You're a lumberjack, and it's OK.

Johan Montelius

Spring Term 2021

Introduction

This is an exercise in how to solve a problem using dynamic programming. You should know how dynamic programming works before taking on this assignment.

As with all dynamic programming examples the problem is a bit artificial but shows well the difference between a naive solution and a dynamic programming solution. As often there might be an even smarter way of solving the problem so we will see if we can find one.

If you search for dynamic programming problems you will find several where the scenario is that you need to cut a log in smaller pieces. The problem might be to maximize profit or to reduce some cost. In the problem that I have chosen our task is to reduce the cost.

Assume that your the manager of a timber yard and you have orders on logs of different length. Cutting a log in two pieces induces a cost and strangle enough this cost is proportional to the length of a log. It does not matter where you cut the log, the cost is the same.

Now take for example that you have a log of 6 meters and you should cut it up in three pieces: 1, 2 and 3 meters, how are you going to cut it? You could of course start by cutting it in two pieces, 1 and 5 meters, and then cut the larger log in two but then the total cost would be 6 plus5. A better strategy is to cut the log in half, 3 and 3 meters, and then take one piece and cut it one of them in two. Now the cost is only 6 plus 3 i.e. 9.

Your task is to given a sequence of lengths, for example `[1,2,3,2,1]` (the order does not matter), find the best way to cut a log (that has the length of the sum) to minimize the cost. Even in this small example it takes some pondering before one finds the solution (give it a try with pen and paper, the minimum cost is 20).

The recursive solution

The recursive solution to this problem goes as follows: divide the sequence in two, cut the log in two to match the length of the two sequences and

then minimize the cost of cutting up the two logs. If we take the sequence `[1,2,3,2,1]` we can divide it into `[1,2]` and `[3,2,1]`. The minimum cost of cutting a log into `[1,2]` is of course 3 and cutting a log into `[3,2,1]` is 9 so the total cost is $3 + 9 + 9 = 21$.

Hmm, if we cut the large log into two,4 and 5 meters, then we can cut two two logs into `[1,2,1]` and `[2,3]`. The cost would then be only $6 + 5 + 9 = 20$ i.e. less than in our first attempt. In order to be sure that this is the minimum cost we would have to try all possible ways of cutting the large log in two and there are plenty of alternatives.

0.1 Split a sequence

Let's start by defining a function that will split a sequence in all possible ways. We will return a tuple of the split we come up with and the total length of the log that we're cutting (we will need this later). Why not try a tail recursive function were we take a sequence and place the first element of the sequence either in the left pile or the right pile.

```
def split(seq) do split(seq, ..., ..., ...) end

def split([], l, left, right)  do
  [...]
end
def split([s|rest], l, left, right) do
split(rest, ..., ..., ...) ++
  split(rest, ..., ..., ...)
end
```

You notice here that I'm using "++" and that might be costly but trust me, this is the least problem we will have in this implementation.

Give it a try and split the sequence `[1,2,3]` into all possible combinations. What would happen if we recursively tried to split the sub-sequences into two parts? What is the problem when we start working on the alternative {`[],[3,2,1], 6`}?

We must prevent our split function to produce an alternative with an empty sequence in on part so we add two special cases.

```
def split(seq) do split(seq, ..., ..., ...) end

def split([], l, left, right)  do
  [...]
end
def split([s], l, [], right) do
  ...
end
```

```
def split([s], l, left, []) do
  ...
end
def split([s|rest], l, left, right) do
split(rest, ..., ..., ...) ++
  split(rest, ..., ..., ...)
end
```

We still have some redundancy since we produce solutions that for our problem are redundant. If we divide a sequence into {`[3],[2,1], 6`}? or {`[2,1],[3], 6`} does not matter for our problem. We should be able to cut the number of alternatives in half but we can wait with this until we have a first solution to our problem.

Adding up the cost

Let's now instead of generating a list of all possible ways of splitting a sequence, instead calculated the minimum cost. Make a copy of your split function and call it `cost` and make some changes to it to make it calculate the cost.

If you have divided your sequence, with a sum of `l`, into two parts, `left` and `right`, what is then the minimum cost? First of all you need to cut the log, of length `l`, into two parts and then you need to find the minimum cost of cutting a log in the parts described in the left sequence and in the right sequence.

