today

• 2-person games, continued
sequences

signature SEQ =
 sig
type 'a seq
exception Range
val nth : int -> 'a seq -> 'a
val length : 'a seq -> int
val tabulate : (int -> 'a) -> int -> 'a seq
val empty : unit -> 'a seq
val null : 'a seq -> bool
val map : ('a -> 'b) -> ('a seq -> 'b seq)
val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
val reduce1 : ('a * 'a -> 'a) -> 'a seq -> 'a
val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end
context

• Given a structure Game : GAME

• For simplicity, assume we’ve opened Game

  type state
  type move
  val step : state * move -> state
  val moves : state -> move Seq.seq
  val score : state -> int

• We defined mutually recursive functions

  F : state -> int
  G : state -> int

  that compute the best possible outcome for each of the game players,
  assuming they both try their best strategy
fun F s = 
  let
    val M = moves s 
  in
  if (null M) then score s else
    reduce1 Int.max (map (fn m => G(step(s, m))) M)
end

and
G s = 
  let
    val M = moves s 
  in
  if (null M) then ~(score s) else
    reduce1 Int.min (map (fn m => F(step(s, m))) M)
end

minimax

F calls G, 
uses Int.max

G calls F, 
uses Int.min
fun duration (s : state) : int =
    let
        val M = moves s
    in
        if (null M) then 0 else
            1 + reduce1 Int.max
            (map (fn m => duration(step(s,m))) M)
    end

\[ \text{duration}(s) = \text{longest move chain from s} \]
terminating

Definition

Game : GAME is terminating

iff for all s : state,

duration s evaluates to a (non-negative) integer

We assume that
score s is valuable,
for all terminal states s

s is terminal
iff moves s is empty
F and G

• How can we reason about mutually recursive functions?

• Use a mutually inductive proof!

Theorem

Let Game : GAME be terminating. Then F and G are total functions, and for all s : state, (F s) = ~(G s).

Proof? By induction on duration(s).

Key fact:  \( \text{Int.min} (\sim v_1, \sim v_2) = \sim \text{Int.max} (v_1, v_2) \)
We could have defined $F$ by itself, recursively:

```ocaml
fun F s =
    let
        val M = moves s
    in
        if (null M) then score s else
            reduce1 Int.max (map (fn m => ~ F(step(s, m)))) M
    end
```
move order

The *enumeration* order of $\text{moves}(s)$ does not affect the value of $F(s)$

• Proof?

Assume that for all $s$:state, $\text{moves}_1(s)$ is a permutation of $\text{moves}_2(s)$. Let $F_1, G_1$ use $\text{moves}_1$ and $F_2, G_2$ use $\text{moves}_2$.

Show that

$$F_1 = F_2 \quad G_1 = G_2$$

Key facts: $\text{Int.max, Int.min}$ are associative and commutative
cost analysis

- Let $W_F(d)$ be the work of $F(s)$ and $S_F(d)$ be the span of $F(s)$ for a state $s$ with duration $d$

  $$W_F(d) = W_G(d)$$
  $$S_F(d) = S_G(d)$$

  **Why? ... symmetry**

  Under reasonable assumptions, $F$ does $O(1)$ work between recursive calls.

  So, up to big-O, the work/span for $F(s)$ depends only on duration(s).
cost analysis

- Assume game has branching factor $k$: **at most** $k$ next moves are possible, from any state
- Assume moves, score, step are constant-time

Under these assumptions, the work/span for $F(s)$ depend only on duration(s)
cost analysis

\[ F(s) = \text{score } s \]
when \( \text{moves } s \) is empty

\[ F(s) = \text{reduce1 Int.max } (\text{map } (\text{fn } m \Rightarrow G(\text{step}(s, m))) \text{ (moves } s)) \]
otherwise

\[
\begin{align*}
W_F(0) &= c \\
W_F(d) &\leq k * W_F(d-1) + O(k) \\
& \text{for } d > 0
\end{align*}
\]
\( W_F(d) \) is \( O(k^d) \)

\[
\begin{align*}
S_F(0) &= c \\
S_F(d) &\leq S_F(d-1) + O(\log k) \\
& \text{for } d > 0
\end{align*}
\]
\( S_F(d) \) is \( O(d \log k) \)
Nim work

• For Nim, the branching factor $k$ is 3
• Starting from 15 sticks, the duration $d$ is 15
• So the work for $F_{15}$ is bounded by $3^{15} = 14348907$
• And the work for $F_n$ is $O(3^n)$

Standard ML of New Jersey runs sequentially, so that's why $F_{30}$ takes so long!
move order

• The order in which moves are enumerated does not affect the *extensional behavior* of F and G

