 1$  do
```

```
  print "{ "
```

```
  for i = 0 to n - 1 do
```

```
    if (j &&  $2^i$ )  $\neq$  0 then
```

```
      print i, " "
```

```
  println "}"
```

% Bitwise and

Complessità: $\Theta(n \cdot 2^n)$

Permutazioni

Stampa tutte le permutazioni di un insieme A

```
permRec(SET  $A$ , ITEM[]  $S$ , int  $i$ )
```

```
% Se  $A$  è vuoto,  $S$  è ammissibile
```

```
if  $A$ .isEmpty() then
```

```
    | print  $S$ 
```

```
else
```

```
    % Copia  $A$  per il ciclo foreach
```

```
    SET  $C$  = copy( $A$ )
```

```
    foreach  $c \in C$  do
```

```
        |  $S[i] = c$ 
```

```
        |  $A$ .remove( $c$ )
```

```
        | permRec( $A$ ,  $S$ ,  $i + 1$ )
```

```
        |  $A$ .insert( $c$ )
```

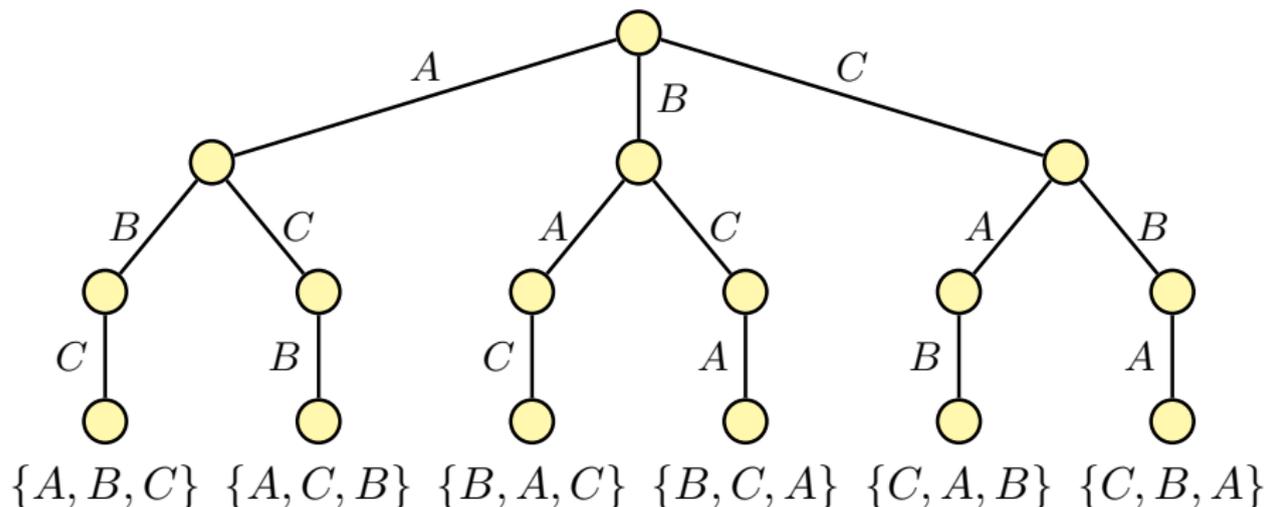
```
permutations(SET  $A$ )
```

```
int  $n$  = size( $A$ )
```

```
int[]  $S$  = new int[1... $n$ ]
```

```
permRec( $A$ ,  $S$ , 1)
```

Permutazioni – Albero delle decisioni



Permutazioni – Complessità

Complessità

- Costo della stampa: $\Theta(n)$
- Costo delle copie del vettore lungo un cammino radice-foglia:
$$\sum_{i=1}^n O(i) = O(n^2)$$
- Numero di foglie: $n!$
- Complessità $O(n^2n!)$

Permutazioni – Complessità

Altre implementazioni

- L'algoritmo più efficiente per generare le permutazioni è l'Algoritmo di Heap
 - B.R. Heap. ^a "Permutations by Interchanges". The Computer Journal, 6(3):293-294. (1963) [[Link](#)]
- Per i più coraggiosi, Knuth affronta il problema delle permutazioni nel suo Art of Computer Programming ^b
 - Knuth, Donald. "Section 7.2.1.2: Generating All Permutations", The Art of Computer Programming, volume 4A. (2011) [[Link](#)]

^aSì, esiste un tizio che si chiama Heap di cognome...

^bSolo 66 pagine!

Permutazioni – Una versione più pulita

Stampa tutte le permutazioni di un vettore S

```
permRec(ITEM[] S, int i)
```

```
% Caso base, un elemento
```

```
if i == 1 then
```

```
    | println S
```

```
else
```

```
    | for j = 1 to i do
```

```
        | swap(S, i, j)
```

```
        | permRec(S, i - 1)
```

```
        | swap(S, i, j)
```

```
permutations(ITEM[] S, int n)
```

```
permRec(S, n)
```

Complessità

- $n!$ permutazioni
- $\Theta(n)$ per stamparle tutte
- $2n$ swap per ogni permutazione
- Costo totale $\Theta(n \cdot n!)$

Enumerazione – Schema completo

```
enumeration( $\langle$ dati problema $\rangle$ , ITEM[]  $S$ , int  $i$ ,  $\langle$ dati parziali $\rangle$ )
```

```
if accept( $\langle$ dati problema $\rangle$ ,  $S$ ,  $i$ ,  $\langle$ dati parziali $\rangle$ ) then
```

```
    % Processo la soluzione  $S$  in quanto ammissibile
```

```
    processSolution( $\langle$ dati problema $\rangle$ ,  $S$ ,  $i$ ,  $\langle$ dati parziali $\rangle$ )
```

```
else if reject( $\langle$ dati problema $\rangle$ ,  $S$ ,  $i$ ,  $\langle$ dati parziali $\rangle$ ) then
```

```
    return
```

```
else
```

```
    % Ricorsione
```

```
    SET  $C$  = choices( $\langle$ dati problema $\rangle$ ,  $S$ ,  $i$ ,  $\langle$ dati parziali $\rangle$ )
```

```
    foreach  $c \in C$  do
```

```
         $S[i] = c$ 
```

```
        enumeration( $\langle$ dati problema $\rangle$ ,  $S$ ,  $i + 1$ ,  $\langle$ dati parziali $\rangle$ )
```

k-Sottoinsiemi – Tentativo 1

Elencare tutti i sottoinsiemi di k elementi di un insieme $\{1, \dots, n\}$

