CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 11: Dynamic Programming

We introduce the technique of dynamic programming through some well-known examples.

Introduction

On a high level, we can solve a problem by dynamic programming if there is a recurrence relation with only a small number of subproblems (i.e. polynomially many).

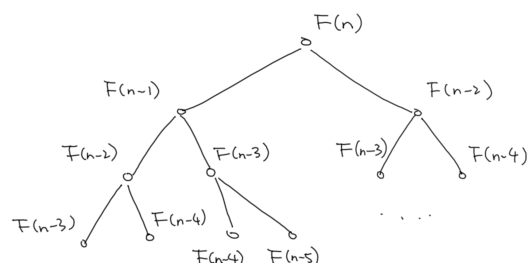
This is a general and powerful technique, and also simple to use once we learnt it well.

To illustrate the idea using a toy example, consider the problem of computing the Fibonacci sequence

$$F(n) = F(n-1) + F(n-2) \quad ; \qquad F(1) = F(2) = 1.$$

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 (MATH 239), then we find out the runtime of this algorithm is $\Theta(1.618^n)$.

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

There are only n subproblems. Why do we waste so much time.

There are two approaches to solve the problem efficiently.

Top-Down Memorization

As in BFS / DFS, we use an array visited[i] to ensure that we only compute each subproblem at most once, and we use an array answer[i] to store the value F(i) for future lookup.

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

F(i)    // recursive function
    if visited[i] = true, return answer[i].
    if i=1 or i=2, return 1.
    answer[i] = F(i-1) + F(i-2).
    visited[i] = true.
    return answer[i].
```

Time Complexity Each subproblem is only computed once, when visited[i] = false.

When it is computed, it looks up two values. So, the total time 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

For Fibonacci sequence, there is a straightforward algorithm to solve 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 this solves the problem in $O(n)$ additions.

(Note that the values grow exponentially in n, so we cannot assume that each addition can be done in $O(1)$ time.)

Dynamic Programming

So, basically, this is the framework of 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 additional polynomial time processing, then the problem can be solved in polynomial time.

In this viewpoint, designing an efficient dynamic programming algorithm amounts to coming up with a nice recursion.

This is similar to what we have done in designing divide and conquer algorithms, coming up with a right recursion.

We will see that many interesting and seemingly difficult problems have a nice recurrence relation.

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

In practice, the bottom-up implementation is preferred as it is non-recursive and usually more efficient.

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

In some cases, however, it requires clear thinking to find a correct ordering to compute the subproblems, especially when the recurrence relation is complicated, although in principle we just need a topological ordering.

Our main focus will be 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 $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$ with weights $w_1, w_2, \ldots, w_n \in \mathbb{R}$.

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

This is a generalization of the interval scheduling problem in L08, when $w_i = 1 \; \forall i$.

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.

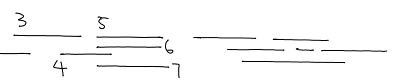
Then, our objective is to maximize our income, while ensuring that there are no conflicts for the accepted requests.

Unlike the special case when $w_i = 1 \; \forall i$, there are no known greedy algorithms for this problem.

Exhaustive Search

To come up with a good recursion for this problem, we start by running an exhaustive search algorithm

and see why it is wasteful and how to improve it.

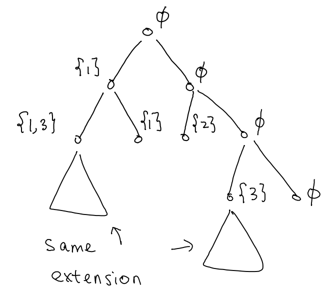


First, we make a decision on interval 1, either choose it or not.

If we choose interval 1, then we cannot choose interval 2, and we need to make a decision on interval 3.

If we do not choose interval 1, then we need to make a decision on interval 2.

Doing this recursively, we have a recursion tree as shown.



This is an exponential time algorithm, but observe a lot of redundency.

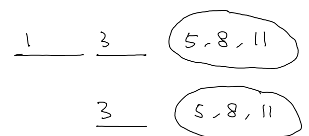
For example, in the branches of $\{1,3\}$ and $\{3\}$,

the subtrees extending these partial solutions are exactly the same.

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

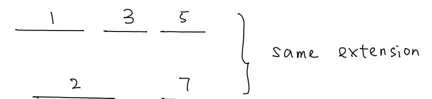
what really matter is the last interval of the current partial solution,

but not anything on the left, since they won't interact with anything on the right.



So, we just need to keep track of the "boundary" of the solution.

In this problem, the boundary is simply the last interval.



This suggests that there should be a recursion with only 1 parameter!

Better Recurrence

To facilitate the algorithm description, we use a good ordering of the intervals and pre-compute useful information.

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

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 $n+1$, representing the end.

In this example, [diagram of intervals 1–11], next[1] = 5, next[2] = 6, next[3] = 4, next[9] = 12, etc.

Since we sort the intervals by non-decreasing starting time, for an interval i, next[i] has the property that intervals $\{i, \ldots, next[i]-1\}$ overlap with interval i while the intervals $\{next[i], \ldots, n\}$ are disjoint from interval i.

Now, we are ready to write a recursion with only one parameter, resulting in only n subproblems!

Let $opt(i)$:= maximum income that we can earn using the intervals in $\{i, i+1, \ldots, n\}$ only.

Then $opt(1)$ is the optimal value that we would like to compute.

To compute $opt(1)$, there are only two options for the solutions:

① The solutions that choose interval 1.

For these solutions, we earn w_1 dollars by choosing interval 1.

But then we cannot choose the intervals in $\{2, \ldots, next[1]-1\}$ since these intervals overlap with interval 1.

So, to find the optimal value of choosing interval 1, we need to find an optimal way to choose from $\{next[1], \ldots, n\}$.

Therefore, the optimal value for solutions choosing interval 1 is $w_1 + opt(next[1])$.

② The solutions that don't choose interval 1.

Then, by definition of $opt(2)$, the optimal value for solutions not choosing interval 1 is $opt(2)$.

Combining the two cases, we get that $opt(1) = \max\{w_1 + opt(next[1]),\ opt(2)\}$.

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

<u>Algorithm</u> (top-down weighted interval scheduling)

1. Sort the intervals by non-decreasing starting time so that $s_1 \le s_2 \le \ldots \le s_n$.
2. Compute next[i] for $1 \le i \le n$. Set visited[i] = false for $1 \le i \le n$.
3. Return opt(1)

