INF280: Competitive programming

Advanced datastructure algorithms

Louis Jachiet

Dynamic programming

What is Dynamic Programming (DP)?

Hard to define but roughly:

- we have a question depending on parameters
- that can be answered recursively
- the subproblems might appear multiple times or overlap

We don't really care what is *officially* a dynamic programming problem...

What is Dynamic Programming (DP)?

Alternative definition:

- we have a state (the parameters)
- we have transitions (the recursion)
- we compute a function over the states using the transitions

What is Dynamic Programming (DP)?

Alternative definition:

- we have a state (the parameters)
- we have transitions (the recursion)
- we compute a function over the states using the transitions

$\Rightarrow$ a graph problem! (usually acyclic graph)

Some typical DP problems

Compute F_n the n-th Fibonacci number

- **question(parameter):** compute fibo(n)
- **recursion:** $F_n = F_{n-1} + F_{n-2}$
- **overlapping subproblems:**

$$F_{n+2} = F_{n+1} + F_n = (F_n + F_{n-1}) + F_n$$

Some typical DP problems

I have weights $w_1 \dots w_k$ can I reach a weight of T

- **question(parameter):** $\text{reach}(w_1, \dots, w_k, T)$
- **recursion:** $\text{reach}(w_1, \dots, w_k, T) = \text{reach}(w_2, \dots, w_k, T) \vee \text{reach}(w_2, \dots, w_k, T - w_1)$
- **overlapping subproblems:**
  if, e.g., $w_1 = 1, w_2 = 2, w_3 = 3$ we can achieve $T = 3$ in two different ways

DP vs Memoization

Memoization

Memoization consists of storing the result of a function, so a new call with the parameters can be answered directly.

DP vs Memoization

Typical DP solutions use memoization but DP can be seen as something much larger...

DP vs Greedy algorithms

Both DP and Greedy algorithms can be applied to problems with a large set of configurations to be explored.

DP vs Greedy algorithms

Both DP and Greedy algorithms can be applied to problems with a large set of configurations to be explored.

Dynamic Programming

DP approach generally explores the full set of configurations by breaking down large problems into smaller problems while avoiding to compute twice the same thing

DP vs Greedy algorithms

Both DP and Greedy algorithms can be applied to problems with a large set of configurations to be explored.

Dynamic Programming

DP approach generally explores the full set of configurations by breaking down large problems into smaller problems while avoiding to compute twice the same thing

---

Greedy algorithms

A greedy approach makes locally optimal choices leading to a globally optimal solution.

DP vs Greedy algorithms

Both DP and Greedy algorithms can be applied to problems with a large set of configurations to be explored.

Dynamic Programming

DP approach generally explores the full set of configurations by breaking down large problems into smaller problems while avoiding to compute twice the same thing

---

Greedy algorithms

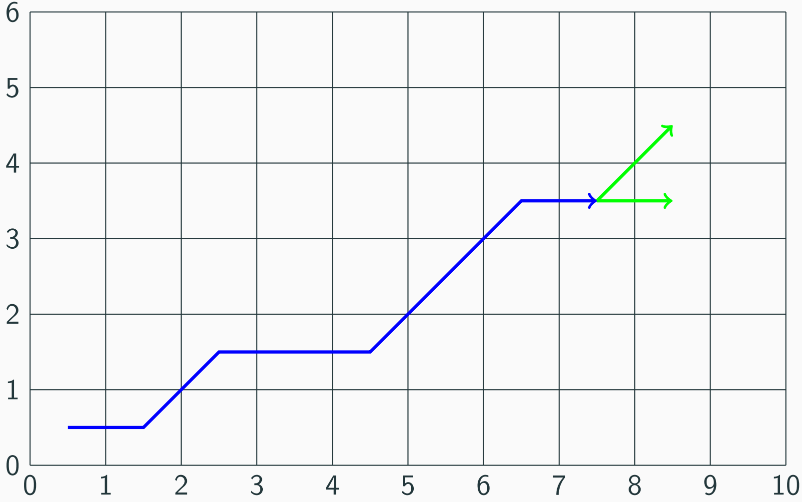
A greedy approach makes locally optimal choices leading to a globally optimal solution. Some algorithms are said greedy even if they lead to non optimal solutions

Let us solve some simple DP problems!