• But may have drastic effect on efficiency (we will see this later!)
For MaxiMe, a best move is one that leads to the maximum outcome.

```ml
type edge = move * int

fun max_edge ((m1,v1), (m2,v2)) =  
  if v1 < v2 then (m2,v2) else (m1,v1)

fun maxbest (M : edge Seq.seq) = reduce1 max_edge M

fun MaxieBestMove s =  
  let
    val M = moves s
    val (m,_) = maxbest (map (fn m => (m, G(step(s, m)))) M)
  in
    m
  end
```
PLAYER

signature PLAYER =
sig
  structure Game : GAME
  val player : Game.state -> Game.move
end
functor MaxiMe(Game : GAME) : PLAYER =
struct
  structure Game = Game
  open Game

  fun F p = ...
  and G p = ...

  type edge = ...
  fun maxmove ...
  fun maxbest ...

  fun player p =
    let
      val M = moves p
      val (m,_) = maxbest(map (fn m => (m, G(step(p,m)))) M)
    in
      m
    end
end

player : Game.state -> Game.move
picks optimal move for Maxie
Standard ML of New Jersey

- structure MaxiNim = MaxiMe(Nim);
structure MaxiNim : PLAYER

- MaxiNim.player 15;
val it = 2 : Nim.move

- MaxiNim.player 35;
(* takes a long time! *)
The MaxieNim player explores the entire game tree.

**solution**

Bounded minimax:

- use minimax up to depth $d$
- use heuristic function $estimate : state \rightarrow int$ to guess outcome at depth $> d$
bounded minimax

fun $F'(d : \text{int}) (s : \text{state}) : \text{int} =$
  let
    \text{val} M = \text{moves} \ s
  in
    \text{if} \ (\text{null} \ M) \ \text{then} \ (\text{score} \ s) \ \text{else}
    \text{if} \ d=0 \ \text{then} \ (\text{estimate} \ s) \ \text{else}
    \ \\text{reduce1 \ \text{Int.max} \ (\text{map} \ (\text{fn} \ m \Rightarrow \ G' \ (d-1) \ (\text{step}(s, m))) \ M)}
  end

and

$G'(d : \text{int}) (s : \text{state}) : \text{int} =$
  let
    \text{val} M = \text{moves} \ s
  in
    \text{if} \ (\text{null} \ M) \ \text{then} \ -(\text{score} \ s) \ \text{else}
    \text{if} \ d=0 \ \text{then} \ -(\text{estimate} \ s) \ \text{else}
    \\text{reduce1 \ \text{Int.min} \ (\text{map} \ (\text{fn} \ m \Rightarrow \ F' \ (d-1) \ (\text{step}(s, m))) \ M)}
  end

$F', G' : \text{int} \rightarrow \text{state} \rightarrow \text{int}$
guessing Nim

For Nim there is a genius heuristic

```plaintext
fun estimate k =
if (k mod 4 = 1) then ~1 else 1
```
Nim theorem

• Let $F$ be the Nim Maxie function

$$F_0 = 1 \quad F_1 = \sim 1 \quad F_2 = 1$$

$$F_k = \text{reduce1} \ \text{Int.max} \langle \sim F(k-3), \sim F(k-2), \sim F(k-1) \rangle$$

• Prove that for all $k \geq 0$

$$F_k = \text{if } (k \mod 4 = 1) \ \text{then} \ \sim 1 \ \text{else} \ 1$$
• If the depth bound is large enough, bounded minimax is perfect

\[ \text{If } d \geq \text{duration}(s), \text{ then } F' \ d \ s = F \ s \]

• If the estimation function is perfect, bounded minimax to any depth is perfect

\[ \text{If } F = \text{estimate}, \text{ then } F' \ d = F \]
reflection

• We used a very simple GAME signature

• Could have added extra features to print states and moves

• Could have allowed for a more general type of outcome or estimation value, e.g.
  
  \[
  \text{score, estimate : state} \rightarrow \text{real}
  \]

• Could design referee to use exceptions...

  \[\text{raise YellowCard} \]

  \[\text{... handle Invalid _} \Rightarrow \text{...}\]
reflection

• Can use *depth-bounding* to limit runtime, but this may lead to sub-optimal results
  
  • the quality of results will depend on *search depth* and how well you *estimate*  
  
  • even *bounded* minimax may *waste time* searching fruitlessly...
doing better

- Choose *optimal order* to explore moves
  - using game-specific knowledge
- Propagate *extra information* to avoid wasteful computation
  - alpha-beta pruning

**Alpha–beta pruning** is a search algorithm that seeks to decrease the number of game tree nodes that get evaluated by the *minimax algorithm*. It stops evaluating a move when at least one possibility is found that proves the move to be worse than a previously examined move.