```
opt(i)    // recursive function

    if i = n+1, return 0.          // base case

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

    answer[i] = max { w_i + opt(next[i]), opt(i+1) }.

    visited[i] = true.

    return answer[i].
```

<u>Correctness</u> The correctness of the algorithm follows from the explanation of the recurrence relation above.

<u>Time complexity</u> Sorting and the next[] array can be computed in $O(n\log n)$ time. (why?)

After that, the top-down memorization implements the recursion in $O(n)$ time, since

there are only n subproblems and each subproblem only needs to look up two values.

<u>Bottom-up implementation</u> It is simple in this problem.

$opt(n+1) = 0$

for i from n downto 1 do

$opt(i) = \max\{w_i + opt(next[i]),\ opt(i+1)\}$.

Quite surprisingly, this has the same time complexity as the greedy algorithm!

That is, somehow we can implement an exhaustive search algorithm as efficient as the greedy algorithm!

We can see why dynamic programming is general and powerful, because it is very systematic

(so that we may not need problem specific insight) and yet we get very competitive algorithms.

<u>Exercise</u>: Write a program to print out an optimal solution (i.e. which intervals to choose).

Subset-Sum and Knapsack Problem [KT 6.4]

We consider two related and useful problems.

<u>Subset-Sum</u>

Input: n positive integers $a_1, a_2, \ldots, a_n$, and an integer K.

Output: a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = K$, or report that no such subset exists.

For example, given 1, 3, 10, 12, 14, is there a subset with sum 27? Yes, {3, 10, 14}. sum 29? No.

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

This is the version in [KT] and is slightly more general, but once we solve our equality version

it should be clear how to solve the inequality version as well.

<u>Knapsack</u>

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 clipartnut)

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

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

Alternatively, we can think of W as the total time that we have, and 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 solved subset-sum. So, let's start with subset-sum.

Recurrence

To come up with the recurrence, we start with the exhaustive search algorithm for the subset-sum problem.
We will consider all possibilities.
Start with the first number a_1. Then, 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, if we don't choose a_1, then we need to choose a subset from $\{2,\ldots,n\}$ so that the sum is K.

A naive implementation will consider all subsets, and this gives an exponential time algorit

The observation is, we don't really need to keep track of which subset we have chosen so far, as long as they have the same sum.
This suggests that we can just keep track of the (partial) sum in the recursion, which allows us to reduce the search space significantly.

The subproblems that we will consider are $\text{subsum}[i,L]$ for $1 \le i \le n$ and $L \le K$, where $\text{subsum}[i,L]$ returns true if and only if there is a subset in $\{i,\ldots,n\}$ with sum L.
Then the original problem that we would like to solve is $\text{subsum}[1,K]$.
The recurrence relation is $\text{subsum}[i,L] = (\text{subsum}[i+1, L-a_i] \text{ OR } \text{subsum}[i+1, L])$.
The former case corresponds to choosing a_i, while the latter case corresponds to not choosing a_i.
If either case returns true, then return true. Otherwise, when both return false, return false.

Algorithm (top-down subset-sum)

From the recurrence relation to a correct algorithm, we just need to be careful about the base cases.