Exercise 1 to 4

Classical types of DP problems

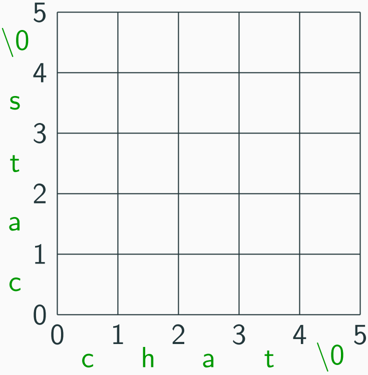
Path on grids



Applications: Number of down-right paths, Levenshtein distance

Levenshtein distance

Given two words $u_1 \ldots u_\ell$, $v_1 \ldots v_k$ what is the number of edits (replace, delete, or insert letter) needed to transform u into v?



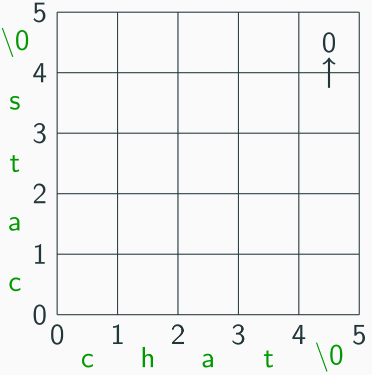
dist(i, j)= min

- dist$(i, j+1)$+1,
- dist$(i+1, j)$+1,
- dist$(i+1, j+1)$+1,
- dist$(i{+}1, j{+}1)$) si $v_i = u_j$

$\Rightarrow O(n^2)$ solution!

Levenshtein distance

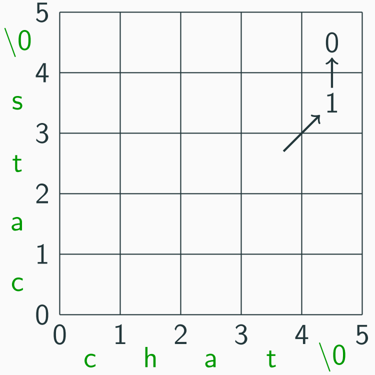
Given two words $u_1 \dots u_\ell$, $v_1 \dots v_k$ what is the number of edits (replace, delete, or insert letter) needed to transform u into v?



- insert s

Levenshtein distance

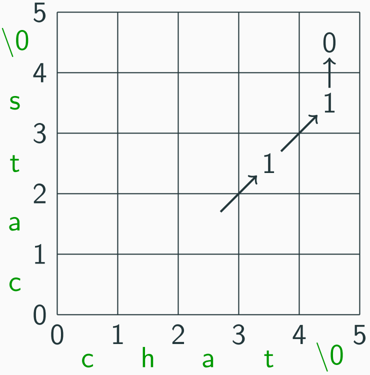
Given two words $u_1 \dots u_\ell$, $v_1 \dots v_k$ what is the number of edits (replace, delete, or insert letter) needed to transform u into v?



- insert s
- t=t

Levenshtein distance

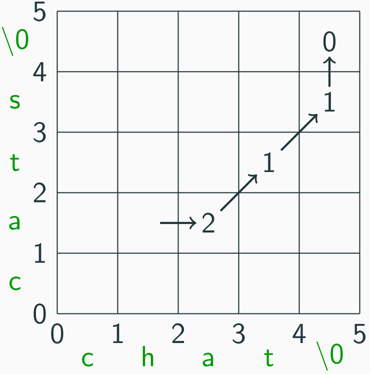
Given two words $u_1 \dots u_\ell$, $v_1 \dots v_k$ what is the number of edits (replace, delete, or insert letter) needed to transform u into v?



