Algorithmen und Datenstrukturen

4. ENTWURFSMUSTER

Überblick

- Entwurfsprinzipien
- Greedy-Algorithmen
- Divide and Conquer
- Backtracking
- Dynamische Programmierung

Weitere Muster (die wir nicht behandeln):

- Problem-solving methods
  - Propose-and-revise
  - Propose-and-backtrack
  - Hill climbing
  - Propose and exchange
  - …
- Constraint satisfaction
- Genetische Algorithmen
- Ameisenalgorithmen
- …

Einführung

- Ableitung eines optimalen Algorithmus aus Anforderungsbeschreibung ist nicht automatisierbar
- Algorithmenentwurf ist kreative Tätigkeit
- Unterstützung durch „Muster“ (best practices)
  - Vgl. Muster in der Architektur von Gebäuden
  - Vgl. Muster in der Softwarearchitektur

Einsatz von Algorithmenmustern

- Idee:

Anpassung von generischen Algorithmenmustern für bestimmte Problemklassen an eine konkrete Aufgabe

  - Dokumentation des Lösungsverfahrens am Beispiel eines einfachen Vertreters der Problemklasse
  - Bibliothek von Mustern („Design Pattern“) zur Ableitung eines abstrakten Programmrahmens
  - Programmiersprachenunterstützung durch parametrisierte Algorithmen und Vererbung

Algorithmen und Datenstrukturen

4.1 GREEDYALGORITHMEN

Algorithmenmuster: Greedy

- Greedy = „gierig“
- Prinzip
  - Versuche mit jedem Teilschritt so viel wie möglich zu erreichen

Greedy-Algorithmen (gierige Algorithmen) zeichnen sich dadurch aus, dass sie immer denjenigen Folgezustand auswählen, der zum Zeitpunkt der Wahl den größen Gewinn bzw. das beste Ergebnis verspricht

Greedy-Algorithmen am Beispiel

- Herausgabe von Wechselgeld auf Beträge unter 1 Euro
- Verfügbare Münzen mit Werten zu 50, 20, 10, 5, 2, 1 Cent

- Ziel: so wenige Münzen wie möglich ins Portemonnaie bekommen

Beispiel:

78 Cent = 50 + 2 * 10 + 5 + 2 + 1

Das Münzwechselproblem - Formalisierung

Gesucht ist ein Algorithmus mit folgenden Eigenschaften:

- Eingabe
  1. Eine natürliche Zahl `amount>0`
  2. Eine Menge von Münzwerten `currency` $=\{c_1,...,c_n\}$ (z.B. `{1,2,5,10,20,50}`)
- Ausgabe

  Ganze Zahlen `change[1],...,change[n]` mit den Eigenschaften:
  1. `change[1]*`c_1`+...+change[n]*`c_n `= amount`
  2. `change[1]+...+change[n]` ist minimal unter allen Lösungen für 1.

Anmerkung:

`change[i]` ist die Anzahl der Münzen mit Münzwert c_i für `i=1,...,n`

Das Münzwechselproblem: Algorithmus

- Algorithmus:
  1. Nehme jeweils immer die größte Münze unter Zielwert und ziehe sie von diesem ab.
  2. Verfahre derart bis Zielwert gleich Null.

Das Münzwechselproblem: Algorithmus (2)

- Algorithmus in Java:

```
public int[] moneyChange(int[] currency, int amount){
    int[] change = new int[currency.length];
    int currentCoin = currency.length-1;
    while(amount > 0){
        while(amount < currency[currentCoin] && currentCoin > 0)
            currentCoin--;
        if(amount >= currency[currentCoin] && currentCoin >= 0){
            amount -= currency[currentCoin];
            change[currentCoin]++;
        }else return null;
    }
    return change;
}
```

- Aufruf durch

```
int[] currency = {1,2,5,10,20,50};
int amount = 78;
int[] change = moneyChange(currency, amount);
```

Problem: Lokales Optimum

- Greedy-Algorithmen berechnen in jedem Schritt lokales Optimum → globales Optimum kann verfehlt werden!

- Beispiel:
  - Münzen: 11, 5, und 1
  - Zielwert: 15
  - Greedy: 11 + 1 + 1 + 1 + 1
  - Optimum: 5 + 5 + 5

- Aber: in vielen Fällen entsprechen lokale Optima den globalen, bzw. es reicht ein lokales Optimum aus!
  - Frage: wie sieht es mit unseren aktuellen Münzen aus?

Analyse

Theorem

Für `currency` endlicher Länge und mit endlichen positiven Werten und endlichem positivem `amount`, terminiert der Algorithmus `moneyChange` nach endlich vielen Schritten.

Analyse

Beweis:

```
01  public int[] moneyChange(int[] currency, int amount){
02      int[] result = new int[currency.length];
03      int currentCoin = currency.length-1;
04      while(amount > 0){
05          while(amount < currency[currentCoin] && currentCoin > 0)
06              currentCoin--;
07          if(amount >= currency[currentCoin] && currentCoin >= 0){
08              amount -= currency[currentCoin];
09              result[currentCoin]++;
10          }else return null;
11      }
12      return result;
13  }
```

- In 03 wird `currentCoin` mit einem endlichen positiven Wert initialisiert
- In 05/06 wird `currentCoin` nur dekrementiert, spätestens beim Wert 0 wird die Schleife beendet (also eine endliche Wiederholung)
- Falls 08/09 nicht ausgeführt wird, endet die Berechnung direkt in 10; andernfalls wird `amount` in 08 echt kleiner
- Irgendwann ist also die Bedingung in 04 nicht mehr gegeben und die Berechnung terminiert

Analyse

Theorem

Für Eingaben `currency` mit |`currency`|=m und `amount`=n hat der Algorithmus `moneyChange` eine Laufzeit von O(m+n).

Beweis:

```
public int[] moneyChange(int[] currency, int amount){
    int[] result = new int[currency.length];
    int currentCoin = currency.length-1;
    while(amount > 0){
        while(amount < currency[currentCoin] && currentCoin > 0)
            currentCoin--;
        if(amount >= currency[currentCoin] && currentCoin >= 0){
            amount -= currency[currentCoin];
            result[currentCoin]++;
        }else return null;
    }
    return result;
}
```

wird maximal m-mal ausgeführt

wird maximal n-mal ausgeführt (falls es nur eine Münze mit dem Wert “1” gibt)

□

Analyse

Theorem

Der Algorithmus `moneyChange` löst für `currency = {1,2,5,10,20,50}` das Münzwechselproblem.

Beweis:

Zur Erinnerung, für die Lösung muss gelten:

1. `change[1]*`c_1`+...+change[n]*`c_n `= amount`
2. `change[1]+...+change[n]` ist minimal unter allen Lösungen für 1.

Zu 1.: `amount` wird stets um den Wert einer Münze c_i verringert während `change[i]` um eins inkrementiert wird.

Analyse

Zu 2. (informell):

Wir zeigen die Aussage nur für Münzen mit den Werten 1, 2, und 5 (Argumente sind analog für 10, 20, und 50).

Zunächst gilt: 2er-Münzen sind stets 1er-Münzen zu bevorzugen (es macht keinen Sinn im Algorithmus auf eine 2er-Münze zu verzichten, um dann im nächsten Schritt mehr 1er-Münzen zu nehmen); eine optimale Lösung hat also maximal eine 1er-Münze

Weiterhin gilt, eine optimale Lösung hat nicht mehr als zwei 2er-Münzen (drei 2er Münzen können besser durch eine 1er und eine 5er-Münze dargestellt werden).

Weiterhin gilt, eine optimale Lösung kann nicht gleichzeitig eine 1er-Münze und zwei 2er-Münzen enthalten (dann wäre eine 5er-Münze optimal)

Es folgt, dass der durch 1er- und 2er-Münzen dargestellte Betrag nicht mehr als 4 sein kann. Also ist eine maximale Wahl von 5er-Münzen im Greedy-Verfahren optimal. □

Problemklasse für Greedy-Algorithmen

- Gegebene Menge von Eingabewerten
- Menge von Lösungen, die aus Eingabewerten aufgebaut sind
- Lösungen lassen sich schrittweise aus partiellen Lösungen, beginnend bei der leeren Lösung, durch Hinzunahme von Eingabewerten aufbauen
  - alternativ: bei einer ganzen Menge beginnend schrittweise jeweils ein Element wegnehmen
- Bewertungsfunktion für partielle und vollständige Lösungen
- Gesucht wird die/eine optimale Lösung

Algorithmen und Datenstrukturen

4.2 DIVIDE AND CONQUER

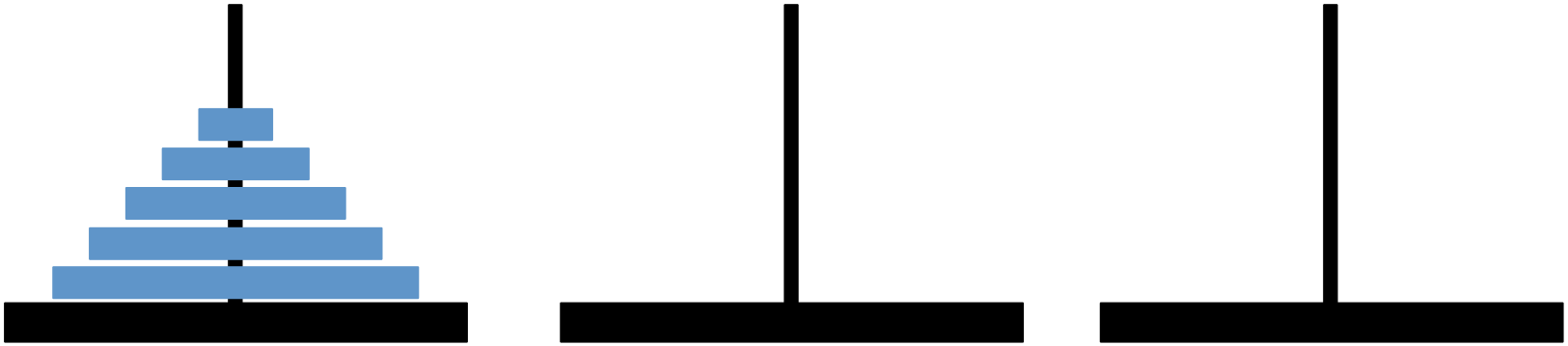
Divide and Conquer

- Auch: „Teile und Herrsche“ (divide et impera)

- Prinzip:
  - Rekursive Rückführung auf identisches Problem mit kleinerer Eingabemenge

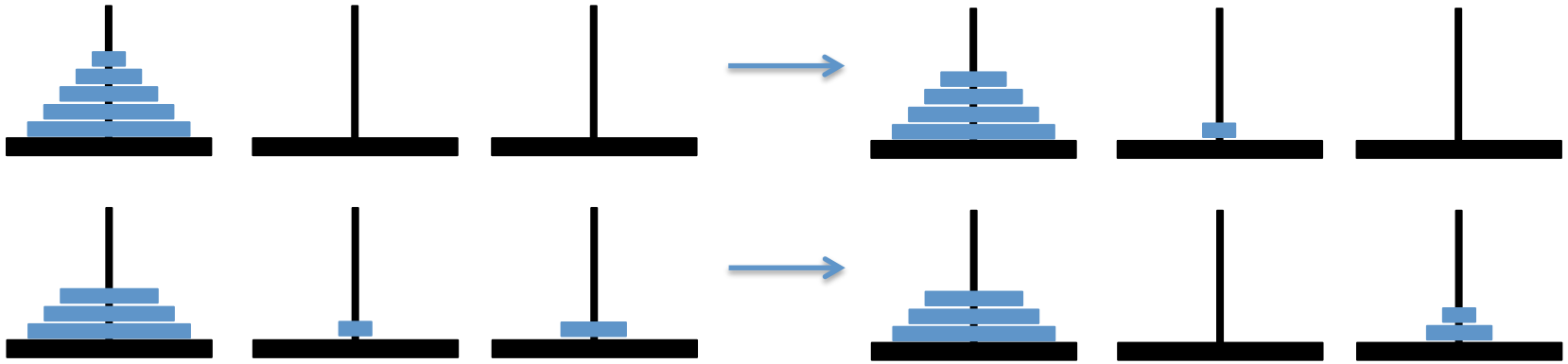
Beispiel: Türme von Hanoi 1/3

- Gegeben:
  - 3 Stapel mit Scheiben unterschiedlicher Größe
  - In jedem Schritt darf eine Scheibe, die ganz oben auf einem Stapel liegt, auf einen anderen Stapel gelegt werden; allerdings unter der Bedingung, dass sie nur auf eine kleinere Scheibe gelegt werden darf
- Ziel:
  - Verschiebe alle Scheiben vom ganz linken Stapel auf den ganz rechten Stapel



Beispiel: Türme von Hanoi 2/3

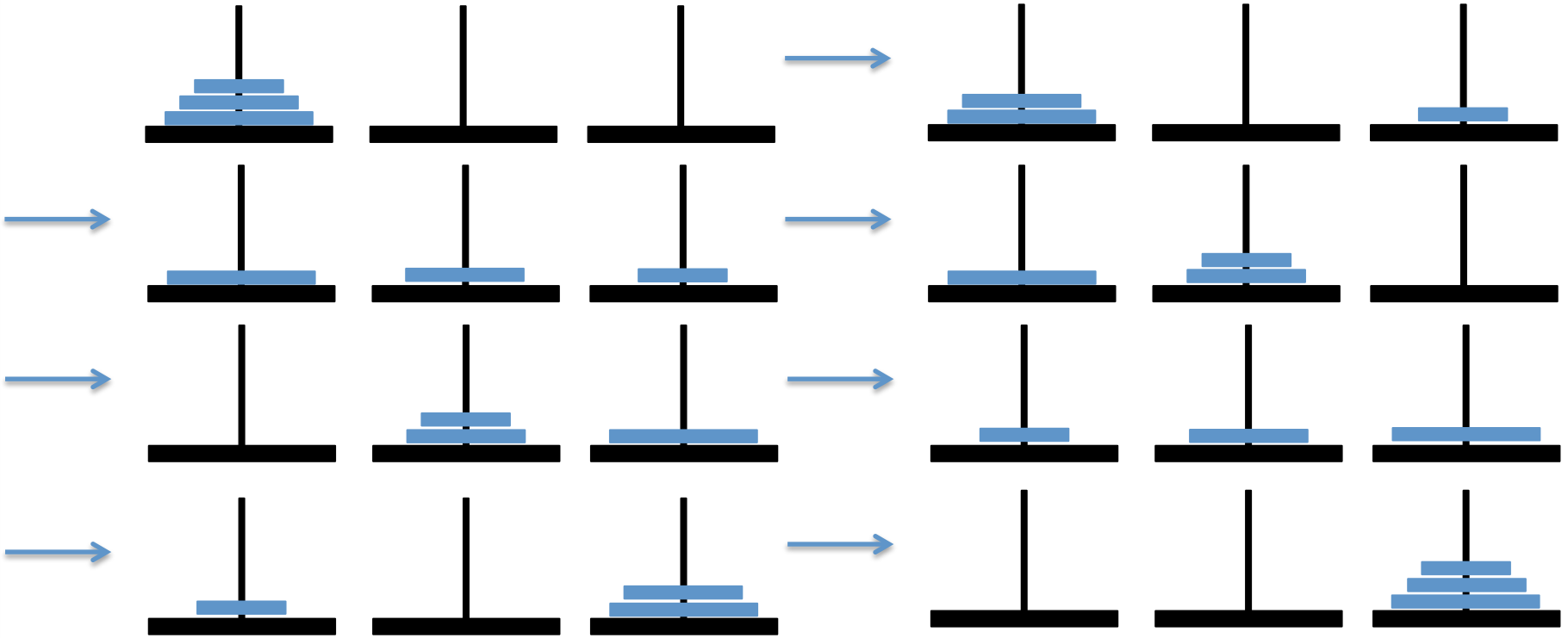
- Legale Spielzüge:



- Illegaler Spielzug:

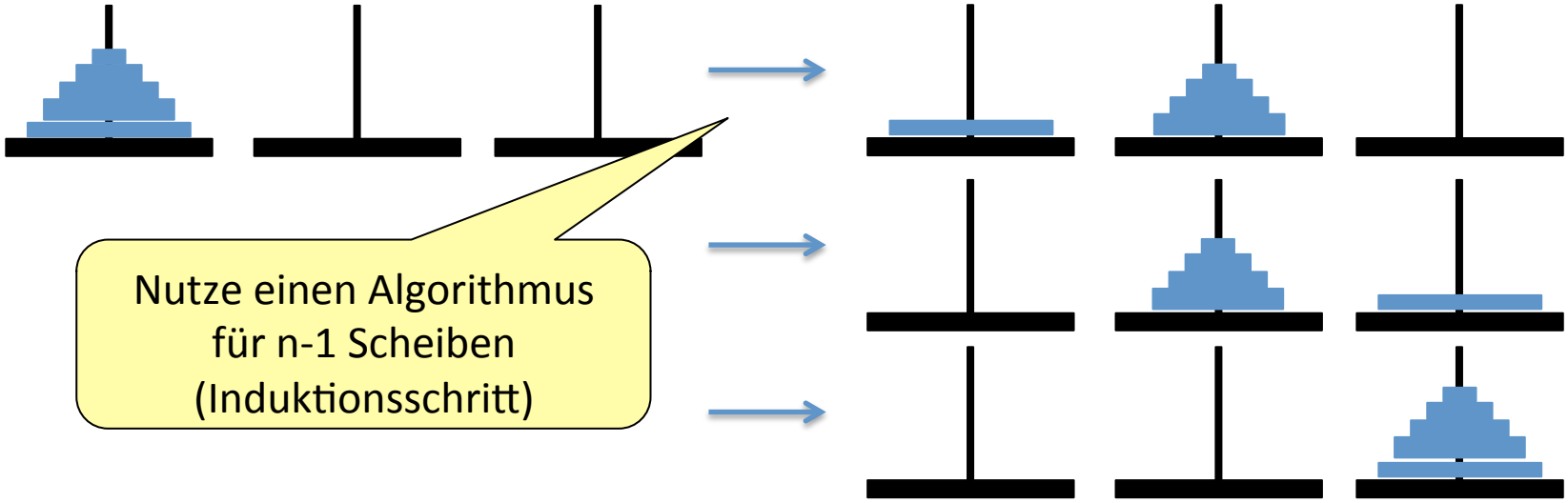
Beispiel: Türme von Hanoi 3/3

- Lösung für 3 Scheiben:

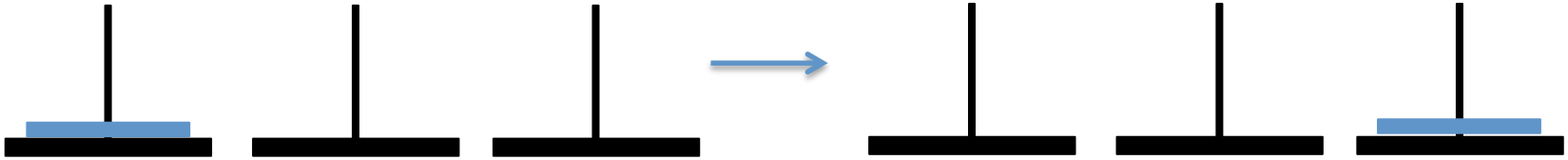


Algorithmenentwurf 1/4

- Reduziere das Problem n Scheiben zu verschieben darauf n-1 Scheiben und anschliessend nur noch eine Scheibe zu verschieben (ähnliches Prinzip wie bei Induktionsbeweisen):

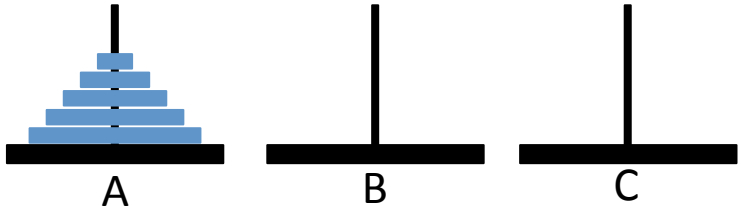


- Basisfall (Induktionsanfang) für eine Scheibe ist trivial:



Algorithmenentwurf 2/4

- Beachte, dass beim Verschieben von mehr als einer Scheibe der dritte Stapel stets als “Zwischenlager” genutzt werden muss
- Die “Rollen” der Stapel wechseln je nach Schritt



- Für das Gesamtziel (alle 5 Scheiben) dient A als Startstapel, C als Zielstapel und B als Zwischenlager
- Für das erste Unterziel (verschiebe die obersten vier Scheiben) dient A als Startstapel, B als Zielstapel und C als Zwischenlager
- Es muss stets gewährleistet sein, dass das Zwischenlager nach Abschluss wieder wie zuvor aussieht

Algorithmenentwurf 3/4

- Pseudocode

```
Algorithmus hanoi
Eingabe:
     Startstapel S,
     Zielstapel Z,
     Zwischenlager L,
     Anzahl der Scheiben n
Ausgabe:     Aktionenfolge um alle Scheiben von S nach
             L zu veschieben

Falls n=1
     Entferne die oberste Scheibe k von S und füge sie Z hinzu
     Gib aus „Verschiebe k von S nach Z“
Ansonsten
     hanoi(S,L,Z,n-1);
     Entferne die oberste Scheibe k von S und füge sie Z hinzu
     Gib aus „Verschiebe k von S nach Z“
     hanoi(L,Z,S,n-1);
```

Algorithmenentwurf 4/4

- Für die Implementierung in Java benutzen wird als Repräsentation der Stapel je ein `Stack` und für eine Scheibe je ein `int` (der Wert gibt die Größe der Scheibe an)

```
public void hanoi(Stack<Integer> start, Stack<Integer> goal,
     Stack<Integer> tmp, int numDiscs){
     if(numDiscs == 1){
          int disc = start.pop();
          goal.push(disc);
          System.out.println("Moving disc " + disc);
     }else{
          hanoi(start, tmp, goal, numDiscs-1);
          int disc = start.pop();
          goal.push(disc);
          System.out.println("Moving disc " + disc);
          hanoi(tmp, goal, start, numDiscs-1);
     }
}
```

- Aufruf durch

```
Stack<Integer> start = new Stack<Integer>();
for(int i = 5; i > 0; i--) start.push(i);
hanoi(start, new Stack<Integer>(), new Stack<Integer>(), 5);
```

Analyse

Theorem

Der Algorithmus `hanoi` terminiert nach endlich vielen Schritten (bei positiver Anzahl von Scheiben).

Beweis:

```
public void hanoi(Stack<Integer> start, Stack<Integer> goal,
     Stack<Integer> tmp, int numDiscs){
     if(numDiscs == 1){
          int disc = start.pop();
          goal.push(disc);
          System.out.println("Moving disc " + disc);
     }else{
          hanoi(start, tmp, goal, numDiscs-1);
          int disc = start.pop();
          goal.push(disc);
          System.out.println("Moving disc " + disc);
          hanoi(tmp, goal, start, numDiscs-1);
     }
}
```

Rekursionsende bei numDiscs = 1

Rekursive Aufrufe verringern den Wert von numDiscs stets um 1

□

Analyse

Theorem

Für n Scheiben hat der Algorithmus `hanoi` eine Laufzeit von $O(2^n)$.

Beweis:

Rekursionsgleichung für `hanoi`:

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n-1) + O(1) & \text{sonst} \end{cases}$$

Beweis durch Induktion trivial (Übung). □

Analyse

Theorem

Der Algorithmus `hanoi` löst das Problem der Türme von Hanoi.

Beweis:

Wir müssen folgende Aussagen zeigen:

1. Der Algorithmus hält sich an die Spielregeln (in jedem Zug wird eine Scheibe von oben von einem Stapel entfernt und auf einen leeren Stapel oder auf eine größere Scheibe gelegt)
2. Bei der Terminierung sind alle Scheiben auf dem Zielstapel

Analyse

Wir zeigen beide Aussagen durch Induktion nach der Anzahl der Scheiben n:

- n=1: hier wird eine Scheibe direkt vom Startstapel zum leeren Zielstapel verschoben (beide Bedingungen sind erfüllt)
- n-1→ n: Es sind n Scheiben von Stapel A zu C zu verschieben (sei B der dritte Stapel).
  Zunächst wird der Algorithmus rekursiv für die obersten n-1 Scheiben mit Zielstapel B aufgerufen. Da alle diese n-1 Scheiben kleiner als die unterste Scheibe ist (und diese nicht bewegt wird) ist dies das gleiche Problem, als wenn die unterste Scheibe gar nicht da wäre.

