Part I: Generative Recursion

Marco T. Morazán

Seton Hall University

Outline

- Language: Intermediate Student with lambda Language (`ISL+`)
- All `BSL` and `ISL` programs will still work

Intro to Generative Recursion

- `generate-password` ⇒ useful recursive call may be made without using the substructure of any input

Intro to Generative Recursion

- `generate-password` ⇒ useful recursive call may be made without using the substructure of any input
- Insights made such recursive calls useful

Intro to Generative Recursion

- `generate-password` ⇒ useful recursive call may be made without using the substructure of any input
- Insights made such recursive calls useful
- How do we know such a function will ever halt?

Intro to Generative Recursion

- `generate-password` ⇒ useful recursive call may be made without using the substructure of any input
- Insights made such recursive calls useful
- How do we know such a function will ever halt?
- Most functions we write are deterministic and we ought to argue that they halt

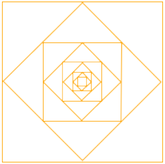
Intro to Generative Recursion

Generating Nested Squares

- Is recursion not based on the structure of the data ever useful in other settings?

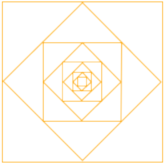
Intro to Generative Recursion

Generating Nested Squares

- Is recursion not based on the structure of the data ever useful in other settings?
-  
- How are these images created?

Intro to Generative Recursion

Generating Nested Squares

- Is recursion not based on the structure of the data ever useful in other settings?
-  
- How are these images created?
- A nested-squares image is a composition of two images: a large square and a smaller nested-squares image
- Recursive images are examples of *fractals*
- A fractal is a never-ending pattern across different scales

Intro to Generative Recursion

Generating Nested Squares

- Basic idea: from a given square image a smaller square image is computed and processed recursively
- The result of the recursive call is placed over the given square image

Intro to Generative Recursion

Generating Nested Squares

- Basic idea: from a given square image a smaller square image is computed and processed recursively
- The result of the recursive call is placed over the given square image
- This is not structural recursion
- A smaller square image is not part of structure of the given square image

Intro to Generative Recursion

Generating Nested Squares

- Basic idea: from a given square image a smaller square image is computed and processed recursively
- The result of the recursive call is placed over the given square image
- This is not structural recursion
- A smaller square image is not part of structure of the given square image
- What is used as the input to a recursive call?

Intro to Generative Recursion

Generating Nested Squares

- Basic idea: from a given square image a smaller square image is computed and processed recursively
- The result of the recursive call is placed over the given square image
- This is not structural recursion
- A smaller square image is not part of structure of the given square image
- What is used as the input to a recursive call?
- A new problem has been generated: nested-squares for a smaller square

Intro to Generative Recursion

Generating Nested Squares

- Basic idea: from a given square image a smaller square image is computed and processed recursively
- The result of the recursive call is placed over the given square image
- This is not structural recursion
- A smaller square image is not part of structure of the given square image
- What is used as the input to a recursive call?
- A new problem has been generated: nested-squares for a smaller square
- Recursion based on generating one or more new instances of a problem (i.e., the subproblems) and creating a solution from the solutions of the subproblems is known as *generative recursion*.

Intro to Generative Recursion

Generating Nested Squares

- How is the recursion stopped?

- How are the subproblems generated?

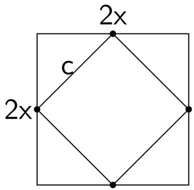
Intro to Generative Recursion

Generating Nested Squares

- How is the recursion stopped?
- Stop when the square's length becomes too small for the human eye to see
- How are the subproblems generated?

Intro to Generative Recursion

Generating Nested Squares

- How is the recursion stopped?
- Stop when the square's length becomes too small for the human eye to see
- How are the subproblems generated?
- 

- $\frac{c}{2x} = \frac{\sqrt{2}x}{2x} \approx 0.711$
- `B`'s length is 71.1% of `A`'s length
- Smaller square image needs to be rotated by 45 degrees

Intro to Generative Recursion

- `(define TOO-SMALL-LEN 20)`

Intro to Generative Recursion

- `(define TOO-SMALL-LEN 20)`
- ```
  (define SQ1 (square 15   ...))  (define SQ2 (square 8   ...))
  (define SQ3 (square 1000 ...))  (define SQ4 (square 800 ...))
  ```
  ```

Intro to Generative Recursion

- `(define TOO-SMALL-LEN 20)`
- ```
  (define SQ1 (square 15   ...))  (define SQ2 (square 8   ...))
  (define SQ3 (square 1000 ...))  (define SQ4 (square 800 ...))
  ```
- ```
  ;; Sample expressions for nested-squares
  (define CRAZY-SQ1 SQ1)  (define CRAZY-SQ2 SQ2)
  (define CRAZY-SQ3 (overlay (nested-squares
                               (rotate 45 (scale 0.711 SQ3)))
                             SQ3))
  (define CRAZY-SQ4 (overlay (nested-squares
                               (rotate 45 (scale 0.711 SQ4)))
                             SQ4))
  ```

Intro to Generative Recursion

- `(define TOO-SMALL-LEN 20)`
-
```
(define SQ1 (square 15   ...))  (define SQ2 (square 8   ...))
(define SQ3 (square 1000 ...))  (define SQ4 (square 800 ...))
```
-
```
;; Sample expressions for nested-squares
(define CRAZY-SQ1 SQ1)  (define CRAZY-SQ2 SQ2)
(define CRAZY-SQ3 (overlay (nested-squares
                             (rotate 45 (scale 0.711 SQ3)))
                           SQ3))
(define CRAZY-SQ4 (overlay (nested-squares
                             (rotate 45 (scale 0.711 SQ4)))
                           SQ4))
```
-
```
;; image → image  Purpose: Generate nested squares image
;; Assumption: Given image is a square
;; How: Overlay over the given image the nested squares image
;;      computed from 0.711 given img rotated 45 degrees.
(define (nested-squares sqr-img)
```

Intro to Generative Recursion

- `(define TOO-SMALL-LEN 20)`
- ```
  (define SQ1 (square 15   ...))  (define SQ2 (square 8   ...))
  (define SQ3 (square 1000 ...))  (define SQ4 (square 800 ...))
  ```
- ```
  ;; Sample expressions for nested-squares
  (define CRAZY-SQ1 SQ1)  (define CRAZY-SQ2 SQ2)
  (define CRAZY-SQ3 (overlay (nested-squares
                               (rotate 45 (scale 0.711 SQ3)))
                             SQ3))
  (define CRAZY-SQ4 (overlay (nested-squares
                               (rotate 45 (scale 0.711 SQ4)))
                             SQ4))
  ```
- ```
  ;; image → image  Purpose: Generate nested squares image
  ;; Assumption: Given image is a square
  ;; How: Overlay over the given image the nested squares image
  ;;      computed from 0.711 given img rotated 45 degrees.
  (define (nested-squares sqr-img)
  ```
- ```
  ;; Tests using sample computations for nested-squares
  (check-expect (nested-squares SQ1) CRAZY-SQ1)
  (check-expect (nested-squares SQ2) CRAZY-SQ2)
  (check-expect (nested-squares SQ3) CRAZY-SQ3)
  (check-expect (nested-squares SQ4) CRAZY-SQ4)
  ```

Intro to Generative Recursion

- `(define TOO-SMALL-LEN 20)`
- ```
  (define SQ1 (square 15   ...))  (define SQ2 (square 8   ...))
  (define SQ3 (square 1000 ...))  (define SQ4 (square 800 ...))
  ```
- ```
  ;; Sample expressions for nested-squares
  (define CRAZY-SQ1 SQ1)  (define CRAZY-SQ2 SQ2)
  (define CRAZY-SQ3 (overlay (nested-squares
                               (rotate 45 (scale 0.711 SQ3)))
                             SQ3))
  (define CRAZY-SQ4 (overlay (nested-squares
                               (rotate 45 (scale 0.711 SQ4)))
                             SQ4))
  ```
- ```
  ;; image → image  Purpose: Generate nested squares image
  ;; Assumption: Given image is a square
  ;; How: Overlay over the given image the nested squares image
  ;;      computed from 0.711 given img rotated 45 degrees.
  (define (nested-squares sqr-img)
  ```
- ```
    (if (<= (image-width sqr-img) TOO-SMALL-LEN)
        sqr-img
        (overlay (nested-squares (rotate 45 (scale 0.711 sqr-img)))
                 sqr-img)))
  ```
- ```
  ;; Tests using sample computations for nested-squares
  (check-expect (nested-squares SQ1) CRAZY-SQ1)
  (check-expect (nested-squares SQ2) CRAZY-SQ2)
  (check-expect (nested-squares SQ3) CRAZY-SQ3)
  (check-expect (nested-squares SQ4) CRAZY-SQ4)
  ```
  ```

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.
2. Define constants for the value of sample expressions.

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature, purpose statement, how statement, and function header.

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature, purpose statement, how statement, and function header.
5. Write tests.

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature, purpose statement, how statement, and function header.
5. Write tests.
6. Write the function's body.

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature, purpose statement, how statement, and function header.
5. Write tests.
6. Write the function's body.
7. Write a termination argument.

Intro to Generative Recursion

Design Recipe

1. Perform problem and data analysis.
2. Define constants for the value of sample expressions.
3. Identify and name the differences among the sample expressions.
4. Write the function's signature, purpose statement, how statement, and function header.
5. Write tests.
6. Write the function's body.
7. Write a termination argument.
8. Run the tests and, if necessary, redesign.

Intro to Generative Recursion

Design Recipe

```
(define (gen-rec-f prob-inst ...)
  (cond [(trivial-prob1? prob-inst) solution for prob-inst1]
         ...
        [(trivial-probk? prob-inst) solution for prob-instk]
        [else
          (local [(define genprob1 (generate-prob1 problem ...))
                           ...
                  (define genprobn (generate-probn problem ...))]
            (combine ... problem
                     ... (gen-rec-f genprob1 ...) ...
                     ... (gen-rec-f genprobn))))]))
;; Sample instances of problem
(define TRV1  ...) ... (define TRVK  ...)
(define NTRV1 ...) ... (define NTRVN ...)
;; Sample expressions for gen-rec-f
(define TRV1-VAL ...) ... (define TRVK-VAL  ...)
(define NTRV1-VAL ...) ... (define NTRVN-VAL ...) ...
;; Tests using sample computations for gen-rec-f
(check-expect (gen-rec-f TRV1  ...) TRV1-VAL) ...
(check-expect (gen-rec-f NTRV1 ...) NTRV1-VAL) ...
;; Tests using sample values for gen-rec-f
(check-expect (gen-rec-f ... ...) ...) ...
```

Intro to Generative Recursion

HOMEWORK AND QUIZ

- HOMEWORK: 1 and 3

Intro to Generative Recursion

HOMEWORK AND QUIZ

- HOMEWORK: 1 and 3
- QUIZ: 2 (due in one week before class)

Intro to Generative Recursion

All Primes $\leq$ `n`

- Problem: compute all prime numbers less than or equal to a natnum `n`

Intro to Generative Recursion

All Primes $\leq$ `n`

- Problem: compute all prime numbers less than or equal to a natnum `n`
- Design around `n`?