- insert s
- t=t
- a=a

Levenshtein distance

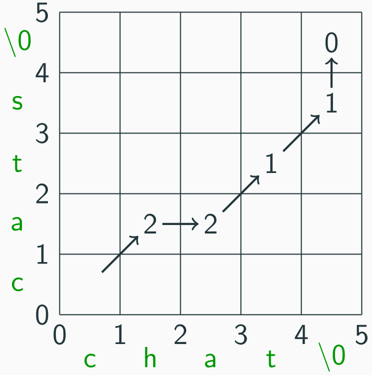
Given two words $u_1 \ldots u_\ell$, $v_1 \ldots v_k$ what is the number of edits (replace, delete, or insert letter) needed to transform u into v?



- insert s
- t=t
- a=a
- delete h

Levenshtein distance

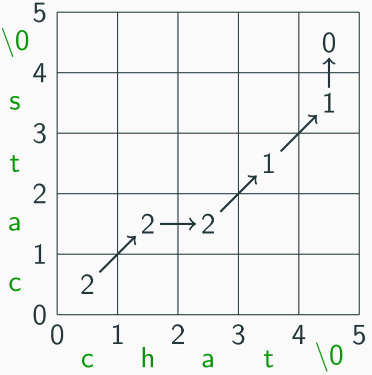
Given two words $u_1 \ldots u_\ell$, $v_1 \ldots v_k$ what is the number of edits (replace, delete, or insert letter) needed to transform u into v?



- insert s
- t=t
- a=a
- delete h
- c=c

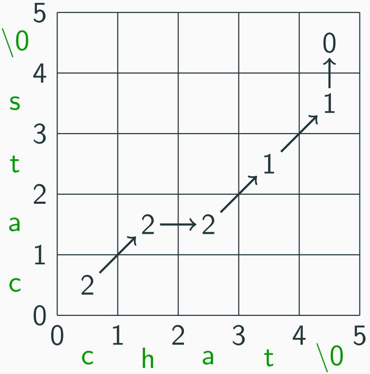
Levenshtein distance

Given two words $u_1 \ldots u_\ell$, $v_1 \ldots v_k$ what is the number of edits (replace, delete, or insert letter) needed to transform u into v?



- insert s
- t=t
- a=a
- delete h
- c=c

Levenshtein distance



Alternative solution

- we have a graph
- we can run a shortest path algorithm!

$\Rightarrow$ in $O(n \times d)$ where d is the distance.

Fix any ordering and then:

subsets$(e_1, \ldots, e_n)$ = subsets$(e_2, \ldots, e_n)$ with or without e_1

Some considerations:

- the target function needs to be "composable"
- sometimes the order matters
- using bitmasks might help

Range DP

Range DP problem

Given an array A compute some metric on all subarrays $A[i:j]$.

- in the simple case $do(i,j) \rightarrow \forall_{i<k<j} do(i,k) \wedge do(k,j)$ $\quad O(n^3)$
- sometimes $do(i,j) = do(i+1,j) \wedge do(i,j-1)$ $\quad O(n^2)$
- sometimes you need to have a clever trick to compute the full solution...

Sparse DP

Generally memory is not an issue with DP but you might have very few possible values over a large possible universe.

Sparse DP

Generally memory is not an issue with DP but you might have very few possible values over a large possible universe.

Use sets and hashsets!

Special cases of DP

Implementing a DP requires an acyclic recursion

What to do when the recursion might be cyclic?

- not care about it
- enforce it with a new parameter
- changing the problem

Examples

- use a DFS (DFS can be seen as DP with cyclicity)
- use a shortest path
- use the DAG of strongly connected components
- use an ad-hoc solution

How to improve an inefficient DP solution?

The systematic method

Write the recursive decision problem and

- **for each parameter:**
  - what are the possible values (min/max/nb)?
  - can it be deduced from the other parameters?
  - is it a strict equality?
- **for the recursion formula:**
  - can it be simplified?
  - are we recomputing the same thing twice?
  - can we precompute some part of it?
  - can we use an approach different from memoization?

How to implement DP solutions?

The problem and its solution

Levenshtein

we have two words $u_1 ... u_\ell$ and $v_1 ... v_k$ what is the edit distance between them?