```
subsum(i, L)          // recursive function
    if (L=0)          return true.
    if (i>n or L<0)   return false      // running out of numbers, or partial sum too large.
    return ( subsum(i+1, L-a_i) OR subsum(i+1, L) ).
```

Correctness follows from the recurrence relation explained above.

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

Each subproblem can be solved by looking up two values.

Using top-down memorization, the time complexity is $O(nk)$.

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

When k is small, this is fast.

But k could be exponential in n (as n-bit number can be as big as 2^n), in which case this is even slower than the naive exhaustive search (and also uses much more space).

We call this type of time complexity pseudo-polynomial.

This is probably unavoidable, as we will see that the subset-sum problem is NP-complete.

Implementations

Bottom-up computation

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

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

```
subsum[i][L] = false   for all 1≤i≤n and 0≤L≤K      // initialization
subsum[n][a_n] = subsum[n][0] = true     // the YES case for the size 1 problem where we only have a_n.
subsum[i][0] = true   for all 1≤i≤n      // the YES case when the target is zero.
for i from n downto 1 do
    for L from 1 to K do
        if subsum[i+1][L] = true, then subsum[i][L] = true.
        if (L-a_i ≥ 0 and subsum[i+1][L-a_i] = true), then subsum[i][L] = true.
```

Space-Efficient Implementation

With this bottom-up implementation, we see that we don't need to use a $n \times k$ array.

When we are computing subsum[i][*] in the outer for-loop, we just use the values of subsum[i+1][*].

So, we can throw away the values subsum[≥i+2][*] and only use a $2 \times k$ array, a significant saving.

Top-Down vs Bottom-Up

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

But the run-time is always exactly nk, to compute all the subproblems.

When there is a solution, the top-down approach may run faster, as it may be able to find

a Solution by solving only a few subproblems.

From this perspective, it may make a difference by solving subsum$(i+1, L-a_i)$ before subsum$(i+1, L)$. (why?)

Tracing a Solution

By following a path of "true" from subsum$(1,K)$, we can find a subset of sum K.

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

Dynamic Programming and Graph Search

Dynamic programming reduces the search space to polynomial size. We can draw a graph.

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

That is, there is a direct edge from (i,L) to $(i+1,L)$ and from (i,L) to $(i+1, L-a_i)$ if $L-a_i \geq 0$.

Then the problem is equivalent to determining whether there is a directed path from the starting state $(1,K)$ to a "true" "base state" (e.g. $(i,0)$ for some i).

The way the top-down memorization algorithm works is basically doing a DFS on this "subproblem graph", by recursing and marking subproblems visited.

Knapsack (Sketch)

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

From now on, we won't start from the exhaustive search again.

First Approach

We use a similar recurrence as in subset-sum.

The subproblems are knapsack(i,W,V), which is true if and only if there is a subset in $\{i, i+1, \ldots, n\}$ with total weight W and total value V.

With this 3D-table, you should be able to write a recurrence as in subset-sum to solve the problem.

Better recurrence

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

Note that some subproblems dominate other subproblems, e.g. if knapsack$(i,W,V+1)$ = true, then we can ignore the subproblem knapsack(i,W,V).

So, the idea is to only have subproblems knapsack(i,W) and keep track of the max value achievable.

Define knapsack(i,W) as the maximum value that we can earn using items in $\{i,\ldots,n\}$ with total weight $\leq W$.

More precisely, let knapsack$(i,W) = \max_{S \subseteq \{i,\ldots,n\}} \{ \sum_{j\in S} v_j \mid \sum_{j \in S} w_j \leq W \}$.

More precisely, let $knapsack(i, W) = \max_{S \subseteq \{i, \ldots, n\}} \{ \sum_{j \in S} v_j \mid \sum_{j \in S} w_j \leq W \}$.

Then, the recurrence relation is $knapsack(i, W) = \max\{ v_i + knapsack(i+1, W - w_i),\ knapsack(i+1, W) \}$.

The first case corresponds to choosing item i, thus earning v_i and the maximum value of using items from $i+1$ to n when the capacity 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 subset-sum.

Just need to be careful in the base cases: when $W < 0$, return $-\infty$; when $i > n$, return 0.

We leave the details to the reader. Go through the subset-sum section again if necessary.

The time complexity is $O(nW)$.