Lecture 22: Memoization

Even when programming in a functional style, $O(1)$ mutable map abstractions like arrays and hash tables can be extremely useful. One important use of hash tables is for ***memoization***, in which a previously computed result is stored in the table and retrieved later. Memoization is a powerful technique for building efficient algorithms, especially in a functional language.

For example, consider the problem of computing the *n*-th Fibonacci number, defined as $f(n) = f(n-1) + f(n-2)$. We can translate this directly into code:

```
let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Unfortunately, this code takes exponential time: $\Theta(\varphi^n)$, where φ is the golden ratio, (1 + √5)/2 ≈ 1.618... We can easily verify these asymptotic bounds by using the substitution method. Using the recurrence $T(n) = T(n-1) + T(n-2) + 1$, we can show by induction that $T(n) \leq \varphi^n - 1$:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\leq k\varphi^{n-1} - 1 + k\varphi^{n-2} - 1 + 1 \\ &\leq k\varphi^{n-1} + k\varphi^{n-2} - 1 \end{aligned}$$

But φ has the property that $\varphi^2 = \varphi + 1$, so:

$$\begin{aligned} & k\varphi^{n-1} + k\varphi^{n-2} \\ &= k\varphi^{n-2}\,(1 + \varphi) \\ &= k\varphi^{n-2}\,\varphi^2 = k\varphi^n \end{aligned}$$

Therefore $T(n) \leq k\varphi^n - 1$ and T is $O(\varphi^n)$. The Ω direction is shown similarly.

The key observation is that the recursive implementation is inefficient because it recomputes the same Fibonacci numbers over and over again. If we record Fibonacci numbers as they are computed, we can avoid this redundant work. The idea is that whenever we compute `f(n)`, we store it in a table indexed by `n`. In this case the indexing keys are integers, so we can implement this table using an array:

```
(* Straightforward implementation of fib has exponential number of
 * recursive calls. Requires n non-negative *)
let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)

(* However most of these recursive calls share common subproblems -
 * there are only n distinct subproblems to solve, one for each value
 * of n.  The idea of memoization is to keep track of these results so
 * they can be looked up rather than recomputed. *)
let fibm n =
  let memo : int option array = Array.create (n+1) None in
```

```
let rec f_mem n =
  match memo.(n) with
    Some result -> result        (* computed already! *)
  | None ->
      let result = if n < 2 then 1 else f_mem (n-1) + f_mem (n-2) in
      memo.(n) <- Some result;   (* record in table *)
      result
in f_mem n
```

The function `f_mem` defined inside `fibm` contains the original recursive algorithm, except before doing that calculation, it first checks if the result has already been computed and stored in the table, in which case it simply returns the result. How do we analyze the running time of this function? The time spent in a single call to `f_mem` is $O(1)$ if we exclude the time spent in any recursive calls that it happens to make. Now we look for a way to bound the total number of recursive calls by finding some measure of the progress that is being made.

A good choice of progress measure, not only here but also for many uses of memoization, is the number of nonempty entries in the table (i.e. entries that contain `Some` integer value rather than `None`). Each time `f_mem` makes the two recursive calls it also increases the number of nonempty entries by one (filling in a formerly empty entry in the table with a new value). Since the table has only n entries, there can thus only be a total of $O(n)$ calls to `f_mem`, for a total running time of $O(n)$ (because we established above that each call takes $O(1)$ time). This speedup from memoization thus reduces the running time from exponential to linear, a huge change; for instance for $n = 32$ the speedup from memoization is more than a factor of a million.

Memoziation is beneficial when there are common subproblems that are being solved repeatedly. Thus we are able to use some extra storage to save on repeated computation.

Although this code uses imperative constructs (e.g., `Array` assignment), the side effects are not visible outside the function `fibm`. Therefore these are ***benign*** side effects that do not need to be mentioned in the specification of `fibm`.

Party Optimization

Suppose we want to throw a party for a company whose organization chart is a binary tree. Each employee has an associated *fun value*, and we want the set of invited employees to have the maximum total fun, which is the sum of the fun values of the invited employees. However, no one has fun if some employee and that employee's direct superior are both invited, so we never invite two employees who are directly connected in the organization chart. (The less whimsical name for this problem is the ***maximum weight independent set in a tree***.)

There are 2^n possible invitation lists, so the naive algorithm that computes the total fun value of every invitation list takes exponential time.

We can use memoization to turn this into a linear-time algorithm. We start by defining a variant type to represent the employees. The `int` at each node is the fun value.