Since we're now making a recursive call we need to make sure that we're not ending up in an endless loop. We add the following clauses for `cost/1` (the first case will never happen if you don't call cost with an empty sequence to start with).

```
def cost([]) do ... end
def cost([_]) do ... end
def cost(seq) do cost(seq, ..., ..., ...) end
```

Once we know that we can handle sequences of length one without ending up in a loop we adapt the rest of the code. Let's look at the base cases:

```
def cost([], l, left, right)  do
 ... + ... + ...
end
def cost([s], l, [], right)  do
  ... + ... + ...
end
def cost([s], l, left, [])  do
  ... + ... + ...
end
```

Then we are ready to do explore the two alternatives. We calculate the minimum cost of the alternatives and then select the smallest.

```
def cost([s|rest], l, left, right) do
  ... = cost(rest, l+s, [s|left], right)
  ... = cost(rest, l+s, left, [s|right])
  if ... do
    ...
  else
    ...
  end
end
```

You can now calculate the minimum cost even if you don't present exactly how the log is to be cut up. We can change the cost function to return a tuple {`cost, tree`} where `cost` is the minimum cost as before and `tree` is a representation of how to do the cutting. If you get the base cases right you could end up with the following solution:

```
 > Lumber.cost([1,2,3,2,1])
{20, {{3, 2}, {2, {1, 1}}}}
```

The real problem

So now that you think that you have solved the problem we should face the real problem. Try the following problems:

- `cost([1,2,3,4,5,6])`
- `cost([1,2,3,4,5,6,7,8])`
- `cost([1,2,3,4,5,6,7,8,9,10])` (don't hold your breath)

Things do take time! Try the following benchmark; start with `bench(10)` and ponder what this means. If it's now past midnight you can give `bench(12)` a try and go to bed but my guess is that it will not be done by the time you wake up (try to estimate how long time it would take).

```
def bench(n) do
  for i <- 1..n do
    {t,_} = :timer.tc(fn() -> cost(Enum.to_list(1..i)) end)
    IO.puts(" n = #{i}\t t = #{t} us")
  end
end
```

Hmm, you're using your laptop and you have problems figuring out how to cut up a log in twelve pieces - would things change if you had access to a super computer.

The solution

This is where "dynamic programming" comes to our rescue; we should memorize solutions to sub-problems that we have seen before. The search tree is filled with sub-problems that we solve over and over again, if re-computation can be avoided things will change dramatically.

The only thing wee need to solve is how to represent a problem and how to adapt our cost function to update and make use a memory. The first problem is not trivial but why not pick the simplest possible solution: represent the problem of dividing a log into the segments `[1,2,3]` by the list `[1,2,3]`. This is not a perfect solution since the problem `[1,2,3]` is identical to `[3,1,2]` but we will treat them as two different problems. We ignore this problem for the time being and create a memory using maps.

```
defmodule Memo do

  def new() do %{} end

  def add(mem, key, val) do Map.put(mem, key, val) end

  def lookup(mem, key) do  Map.get(mem, key) end

end
```

Now the tricky part. You should change the cost function so it takes a memory and returns not only the solution but an updated memory. Let's first define a function that takes care of the original call. We here check that we actually have a sequence and if so we call the function `cost/2` that uses the memory. The function will return an updated memory but we're only interested in the cost and the tree.

```
  def cost([]) do {0, :na} end
  def cost(seq) do
    {cost, tree, _} = cost(seq, Memo.new())
    {cost, tree}
  end
```

Now we have the function `cost/2` that also have a base clause and a general clause. Note that we will never call with an empty sequence but need to handle the case where we only have one element to avoid an infinite loop. In the general case we call `cost/5` and then adds this solution to the memory. This is the only place where we add something to the memory.

```
  def cost([s], mem) do {0, s, mem} end
  def cost(seq, mem) do
```

```
    {c, t, mem} = cost(seq, 0, [], [], mem)
    {c, t, Memo.add(mem, seq, {c, t})}
  end
```

We now define a function `check/2` that we will use instead of calling `cost/2` directly. This is where we first check if we have been in this position before and if so use the solution that we have. Note that we should return not only the solution but also the memory.

```
def check(seq,  mem) do
  case Memo.lookup(mem, seq) do
    nil ->
      cost(seq, mem)
    {c, t} ->
      {c, t, mem}
  end
end
```

Now for you to complete `cost/5`. This function will look very much as before but you need to "thread" the memory argument through the code. Note, instead of making a recursive call to `cost/2` you make a call to `check/2` to first check if there is a solution for already.