```
kssRec(int n, int k, int[] S, int i)
```

```
int size = count(S, n)
```

```
if i > n and size == k then
```

```
    | processSolution(S, n)
```

```
else if i > n and size ≠ k then
```

```
    | return
```

```
else
```

```
    foreach c ∈ {0, 1} do
```

```
        | S[i] = c
```

```
        | kssRec(n, k, S, i + 1)
```

```
kSubsets(int n, int k)
```

```
int[] S = new int[1...n]
```

```
kssRec(n, k, S, 1)
```

```
int count(int[] S, int n)
```

```
int count = 0
```

```
for j = 1 to n do
```

```
    | count = count + S[j]
```

```
return count
```

k -Sottoinsiemi – Tentativo 2

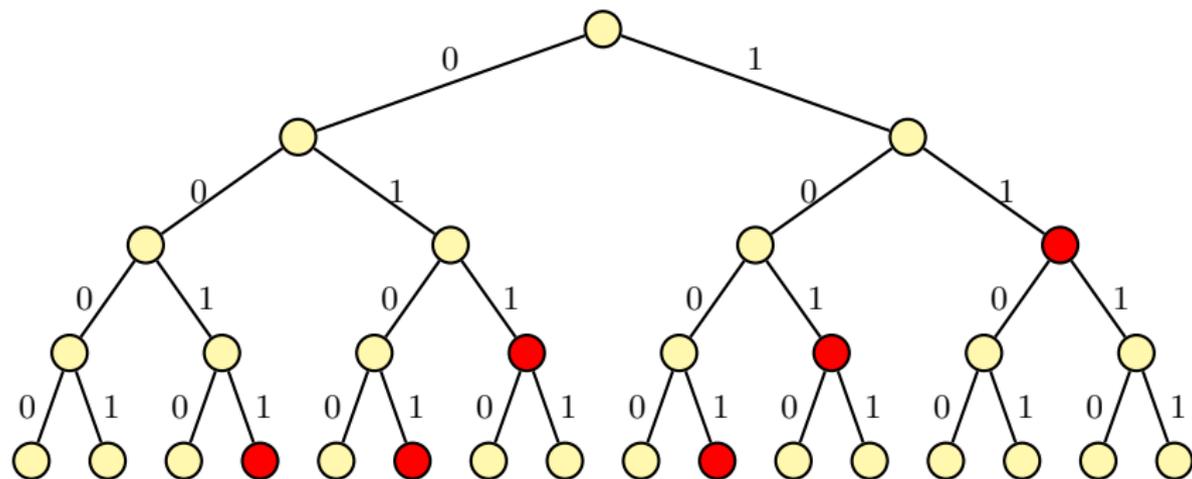
```
kssRec(int n, int missing, int[] S, int i)
```

```
if i > n and missing == 0 then
    | processSolution(S, n)
else if i > n and missing ≠ 0 then
    | return
else
    | foreach c ∈ {0, 1} do
    |     | S[i] = c
    |     | kssRec(n, missing - c, S, i + 1)
```

Note

- Evitiamo di ricontare tutte le volte i bit 1

k -Sottoinsiemi – Albero delle decisioni, per $n = 4, k = 2$



Note

- Questa soluzione suggerisce un ulteriore miglioramento...
- I nodi rossi sono nodi in cui *missing* diventa uguale a zero per la prima volta nel percorso radice–foglia

k -Sottoinsiemi – Tentativo 3 (Pruning parziale)

```
kssRec(int n, int missing, int[] S, int i)
```

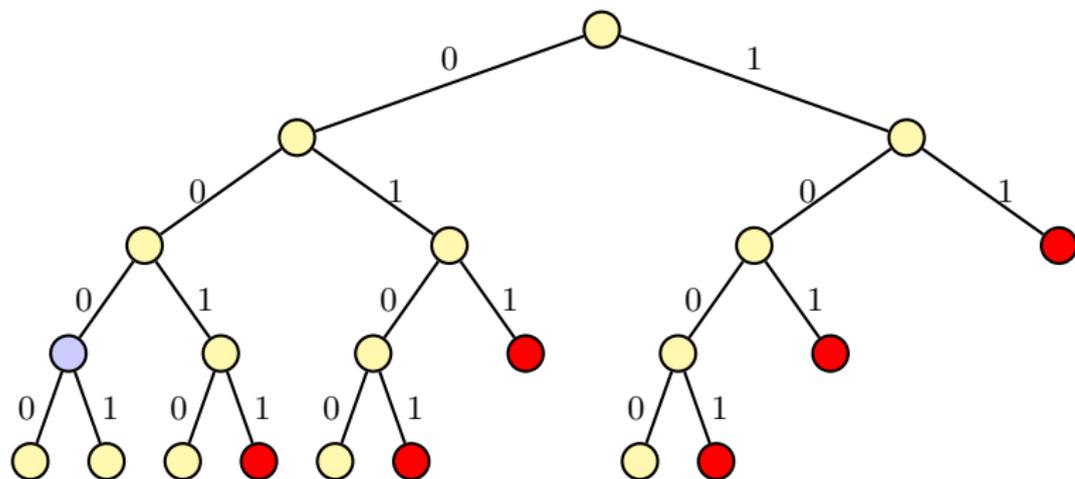
```
if i > n and missing == 0 then  
    processSolution(S, i - 1)  
else if i > n or missing < 0 then  
    return  
else  
    foreach c ∈ {0, 1} do  
        S[i] = c  
        kssRec(n, missing - c, S, i + 1)
```

Note

- Se $missing = 0$, la soluzione è ammissibile
- Non ha senso continuare oltre

Pruning – Esempio: $n = 4, k = 2$

- “Rami” dell’albero che sicuramente non portano a soluzioni ammissibili possono essere “potati” (pruned)
- La valutazione viene fatta nelle soluzioni parziali radici del sottoalbero da potare



k -Sottoinsiemi – Tentativo 4 (Pruning totale)

