Preparazione alle Olimpiadi di Informatica

UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

Algoritmi di ricerca completa e greedy

Mario Alviano

Parte 1

Algoritmi di ricerca completa

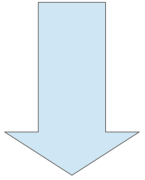
Ricerca completa

- Algoritmi che esplorano tutto lo spazio di ricerca
- Piccole modifiche per
  - computare una soluzione
  - computare tutte le soluzioni
  - contare le soluzioni
  - trovare la soluzione **migliore**

Generare sottoinsiemi (1)

- Dato un insieme, generare tutti i possibili sottoinsiemi
- Possiamo sempre ricondurre il problema alla generazione di insiemi di indici

$$\{0,1,2\}$$



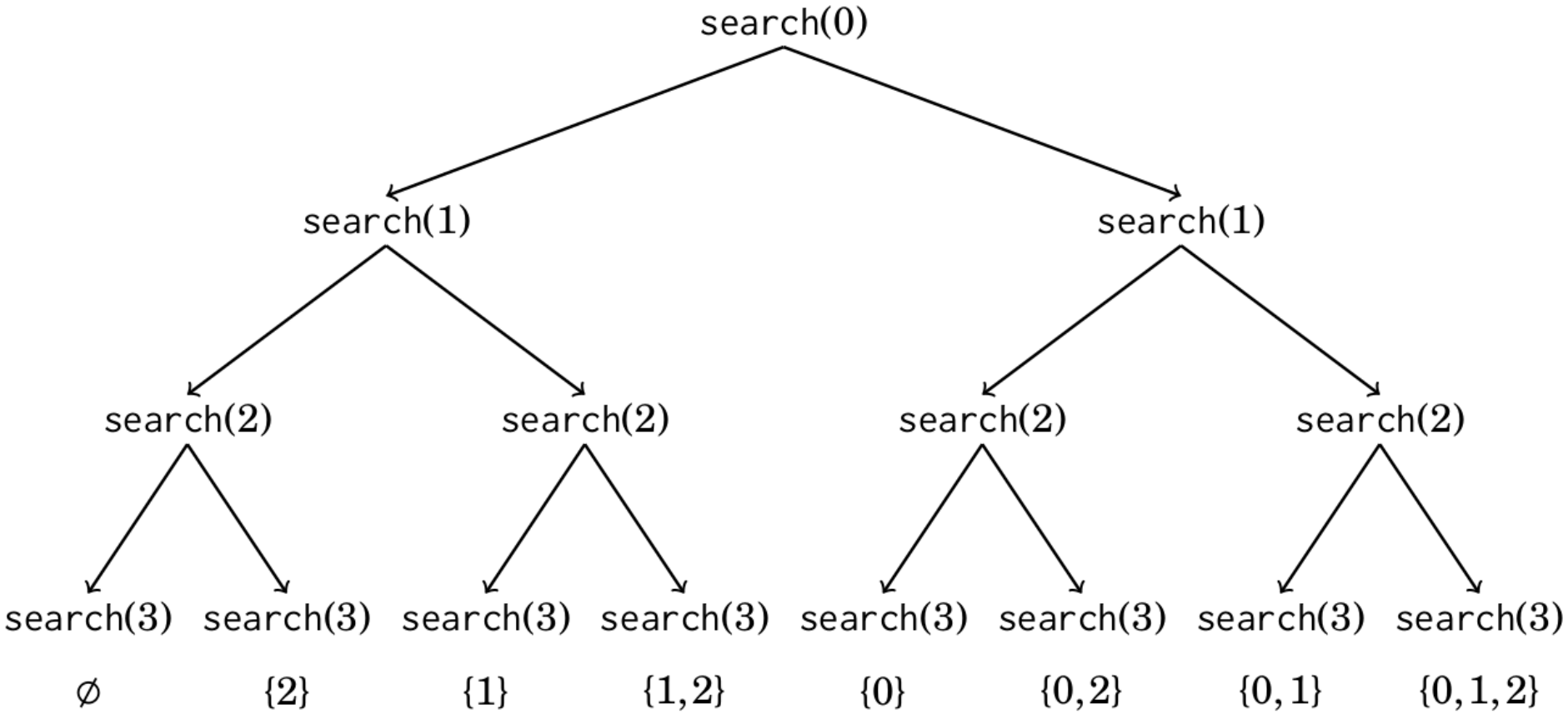
$$\emptyset,\ \{0\},\ \{1\},\ \{2\},\ \{0,1\},$$
$$\{0,2\},\ \{1,2\}\ \text{and}\ \{0,1,2\}$$

Generare sottoinsiemi (2)

- Si parte con search(0)
- Si usa ricorsione
- A ogni passo
  - ignora k, oppure
  - aggiungi k
- Dopo n chiamate si genera l'insieme vuoto
- Via via gli altri sottoinsiemi

```
void search(int k) {
    if (k == n) {
        // process subset
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

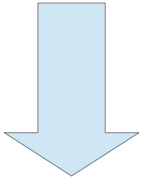
Generare sottoinsiemi (3)



Generazione di permutazioni (1)

- Una permutazione è un cambio di ordine

$$\{0,1,2\}$$



$$(0,1,2),\ (0,2,1),\ (1,0,2),\ (1,2,0),$$
$$(2,0,1) \text{ and } (2,1,0)$$

Generazione di permutazioni (2)

- A ogni passo si aggiunge un nuovo elemento
- La funzione è ricorsiva
- Dopo n chiamate si genera la permutazione identità
- Via via le altre permutazioni

```
void search() {
    if (permutation.size() == n) {
        // process permutation
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

Generazione di permutazioni (3)

- La funzione next_permutation dell'STL genera una permutazione di un vettore secondo uno schema ciclico

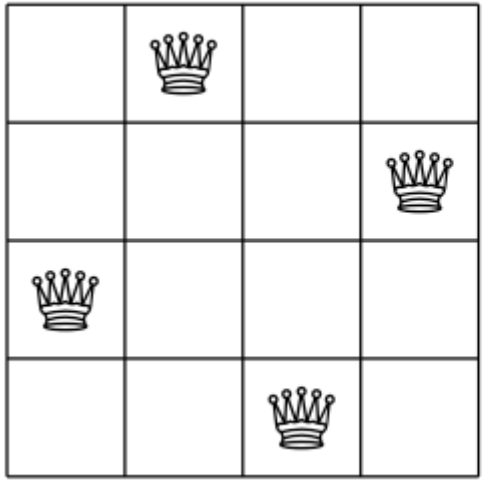
```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(),permutation.end()));
```