Analyse

Nach rekursiven Aufruf und IV werden also die n-1 Scheiben legal nach B verschoben, C ist anschliessend wieder leer.
Die Verschiebung der untersten Scheibe nach C ist legal.
Die rekursive Verschiebung der auf B liegenden n-1 Scheiben nach C ist nach IV wieder legal (und aufgrund der Tatsache, dass auf C nur eine größere Scheibe liegt). □

Anmerkungen

- Der vorgestellte Algorithmus ist auch optimal, d.h. er findet eine minimale Anzahl von Zügen zur Lösung des Problems
- Eine schöne Animation des Algorithmus können Sie übrigens unter
  http://britton.disted.camosun.bc.ca/hanoi.swf
  finden.

Problemklasse für Divide and Conquer

- Problem ist *selbstähnlich*, d.h. Teilmengen sind wieder Instanzen des Problems
- Teilung und Kombination von Teillösungen zu Gesamtlösung ist verhältnismäßig einfach
- Kombination von Teillösungen
  - Aufruf als Subprozedur (Türme von Hanoi)
  - Auswahl der besten Teillösung (Binäre Suche)
  - Allgemeine Kombination (MergeSort)
- Normalerweise nicht für Optimierungsprobleme anwendbar

Algorithmen und Datenstrukturen

4.3 BACKTRACKING

Backtracking: Idee

Backtracking: Versuch-und-Irrtum-Prinzip (trial and error)
- Versuche, erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen
- falls Teillösung nicht zu einer Lösung führen kann
  - letzten Schritt bzw. die letzten Schritte zurücknehmen und stattdessen alternative Wege probieren

- Alle in Frage kommenden Lösungswege werden ausprobiert
- Vorhandene Lösung wird entweder gefunden (unter Umständen nach sehr langer Laufzeit) oder es existiert definitiv keine Lösung

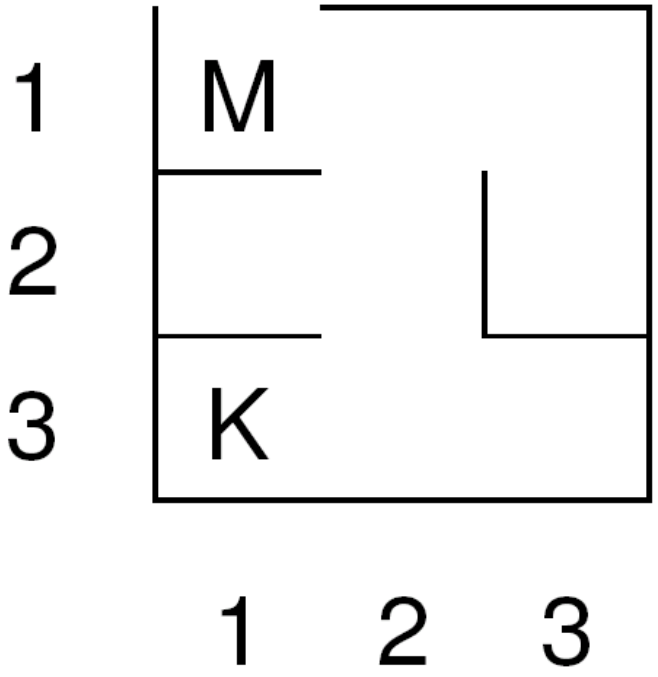
Backtracking

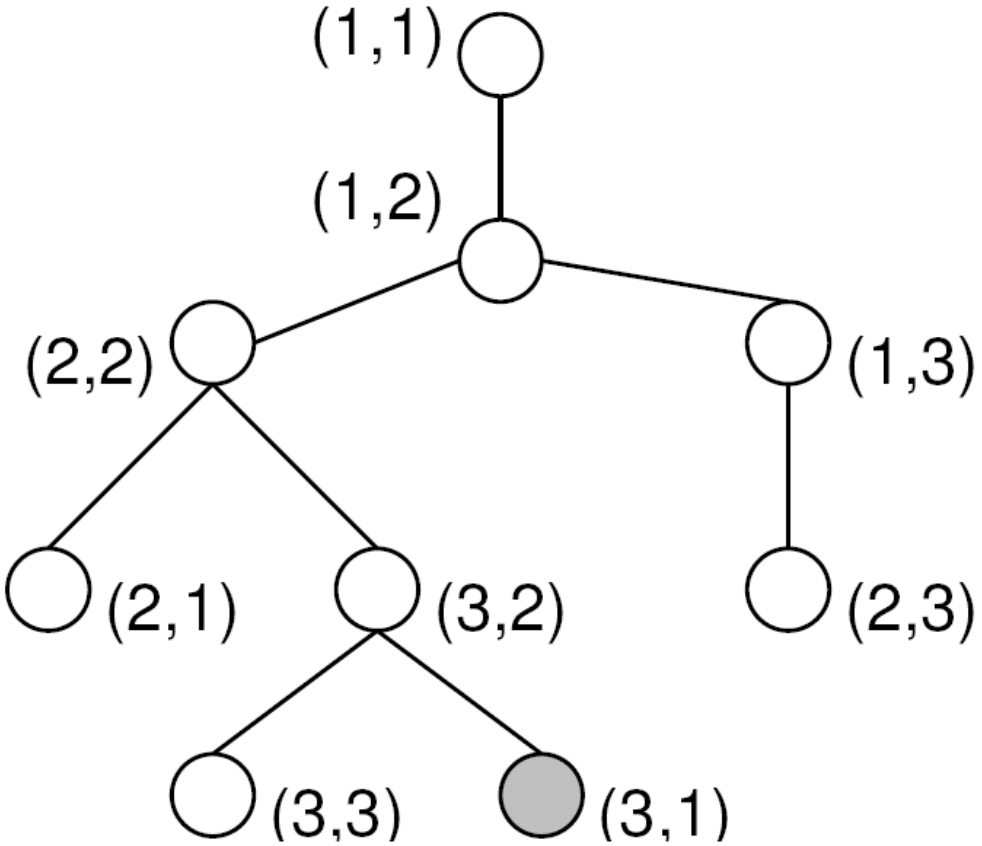
- Backtracking (“Zurückverfolgen“): Allgemeine systematische Suchtechnik
  - KF Menge von Konfigurationen K (Konfigurationsraum)
  - K_0 Anfangskonfiguration
  - für jede Konfiguration K_i direkte Erweiterungen $K_{i,1}$, …,K_{i,n_i}
  - Für jede Konfiguration ist entscheidbar, ob sie eine Lösung ist

- Aufruf: BACKTRACK(K_0)

Labyrinth-Suche

Wie findet die Maus den Käse?





Backtracking-Muster

```
procedure BACKTRACK (K: Konfiguration)
begin
  …
  if [ K ist Lösung ]
  then [ gib K aus ]
  else
   for each [ jede direkte Erweiterung K⁰ von
  K ]
      do
         BACKTRACK (K⁰)
      od
  fi
end
```

Typische Einsatzfelder des Backtracking

- Spielprogramme (Schach, Dame, Labyrinthsuche, …)
- Erfüllbarkeit von logischen Aussagen
  - logische Programmiersprachen,
  - Optimierung von Gattern
  - Model checking → Theorembeweiser
- Planungsprobleme, Konfigurationen
  - Logistische Fragestellungen (Traveling Salesman)
  - Kürzeste Wege, optimale Verteilungen,…
  - Färben von Landkarten
  - Nichtdeterministisch-lösbare Probleme

Beispiel: Acht-Damen-Problem

Gesucht: Konfiguration von 8 Damen auf einem 8x8-Schachbrett, so dass keine Dame eine andere bedroht

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

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

Menge der Konfigurationen

- Gesucht: geeignetes KF
  1. $L \subseteq KF$ für Lösungskonfigurationen L
  2. Für jedes $k \in KF$ ist leicht entscheidbar, ob $k \in L$
  3. Konfigurationen lassen sich schrittweise erweitern
     $\rightarrow$ hierarchische Struktur
  4. KF sollte nicht zu groß sein!

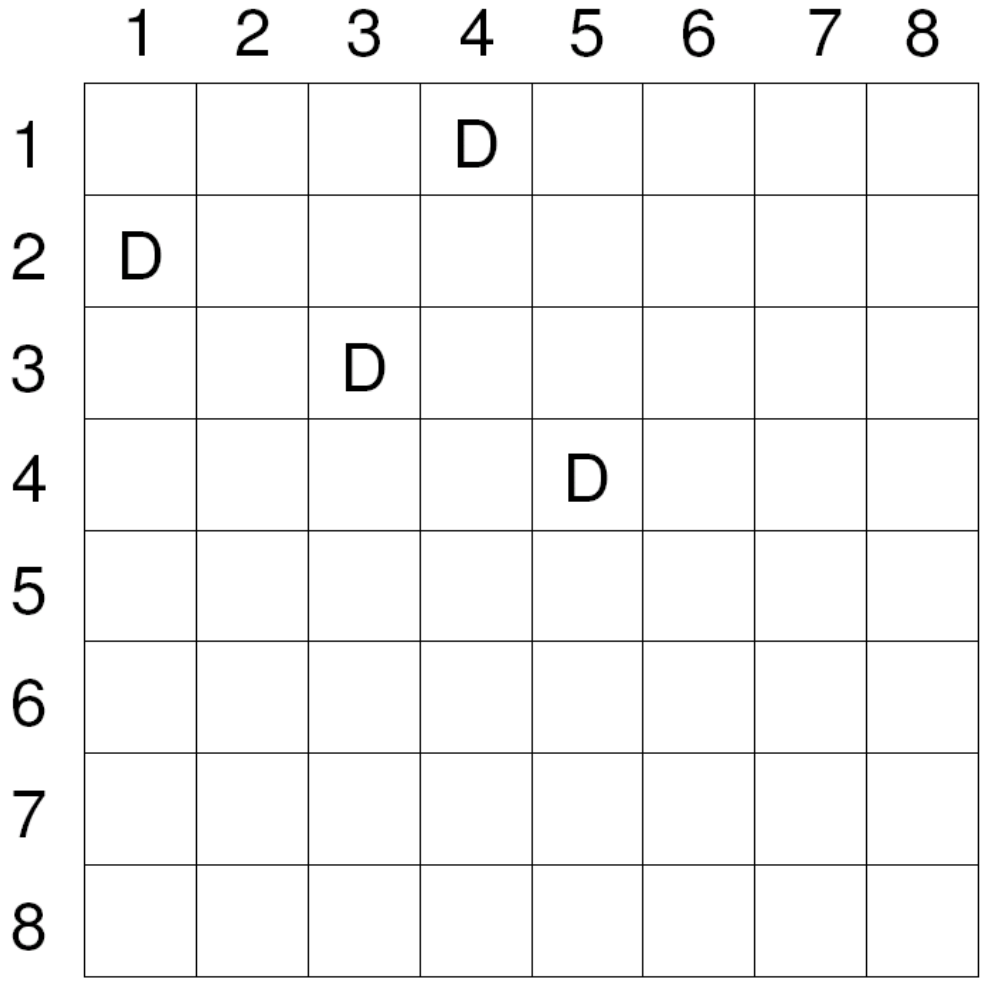
Menge der Lösungskonfigurationen L

L_1: Es sind 8 Damen auf dem Brett.

L_2: Keine zwei Damen bedrohen sich.

- Geschickte Wahl von KF:
  - Konfigurationen mit je einer Dame in den ersten n Zeilen,
    $0 \leq n \leq 8$, so dass diese sich nicht bedrohen

Nicht erweiterbare Konfigurationen

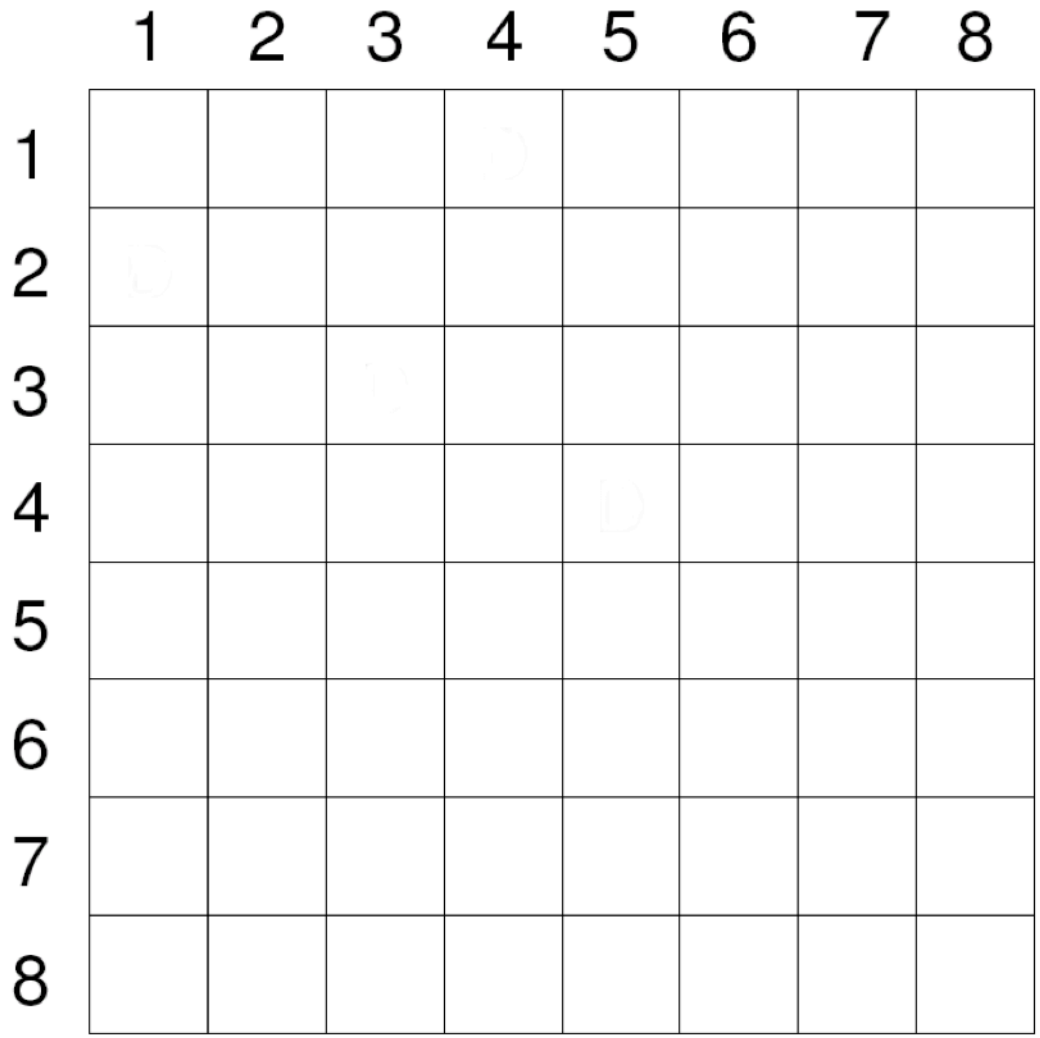


…jedes Feld in Zeile 7 ist bereits bedroht!

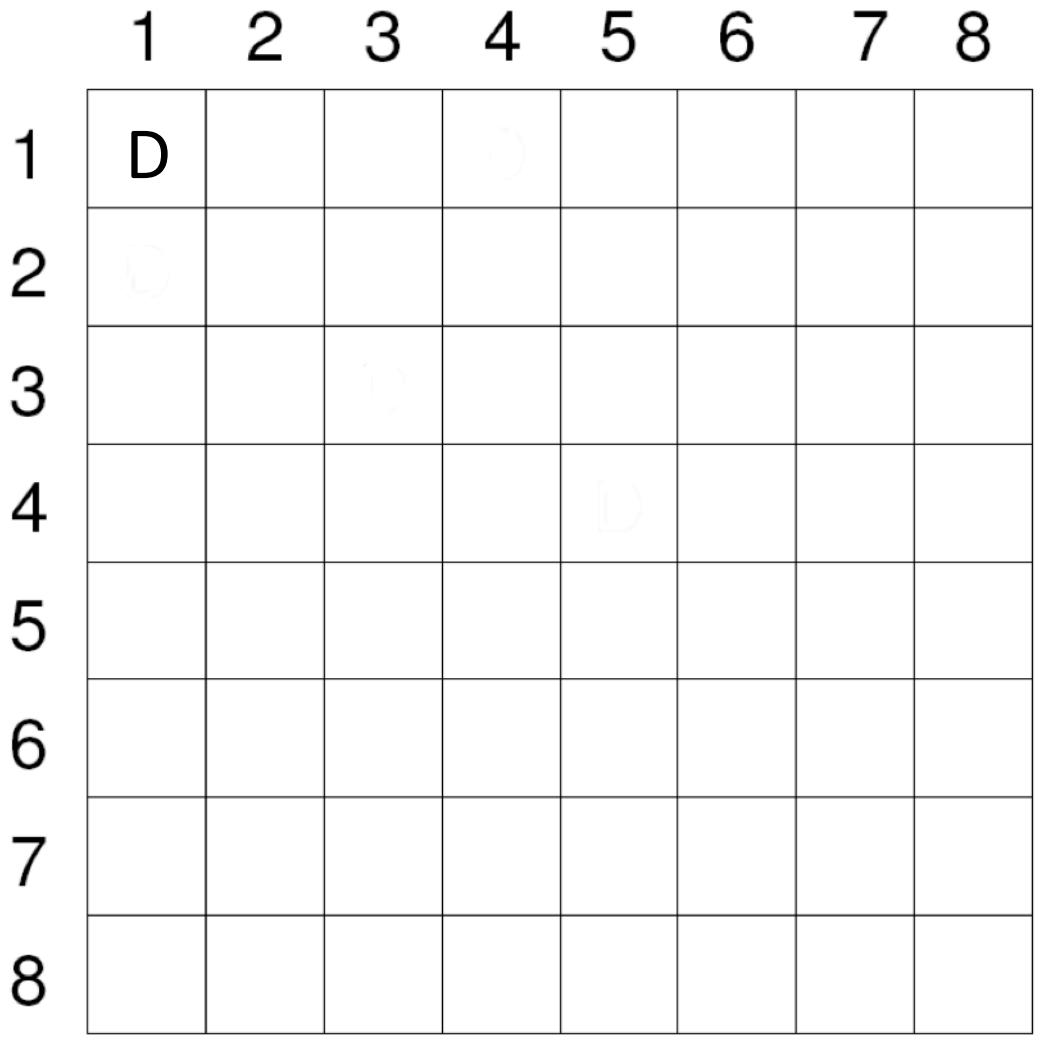
Acht-Damen-Problem

```
Procedure PLATZIERE ( i : [1…8]);
begin
 var h : [1…8];
 for h :=1 to 8 do
  if [Feld in Zeile i , Spalte h nicht bedroht]
  then
    [Setze Dame auf dieses Feld (i, h) ];
    if [Brett voll] /* i = 8 */
    then [Gib Konfiguration aus]
    else PLATZIERE (i+1)
   fi
  fi
 od
end
```