Intro to Generative Recursion

All Primes $\leq$ `n`

- Problem: compute all prime numbers less than or equal to a natnum `n`
- Design around `n`?
- If `n` is 0 then there are no primes to return and the answer is the empty list

Intro to Generative Recursion

All Primes $\leq$ n

- Problem: compute all prime numbers less than or equal to a natnum n
- Design around n?
- If n is 0 then there are no primes to return and the answer is the empty list
- If n is greater than 0 then the function must decide if n is added to the result
  - If n is prime then it is consed with the result of processing n-1
  - If n is not prime then the result is obtained by processing n-1

Intro to Generative Recursion

All Primes $\leq$ n

- Problem: compute all prime numbers less than or equal to a natnum `n`
- Design around `n`?
- If `n` is 0 then there are no primes to return and the answer is the empty list
- If `n` is greater than 0 then the function must decide if `n` is added to the result
  - If `n` is prime then it is consed with the result of processing `n-1`
  - If `n` is not prime then the result is obtained by processing `n-1`
- Three cases that must be distinguished
- An auxiliary predicate to determine if a given natural number is prime is needed

Intro to Generative Recursion

All Primes $\leq$ `n`

- To determine if `n` is prime there are two cases to distinguish:
  - $\texttt{n} < 2$
  - $\texttt{n} \geq 2$

Intro to Generative Recursion

All Primes $\leq$ `n`

- To determine if `n` is prime there are two cases to distinguish:
  - `n` $<$ 2
  - `n` $\geq$ 2
- If `n` is less than 2 then the answer is `#false` because neither 0 nor 1 are prime

Intro to Generative Recursion

All Primes $\leq$ `n`

- To determine if `n` is prime there are two cases to distinguish:
  - `n` $<$ 2
  - `n` $\geq$ 2
- If `n` is less than 2 then the answer is `#false` because neither 0 nor 1 are prime
- Otherwise, it must be determined if `n` is prime
  - Any natural number that divides `n` must be $\geq$ 2 and $\leq$ to the quotient of `n` and 2.
  - Suggests processing the interval `[2..(quotient n 2)]` using an auxiliary function

Intro to Generative Recursion

All Primes $\leq$ n

- ```
  ;; Sample natnums
  (define ZERO 0)  ...
  ```
  ```

Intro to Generative Recursion

All Primes $\leq$ n

- ```
  ;; Sample natnums
  (define ZERO 0)  ...
  ```
- ```
  ;; natnum → (listof natnum)
  ;; Purpose: Compute primes ≤ to given natnum
  (define (all-primes<=n n)
  ```

Intro to Generative Recursion

All Primes $\leq$ n

- ```
  ;; Sample natnums
  (define ZERO 0)  ...
  ```
- ```
  ;; natnum → (listof natnum)
  ;; Purpose: Compute primes ≤ to given natnum
  (define (all-primes<=n n)
  ```
- ```
  ;; Sample expressions for all-primes<=n In the textbook...
  ;; Tests using sample computations for all-primes<=n In the textboo
  ;; Tests using sample values for all-primes<=n In the textbook...
  ```
  ```

Intro to Generative Recursion

All Primes ≤ n

- ```
  ;; Sample natnums
  (define ZERO 0)  ...
  ```
- ```
  ;; natnum → (listof natnum)
  ;; Purpose: Compute primes ≤ to given natnum
  (define (all-primes<=n n)
  ```
- ```
    (local
  ```
- ```
     (cond [(= n 0) '()]
           [(prime? n) (cons n (all-primes<=n (sub1 n)))]
           [else (all-primes<=n (sub1 n))])))
  ```
- ```
  ;; Sample expressions for all-primes<=n In the textbook...
  ;; Tests using sample computations for all-primes<=n In the textboo
  ;; Tests using sample values for all-primes<=n In the textbook...
  ```
  ```

Intro to Generative Recursion

All Primes ≤ n

- ```
  ;; Sample natnums
  (define ZERO 0)  ...
  ```
- ```
  ;; natnum → (listof natnum)
  ;; Purpose: Compute primes ≤ to given natnum
  (define (all-primes<=n n)
  ```
- ```
    (local
  ```
- ```
     [;; natnum → Boolean Purpose: Is given natnum prime?
      (define (prime? n)
       (local [;; [int int] → Boolean
               ;; Purpose: Any interval number divides n?
               (define (any-divide? low high)
                 (if (< high low) #false
                     (or (= (remainder n high) 0)
                         (any-divide? low (sub1 high)))))]
         (if (< n 2) #false
             (not (any-divide? 2 (quotient n 2))))))]
  ```
- ```
     (cond [(= n 0) '()]
           [(prime? n) (cons n (all-primes<=n (sub1 n)))]
           [else (all-primes<=n (sub1 n))])))
  ```
- ;; Sample expressions for all-primes<=n In the textbook...
  ;; Tests using sample computations for all-primes<=n In the textboo
  ;; Tests using sample values for all-primes<=n In the textbook...
  ```

Intro to Generative Recursion

All Primes $\leq$ `n`

- Tests take a bit of time to run
- Can we do better?

Intro to Generative Recursion

All Primes $\leq$ `n`

- Tests take a bit of time to run
- Can we do better?
- Finding a different way to solve a problem may be very challenging
- We can benefit from insights from others

Intro to Generative Recursion

All Primes $\leq$ `n`

- Eratosthenes of Cyrene, a `3rd` century BCE Greek mathematician, suggested using a sieving method
- Start with a list of natural numbers from 2 (the smallest prime number) to `n`.

Intro to Generative Recursion

All Primes ≤ `n`

- Eratosthenes of Cyrene, a `3rd` century BCE Greek mathematician, suggested using a sieving method
- Start with a list of natural numbers from 2 (the smallest prime number) to `n`.
- At each step, the first number in the list, `i`, is a prime that is added to the result and the process is repeated with the members of the rest of the list that are not multiples of `i`.
- Observe that it is generative recursion.

Intro to Generative Recursion

All Primes ≤ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`

Intro to Generative Recursion

All Primes $\leq$ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`
- No multiples of any number greater than or equal to `(quotient n 2)`

Intro to Generative Recursion

All Primes $\leq$ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`
- No multiples of any number greater than or equal to `(quotient n 2)`
- `(sieve '(2 3 4 5 6 7 8 9 10) 5)`
- 5 is the limit value to stop the recursion when `n = 10`

Intro to Generative Recursion

All Primes $\leq$ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`
- No multiples of any number greater than or equal to `(quotient n 2)`
- `(sieve '(2 3 4 5 6 7 8 9 10) 5)`
- 5 is the limit value to stop the recursion when `n = 10`
- `(cons 2 (sieve '(3 5 7 9) 5))`

Intro to Generative Recursion

All Primes ≤ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`
- No multiples of any number greater than or equal to `(quotient n 2)`
- `(sieve '(2 3 4 5 6 7 8 9 10) 5)`
- 5 is the limit value to stop the recursion when `n = 10`
- `(cons 2 (sieve '(3 5 7 9) 5))`
- `(cons 2 (cons 3 (sieve '(5 7) 5)))`

Intro to Generative Recursion

All Primes ≤ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`
- No multiples of any number greater than or equal to `(quotient n 2)`
- `(sieve '(2 3 4 5 6 7 8 9 10) 5)`
- 5 is the limit value to stop the recursion when `n = 10`
- `(cons 2 (sieve '(3 5 7 9) 5))`
- `(cons 2 (cons 3 (sieve '(5 7) 5)))`
- `(cons 2 (cons 3 (cons 5 (sieve '(7) 5))))`

Intro to Generative Recursion

All Primes ≤ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`
- No multiples of any number greater than or equal to `(quotient n 2)`
- `(sieve '(2 3 4 5 6 7 8 9 10) 5)`
- 5 is the limit value to stop the recursion when `n = 10`
- `(cons 2 (sieve '(3 5 7 9) 5))`
- `(cons 2 (cons 3 (sieve '(5 7) 5)))`
- `(cons 2 (cons 3 (cons 5 (sieve '(7) 5))))`
- Process stops because first list element is > limit
- `(cons 2 (cons 3 (cons 5 '(7))))`

Intro to Generative Recursion

All Primes ≤ `n`

- `n = 10`
- `(sieve '(2 3 4 5 6 7 8 9 10))`
- No multiples of any number greater than or equal to `(quotient n 2)`
- `(sieve '(2 3 4 5 6 7 8 9 10) 5)`
- 5 is the limit value to stop the recursion when `n = 10`
- `(cons 2 (sieve '(3 5 7 9) 5))`
- `(cons 2 (cons 3 (sieve '(5 7) 5)))`
- `(cons 2 (cons 3 (cons 5 (sieve '(7) 5))))`
- Process stops because first list element is > limit
- `(cons 2 (cons 3 (cons 5 '(7))))`
- When `n` is less than 2 (i.e., 0 or 1) the solution is the empty list because there are no prime numbers less than 2

Intro to Generative Recursion

All Primes $\leq$ n

- ```
  ;; Sample expressions for the-primes<=n
  (define ZERO-VALUE  '())
  (define ONE-VALUE   '())
  ```
  ```

Intro to Generative Recursion

All Primes ≤ n

- ```
  ;; Sample expressions for the-primes<=n
  (define ZERO-VALUE  '())
  (define ONE-VALUE   '())
  ```
- ```
  (define FIVE-VALUE (sieve (build-list (- FIVE 1)
                                        (λ (i) (+ i 2)))
                            (quotient FIVE 2)))
  (define SEVEN-VALUE (sieve (build-list (- SEVEN 1)
                                         (λ (i) (+ i 2)))
                             (quotient SEVEN 2)))
  (define SIX-VALUE (sieve (build-list (- SIX 1)
                                       (λ (i) (+ i 2)))
                           (quotient SIX 2)))
  (define 12K-VALUE (sieve (build-list (- 12K 1)
                                       (λ (i) (+ i 2)))
                           (quotient 12K 2)))
  ```

Intro to Generative Recursion

All Primes $\leq$ n

```
;; natnum → (listof natnum)
;; Purpose: Compute all primes ≤ to given natnum
(define (the-primes<=n n) ...)
```

Intro to Generative Recursion

All Primes $\leq$ n

```
;; Tests using sample computations for the-primes<=n
(check-expect (the-primes<=n ZERO)  ZERO-VALUE)
(check-expect (the-primes<=n ONE)   ONE-VALUE)
(check-expect (the-primes<=n FIVE)  FIVE-VALUE)
(check-expect (the-primes<=n SEVEN) SEVEN-VALUE)
(check-expect (the-primes<=n 12K)   12K-VALUE)

;; Tests using sample values for the-primes<=n2
(check-expect (the-primes<=n 17) '(2 3 5 7 11 13 17))
(check-expect (the-primes<=n 3)  '(2 3))
```

Intro to Generative Recursion

All Primes $\leq$ n

```
(if (< n 2)
   '()
   (sieve (build-list (- n 1) (λ (i) (+ i 2)))
          (quotient n 2)))
```

Intro to Generative Recursion

All Primes $\leq$ `n`

- We now focus on developing the auxiliary function `sieve` to eliminate the nonprimes

Intro to Generative Recursion

All Primes $\leq$ `n`

- We now focus on developing the auxiliary function `sieve` to eliminate the nonprimes
- The input to this function must be a list of natural numbers in nondecreasing such that first element is a prime and the rest of the list does not contain a multiple of a number less than the first element

Intro to Generative Recursion

All Primes $\leq$ n

- We now focus on developing the auxiliary function sieve to eliminate the nonprimes
- The input to this function must be a list of natural numbers in nondecreasing such that first element is a prime and the rest of the list does not contain a multiple of a number less than the first element
- Testing lists:

```
(define L1 '(7))
(define L2 '(11 13 17))
(define L3 '(5 7 11))
(define L4 '(2 3 4 5 6 7 8))
```

Intro to Generative Recursion

All Primes $\leq$ `n`

- ```
  ;; Sample expressions for sieve
  (define L1-VAL L1)
  (define L2-VAL L2)
  ```

- The first two sample expressions are written for lists that have a first element larger than the limit value (respectively, 4 and 9)
  ```

Intro to Generative Recursion

All Primes ≤ n

- ```
  ;; Sample expressions for sieve
  (define L1-VAL L1)
  (define L2-VAL L2)
  ```
- ```
  (define L3-VAL
          (local
            [(define new-inst
                     (filter
                      (λ (n)
                       (not (= (remainder n (first L3)) 0)))
                      (rest L3)))]
           (cons (first L3) (sieve new-inst 6))))
  (define L4-VAL
          (local
            [(define new-inst
                     (filter
                      (λ (n)
                       (not (= (remainder n (first L4)) 0)))
                      (rest L4)))]
           (cons (first L4) (sieve new-inst 4))))
  ```
- The first two sample expressions are written for lists that have a first element larger than the limit value (respectively, 4 and 9)
- The second two are for lists that must be recursively processed