```
type tree = Empty | Node of int * tree * tree
```

Now, how can we solve this recursively? One important observation is that in any tree, the optimal invitation list that doesn't include the root node will be the union of optimal invitation lists for the left and right subtrees. And the optimal invitation list that does include the root node will be the union of optimal invitation lists for the left and right children that do not include their respective root nodes. So it seems useful to have functions that optimize the invitation lists for the case where the root node is included and for the case where the root node is excluded. We'll call these two functions `party_in` and `party_out`. Then the result of `party` is just the maximum of these two functions:

```
(* Maximum weight independent set in a tree *)
(* AKA the office party optimization problem *)
type tree = Empty | Node of int * tree * tree

(* Returns optimum fun for t *)
let rec party t : int = max (party_in t) (party_out t)

(* Returns optimum fun for t assuming the root node of t is included *)
and party_in t =
  match t with
    Empty -> 0
  | Node (v, left, right) -> v + party_out left + party_out right

(* Returns optimum fun for t assuming the root node of t is excluded *)
and party_out t =
  match t with
    Empty -> 0
  | Node (v, left, right) -> party left + party right
```

This code has exponential performance. But notice that there are only n possible distinct calls to `party`. If we change the code to memoize the results of these calls, the performance will be linear in n. Here is a version that memoizes the result of `party` and also computes the actual invitation lists. Notice that this code memoizes results directly in the tree.

```
(* This version memoizes the optimal fun value for each tree node.
   It also remembers the best invite list. Each tree node has the
   name of the employee as a string. *)
type tree = Empty
  | Node of int * string * tree * tree *
  (int * string list) option ref

let rec party t : int * string list =
  match t with
    Empty -> (0, [])
  | Node (v, name, left, right, memo) ->
      match !memo with
        Some result -> result
      | None ->
          let (infun, innames) = party_in t in
```

```
          let (outfun, outnames) = party_out t in
          let result =
            if infun > outfun then (v + infun, name :: innames)
            else (outfun, outnames) in
          memo := Some result; result

and party_in t =
  match t with
    Empty -> (0, [])
  | Node (v, name, l, r, _) ->
      let (lfun, lnames) = party_out l
      and (rfun, rnames) = party_out r in
      (v + lfun + rfun, name :: lnames @ rnames)

and party_out t =
  match t with
    Empty -> (0, [])
  | Node (v, _, l, r, _) ->
      let (lfun, lnames) = party l
      and (rfun, rnames) = party r in
      (lfun + rfun, lnames @ rnames)
```

Why was memoization so effective for solving this problem? As with the Fibonacci algorithm, we had the ***overlapping subproblems*** property, in which the naive recursive implementation called the function `party` many times with the same arguments. Memoization saves all those calls. Furthermore, the party optimization problem has the property of ***optimal substructure***, meaning that the optimal answer to a problem is computed from optimal answers to subproblems. Not all optimization problems have this property. The key to using memoization effectively for optimization problems is to figure out how to write a recursive function that implements the algorithm and has the two properties. Sometimes this requires thinking carefully.

Optimal Line Breaking

Here is a more involved example of memoization. Suppose that we have some text that we want to format as a paragraph within a certain column width. For example, we might have to do this if we were writing a web browser. For simplicity we will assume that all characters have the same width. A formatting of the text consists of choosing certain pairs of words to put line breaks in between. For example, when applied to the list of words in this paragraph, with width 60, we want output like the following:

```
let it =
  ["Here is a more involved example of memoization. Suppose that";
   "we have some text that we want to format as a paragraph";
   ...
   "applied to the list of words in this paragraph, with width";
   "60, we want output like the following:"] : string list
```

A good formatting uses up a lot of each column, and also gives each line similar widths.

The ***greedy*** approach would just fill each line as much as possible, but this can result in lines with very different lengths. For example, if we format the string "this may be a difficult example" at a width of 13 characters, we get a formatting that could be improved:

Greedy

```
this may be a
difficult
example
```

Optimal

```
this may be
a difficult
example
```

The TeX formatting program does a good job of keeping line widths similar by finding the formatting that minimizes the sum of the *cube* of the leftover space in each line (except the last). However, for *n* words, there are $\Omega(2^n)$ possible formattings, so the algorithm cannot possibly check them all for large text inputs. Remarkably, we can use memoization to find the optimal formatting efficiently. In fact, memoization is useful for many optimization problems.

We start by writing a simple recursive algorithm to walk down the list and try either inserting a line break after each word, or not inserting a linebreak:

```
(*  Determine the best way to break a list of words with lengths in the
 *  list "lengths" into a series of lines with maximum length "target".
 *  Different possible ways of doing this are compared based on the sum
 *  of the cubes of the difference between the target width and the actual
 *  width; this has the effect of tending to equalize line lengths.
 *)
let cube (x : int) = x * x * x
let big = 10000

(* Result of formatting a string. A result (lst, n)
   means a string was formatted into the lines in lst,
   with a total sum-of-cubes cost of n. *)
type breakResult = string list * int

(* Result: format the words in "words" into a list of lines optimally,
 * minimizing the sum of the cubes of differences between the line lengths
 * and "target".
 * Performance: worst-case time is exponential in the number of words.
 *)
let linebreak1 (words : string list) (target : int) : string list =
  let rec lb (clen : int) (words : string list) : breakResult =
    match words with
      [] -> ([""], 0) (* no charge for last line *)
    | word :: rest ->
      (* Try two ways of doing it: (1) insert a linebreak right after
       * current word, or (2) continue the current line.  Pick the
       * better one. *)
      let wlen = String.length word in
      let contlen = if clen = 0 then wlen else clen + 1 + wlen in
      let (l1, c1') = lb 0 rest in
```