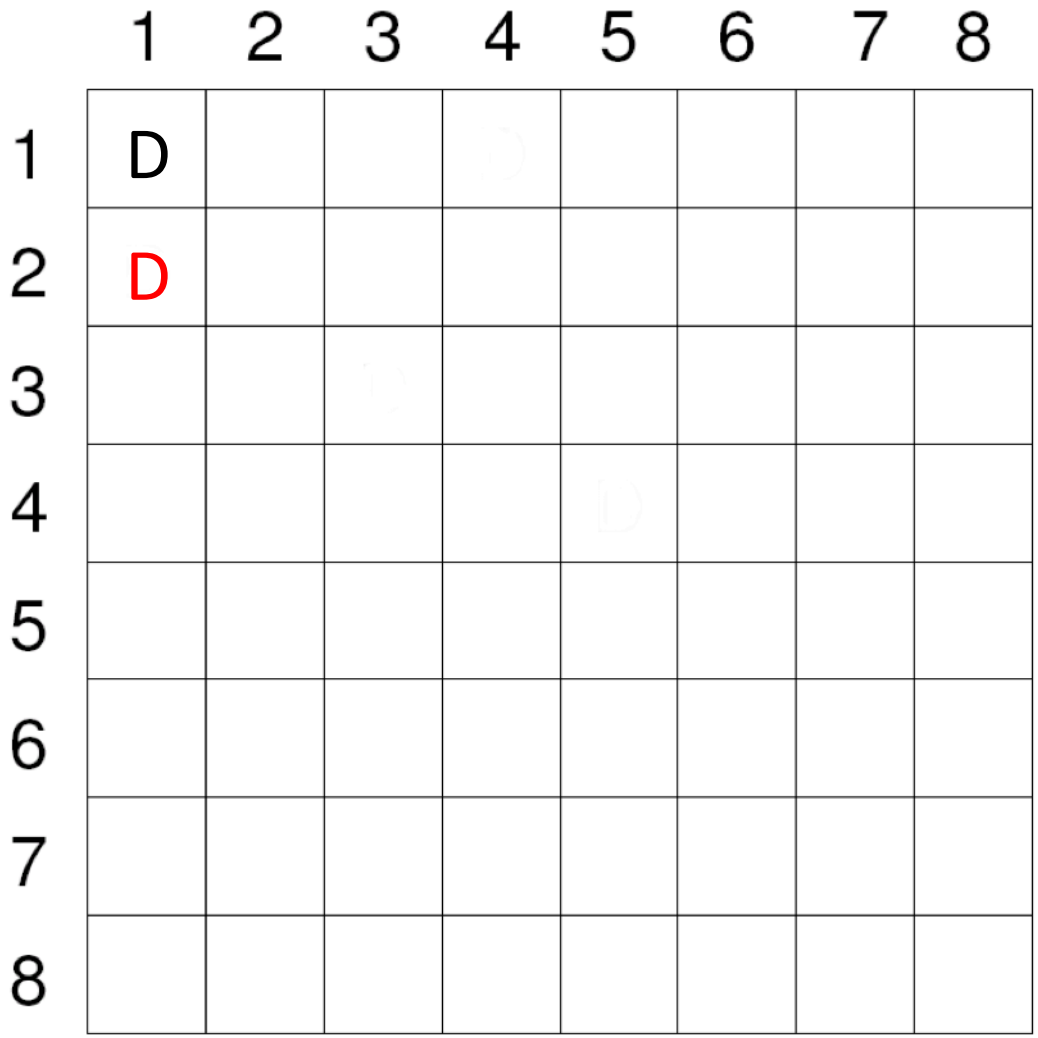
Beispiel



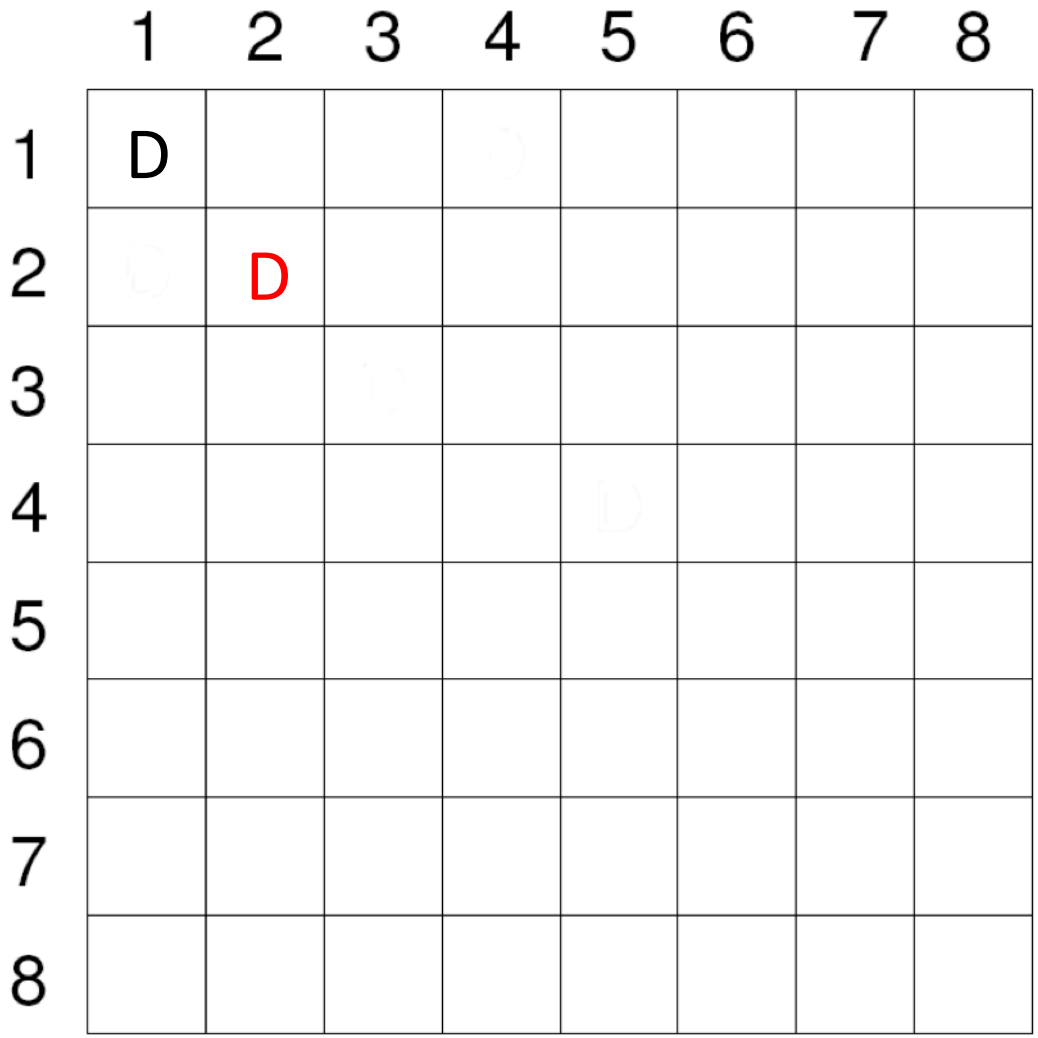
Beispiel



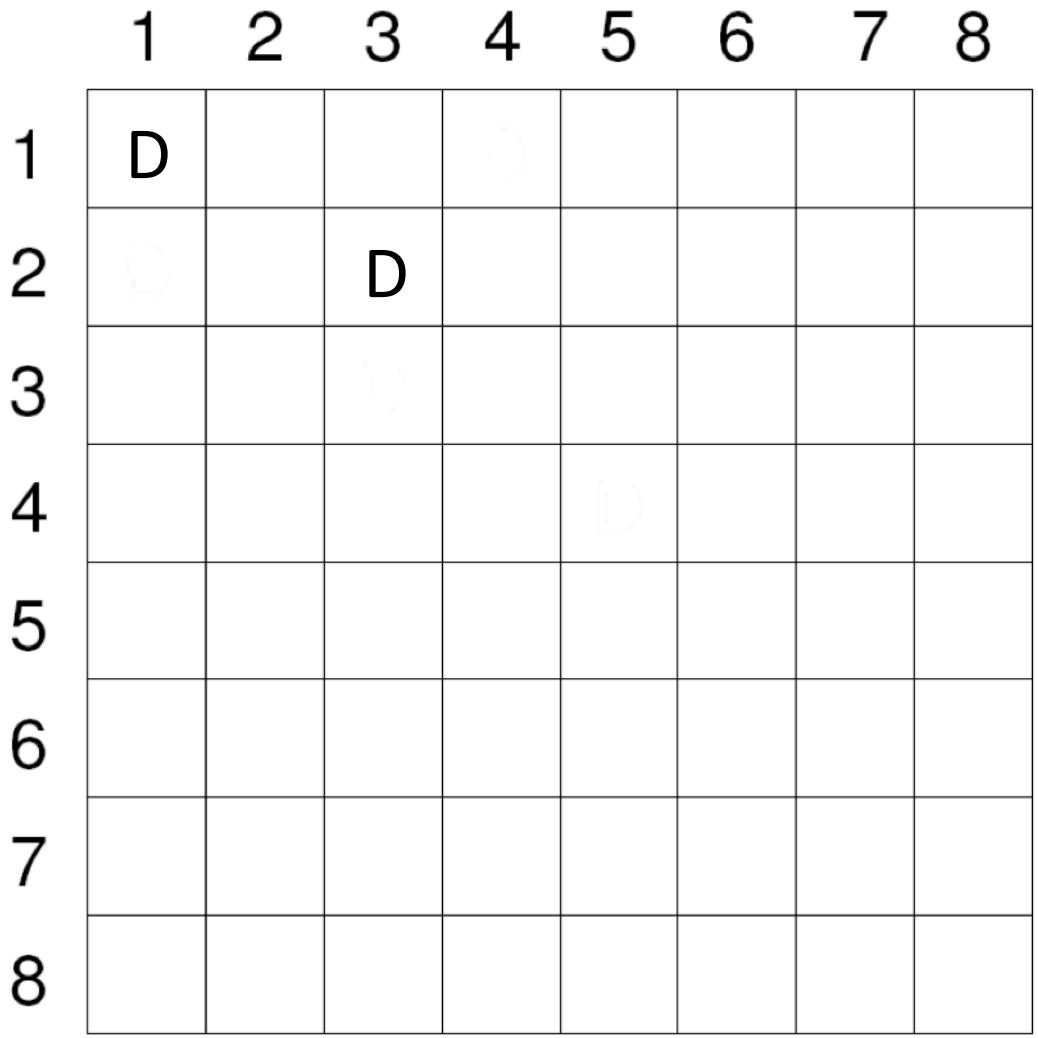
Beispiel



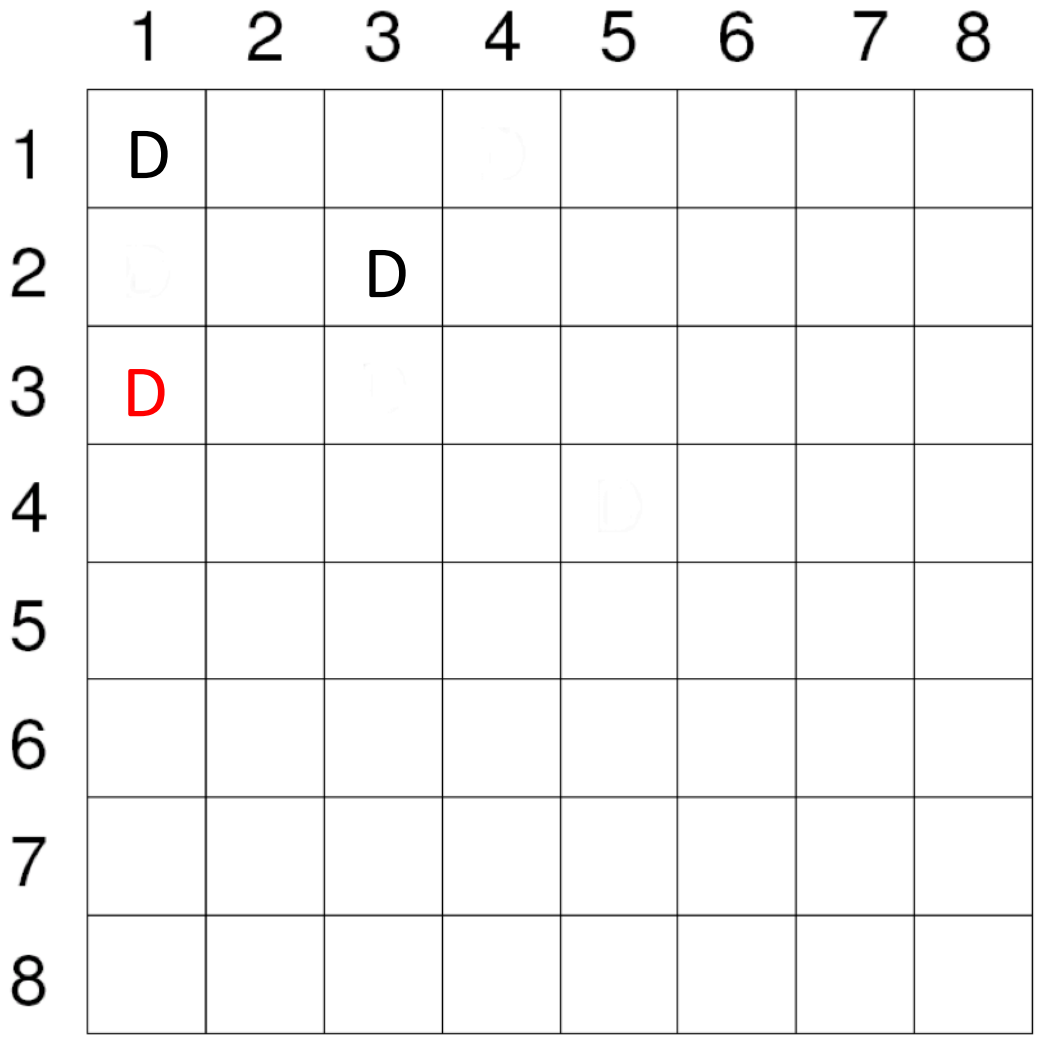
Beispiel



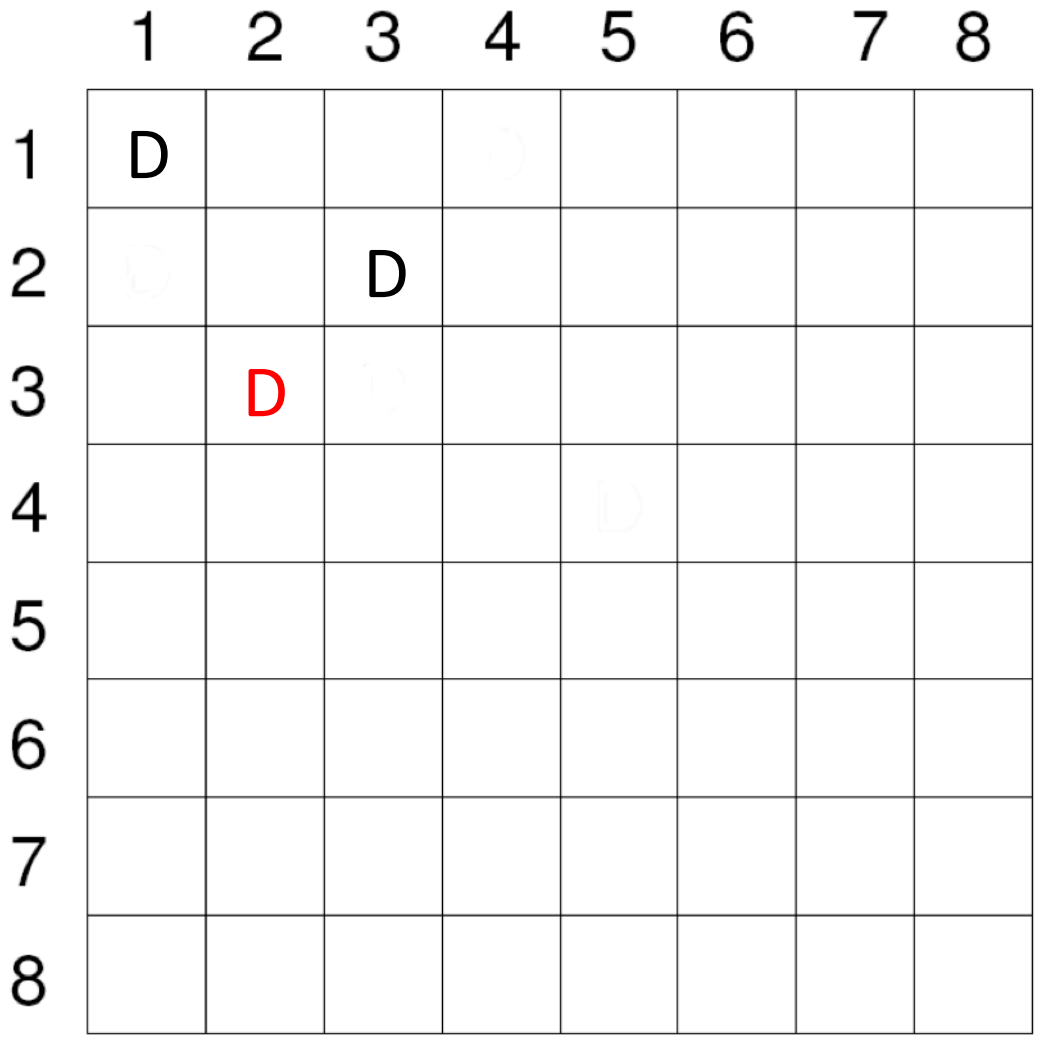
Beispiel



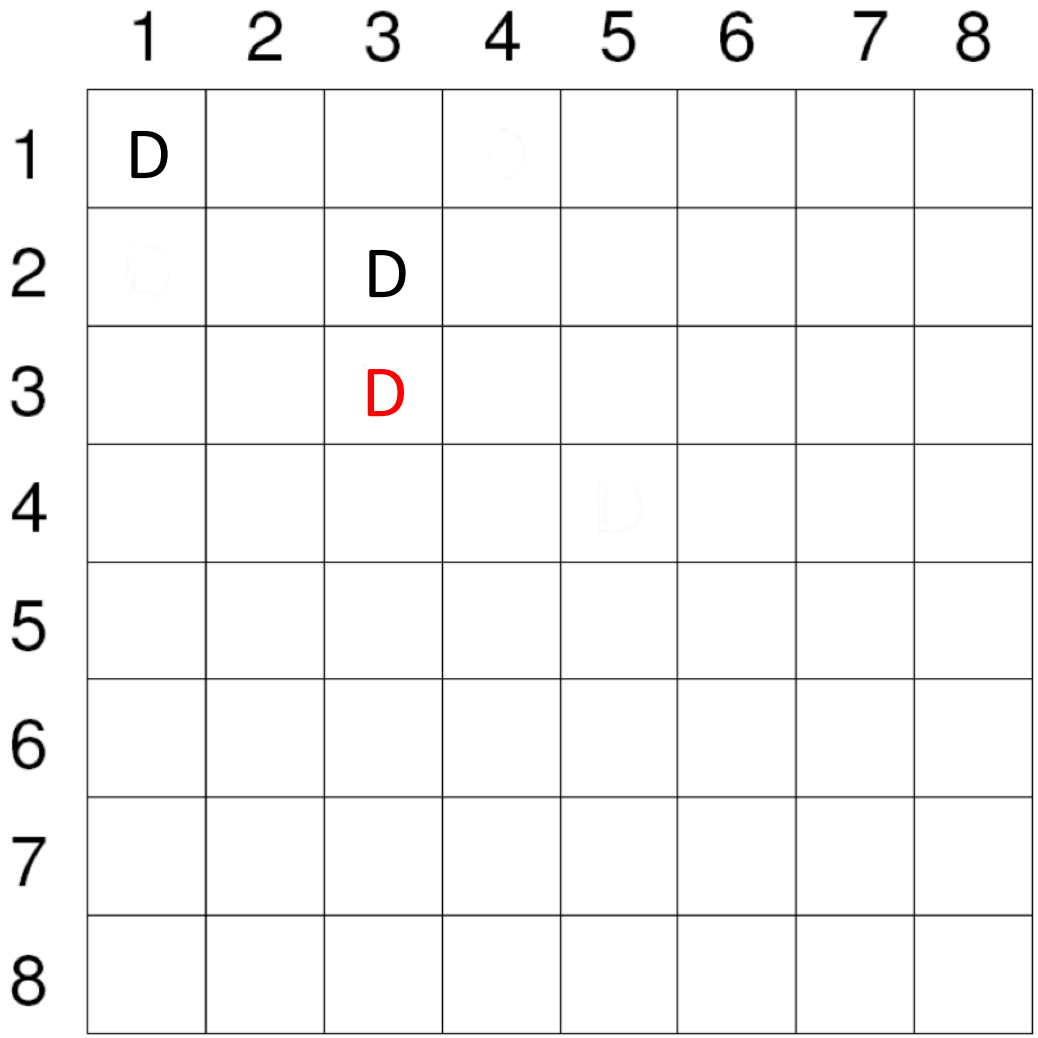
Beispiel



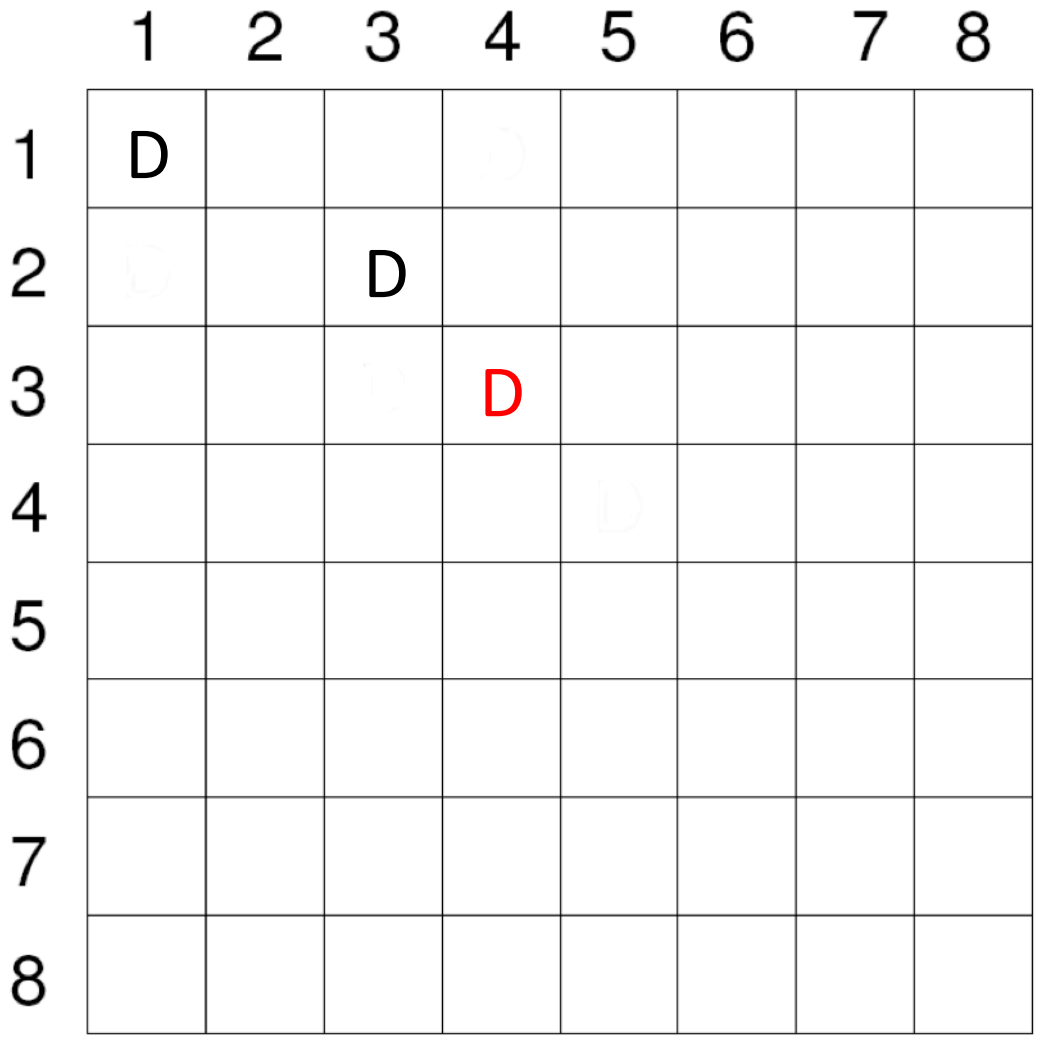
Beispiel



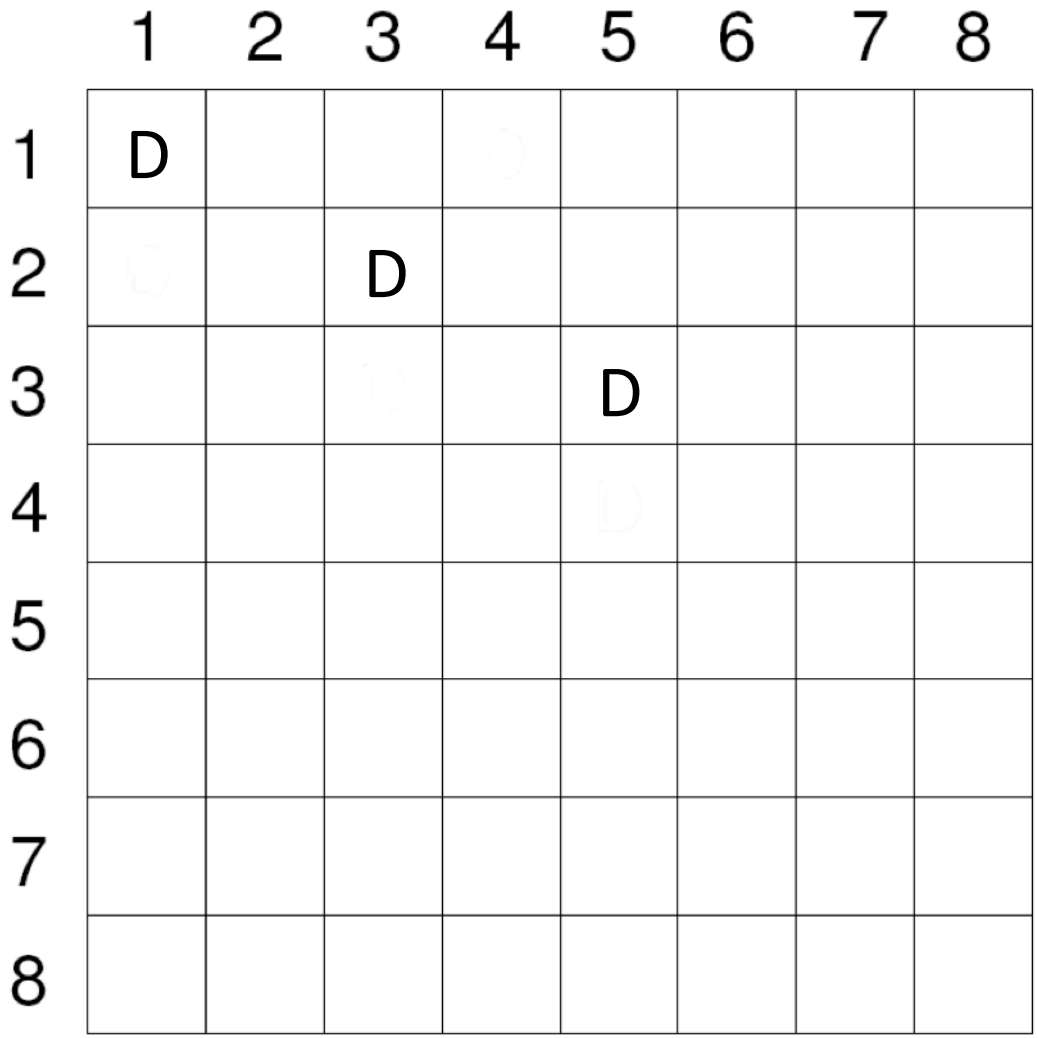
Beispiel



