CS 341 - Algorithms, Spring 2017. University of Waterloo

Lecture 11: Dynamic programming

We study the technique of dynamic programming through various examples.

Introduction

In short, we can solve a problem by dynamic programming if there is a recurrence relation with only a "small" (polynomially many) subproblems.

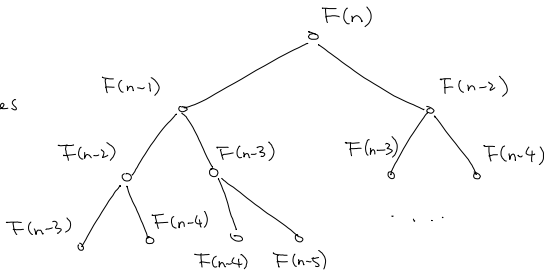
To illustrate it with a simple example, consider the problem of computing the Fibonacci sequence:

$$F(n) = F(n-1) + F(n-2);$$

$$F(1) = F(2) = 1.$$

The function is defined recursively with the base cases given.

It is then natural to compute it using recursion, but if we trace the recursion tree, we find out that it is huge.



If we solve the recurrence relation (using the techniques learned in MATH 239), then we find out that the runtime of this algorithm is $\Theta(1.618^n)$.

Observe that the recursion tree is highly redundant, many subproblems are evaluated over and over again.

There are only n subproblems $F(n), \ldots, F(1)$. Why are we wasting so much time?

There are two approaches to solve this problem efficiently.

Top-down memorization

Like BFS/DFS, we just used an array visited[i] to make sure that we only compute each subproblem at most once.

```
visited[i] = false    for 1 ≤ i ≤ n.
F(n).                                  } main program

F(n)    // recursive function
```

if visited[n] = true , return answer[n].

if n = 1 or n = 2 , return 1.

answer[n] = F(n-1) + F(n-2) .

visited[n] = true

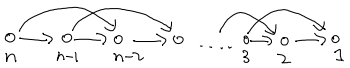
return answer[n].

Time complexity Each subproblem is only called once (when visited[i] = false)

When it is called, it looks up two values.

So, the total complexity is O(n).

Alternatively, we can think of it as doing a graph search on a directed acyclic graph, with only n vertices and 2(n-1) edges.



Bottom-up computation

In this problem, there is a very simple program that solves the problem in O(n) time.

F(1) = F(2) = 1

for $3 \le i \le n$ do

F(i) = F(i-1) + F(i-2) .

It is clear that it solves the problem in O(n) time (assuming the additions can be done using word operations, not such a realistic assumption as the values grow exponentially).

Dynamic programming

So, this is dynamic programming. to store the intermediate values so that we don't need to compute it again.

From the top-down approach. it should be clear that if we write a recursion with only polynomially many subproblems with polynomial time additional processing, then the problem can be solved in polynomial time.

With this viewpoint, designing an efficient dynamic programming algorithm amounts to coming up with a recurrence relation with only polynomially many subproblems.

In this sense, designing a dynamic programming algorithm is similar to designing a divide-and-conquer algorithm, just coming up with the right recurrence.

Of course, it requires some skills to write a good recurrence to solve the problems, and this is what we will focus on in the many examples to follow.

The bottom-up implementation is usually preferred in practice as it is more efficient.

In some cases, it is easy to translate a top-down solution into a bottom-up implementation.

In some cases, however, it requires some effort and clear thinking to find a correct ordering to compute the subproblems, especially when the recurrence relation is complicated.

Our main focus is to come up with the right recurrence, as that would already imply an efficient algorithm.

We will also mention the bottom up implementations as much as possible.

Weighted interval scheduling [KT 6.1]

Input: n intervals $[s_1, f_1], \ldots, [s_n, f_n]$ with weights $w_1, \ldots, w_n$.

Output: a subset S of disjoint intervals that maximizes $\sum_{i \in S} w_i$.

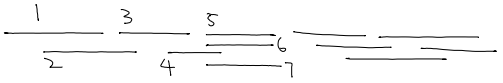
This is a generalization of the interval scheduling problem we saw in greedy algorithms.

We can think of the intervals are requests to book our room from time s_i to f_i, and the i-th request is willing to pay w_i dollars if the request is accepted.

Then, our objective is to maximize our income, while guaranteeing that there are no time conflicts for the accepted requests.

There are no known greedy algorithms for this problem.

Recursion

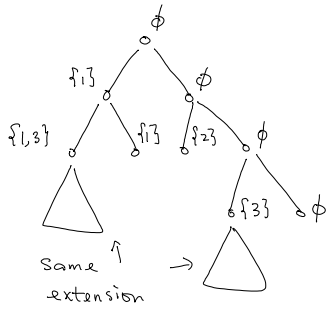


We start by thinking of an exhaustive search algorithm.

We either choose interval 1 or not.

If we choose interval 1, then we can not choose interval 2 and we need to make a decision for interval 3. If we don't choose interval 1, we make a decision for interval 2.