```
kssRec(int n, int missing, int[] S, int i)
```

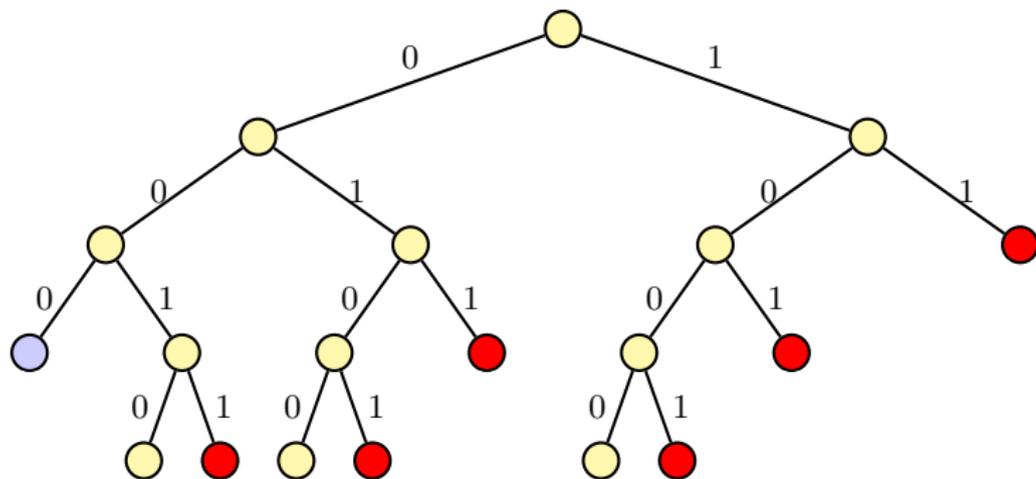
```
if missing == 0 then
    | processSolution(S, i - 1)
else if i > n or missing < 0 or  $n - (i - 1) < missing$  then
    | return
else
    | foreach  $c \in \{0, 1\}$  do
        |  $S[i] = c$ 
        | kssRec(n, missing - c, S, i + 1)
```

Note

- Evitiamo di proseguire in rami che non possono dare origine alla soluzione

k -Sottoinsiemi – Albero delle decisioni, per $n = 4, k = 2$

- I nodi interni in azzurro rappresentano i nodi che non possono dare origine a soluzioni



k -Sottoinsiemi – Tentativo 5 (Clean-up)

```
kssRec(int n, int missing, int[] S, int i)
```

```
if missing == 0 then
```

```
    processSolution(S, i - 1)
```

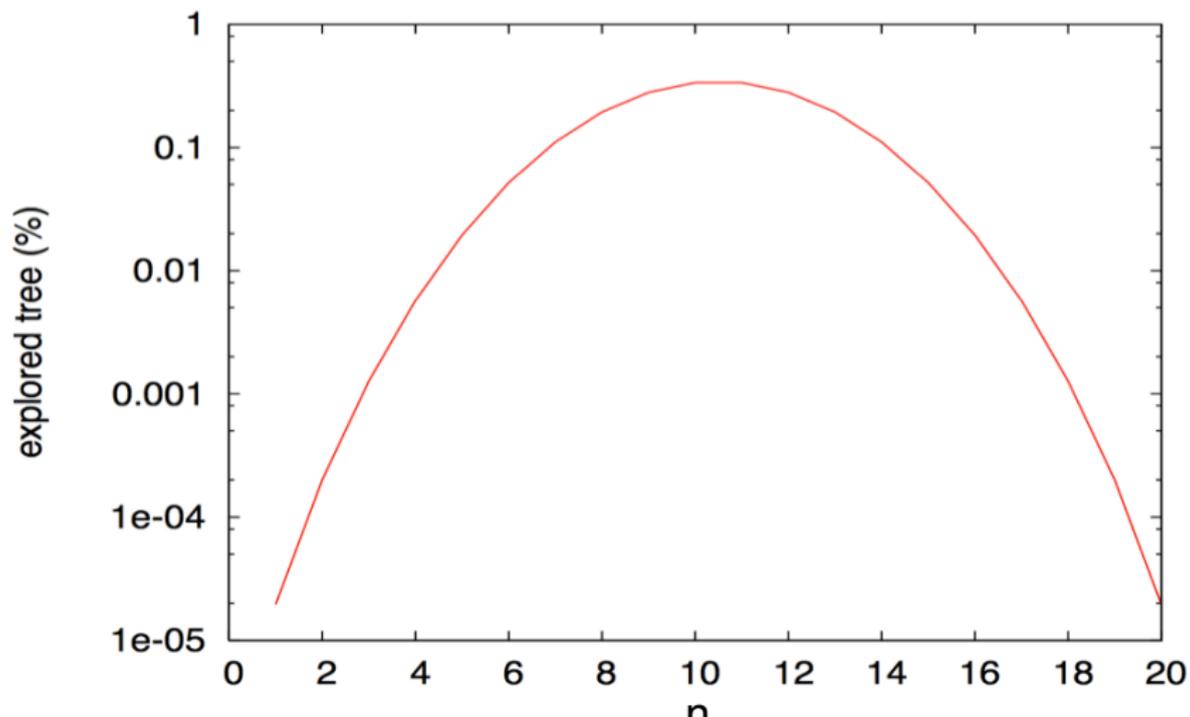
```
else if  $i \leq n$  and  $0 < missing \leq n - (i - 1)$  then
```

```
    foreach  $c \in \{0, 1\}$  do
```

```
         $S[i] = c$ 
```

```
        kssRec(n, missing - c, S, i + 1)
```

k -Sottoinsiemi – Vantaggi



k -Sottoinsiemi – Sommario

Cosa abbiamo imparato?

- “Specializzando” l’algoritmo generico, possiamo ottenere una versione più efficiente
- Versione efficiente per
 - valori di k “piccoli” (vicini a 1)
 - valori di k “grandi” (vicini a n)
- Miglioramento solo parziale verso $n/2$
- Possiamo ottenere la stessa efficienza con un algoritmo iterativo?

Subset Sum

Somma di sottoinsieme (Subset sum)

Dati un vettore A contenente n interi positivi ed un intero positivo k , **esiste** un sottoinsieme $S \subseteq \{1 \dots n\}$ tale che $\sum_{i \in S} a[i] = k$?

i	1	2	3	4	5	6	7	8
A	1	4	3	12	7	2	21	55

$$k = 23$$

$$S_1 = \{2, 4, 5\}$$

$$S_2 = \{6, 7\}$$

Subset Sum

Analisi del problema

- Lo risolviamo qui tramite backtracking in tempo $O(2^n)$
- Può essere risolto tramite programmazione dinamica in tempo $O(kn)$
 - Questa complessità è pseudo-polinomiale
 - Ci torneremo sopra
- Non siamo interessati a tutte le soluzioni, ce ne basta una
- Interrompiamo l'esecuzione alla prima soluzione trovata

Enumerazione – Interruzione alla prima soluzione