Intro to Generative Recursion

All Primes $\leq$ `n`

- There are two differences in the expressions for the nontrivial cases: the list of numbers processed and the limit value
- ```
  ;; (listof natnum) natnum → (listof natnum)
  ;; Purpose: Extract the prime numbers in the given list
  ```
  ```

Intro to Generative Recursion

All Primes $\leq$ n

- There are two differences in the expressions for the nontrivial cases: the list of numbers processed and the limit value
- ```
  ;; (listof natnum) natnum → (listof natnum)
  ;; Purpose: Extract the prime numbers in the given list
  ```
- ```
  ;; Assumption:
  ;;   The given list of natural numbers is nonempty, its
  ;;   first element is prime, and contains no numbers
  ;;   that are divisible by a number less than the first
  ;;   element.
  ```

Intro to Generative Recursion

All Primes $\leq$ n

- There are two differences in the expressions for the nontrivial cases: the list of numbers processed and the limit value
- ```
  ;; (listof natnum) natnum → (listof natnum)
  ;; Purpose: Extract the prime numbers in the given list
  ```
- ```
  ;; Assumption:
  ;;   The given list of natural numbers is nonempty, its
  ;;   first element is prime, and contains no numbers
  ;;   that are divisible by a number less than the first
  ;;   element.
  ```
- ```
  ;; How: If the first list element is greater than the
  ;;      limit stop. Otherwise, add the first number to
  ;;      the result and repeat the process by removing
  ;;      the multiples of the first element from the rest
  ;;      of the given list using the same limit value.
  (define (sieve lon limit) ...)
  ```
  ```

Intro to Generative Recursion

All Primes $\leq$ n

```
;; Tests using sample computations for sieve
(check-expect (sieve L1 4) L1-VAL)
(check-expect (sieve L2 9) L2-VAL)
(check-expect (sieve L3 6) L3-VAL)
(check-expect (sieve L4 4) L4-VAL)

;; Tests using sample computations for sieve
(check-expect (sieve '(5 7) 4) '(5 7))
(check-expect (sieve '(5 7 11 13 15) 8)
              '(5 7 11 13))
```

Intro to Generative Recursion

All Primes $\leq$ n

```
(if  (or (empty? lon)
         (> (first lon) limit))
     lon
     (local
       [(define new-inst
                   (filter
                     (λ (n)
                       (not (= (remainder n (first lon)) 0)))
                     (rest lon)))]
      (cons (first lon) (sieve new-inst limit))))
```

Intro to Generative Recursion

All Primes $\leq$ n

- ```
  #|
      Every recursive call is made with a shorter list given
      that at the very least the first element of the given
      list is removed. In addition, with every recursive
      call made the first element of the list becomes larger
      when the list is nonempty. These observations put
      together mean that the eventually list becomes empty or
      the first element of the list becomes larger than the
      given limit value and the function halts.
   |#
  ```
  ```

Intro to Generative Recursion

All Primes $\leq$ n

- ```
  #|
      Every recursive call is made with a shorter list given
      that at the very least the first element of the given
      list is removed. In addition, with every recursive
      call made the first element of the list becomes larger
      when the list is nonempty. These observations put
      together mean that the eventually list becomes empty or
      the first element of the list becomes larger than the
      given limit value and the function halts.
  |#
  ```
- Run the tests and make sure they all pass
  ```

Intro to Generative Recursion

All Primes $\leq$ `n`

- Why should anybody care?

Intro to Generative Recursion

All Primes $\leq$ `n`

- Why should anybody care?
- ```
  (define T3 (time (all-primes<=n 50000)))
  (define T4 (time (the-primes<=n 50000)))
  ```
  ```

Intro to Generative Recursion

All Primes $\leq$ n

- Why should anybody care?
- ```
  (define T3 (time (all-primes<=n 50000)))
  (define T4 (time (the-primes<=n 50000)))
  ```
- ```
  cpu time: 25968 real time: 27559 gc time: 6859
  cpu time: 1359  real time: 1413  gc time: 78
  ```
- CPU time which is expressed in milliseconds
- The CPU time includes the garbage collection (gc) time
- Subtract the garbage collection time, we have that `all-primes<=n` computed the result in 19,099 milliseconds and `the-primes<=n` computed the result in 1281 milliseconds
- This about one order of magnitude faster

Intro to Generative Recursion

All Primes $\leq$ `n`

- Run the experiment multiple times

Intro to Generative Recursion

All Primes $\leq$ `n`

- Run the experiment multiple times
- You get different timing data

Intro to Generative Recursion

All Primes $\leq$ `n`

- Run the experiment multiple times
- You get different timing data
- This means that timing results are unreliable

Intro to Generative Recursion

All Primes $\leq$ n

- Run the experiment multiple times
- You get different timing data
- This means that timing results are unreliable
- What is needed is complexity analysis

Intro to Generative Recursion

All Primes $\leq$ `n`

- For `sieve`:
  - The first call requires about $\frac{n}{2}$ filtering steps to remove the even numbers
  - The next call requires at most $\frac{n}{3}$ filtering steps to remove the remaining multiples of 3
  - The next call requires at most $\frac{n}{5}$ filtering steps to remove the remaining multiples of 5 In general, the number of steps done by `sieve` is proportional to:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

Intro to Generative Recursion

All Primes $\leq$ n

- For sieve:
    - The first call requires about $\frac{n}{2}$ filtering steps to remove the even numbers
    - The next call requires at most $\frac{n}{3}$ filtering steps to remove the remaining multiples of 3
    - The next call requires at most $\frac{n}{5}$ filtering steps to remove the remaining multiples of 5 In general, the number of steps done by sieve is proportional to:

    $$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

- By factoring out n we obtain:

  $$n*(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$$

Intro to Generative Recursion

All Primes $\leq$ n

- For sieve:
  - The first call requires about $\frac{n}{2}$ filtering steps to remove the even numbers
  - The next call requires at most $\frac{n}{3}$ filtering steps to remove the remaining multiples of 3
  - The next call requires at most $\frac{n}{5}$ filtering steps to remove the remaining multiples of 5 In general, the number of steps done by sieve is proportional to:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

- By factoring out n we obtain:

$$n*(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$$

- Mathematicians have proven that the above series converges to: $\log(\log(n))$

Intro to Generative Recursion

All Primes $\leq$ n

- For sieve:
  - The first call requires about $\frac{n}{2}$ filtering steps to remove the even numbers
  - The next call requires at most $\frac{n}{3}$ filtering steps to remove the remaining multiples of 3
  - The next call requires at most $\frac{n}{5}$ filtering steps to remove the remaining multiples of 5 In general, the number of steps done by sieve is proportional to:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

- By factoring out n we obtain:

$$n*(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$$

- Mathematicians have proven that the above series converges to: $\log(\log(n))$
- This means that the abstract running time for sieve is: $O(n \log(\log(n)))$

Intro to Generative Recursion

All Primes $\leq$ `n`

- For `sieve`:
  - The first call requires about $\frac{n}{2}$ filtering steps to remove the even numbers
  - The next call requires at most $\frac{n}{3}$ filtering steps to remove the remaining multiples of 3
  - The next call requires at most $\frac{n}{5}$ filtering steps to remove the remaining multiples of 5 In general, the number of steps done by `sieve` is proportional to:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

- By factoring out `n` we obtain:

$$\texttt{n*}(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$$

- Mathematicians have proven that the above series converges to: $\log(\log(n))$
- This means that the abstract running time for `sieve` is: $O(n \log(\log(n)))$
- Grows much slower than $O(n^2)$

Intro to Generative Recursion

All Primes $\leq$ n

- For `sieve`:
  - The first call requires about $\frac{n}{2}$ filtering steps to remove the even numbers
  - The next call requires at most $\frac{n}{3}$ filtering steps to remove the remaining multiples of 3
  - The next call requires at most $\frac{n}{5}$ filtering steps to remove the remaining multiples of 5 In general, the number of steps done by `sieve` is proportional to:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

- By factoring out `n` we obtain:

$$\texttt{n}*(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$$

- Mathematicians have proven that the above series converges to: $\log(\log(n))$
- This means that the abstract running time for `sieve` is: $O(n \log(\log(n)))$
- Grows much slower than $O(n^2)$
- Now you truly understand why `the-primes<=n` runs much faster

Intro to Generative Recursion

All Primes $\leq$ n

- For sieve:
  - The first call requires about $\frac{n}{2}$ filtering steps to remove the even numbers
  - The next call requires at most $\frac{n}{3}$ filtering steps to remove the remaining multiples of 3
  - The next call requires at most $\frac{n}{5}$ filtering steps to remove the remaining multiples of 5 In general, the number of steps done by sieve is proportional to:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \ldots$$

- By factoring out n we obtain:

  $$\mathtt{n}*(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$$
- Mathematicians have proven that the above series converges to: $\log(\log(n))$
- This means that the abstract running time for sieve is: $O(n \log(\log(n)))$
- Grows much slower than $O(n^2)$
- Now you truly understand why the-primes<=n runs much faster
- Caution: It is not the case that generative recursion is always faster than structural recursion

Intro to Generative Recursion

HOMEWORK

- HW: 4-7
- QUIZ: 8 (due in 1 week)

Sorting

- Sorting has been studied extensively to make it as efficient as possible, because it improves the efficiency of solutions to other problems

Sorting

- Sorting has been studied extensively to make it as efficient as possible, because it improves the efficiency of solutions to other problems
- The result of any sorting algorithm must satisfy two properties:
  - The must be *monotonic*
  - The result must be a *permutation* of the input

Sorting

- Sorting has been studied extensively to make it as efficient as possible, because it improves the efficiency of solutions to other problems
- The result of any sorting algorithm must satisfy two properties:
  - The must be *monotonic*
  - The result must be a *permutation* of the input
- We shall explore different sorting designs

Sorting

- Sorting has been studied extensively to make it as efficient as possible, because it improves the efficiency of solutions to other problems
- The result of any sorting algorithm must satisfy two properties:
  - The must be *monotonic*
  - The result must be a *permutation* of the input
- We shall explore different sorting designs
- To test the different algorithms the following `lon`s are defined:

```
(define LON0 '())
(define LON1 '(71 81 21 28 72 19 49 64 4 47 81 4))
(define LON2 '(91 57 93 5 16 56 61 59 93 49 -3))
(define LON3 (build-list 2500 (λ (i) (- 100000 i))))
(define LON4 (build-list 200  (λ (i) (random 100000000))))
(define LON5 (build-list 1575 (λ (i) (random 10000000))))
(define LON6 (build-list 1575 (λ (i) i)))
```

- Properties of testing lists:
  - Sample `lon`s for both variants
  - `lon`s of even and odd length
  - A sorted `lon`
  - A `lon` in reversed order
  - Randomly generated `lon`s to protect ourselves from any possible bias

Sorting

Insertion Sorting

```
;; sort: lon → lon  Purpose: Sort given lon in nondecreasing order
(define (insertion-sorting a-lon)
  (local [;; insert: a-num lon → lon
          ;; Purpose: To insert a num into a lon sorted in
          ;;          non-decreasing order
          (define (insert a-num a-lon)
            (cond [(empty? a-lon) (cons a-num empty)]
                  [(<= a-num (first a-lon)) (cons a-num a-lon)]
                  [else (cons (first a-lon)
                                    (insert a-num (rest a-lon)))]))]
    (cond [(empty? a-lon) empty]
          [else (insert (first a-lon)
                        (insertion-sorting (rest a-lon)))])))
;; Tests using sample values for insertion-sorting
(check-expect (insertion-sorting LON0) '())
(check-expect (insertion-sorting LON1)
              (list 4 4 19 21 28 47 49 64 71 72 81 81))
(check-expect (insertion-sorting LON2)
              (list -3 5 16 49 56 57 59 61 91 93 93))
(check-expect (insertion-sorting LON3) (reverse LON3))
(check-satisfied (insertion-sorting LON4) is-sorted?)
(check-satisfied (insertion-sorting LON5) is-sorted?)
(check-expect (insertion-sorting LON6) LON6)
```

Sorting

Insertion Sorting

- Let us explore the performance of insertion sorting:

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0

- Does well for most of our sample lists

Sorting

Insertion Sorting

- Let us explore the performance of insertion sorting:

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0

- Does well for most of our sample lists
- What about LON3?

Sorting

Insertion Sorting

- Let us explore the performance of insertion sorting:

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0

- Does well for most of our sample lists
- What about `LON3`?
- Think carefully about what insertion sorting is doing. Let us consider the first call:

```
(insertion-sorting '(100000 ... 97501))
```

Sorting

Insertion Sorting

- Let us explore the performance of insertion sorting:

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0

- Does well for most of our sample lists
- What about LON3?
- Think carefully about what insertion sorting is doing. Let us consider the first call:

```
(insertion-sorting '(100000 ... 97501))
```

- Given that the list is not empty this call generates the following call:

