Dynamic Programming. Divide and Conquer. Backtracking

Paul Diac
Faculty of Computer Science Iaşi

November 28, 2018

Outline

Content

Outline

Content

Dynamic Programming is probably the most frequent type of problems meet in algorithmic contests. Moreover, DP problems can also require a large other techniques such as: graphs, trees, data structures, optimizations, mathematics: combinatorics, geometry, game theory, number theory, backtracking and more.

DP Paradigm : algorithmic technique which is usually based on a recurrent formula / structure and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity. [topcoder].

DP Paradigm : algorithmic technique which is usually based on a recurrent formula / structure and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity. [topcoder].

Sub-problems should be understood as sub-problem **instances** and they can *overlap*. The answer for a problem instance can be computed based on answers of other problem instances for which some atomic information is enough (like some number: minimum / sum / counter or other).

Classical DP problems. Fibonacci sequence.

- $F_1 = F_2 = 1$

Classical DP problems. Fibonacci sequence.

- $F_1 = F_2 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for all i $\geqslant 3$

Classical DP problems. Fibonacci sequence.

- $F_1 = F_2 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for all i $\geqslant 3$
- Given n, find F_n

Classical DP problems. Fibonacci sequence.

- $F_1 = F_2 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for all i $\geqslant 3$
- Given n, find F_n
- Most natural solution could be:

```
int fibo(int i) {
    if (i >= 3) {
        return fibo(i-1) + fibo(i-2);
    } else {
        return 1;
    }
} // solution is fibo(n)
```

Classical DP problems. Fibonacci sequence.

- $F_1 = F_2 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for all i $\geqslant 3$
- Given n, find F_n
- Most natural solution could be:

```
int fibo(int i) {
    if (i >= 3) {
        return fibo(i-1) + fibo(i-2);
    } else {
        return 1;
    }
} // solution is fibo(n)
```

- DP solution is:

```
int v[NMax];
v[1] = v[2] = 1;
for (int i = 3; i <= n; i++) {
    v[i] = v[i-1] + v[i-2];
} // solution is v[n];
```

Classical DP problems. Triangle: IOI 1994.

- Given some numbers arranged like in the triangle below, find the maximum sum of elements on a path starting from the first number that reaches the bottom level and on each consecutive level goes either left or right.

Classical DP problems. Triangle: IOI 1994.

- Given some numbers arranged like in the triangle below, find the maximum sum of elements on a path starting from the first number that reaches the bottom level and on each consecutive level goes either left or right.

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

Classical DP problems. Triangle: IOI 1994.

- Given some numbers arranged like in the triangle below, find the maximum sum of elements on a path starting from the first number that reaches the bottom level and on each consecutive level goes either left or right.

$$
\begin{array}{ccccccccc}
 & & & & 7 & & & & \\
 & & & 3 & & 8 & & & \\
 & & 8 & & 1 & & 0 & & \\
 & 2 & & 7 & & 4 & & 4 & \\
4 & & 5 & & 2 & & 6 & & 5
\end{array}
$$

- The solution is **30** = 7 + 3 + 8 + 7 + 5.

Classical DP problems. Longest Increasing Subsequence.

- Given an array of integers, find the longest subsequence of increasing numbers.

Classical DP problems. Longest Increasing Subsequence.

- Given an array of integers, find the longest subsequence of increasing numbers.
- Or, given $v_1, v_2, ..., v_n$ find $v_{s_1}, v_{s_2}, ..., v_{s_k}$ with $s_1 < s_2 < ... < s_k$ such that $v_{s_1} \leqslant v_{s_2} \leqslant ... \leqslant v_{s_k}$ and k is as high as possible.

Classical DP problems. Longest Increasing Subsequence.

- Given an array of integers, find the longest subsequence of increasing numbers.
- Or, given $v_1, v_2, ..., v_n$ find $v_{s_1}, v_{s_2}, ..., v_{s_k}$ with $s_1 < s_2 < ... < s_k$ such that $v_{s_1} \leqslant v_{s_2} \leqslant ... \leqslant v_{s_k}$ and k is as high as possible.
- Let $best_i$ be the result for the problem considering the first i numbers. Then, the DP formula is:

Classical DP problems. Longest Increasing Subsequence.

- Given an array of integers, find the longest subsequence of increasing numbers.
- Or, given $v_1, v_2, ..., v_n$ find $v_{s_1}, v_{s_2}, ..., v_{s_k}$ with $s_1 < s_2 < ... < s_k$ such that $v_{s_1} \leqslant v_{s_2} \leqslant ... \leqslant v_{s_k}$ and k is as high as possible.
- Let $best_i$ be the result for the problem considering the first i numbers. Then, the DP formula is:

$$best_i = max(best_j) + 1, \text{for all } j \text{ such that } j < i \text{ and } v[j] < v[i]$$

$$best_i = 1 \text{ , if no such } j \text{ exists}$$

Classical DP problems. Longest Increasing Subsequence.

- Given an array of integers, find the longest subsequence of increasing numbers.
- Or, given $v_1, v_2, ..., v_n$ find $v_{s_1}, v_{s_2}, ..., v_{s_k}$ with $s_1 < s_2 < ... < s_k$ such that $v_{s_1} \leqslant v_{s_2} \leqslant ... \leqslant v_{s_k}$ and k is as high as possible.
- Let $best_i$ be the result for the problem considering the first i numbers. Then, the DP formula is:

$$best_i = max(best_j) + 1, \text{for all } j \text{ such that } j < i \text{ and } v[j] < v[i]$$

$$best_i = 1 \text{ , if no such } j \text{ exists}$$

- O(NlogN) solution?