```
enumeration(<dati problema>, ITEM[] S, int i, <dati parziali>)  
if accept(<dati problema>, S, i, <dati parziali>) then  
    | processSolution(<dati problema>, S, i, <dati parziali>)  
    | return true % Trovata soluzione, restituisco true  
else if reject(<dati problema>, S, i, <dati parziali>) then  
    | return false % Impossibile trovare soluzioni, restituisco false  
else  
    | SET C = choices(<dati problema>, S, i, <dati parziali>)  
    | foreach c ∈ C do  
    | | S[i] = c  
    | | if enumeration(<dati problema>, S, i + 1, <dati parziali>) then  
    | | | return true % Trovata soluzione, restituisco true  
    | return false % Nessuna soluzione, restituisco false
```

Subset sum

```
boolean ssRec(int[] A, int n, int missing, int[] S, int i)


---


if missing == 0 then
    processSolution(S, i - 1)           % Stampa gli indici della soluzione
    return true
else if i > n or missing < 0 then
    return false                       % Terminati i valori o somma eccessiva
else
    foreach c ∈ {0, 1} do
        S[i] = c
        if ssRec(A, n, missing - A[i] · c, S, i + 1) then
            return true
    return false
```

Subset sum

```
subsetSum(int[] A, int n, int k)
```

```
int[] S = new int[1...n]  
ssRec(A, n, k, S, 1)
```

Note

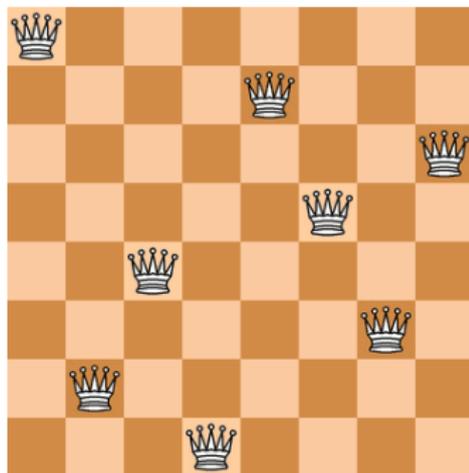
Avendo informazioni sulle somme cumulative, sarebbe possibile portare ulteriormente

Problema delle otto regine

Problema

Posizionare n regine in una scacchiera $n \times n$, in modo tale che nessuna regina ne "minacci" un'altra.

- Un po' di storia:
 - Introdotto da Max Bezzel (1848)
 - Gauss trovò 72 delle 92 soluzioni
- Partendo da un approccio "stupido", raffiniamo mano a mano la soluzione
- L'idea è mostrare l'effetto della modellazione sull'efficienza



https://en.wikipedia.org/wiki/Eight_queens_puzzle

Problema delle otto regine

Idea: Ci sono n^2 caselle dove piazzare una regina

$S[1 \dots n^2]$ array binario	$S[i] = \mathbf{true} \Rightarrow$ "regina in $S[i]$ "
controllo soluzione	se $i = n^2$
$\text{choices}(S, n, i)$	$\{\mathbf{true}, \mathbf{false}\}$
pruning	se la nuova regina minaccia una delle regine esistenti, restituisce \emptyset
# soluzioni per $n = 8$	$2^{64} \approx 1.84 \cdot 10^{19}$

Commenti

- Forse abbiamo un problema di rappresentazione?
- Matrice binaria molto sparsa

Problema delle otto regine

Idea: Dobbiamo piazzare n regine, ci sono n^2 caselle

$S[1 \dots n]$ coordinate in $\{1 \dots n^2\}$	$S[i]$ coordinata della regina i
controllo soluzione	se $i = n$
$\text{choices}(S, n, i)$	$\{1 \dots n^2\}$
pruning	restituisce il sottoinsieme di mosse legali
# soluzioni per $n = 8$	$(n^2)^n = 64^8 = 2^{48} \approx 2.81 \cdot 10^{14}$

Commenti

- C'è un miglioramento, ma lo spazio è ancora grande ...
- Problema: come si distingue una soluzione "1-7-..." da "7-1-..." ?

Problema delle otto regine

Idea: non mettere regine in caselle precedenti a quelle già scelte

$S[1 \dots n]$ coordinate in $\{1 \dots n^2\}$	$S[i]$ coordinata della regina i
controllo soluzione	se $i = n$
$\text{choices}(S, n, i)$	$\{1 \dots n^2\}$
pruning	restituisce mosse legali, $S[i] > S[i - 1]$
# soluzioni per $n = 8$	$(n^2)^n / n! = 2^{48} / 40320 \approx 6.98 \cdot 10^9$

Commenti

- Ottimo, abbiamo ridotto molto, ma si può ancora fare qualcosa

Problema delle otto regine

Idea: ogni riga della scacchiera deve contenere esattamente una regina

$S[1 \dots n]$ coordinate in $\{1 \dots n\}$	$S[i]$ colonna della regina i , dove riga = i
controllo soluzione	se $i = n$
$\text{choices}(S, n, i)$	$\{1 \dots n\}$
pruning	restituisce le colonne legali
# soluzioni per $n = 8$	$n^n = 8^8 \approx 1.67 \cdot 10^7$

Commenti

- Quasi alla fine

Problema delle otto regine

Idea: anche ogni colonna deve contenere esattamente una regina

$S[1 \dots n]$ coordinate in $\{1 \dots n\}$	permutazione di $\{1 \dots n\}$
controllo soluzione	se $i = n$
$\text{choices}(S, n, i)$	$\{1 \dots n\}$
pruning	elimina le diagonali
# soluzioni per $n = 8$	$n! = 8! = 40320$

Commenti

- Soluzioni effettivamente visitate = 15720

Problema delle n regine

```
queens(int  $n$ , int[]  $S$ , int  $i$ )
```

```
if  $i > n$  then
```

```
    | print  $S$ 
```

```
else
```

```
    for  $j = 1$  to  $n$  do % Prova a piazzare la regina nella colonna  $j$ 
```