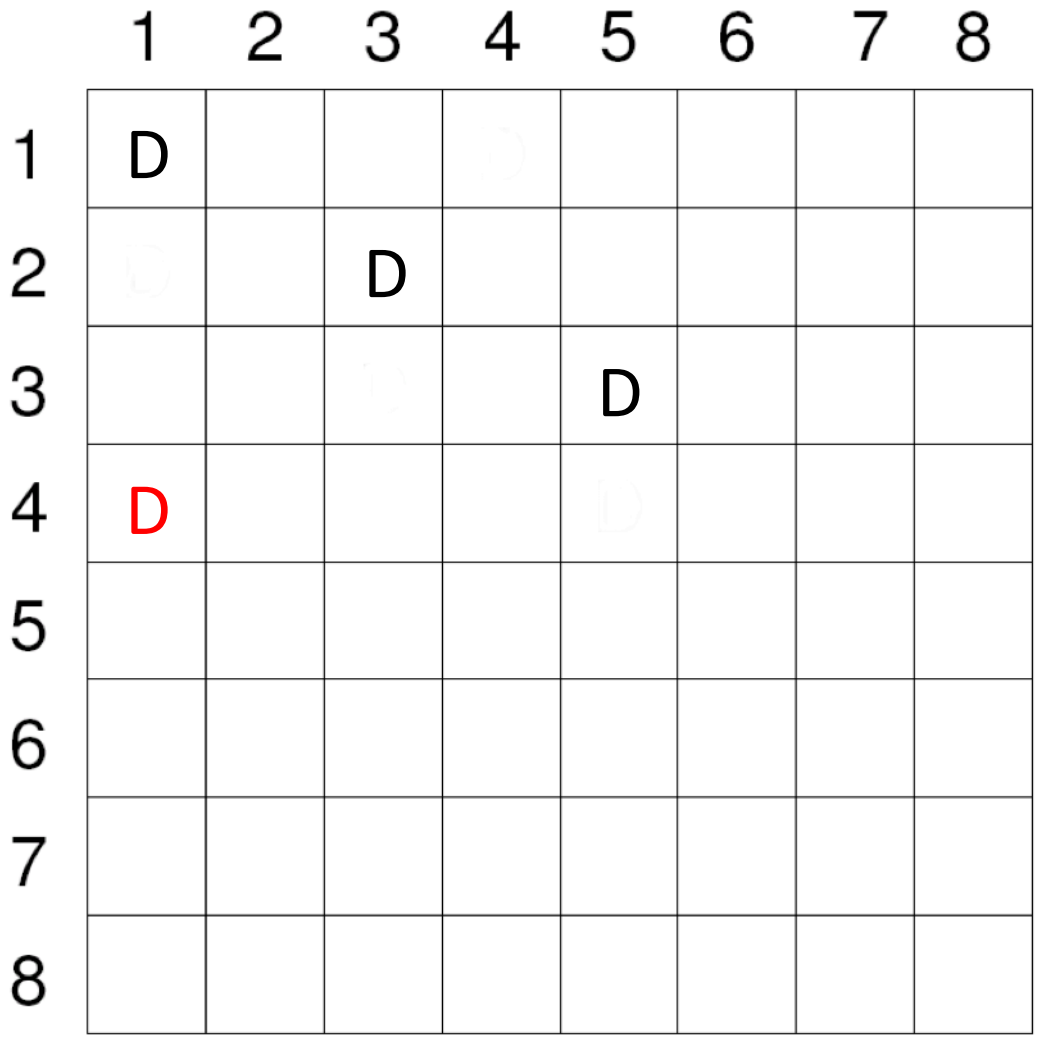
Beispiel



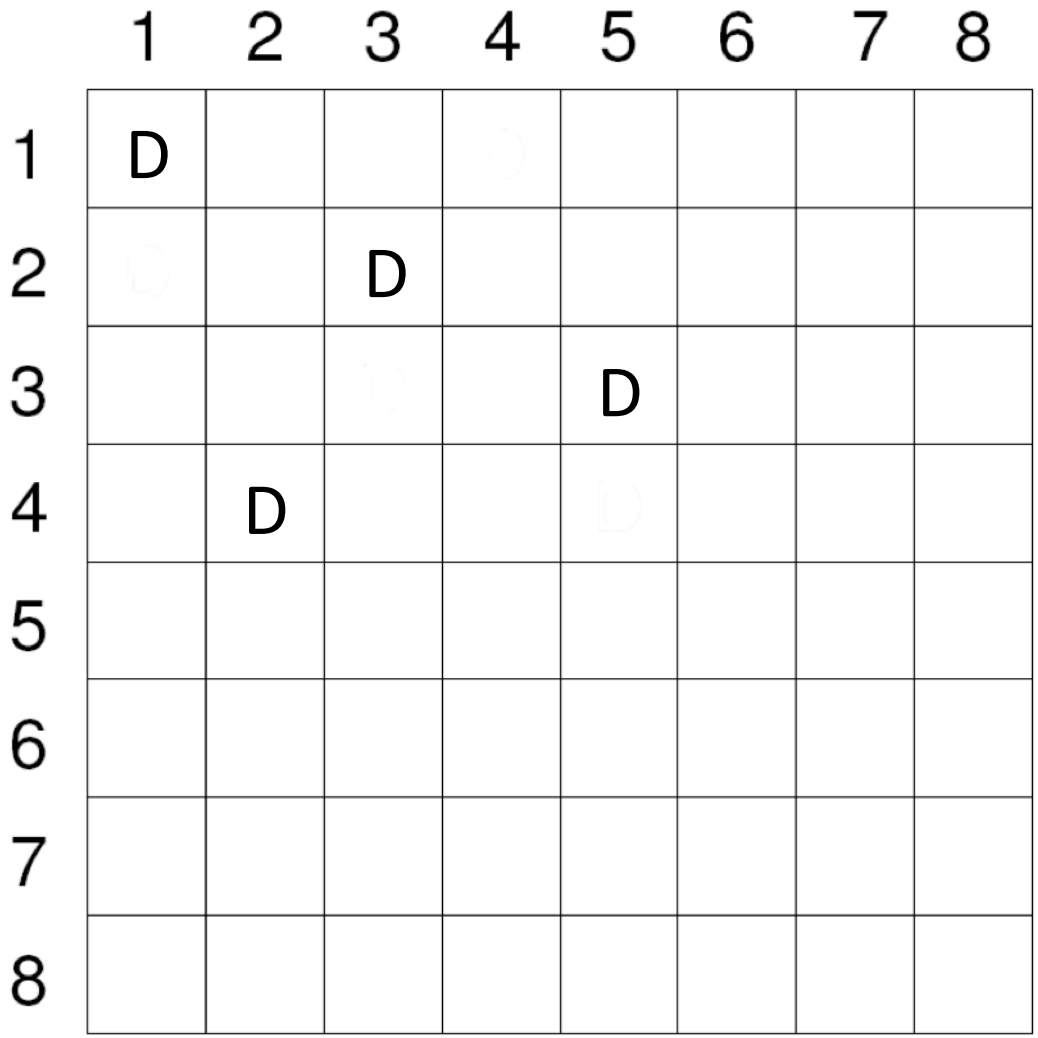
Beispiel



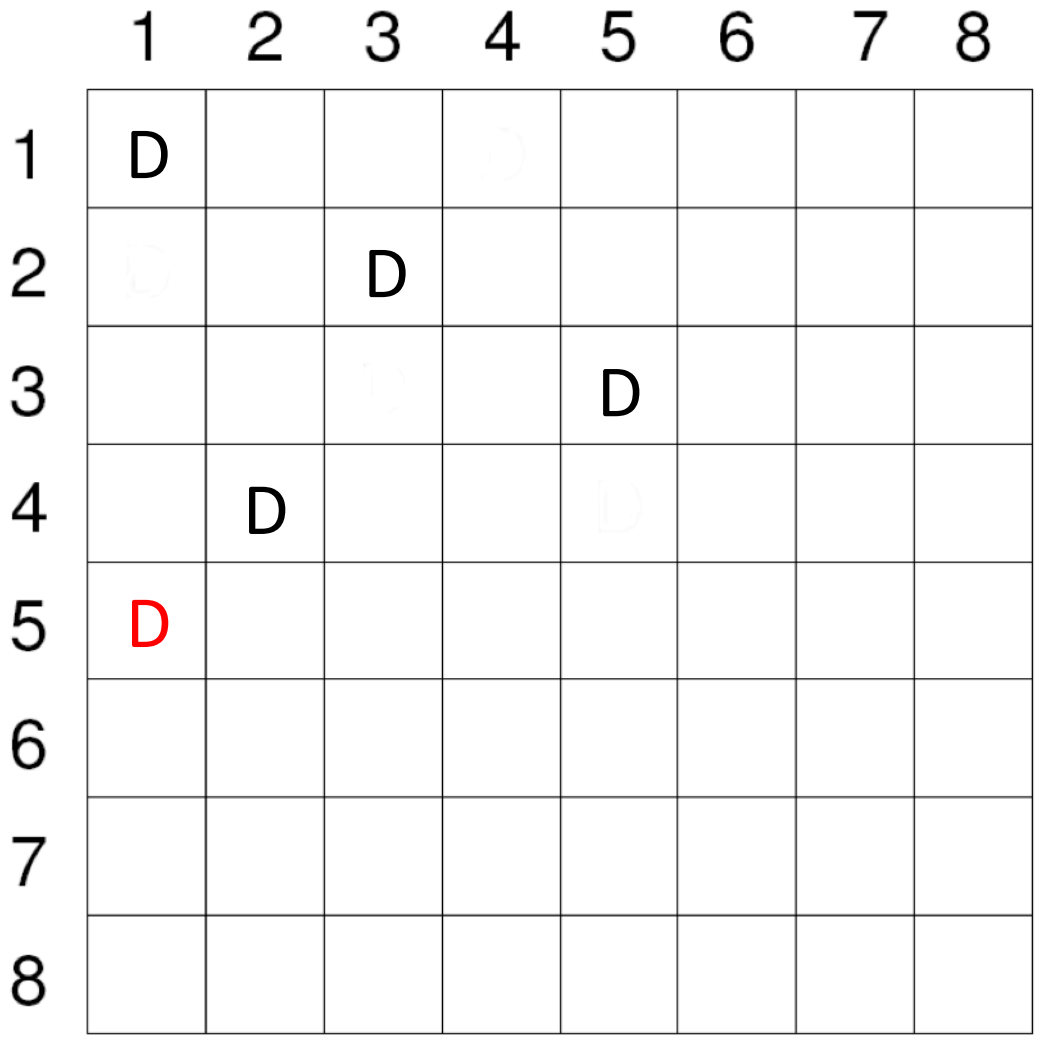
Beispiel



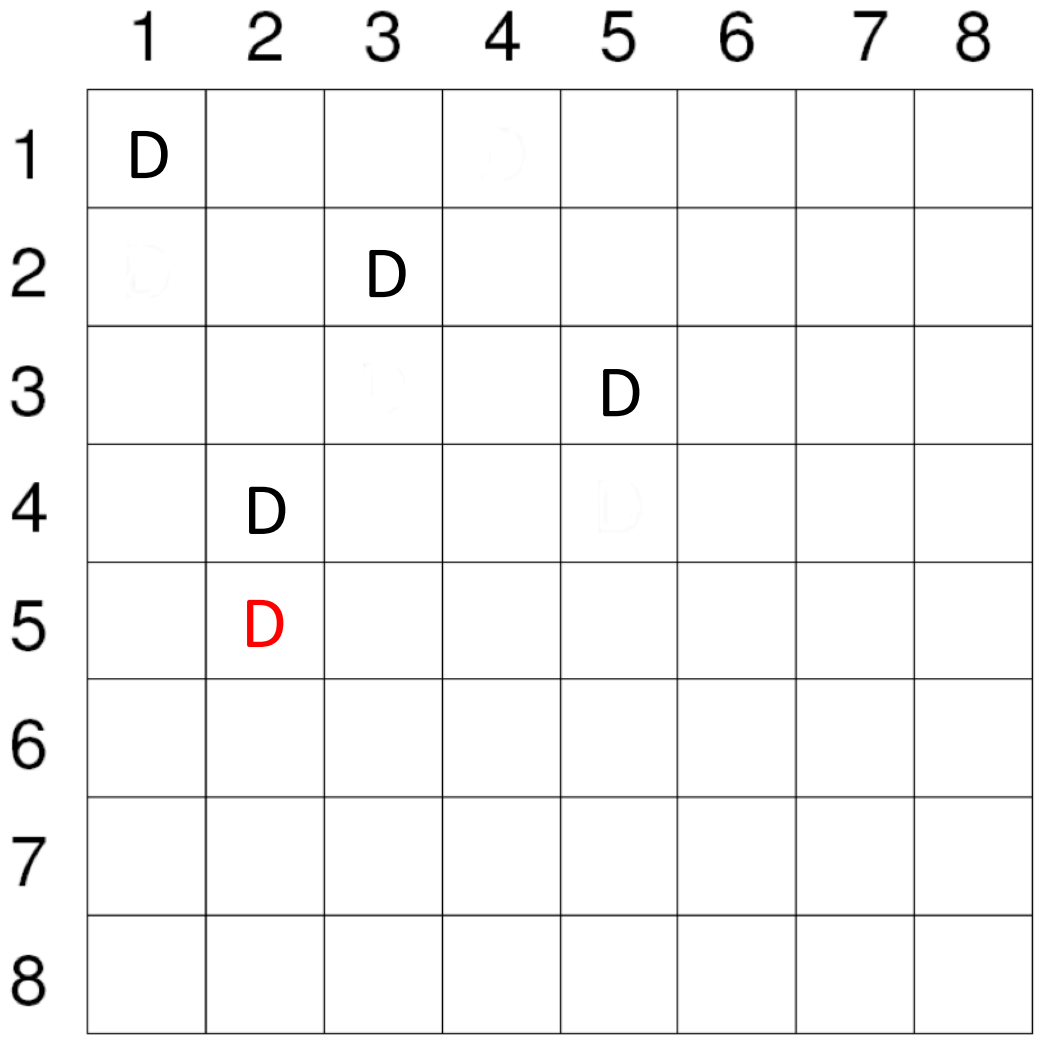
Beispiel



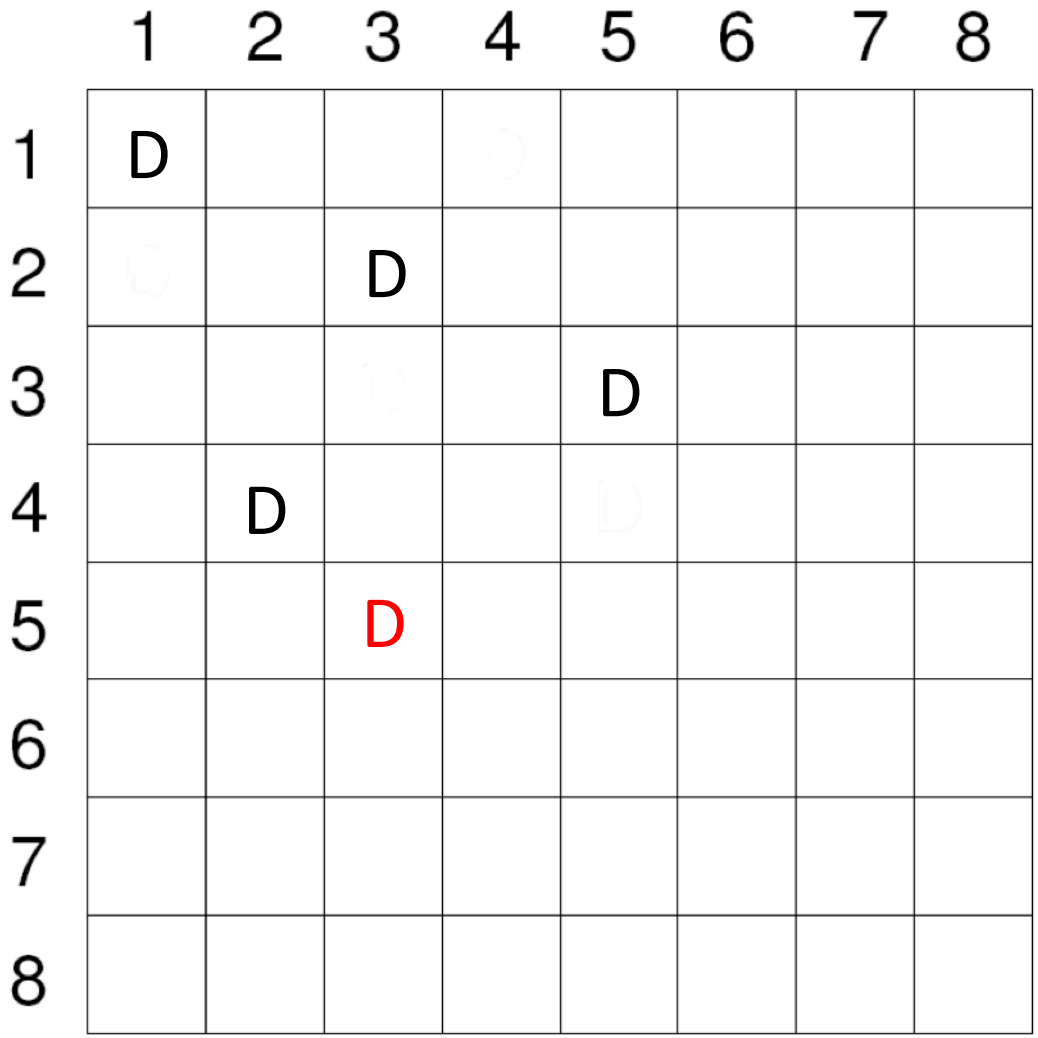
Beispiel



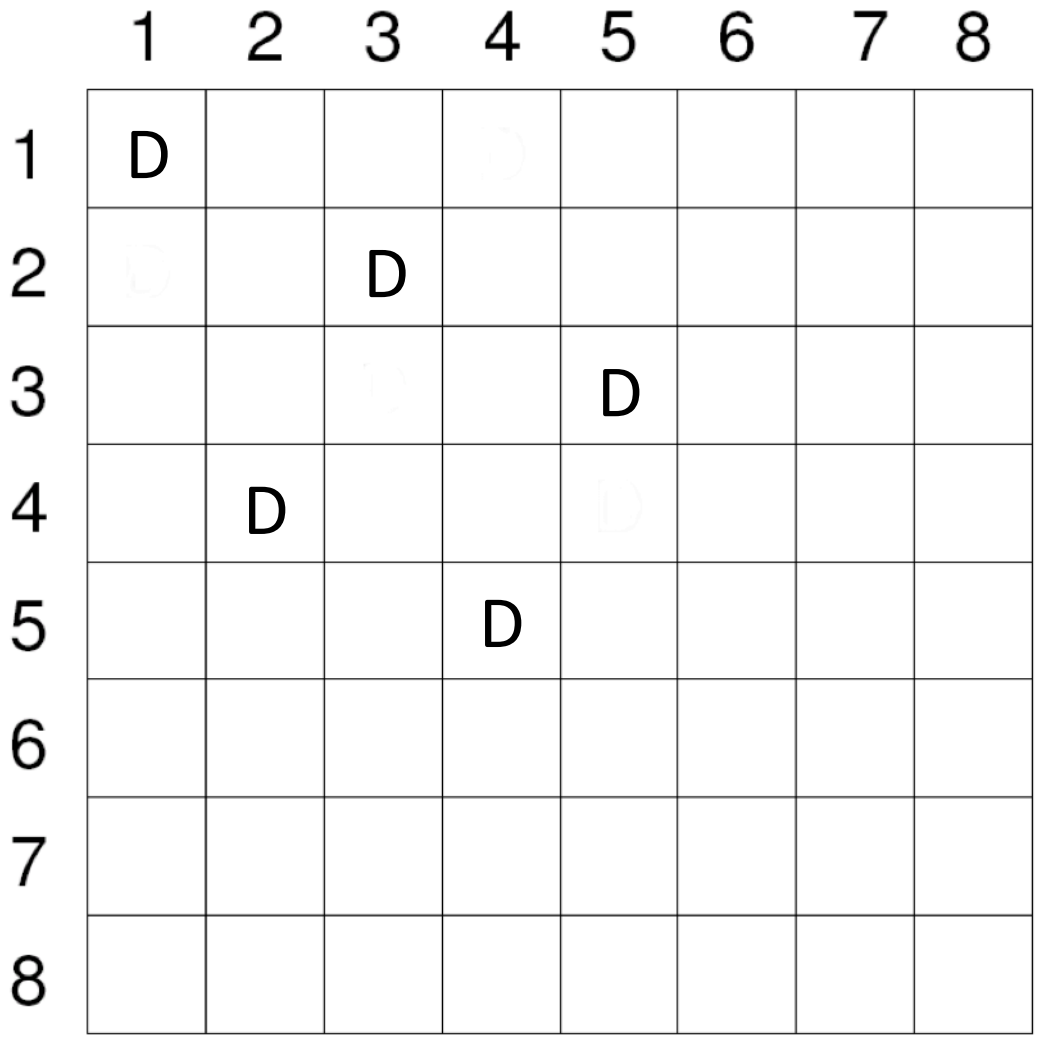
Beispiel



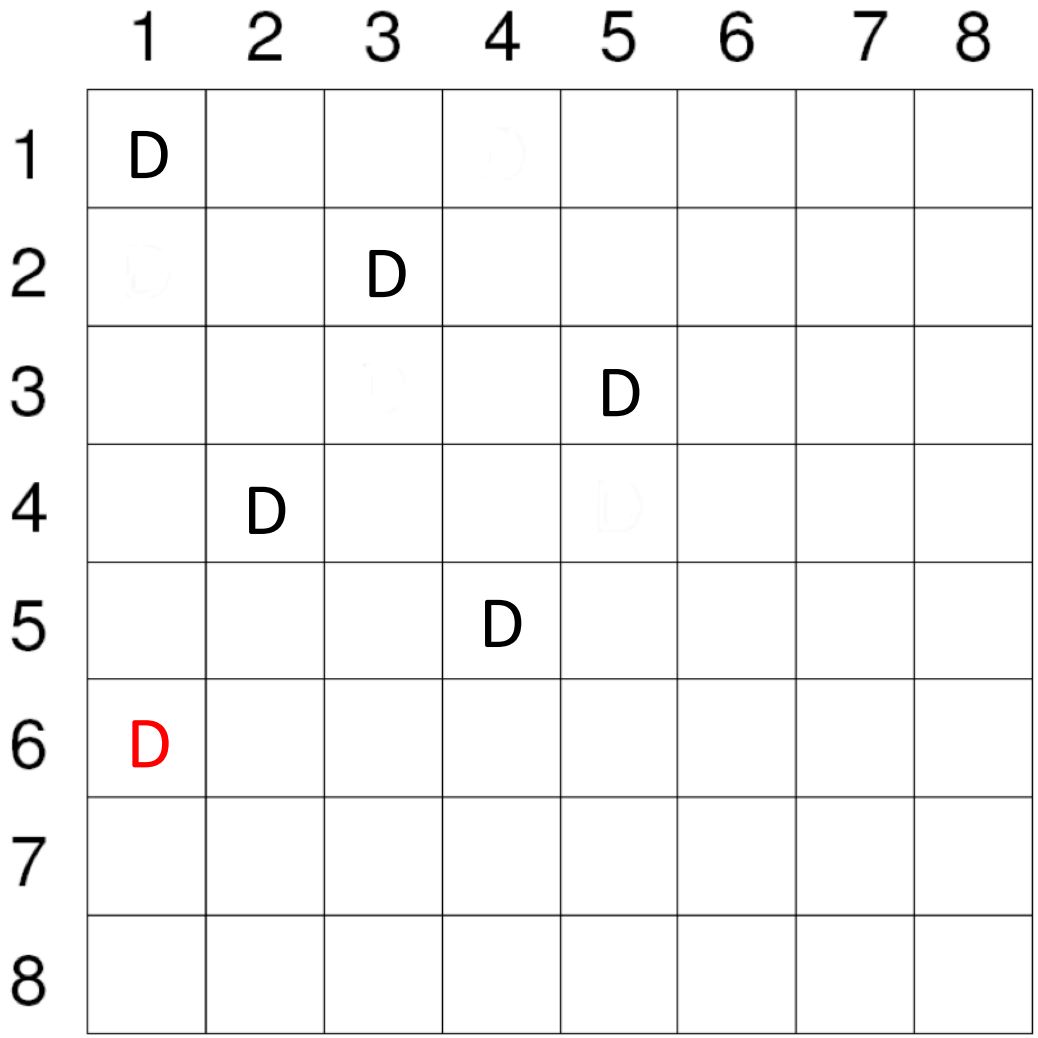
Beispiel



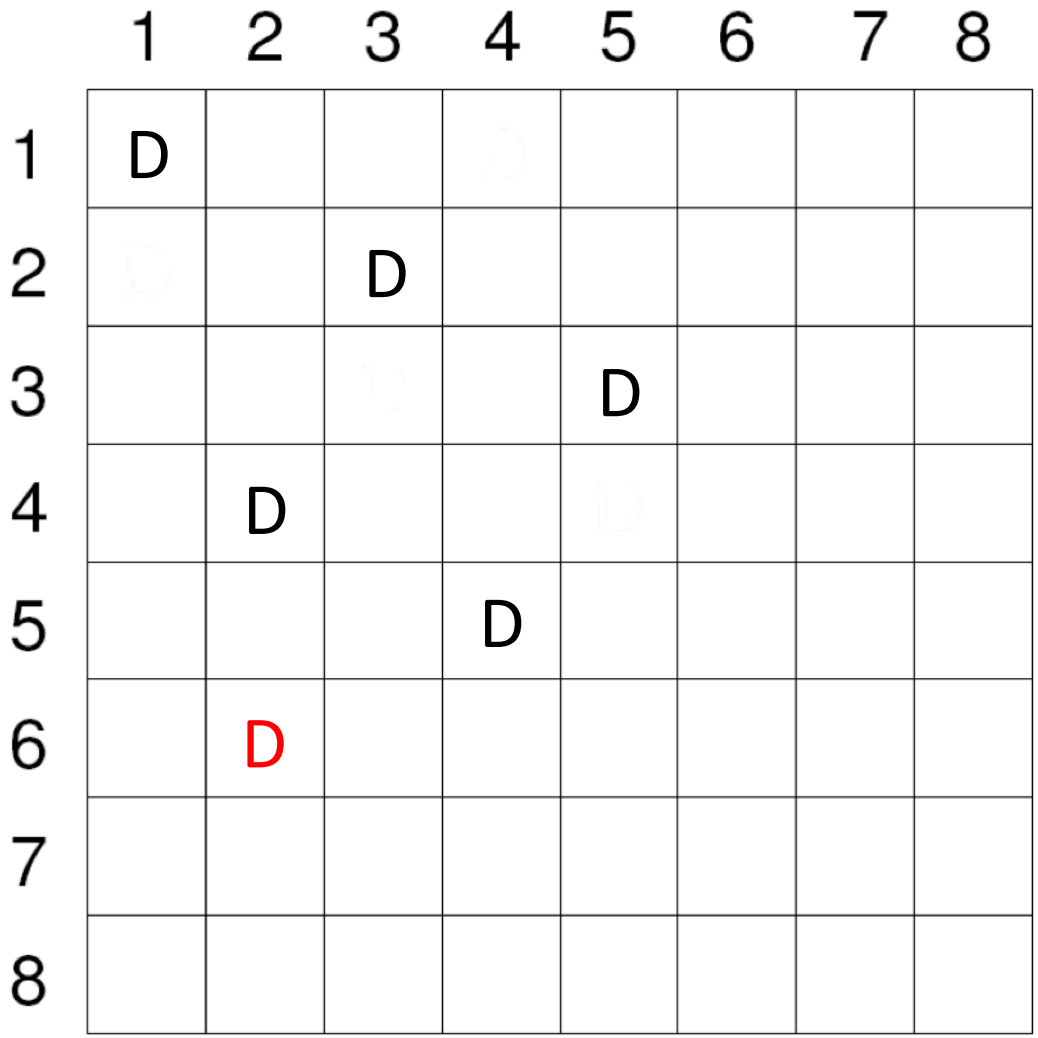
