# Dynamic programming

beOI 2015 Carnival Training
Day 2 - Floris Kint

Notes: http://bit.ly/1CLLzQr

# Key principle

Don’t do anything stupid

**Don’t do anything twice**

# Ex 1: Fibonacci numbers

```
int fibo(int n)
{
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
        return fibo(n-1) + fibo(n-2);
}
```

Time complexity: O(2^n)

Space complexity (stack): O(n)

# Ex 1: Fibonacci numbers

```
int fibo(int n)
{
    vector<int> f;
    f.push_back(0);
    f.push_back(1);
    for(int i=2; i<=n; i++)
        f.push_back(f[i-1] + f[i-2]);
    return f[n];
}
```

Time complexity: O(n)

Space complexity: O(n)

# Ex 2: Longest Increasing subsequence

```
int lis(vector<int> &a, int min_value, int start_index)
{
  int best = 0;
  int smallest_found = 999999999;
  for(int i = start_index; i < a.size(); ++i)
  {
    if(a[i] > min_value && a[i] < smallest_found)
    {
      best = max(best, 1+lis(a, a[i], i+1));
      smallest_found = a[i];
    }
  }
  return best;
}
```

Time complexity: very bad

# Ex 2: Longest increasing subsequence

```
int lis(vector<int> a)
{
  int n = a.size();
  vector<int> ending_with;
  for(int i=0; i<n; i++)
    ending_with.push_back(0);
  int best = 0;
  for(int i=0; i<n; i++)
    for(int j=0; j < a[i]; j++){
      ending_with[a[i]] = max(ending_with[a[i]], ending_with[j]+1);
      best = max(best, ending_with[a[i]]);
  }
  return best;
}
```

Time complexity: O(N^2)

# Ex 2: Longest increasing subsequence

```
vector<int> lis(vector<int> a)
{
  int L=1; // Longest so far
  vector<int> smallest_end_for;
  smallest_end_for.push_back(-1);
  smallest_end_for.push_back(0);
  for(int i=1; i<a.size(); i++)
  {
    // Binary search for the best length before a[i]
    int lower=0, upper=L+1;
    while(lower+1 < upper)
    {
      int middle = (lower+upper)/2;
      if(a[smallest_end_for[middle]] < a[i])
        lower = middle;
      else
        upper = middle;
    }
    int prev_len = lower;
    // If the length is the best so far
    if(prev_len + 1 > L)
    {
      smallest_end_for.push_back(i);
      L++;
    }
    // If this value is the new smallest for this length
    else if(a[i] < a[smallest_end_for[prev_len+1]])
    {
      smallest_end_for[prev_len+1] = i;
    }
  }
  return L;
}
```

Time complexity: O(N*log(N))

# Approaches

**Top-down (lazy approach)**

*If not previously calculated: calculate & store*

*Use stored solution*

**Bottom-up**

*Build table of solutions from small to large, each time using the solutions of smaller problems*

# 2-Dimensional DP

Cheapest way to go right/down.

| | | | |
|---|---|---|---|
| 1 | 2 | 1 | 3 |
| 5 | 1 | 10 | 8 |
| 3 | 12 | 1 | 8 |
| 13 | 5 | 3 | 2 |

| | | | |
|---|---|---|---|
| 1 (1) | 2 (3) | 2 (5) | 3 (8) |
| 5 (6) | 1 (4) | 10 (14) | 8 (16) |
| 3 (9) | 12 (16) | 1 (15) | 8 (23) |
| 13 (32) | 5 (21) | 3 (18) | 2 (20) |

# 2-Dimensional DP: Top-down

```
1  int cheapest(int row, int col){
2    if(row == 0 && col == 0)
3      return grid[0][0];
4    if(calculated[row][col]==-1){
5      if(row == 0)
6        calculated[row][col] = grid[row][col] + cheapest(
             row, col-1);
7      else if(col == 0)
8        calculated[row][col] = grid[row][col] + cheapest(
             row-1, col);
9      else
10       calculated[row][col] = grid[row][col] + min(
             cheapest(row-1, col), cheapest(row, col-1));
11   }
12   return calculated[row][col];
13 }
```

# 2-Dimensional DP: Bottom-up

```
1  int cheapest(int row, int col){
2    calculated[0][0] = grid[0][0];
3    for(int c = 1; c <= col; ++c)
4      calculated[0][c] = calculated[0][c-1]+grid[0][c];
5    for(int r = 1; r <= row; ++r)
6      calculated[r][0] = calculated[r-1][0] + grid[r][0];
7    for(int r = 1; r <= row; ++r)
8      for(int c = 1; c <= col; ++c)
9        calculated[r][c] = grid[r][c] + min(calculated[r
             -1][c], calculated[r][c-1]);
10   return calculated[row][col];
11 }
```

# How to recognize

Knapsack

Sequences (common, increasing…)

Divide and Conquer

# Applications

Floyd-Warshall

Bellman-Ford

# Exercises