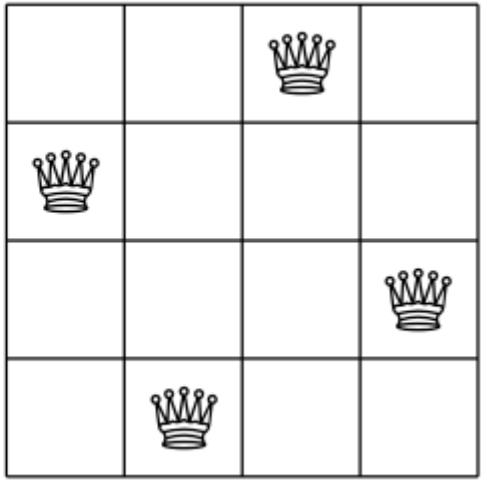
Backtracking

- A ogni passo si estende una soluzione parziale in tutti i modi possibili
- Si inizia con la soluzione (parziale) vuota
- Se una soluzione (parziale) è non valida, si scarta
- Vediamo due esempi
  - n-queens
  - numero di path in una griglia

Soluzioni di n-queen (1)

- Data una scacchiera n x n, in quanti modi si possono posizionare n regine?
- Ricorda che le regine mangiano in verticale, orizzontale e diagonale





Soluzioni di n-queen (2)

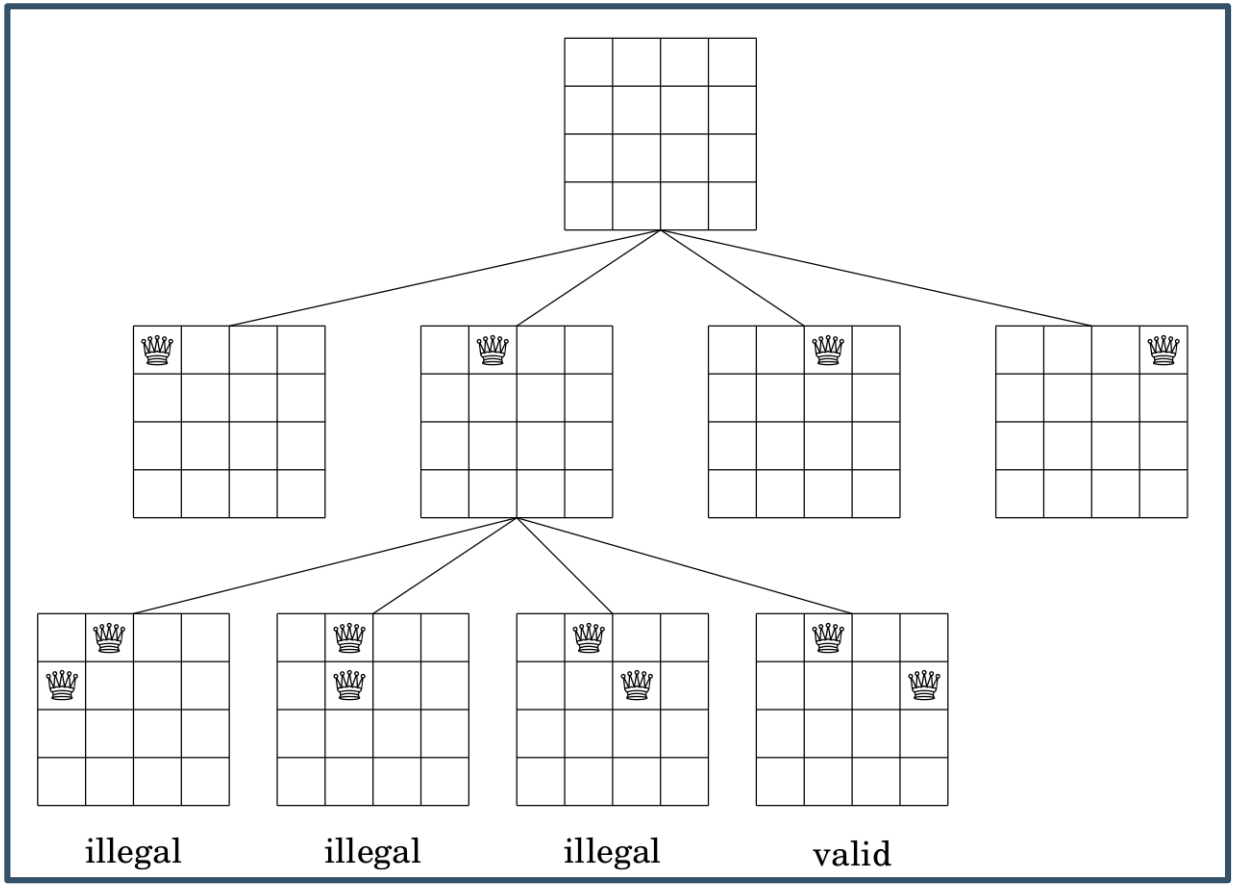
- Una regina per riga
- Una regina per colonna
- Al più una regina in ogni diagonale

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

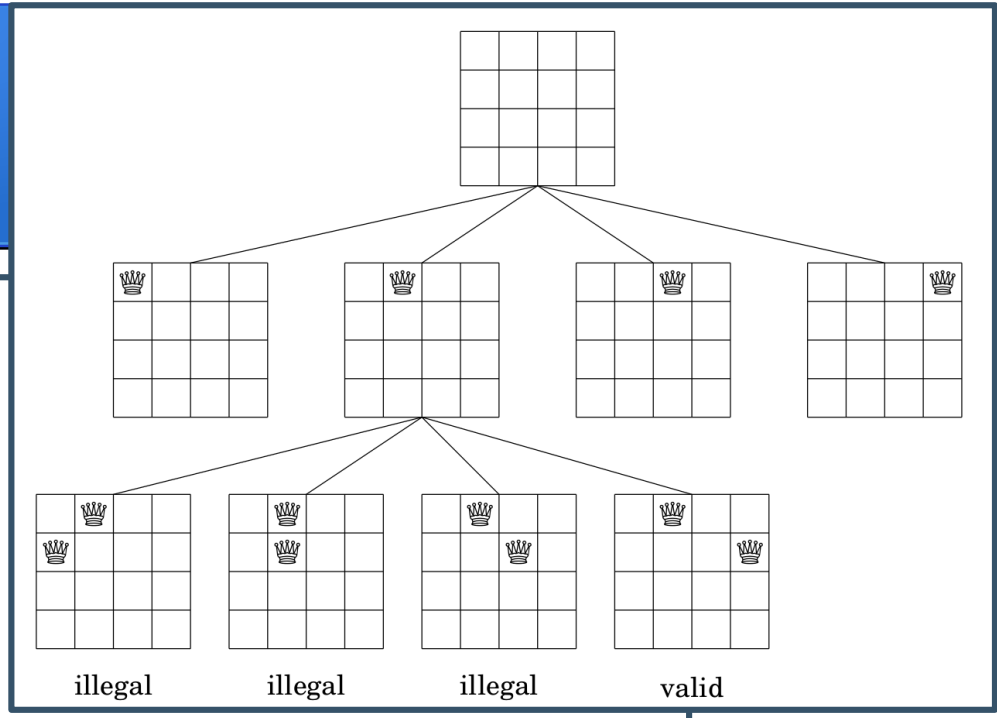
3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2