```
(insert 100000 (insertion-sorting '(99999 ... 97501))
```

Sorting

Insertion Sorting

- Let us explore the performance of insertion sorting:

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0

- Does well for most of our sample lists
- What about LON3?
- Think carefully about what insertion sorting is doing. Let us consider the first call:

```
(insertion-sorting '(100000 ... 97501))
```

- Given that the list is not empty this call generates the following call:

```
(insert 100000 (insertion-sorting '(99999 ... 97501))
```

- Substituting the value of the recursive call yields:

```
(insert 100000 '(97501 ... 99999))
```

- Observe that inserting 100000 requires traversing the entire list returned by the recursive call
- This represents the worse-case scenario for insertion sorting and explains why insertion sorting is significantly slower when given LON3.

Sorting

Insertion Sorting

- Let us explore the performance of insertion sorting:

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0

- Does well for most of our sample lists
- What about LON3?
- Think carefully about what insertion sorting is doing. Let us consider the first call:

```
(insertion-sorting '(100000 ... 97501))
```

- Given that the list is not empty this call generates the following call:

```
(insert 100000 (insertion-sorting '(99999 ... 97501))
```

- Substituting the value of the recursive call yields:

```
(insert 100000 '(97501 ... 99999))
```

- Observe that inserting 100000 requires traversing the entire list returned by the recursive call
- This represents the worse-case scenario for insertion sorting and explains why insertion sorting is significantly slower when given LON3.
- The complexity is $O(n^2)$.

Sorting

Quick Sorting

- The weakness of insertion sorting stems from always inserting into a sorted list
- Ask yourself is this must always be done

Sorting

Quick Sorting

- The weakness of insertion sorting stems from always inserting into a sorted list
- Ask yourself is this must always be done
- The British computer scientist and 1980 Turing Award recipient, Sir Charles Antony Richard Hoare, observed that instead of finding the given list's first element's position after sorting the rest of the list we can find the first element's position and then sort the remaining elements

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?
- You cannot possibly know the index of the first element in the sorted result before sorting the rest of the elements, but you know the relative position

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?
- You cannot possibly know the index of the first element in the sorted result before sorting the rest of the elements, but you know the relative position
- In the sorted result the first element, called the *pivot*, goes between the elements that are less than or equal to it and the elements that are greater than it.

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?
- You cannot possibly know the index of the first element in the sorted result before sorting the rest of the elements, but you know the relative position
- In the sorted result the first element, called the *pivot*, goes between the elements that are less than or equal to it and the elements that are greater than it.
- Assuming `L = (pivot (rest L))` we can visualize this idea as follows:

```
(quick-sorting L)
=
(append (quick-sorting [<elements <= pivot in (rest L)])
        (cons pivot
              (quick-sorting [elements > pivot in (rest L)])))
```

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?
- You cannot possibly know the index of the first element in the sorted result before sorting the rest of the elements, but you know the relative position
- In the sorted result the first element, called the *pivot*, goes between the elements that are less than or equal to it and the elements that are greater than it.
- Assuming `L = (pivot (rest L))` we can visualize this idea as follows:

```
(quick-sorting L)
=
(append (quick-sorting [<elements <= pivot in (rest L)])
        (cons pivot
              (quick-sorting [elements > pivot in (rest L)])))
```

- A divide-and-conquer algorithm
- The rest of `L` is divided into two lists recursively sorted

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?
- You cannot possibly know the index of the first element in the sorted result before sorting the rest of the elements, but you know the relative position
- In the sorted result the first element, called the *pivot*, goes between the elements that are less than or equal to it and the elements that are greater than it.
- Assuming `L = (pivot (rest L))` we can visualize this idea as follows:

```
(quick-sorting L)
=
(append (quick-sorting [<elements <= pivot in (rest L)])
        (cons pivot
              (quick-sorting [elements > pivot in (rest L)])))
```

- A divide-and-conquer algorithm
- The rest of `L` is divided into two lists recursively sorted
- How is the recursion stopped?

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?
- You cannot possibly know the index of the first element in the sorted result before sorting the rest of the elements, but you know the relative position
- In the sorted result the first element, called the *pivot*, goes between the elements that are less than or equal to it and the elements that are greater than it.
- Assuming `L = (pivot (rest L))` we can visualize this idea as follows:

```
(quick-sorting L)
=
(append (quick-sorting [<elements <= pivot in (rest L)])
        (cons pivot
              (quick-sorting [elements > pivot in (rest L)])))
```

- A divide-and-conquer algorithm
- The rest of `L` is divided into two lists recursively sorted
- How is the recursion stopped?
- When the given list is empty the sorted list is empty

Sorting

Quick Sorting

- How can you possibly know the position of the first element before sorting the rest of the list?
- You cannot possibly know the index of the first element in the sorted result before sorting the rest of the elements, but you know the relative position
- In the sorted result the first element, called the *pivot*, goes between the elements that are less than or equal to it and the elements that are greater than it.
- Assuming `L = (pivot (rest L))` we can visualize this idea as follows:

```
(quick-sorting L)
=
(append (quick-sorting [<elements <= pivot in (rest L)])
        (cons pivot
              (quick-sorting [elements > pivot in (rest L)])))
```

- A divide-and-conquer algorithm
- The rest of `L` is divided into two lists recursively sorted
- How is the recursion stopped?
- When the given list is empty the sorted list is empty
- This is generative recursion

Sorting

Quick Sorting

- ```
  ;; Sample expressions for quick-sorting
  (define LONO-VAL '())
  ```
  ```

Sorting

Quick Sorting

- ```
  ;; Sample expressions for quick-sorting
  (define LON0-VAL '())
  ```
- ```
  (define LON1-VAL
          (local
             [(define SMALLER= (filter (λ (i) (<= i (first LON1)))
                                       (rest LON1)))
              (define GREATER  (filter (λ (i) (> i (first LON1)))
                                       (rest LON1)))]
  ```
- ```
              (append (quick-sorting SMALLER=)
                      (cons (first LON1) (quick-sorting GREATER)))))
  ```
  ```

Sorting

Quick Sorting

- ```
  ;; Sample expressions for quick-sorting
  (define LON0-VAL '())
  ```
- ```
  (define LON1-VAL
          (local
             [(define SMALLER= (filter (λ (i) (<= i (first LON1)))
                                       (rest LON1)))
              (define GREATER  (filter (λ (i) (> i (first LON1)))
                                       (rest LON1)))]
  ```
- ```
              (append (quick-sorting SMALLER=)
                      (cons (first LON1) (quick-sorting GREATER)))))
  ```
- ```
  (define LON2-VAL
          (local
             [(define SMALLER= (filter (λ (i) (<= i (first LON2)))
                                       (rest LON2)))
              (define GREATER  (filter (λ (i) (> i (first LON2)))
                                       (rest LON2)))]
            (append (quick-sorting SMALLER=)
                    (cons (first LON2) (quick-sorting GREATER)))))
  ```

Sorting

Quick Sorting

- ```
  ;; Sample expressions for quick-sorting
  (define LON0-VAL '())
  ```
- ```
  (define LON1-VAL
          (local
             [(define SMALLER= (filter (λ (i) (<= i (first LON1)))
                                       (rest LON1)))
              (define GREATER  (filter (λ (i) (> i (first LON1)))
                                       (rest LON1)))]
  ```
- ```
            (append (quick-sorting SMALLER=)
                    (cons (first LON1) (quick-sorting GREATER)))))
  ```
- ```
  (define LON2-VAL
          (local
             [(define SMALLER= (filter (λ (i) (<= i (first LON2)))
                                       (rest LON2)))
              (define GREATER  (filter (λ (i) (> i (first LON2)))
                                       (rest LON2)))]
            (append (quick-sorting SMALLER=)
                    (cons (first LON2) (quick-sorting GREATER)))))
  ```
- There is only one difference: the list being sorted

Sorting

Quick Sorting

- ```
  ;; lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  ```
  ```

Sorting

Quick Sorting

- ```
  ;; lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  ```
- ```
  ;; How: When the given list is empty stop and return the
  ;;   empty list. Otherwise, place the given's list
  ;;   first number between the sorted numbers less than
  ;;   or equal to the first number and the sorted numbers
  ;;   greater than the first number.
  (define (quick-sorting a-lon)
  ```

Sorting

Quick Sorting

```
;; Tests using sample computations for quick-sorting
(check-expect (quick-sorting LON0) LON0-VAL)
(check-expect (quick-sorting LON1) LON1-VAL)
(check-expect (quick-sorting LON2) LON2-VAL)
(check-expect (quick-sorting LON3) LON3-VAL)

;; Tests using sample values for quick-sorting
(check-satisfied (quick-sorting LON4) is-sorted?)
(check-satisfied (quick-sorting LON5) is-sorted?)
(check-expect    (quick-sorting LON6) LON6)
(check-expect    (quick-sorting '(74 83 -72 2))
                 '(-72 2 74 83))
```

Sorting

Quick Sorting

```
(if (empty? a-lon)
    '()
    (local [(define SMALLER= (filter
                               (λ (i) (<= i (first a-lon)))
                               (rest a-lon)))
            (define GREATER  (filter
                               (λ (i) (> i (first a-lon)))
                               (rest a-lon)))]
      (append (quick-sorting SMALLER=)
              (cons (first a-lon)
                    (quick-sorting GREATER)))))
```

Sorting

Quick Sorting

- Each recursive call is made with a `lon` that is at least one shorter than the given list
- The given list eventually becomes empty and the function halts

Sorting

Quick Sorting

-

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0
quick	0	0	1172	0	15	485

Sorting

Quick Sorting

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0
quick	0	0	1172	0	15	485

- Quick sorting is faster on `LON3` and most sample `lons`

Sorting

Quick Sorting

•

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0
quick	0	0	1172	0	15	485

- Quick sorting is faster on LON3 and most sample lons
- What about LON6?

Sorting

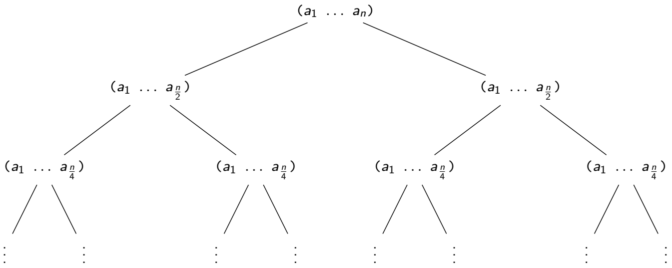
Quick Sorting

- Let us try to understand these numbers by performing complexity analysis

Sorting

Quick Sorting

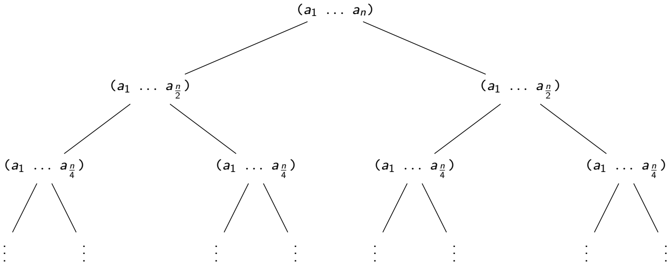
- Let us try to understand these numbers by performing complexity analysis
- Consider sorting a list of length n that always splits evenly. The calls generated may be visualized as follows:



Sorting

Quick Sorting

- Let us try to understand these numbers by performing complexity analysis
- Consider sorting a list of length `n` that always splits evenly. The calls generated may be visualized as follows:

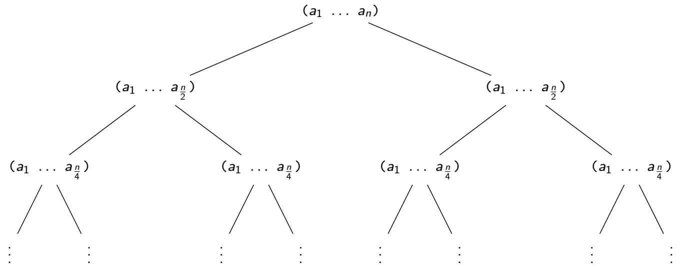


- Every time `quick-sorting` is called with:
  - Extracting the numbers less/greater than or equal to the pivot takes a number of operations proportional to `n`; $\frac{n}{2}$ for appending
  - The number of operations performed for every call to `quick-sorting` is proportional to `n + n +` $\frac{n}{2} = O(n)$

Sorting

Quick Sorting

- Let us try to understand these numbers by performing complexity analysis
- Consider sorting a list of length `n` that always splits evenly. The calls generated may be visualized as follows:

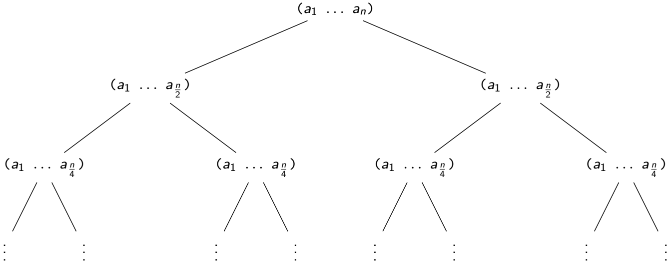


- Every time `quick-sorting` is called with:
  - Extracting the numbers less/greater than or equal to the pivot takes a number of operations proportional to `n`; $\frac{n}{2}$ for appending
  - The number of operations performed for every call to `quick-sorting` is proportional to `n + n +` $\frac{n}{2} = O(n)$
- How many operations are performed at each level of the binary tree?

