Overview

- The Power of Precomputation,
- Recursion (pars prima),

Maximum unit submatrix

Problem: Given an $m \times n$ matrix filled by zeroes and ones we want to find the largest (continuous) submatrix that contains only ones (numbers 1).

Naive approach

- Find all candidates for upper left and lower right corners. Inspect the interior.

Naive approach

- Find all candidates for upper left and lower right corners. Inspect the interior.
- This algorithm works. What is its complexity?

Naive approach

- Find all candidates for upper left and lower right corners. Inspect the interior.
- This algorithm works. What is its complexity?
- $\Theta(mn)$ left-upper-corner candidates, $\Theta(mn)$ right-lower..., $\Theta(mn)$ elements inside the candidate matrix (why?), altogether $\Theta(m^3n^3)$.

Naive approach

- Find all candidates for upper left and lower right corners. Inspect the interior.
- This algorithm works. What is its complexity?
- $\Theta(mn)$ left-upper-corner candidates, $\Theta(mn)$ right-lower..., $\Theta(mn)$ elements inside the candidate matrix (why?), altogether $\Theta(m^3n^3)$.
- Ideas for improvement?

Precomputation

- For each 1-element we compute the number of ones lying (immediately) below it (i.e., in a column without being interrupted by 0).

Precomputation

- For each 1-element we compute the number of ones lying (immediately) below it (i.e., in a column without being interrupted by 0).
- We index each such candidate by the left- and right- upper corner.

Precomputation

- For each 1-element we compute the number of ones lying (immediately) below it (i.e., in a column without being interrupted by 0).
- We index each such candidate by the left- and right- upper corner.
  - For each left upper corner try all possibilities of right upper corner (i.e., in the same row).

Precomputation

- For each 1-element we compute the number of ones lying (immediately) below it (i.e., in a column without being interrupted by 0).
- We index each such candidate by the left- and right- upper corner.
  - For each left upper corner try all possibilities of right upper corner (i.e., in the same row).
  - These candidates must not be separated by 0 (i.e., they belong to the same block of 1's in the row).

Precomputation

- For each 1-element we compute the number of ones lying (immediately) below it (i.e., in a column without being interrupted by 0).
- We index each such candidate by the left- and right- upper corner.
    - For each left upper corner try all possibilities of right upper corner (i.e., in the same row).
    - These candidates must not be separated by 0 (i.e., they belong to the same block of 1's in the row).
    - As we know numbers of 1's below each element, the height of such matrix gets determined as minimum of these numbers.

Precomputation

- For each 1-element we compute the number of ones lying (immediately) below it (i.e., in a column without being interrupted by 0).
- We index each such candidate by the left- and right- upper corner.
    - For each left upper corner try all possibilities of right upper corner (i.e., in the same row).
    - These candidates must not be separated by 0 (i.e., they belong to the same block of 1's in the row).
    - As we know numbers of 1's below each element, the height of such matrix gets determined as minimum of these numbers.
    - The rest is just multiplying (the sizes) and comparisons (of the sizes).

Precomputation

- For each 1-element we compute the number of ones lying (immediately) below it (i.e., in a column without being interrupted by 0).
- We index each such candidate by the left- and right- upper corner.
  - For each left upper corner try all possibilities of right upper corner (i.e., in the same row).
  - These candidates must not be separated by 0 (i.e., they belong to the same block of 1's in the row).
  - As we know numbers of 1's below each element, the height of such matrix gets determined as minimum of these numbers.
  - The rest is just multiplying (the sizes) and comparisons (of the sizes).
- Complexity: Precomputation $O(mn)$, computation $O(m^2n)$.

Can we find a better algorithm?

Surprisingly, yes. And the algorithm also uses a precomputation.

- Determine the number of ones below each element ($\rightarrow B$),

Can we find a better algorithm?

Surprisingly, yes. And the algorithm also uses a precomputation.

- Determine the number of ones below each element ($\rightarrow B$),
- Determine the number of ones above each element ($\rightarrow C$),

Can we find a better algorithm?

Surprisingly, yes. And the algorithm also uses a precomputation.

- Determine the number of ones below each element ($\rightarrow B$),
- Determine the number of ones above each element ($\rightarrow C$),
- Index the candidate-matrices by the left critical end, i.e., the left end where the matrix neighbors with a zero-element, i.e., $a_{i,j} = 1$ and $a_{i,j-1} = 0$ or $j = 1$ ($a_{i,j-1}$ is not a member of a matrix).

Can we find a better algorithm?

Surprisingly, yes. And the algorithm also uses a precomputation.

- Determine the number of ones below each element ($\rightarrow B$),
- Determine the number of ones above each element ($\rightarrow C$),
- Index the candidate-matrices by the left critical end, i.e., the left end where the matrix neighbors with a zero-element, i.e., $a_{i,j} = 1$ and $a_{i,j-1} = 0$ or $j = 1$ ($a_{i,j-1}$ is not a member of a matrix).
- Try all possible candidates for the right end (in the appropriate line).

Complexity analysis

- Precomputation (determining the matrices B and C): $\Theta(mn)$,
- although it seems that the complexity does not change, the truth is different:
- We are trying each right-end-candidate at most once!
- Therefore, altogether, $\Theta(mn)$. As the complexity of the problem is $\Omega(mn)$, we have estimated the complexity of the problem ($\Theta(mn)$) and thus the algorithm is optimal (up to a (multiplicative) constant).

Recursion

- It sometimes makes sense to call a function directly from itself.