Soluzioni di n-queen (3)

```
void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```



Soluzioni di n-queen (4)

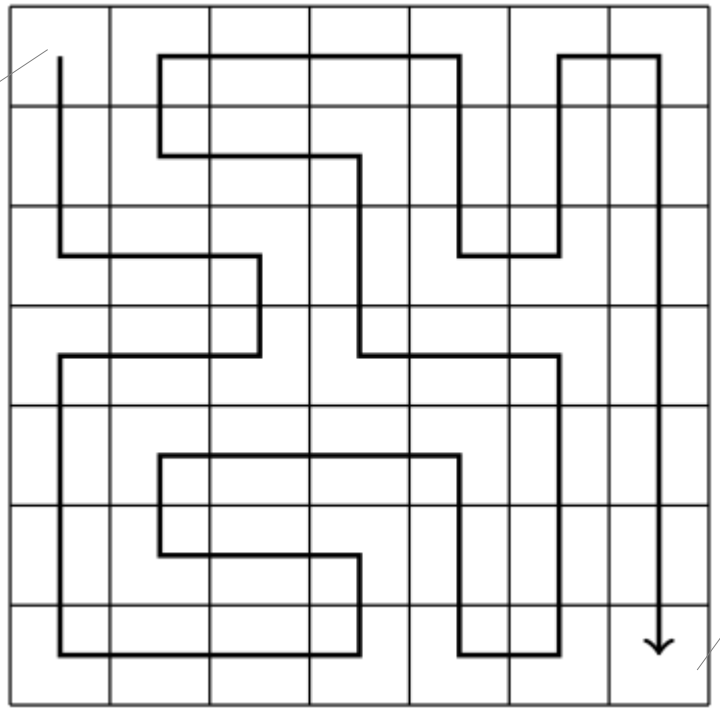
- Il numero di soluzioni cresce rapidamente
- Lo spazio di ricerca cresce rapidamente
- Attenzione al timeout!

```
n = 1
        solutions = 1
        time (s) = 0.000001
n = 2
        solutions = 0
        time (s) = 0.000000
n = 3
        solutions = 0
        time (s) = 0.000001
n = 4
        solutions = 2
        time (s) = 0.000001
n = 5
        solutions = 10
        time (s) = 0.000004
n = 6
        solutions = 4
        time (s) = 0.000012
n = 7
        solutions = 40
        time (s) = 0.000041
n = 8
        solutions = 92
        time (s) = 0.000166
```

```
n = 9
        solutions = 352
        time (s) = 0.000702
n = 10
        solutions = 724
        time (s) = 0.003158
n = 11
        solutions = 2680
        time (s) = 0.013234
n = 12
        solutions = 14200
        time (s) = 0.064606
n = 13
        solutions = 73712
        time (s) = 0.364948
n = 14
        solutions = 365596
        time (s) = 2.208589
n = 15
        solutions = 2279184
        time (s) = 14.228017
n = 16
        solutions = 14772512
        time (s) = 100.403137
```

Numero di path in una griglia (1)

Da qui



A qui

Numero di path in una griglia (2)

```
int solutions = 0;  // number of solutions (must be 0 when the search starts)
bool grid[M][N];    // all elements false when the search starts
int counter = 0;    // number of visited cells (must be 0 when the search starts)
```

```
void search(int x = 0, int y = 0) {
    calls++;

    if(x < 0 || x >= M || y < 0 || y >= N) return;  // Oops... gonna out of the grid!
    if(grid[x][y]) return;      // Oops... cannot visit the same cell twice!
    if(x == M-1 && y == N-1 && counter+1 == M*N) { solutions++; return; }

    // visit cell
    grid[x][y] = true;
    counter++;

    search(x+1, y); // go down
    search(x, y+1); // go right
    search(x-1, y); // go up
    search(x, y-1); // go left

    // unroll
    grid[x][y] = false;
    counter--;
}
```

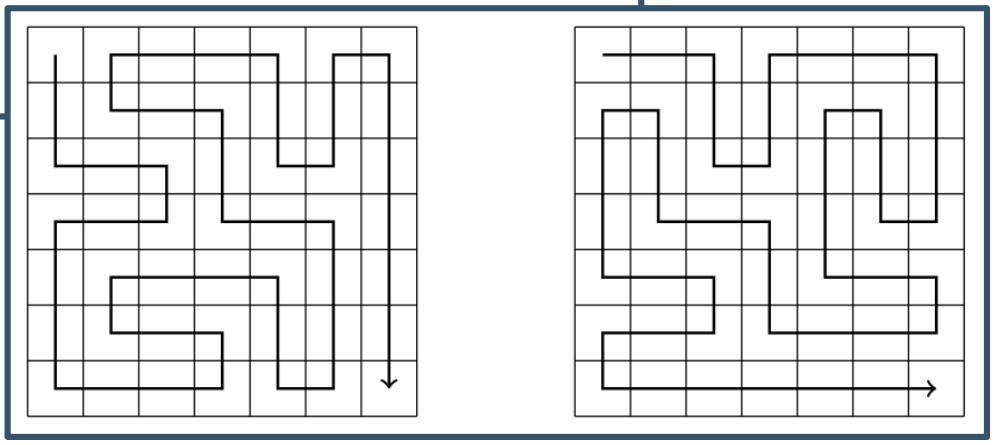
```
solutions = 111712
time (s) = 329.613621
calls = 76048396021
```

Numero di path in una griglia (3)

- Soluzioni simmetriche: inizia sempre andando in basso