Sorting

Quick Sorting

- Let us try to understand these numbers by performing complexity analysis
- Consider sorting a list of length `n` that always splits evenly. The calls generated may be visualized as follows:



- Every time `quick-sorting` is called with:
  - Extracting the numbers less/greater than or equal to the pivot takes a number of operations proportional to `n`; $\frac{n}{2}$ for appending
  - The number of operations performed for every call to `quick-sorting` is proportional to `n + n +` $\frac{n}{2} = O(n)$
- How many operations are performed at each level of the binary tree?
- At every level of the binary tree above $O(n)$ operations are performed

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed
- To establish the abstract running time we need to know the binary tree's height
- How many times is `n` divided in order for the quotient to be 0?

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed
- To establish the abstract running time we need to know the binary tree's height
- How many times is `n` divided in order for the quotient to be 0?
- ```
  (quotient 16 2) = 8
  (quotient  8 2) = 4
  (quotient  4 2) = 2
  (quotient  2 2) = 1
  (quotient  1 2) = 0
  ```
  16 may be divided by 2 5 times
  ```

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed
- To establish the abstract running time we need to know the binary tree's height
- How many times is `n` divided in order for the quotient to be 0?
- ```
  (quotient 16 2) = 8
  (quotient  8 2) = 4
  (quotient  4 2) = 2
  (quotient  2 2) = 1
  (quotient  1 2) = 0
  ```
  16 may be divided by 2 5 times
- Observe that 64 can be divided 2 7 times.
- What is the pattern?
  ```

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed
- To establish the abstract running time we need to know the binary tree's height
- How many times is `n` divided in order for the quotient to be 0?
- ```
  (quotient 16 2) = 8
  (quotient  8 2) = 4
  (quotient  4 2) = 2
  (quotient  2 2) = 1
  (quotient  1 2) = 0
  ```
  16 may be divided by 2 5 times
- Observe that 64 can be divided 2 7 times.
- What is the pattern?
- Observe that:

  $$lg(16) + 1 = 5$$

  $$lg(64) + 1 = 7$$
- In general, the number of times `n` is divided by 2 to reach 0 is $lg(n) + 1$.
  ```

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed
- To establish the abstract running time we need to know the binary tree's height
- How many times is `n` divided in order for the quotient to be 0?
- ```
  (quotient 16 2) = 8
  (quotient  8 2) = 4
  (quotient  4 2) = 2
  (quotient  2 2) = 1
  (quotient  1 2) = 0
  ```
  16 may be divided by 2 5 times
- Observe that 64 can be divided 2 7 times.
- What is the pattern?
- Observe that:

  $$lg(16) + 1 = 5$$

  $$lg(64) + 1 = 7$$
- In general, the number of times `n` is divided by 2 to reach 0 is $lg(n) + 1$.
- This means that the height of the binary tree above is $O(\lg(n))$.
  ```

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed
- To establish the abstract running time we need to know the binary tree's height
- How many times is `n` divided in order for the quotient to be 0?
-
```
(quotient 16 2) = 8
(quotient  8 2) = 4
(quotient  4 2) = 2
(quotient  2 2) = 1
(quotient  1 2) = 0
```
  16 may be divided by 2 5 times
- Observe that 64 can be divided 2 7 times.
- What is the pattern?
- Observe that:

  $lg(16)$ + 1 = 5

  $lg(64)$ + 1 = 7
- In general, the number of times `n` is divided by 2 to reach 0 is $lg(n)$ + 1.
- This means that the height of the binary tree above is $O(\lg(n))$.
- The abstract running time for quick sorting is $O(n * \lg(n))$.

Sorting

Quick Sorting

- At every level of the binary tree above $O(n)$ operations are performed
- To establish the abstract running time we need to know the binary tree's height
- How many times is `n` divided in order for the quotient to be 0?
- ```
  (quotient 16 2) = 8
  (quotient  8 2) = 4
  (quotient  4 2) = 2
  (quotient  2 2) = 1
  (quotient  1 2) = 0
  ```
  16 may be divided by 2 5 times
- Observe that 64 can be divided 2 7 times.
- What is the pattern?
- Observe that:

  $lg(16)$ + 1 = 5

  $lg(64)$ + 1 = 7
- In general, the number of times `n` is divided by 2 to reach 0 is $lg(n)$ + 1.
- This means that the height of the binary tree above is $O(\lg(n))$.
- The abstract running time for quick sorting is $O(n * \lg(n))$.
- This is much better that `insertion-sorting`'s $O(n^2)$
  ```

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?
- Let us consider the calls made when the given list is sorted:

```
(quick-sorting '(1 2 3 4))
```

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?
- Let us consider the calls made when the given list is sorted:

```
(quick-sorting '(1 2 3 4))
```

- This leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 1 (quick-sorting '(2 3 4))))
```

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?
- Let us consider the calls made when the given list is sorted:

```
(quick-sorting '(1 2 3 4))
```

- This leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 1 (quick-sorting '(2 3 4))))
```

- In turn this leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 2 (quick-sorting '(3 4))))
```

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?
- Let us consider the calls made when the given list is sorted:

```
(quick-sorting '(1 2 3 4))
```

- This leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 1 (quick-sorting '(2 3 4))))
```

- In turn this leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 2 (quick-sorting '(3 4))))
```

- The right recursive call leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 3 (quick-sorting '(4))))
```

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?
- Let us consider the calls made when the given list is sorted:

```
(quick-sorting '(1 2 3 4))
```

- This leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 1 (quick-sorting '(2 3 4))))
```

- In turn this leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 2 (quick-sorting '(3 4))))
```

- The right recursive call leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 3 (quick-sorting '(4))))
```

- The recursive call with 4 leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 4 (quick-sorting '())))
```

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?
- Let us consider the calls made when the given list is sorted:

```
(quick-sorting '(1 2 3 4))
```

- This leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 1 (quick-sorting '(2 3 4))))
```

- In turn this leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 2 (quick-sorting '(3 4))))
```

- The right recursive call leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 3 (quick-sorting '(4))))
```

- The recursive call with 4 leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 4 (quick-sorting '())))
```

- Observe that the argument to the left recursive call is always empty
- This is the worst-case scenario when we hope to divide the list evenly
- This means that the height of the binary tree describing the calls made to `quick-sorting` is `n` (not lg (n))

Sorting

Quick Sorting

- Why is `quick-sorting` slower when given a sorted or a in reversed sorted order list?
- Let us consider the calls made when the given list is sorted:

```
(quick-sorting '(1 2 3 4))
```

- This leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 1 (quick-sorting '(2 3 4))))
```

- In turn this leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 2 (quick-sorting '(3 4))))
```

- The right recursive call leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 3 (quick-sorting '(4))))
```

- The recursive call with 4 leads to the evaluation of:

```
(append (quick-sorting '())
        (cons 4 (quick-sorting '())))
```

- Observe that the argument to the left recursive call is always empty
- This is the worst-case scenario when we hope to divide the list evenly
- This means that the height of the binary tree describing the calls made to `quick-sorting` is `n` (not lg (n))
- This makes the abstract running time $O(n * O(n)) = O(n^2)$
- The same abstract running time as `insertion-sorting`!

Sorting

Quick Sorting

HOMEWORK: 1-2

Sorting

Merge Sorting

- Is there any way to always sort a list of numbers in quick sort's best-case of $O(n * \lg(n))$ steps?

Sorting

Merge Sorting

- John von Neumann: start with `n` `lons` of length 1

Sorting

Merge Sorting

- John von Neumann: start with `n` `lons` of length 1
- Repeatedly merge adjacent sublists until there is a single sublist

Sorting

Merge Sorting

- John von Neumann: start with `n` `lon`s of length `1`
- Repeatedly merge adjacent sublists until there is a single sublist
- Guarantees that the empty `lon` is never combined with a nonempty `lon`

Sorting

Merge Sorting

- John von Neumann: start with `n` `lons` of length 1
- Repeatedly merge adjacent sublists until there is a single sublist
- Guarantees that the empty `lon` is never combined with a nonempty `lon`
- The idea may be summarized as follows:
  1. Convert a `lon`, `L`, of length `n` into a `(listof lon)`, where each sub`lon` has length 1
  2. Repeatedly merge adjacent sub`lon`s until the `(listof lon)` is of length 1.
- When the `(listof lon)` is of length 1 it contains `L` sorted

Sorting

The `merge-sorting` Function

- Only a nonempty `lon` may be converted to a `(listof lon)` with all sublons having length 1

Sorting

The `merge-sorting` Function

- Only a nonempty `lon` may be converted to a `(listof lon)` with all sublons having length 1
- The `merge-sorting` function must determine if the list is empty

Sorting

The merge-sorting Function

- Only a nonempty lon may be converted to a (listof lon) with all sublons having length 1
- The merge-sorting function must determine if the list is empty
- If the given lon is the empty list the result is the empty

Sorting

The `merge-sorting` Function

- Only a nonempty `lon` may be converted to a `(listof lon)` with all sublons having length 1
- The `merge-sorting` function must determine if the list is empty
- If the given `lon` is the empty list the result is the empty
- If the given list is not empty then the given list is converted into a `(listof lon)`
- Processing a `(listof lon)` is a different problem and, therefore, an auxiliary function is needed

Sorting

The `merge-sorting` Function

- Only a nonempty `lon` may be converted to a `(listof lon)` with all sublons having length 1
- The `merge-sorting` function must determine if the list is empty
- If the given `lon` is the empty list the result is the empty
- If the given list is not empty then the given list is converted into a `(listof lon)`
- Processing a `(listof lon)` is a different problem and, therefore, an auxiliary function is needed
- This auxiliary function must return a `(listof lon)` of length 1
- The `merge-sorting` function returns the single element

Sorting

The merge-sorting Function

- ```
  ;; Sample expressions for merge-sorting
  (define MS-LON0-VAL '())
  ```
  ```

Sorting

The merge-sorting Function

- ```
  ;; Sample expressions for merge-sorting
  (define MS-LON0-VAL '())
  ```
- ```
  (define MS-LON1-VAL (first (merge-sort-helper
                               (map (λ (n) (list n)) LON1))))
  (define MS-LON2-VAL (first (merge-sort-helper
                               (map (λ (n) (list n)) LON2))))
  (define MS-LON3-VAL (first (merge-sort-helper
                               (map (λ (n) (list n)) LON3))))
  ```

Sorting

The merge-sorting Function

- ```
  ;; Sample expressions for merge-sorting
  (define MS-LON0-VAL '())
  ```
- ```
  (define MS-LON1-VAL (first (merge-sort-helper
                               (map (λ (n) (list n)) LON1))))
  (define MS-LON2-VAL (first (merge-sort-helper
                               (map (λ (n) (list n)) LON2))))
  (define MS-LON3-VAL (first (merge-sort-helper
                               (map (λ (n) (list n)) LON3))))
  ```
- The only difference is the list to sort

Sorting

The merge-sorting Function

- ```
  ;; lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  (define (merge-sorting a-lon)
  ```
  ```