Recursive solution

- $dist(i, j) =$ distance between $u_0 ... u_i$ and $v_0 ... v_j$

The problem and its solution

Levenshtein

we have two words $u_1 ... u_\ell$ and $v_1 ... v_k$ what is the edit distance between them?

Recursive solution

- $dist(i,j)$ = distance between $u_0 ... u_i$ and $v_0 ... v_j$
- $dist(-1,-1) = i + j + 2$ when $i < 0$ or $j < 0$
- $dist(i,j) = dist(i-1, j-i)$ when $u_i = v_j$
- $dist(i,j) = 1 + min(dist(i-1,j), dist(i,j-1), dist(i-1,j-1))$

Levenshtein

we have two words $u_1 \ldots u_\ell$ and $v_1 \ldots v_k$ what is the edit distance between them?

Constructive solution

$dist(i,j) =$ distance between $u_0 \ldots u_{i-1}$ and $v_0 \ldots v_{j-1}$. $dist(i,j)$ is the biggest number such that:

The problem and its solution

Levenshtein

we have two words $u_1 ... u_\ell$ and $v_1 ... v_k$ what is the edit distance between them?

Constructive solution

$dist(i,j) =$ distance between $u_0 ... u_{i-1}$ and $v_0 ... v_{j-1}$. $dist(i,j)$ is the biggest number such that:

- we have $dist(0,0) = 0$
- $dist(i+1, j+1) = dist(i,j)$ when $u_i = v_j$
- $dist(i+1, j) \leq 1 + dist(i,j)$
- $dist(i, j+1) \leq 1 + dist(i,j)$
- $dist(i+1, j+1) \leq 1 + dist(i,j)$

The recursive approach

```
const char u[Tm], v[Tm] ;
int dyn[Tm][Tm] ; // initialized to -INF
int dist( int i , int j ) {
  if(i<0 || j<0) // can only be -1 if negative
    return i+j+2; // avoid out of bounds access
    // i+j+2 = size of the non-empty string
  int & cur = dyn[i][j] ;
  if ( cur == -INF ) {
    if(u[i] == v[j])
      cur = dist(i-1,j-1);
    else
      cur = 1 + min(dist(i-1,j-1), dist(i-1,j), dist(i,j-1)) ;
  }
  return cur ;
}
```

The iterative approach

```
const char u[Tm], v[Tm] ;
int dist[Tm][Tm]; // dist[i][j]= dist(u_0..u_i-1, v_0..v_j-1)
void min_equal(int & a, int b) { if(a>b) a=b;}
void compute_dist() {
 fill(dist[0], dist[Tm], INF) ;
 dist[0][0] = 0;
 for(int i = 0 ; u[i] ; i++)
  for(int j = 0 ; v[j]  ; j++) {
   // at step (i,j) we set the value dist[i][j]
   if(i > 0) min_egal(dist[i][j],1+dist[i-1][j]);
   if(j > 0) min_egal(dist[i][j],1+dist[i][j-1]);
   if(i > 0 && j>0 )
    if(u[i-1] == v[j-1]) min_egal(dist[i][j], dist[i-1][j-1]);
    else min_egal(dist[i][j], 1+dist[i-1][j-1]);
} // answer in dist[lengthU-1][lengthV-1]
```

The iterative approach (alternative)

```
const char u[Tm], v[Tm] ;
int dist[Tm][Tm] ;

void compute_dist() {
  fill(dist[0], dist[Tm], INF) ;
  dist[0][0] = 0;
  for(int i = 0 ; u[i] ; i++)
    for(int j = 0 ; v[j]  ; j++) {
      // at step (i,j) we ``propagate'' the value dist[i][j]
      if(u[i] == v[j]) min_egal(dist[i+1][j+1], dist[i][j]);
      min_egal(dist[i+1][j+1], 1+dist[i][j]);
      min_egal(dist[i+1][j], dist[i][j]);
      min_egal(dist[i][j+1], dist[i][j]);
    }
} // answer in dist[lengthU][lengthV]
```