```
if(x == 0 && y == 0) search(x+1, y);    // always start going down
else {
    search(x+1, y); // go down
    search(x, y+1); // go right
    search(x-1, y); // go up
    search(x, y-1); // go left
}
```

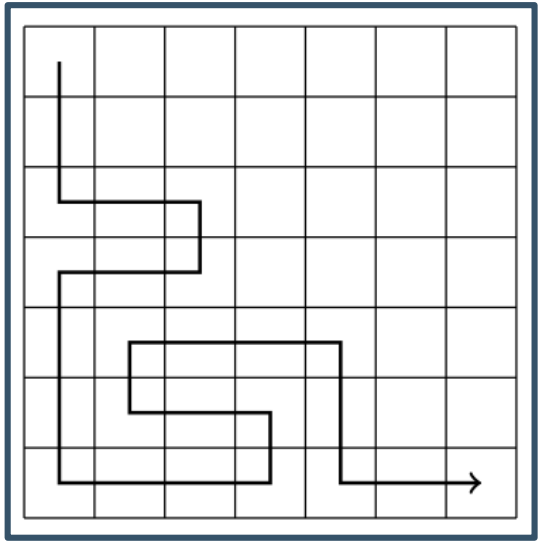
```
solutions = 111712
time (s) = 166.941440
calls = 38023974586
```



Numero di path in una griglia (4)

- La cella in basso a destra deve essere l'ultima!

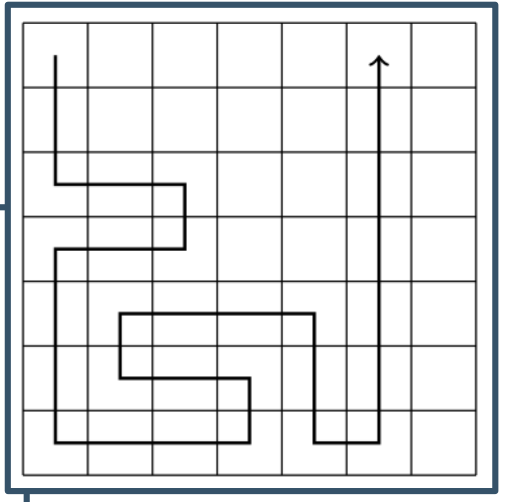
```
if(x == M-1 && y == N-1) {  // this cell must be the last one
    if(counter+1 == M*N) solutions += 2;    // this is a solution, and a symmetric one do exist
    return;
}
```



```
solutions = 111712
time (s) = 108.478530
calls = 20781409154
```

Numero di path in una griglia (5)

- Se si raggiunge un muro e si può girare sia a sinistra che a destra, allora la griglia è divisa in due parti
- Impossibili visitare tutte le celle

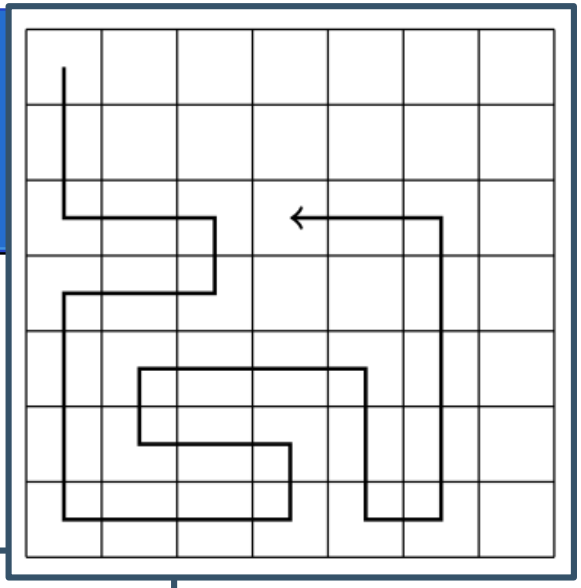


```
// check top and bottom walls
if(x == 0 || x == M-1) {
    if(y > 0 && y < N && !grid[x][y-1] && !grid[x][y+1])
        return;  // hey man... the grid is split in two parts!
}

// check left and right walls
if(y == 0 || y == N-1) {
    if(x > 0 && x < M && !grid[x-1][y] && !grid[x+1][y])
        return;  // hey man... the grid is split in two parts!
}
```

```
solutions = 111712
time (s) = 1.094212
calls = 221857654
```

Numero di path in una griglia (6)



- Generalizziamo l'idea precedente
- Alziamo un muro su ogni cella visitata

```
// check for splits
if(x == 0 || x == M-1 || (grid[x-1][y] && grid[x+1][y])) {
    if(y > 0 && y < N && !grid[x][y-1] && !grid[x][y+1])
        return;  // hey man... the grid is split in two parts!
}

// check for splits
if(y == 0 || y == N-1 || (grid[x][y-1] && grid[x][y+1])) {
    if(x > 0 && x < M && !grid[x-1][y] && !grid[x+1][y])
        return;  // hey man... the grid is split in two parts!
}
```

```
solutions = 111712
time (s) = 0.405329
calls = 69101250
```

Numero di path in una griglia (7)

- Il codice può essere semplificato se aggiungiamo un bordo

```
int solutions = 0;      // number of solutions (must be 0 when the search starts)
bool grid[M+2][N+2];    // all elements false when the search starts, but the boundary
int counter = 0;        // number of visited cells (must be 0 when the search starts)
```

```
// init boundary
for(int x = 0; x < M+2; x++) grid[x][0] = grid[x][N+1] = true;
for(int y = 0; y < N+2; y++) grid[0][y] = grid[M+1][y] = true;
```

```
// don't split!
if(grid[x-1][y] && grid[x+1][y] && !grid[x][y-1] && !grid[x][y+1]) return;
if(grid[x][y-1] && grid[x][y+1] && !grid[x-1][y] && !grid[x+1][y]) return;
```

```
solutions = 111712
time (s) = 0.381874
calls = 69101250
```

Meet in the middle (1)

- Alcuni problemi si possono dividere in due sottoproblemi
- Le soluzioni dei due sottoproblemi devono essere ricombinate
- La complessità scende da $O(2^n)$ a $O(2^{n/2})$
- Vediamo un esempio
  - Dato un insieme di interi A e un intero S, qual è la massima somma S' di un sottoinsieme di A tale che S' <= S

Meet in the middle (2)