```
        | boolean  $legal = true$ 
```

```
        for  $k = 1$  to  $i - 1$  do % Verifica le regine precedenti
```

```
            | if  $S[k] == j$  or  $S[k] == j + i - k$  or  $S[k] == j - i + k$  then
```

```
                |  $legal = false$ 
```

```
        | if  $legal$  then
```

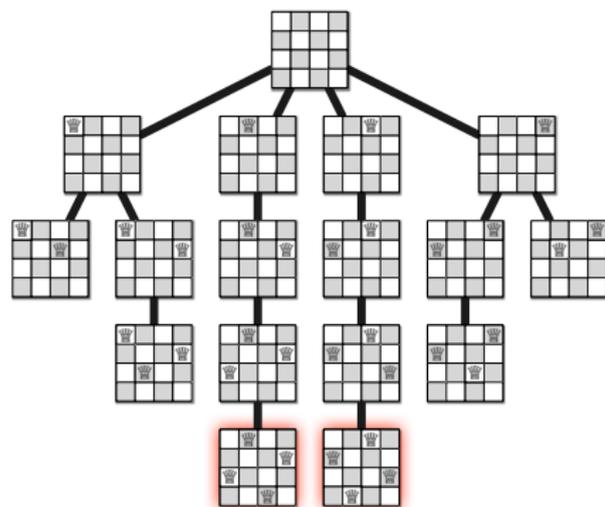
```
            |  $S[i] = j$ 
```

```
            | queens( $n$ ,  $S$ ,  $i + 1$ )
```

Problema delle n regine

n	1	2	3	4	5	6	7	8	9	10	11	12
#Sol	1	0	0	2	10	4	40	92	352	724	2680	14200

- In alto, il numero di soluzioni al variare di n
- A destra, l'albero della ricorsione per $n = 4$



(Jeff Erickson)

Problema delle n regine

Minimum-conflicts heuristic

Si parte da una soluzione iniziale “ragionevolmente buona”, e si muove il pezzo con il più grande numero di conflitti nella casella della stessa colonna che genera il numero minimo di conflitti. Si ripete fino a quando non ci sono più pezzi da muovere.

- Algoritmo in tempo lineare
- Ad esempio, con $n = 1\,000\,000$, richiede 50 passi in media
- Questo algoritmo non garantisce che il risultato sia corretto
 - Greedy (Capitolo 14)
 - Ricerca locale (Capitolo 15)
 - Algoritmi probabilistici (Capitolo 17)
 - Soluzioni per problemi intrattabili (Capitolo 19)

Sudoku - "Suuji wa dokushin ni kagiru"

2	5			9			7	6
			2		4			
		1	5		3	9		

	8	9	4		5	2	6	
1				2				4
	2	5	6			7	3	

		8	3		2	1		
			9		7			
3	7			8			9	2

2	5	3	8	9	1	4	7	6
8	9	7	2	6	4	3	1	5
6	4	1	5	7	3	9	2	8

7	8	9	4	3	5	2	6	1
1	3	6	7	2	9	8	5	4
4	2	5	6	1	8	7	3	9

9	6	8	3	5	2	1	4	7
5	1	2	9	4	7	6	8	3
3	7	4	1	8	6	5	9	2

Sudoku

```
boolean sudoku(int[][] S, int i)
```

```
if i == 81 then
```

```
    processSolution(S, n)
    return true
```

```
int x = i mod 9
```

```
int y =  $\lfloor i/9 \rfloor$ 
```

```
SET C = moves(S, x, y)
```

```
int old = S[x][y]
```

```
foreach c ∈ C do
```

```
    S[x][y] = c
    if sudoku(S, i + 1) then
        return true
```

```
S[x][y] = old
```

```
return false
```

```
SET moves(int[][] S, int x, int y)
```

```
SET C = Set()
```

```
if S[x][y] ≠ 0 then
```

```
    % Numero pre-inserito
    C.insert(S[x][y])
```

```
else
```

```
    % Verifica conflitti
```

```
    for c = 1 to 9 do
```

```
        if check(S, x, y, c) then
            C.insert(c)
```

```
return C
```

Sudoku

```
boolean check(int[][] S, int x, int y, int c)
```

```
for j = 0 to 8 do
```

```
    if S[x][j] == c then
```

```
        return false
```

```
% Controllo sulla colonna
```

```
    if S[j][y] == c then
```

```
        return false
```

```
% Controllo sulla riga
```

```
int bx = ⌊x/3
```

```
int by = ⌊y/3
```

```
for ix = 0 to 2 do
```

```
    for iy = 0 to 2 do
```

```
% Controllo sulla sottotabella
```

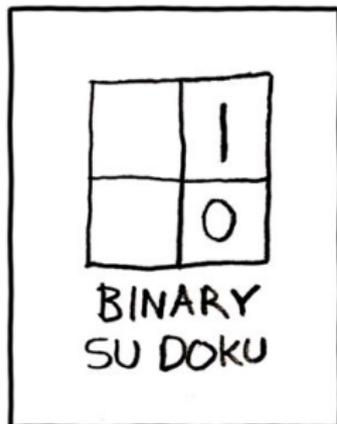
```
        if S[bx · 3 + ix][by · 3 + iy] = c then
```