Do some small tests and if things look like they are working try `bench(15)`. We have certainly made progress but it looks like calculating the result of a sequence of twenty pieces would still take some hours. Can we do better?

Better

There are two things that we have skipped for later and this is probably the point where we need to think about these issues. The first one was the fact that we are still exploring mirror solutions We should be able to cut the execution time in half if we can avoid mirror images. The second thing was the key we use in the memory. If we have a solution for `[1,2,3]` we will not find this if we are looking for a solution for `[2,3,1]`. As sequences gets longer this becomes a huge problem. How many permutations are there for a sequence of n elements?

Let's start by cutting the execution in half by avoiding mirror computations. There is a simple solution to this problem even if it requires some coding. The trick is to place the first element in a sequence in the left pile explicitly. This does gives us a better starting point but we are still far from solving problems of size 20.

```
def cost([s], mem) do {0, s, mem}end
def cost([s|rest]=seq, mem) do
  {c, t, mem} = cost(rest, s, [s], [], mem)
```

```
    {c, t, Memo.add(mem, seq, {c, t})}
  end
```

How about the second problem; could we find a unique key for two sequences that should be treated as the same? Could it be as simple as ordering the sequence? Could we order the sequence at start and then make sure that sub-sequences are order (we need to reverse the accumulated lists before doing a recursive call).

n	execution time	increase
12	0.085 s	-
13	0.27 s	3.1
14	0.87 s	3.3
15	4.5 s	5.2
16	20 s	4.3
17	78 s	4.0
18	270 s	3.5

Table 1: Execution time using lists as keys.

As you see we still have a exponential component in the execution time. As we increase the number of segments the execution time increase by a factor three to five. A good estimate is that a sequence of length 20 would take more than an hour.

Using maps is of course very convenient but the data structure can not do miracles. We are using a list of integers as a key and this does of course make the searching a bit more complicated. We might help the map functions by first converting our list of integers to something more convenient. If we assume that all integers are in the range from 0 to 255 we could turn a list into a "binary" before using it as a key.

```
def add(mem, key, val) do
   Map.put(mem, :binary.list_to_bin(key), val)
end

def lookup(mem, key) do
   Map.get(mem, :binary.list_to_bin(key))
end
```

This does improve the execution speed, at least for the larger benchmarks.

Could we design our own memory module that will outperform the maps that we use now? The key that we use is a list of integers, what would happen if we create a tree where each node is represented by a tuple {`n, value, branches`} where branches is a list of nodes. A tree with values for the keys `[2]`, `[1,2]`, `[1,3]` and `[1,2,3]` would look like follows:

n	execution time	increase
12	0.10 s	-
13	0.34 s	3.3
14	0.99 s	2.9
15	3.5 s	3.5
16	13 s	3.7
17	45 s	3.5
18	160 s	3.6

Table 2: Execution time using a binary as key.

```
[ {1, nil, [ {2, nil, [{3, :onetwothree, []}]},
             {3, :onethree, []}
           ]}
  {2, :two, []}
```

It's a fun exercise to implement the `add/3` and `lookup/2` functions for this tree but it does pay of. The improvement is not dramatic but it's fun to implement something that does better than maps.

n	execution time	increase
12	0.17 s	-
13	0.35 s	3.0
14	1.1 s	3.2
15	3.7 s	3.4
16	12 s	3.2
17	40 s	3.4
18	120 s	3.2

Table 3: Execution time using a dedicated memory.

We could probably do better since the tree we create is not well balanced. The root of the tree consist of a list of branches but the branch that starts with 1 is of course much longer than the branch that starts with 20 (since the numbers in a key are ordered). We could also represent the list of branches as trees but it will not make any dramatic improvements.

Could it be that this is the end; we can solve problems up to a sequence of twenty-plus elements but a sequence of thirty is out of reach?