```
int solve_with_meet_in_the_middle(int array[], int size, int S) {
    // split the array and generate all sums of the two halves
    vector<int> all_sums_1, all_sums_2;
    generate_sums(array, size / 2, S + 1, all_sums_1);
    generate_sums(array + size / 2, (size / 2 + size % 2), S + 1, all_sums_2);

    // sort the two sets of sums with different criteria
    sort(all_sums_1.begin(), all_sums_1.end());
    sort(all_sums_2.begin(), all_sums_2.end(), greater<int>());

    // sum up the last elements, and remove one of the two
    int res = 0;
    while(!all_sums_1.empty() && !all_sums_2.empty()) {
        int s = all_sums_1.back() + all_sums_2.back();
        if(s > S) { all_sums_1.pop_back(); continue; }
        res = max(res, s);
        all_sums_2.pop_back();
    }
    return res;
}
```

```
res = 12345678
time (s) = 7.786564
res = 12345678
time (s) = 0.078997
```

Esercizi suggeriti

- Piastrellature (piastrelle)
  - Branch sui due tipi di piastrelle, prima quella singola poi la doppia
- Cerca le somme (cercalesomme)
  - Provare ad aggiungere una + in ogni posizione possibile e risolvere il sottoproblema risultante
- Numero della cabala (cabala)
  - Generare tutti i numeri ammissibili e mantenere il massimo
- Menù giapponese (menu)
  - Meet in the middle + mappa per ricostruire i sottoinsiemi associati alla soluzione massima
- Gioco del tris (tris)
  - Alternare fra **esiste una mossa vincente per X** e **X vince per ogni mossa di O**
- Mosaic Composition (mosaic)
  - Rappresentare i lati con bit
  - Definire liste di tasselli compatibili sui vari lati
  - Backtracking standard, iterando sulla lista più corta di tasselli compatibili
- Forgotten attractions (coordonate)
  - I due estremi sono 0 e il massimo valore in input
  - A ogni passo, il massimo valore rimasto deve essere la prossima altezza a sinistra o a destra

Parte 2

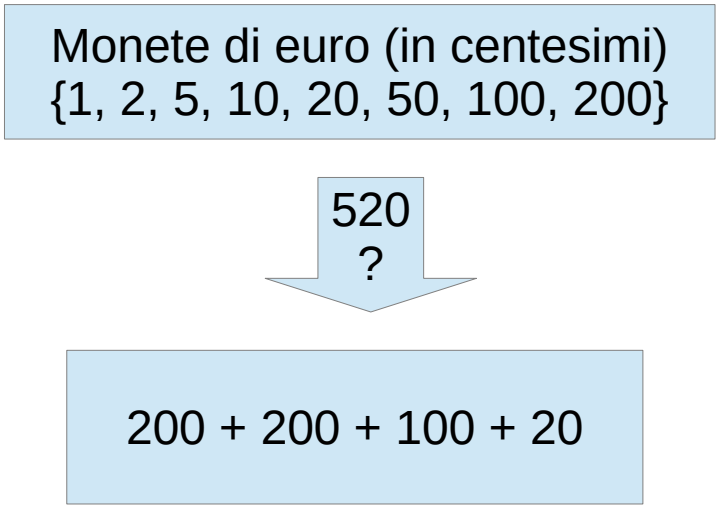
Algoritmi greedy

Proprietà di un algoritmo greedy

- Fa sempre la scelta che sembra migliore
- Non ritrattare mai le sue scelte
- Costruisce direttamente la soluzione finale
- Trovare una strategia greedy non è semplice
- Dimostrare che una strategia greedy è ottima non è banale (per fortuna non è richiesto fare dimostrazioni alle olimpiadi)

Problema delle monete (1)

Dato un insieme di (tagli di) monete (che include 1), qual è il numero minimo di monente per ottenere una data somma?



Problema delle monete (2)

- Algoritmo greedy
  1) Seleziona la moneta di valore più alto possible
  2) Decrementa la somma
  3) Se la somma non è 0, continua da 1)
- Non dà sempre l'ottimo
  - monete = {1, 3, 4} e somma = 6: 4 + 1 + 1 vs 3 + 3
- Dà sempre l'ottimo per le monete di euro
  - Anche per altri sistemi di monete (chiamati canonici)

Problema delle monete (3)

