Sorting
• Introduction Racket and recursion; analysis
• Test-driven programming; tools to aid recursive design
• Varying forms of recursion; “partial result” tricks
• Higher order procedures
• Combinatorial procedures and their design
• Ocaml: matching as a syntactic structure; types
• Interpreting a program
• Sorting and analysis
• Optimization and min-max search
• Theoretical ideas: undecidability, “lambda is enough”
• Applications
Today

• Review of closures, environments, rules of evaluation
• Introduction to sorting
(define a 3)

• It’s a definition
• 2\textsuperscript{nd} item must be an identifier
  • Check!
• 3\textsuperscript{rd} item must be an expression
  • Check!
• Evaluate 3\textsuperscript{rd} item
  • Value is the number 3
• Add binding to TLE a -> 3.

+ -> addn builtin proc
* -> mult builtin proc
cons -> cons builtin proc
... 
a -> 3
\((+ \ 3 \ 2)\)

- It’s a proc-app expr
- 1\textsuperscript{st} item must be evaluate to a proc
  - Check!
- Evaluate other items to get “actuals”
- Apply proc to actual args to get value
- Print value (because it’s a top-level expression)
(let ((a 2))
  (+ a 1))

• It’s a let-expression
• 2\textsuperscript{nd} item is a list of ident-exp pairs
• 3\textsuperscript{rd} item must be an expression
• Temporarily extend current envt with new bindings from 2\textsuperscript{nd} item
• Evaluate 3\textsuperscript{rd} item in extended envt
  • Get 3
• Remove temporary bindings
• Note the “lookup from the bottom” rule
(lambda (x) (+ x 1))

• It’s a lambda-expn
• Evaluates to a closure w/ 3 parts
  • Arg list
  • Body
  • Local envt
• Local envt contains all bindings of current envt not in TLE
• Print value (because it’s a top-level expn)
  <user-proc>
• Value disappears
\[(\text{let } ((a \ 2)) \ (\text{lambda } (x) \ (+ \ x \ 1)))\]

• It’s a let expression
• Add temporary bindings
• Expr evaluates to a closure w/ 3 parts
• Local envt contains all bindings of current envt not in TLE
• Print value (because it’s a top-level expn)

\text{<user-proc>}

• Value disappears
• Temp bindings disappear
(let ((a 2))
  (let ((f (lambda (x) (+ x 1))))
    (f 6)))

- It’s a let expression
- Add temporary bindings
  - It’s a let expr
  - Add temp bindings
  - Evaluate (f 6)
    - Proc-app expr; user-def’d proc
      - Evaluate actuals: 6
      - Create new temporary bindings from closure’s envt
      - Add new bindings from formal to actuals
      - Evaluate body: 7
      - Print
    - Remove temp bindings
  - Remove temp bindings
- Remove temp bindings

+ -> addn builtin proc
* -> mult builtin proc
cons -> cons builtin proc
(let ((a 2))
  (let ((f (lambda (x) (+ x a))))
      (f 6)))

• It’s a let expression
• Add temporary bindings
  • It’s a let expr
  • Add temp bindings
• Evaluate (f 6)
  • Proc-app expr; user-def’d proc
    • Evaluate actuals: 6
    • Create new temporary bindings from closure’s envt
    • Add new bindings from formal to actuals
    • Evaluate body: 8
    • Print
    • Remove temp bindings
• Remove temp bindings
• Remove temp bindings
(let ((a 2))
  (let ((f (lambda (x) (+ x a))))
    (f 6)))

• Once we bind f, it’s the “add two” function
• Values in racket aren’t allowed to change
• That same “f” can’t become the add-3 function, or all our assumptions will break!
• Let’s try one level deeper
(let ((a 2))
  (let ((f (lambda (x) (+ x a))))
    (let ((a 13))
      (f 6))))

• Same sequence as before
• Just before evaluating (f 6), envt is this:
• If we evaluated “a”, we’d get 13.
• Evaluate (f 6)
  • Add bindings from closure’s envt
  • Add bindings from formals to actuals
  • Evaluate body; print
  • Gradually remove bindings for all stages as before
Sorting

• Applied in just about every program in the world
• We’ll illustrate with integers
  • All you really need are
    • Data
    • A “consistent” way to compare data
      • If $a > b$ and $b > c$, need $a > c$ for consistency
• There are many sorting algorithms
• Fast ones can be quite clever
• Different resource use in terms of number of elementary ops, storage required, etc.
• For large datasets, too huge to fit in memory, “cost” of accessing different parts of data is an important measure too.
  • CS157
Sorting example

• Sort [8; 3; 4; 2] in increasing order
• Recursive diagram
• Original input: [8; 3; 4; 2]
  • Recursive input: [3; 4; 2]
  • Recursive output: [2; 3; 4]

8 = head of list
Append it at end?
Insert it in proper place? Helper proc?

• Overall output: [2; 3; 4; 8]
Helper: insert item in sorted list

- Original input: 5, [2; 3; 7; 8]
  - Recursive input: 5, [3; 7; 8]
  - Recursive output: [3; 5; 7; 8]

Cons head onto front of recursive output?

- Overall output: [2; 3; 5; 7; 8]
Helper: insert item in sorted list

• Original input: 1, [2; 3; 7; 8]
  • Recursive input: 1, [3; 7; 8]
  • Recursive output: [1; 3; 7; 8]

Now we need to insert the “2” into the sorted recursive output!

