Announcements:

- PS #6 due Thursday December 1, 11:59PM
- Guest lecture on Tuesday
- Final quiz on Tuesday 11/29

- P2 comments: you will see the hard problems again on the final. Reversing a list is a classic test of your understanding of pointers.
- Only source of frustration: if I give you code that calls a function mi1 and ask you to define mi1 you are not allowed to change the code I gave you.
    - File this under “read the question”.

- Key environment model examples

```
let ctr1 =
  let v = ref 0 in
  fun(x) -> (v := !v + x; v) (* or !v to be functional *)

let ctr2 =
  fun(x) ->
  let v = ref 0 in
    (v := !v + x; v)

# ctr1(5);;
- : int ref = {contents = 5}
# ctr1(5);;
- : int ref = {contents = 10}
# ctr2(5);;
- : int ref = {contents = 5}
# ctr2(5);;
- : int ref = {contents = 5}
```

- Memoization

- Recall definition of fact and fib:

```
let rec fact =
 fun(x) -> if x=0 then 1 else x*fact(x-1)

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

- Today: memoization to speed up the computation
- Basic idea: store previous answer so we don't need to recompute it!
  - Classic time-space tradeoff, but can pay big dividends
  - Closely related to Dijkstra
- Note: side effects to speed up a functional program!
  - Not part of contract (hidden state)

```
let lx = ref 0
and lfact = ref 0 in
 let rec mfact =
    fun(x:int) -> if x=0 then 1
     else
    if !lx=x then
     begin
       print_string("Found it");
      !lfact;
     else
        begin
         lx := x; lfact := x*mfact(x-1); !lfact
        end
```

- We can do same thing for fib, with huge payoff
- Naïve fib is O(ϕn), where ϕ is the golden ratio 1.618
    - Proof of this, plus the fact this bound is tight, in section
    - Nice use of substitution method plus induction

```
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.
- What does this buy us? Lots!

- 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.
- Again, compare with Dijkstra's algorithm.
- This is a really powerful argument for CS
    - Taking a problem that seems to require exponential time and solving it in linear time!
- Many famous CS results are of this form

- We can use higher order functions to memoize any function.
- First consider the case of memoizing a non-recursive function `f`.
- In that case we simply need to create a hash table that stores the corresponding value for each argument that `f` is called with
    - (and to memoize multi-argument functions we can use currying and uncurrying to convert to a single argument function).
- We'll do recursive ones in section, or maybe at the end of lecture (hah!)

```
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
```

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.

- For recursive functions, however, the recursive call structure needs to be modified.
- This can be abstracted out independent of the function that is being memoized:

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

- Now we can slightly rewrite the original `fib` function from the beginning of lecture using this general memoization technique:

```
let fib_memo =
  let rec fib self = function (n) ->
    if n<2 then 1 else self(n-1) + self(n-2)
  in
    memo_rec fib
```