```
int coin_problem(const vector<int>& coins, int amount) {
    int res = 0;
    int next = coins.size() - 1;
    while(amount > 0) {
        while(coins[next] > amount) next--;
        amount -= coins[next];
        res++;
    }
    return res;
}
```

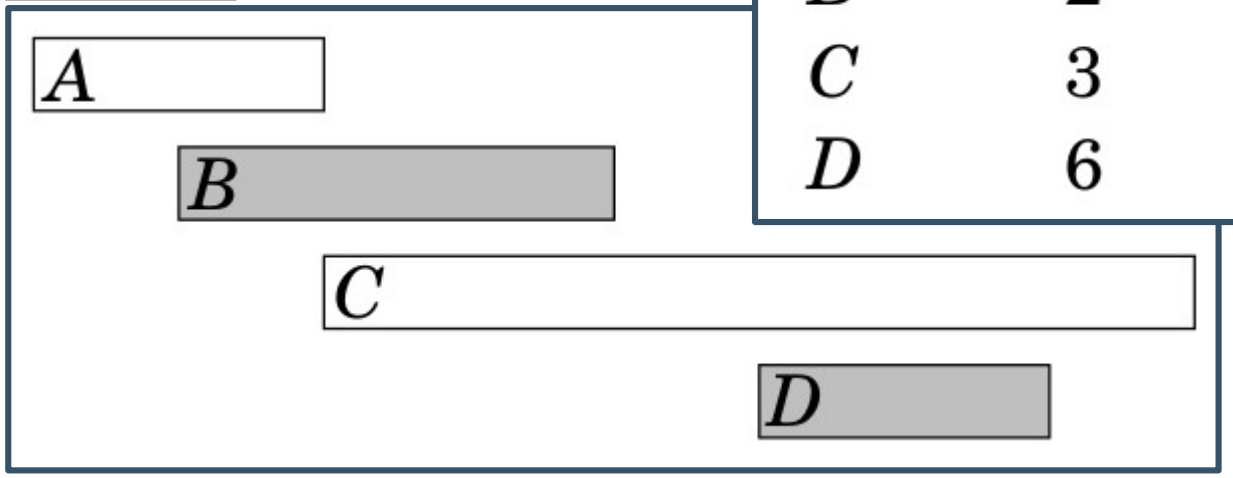
Scheduling (1)

Dati n eventi con i loro tempi di inizio e fine, trovare una programmazione che include più eventi possibile.

Istanza

event	starting time	ending time
A	1	3
B	2	5
C	3	9
D	6	8

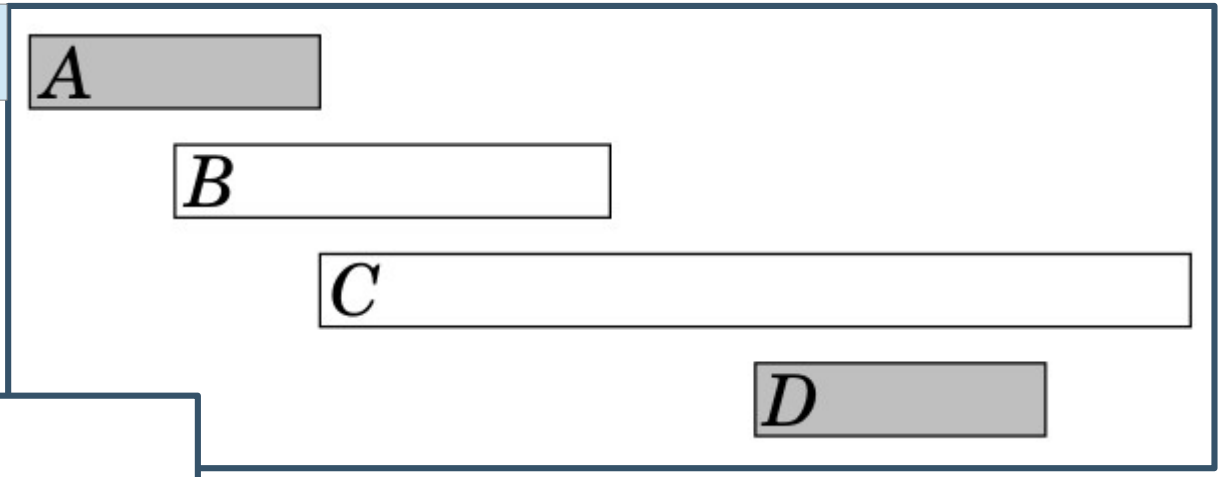
Soluzione



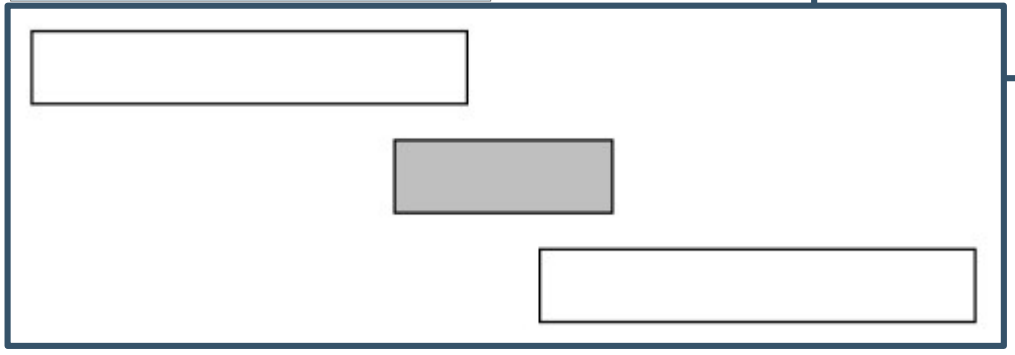
Scheduling (2)

Algoritmo 1: Seleziona gli eventi più brevi

Soluzione esempio precedente



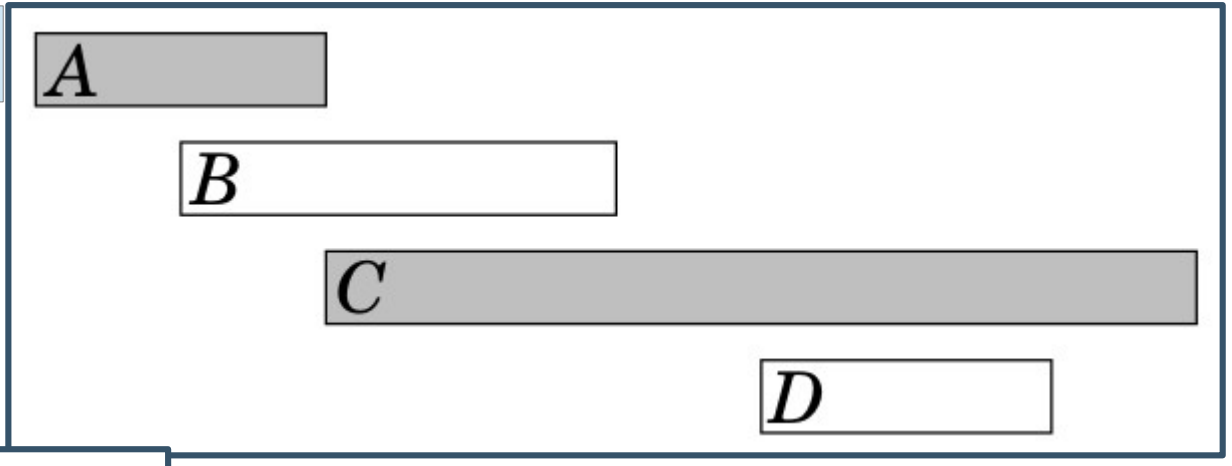
Non sempre ottimale



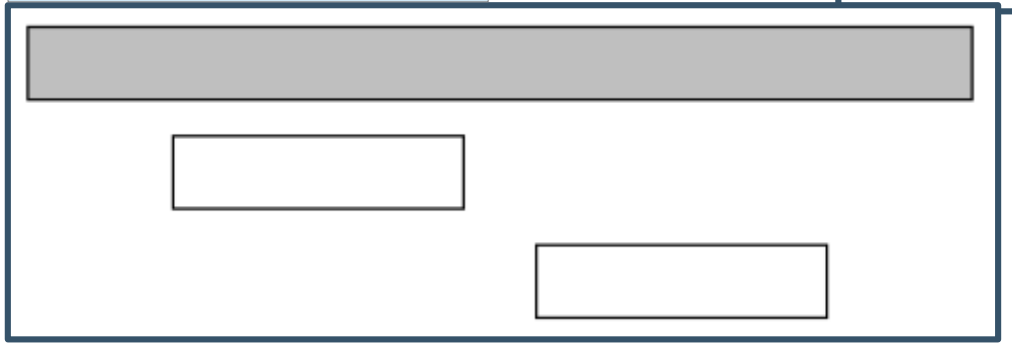
Scheduling (3)

Algoritmo 2: Seleziona l'evento che inizia prima (fra i possibili)

Soluzione esempio precedente



Non sempre ottimale

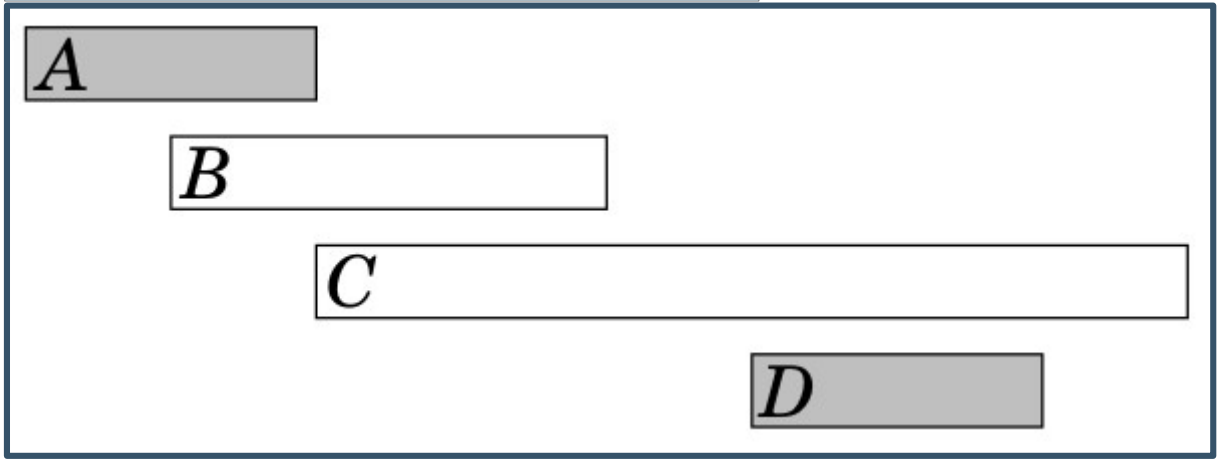


Scheduling (4)

Algoritmo 3:

Seleziona l’evento che termina prima (fra i possibili)

Soluzione esempio precedente



Sempre ottimale!

Scheduling (5)