Sorting

The merge-sorting Function

- ```
  ;; lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  (define (merge-sorting a-lon)
  ```

- ```
  ;; Tests using sample values for merge-sorting
  (check-expect (merge-sorting LON0) MS-LON0-VAL)
  (check-expect (merge-sorting LON1) MS-LON1-VAL)
  (check-expect (merge-sorting LON2) MS-LON2-VAL)
  (check-expect (merge-sorting LON3) MS-LON3-VAL)

  ;; Tests using sample values for merge-sorting
  (check-satisfied (merge-sorting LON4) is-sorted?)
  (check-satisfied (merge-sorting LON5) is-sorted?)
  (check-expect    (merge-sorting LON6) LON6)
  (check-expect    (merge-sorting '(74 83 -72 2))
                   '(-72 2 74 83))
  ```

Sorting

The merge-sorting Function

- ```
  ;; lon → lon
  ;; Purpose: Sort given lon in nondecreasing order
  (define (merge-sorting a-lon)
  ```
- ```
    (if (empty? a-lon)
        '()
        (first (merge-sort-helper (map (λ (n) (list n))
                                       a-lon)))))
  ```
- ```
  ;; Tests using sample values for merge-sorting
  (check-expect (merge-sorting LON0) MS-LON0-VAL)
  (check-expect (merge-sorting LON1) MS-LON1-VAL)
  (check-expect (merge-sorting LON2) MS-LON2-VAL)
  (check-expect (merge-sorting LON3) MS-LON3-VAL)

  ;; Tests using sample values for merge-sorting
  (check-satisfied (merge-sorting LON4) is-sorted?)
  (check-satisfied (merge-sorting LON5) is-sorted?)
  (check-expect    (merge-sorting LON6) LON6)
  (check-expect    (merge-sorting '(74 83 -72 2))
                   '(-72 2 74 83))
  ```
  ```

Sorting

The `merge-sort-helper` Function

- Repeatedly merge pairs of neighboring `lons` until there is a single `lon` left

Sorting

The `merge-sort-helper` Function

- Repeatedly merge pairs of neighboring `lons` until there is a single `lon` left
- The given `(listof lon)` cannot be empty
- The halting condition for the recursion is when the length of the given `(listof lon)` is 1.

Sorting

The `merge-sort-helper` Function

- Repeatedly merge pairs of neighboring `lon`s until there is a single `lon` left
- The given `(listof lon)` cannot be empty
- The halting condition for the recursion is when the length of the given `(listof lon)` is 1.
- What if the length of the given `lon` is greater than 1?

Sorting

The `merge-sort-helper` Function

- Repeatedly merge pairs of neighboring `lon`s until there is a single `lon` left
- The given `(listof lon)` cannot be empty
- The halting condition for the recursion is when the length of the given `(listof lon)` is 1.
- What if the length of the given `lon` is greater than 1?
- Merge pairs of neighboring `lon`s and process recursively
- This is generative recursion.

Sorting

The merge-sort-helper Function

- Repeatedly merge pairs of neighboring `lon`s until there is a single `lon` left
- The given `(listof lon)` cannot be empty
- The halting condition for the recursion is when the length of the given `(listof lon)` is 1.
- What if the length of the given `lon` is greater than 1?
- Merge pairs of neighboring `lon`s and process recursively
- This is generative recursion.
- Testing `lon`s:

```
(define LOLON1 (list (list 1 2 3 4)))
(define LOLON2 (list (list -6 8 10 67)))
(define LOLON3 (list (list 1 2 3) (list -4 0 8 74) (list 5)))
(define LOLON4 (list (list 76 89 99) (list -77) (list 5 8 9)))
```

Sorting

The merge-sort-helper Function

- ```
  ;; Sample expressions for merge-sort-helper
  (define MSH-LOLON1-VAL LOLON1)
  (define MSH-LOLON2-VAL LOLON2)
  ```
  ```

Sorting

The merge-sort-helper Function

- ```
  ;; Sample expressions for merge-sort-helper
  (define MSH-LOLON1-VAL LOLON1)
  (define MSH-LOLON2-VAL LOLON2)
  ```
- ```
  (define MSH-LOLON3-VAL
            (local [(define NEW-LOLON (merge-neighs LOLON3))]
              (merge-sort-helper NEW-LOLON)))
  (define MSH-LOLON4-VAL
            (local [(define NEW-LOLON (merge-neighs LOLON4))]
              (merge-sort-helper NEW-LOLON)))
  ```

Sorting

The merge-sort-helper Function

- ```
  ;; (listof lon) → (listof lon)
  ;; Purpose: Sort the numbers in the given (listof lon)
  ```
  ```

Sorting

The `merge-sort-helper` Function

- ```
  ;; (listof lon) → (listof lon)
  ;; Purpose: Sort the numbers in the given (listof lon)
  ```
- ```
  ;; How: If the length of the given (listof lon) is 1 return it.
  ;;      Otherwise, every two neighboring lons are merged to
  ;;      create a new problem instance that is recursively
  ;;      processed.
  ;; Assumption: The given (listof lon) has a length greater or
  ;;             equal to 1 and all sublons are sorted in
  ;;             nondecreasing order.
  ```

Sorting

The merge-sort-helper Function

- ```
  ;; (listof lon) → (listof lon)
  ;; Purpose: Sort the numbers in the given (listof lon)
  ```
- ```
  ;; How: If the length of the given (listof lon) is 1 return it.
  ;;      Otherwise, every two neighboring lons are merged to
  ;;      create a new problem instance that is recursively
  ;;      processed.
  ;; Assumption: The given (listof lon) has a length greater or
  ;;             equal to 1 and all sublons are sorted in
  ;;             nondecreasing order.
  ```
- ```
      (define (merge-sort-helper a-lolon)
  ```
  ```

Sorting

The merge-sort-helper Function

- ```
  ;; Tests using sample computations for merge-sort-helper
  (check-expect (merge-sort-helper LOLON1) MSH-LOLON1-VAL)
  (check-expect (merge-sort-helper LOLON2) MSH-LOLON2-VAL)
  (check-expect (merge-sort-helper LOLON3) MSH-LOLON3-VAL)
  (check-expect (merge-sort-helper LOLON4) MSH-LOLON4-VAL)
  ```
  ```

Sorting

The merge-sort-helper Function

- ```
  ;; Tests using sample computations for merge-sort-helper
  (check-expect (merge-sort-helper LOLON1) MSH-LOLON1-VAL)
  (check-expect (merge-sort-helper LOLON2) MSH-LOLON2-VAL)
  (check-expect (merge-sort-helper LOLON3) MSH-LOLON3-VAL)
  (check-expect (merge-sort-helper LOLON4) MSH-LOLON4-VAL)
  ```
- ```
  ;; Tests using sample values for merge-sort-helper
  (check-expect (merge-sort-helper '((8) (7) (4)))
                '((4 7 8)))
  (check-expect (merge-sort-helper '((8 9) (-87) (-4 99 678)))
                '((-87 -4 8 9 99 678)))
  ```

Sorting

The merge-sort-helper Function

```
(if (= (length a-lolon) 1)
    a-lolon
    (local [(define NEW-LOLON (merge-neighs a-lolon))]
      (merge-sort-helper NEW-LOLON)))
```

Sorting

The `merge-sort-helper` Function

- Assume that the function `merge-neighs` terminates and works

Sorting

The `merge-sort-helper` Function

- Assume that the function `merge-neighs` terminates and works
- The value returned by `merge-neighs` is always shorter than `a-lolon` because neighboring `lons` are merged

Sorting

The `merge-sort-helper` Function

- Assume that the function `merge-neighs` terminates and works
- The value returned by `merge-neighs` is always shorter than `a-lolon` because neighboring `lons` are merged
- This means that `merge-sort-helper` is always called with a shorter `(listof lon)`

Sorting

The `merge-sort-helper` Function

- Assume that the function `merge-neighs` terminates and works
- The value returned by `merge-neighs` is always shorter than `a-lolon` because neighboring `lons` are merged
- This means that `merge-sort-helper` is always called with a shorter `(listof lon)`
- This shorter `(listof lon)` is never empty because merging neighbors in a `(listof lon)` of length greater than 1 never produces an empty `(listof lon)`

Sorting

The `merge-sort-helper` Function

- Assume that the function `merge-neighs` terminates and works
- The value returned by `merge-neighs` is always shorter than `a-lolon` because neighboring `lon`s are merged
- This means that `merge-sort-helper` is always called with a shorter `(listof lon)`
- This shorter `(listof lon)` is never empty because merging neighbors in a `(listof lon)` of length greater than 1 never produces an empty `(listof lon)`
- Given that the argument to `merge-sort-helper` is always shorter and never length 0 we may conclude that eventually `merge-sort-helper` is given a `(listof lon)` of length 1 and the function halts.

Sorting

The `merge-neighs` Function

- To merge two neighboring `lons` the given `(listof lon)` must be of at least length 2

Sorting

The `merge-neighs` Function

- To merge two neighboring `lon`s the given `(listof lon)` must be of at least length 2
- If the given `(listof lon)` is of length 0 or 1 then the answer is the given `(listof lon)`.

Sorting

The `merge-neighs` Function

- To merge two neighboring `lon`s the given `(listof lon)` must be of at least length 2
- If the given `(listof lon)` is of length 0 or 1 then the answer is the given `(listof lon)`.
- If the given `(listof lon)` has length 2 or greater the first two `lon`s need to be merged
- The neighbors in the `(listof lon)` remaining after removing the first two must be recursively merged.
- This is a generative recursive algorithm.

Sorting

The merge-neighs Function

- ```
  ;; Sample expressions for merge-neighs
  (define MN-LOLON1-VAL LOLON1)
  (define MN-LOLON2-VAL LOLON2)
  ```
  ```

Sorting

The merge-neighs Function

- ```
  ;; Sample expressions for merge-neighs
  (define MN-LOLON1-VAL LOLON1)
  (define MN-LOLON2-VAL LOLON2)
  ```
- ```
  (define MN-LOLON3-VAL
            (local [(define NEW-LOLON (rest (rest LOLON3)))]
               (cons (merge (first LOLON3) (second LOLON3))
                     (merge-neighs NEW-LOLON))))
  (define MN-LOLON4-VAL
            (local [(define NEW-LOLON (rest (rest LOLON4)))]
               (cons (merge (first LOLON4) (second LOLON4))
                     (merge-neighs NEW-LOLON))))
  ```

Sorting

The merge-neighs Function

- ```
  ;; Sample expressions for merge-neighs
  (define MN-LOLON1-VAL LOLON1)
  (define MN-LOLON2-VAL LOLON2)
  ```
- ```
  (define MN-LOLON3-VAL
            (local [(define NEW-LOLON (rest (rest LOLON3)))]
               (cons (merge (first LOLON3) (second LOLON3))
                     (merge-neighs NEW-LOLON))))
  (define MN-LOLON4-VAL
            (local [(define NEW-LOLON (rest (rest LOLON4)))]
               (cons (merge (first LOLON4) (second LOLON4))
                     (merge-neighs NEW-LOLON))))
  ```
- There ia a single difference: the `(listof lon)` processed

Sorting

The merge-neighs Function

- ```
  ;; (listof lon) → (listof lon)
  ;; Purpose: Merge every two adjacent lons in nondecreasing
  ;;          order
  ```
  ```

Sorting

The merge-neighs Function

- ```
  ;; (listof lon) → (listof lon)
  ;; Purpose: Merge every two adjacent lons in nondecreasing
  ;;          order
  ```
- ```
  ;; How: If the given (listof lon) has a length less
  ;;      than 2 there are no lons to merge and the
  ;;      answer is the given (listof lon). Otherwise,
  ;;      the merging of the first two lons is added
  ;;      to the front of the result of processing
  ;;      all the remaining lons after the first two.
  ;; Assumption: Nested lons are in nondecreasing order
  ```

Sorting

The merge-neighs Function

- ```
  ;; (listof lon) → (listof lon)
  ;; Purpose: Merge every two adjacent lons in nondecreasing
  ;;          order
  ```
- ```
  ;; How: If the given (listof lon) has a length less
  ;;      than 2 there are no lons to merge and the
  ;;      answer is the given (listof lon). Otherwise,
  ;;      the merging of the first two lons is added
  ;;      to the front of the result of processing
  ;;      all the remaining lons after the first two.
  ;; Assumption: Nested lons are in nondecreasing order
  ```
- ```
  (define (merge-neighs a-lolon)
  ```
  ```