Beispiel



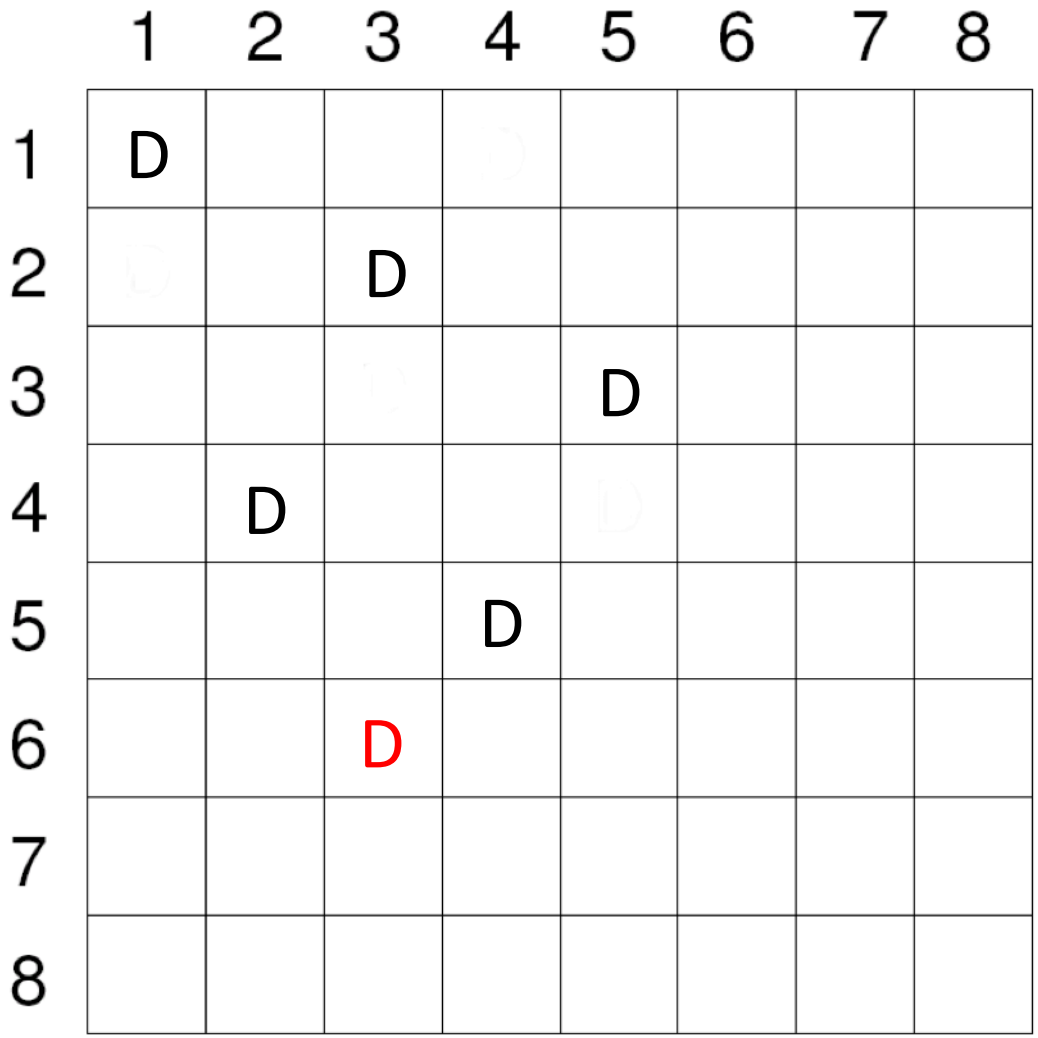
Beispiel



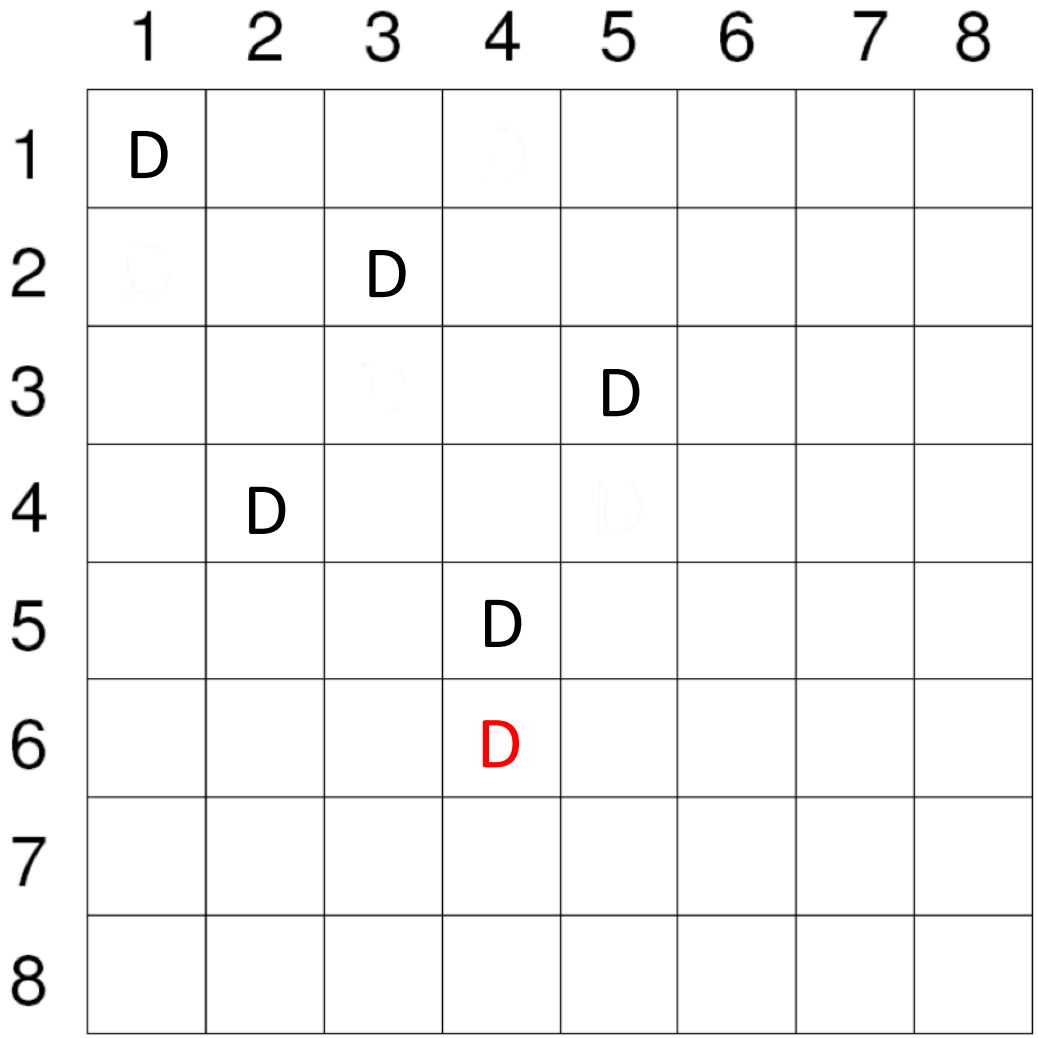
Beispiel



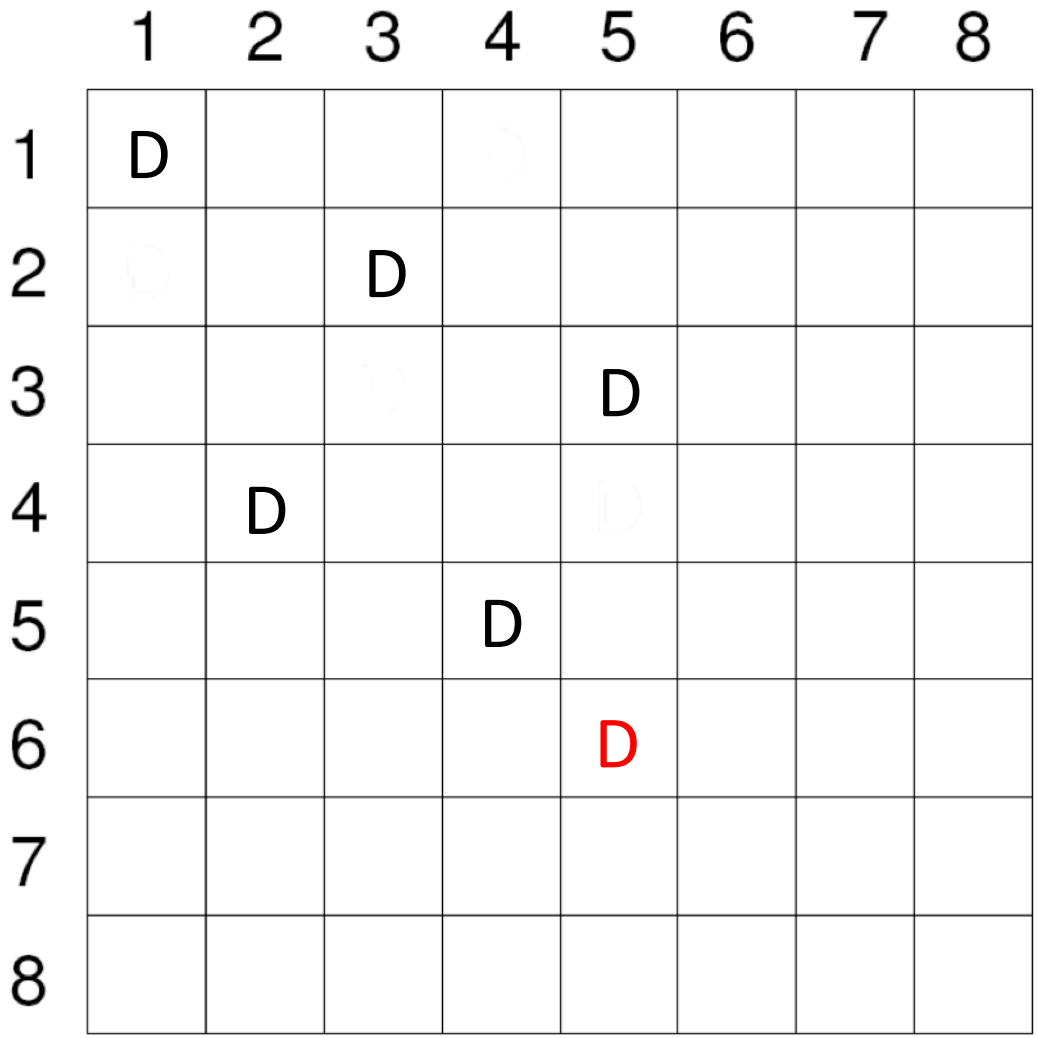
Beispiel



Beispiel



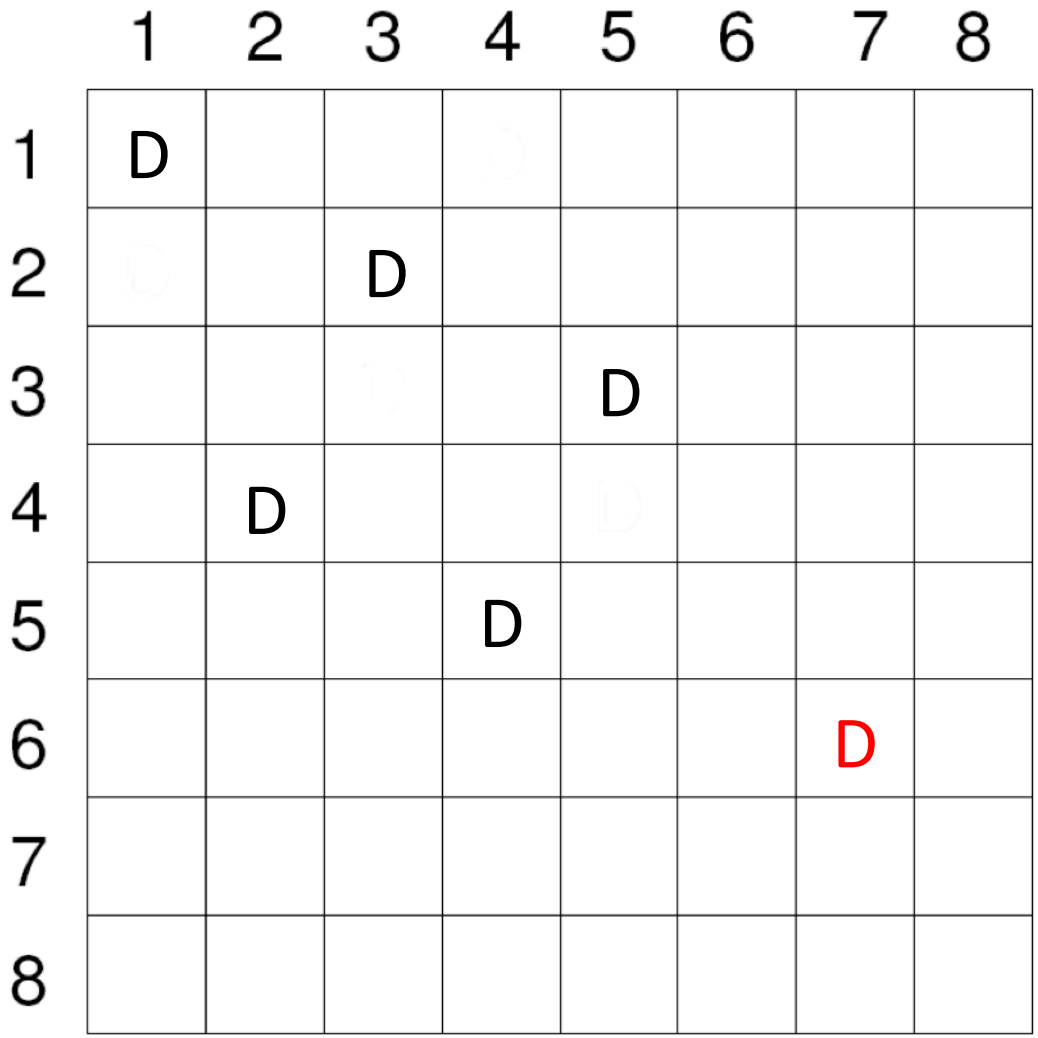
Beispiel



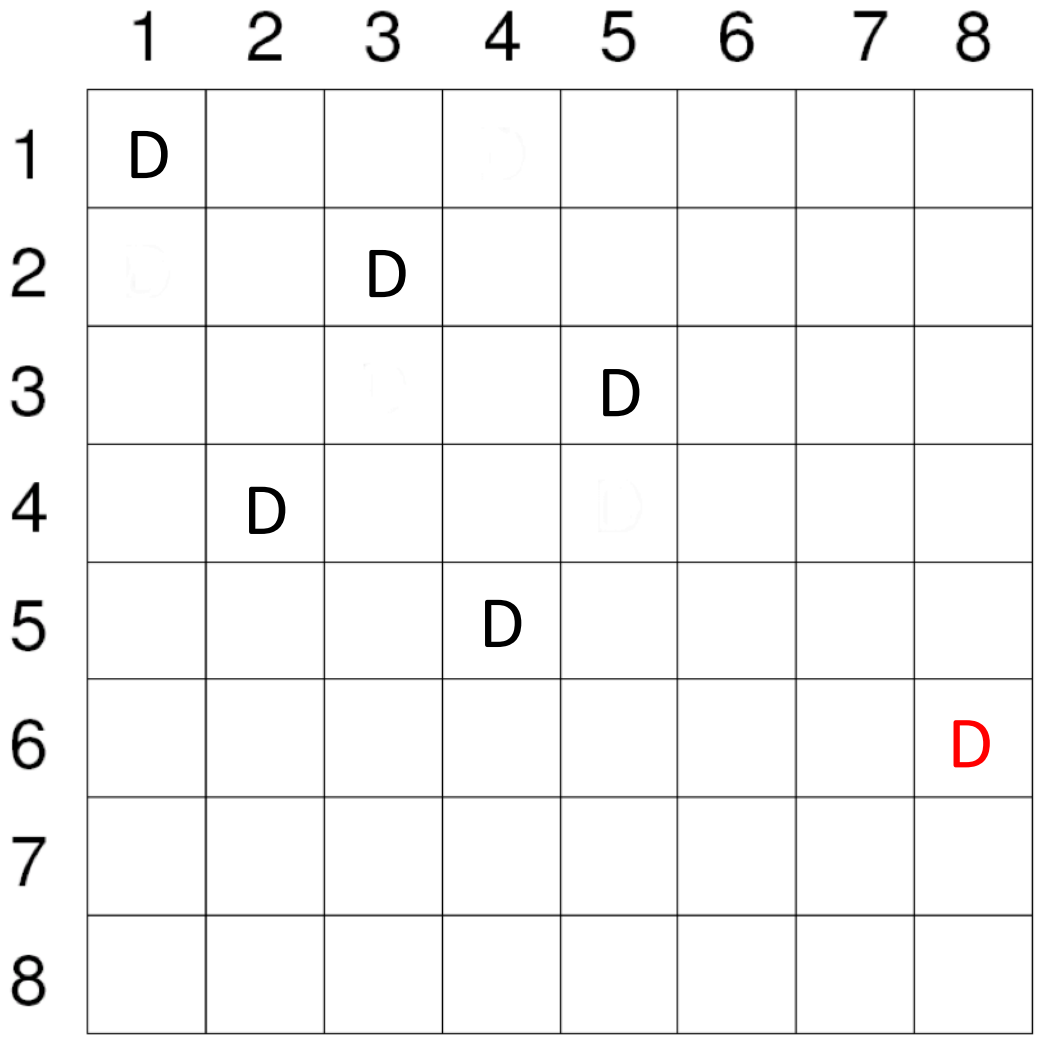
Beispiel

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

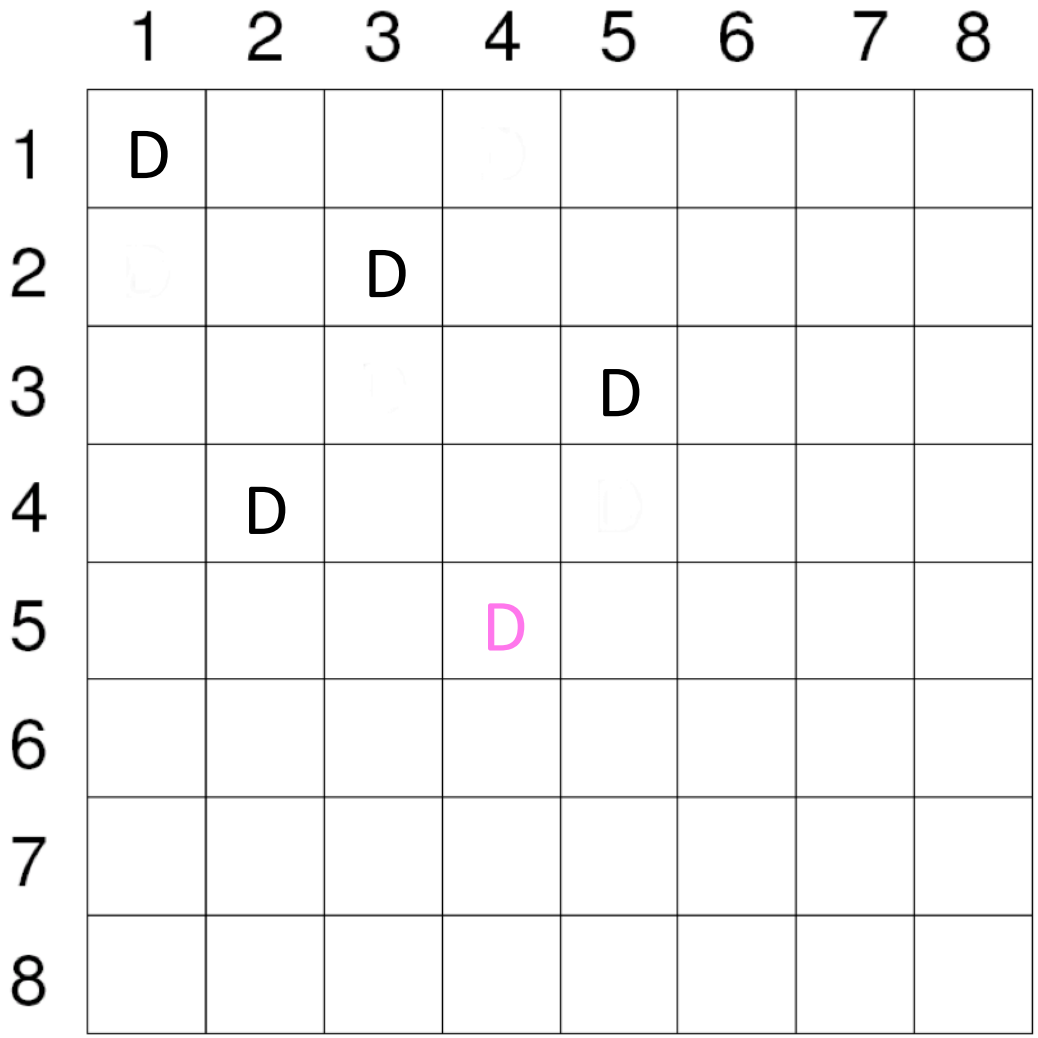
Beispiel



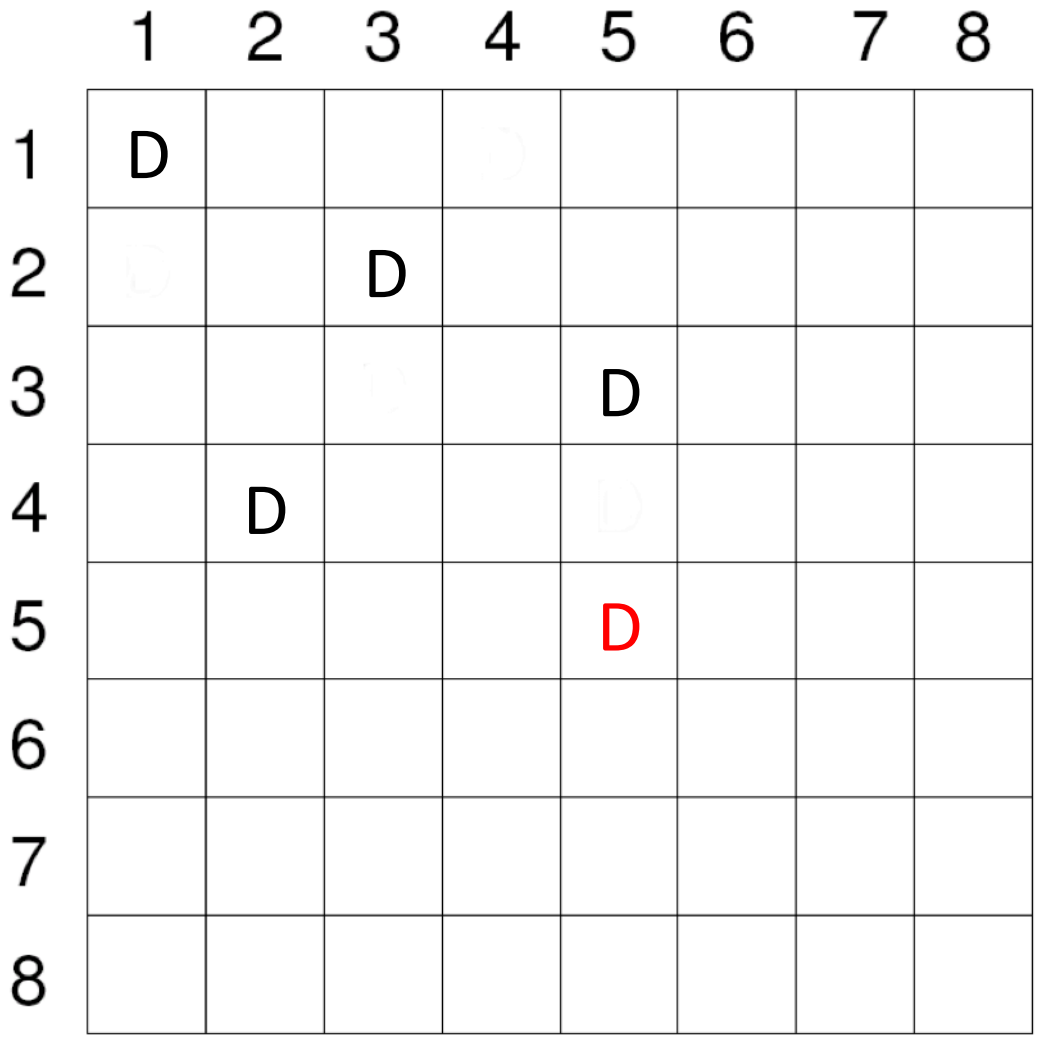