Recursion

- It sometimes makes sense to call a function directly from itself.
- This is called a **recursion.**

Recursion

- It sometimes makes sense to call a function directly from itself.
- This is called a **recursion.**
- Recursion is nothing else than just a renamed induction!

Recursion

- It sometimes makes sense to call a function directly from itself.
- This is called a **recursion.**
- Recursion is nothing else than just a renamed induction!
- Examples: Clerks at the authority-offices, factorial, Caesar's cipher...

Recursion

- It sometimes makes sense to call a function directly from itself.
- This is called a **recursion.**
- Recursion is nothing else than just a renamed induction!
- Examples: Clerks at the authority-offices, factorial, Caesar's cipher...
- Note that we are showing problems where the recursion can be applied (not necessarily problems optimally solved by recursion)!

Clerks in bureaus

- A citizen wants to perform a legal decision.

Clerks in bureaus

- A citizen wants to perform a legal decision.
- A clerk wants particular forms to get filled-in (which requires visits of further authorities).

Clerks in bureaus

- A citizen wants to perform a legal decision.
- A clerk wants particular forms to get filled-in (which requires visits of further authorities).
- Solution:

```
procedure fill_in(to_fill:list_of_forms);
var x:list_of_forms;
for form in to_fill do
begin
    x:=ask_a_clerk(form);
    fill_in(x);
end;
```

Factorial

- $n! = 1 \cdot 2 \cdot \ldots \cdot n$

Factorial

- $n! = 1 \cdot 2 \cdot \ldots \cdot n$
- How to implement this?

Factorial

- $n! = 1 \cdot 2 \cdot \ldots \cdot n$
- How to implement this?
- Using a loop:

```
fakt:=1;
for i:=1 to n do
    fakt:=fakt*i;
```

Factorial

- $n! = 1 \cdot 2 \cdot \ldots \cdot n$
- How to implement this?
- Using a loop:

```
fakt:=1;
for i:=1 to n do
    fakt:=fakt*i;
```

- or using recursion.

Factorial using recursion

```
function factorial(a:integer):integer;
begin
    if a<2 then
          factorial:=1;
    else  factorial:=a*factorial(a-1);
end;
```

Computational complexity of this function?

Lecturer goes to the lecture-room

- When going to the lecture-room, the lecturer uses a stair-case. When making a step he has two options. Place his foot on the next step (in the stair) or to skip one step (and place his foot on the step beyond that.
- In how many distinct ways he can reach the room S11? (do not calculate exact number of stairs, try to estimate with a reasonable precision)
- Ideas?

Lecturer goes to the lecture-room – a solution

- We get a recurrence $f_n = f_{n-1} + f_{n-2}$.

Lecturer goes to the lecture-room – a solution

- We get a recurrence $f_n = f_{n-1} + f_{n-2}$.
- Recurrence is nothing else than a mathematically notated recursion.

Lecturer goes to the lecture-room – a solution

- We get a recurrence $f_n = f_{n-1} + f_{n-2}$.
- Recurrence is nothing else than a mathematically notated recursion.
- Solution:

```
function stairs(a:integer):integer;
begin
    if a=1 then stairs=1;
    else if a=2 then stairs=2;
        else
            stairs:=stairs(a-1)+stairs(a-2);
end;
```

Lecturer goes to the lecture-room – a solution

- We get a recurrence $f_n = f_{n-1} + f_{n-2}$.
- Recurrence is nothing else than a mathematically notated recursion.
- Solution:

```
function stairs(a:integer):integer;
begin
    if a=1 then stairs=1;
    else if a=2 then stairs=2;
        else
            stairs:=stairs(a-1)+stairs(a-2);
end;
```

- What is the problem (with this solution)?

Lecturer goes to the lecture-room – a solution

- We get a recurrence $f_n = f_{n-1} + f_{n-2}$.
- Recurrence is nothing else than a mathematically notated recursion.
- Solution:

```
function stairs(a:integer):integer;
begin
    if a=1 then stairs=1;
    else if a=2 then stairs=2;
        else
            stairs:=stairs(a-1)+stairs(a-2);
end;
```

- What is the problem (with this solution)?
- Complexity!

The Basic Idea behind Recursion

- Recursion is a method how to solve a given problem in such a way that in particular (consecutive) steps we are decreasing the size of the instance (up to a small-enough instance) and then we are extending the solutions (for the smaller instances) to the solution of the given (larger) instance.

The Basic Idea behind Recursion

- Recursion is a method how to solve a given problem in such a way that in particular (consecutive) steps we are decreasing the size of the instance (up to a small-enough instance) and then we are extending the solutions (for the smaller instances) to the solution of the given (larger) instance.
- Further example: Output all the numbers in a given numeral system (with a given base and length).

The Main Program

```
program q;
const MAX=10;
var dig,base:integer;
      arr:array[1..MAX] of integer;
begin
      write('Input the number of digits:  ');
      readln(dig);
      if(dig>MAX) then
            halt;{Number too long}
      write('Input the base of the system:  ');
      readln(base);
      if base>10 then
            halt;{Too large base!}
      fill(1);
end.
```

The Recursive Kernel

```
procedure fill(where:integer);
var i:integer;
begin
      if(where<=dig) then
            for i:=0 to base-1 do
            begin
                  arr[where]:=i;
                  fill(where+1);
            end
      else output;
end;
```

The Output-procedure

```
procedure output;
var i:integer;
start:boolean;
begin
      start:=true;
      for i:=1 to dig do
            if((not start) or (arr[i]<>0)) then
            begin
                  start:=false;
                  write(arr[i]);
            end;
      if start then write(0);
      writeln;
end;
```