```
bool ends_first(const tuple<string,int,int>& a, const tuple<string,int,int>& b) {
    return get<2>(a) < get<2>(b);
}

void schedule(vector<tuple<string,int,int>> events, vector<string>& res) {
    sort(events.begin(), events.end(), ends_first);
    for(int i = 0, free = 0; i < events.size(); i++) {
        if(get<1>(events[i]) < free) continue;
        res.push_back(get<0>(events[i]));
        free = get<2>(events[i]);
    }
}
```

Task con deadline (1)

Dati n task con durata e deadline, ordinare i task in modo da massimizzare $\sum(\text{deadline}_i - \text{tempo\_completamento}_i)$.

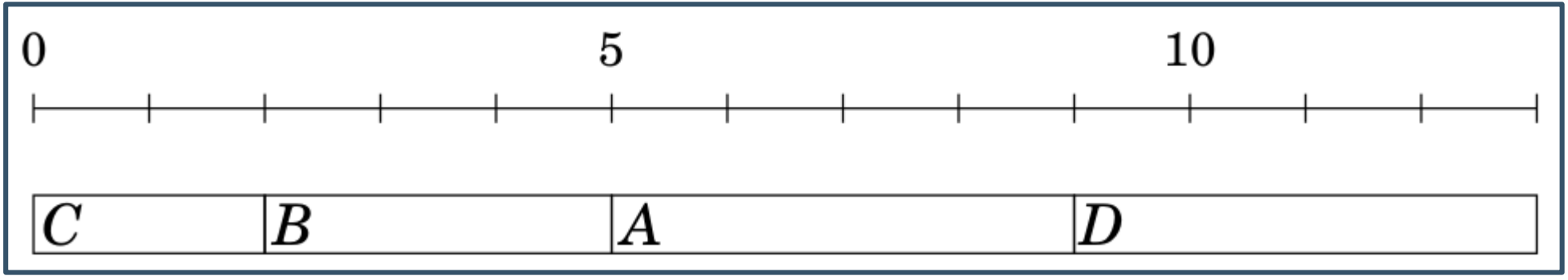
task	duration	deadline
A	4	2
B	3	5
C	2	7
D	4	5

Task con deadline (2)

task	duration	deadline
A	4	2
B	3	5
C	2	7
D	4	5

Ordinare i task per durata crescente
(deadline completamente irrilevanti)

Sempre ottimale!



Task con deadline (3)

```
bool faster(const tuple<string,int,int>& a, const tuple<string,int,int>& b) {
    return get<1>(a) < get<1>(b);
}

void schedule(vector<tuple<string,int,int>> events, vector<string>& res) {
    sort(events.begin(), events.end(), faster);
    for(int i = 0; i < events.size(); i++)
        res.push_back(get<0>(events[i]));
}
```

Minimizzazione di somme (1)

Dati n numeri a_i, trovare x che minimizza

$$|a_1 - x|^c + |a_2 - x|^c + \cdots + |a_n - x|^c$$

per

- c = 1, ovvero x minimizza lo scarto medio
- c = 2, ovvero x minimizza lo scarto quadratico medio

Minimizzazione di somme (2)

c = 1: x che minimizza

$$|a_1 - x| + |a_2 - x| + \cdots + |a_n - x|$$

$[1,2,9,2,6]$ → sort → $[1,2,2,6,9]$

mediana
x = 2

$$|1-2| + |2-2| + |9-2| + |2-2| + |6-2| = 12$$

Sempre ottimale!

Minimizzazione di somme (3)

c = 2: x che minimizza

$$(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2$$

$[1,2,9,2,6]$ → media → $(1+2+9+2+6)/5 = 4$

media
x = 4

$$(1-4)^2 + (2-4)^2 + (9-4)^2 + (2-4)^2 + (6-4)^2 = 46$$

Sempre ottimale!

Minimizzazione di somme (4)