Beispiel



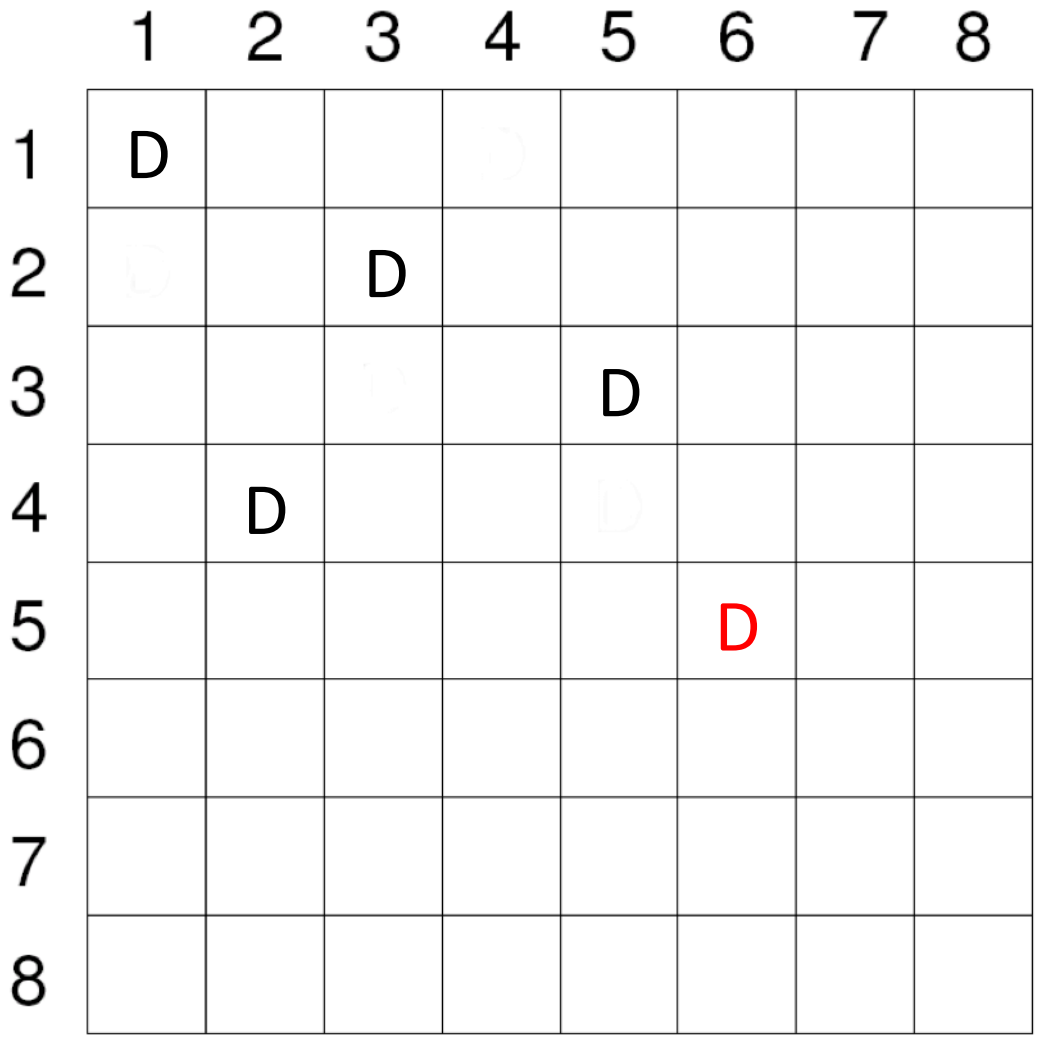
Beispiel



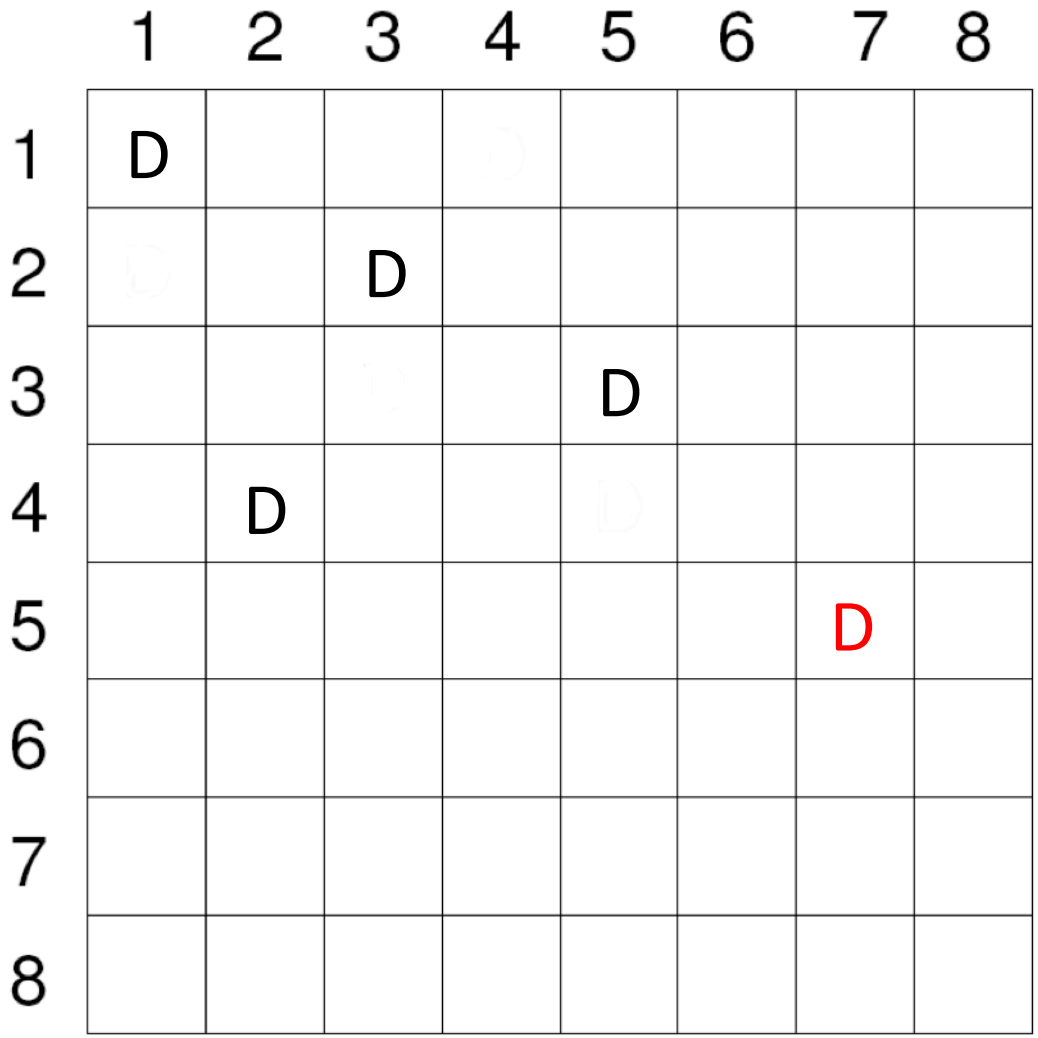
Beispiel



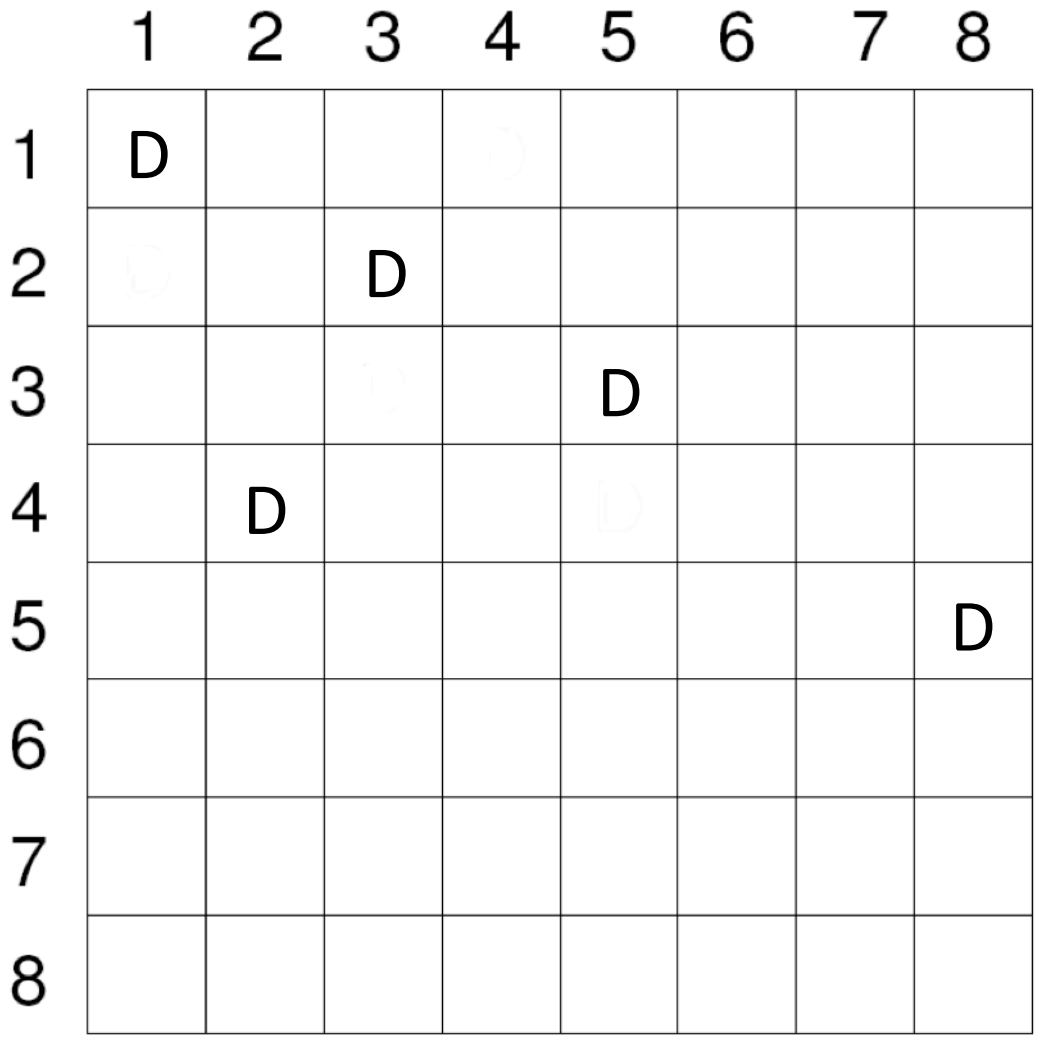
Beispiel



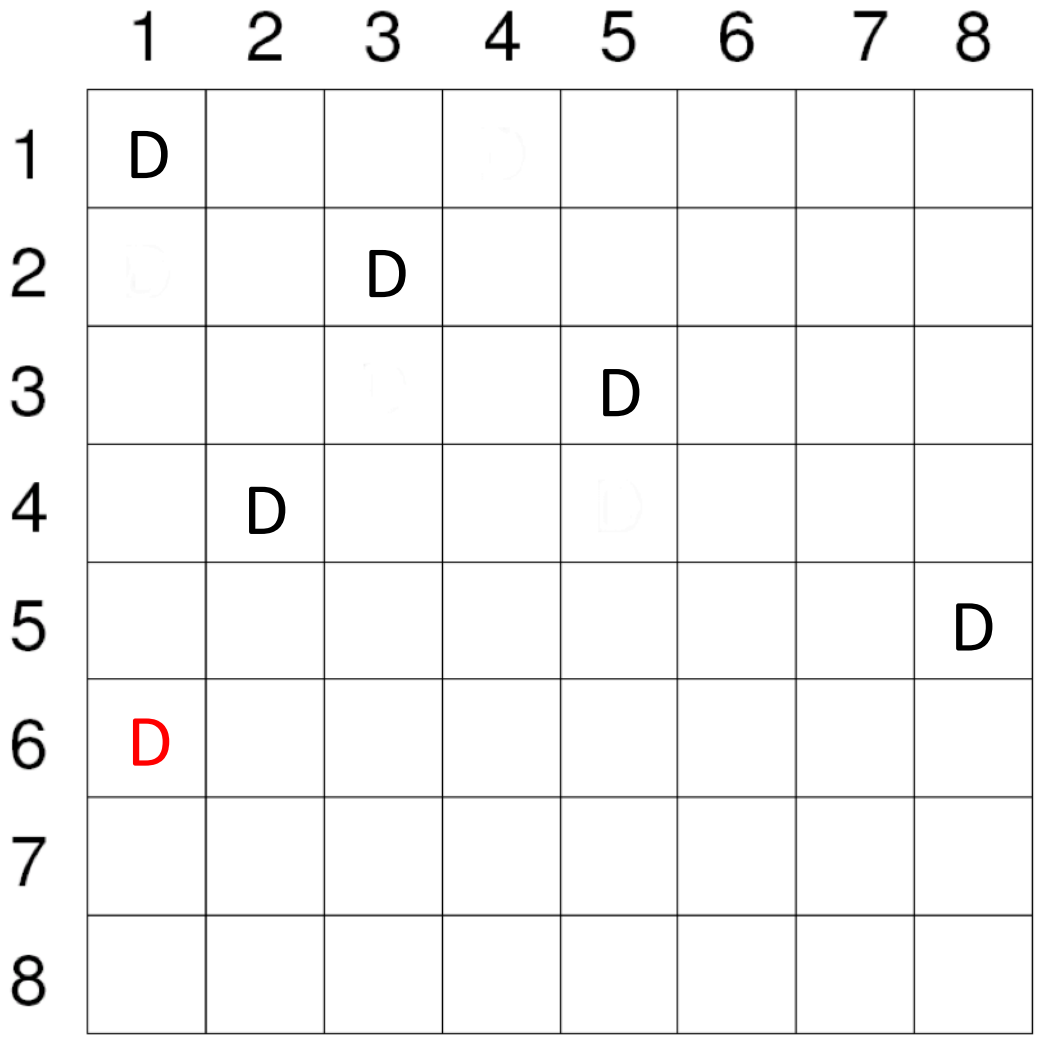
Beispiel



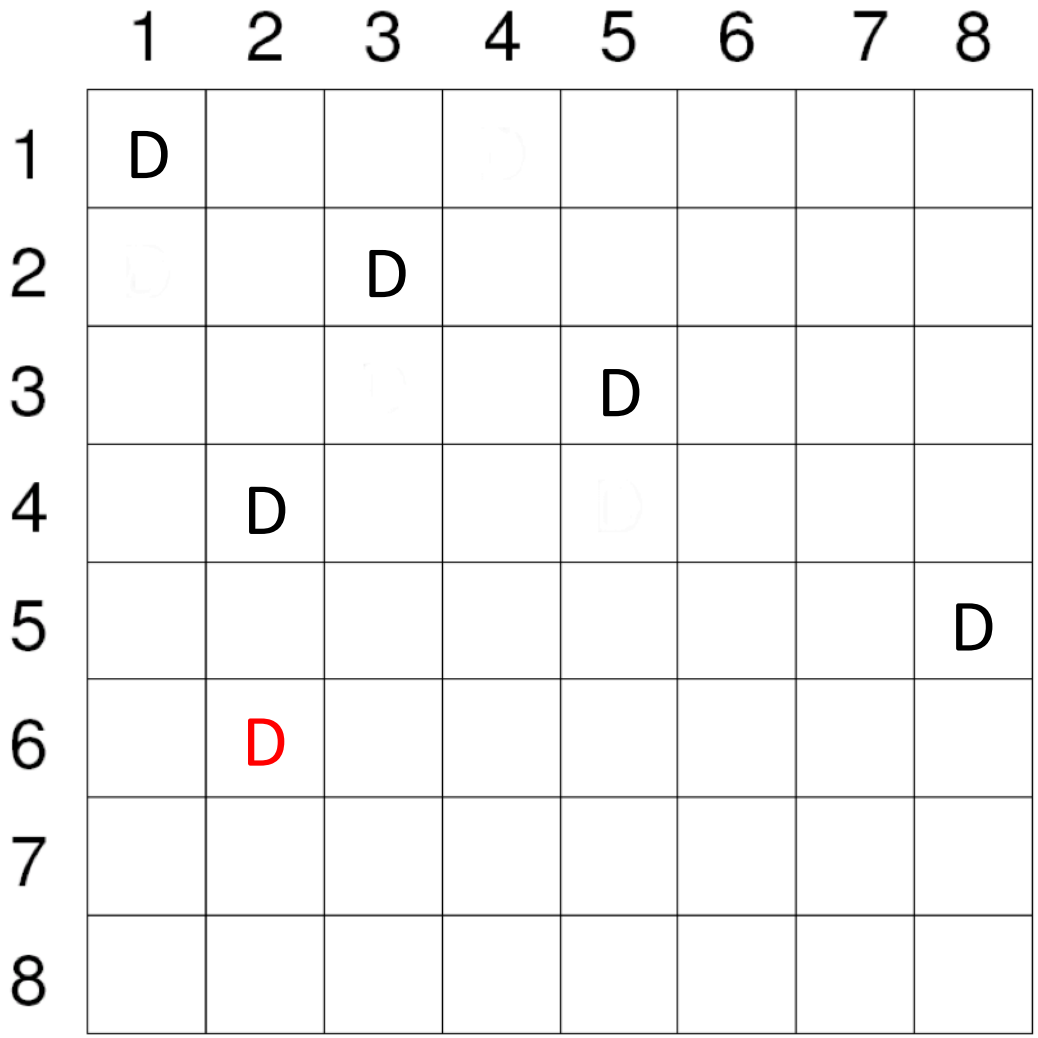
Beispiel



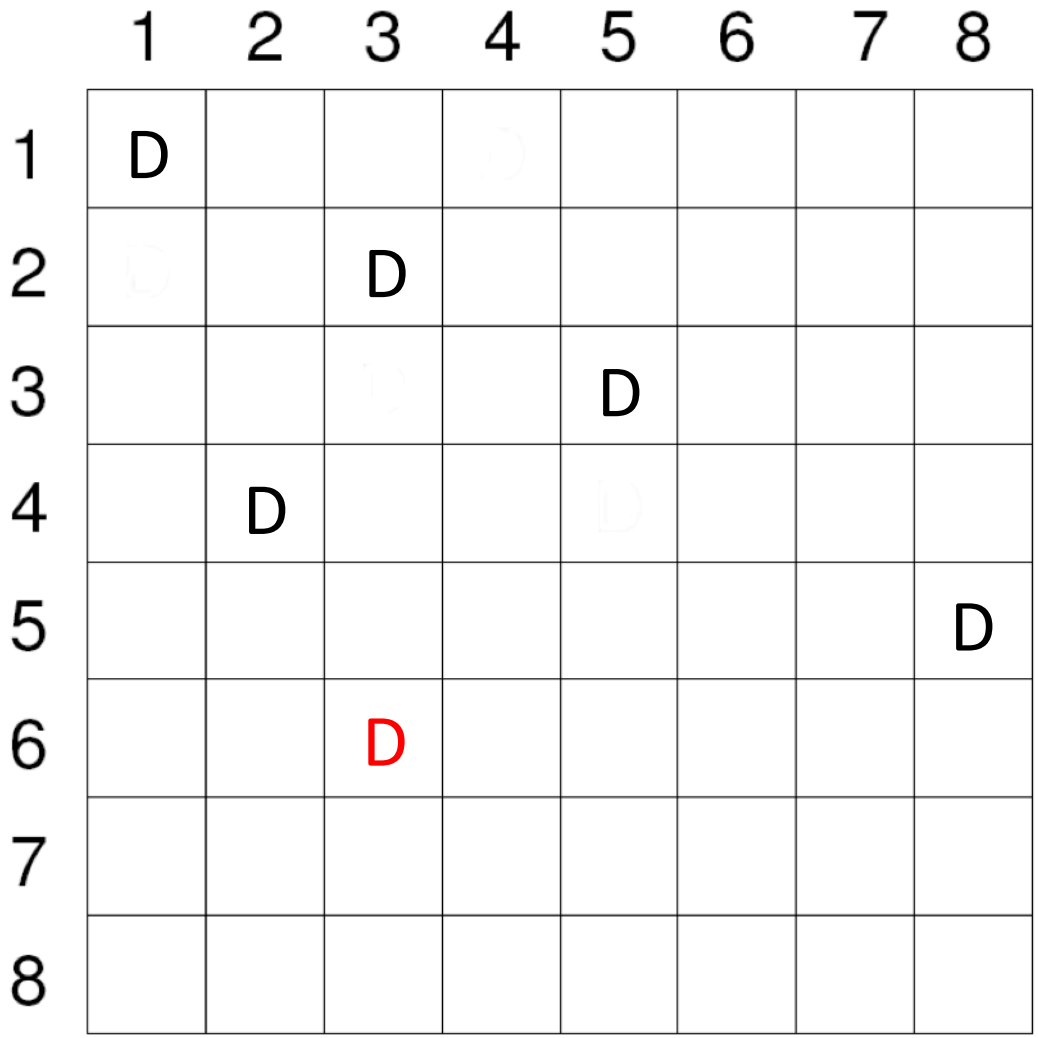
Beispiel



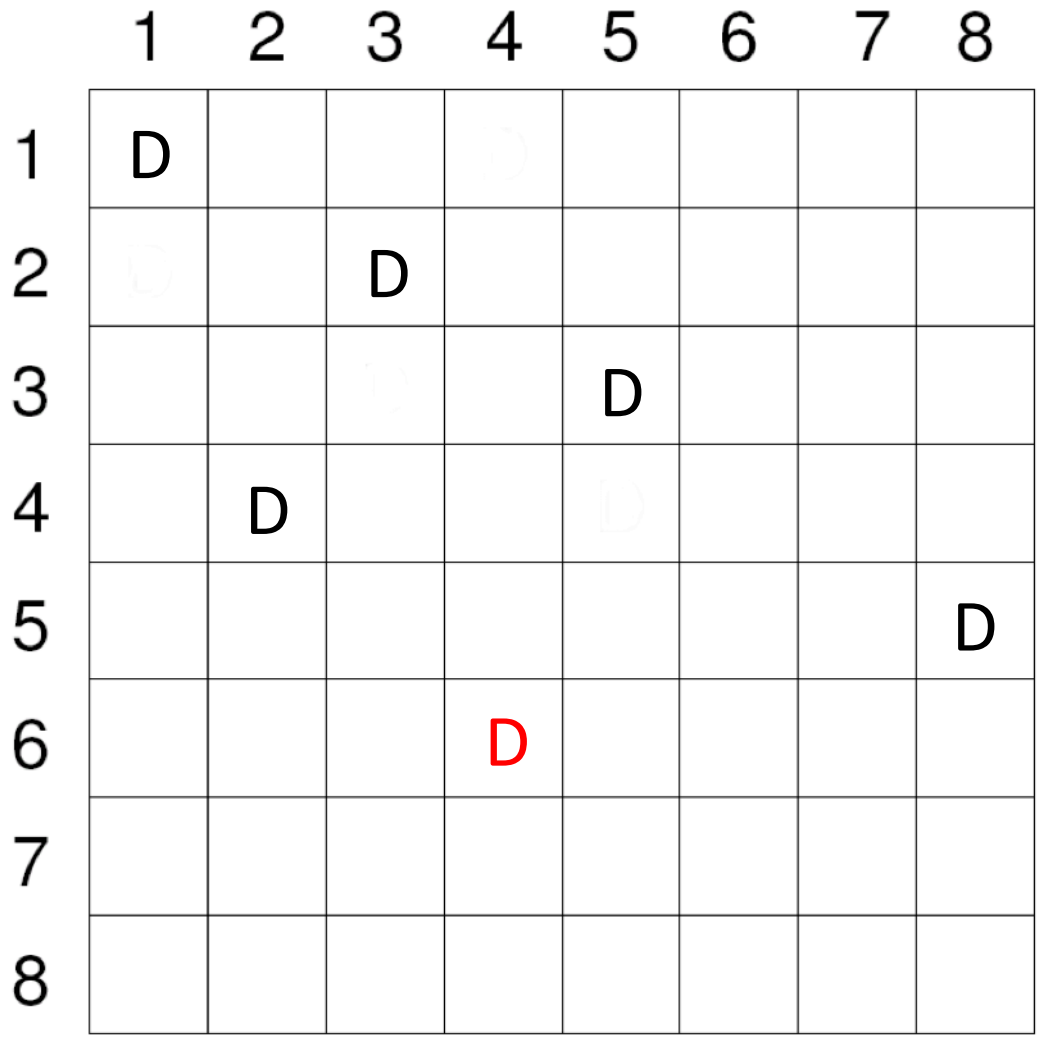
Beispiel



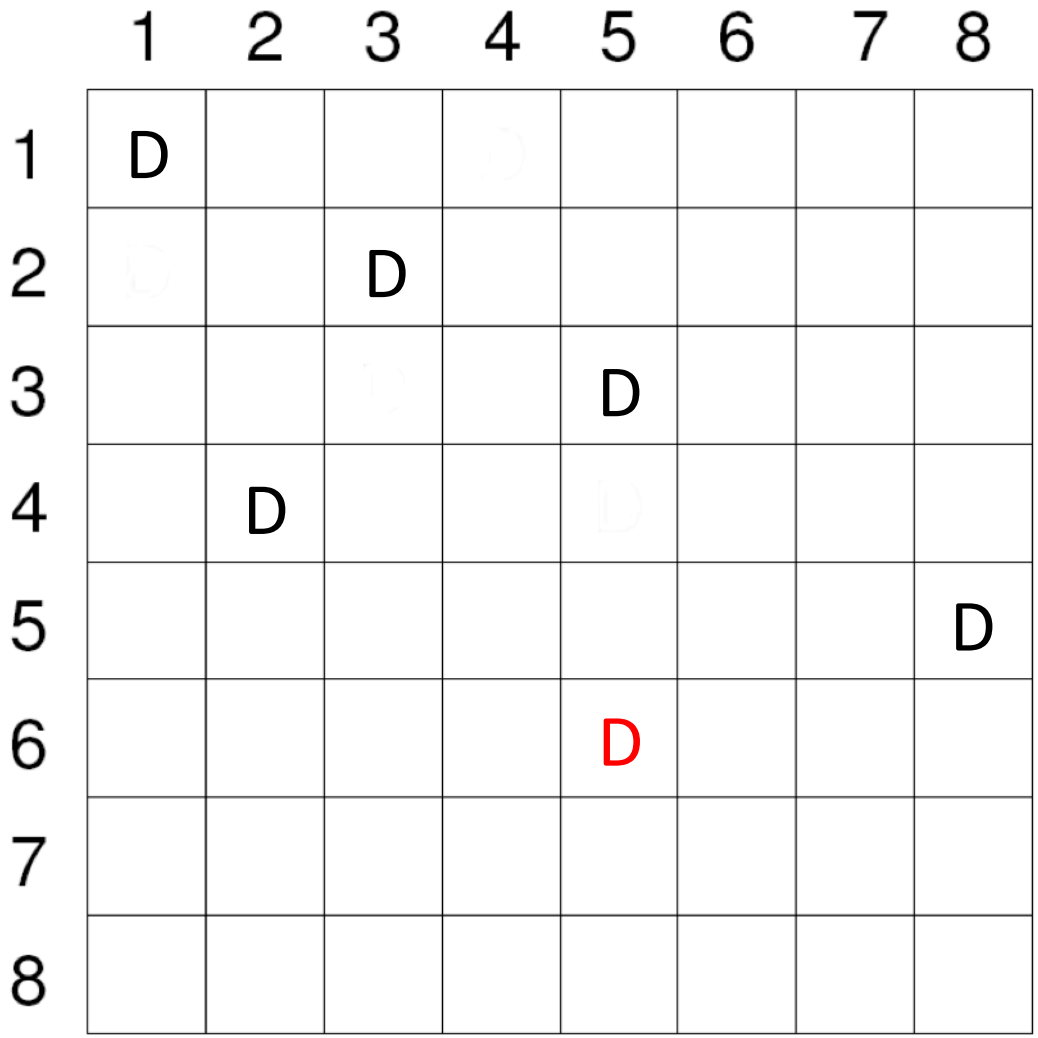
