DP

Dynamic programming

Stefanie Zbinden

2018-12-02

Swiss Olympiad in Informatics

DP

Don't calculate again what you already calculated before.

Repetition - Fibonacci

Repetition - Fibonacci

Task: Given a number n, we want to compute the n^{th} Fibonacci number, where the n^{th} Fibonacci number is defined as

$$f_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

Repetition - Fibonacci

Solution: This is an easy recursion, thus we can calculate the number via a recursive function

```
int f (int n) {
    if (n<=1) return n;
    return f(n-1)+f(n-2);
}
```

Repetition - Fibonacci

Solution 2: We calculate $f(n-2)$ multiple times. Instead of calculating it over and over again, we apply DP and store the value

```
vector <int> fibonacci;
int f (int n){
    if (fibonacci[n]!=-1) return fibonacci[n];
    if (n<=1) return fibonacci[n]=n;
    return fibonacci[n]=f(n-1)+f(n-2);
}
int main(){
    int n;
    cin >> n;
    fibonacci=vector<int> (n+1, -1);
    cout << f(n) << "\n";
}
```

Repetition - Fibonacci

Improvement: Our first solution needed exponential time, by saving the values of $f(n)$, once calculated, we managed to derive a solution which is only linear time.

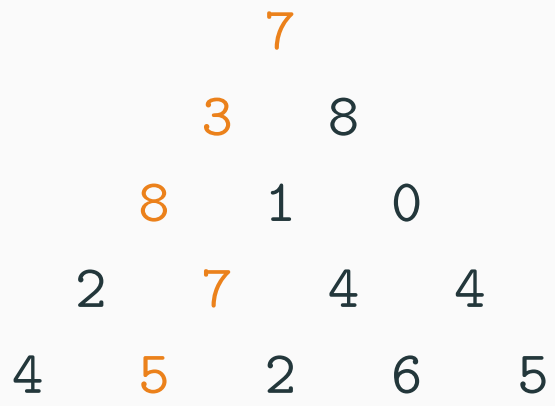
Memoization: Using a recursive function to implement the DP is called memoization.

DP examples

Trisum

Task: Given a triangle with numbers in it. Calculate the sum of the maximal path from the bottom to the top. In each step, one can only go down to the left or to the right.

Example:



Trisum

Observation 1: The optimal path from the bottom to the top is either the optimal path from the bottom to the left of the top and then to the top or the optimal path from the bottom to the right of the top and then to the top.

Observation 2: We are not really interested in the path, only in finding the maximum value.

Observation 3: The same two observations don't only hold for the top but for any number in the triangle.

Trisum

Our observations lead to the following recursion:

```
maxsum[i][j]=max(maxsum[i+1][j+1], maxsum[i+1][j])
                + a[i][j]
```

Trisum- Bottom Up Solution

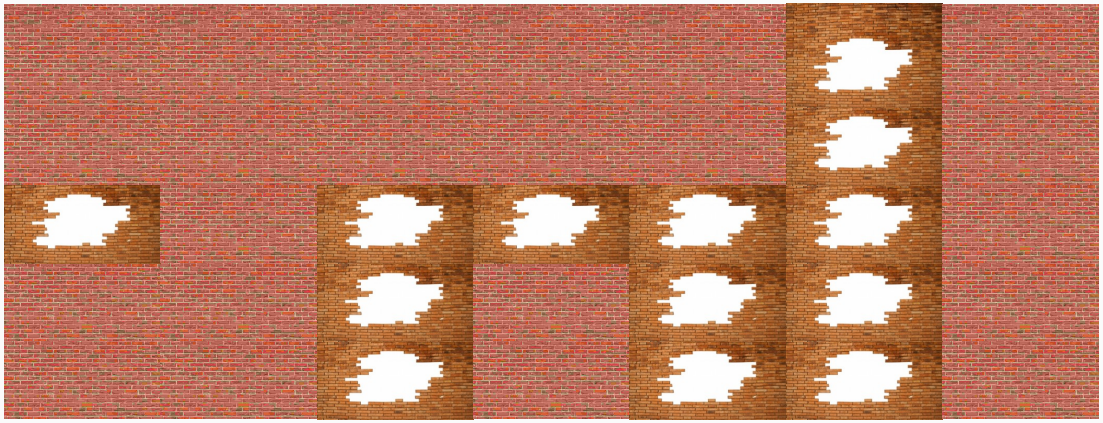
```
vector<vector <int> > maxsum (n, vector <int> (n));
for (int j=0; j<n; j++){ //Init the bottom line
    maxsum[n-1][j]=a[n-1][j];
}

for (int i=n-1; i>=0; i--){
    for (int j=0; j<=i; j++){
        maxsum[i][j]=max(maxsum[i+1][j],
                         maxsum[i+1][j+1])+a[i][j];
    }
}
cout << maxsum[0][0] << '\n';
```

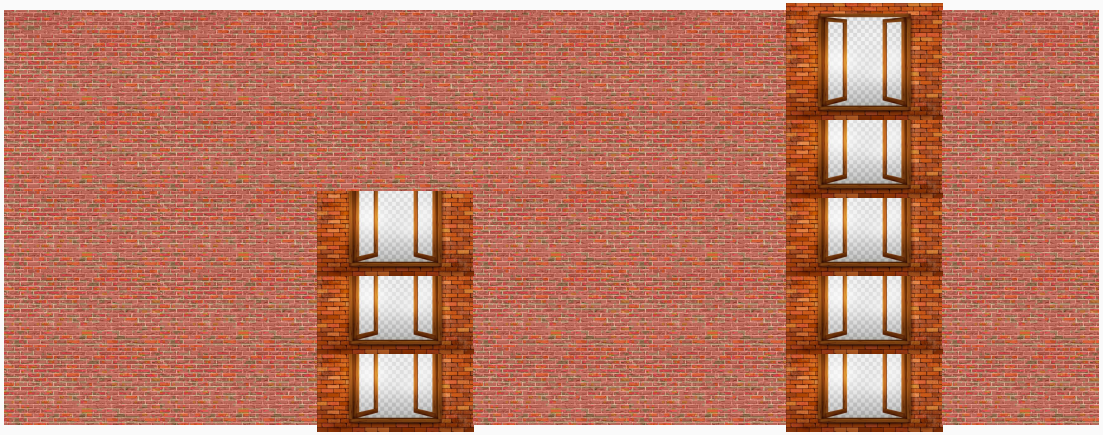
Renovate

Task: Stofl wants to renovate a building front consisting of a n times m grid. If a cell is damaged, Stofl can either fix it or tear it out and place a window. However, each window has to have a non window column on it's left and right. Find out, how many windows Stofl can build.

Renovate

Example:





Renovate

Recursion:

```
//a[i] is the number of damaged cells in column i
wind[i]=wall[i-1]+a[i];
wall[i]=max(wind[i-1], wall[i-1]);
```

Renovate

```
//house: vector that has a 1, if the cell is fine
// and a 0 if it is damaged
vector<int> a (n);
for (int i=0; i<n; i++){
    for (int j=0; j<m; j++){
        a[i]+=house[i][j];
    }
}
vector<int> wall (n);
vector<int> window (n);
for (int i=1; i<n; i++){
    wall[i]=max(wind[i-1], wall[i-1]);
    wind[i]=a[i]+wall[i-1];
}
cout << wall[n-1] << '\n';
```

Remarks

Important things you should remember:

- Don't calculate things multiple times
- Try to find the optimal solutions for a subproblem in order to calculate the whole solution.
- Analyse the problem and it's properties: What assumptions can we make about the optimal solution? Which statements are true for any solution?
- **First think, then code!**