```
      let c1 = c1' + cube (target - contlen) in
      if contlen <= target
      then
        let (h2 :: t2, c2) = lb contlen rest in
        if c1 < c2 then (word :: l1, c1)
                   else ((if h2 = ""
                          then word
                          else word ^ " " ^ h2) :: t2, c2)
      else (word :: l1, big) in
  let (result, cost) = lb 0 words in
  result
```

This algorithm is exponential because it computes all possible formattings. It is therefore much too slow to be practical.

The key observation is that in the optimal formatting of a paragraph of text, the formatting of the text past any given point is the optimal formatting of just that text, given that its first character starts at the column position where the prior formatted text ends. Thus, the formatting problem has ***optimal substructure*** when cast in this way. So if we compute the best formatting after a particular line break position, that formatting is the best for all possible formattings of the text before the break.

We can make `linebreak` take linear time by memoizing the best formatting for the calls where `clen = 0`. (We could memoize all calls, but that wouldn't improve speed much. This requires just introducing a function `lb_mem` that looks up and records memoized formatting results:

```
(* Same spec as linebreak1.
   Performance: worst-case time is linear in the number of words. *)
let linebreak2 (words : string list) (target : int) : string list =
  let memo : breakResult option array =
    Array.create (List.length words + 1) None in
  let rec lb_mem (words : string list) : breakResult =
    let n = List.length words in
    match Array.get memo n with
      Some br -> br
    | None -> let br = lb 0 words in
        Array.set memo n (Some br); br
  and lb (clen : int) (words : string list) : breakResult =
    match words with
      [] -> ([""], 0) (* no charge for last line *)
    | word :: rest ->
        let wlen = String.length word in
        let contlen = if clen = 0 then wlen else clen + 1 + wlen in
        let (l1, c1') = lb_mem rest in
        let c1 = c1' + cube (target - contlen) in
        if contlen > target then (word :: l1, big)
        else
          let (h2 :: t2, c2) = lb contlen rest in
          if c1 < c2 then (word :: l1, c1)
```

```
                    else ((if h2 = ""
                           then word
                           else word ^ " " ^ h2) :: t2, c2) in
  let (result, cost) = lb 0 words in
  result
```

Automatic Memoization Using Higher Order Functions

In the above examples, we manually inserted the code to do the memoization. However, we can automate this process using higher order functions. First consider the case of memoizing an arbitrary non-recursive function `f`. In that case we simply create a hash table that stores the corresponding value for each argument on which `f` is ever called. (We assume that all functions have a single argument; to memoize multi-argument function, we use currying to convert to a single-argument function.)

```
let memo f =
  let h = Hashtbl.create 11 in
    fun x ->
      try Hashtbl.find h x
      with Not_found ->
        let y = f x in
        Hashtbl.add h x y; y
```

Note that this particular construction only works for nonrecursive functions. The problem is that a recursive function is already defined to call itself on recursive calls, not its memoized version, so it is already too late to memoize it; we can only memoize the first call.

However, it is possible to automatically memoize recursive functions provided we modify the recursive call structure slightly. Specifically, we need to make explicit the construction of a recursive function as the fixpoint of a recursive functional.

To illustrate, consider the recursive definition of the `fib` function.

```
let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Notice that the identifier `fib` occurs both on the left and the right of the equals sign, thus the `fib` function is the solution to a recursive equation. Equivalently, it is a fixpoint of the non-recursive functional (higher-order function) `t_fib` defined by

```
let t_fib g n = if n < 2 then 1 else g (n-1) + g (n-2)
```

of type

```
t_fib : (int -> int) -> int -> int
```

The functional `t_fib` "unwinds" the definition once, taking an arbitrary function `g` and producing a new function

```
fun n -> if n < 2 then 1 else g (n-1) + g (n-2)
```

that is a better approximation to the fixpoint `fib` in the sense that it is correct for more values. Now we can obtain the fixpoint `fib` by applying a general fixpoint functional to `t_fib`:

```
(* fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b *)
(* gives a fixpoint of the functional t *)
let fix t = let rec g x = t g x in g
```

Applied to `t_fib`, this yields `fib`:

```
(* the fibonacci function without memoization *)
let fib = fix t_fib
```

Now that we have made the construction of the fixpoint explicit, we can produce memoized versions automatically by applying a general memoizing fixpoint functional:

```
(* fix_memo : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b *)
(* computes the least fixpoint of a given functional *)
(* with memoization *)
let fix_memo t =
  let memo = Hashtbl.create 11 in
  let rec g x =
    try Hashtbl.find memo x
    with Not_found ->
      let y = t g x in
      Hashtbl.add memo x y; y in
  g
```

Applying this to `t_fib`, we obtain a memoized version of `fib`:

```
(* the fibonacci function with memoization *)
let fib_memo = fix_memo t_fib
```

Try it!

Conclusion

Memoization is a powerful technique for asymptotically speeding up simple recursive algorithms without having to change the way the algorithm works. Memoization is an approach to the extremely useful algorithmic technique called ***dynamic programming***, which you will see in CS4820.