• Overall output: [1; 2; 3; 5; 7; 8]
Helper: insert item in sorted list

• Re-examine problem:
• Original input: 1, [2; 3; 7; 8]
• Overall output: [1; 2; 3; 5; 7; 8]
• No recursion at all: just cons on the 1
• Key difference: 1 < 2, but 5 > 2.
insert_in_sorted

(* insert_in_sorted: int  -> int list -> int list
   inputs:
   an integer n
   a sorted (nondecreasing) list aloi of integers
   Output:
   a sorted list containing the same items as aloi, but n as well
   *)

let rec insert_in_sorted n aloi = match aloi with
  | [] -> [n]
  | hd::tl -> if n <= hd
    then
      n::aloi
    else
      hd:: (insert_in_sorted n tl)

check_expect (insert_in_sorted 3 [] ) [3];
check_expect (insert_in_sorted 1 [2;3;5]) [1;2;3;5];
check_expect (insert_in_sorted 4 [2;3;5]) [2;3;4;5];
Sorting example refresher

• Sort [8; 3; 4; 2] in increasing order
• Recursive diagram
• Original input: [8; 3; 4; 2]
  • Recursive input: [3; 4; 2]
  • Recursive output: [2; 3; 4]

8 = head of list
Append it at end?
Insert it in proper place? Helper proc?

• Overall output: [2; 3; 4; 8]
(* insertion_sort: int list -> int list
   inputs:
   a list aloi of integers
Output:
   a sorted list containing the same items as aloi *)

let rec insertion_sort aloi = match aloi with
| [] -> []
| hd::tl -> insert_in_sorted hd (insertion_sort tl)

check_expect (insertion_sort []) [];
check_expect (insertion_sort [2; 4; 3; 8]) [2; 3; 4; 8];
check_expect (insertion_sort [5; 2; 1]) [1; 2; 5];
Analysis

• Let $U(n)$ be max ops in `insert_in_sorted` for an n-item list
  \[ U(0) = A \]
  \[ U(n) \leq B + U(n - 1) \]

• $U \in O(n \mapsto n)$ by theorem.

• Let $K(n)$ be max ops in `insertion_sort` on n-item list
  \[ K(0) = A \]
  \[ K(n) \leq B + U(n - 1) + K(n - 1) \]
  \[ K(n) \leq B + C(n - 1) + D + K(n - 1) \]
  \[ K(n) \leq Q + Rn + K(n - 1) \]

• $K \in O(n \mapsto n^2)$ by theorem.
Where’s the problem?

• All that time taken to figure out where to put the item in the sorted list adds up!
• What if we simply chose the *least* item as the one to hold out?
Selection sort

- Find least item in list, and the remainder of the list (all but the least item)
- Sort the remainder recursively
- Cons the least item to the front of the recursive output
let rec ssort (alon: int list) = match alon with
| [] -> []
| _ -> let (min, rest) = find_min(alon)
  in
    min :: (ssort rest)
and
find_min (alon: int list): int * (int list) = match alon with
| [item] -> (item, [])
| hd :: tl -> let (min2, rest2) = find_min(tl)
  in
    if (hd < min2)
    then
      (hd, min2::rest2)
    else
      (min2, hd::rest2)
| [] -> failwith "find_min undefined for empty list" ;;
A different approach

• Divide and conquer
  • So far we’ve divided into an item and a shorter list
  • Alternative: divide into two equal-sized lists
Merge sort idea

• Divide input into two lists somehow
• Sort each one
• “merge” together the results
Merging: recursive diagram

• Input: two sorted lists
• Output: sorted list containing all items of both lists
• Original input: [1; 3; 6] [2; 7; 8]
  • Recursive input: [3; 6] [7; 8]
  • Recursive output: [3; 6; 7; 8]

Cons heads of two lists onto recursive output, in the right order
ugly

• Overall output: [1; 2; 6; 7; 8]
Revised recursive diagram

• Input: two sorted lists
• Output: sorted list containing all items of both lists
• Original input: [1; 3; 6] [2; 7; 8]
  • Recursive input: [3; 6] [2; 7; 8]
  • Recursive output: [2; 3; 6; 7; 8]

Cons smaller of two heads onto result of merging everything else

• Overall output: [1; 2; 6; 7; 8]
let rec merge (sl1: int list) (sl2: int list): int list =
match (sl1, sl2) with
| [], _ -> sl2
| _, [] -> sl1
| hd1::tl1, hd2:tl2 -> if (hd1 < hd2)
  then
  hd1::(merge tl1 sl2)
  else
  hd2::(merge sl1 tl2);;
let rec merge_sort (aloi: int list) = match aloi with
| [] -> []
| [n] -> [n]
| _ -> let
    (part1, part2) = split(aloi)
    in
    merge (merge_sort part1) (merge_sort part2);;

let rec split (aloi: int list): ((int list) * (int list)) = match aloi with
| [] -> ([], [])
| [n] -> ([n], [])
| a::b::rest -> let (s1, s2) = split(rest) in (a::s1, b::s2);;

• Obs: runtime of “split” is in $O(n \mapsto n)$
Analysis

```
let rec merge_sort (aloi: int list) = match aloi with
  | [] -> []
  | [n] -> [n]
  | _ -> let
    (part1, part2) = split(aloi)
    in
    merge (merge_sort part1) (merge_sort part2);
```

• Let $M(n)$ be worst runtime for mergesort on $n$-item list.

$$M(n) \leq 2M\left(\frac{n}{2}\right) + Q(n)$$

$$M(n) \leq 2 M\left(\frac{n}{2}\right) + An + B$$
\[ M(n) \leq 2 \cdot M\left(\frac{n}{2}\right) + An + B \]

- Pattern: for powers of 2, we have
  \[ M\left(2^k\right) \leq 2^k(C + kA) + B \]
- Prove via well-ordering
- Dominant term: \( A \cdot 2^k \cdot k = A \cdot n \log n \)
- Mergesort’s op-counter is in \( O(n \mapsto n \log n) \)