Some analysis

We obviously still have an exponential component in our algorithm and before we go further we might ponder where this is coming from. Our

program consists of two modules, the cost search module and the memory module. To identify where the exponential component comes from we can look at these modules one by one. In the analysis n represents the length of the given sequence.

the memory module

The memory module will store values under keys and the keys are lists of n numbers. Some keys are of course short and some longer but no keys are longer than n. If we double the length of the sequence the keys will double in length.

In our dedicated memory module the add and lookup operations have time complexity $O(n)$. For each number in the key you will have to search through a list of at most n branches and you have n numbers in your key. We know that since keys are ordered the root could have n branches but the level below at most have $n - 1$ branches but this does not change the asymptotic run-time complexity.

We do however here make an assumption that might not always hold and our benchmarks is actually increasing two things when it increases n. We have a benchmark where a sequence of n numbers is made up by n different numbers. What would change of we instead choose n number from a set of k numbers? Let's make an experiment as see how the execution time changes when we have a sequence of 20 elements but change the k value.

k	**execution time**
2	1.0 s
5	6.9 s
10	78 s

Table 4: Execution time for a sequence of 20 elements chosen from k values.

If we hold k fixed at for example 5 the complexity of our memory module changes to $O(n)$. This will of course have a huge impact as n grows. Let's see what it looks like when we choose the sequence from only five elements.

A dramatic change in execution time but as you see we still have an exponential component.

the search module

The search module is of course a beast. The naive implementation had obvious redundancies in the computation and our hope was to reduce this so that we would end up with a polynomial complexity. So far we have failed but if we take a closer look at the search module we might kill the beast. To start with we will take a closer look at the problem we have.

n	execution time	increase
12	0.015 s	-
13	0.035 s	2.3
14	0.075 s	2.1
15	0.16 s	2.1
16	0.33 s	2.1
17	0.71 s	2.1
18	1.54 s	2.2
19	3.5 s	2.2
20	7.1 s	2.1

Table 5: Execution time for a sequence of 20 elements chosen from five values.

We're given a sequence of n numbers and there are k different numbers. If we construct all possible combinations from this sequence we can construct 2^n sequences by either including or excluding a number in the sequence. Many of these sequences are however equal and it of course depends on the k value how many unique sequences we have.

In our benchmark where we have a sequence of 20 elements with four elements each from 1 to 5 the number of unique sequences is only 3119 You can create any sequence by first selecting from zero to four ones, then zero to four twos etc. This gives you 5^5 sequences but then you also count the single number sequences and the empty sequence $((k+1)^{n/k} - k - 1)$

So the number of sequences that we need to evaluate the answer for is rather small. But since each sequence requires so many lookup operations we're close to stuck.

Take for example the sequence `[1,2,3,4]`. To find the lowest cost we need to lookup the cost for: `[1,2]`, `[1,3]`, `[1,4]`, `[2,3]`, `[2,4]`, `[3,4]`, `[1,2,3]`, `[1,2,4]`, `[1,3,4]` and `[2,3,4]` (not counting the single number sequences). There is not much we can do about this but this will not be the typical solution.

If we're given the sequence `[1,1,2,2]` our search algorithm will create the following sequences: `[1,1]`, `[1,2]`, `[1,2]`, `[1,2]`, `[1,2]`, `[2,2]`, `[1,1,2]`, `[1,1,2]`, `[1,2,2]` and `[1,2,2]`. It will only realize that there are duplicates once it does a lookup in memory and finds an already computed solution.

If we could avoid generating the duplicates and still compute the minimum cost for the sequence much would be gained. If we represent a sequence as `[2,1, 2,2]` meaning that there are two ones followed by two twos. We would then quite easily be able to generate the sequences: `[2,1]`, `[1,1, 1,2]`, `[2,2]`, `[2,1, 1,2]`, and `[1,1, 2,2]`.

This is an idea that is left for the reader to explore. I have not tried it,

but I think its doable and that it would pay of. We could also try something completely different.

Huffman

So we are minimizing the cost of cutting logs and the longer the log the more it costs. No one in their right mind would take a long log and cut of the pieces one by one starting with the smallest. A rough guess would be to divide the sequence int roughly equal parts and the cut the log more or less o the middle. This way we at least get value for the expensive cut.

Can we order the sequences in two parts that are not only roughly equal but as equal as can be? Have we seen a similar problem before? Can you construct a tree with more or less equal weights by starting with the smallest segments and work your way up?

Does this solve the problem?