Doing this recursively, we have a search tree as shown, →



This is an exponential time algorithm,

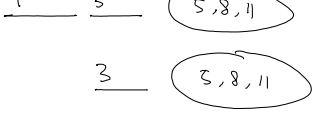
but observe that there are lots of redundancies.

For example, in the branches of {1,3} and {3}, the subtrees extending these solutions are exactly the same.

This is because to determine how to extend these partial solutions,

what really matters is the last interval in the partial solution,

but not anything on the left, since these won't interact with anything on the right of 3.



Therefore, we just need to keep track of the "boundary" of the partial solution -

and this reduces the number of subproblems dramatically.

Better recurrence

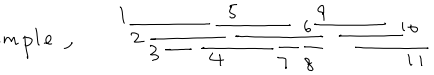
To facilitate our discussion, we find a good ordering of the intervals and pre-compute some useful information.

We sort the intervals by starting time so that $s_1 \le s_2 \le \dots \le s_n$.

For each interval i, we use next[i] to denote the smallest j such that $j > i$ and $f_i < s_j$,

i.e. the first interval on the right of interval i that is not overlapping with interval i.

If no such intervals exist, next[i] is defined as $n+1$ (or infinity).

In this example,  , next[1] = 5, next[2] = 6, next[3] = 4, next[9] = 12, etc.

Now, we are ready to write a recurrence with only n subproblems!

Let opt(i) = maximum income that we can earn using the intervals from i to n only.

Then opt(1) is the optimal value that we want to compute.

To compute opt(1), we have two options:

① if we choose interval 1, then we earn w_1 dollars, but then we cannot choose intervals 2, 3, 4. But we can earn opt(5) = opt(next[1]) from the optimal solutions

using intervals $5-n$ only, since these intervals don't overlap with 1.

So the optimal value using interval 1 is $w_1 + opt(next[1])$.

② if we don't choose interval 1, then we only use intervals from 2 to n, and the optimal value by definition is $opt(2)$.

This recurrence relation is true for every i, and so we have the following recursive algorithm to solve the problem.

Algorithm (top-down weighted interval scheduling)

- sort the intervals by non-decreasing starting time, i.e. $s_1 \le s_2 \le \dots \le s_n$.
- compute $next[i]$ for $1 \le i \le n$.
- compute $opt(1)$.

```
opt(i)    // recursive function
    if i = n+1, return 0.
    opt(i) = max { w_i + opt(next[i]), opt(i+1) }.    // choose i, or not choose i
```

Correctness: It follows from the recurrence relation which we argued previously.

Time complexity: Sorting and the next array can be computed in $O(n \log n)$ time (why?).

After that, using top-down memorization can implement the recursion in $O(n)$ time, since there are only n subproblems and each only needs to look up two values.

Bottom-up implementation: It is simple in this problem.

```
opt(n+1) = 0
for i from n downto 1 do
    opt(i) = max { w_i + opt(next[i]), opt(i+1) }.
```

This is as efficient as the greedy algorithm!

It is essentially an exhaustive search algorithm, but in a much more compact search space!

Exercise: Can you write a program to print out an optimal solution (ie. what intervals to choose)?

Subset-sum and knapsack [kT 6.4]

We consider two related and useful problems.

Subset-sum

Input: n positive integers $a_1, \ldots, a_n$, and an integer K.

Output: a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = K$, or output "No".

For instance, given $1, 3, 10, 12, 14$, whether there is a subset with sum 27? Yes, $\{3, 10, 14\}$.

This problem can be modified to ask for a subset S with $\sum_{i \in S} a_i \leq K$ but maximizes $\sum_{i \in S} a_i$.

That is the version in the textbook and is slightly more general, but once we solve our equality version it will be clear how to solve the inequality version.

Knapsack

Input: n items, each of weight w_i and value v_i, and a positive integer W.

Output: a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \leq W$ that maximizes $\sum_{i \in S} v_i$.

(from cliparthut)

We can think of W as the total weight that the knapsack can hold.

Then, the problem asks us to find a maximum value subset that can fit in the knapsack.

Alternatively, you can think of W as the total time that we have, or our objective is to choose a subset of jobs that can be finished on time while maximizing our income.

Knapsack is more general than subset-sum as there are two parameters to consider, but again it will not be difficult once we know how to solve subset-sum.

So, let's start with subset-sum.

Recurrence

We can start with the exhaustive search algorithm, for the subset-sum problem.

We will consider all possibilities.

Start with the first number a_1. Either we choose it or not.

- If we choose a_1, then we need to choose a subset from $\{2, \ldots, n\}$ so that the sum is $K - a_1$.

- Otherwise, we need to choose a subset from $\{2,\ldots,n\}$ with sum K.

A naive algorithm will consider all subsets, and it will take exponential time.

The point is, we don't really need to keep track of which subset we choose, as long as they have the same sum.

This allows us to reduce the search space efficiently.