```
            return false
```

```
return true
```

Sudoku

- La soluzione proposta è molto veloce per $n = 9$
 - Nel mio portatile, codice in Java: il Sudoku iniziale richiede 0.1 secondi
- È possibile generalizzare per $n = k^2$
 - $k = 4$, Mega-Sudoku (Esadecimale)
- Esistono tecniche euristiche per fissare ulteriori numeri
 - Possono risolvere completamente il problema
 - Possono essere usate come pre-processamento
 - Tom Davis, "The Mathematics of Sudoku" [\[Link\]](#)

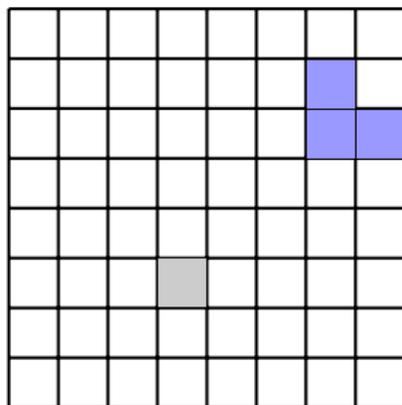


<https://xkcd.com/74/>

Un ulteriore puzzle

Problema

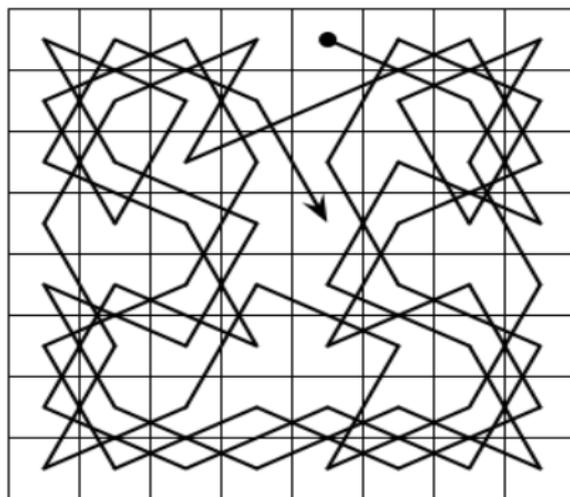
- Si consideri una scacchiera $n \times n$, con $n = 2^k$
- Qualsiasi scacchiera di questo tipo con una cella rimossa può essere ricoperta da triomini a forma di L
- Trovare un algoritmo che trovi una possibile ricopertura della scacchiera



Knight tour

Problema

Si consideri ancora una scacchiera $n \times n$; lo scopo è trovare un “giro di cavallo”, ovvero un percorso di mosse valide del cavallo in modo che ogni casella venga visitata al più una volta



Knight tour

Soluzione

Matrice $n \times n$ le cui celle contengono:

- 0 se la cella non è mai stata visitata
- i se la cella è stata visitata al passo i -esimo

Knight tour

```
boolean knightTour(int[][] S, int i, int x, int y)
```

```
% Se  $i = 64$ , ho fatto 63 mosse e ho completato un tour (aperto)
```

```
if  $i == 64$  then
```

```
    processSolution(S)
```

```
    return true
```

```
SET  $C = \text{moves}(S, x, y)$ 
```

```
foreach  $c \in C$  do
```

```
     $S[x][y] = i$ 
```

```
    if knightTour(S,  $i + 1$ ,  $x + m_x[c]$ ,  $y + m_y[c]$ ) then
```

```
        return true;
```

```
     $S[x][y] = 0$ 
```

```
return false
```

Knight tour

```
SET moves(int[][] S, int x, int y)
```

```
SET C = Set()
```

```
for i = 1 to 8 do
```

```
    nx = x + mx[i]
```

```
    ny = y + my[i]
```

```
    if 1 ≤ nx ≤ 8 and 1 ≤ ny ≤ 8 and S[nx][ny] == 0 then
```

```
        C.insert(i)
```

```
return C
```

$$m_x = \{-1, +1, +2, +2, +1, -1, -2, -2\}$$

$$m_y = \{-2, -2, -1, +1, +2, +2, +1, -1\}$$

Knight tour – Considerazioni su efficienza

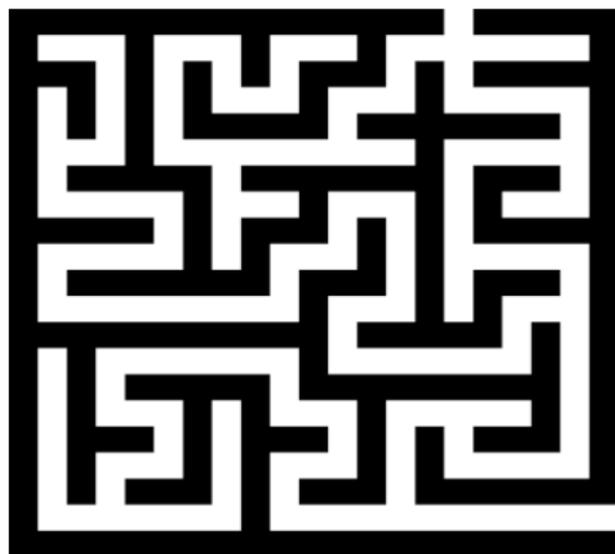
- Ad ogni passo ho al massimo 8 caselle possibili, quindi ne visito al più $8^{63} \approx 7.84 \cdot 10^{55}$
- Grazie al pruning ne visito molto meno, ma resta comunque un problema non affrontabile per valori grandi di n
- È un esempio del più generale "problema del cammino hamiltoniano", per il quale non esistono soluzioni polinomiali
- Per questo particolare problema, tuttavia, esistono degli algoritmi di costo lineare nel numero di caselle

https://en.wikipedia.org/wiki/Knight%27s_tour

Generazione labirinti

Problemi

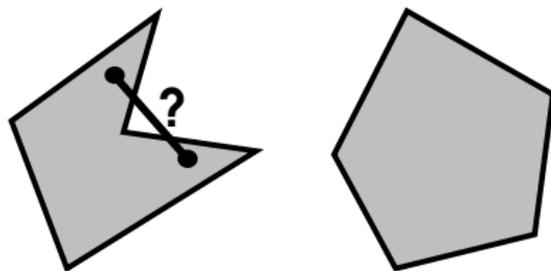
- Come generare un labirinto in una griglia $n \times n$?
- Come uscire da un labirinto?



Inviluppo convesso (Convex Hull)

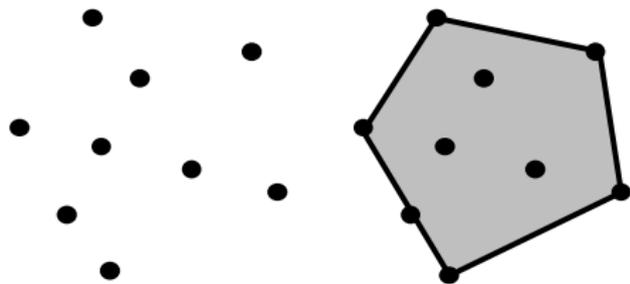
Poligono convesso

Un poligono nel piano è **convesso** se ogni segmento di retta che congiunge due punti del poligono sta interamente nel poligono stesso (bordo incluso).



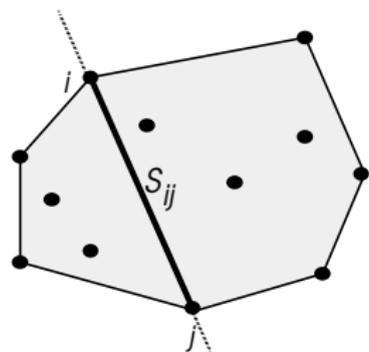
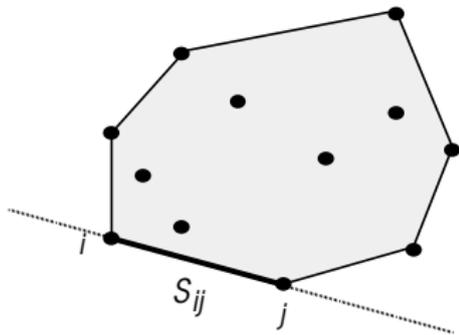
Inviluppo convesso

Dati n punti p_1, \dots, p_n nel piano, con $n \geq 3$, l'**inviluppo convesso** (**convex hull**) è, fra tutti i poligoni convessi che li contengono tutti, quello di superficie minima.



Algoritmo inefficiente – $O(n^3)$

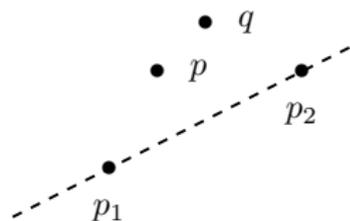
- Un poligono può essere rappresentato per mezzo dei suoi spigoli
- Si consideri la retta che passa per una coppia di punti p_i, p_j , che divide il piano in due semipiani chiusi ($O(n^2)$ coppie)
- Se tutti i rimanenti $n - 2$ punti stanno "dalla stessa parte", allora lo spigolo S_{ij} fa parte dell'inviluppo convesso



"Stessa parte"

Problema

Data una retta definita dai punti p_1 e p_2 , determinare se due punti p e q stanno nello stesso semipiano definito dalla retta.