Classical DP problems. 0-1 Knapsack problem

Classical DP problems. 0-1 Knapsack problem

- Given N objects, each with a profit p_i and a weight w_i. We must choose some objects, such that the total weight does not exceed a given value W, and maximize the profit.

Classical DP problems. 0-1 Knapsack problem

- Given N objects, each with a profit p_i and a weight w_i. We must choose some objects, such that the total weight does not exceed a given value W, and maximize the profit.
- The DP structure is not that obvious here. The solution is to use indices of objects and weights each as a separate *dimension* in a structure on which we can find some recursion.

Classical DP problems. 0-1 Knapsack problem

- Given N objects, each with a profit p_i and a weight w_i. We must choose some objects, such that the total weight does not exceed a given value W, and maximize the profit.
- The DP structure is not that obvious here. The solution is to use indices of objects and weights each as a separate *dimension* in a structure on which we can find some recursion.

$$profit[i][j] = max \begin{cases} profit[i-1][j] \text{ (don't add object i)} \\ profit[i-1][j-w_i] + p_i \text{ (add object i)} \end{cases}$$

for all i and j, avoiding any negative indices.

Path reconstruction. Many problems require finding the solution structure itself, not just one value describing it.

Path reconstruction. Many problems require finding the solution structure itself, not just one value describing it.

First solution for this would be to start from the element describing the solution (whose dimensions describe its state), and then (in a loop until an initial state is reached), re-iterate all the states that the current state could depended on, and find the one from where the optimum came from. Next current state is initialized with that one state and so on until any initial state is reached.

Path reconstruction. Many problems require finding the solution structure itself, not just one value describing it.

First solution for this would be to start from the element describing the solution (whose dimensions describe its state), and then (in a loop until an initial state is reached), re-iterate all the states that the current state could depended on, and find the one from where the optimum came from. Next current state is initialized with that one state and so on until any initial state is reached.

Possibly an improvement (at least in this stage) would be to keep information about this *predecessor* state, for all states. Then, the re-iteration is not necessary and can be done in one operation, just by going to the predecessor.

Path reconstruction. Many problems require finding the solution structure itself, not just one value describing it.

First solution for this would be to start from the element describing the solution (whose dimensions describe its state), and then (in a loop until an initial state is reached), re-iterate all the states that the current state could depended on, and find the one from where the optimum came from. Next current state is initialized with that one state and so on until any initial state is reached.

Possibly an improvement (at least in this stage) would be to keep information about this *predecessor* state, for all states. Then, the re-iteration is not necessary and can be done in one operation, just by going to the predecessor.

In case of criteria like smallest lexicographic solutions, predecessors must be kept accordingly.

Known types of DP problems.

Known types of DP problems.

- DP on trees: asmax, color2

Known types of DP problems.

- DP on trees: asmax, color2
- DP on configurations: pavare, false mirrors
- DP and geometry: pomi, geometrie (hard)

Outline

Content

Divide and Conquer

Divide and Conquer paradigm uses a similar structure with DP, as it reduces the initial problem instance to sub-problems.

Divide and Conquer paradigm uses a similar structure with DP, as it reduces the initial problem instance to sub-problems.

It is different tough because:

- sub-problems don't overlap
- the problem usually doesn't require finding a numeric value but to do some processing (sorting, searching, etc).

Divide and Conquer paradigm uses a similar structure with DP, as it reduces the initial problem instance to sub-problems.

It is different tough because:

- sub-problems don't overlap
- the problem usually doesn't require finding a numeric value but to do some processing (sorting, searching, etc).

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. wikipedia.

Problems

Problems

- Fibonacci ...

Problems

- Fibonacci ...
- binary search ...

Problems

- Fibonacci ...
- binary search ...
- mergesort, quicksort

Problems

- Fibonacci ...
- binary search ...
- mergesort, quicksort
- Maximum Sequence

Problems

- Fibonacci ...
- binary search ...
- mergesort, quicksort
- Maximum Sequence
- Closest Points on a Plane

Outline

Content

Backtracking Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution wikipedia.

Recursive backtracking implementation

Easier to implement recursively. Example: generating all permutations of elements **[1..N]**.

```
int n, v[10], used[10];

void bkt(int k) {
 if (k == n) {
  for (int i = 0; i < n; i++) {
   print(v[i]);
  }
 } else {
  for (v[k] = 1; v[k] <= n; v[k]++) {
   if (used[v[k]] == 0) {
    used[v[k]] = 1;
    bkt(k + 1);
    used[v[k]] = 0;
}}}}
```

```
int main() {
  read(n);
  bkt(0);
  return 0;
}
```

STL next permutation

```
for (int i = 1; i <= n; ++i)
 v.push_back(i);
 do {
  for (int i = 0; i < n; ++i) print(v[i]);
  print('\n');
} while (next_permutation(v.begin(), v.end()));
```

Use bitset to generate all subsets

To generate all subsets of a set of **n** elements.

```
for (int set = 0; set < (1 << n); set++) {
 for (int i = 0; i < n; i++) {
  if (set & (1 << i)) {
   // included
  } else {
   // not included
  }
 }
}
```

Outline

Content

Dynamic Programming problems

rucsac
scmax

Divide and conquer problems

maximum sequence

Backtracking problems

combinari
problema reginelor