The subproblems that we will consider are subsum (i, L) for $1 \le i \le n$ and $L \le K$, where subsum (i, L) returns yes iff there is a subset in $\{i,\ldots,n\}$ with sum L.

The original problem that we like to solve is subsum $(1, K)$.

The recurrence relation is subsum (i,L) = (subsum $(i+1, L-a_i)$ or subsum $(i+1, L)$).

The former case corresponds to choosing a_i, while the latter case corresponds to not choosing a_i.

If either case returns YES, return YES. Otherwise, when both return NO, return NO.

Algorithm

To translate it into a program, we just need to be careful about the boundary cases.

```
subsum (i,L)  // recursive function
    if (L=0)    return YES.
    if (i>n or L<0)    return NO.    // running out of numbers, or sum too large
    return ( subsum(i+1, L-ai) or subsum(i+1, L))
```

Correctness follows from the recurrence relation, whose correctness we have argued before.

Time complexity: There are totally nK subproblems, as there are n choices for i, and k choices for L.

Each subproblem looks up two values.

With top-down memorization, the time complexity is $O(nK)$.

Pseudo-polynomial time: Note that the time complexity is $O(nK)$, which depends on K.

If K is small, this is fast.

But k could be exponential in n (since n bit number can be as big as 2^n), and that would be even slower than exhaustive search.

We call this type of time complexity pseudo-polynomial.

This is probably unavoidable, as we will see that this problem is NP-hard.

Implementations

Bottom-up implementation

We can use a 2D-array answer[n][K] to store the values of all subproblems subsum.

We can compute these values in reverse order from n to 1.

answer[i][L] = NO for all $1 \le i \le n$ and $0 \le L \le K$. // initialization

answer[n][a_n] = answer[n][0] = YES // the YES case of the size 1 problem, i.e. only a_n.

answer[i][0] = YES for all $1 \le i \le n$ // the YES case when the target is zero

for i from n-1 down to 1 do

  for L from 1 to K do

    if answer[i+1][L] = YES, then answer[i][L] = YES.

    if ($L - a_i \ge 0$ and answer[i+1][$L-a_i$] = YES), then answer[i][L] = YES.

A space efficient implementation

With this bottom up implementation. we see that we don't really need to open an n×K array, because when we are computing answer[i][*] in the outer for-loop, we just need the values in answer[i+1][*], and so we can throw away all the values answer[$\ge$i+2][*].

Therefore, we can implement the algorithm with a 2×K array, reducing the space significantly.

Top-down vs bottom-up

The bottom-up implementations don't need recursions, and also leads to a space-efficient algorithm.

But the runtime is always exactly nK, to fill up the whole 2D-array.

If there is a solution, the top-down approach may run faster, since it may be able to find a solution by only solving a few subproblems.

By the way, in this sense, it may make a big difference by solving Subsum(i+1, L-a_i) before Subsum(i+1, L). Do you see why?

Tracing a solution

By following a path of YES from Subsum(1, K), we can find a subset with sum K:

- If Subsum(2, K-a_1) = YES, then put a_1 in our solution and recurse.
- Otherwise, don't put a_1 in the solution and follow a YES-path from Subsum(2, K).

Dynamic programming and graph search

If you think about it, dynamic programming is to reduce the search space to polynomial size.

Each state is a vertex, and the edges are added according to the recurrence relation.

And then the problem is to determine whether the starting state Subsum(1,K) can reach YES.

And the top-down memorization is like doing DFS and mark the states visited.

Knapsack

We outline how to solve the knapsack problem, which is quite similar to solving subset-sum.

We won't start from the naive exhaustive search again from now on.

Using a similar recurrence as in subset-sum, the first recurrence relation may involve three parameters knapsack(i,W,V), which asks whether it is possible to find a subset in $\{i,\dots,n\}$ with total weight W and total value V.

With the whole 3D-table, it should be clear that we can solve the knapsack problem.

It is possible to just use 2 parameters as in subset-sum.

The idea is to use the function value for one parameter.

Define Knapsack(i,W) as the maximum total value that we can earn using items in $\{i,\dots,n\}$ with total weight at most W.

More precisely, let $\text{Knapsack}(i,W) = \max_{S \subseteq \{i,\dots,n\}} \left\{ \sum_{i \in S} v_i \;\middle|\; \sum_{i \in S} w_i \le W \right\}$.

Then, we can write the recurrence relation:

$$\text{Knapsack}(i,W) = \max \left\{ v_i + \text{knapsack}(i+1, W-w_i),\ \text{Knapsack}(i+1, W) \right\}$$

The first case corresponds to choosing item i, thus earning v_i and the optimal profit of using items from i+2 to n, and the total weight left in the knapsack is $W-w_i$.

The second case corresponds to not choosing item i.

With this recurrence relation, it is not difficult to complete the algorithm and the analysis as in the subset-sum problem.

Just need to be careful with the boundary values: if $W<0$, return $-\infty$; if $i>n$, return 0.

We leave the rest as an exercise. Go through the discussion for subset-sum again.

The time complexity is $O(nW)$.