```
boolean sameSide(POINT p1, POINT p2, POINT p, POINT q)
```

```
float dx = p2.x - p1.x
```

```
float dy = p2.y - p1.y
```

```
float dx1 = p.x - p1.x
```

```
float dy1 = p.y - p1.y
```

```
float dx2 = q.x - p2.x
```

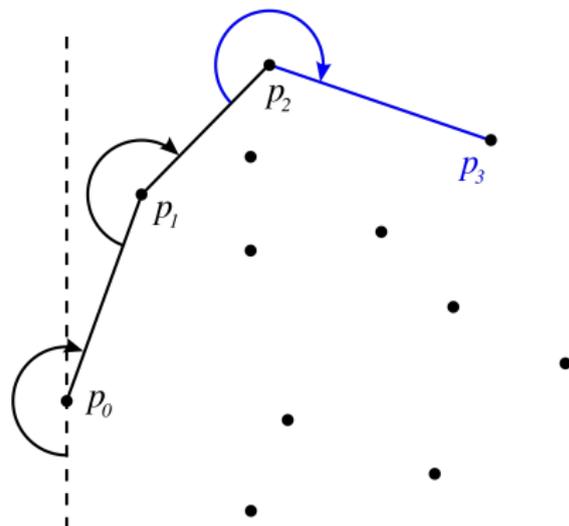
```
float dy2 = q.y - p2.y
```

```
return ((dx * dy1 - dy * dx1) * (dx * dy2 - dy * dx2) >= 0)
```

Algoritmo di Jarvis (Gift Packing) – $O(nh)$

Punti 0,1

- Si assegna a p_0 il punto più a sinistra, che appartiene all'inviluppo convesso ($O(n)$)
- Si calcola l'angolo della retta passante per p_0 e ogni altro punto p_j rispetto alla retta verticale ($O(n)$)
- Si seleziona come punto p_1 il punto con angolo minore (costo $O(n)$)

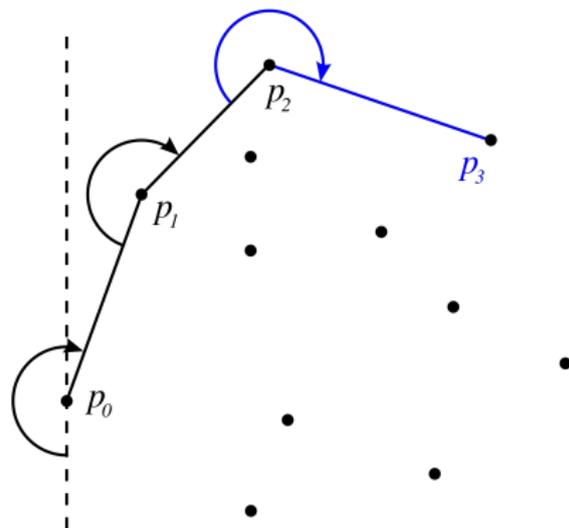


https://en.wikipedia.org/wiki/Gift_wrapping_algorithm#/media/File:Jarvis_march_convex_hull_algorithm_diagram.svg

Algoritmo di Jarvis (Gift Packing) – $O(nh)$

Punto i

- Si considera la retta r passante per i punti p_{i-1}, p_{i-2}
- Si calcola l'angolo passante per p_{i-1} e ogni altro punto non considerato e la retta r
- Si seleziona come punto p_i il punto con angolo minore
- Costo per ogni spigolo: $O(n)$

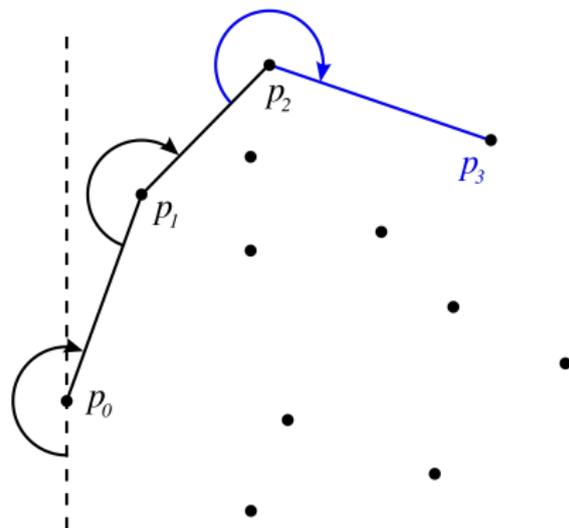


https://en.wikipedia.org/wiki/Gift_wrapping_algorithm#/media/File:Jarvis_march_convex_hull_algorithm_diagram.svg

Algoritmo di Jarvis (Gift Packing) – $O(nh)$

Terminazione

- Si termina quando si torna al punto p_0
- L'algoritmo ha costo $O(nh)$, dove h è il numero di spigoli

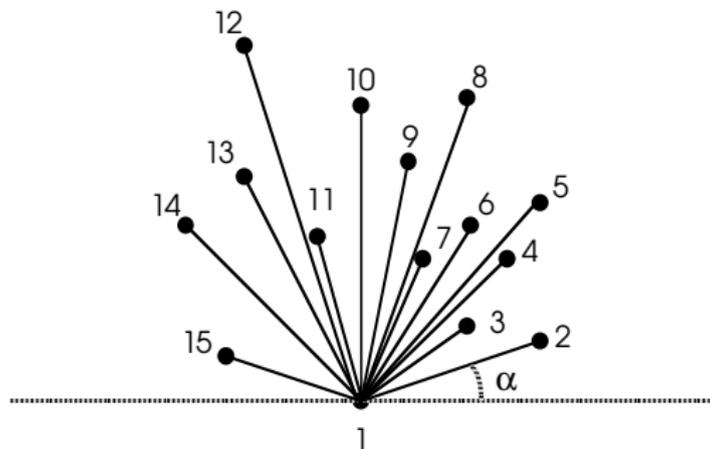


https://en.wikipedia.org/wiki/Gift_wrapping_algorithm#/media/File:Jarvis_march_convex_hull_algorithm_diagram.svg

Algoritmo di Graham

Fase 1

- Il punto con ordinata minima fa parte dell'inviluppo convesso
- Si ordinano i punti in base all'angolo formato dalla retta passante per il punto con ordinata minima e la retta orizzontale



Algoritmo di Graham