Beispiel



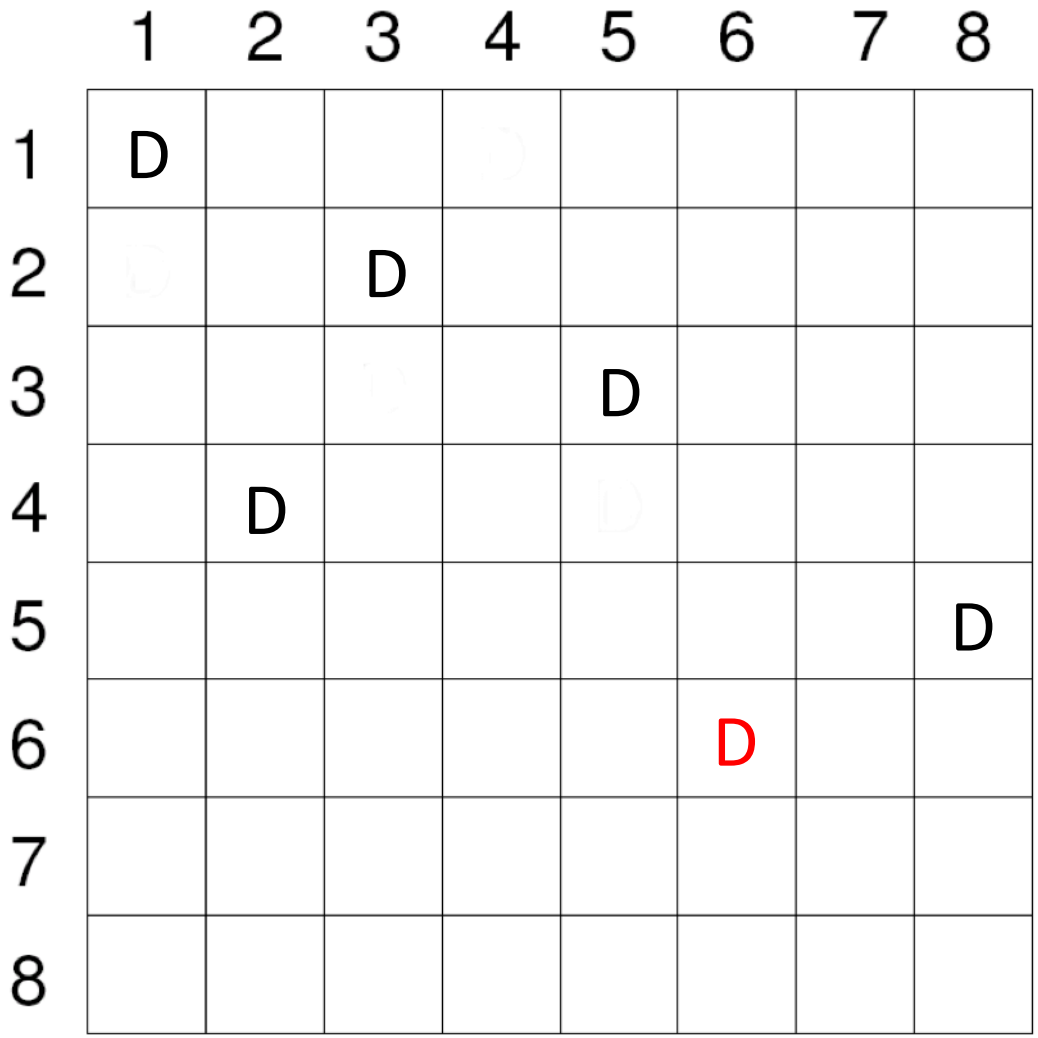
Beispiel



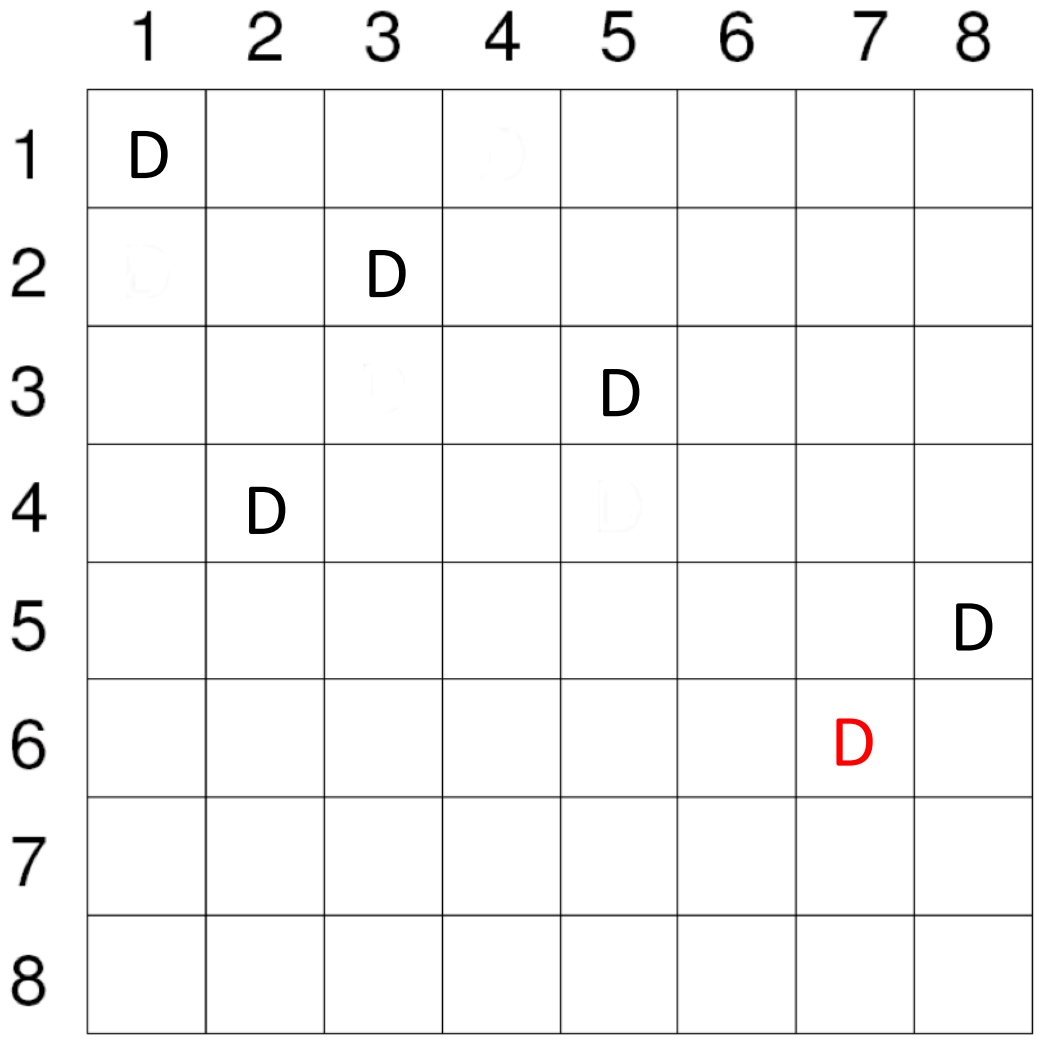
Beispiel



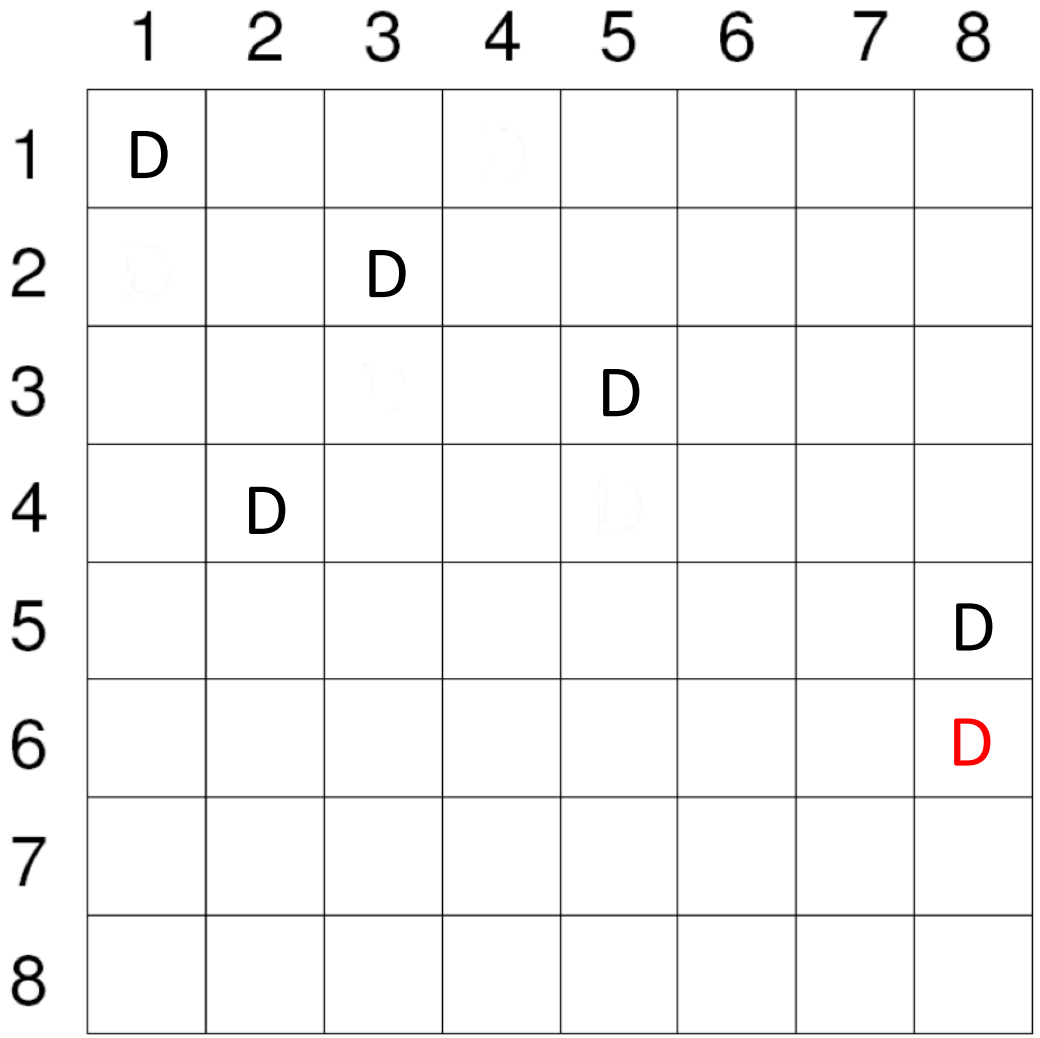
Beispiel



Beispiel



Beispiel



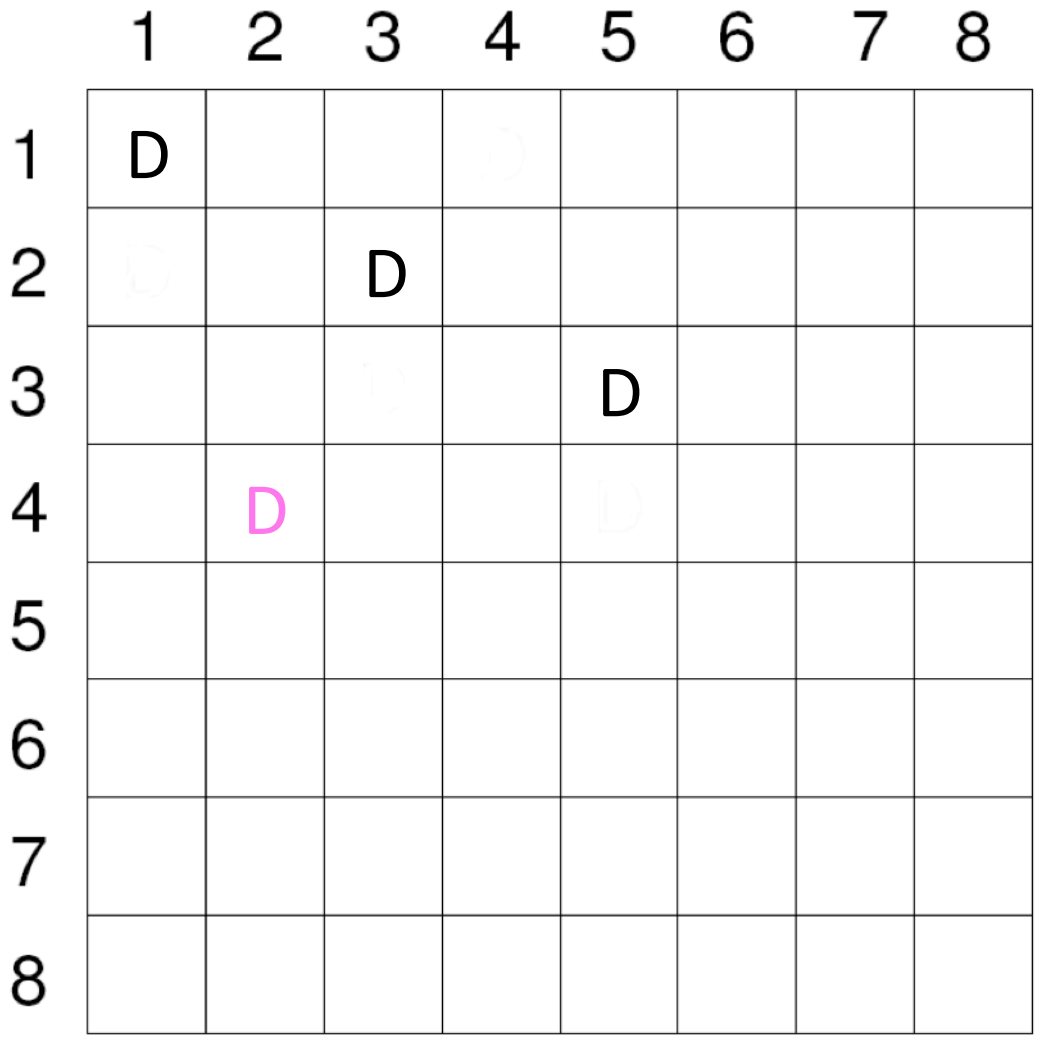
➔Backtrack

Beispiel

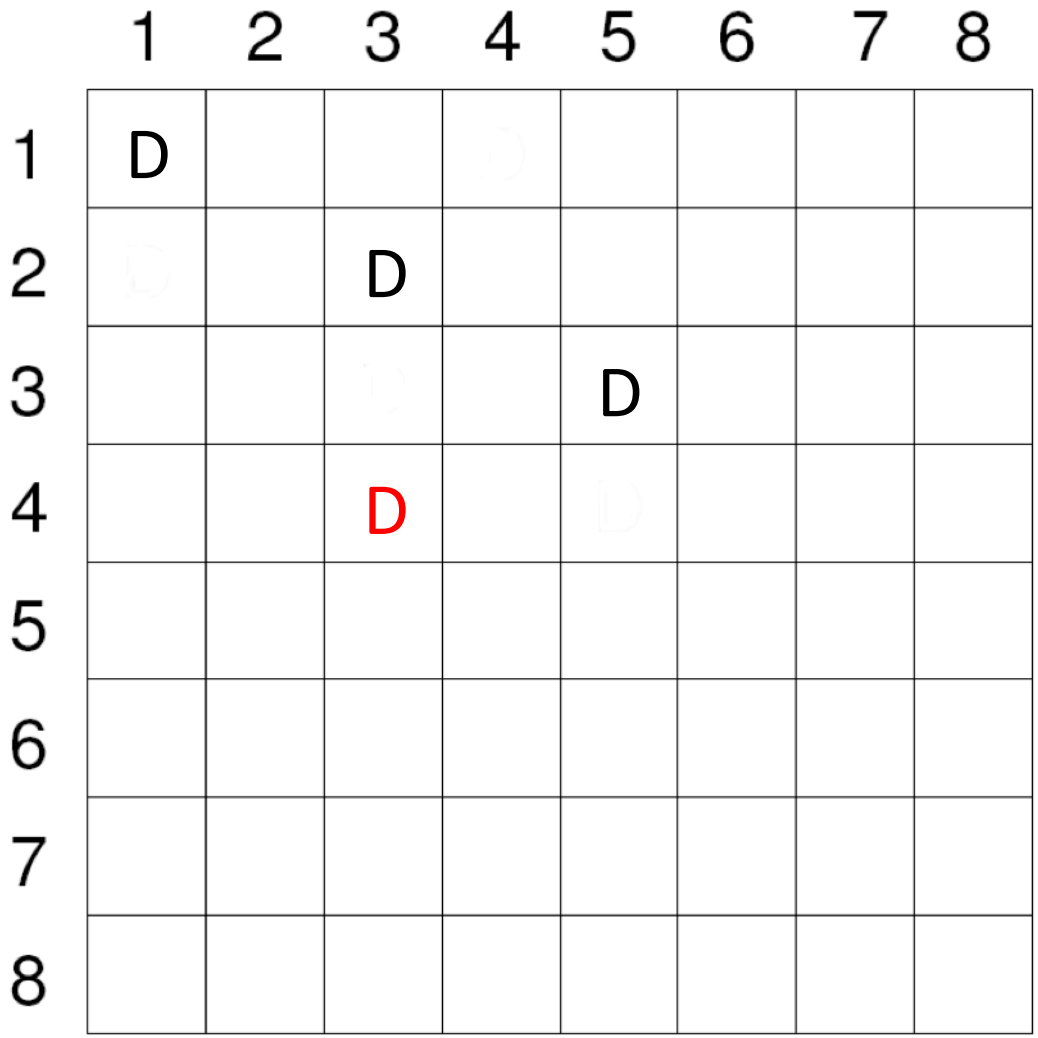
	1	2	3	4	5	6	7	8
1	D							
2			D					
3					D			
4		D						
5								D
6								
7								
8								