Sorting

The merge-neighs Function

- ```
  ;; Tests using sample computations for merge-neighs
  (check-expect (merge-neighs LOLON1) MN-LOLON1-VAL)
  (check-expect (merge-neighs LOLON2) MN-LOLON2-VAL)
  (check-expect (merge-neighs LOLON3) MN-LOLON3-VAL)
  (check-expect (merge-neighs LOLON4) MN-LOLON4-VAL)
  ```
  ```

Sorting

The merge-neighs Function

- ```
  ;; Tests using sample computations for merge-neighs
  (check-expect (merge-neighs LOLON1) MN-LOLON1-VAL)
  (check-expect (merge-neighs LOLON2) MN-LOLON2-VAL)
  (check-expect (merge-neighs LOLON3) MN-LOLON3-VAL)
  (check-expect (merge-neighs LOLON4) MN-LOLON4-VAL)
  ```
- ```
  ;; Tests using sample values for merge-neighs
  (check-expect (merge-neighs '((5 8) (7 9) (-3)))
                '((5 7 8 9) (-3)))
  (check-expect (merge-neighs '((2) (-9) (-3) (8 10)))
                '((-9 2) (-3 8 10)))
  ```

Sorting

The merge-neighs Function

- ```
  (if (< (length a-lolon) 2)
      a-lolon
      (local [(define NEW-LOLON (rest (rest a-lolon)))]
        (cons (merge (first a-lolon) (second a-lolon))
              (merge-neighs NEW-LOLON))))
  ```
  ```

Sorting

The `merge-neighs` Function

- The function halts if the given `(listof lon)` has a length less than 2

Sorting

The `merge-neighs` Function

- The function halts if the given `(listof lon)` has a length less than 2
- When given a `(listof lon)` of greater length the first two elements are removed from it to make a recursive call

Sorting

The `merge-neighs` Function

- The function halts if the given `(listof lon)` has a length less than 2
- When given a `(listof lon)` of greater length the first two elements are removed from it to make a recursive call
- This means that the given `(listof lon)` eventually becomes empty if its length is even and eventually becomes a list of length 1 if its length is odd
- In both cases the function terminates because the length is less than 2.

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lon`s in nondecreasing order

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lon`s in nondecreasing order
- Does one input dominates the other?

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lon`s in nondecreasing order
- Does one input dominates the other?
- Process both inputs must be processed simultaneously?

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lon`s in nondecreasing order
- Does one input dominates the other?
- Process both inputs must be processed simultaneously?
- We must outline the relationship between the two given `lon`

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lon`s in nondecreasing order
- Does one input dominates the other?
- Process both inputs must be processed simultaneously?
- We must outline the relationship between the two given `lon`
- If one `lon` is empty then the answer is the other `lon`

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lons` in nondecreasing order
- Does one input dominates the other?
- Process both inputs must be processed simultaneously?
- We must outline the relationship between the two given `lon`
- If one `lon` is empty then the answer is the other `lon`
- If both `lons` are not empty then their first elements are compared and the smallest is added to the result

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lon`s in nondecreasing order
- Does one input dominates the other?
- Process both inputs must be processed simultaneously?
- We must outline the relationship between the two given `lon`
- If one `lon` is empty then the answer is the other `lon`
- If both `lon`s are not empty then their first elements are compared and the smallest is added to the result
- Recursive call made with the rest of the list that contributes its first element and the other list

Sorting

The `merge` Function

- Needs to create a sorted `lon` from two given `lon`s in nondecreasing order
- Does one input dominates the other?
- Process both inputs must be processed simultaneously?
- We must outline the relationship between the two given `lon`
- If one `lon` is empty then the answer is the other `lon`
- If both `lon`s are not empty then their first elements are compared and the smallest is added to the result
- Recursive call made with the rest of the list that contributes its first element and the other list
- This is structural recursion

Sorting

The merge Function

- Needs to create a sorted lon from two given lons in nondecreasing order
- Does one input dominates the other?
- Process both inputs must be processed simultaneously?
- We must outline the relationship between the two given lon
- If one lon is empty then the answer is the other lon
- If both lons are not empty then their first elements are compared and the smallest is added to the result
- Recursive call made with the rest of the list that contributes its first element and the other list
- This is structural recursion
- Testing lons:

```
(define SL1 '())
(define SL2 '(-98 -76 -8 -1))
(define SL3 '(-87 -28 -6 89))
(define SL4 '(6 7 31 87))
```

Sorting

The `merge` Function

- Four conditions that must be detected:
    1. The first `lon` is empty
    2. The second `lon` is empty
    3. The first element of the first `lon` is less than or equal to the first element of the second `lon`
    4. The first element of the second `lon` is less than the first element of the first `lon`
- Need sample expressions for each of the above four conditions

Sorting

The merge Function

- Four conditions that must be detected:
  1. The first lon is empty
  2. The second lon is empty
  3. The first element of the first lon is less than or equal to the first element of the second lon
  4. The first element of the second lon is less than the first element of the first lon
- Need sample expressions for each of the above four conditions
-

```
;; Sample expressions for merge
(define M-SL1-SL2-VAL SL2)    (define M-SL1-SL3-VAL SL3)
(define M-SL2-SL1-VAL SL2)    (define M-SL3-SL1-VAL SL3)
```

Sorting

The merge Function

- Four conditions that must be detected:
    1. The first lon is empty
    2. The second lon is empty
    3. The first element of the first lon is less than or equal to the first element of the second lon
    4. The first element of the second lon is less than the first element of the first lon
- Need sample expressions for each of the above four conditions
- ```
  ;; Sample expressions for merge
  (define M-SL1-SL2-VAL SL2)   (define M-SL1-SL3-VAL SL3)
  (define M-SL2-SL1-VAL SL2)   (define M-SL3-SL1-VAL SL3)
  ```
- ```
  (define M-SL2-SL3-VAL (cons (first SL2)
                              (merge (rest SL2) SL3)))
  (define M-SL3-SL4-VAL (cons (first SL3)
                              (merge (rest SL3) SL4)))
  ```

Sorting

The merge Function

- Four conditions that must be detected:
  1. The first `lon` is empty
  2. The second `lon` is empty
  3. The first element of the first `lon` is less than or equal to the first element of the second `lon`
  4. The first element of the second `lon` is less than the first element of the first `lon`
- Need sample expressions for each of the above four conditions
- ```
  ;; Sample expressions for merge
  (define M-SL1-SL2-VAL SL2)    (define M-SL1-SL3-VAL SL3)
  (define M-SL2-SL1-VAL SL2)    (define M-SL3-SL1-VAL SL3)
  ```
- ```
  (define M-SL2-SL3-VAL (cons (first SL2)
                              (merge (rest SL2) SL3)))
  (define M-SL3-SL4-VAL (cons (first SL3)
                              (merge (rest SL3) SL4)))
  ```
- ```
  (define M-SL4-SL3-VAL (cons (first SL3)
                              (merge SL4 (rest SL3))))
  (define M-SL3-SL2-VAL (cons (first SL2)
                              (merge SL3 (rest SL2))))
  ```
  ```

Sorting

The merge Function

- Four conditions that must be detected:
  1. The first lon is empty
  2. The second lon is empty
  3. The first element of the first lon is less than or equal to the first element of the second lon
  4. The first element of the second lon is less than the first element of the first lon
- Need sample expressions for each of the above four conditions
- ```
  ;; Sample expressions for merge
  (define M-SL1-SL2-VAL SL2)   (define M-SL1-SL3-VAL SL3)
  (define M-SL2-SL1-VAL SL2)   (define M-SL3-SL1-VAL SL3)
  ```
- ```
  (define M-SL2-SL3-VAL (cons (first SL2)
                              (merge (rest SL2) SL3)))
  (define M-SL3-SL4-VAL (cons (first SL3)
                              (merge (rest SL3) SL4)))
  ```
- ```
  (define M-SL4-SL3-VAL (cons (first SL3)
                              (merge SL4 (rest SL3))))
  (define M-SL3-SL2-VAL (cons (first SL2)
                              (merge SL3 (rest SL2))))
  ```
- Differences: the two lons processed
  ```

Sorting

The merge Function

```
;; lon lon → lon
;; Purpose: Merge the given lons in nondecreasing order
;; Assumption: Given lons are in nondecreasing order
(define (merge l1 l2)
```

Sorting

The merge Function

- ```
  ;; Tests using sample computations for merge
  (check-expect (merge SL1 SL2) M-SL1-SL2-VAL)
  (check-expect (merge SL1 SL3) M-SL1-SL3-VAL)
  (check-expect (merge SL2 SL1) M-SL2-SL1-VAL)
  (check-expect (merge SL3 SL1) M-SL3-SL1-VAL)
  (check-expect (merge SL2 SL3) M-SL2-SL3-VAL)
  (check-expect (merge SL3 SL4) M-SL3-SL4-VAL)
  (check-expect (merge SL4 SL3) M-SL4-SL3-VAL)
  (check-expect (merge SL3 SL2) M-SL3-SL2-VAL)
  ```
  ```

Sorting

The merge Function

- ```
  ;; Tests using sample computations for merge
  (check-expect (merge SL1 SL2) M-SL1-SL2-VAL)
  (check-expect (merge SL1 SL3) M-SL1-SL3-VAL)
  (check-expect (merge SL2 SL1) M-SL2-SL1-VAL)
  (check-expect (merge SL3 SL1) M-SL3-SL1-VAL)
  (check-expect (merge SL2 SL3) M-SL2-SL3-VAL)
  (check-expect (merge SL3 SL4) M-SL3-SL4-VAL)
  (check-expect (merge SL4 SL3) M-SL4-SL3-VAL)
  (check-expect (merge SL3 SL2) M-SL3-SL2-VAL)
  ```
- ```
  ;; Tests using sample values for merge
  (check-expect (merge '() '()) '())
  (check-expect (merge '() '(7 8 9)) '(7 8 9))
  (check-expect (merge '(78 98) '()) '(78 98))
  (check-expect (merge '(1 2 3) '(4 5 6)) '(1 2 3 4 5 6))
  (check-expect (merge '(0 88) '(-5 8 17)) '(-5 0 8 17 88))
  ```

Sorting

The merge Function