```
STACK graham(POINT[] p, int n)
```

```
int min = 1
```

```
for i = 2 to n do
```

```
    if  $p[i].y < p[min].y$  then  
         $min = i$ 
```

```
 $p[1] \leftrightarrow p[min]$ 
```

```
{ riordina  $p[2, \dots, n]$  in base all'angolo formato rispetto all'asse  
  orizzontale quando sono connessi con  $p[1]$  }
```

```
{ elimina eventuali punti "allineati" tranne i più lontani da  $p_1$ ,  
  aggiornando  $n$  }
```

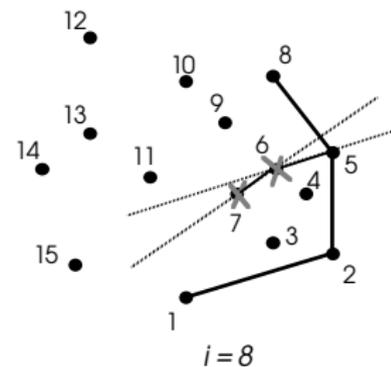
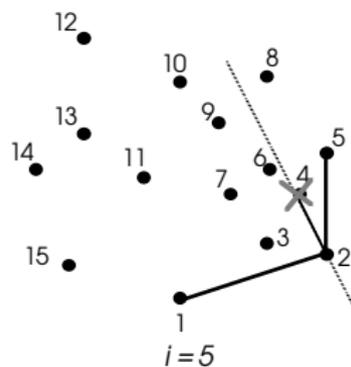
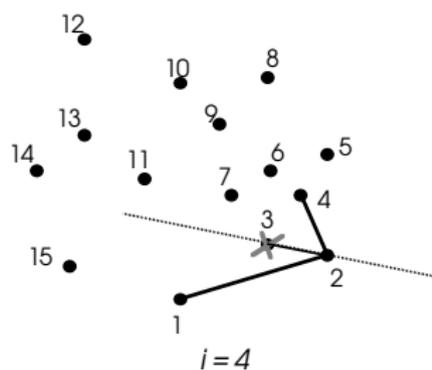
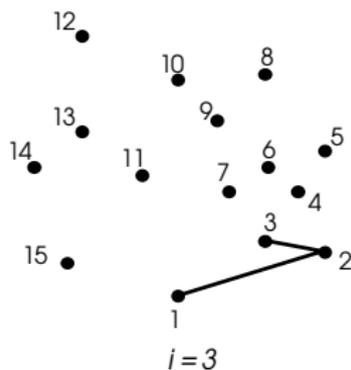
```
[...]
```

Algoritmo di Graham

Fase 2

- Inserisci p_1, p_2 nell'inviluppo corrente
- Per tutti i punti $p_i = 3, \dots, n$:
 - Siano p_{j-1} e p_j il penultimo e ultimo vertice dell'inviluppo corrente
 - Se $\text{sameSide}(p_{j-1}, p_j, p_1, p_i) = \text{false}$, ovvero p_1 e p_i non si trovano dalla stessa parte rispetto alla retta passante per p_{j-1} e p_j , allora elimina p_j dall'inviluppo corrente
 - Termina tale "scansione" se p_j non deve essere eliminato
 - aggiungi p_i all'inviluppo "corrente"

Algoritmo di Graham



Algoritmo di Graham

```
STACK graham(POINT[]  $p$ , int  $n$ ) (continua)
STACK  $S = \text{Stack}()$ 
 $S.\text{push}(p_1); S.\text{push}(p_2)$ 
for  $i = 3$  to  $n$  do
    while not sameSide( $S.\text{top}()$ ,  $S.\text{top}2()$ ,  $p_1, p_i$ ) do
         $S.\text{pop}()$ 
     $S.\text{push}(p_i)$ 
return  $S$ 
```

Algoritmo di Graham – Complessità

Complessità: $O(n \log n)$

- Il calcolo degli angoli richiede $O(n)$
- L'ordinamento richiede $O(n \log n)$
- La rimozione di punti con angolo uguale richiede $O(n)$
- Ogni punto viene aggiunto/rimosso al massimo una volta (costo $O(n)$)

Conclusioni

Algoritmo	Note	Compl.
Gift packing	h numero di punti nell'involuppo convesso (1973)	$O(nh)$
Graham	Backtracking iterativo (1972)	$O(n \log n)$
Monotone Chain	Come Graham, ma senza angoli (1979)	$O(n \log n)$
Preparata, Hong	Divide et impera (1977)	$O(n \log n)$
Chan	Graham + Jarvis (1996)	$O(n \log h)$

Conclusioni

“Convex hull is the favorite paradigm of **computational geometers**. Although the description of the problem is fairly simple, its solution takes into account all aspects of computational geometry.”

O. Devillers (1996)