➔Backtrack

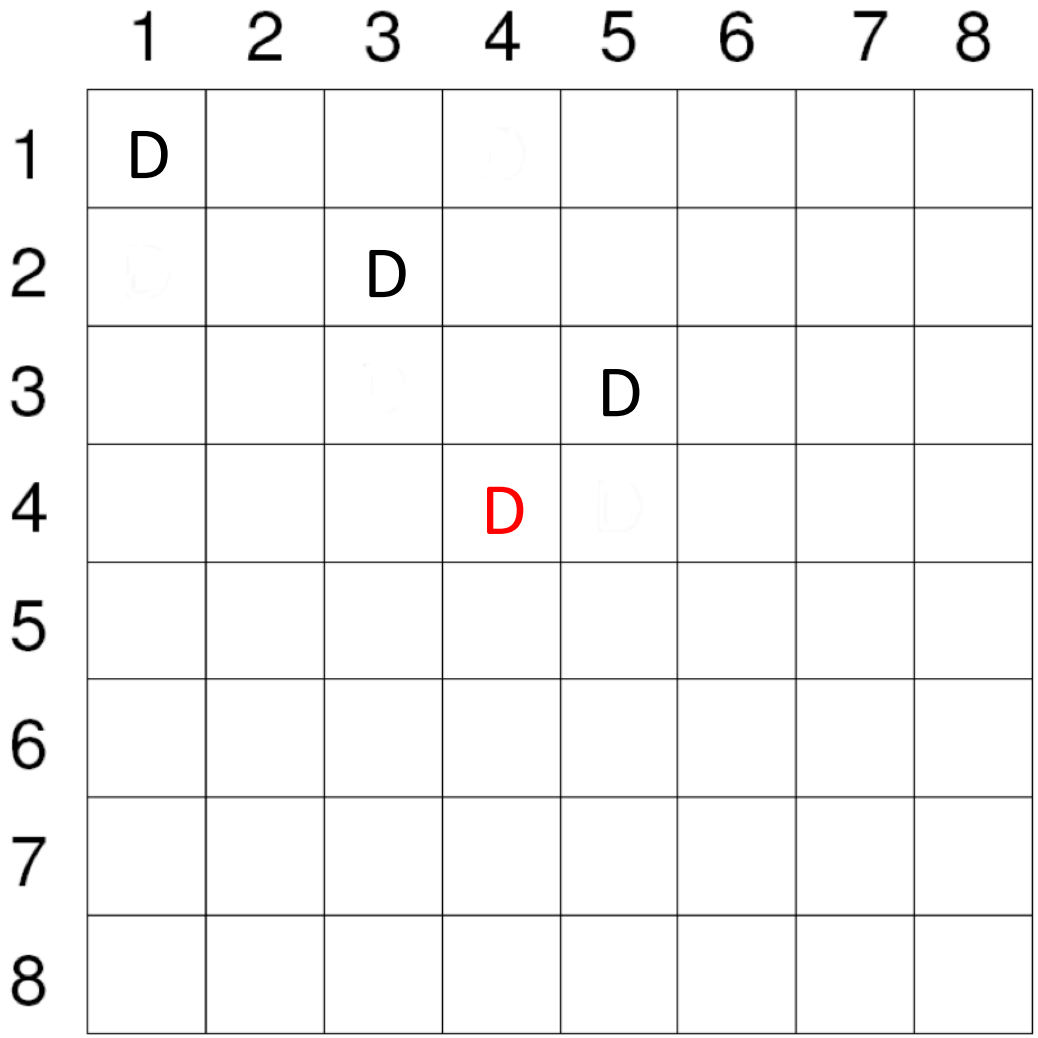
Beispiel



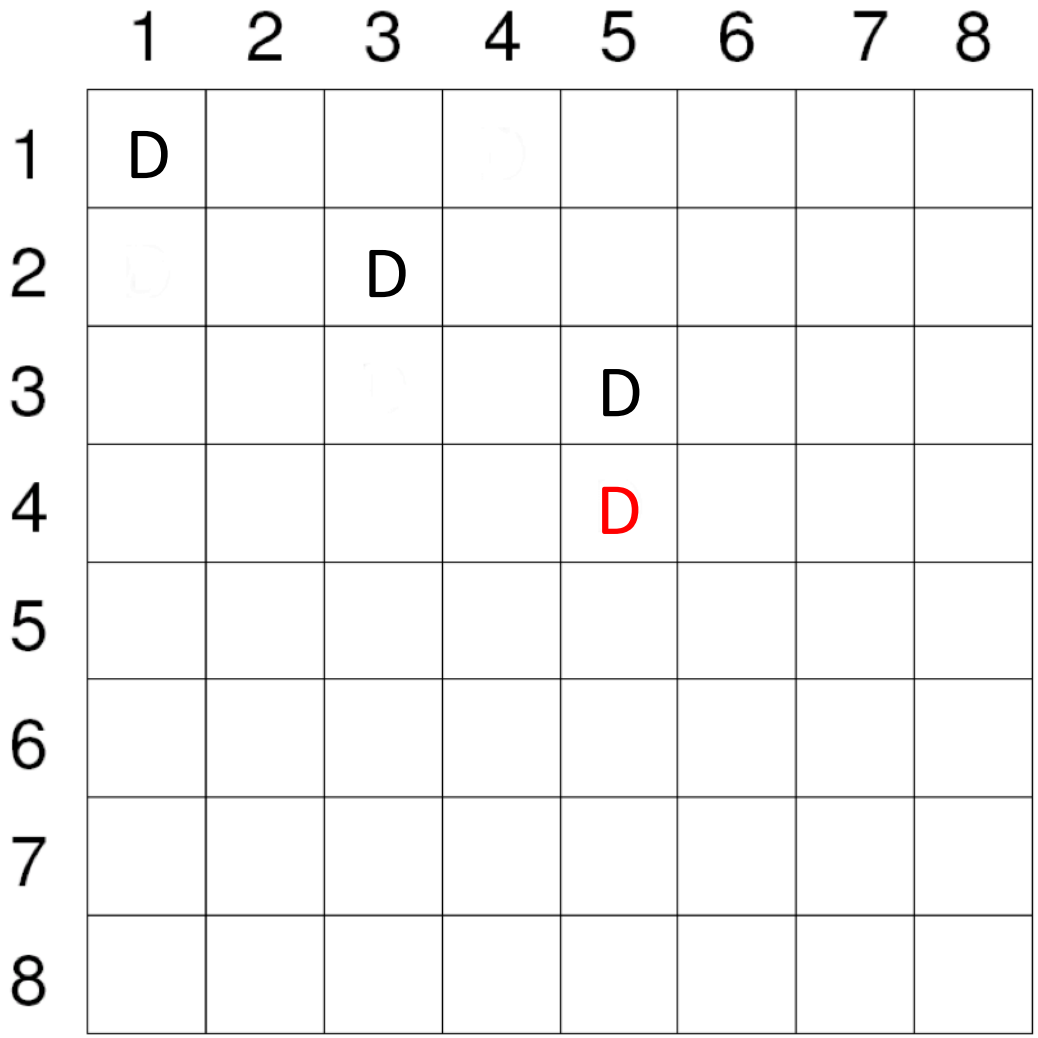
Beispiel



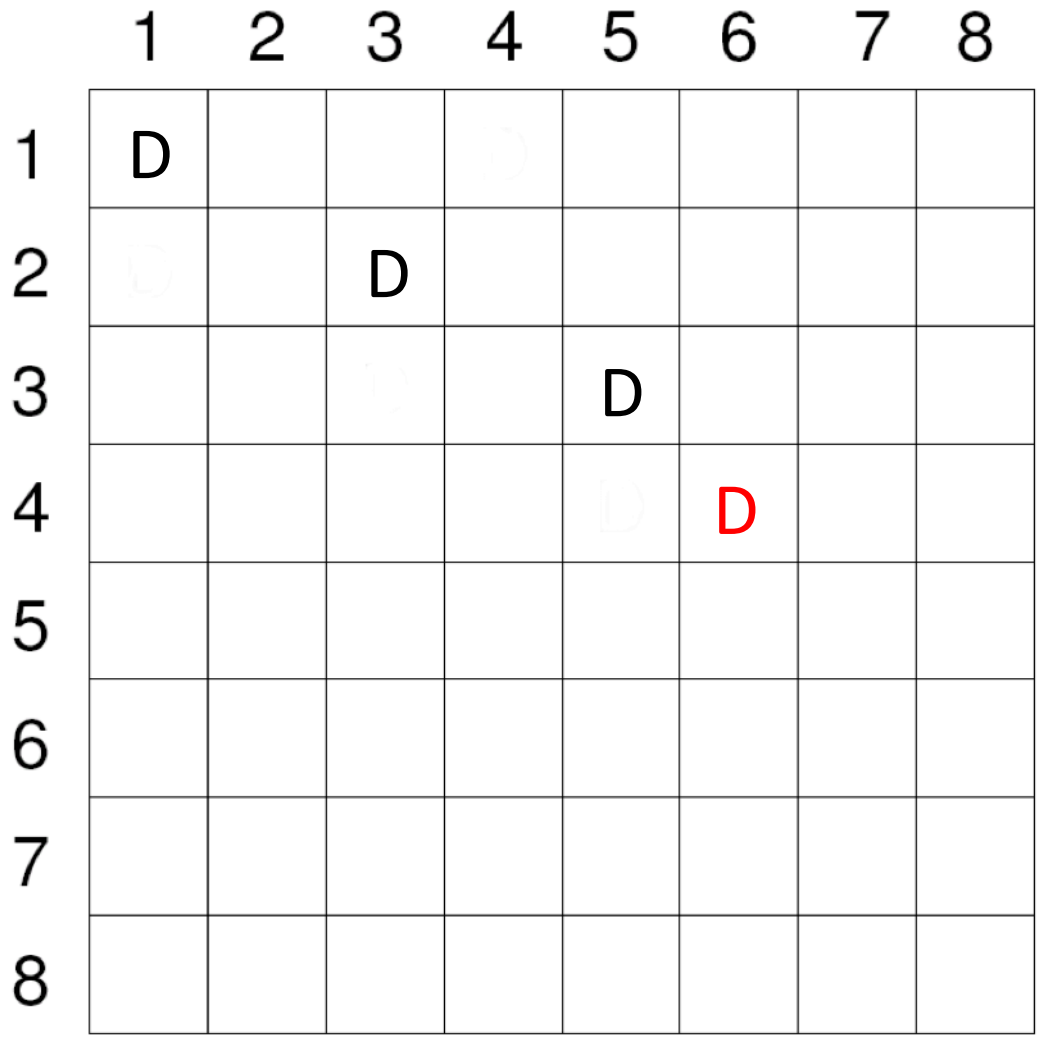
Beispiel



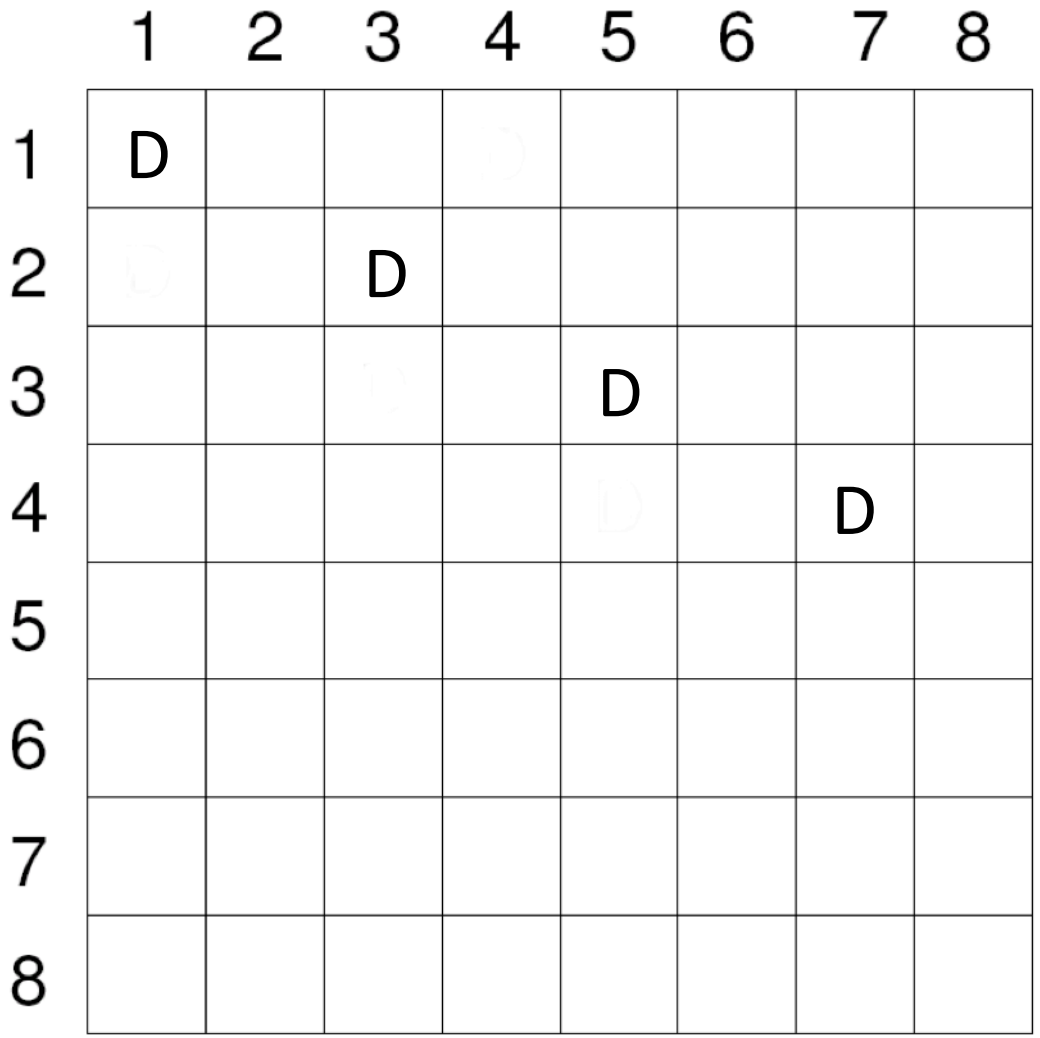
Beispiel



Beispiel



Beispiel



usw.

Algorithmus in Java (für allgemeinen Fall)

```
public boolean isValid(int[] board, int current, int place){
      for(int i = 0; i < current-1; i++){
            if(board[i] == place) return false;
            if(place+current == board[i] + (i+1)) return false;
            if(place-current == board[i] - (i+1)) return false;
      }
      return true;
}

public int[] placeQueen(int[] board, int current){
      int[] tmp;
      for(int i = 0; i < board.length; i++){
            if(isValid(board, current, i)){
                  board[current-1] = i;
                  if(current == board.length) return board;
                  else{
                        tmp = placeQueen(board,current+1);
                        if(tmp != null)
                              return tmp;
                  }
            }
      }
      return null;
}
```

Aufruf durch

```
int[] result = placeQueen(new int[8], 1);
```

Analyse

Theorem

Der Algorithmus `placeQueen` terminiert nach endlich vielen Schritten (bei positiver Anzahl von Feldern).

Beweis:

Die Methode `isValid` terminiert offensichtlich stets.

In `placeQueen` wird rekursiv `placeQueen` stets um einen um 1 erhöhten Parameter `current` aufgerufen; die **`for`**-Schleife hat auch stets eine konstante Zahl and Durchgängen. □

Analyse

Theorem

Für ein Feld der Größe n x n hat der Algorithmus `placeQueen` eine Laufzeit von $O(n^n)$.

Beweis:

Im schlimmsten Fall werden alle Konfigurationen betrachtet:

- n Positionen für eine einzelne Dame
- n Damen sind zu platzieren

-> $O(n^n)$ □

Anmerkungen:

- Die tatsächliche Laufzeit ist weitaus geringer, da viele Konfigurationen schon früh als nicht-erweiterbar erkannt werden
- Dennoch ist die Laufzeit im schlimmsten Fall exponentiell ($O(2^n)$)

Analyse

Theorem

Der Algorithmus `placeQueen` löst das n-Damenproblem.

Beweis:

Übung □

Problemklasse für Backtracking

- Lösung besteht aus Folge von Elementen
- An jeder Stelle der Folge gibt es nur eine endliche Anzahl an Möglichkeiten ein Element auszuwählen
- Überprüfung ob etwas eine Lösung ist, ist einfach
- Wird normalerweise verwendet, wenn ein direkter effizienter Algorithmus nicht offensichtlich ist (insbesondere eine *lokale* Entscheidung nach dem nächsten besten Element nicht möglich ist)

Algorithmen und Datenstrukturen

4.4 DYNAMISCHE PROGRAMMIERUNG

Dynamische Programmierung

- Vereint Ideen verschiedener Muster
  - Greedy: Wahl der optimalen Teillösung
  - Divide-and-Conquer / Backtracking: Rekursion, Konfigurationsbaum
- Unterschiede:
  - Divide-and-Conquer: löst *unabhängige* Teilprobleme
  - Dynamische Programmierung: Optimierung voneinander *abhängiger* Teilprobleme
- Dynamische Programmierung ist „bottom-up“-Realisierung der Backtracking-Strategie
- Anwendungsbereich: wie Greedy, jedoch insbesondere dort, wo Greedy nur suboptimale Lösungen liefert

Dynamische Programmierung: Idee

Bei der dynamischen Programmierung werden
kleinere Teilprobleme zuerst gelöst,
um aus diesen größere Teillösungen
zusammenzusetzen

- Problemlösen „auf Vorrat“
  - Möglichst nur Teilprobleme lösen, die bei der Lösung der großen Probleme auch tatsächlich benötigt werden
- Gewinn falls identische Teilprobleme in mehreren Lösungszweigen betrachtet werden
- Rekursives Problemlösen wird ersetzt durch Iteration und abgespeicherte Teilergebnisse

Schwierigkeiten bei dynamischer Programmierung

- Nicht immer ist es überhaupt möglich, die Lösungen kleinerer Probleme so zu kombinieren, dass sich die Lösung eines größeren Problems ergibt

- Die Anzahl der zu lösenden Probleme kann unvertretbar groß werden.
  - Zu viele Teillösungen, die dann doch nicht benötigt werden
  - Gewinn durch Wiederverwendung zu gering, da Lösungszweige disjunkt

Beispiel: Editierdistanz

Gegeben zwei Zeichenketten **s** und **t**, was ist die minimale Anzahl an Einfüge-, Lösch- und Ersetzoperationen um **s** in **t** zu transformieren?

Beispiel

s = „Haus“

t = „Maus“

Lösung: 1 (Ersetze „H“ durch „M“)

Beispiel: Editierdistanz

Beispiel

s = „Katze“

t = „Glatze“

Lösung: 2 (Ersetze „K“ durch „G“, füge „l“ hinzu)

Anwendungen

Rechtschreibprüfung, Plagiatserkennung

Formalisierung

Definition (Ein-Schritt Modifikation)

Betrachte **s**=$s_1...s_m$

1. Jedes **s'**=$s_1...s_{i-1}s_{i+1}...s_m$ (für i=1,...,m)
2. Jedes **s'**=$s_1...s_{i-1}xs_{i+1}...s_m$ (für i=1,...,m und x!=s_i)
3. Jedes **s'**=$s_1...s_ixs_{i+1}...s_m$ (für i=0,1,...,m und bel. x)

heißt Ein-Schritt Modifikation von **s**

Definition (k-Schritt Modifikation)

Eine Zeichenkette **t** heißt k-Schritt Modifikation (k>1) von **s** wenn es eine Zeichenkette u gibt mit

1. **u** ist eine Ein-Schritt Modifikation von **s**
2. **t** ist eine k-1-Schritt Modifikation von u

Formalisierung

Definition (Editierdistanz, auch Levenshtein-Distanz)

D(**s**,**t**) = min { d | **s** ist eine d-Schritt Modifikation von **t**}

Anmerkungen

- Ist **s** eine d-Schritt Modifikation von **t**, so ist **s** auch eine d+2j Modifikation von **t** (für jedes j>0). Warum?
- Eine minimale Modifikation muss nicht eindeutig sein

Wir sind aber hier nur an dem Wert einer minimalen Modifikation interessiert.

Charakterisierung und Algorithmus

Idee

Führe die Berechnung von D(**s**,**t**) auf die Berechnung von D auf die Präfixe von **s** und **t** zurück.

Definition D_{ij}(**s**,**t**)

Sei **s**=$s_1 \ldots s_m$ und **t**=$t_1 \ldots t_n$

Definiere D_{ij}(**s**,**t**)=D($s_1 \ldots s_i$,$t_1 \ldots t_j$) (für i=0,…,m, j=0,…,n)

Beachte für z.B i=0 haben wir $s_1 \ldots s_i$=ε (leerer String)

Beobachtung

Es gilt D_{mn}(**s**,**t**)=D(**s**,**t**) (dies ist zu berechnen)

Charakterisierung und Algorithmus

Sei $\mathbf{s}=s_1 \ldots s_m$ und $\mathbf{t}=t_1 \ldots t_n$

Beobachtung

- $D_{00}(\mathbf{s},\mathbf{t})=D(\varepsilon,\varepsilon) = 0$
  (zwei leere Strings sind identisch)
- $D_{0j}(\mathbf{s},\mathbf{t}) = D(\varepsilon,t_1 \ldots t_j) = j$ für $j=1,\ldots,n$
  (alle Zeichen $t_1 \ldots t_j$ müssen eingefügt werden)
- $D_{i0}(\mathbf{s},\mathbf{t}) = D(s_1 \ldots s_i ,\varepsilon) = i$ für $i=1,\ldots,m$
  (alle Zeichen $s_1 \ldots s_i$ müssen eingefügt werden)

Charakterisierung und Algorithmus

Theorem (Zentrale Charakterisierung der Editierdistanz)

Falls $s_i=t_j$: $D_{ij}(\mathbf{s},\mathbf{t}) = D_{i-1j-1}(\mathbf{s},\mathbf{t})$

Ansonsten: $$D_{ij}(\mathbf{s},\mathbf{t}) = \min \left\{ \begin{array}{ll} D_{i-1j-1}(\mathbf{s},\mathbf{t}) + 1 & \text{Ersetzung} \\ D_{ij-1}(\mathbf{s},\mathbf{t}) + 1 & \text{Einfügung} \\ D_{i-1j}(\mathbf{s},\mathbf{t}) + 1 & \text{Löschung} \end{array} \right\}$$

Beweis: Übungsaufgabe

Charakterisierung und Algorithmus

Pseudocode:

```
for j=0,…,n set D_0j(s,t) = j
for i=0,…,m set D_i0(s,t) = i
for i=1,…,m
    for j=1,…,n
        if s_i=t_j set D_ij(s,t)=D_i-1j-1(s,t)
        else D_ij(s,t)=
            min{D_i-1j-1(s,t)+1, D_ij-1(s,t)+1, D_i-1j(s,t)+1}
return D_mn(s,t)
```

Beispiel

D_{ij}		0	1	2	3	4	5	
			K	**A**	**T**	**Z**	**E**	**S**
0								
1	**G**							
2	**L**							
3	**A**							
4	**T**							
5	**Z**							
6	**E**							
	t							

Schritt 1: `For j=0,…,n set` D_{0j}`(s,t) = j`

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0					
1	**G**	1					
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Schritt 2: `For i=0,…,m set` D_{i0}`(s,t) = i`

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1					
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Schritt 3:

```
for i=1,…,m
    for j=1,…,n
        if s_i=t_j set D_ij(s,t)=D_i-1j-1(s,t)
        else D_ij(s,t)=
        min{D_i-1j-1(s,t)+1, D_ij-1(s,t)+1, D_i-1j(s,t)+1}
```

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1					
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

"K" != "G", also $D_{11}=\min\{D_{00}+1, D_{01}+1, D_{10}+1\}$

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1				
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5	
			K	**A**	**T**	**Z**	**E**	**S**
0		0	1	2	3	4	5	
1	**G**	1	1					
2	**L**	2						
3	**A**	3						
4	**T**	4						
5	**Z**	5						
6	**E**	6						
	t							

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2			
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3		
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2					
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2				
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2			
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3		
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3					
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3				
4	**T**	4					
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2			
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3		
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4					
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4				
5	**Z**	5					
6	**E**	6					
	t						

S

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3			
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2		
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5					
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5				
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4			
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3		
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	3
6	**E**	6					

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	3
6	**E**	6	6				

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	3
6	**E**	6	6	5			

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	3
6	**E**	6	6	5	4		

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	3
6	**E**	6	6	5	4	3	

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	3
6	**E**	6	6	5	4	3	2

S

t

Beispiel

D_{ij}		0	1	2	3	4	5
			K	**A**	**T**	**Z**	**E**
0		0	1	2	3	4	5
1	**G**	1	1	2	3	4	5
2	**L**	2	2	2	3	4	5
3	**A**	3	3	2	3	4	5
4	**T**	4	4	3	2	3	4
5	**Z**	5	5	4	3	2	3
6	**E**	6	6	5	4	3	2
	t						

S

Analyse

Theorem

Für endliche Zeichenketten **s** und **t** terminiert der Algorithmus `editdistance` nach endlich vielen Schritten.

Beweis: folgt aus dem nächsten Theorem □

Theorem

Für Eingaben $s=s_1 \ldots s_m$ und $t=t_1 \ldots t_n$ hat der Algorithmus eine Laufzeit von $\Theta(mn)$.

Beweis: Einfache Schleifenanalyse □

Analyse

Theorem

Der Algorithmus `editdistance` berechnet die Editierdistanz zweier Zeichenketten **s** und **t**.

Beweis: folgt direkt aus der Zentralen Charakterisierung der Editierdistanz. □

Problemklasse für Dynamische Programmierung

- Ähnliche Klasse wie bei Divide and Conquer
  - Selbstähnlichkeit
  - Kombinierbarkeit von Teillösungen
- Identische Teilprobleme treten häufig(er) auf
- „Wahrscheinlichkeit“ dass Teillösungen nie benötigt werden ist gering (denke an DP-Algorithmus zum Sortieren, der zunächst alle möglichen Teilfolgen sortiert)
- Für Optimierungsprobleme

Algorithmen und Datenstrukturen

4. ENTWURFSMUSTER
ZUSAMMENFASSUNG

Zusammenfassung

- Greedy Algorithmen
  - Wechselgeldalgorithmus
- Divide and Conquer Algorithmen
  - Türme von Hanoi
- Backtracking
  - N-Damen Problem
- Dynamische Programmierung
  - Editierdistanz