```
int min_sum_1(vector<int> a) {
    sort(a.begin(), a.end());
    return a[a.size() / 2];
}

int min_sum_2(const vector<int>& a) {
    int res = 0;
    for(auto x : a) res += x;
    return res / a.size();
}
```

Compressione dati (1)

- Rappresentare una stringa minimizzando i bit

AABACDACA

constant-length

character	codeword
A	00
B	01
C	10
D	11

00 00 01 00 10 11 00 10 00

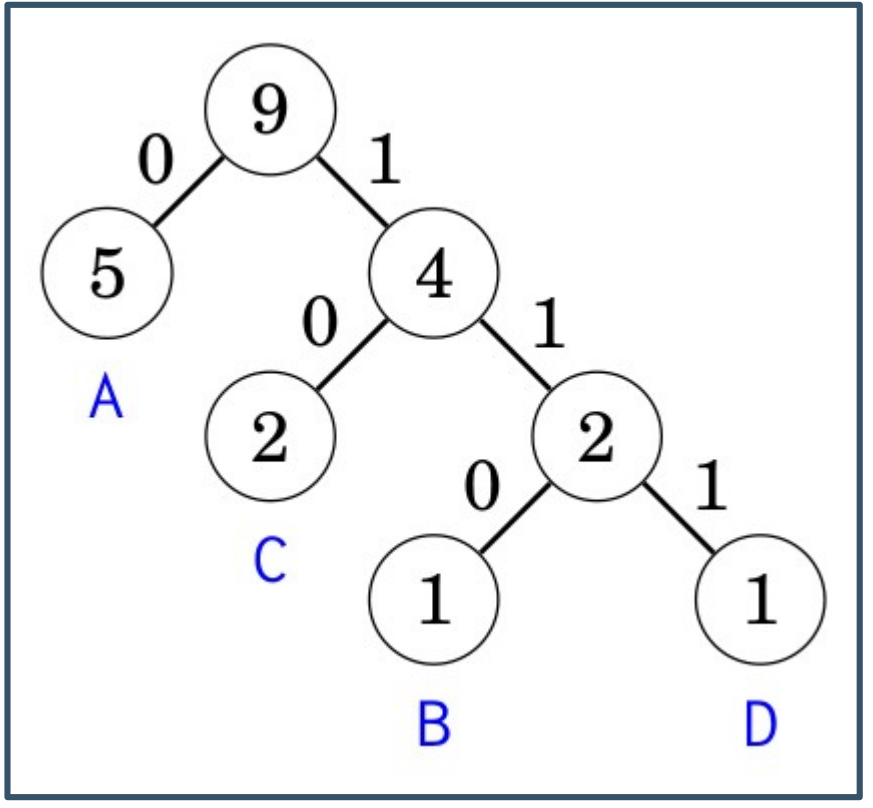
variable-length (unique prefix)

character	codeword
A	0
B	110
C	10
D	111

0 0 110 0 10 111 0 10 0

Compressione dati (2)

- Huffman coding
  - Ogni carattere è un nodo (foglia) con peso dato dalla frequenza
  - Due nodi di peso minimo diventano figli di un nuovo nodo il cui peso è la somma dei pesi dei due nodi
- L'albero rappresenta i codeword
  - 0 quando ci si sposta a sinistra
  - 1 quando ci si sposta a destra



Compressione dati (3)

```
string huffman_coding(const string& in_string) {
    // compute frequencies of chars
    vector<char> chars;
    unordered_map<char, int> freq;
    for(auto c : in_string) if(++freq[c] == 1) chars.push_back(c);

    // each char is a node whose weight is its frequency
    vector<node*> nodes;
    for(auto c : chars) nodes.push_back(new node(freq[c]));

    // iteratively merge the two less frequent nodes
    priority_queue<node*> q(nodes.begin(), nodes.end());
    while(q.size() > 1) {
        auto a = q.top();
        q.pop();
        auto b = q.top();
        q.pop();
        nodes.push_back(new node(b, a));
        q.push(nodes.back());
    }
```

Compressione dati (4)

```
    // compute wordcodes starting from leaves and reaching the root
    unordered_map<char, string> wordcode;
    for(int i = 0; i < chars.size(); i++) {
        stack<char> s;
        for(node* n = nodes[i]; n->parent != NULL; n = n->parent) s.push(n->edge);
        while(!s.empty()) { wordcode[chars[i]] += s.top(); s.pop(); }
    }

    // free dynamic memory
    for(auto x : nodes) delete x;

    // assemble the answer
    string res;
    for(auto c : in_string) res += wordcode[c];
    return res;
}
```

Esercizi suggeriti (1)

- Truffa contabile (truffa)
  - Conviene invertire il numero più negativo
- Viaggio in taxi (taxi)
  - In ogni città, se costa meno cambiare taxi, lo si cambia
- Somme costose (somme)
  - Conviene sempre sommare i due numeri più piccoli (usare un heap)
- Impila la pila (combinazione)
  - A ogni passo bisogna prendere il disco con dimensione maggiore

Esercizi suggeriti (2)

- Canottaggio (canoa)
  - Riformulare la sommatoria per capire qual è il criterio con cui conviene fare le scelte
- Esame di Aritmanzia (pota)
  - Si prende il numero più grande, poi quello più piccolo, e si ripete
- Rispetta i versi (disuguaglianze)
  - Per < conviene prendere il numero più piccolo non ancora usato
  - Per > conviene prendere il numero più grande non ancora usato

Esercizi suggeriti (3)

- Pesci Mangioni (pesci)
  - Usare uno stack per memorizzare i pesci ancora in vita
  - Se un pesce si muove verso destra potrebbe mangiare pesci dallo stack
  - Se non viene mangiato, allora entra nello stack
- Fickle Trends (mode)
  - Computiamo la massima frequenza
  - In ogni passo, se la frequenza che stiamo incrementando è >= del massimo, allora è una moda in quell'istante
  - Usiamo un unordered_set per memorizzare le mode
- Piroette (piroette)
  - Conviene prendere le cifre dalla più grande alla più piccola
  - L'ultima cifra deve essere uno 0
  - La penultima cifra deve essere la più piccola cifra pari

Esercizi suggeriti (4)

- Finanza creativa (bilancio)
  - Identificare la sequenza crescente iniziale
  - Rimuovere gli ultimi elementi della sequenza
  - Eventualmente la sequenza può essere estesa dopo una rimozione
- Treatment Planning (antibiotics)
  - Ordinare per ore modulo T
  - Per ogni classe di equivalenza conservare il rappresentante più grande
  - Se manca qualche classe di equivalenza, restituire la più piccola classe mancante
  - Altrimenti restituire il rappresentante più piccolo

Fine della lezione