```
(cond [(empty? l1) l2]
      [(empty? l2) l1]
      [(<= (first l1) (first l2))
       (cons (first l1) (merge (rest l1) l2))]
      [else (cons (first l2) (merge l1 (rest l2)))])
```

Sorting

The `merge` Function

- Does `merge-sorting` perform better than `quick-` and `insertion-sorting`?

Sorting

The merge Function

- Does merge-sorting perform better than quick- and insertion-sorting?
- The CPU time for merge-sorting is added in the following table:

	LON1	LON2	LON3	LON4	LON5	LON6
insertion	0	0	1953	31	15	0
quick	0	0	1172	0	15	485
merge	0	0	15	0	0	15

- Two conclusions:
  - Merge sorting performs better than quick sorting on a list that is sorted (i.e., LON6) and on a list that is in reverse sorted order (i.e., LON3)
  - For other types of lists quick sorting is faster or just as good as merge sorting.

Sorting

The `merge-sort-helper` Function

- For the call to `sort` `n` steps are taken to convert the given `lon` into a `(listof lon)`

Sorting

The `merge-sort-helper` Function

- For the call to `sort` `n` steps are taken to convert the given `lon` into a `(listof lon)`
- Length `n` to $\frac{n}{2}$: $\frac{n}{2} * 2$ steps

Sorting

The merge-sort-helper Function

- For the call to sort n steps are taken to convert the given lon into a (listof lon)
- Length n to $\frac{n}{2}$: $\frac{n}{2}$ * 2 steps
- Length $\frac{n}{2}$ to $\frac{n}{4}$: $\frac{n}{4}$ * 4 steps

Sorting

The merge-sort-helper Function

- For the call to sort n steps are taken to convert the given lon into a (listof lon)
- Length n to $\frac{n}{2}$: $\frac{n}{2}$ * 2 steps
- Length $\frac{n}{2}$ to $\frac{n}{4}$: $\frac{n}{4}$ * 4 steps
- Length $\frac{n}{4}$ to $\frac{n}{8}$: $\frac{n}{8}$ * 8 steps

Sorting

The `merge-sort-helper` Function

- For the call to `sort` `n` steps are taken to convert the given `lon` into a `(listof lon)`
- Length `n` to $\frac{n}{2}$: $\frac{n}{2}$ * 2 steps
- Length $\frac{n}{2}$ to $\frac{n}{4}$: $\frac{n}{4}$ * 4 steps
- Length $\frac{n}{4}$ to $\frac{n}{8}$: $\frac{n}{8}$ * 8 steps
- In general, `n` steps are required to reduce the list's length is half in the worst case

Sorting

The `merge-sort-helper` Function

- For the call to `sort` `n` steps are taken to convert the given `lon` into a `(listof lon)`
- Length `n` to $\frac{n}{2}$: $\frac{n}{2}$ * 2 steps
- Length $\frac{n}{2}$ to $\frac{n}{4}$: $\frac{n}{4}$ * 4 steps
- Length $\frac{n}{4}$ to $\frac{n}{8}$: $\frac{n}{8}$ * 8 steps
- In general, `n` steps are required to reduce the list's length is half in the worst case
- How many times `sort` called?

Sorting

The merge-sort-helper Function

- For the call to sort n steps are taken to convert the given lon into a (listof lon)
- Length n to $\frac{n}{2}$: $\frac{n}{2} * 2$ steps
- Length $\frac{n}{2}$ to $\frac{n}{4}$: $\frac{n}{4} * 4$ steps
- Length $\frac{n}{4}$ to $\frac{n}{8}$: $\frac{n}{8} * 8$ steps
- In general, n steps are required to reduce the list's length is half in the worst case
- How many times sort called?
- The number of calls is proportional to the number of times n can be divided by 2 before becoming 1: $O(\lg(n))$

Sorting

The merge-sort-helper Function

- For the call to sort n steps are taken to convert the given lon into a (listof lon)
- Length n to $\frac{n}{2}$: $\frac{n}{2}$ * 2 steps
- Length $\frac{n}{2}$ to $\frac{n}{4}$: $\frac{n}{4}$ * 4 steps
- Length $\frac{n}{4}$ to $\frac{n}{8}$: $\frac{n}{8}$ * 8 steps
- In general, n steps are required to reduce the list's length is half in the worst case
- How many times sort called?
- The number of calls is proportional to the number of times n can be divided by 2 before becoming 1: $O(\lg(n))$
- merge-sorting abstract running time is $O(n * \lg(n))$

Sorting

The `merge-sort-helper` Function

- For the call to `sort` `n` steps are taken to convert the given `lon` into a `(listof lon)`
- Length `n` to $\frac{n}{2}$: $\frac{n}{2}$ * 2 steps
- Length $\frac{n}{2}$ to $\frac{n}{4}$: $\frac{n}{4}$ * 4 steps
- Length $\frac{n}{4}$ to $\frac{n}{8}$: $\frac{n}{8}$ * 8 steps
- In general, `n` steps are required to reduce the list's length is half in the worst case
- How many times `sort` called?
- The number of calls is proportional to the number of times `n` can be divided by 2 before becoming 1: $O(\lg(n))$
- `merge-sorting` abstract running time is $O(n * \lg(n))$
- Now we truly understand why merge sorting performs better than quick sorting when the given list is sorted or is reversed sorted order

Sorting
HOMEWORK

- QUIZ (due in 1 week): 4
- HW: 5

Searching

- A search problem attempts to find a value `x` with property `P` in a set `S`

Searching

- A search problem attempts to find a value `x` with property `P` in a set `S`
- If there is an `x`$\in$`S` that satisfies `P` then the algorithm returns `x` or `#true`
- Otherwise, the algorithm returns `#false` or throws an error if appropriate

Searching

- A search problem attempts to find a value `x` with property `P` in a set `S`
- If there is an `x`$\in$`S` that satisfies `P` then the algorithm returns `x` or `#true`
- Otherwise, the algorithm returns `#false` or throws an error if appropriate
- Searching is extensively studied because it is a common operation

Searching

- A search problem attempts to find a value `x` with property `P` in a set `S`
- If there is an `x`∈`S` that satisfies `P` then the algorithm returns `x` or `#true`
- Otherwise, the algorithm returns `#false` or throws an error if appropriate
- Searching is extensively studied because it is a common operation
- Fundamental to implementing player help in the N-puzzle game

Searching

Linear Searching

- Remember finding the index of a number in a `lon`?

Searching

Linear Searching

- Remember finding the index of a number in a `lon`?
- Cases:
  1. Given list is empty
  2. Given list's first number equals given number
  3. Given list's rest does not contain the given number
  4. Given list's rest contains the given number

Searching

Linear Searching

- Remember finding the index of a number in a `lon`?
- Cases:
  1. Given list is empty
  2. Given list's first number equals given number
  3. Given list's rest does not contain the given number
  4. Given list's rest contains the given number
-
```
(define L0 '())
(define L1 '(88 54 4 7 87 98 -7 0 -1))
(define L2 '(9 8 7 6 5  4  3  2 1  0 -1 -2))
(define L3 (build-list 1000000 (λ (i) (random 1000000))))
(define L4 (build-list 1000000 (λ (i) i)))
```

Searching

Linear Searching

- We arbitrarily return the smallest such index

Searching

Linear Searching

- We arbitrarily return the smallest such index
- ```
  ;; A result, res, is either:
  ;;   1. natnum
  ;;   2. #false
  ```
  ```

Searching

Linear Searching

- Searching '()

```
;; Sample expressions for linear-search
(define LS-LO-VAL  #false)
```

Searching

Linear Searching

- Searching '()

```
;; Sample expressions for linear-search
(define LS-L0-VAL  #false)
```

- First number is equal to given number

```
(define LS-L1-VAL1 0)     (define LS-L2-VAL1 0)
```

Searching

Linear Searching

- Searching '()

```
;; Sample expressions for linear-search
(define LS-L0-VAL  #false)
```

- First number is equal to given number

```
(define LS-L1-VAL1 0)     (define LS-L2-VAL1 0)
```

- Given number is not found

```
(define LS-L1-VAL2
        (local
         [(define result-of-rest (linear-search -9 (rest L1)))]
         (if (false? result-of-rest)
             #false
             (add1 result-of-rest))))

     ⋮
```

Searching

Linear Searching

- Searching '()

```
;; Sample expressions for linear-search
(define LS-L0-VAL  #false)
```

- First number is equal to given number

```
(define LS-L1-VAL1 0)     (define LS-L2-VAL1 0)
```

- Given number is not found

```
(define LS-L1-VAL2
        (local
         [(define result-of-rest (linear-search -9 (rest L1)))]
         (if (false? result-of-rest)
             #false
             (add1 result-of-rest))))

     ⋮
```

- Given number is found

```
(define LS-L1-VAL3
        (local
            [(define result-of-rest
                       (linear-search -7 (rest L1)))]
          (if (false? result-of-rest)
              #false
              (add1 result-of-rest))))

     ⋮
```

Searching

Linear Searching

```
;; number lon → res
;; Purpose: Return the index of the first occurrence of
;;          the given number if it is a member of the
;;          given list. Otherwise, return #false
(define (linear-search a-num a-lon)
```

Searching

Linear Searching

- ```
  ;; Tests using sample computations for linear-search
  (check-expect (linear-search 25 L0) LS-L0-VAL)
  (check-expect (linear-search 88 L1) LS-L1-VAL1)
  (check-expect (linear-search  9 L2) LS-L2-VAL1)
  (check-expect (linear-search -9 L1) LS-L1-VAL2)
  (check-expect (linear-search 54 L2) LS-L2-VAL2)
  (check-expect (linear-search -7 L1) LS-L1-VAL3)
  (check-expect (linear-search  2 L2) LS-L2-VAL3)
  ```
  ```

Searching

Linear Searching

- ```
  ;; Tests using sample computations for linear-search
  (check-expect (linear-search 25 L0) LS-L0-VAL)
  (check-expect (linear-search 88 L1) LS-L1-VAL1)
  (check-expect (linear-search  9 L2) LS-L2-VAL1)
  (check-expect (linear-search -9 L1) LS-L1-VAL2)
  (check-expect (linear-search 54 L2) LS-L2-VAL2)
  (check-expect (linear-search -7 L1) LS-L1-VAL3)
  (check-expect (linear-search  2 L2) LS-L2-VAL3)
  ```
- ```
  ;; Tests using sample values for linear-search
  (check-satisfied
   (linear-search 100 L3)
   (λ (a-res) (or (false? a-res)
                  (= (list-ref L3 a-res) 100))))
  (check-expect (linear-search 2 '(1 2 3)) 1)
  (check-expect (linear-search 5 '(1 2 3)) #false)
  (check-expect (linear-search 2000000 L4) #false)
  (check-expect (linear-search 998999  L4) 998999)
  ```

Searching

Linear Searching

```
(cond [(empty? a-lon) #false]
      [(= a-num (first a-lon)) 0]
      [else
       (local
         [(define result-of-rest
                    (linear-search a-num (rest a-lon)))]
         (if (false? result-of-rest)
             #false
             (add1 result-of-rest)))])
```

Searching

Linear Searching

- How well does it perform?

Searching

Linear Searching

- How well does it perform?
-
```
(define LSL0 (time (linear-search 83333 L0)))
(define LSL1 (time (linear-search 0 L1)))
(define LSL2 (time (linear-search 8 L2)))
(define LSL3 (time (linear-search (first L3) L3)))
(define LSL4 (time (linear-search 2000000 L4)))
```

Searching

Linear Searching

- How well does it perform?
- ```
  (define LSL0 (time (linear-search 83333 L0)))
  (define LSL1 (time (linear-search 0 L1)))
  (define LSL2 (time (linear-search 8 L2)))
  (define LSL3 (time (linear-search (first L3) L3)))
  (define LSL4 (time (linear-search 2000000 L4)))
  ```
-

	L0	L1	L2	L3	L4
linear-search	0	0	0	0	1359
  ```

Searching

Linear Searching

- How well does it perform?
-
```
(define LSL0 (time (linear-search 83333 L0)))
(define LSL1 (time (linear-search 0 L1)))
(define LSL2 (time (linear-search 8 L2)))
(define LSL3 (time (linear-search (first L3) L3)))
(define LSL4 (time (linear-search 2000000 L4)))
```

  -

	L0	L1	L2	L3	L4
`linear-search`	0	0	0	0	1359

- In the worst case `linear-search` must compare the given number with every element in the list

Searching

Linear Searching

- How well does it perform?
-
```
(define LSL0 (time (linear-search 83333 L0)))
(define LSL1 (time (linear-search 0 L1)))
(define LSL2 (time (linear-search 8 L2)))
(define LSL3 (time (linear-search (first L3) L3)))
(define LSL4 (time (linear-search 2000000 L4)))
```

  -

	L0	L1	L2	L3	L4
linear-search	0	0	0	0	1359

- In the worst case `linear-search` must compare the given number with every element in the list
- `linear-search` is $O(n)$
- This is why it is called linear search.

Searching

Binary Search

- Can sorting improve searching performance?

Searching
Binary Search

- Can sorting improve searching performance?
- Consider looking for a word in a dictionary

Searching

Binary Search

- Can sorting improve searching performance?
- Consider looking for a word in a dictionary
- Do you start the search with the first word starting with "a" and check every word until you find the word or reach the last word starting with "z"?

Searching

Binary Search

- Can sorting improve searching performance?
- Consider looking for a word in a dictionary
- Do you start the search with the first word starting with "a" and check every word until you find the word or reach the last word starting with "z"?
- Open the dictionary in the middle and decide whether or not the word you are searching for is on the opened page
- If so, you look at its definition
- If not, you decide to search for the word in either the first or second half
- The process is repeated with the chosen half until the word is found or the chosen half is empty

Searching

Binary Search

- Can sorting improve searching performance?
- Consider looking for a word in a dictionary
- Do you start the search with the first word starting with "a" and check every word until you find the word or reach the last word starting with "z"?
- Open the dictionary in the middle and decide whether or not the word you are searching for is on the opened page
- If so, you look at its definition
- If not, you decide to search for the word in either the first or second half
- The process is repeated with the chosen half until the word is found or the chosen half is empty
- At each step half of the remaining dictionary is eliminated from the search
- Compare this with linear searching

Searching

Binary Search

- The elements of a sorted `lon` are numbered by the valid indices into the list

Searching

Binary Search

- The elements of a sorted `lon` are numbered by the valid indices into the list
- If the list is of length `n` then the interval of valid indices is `[0..(n - 1)]`

Searching

Binary Search

- The elements of a sorted `lon` are numbered by the valid indices into the list
- If the list is of length `n` then the interval of valid indices is `[0..(n - 1)]`
- The `lon` defined by this interval must be searched for the given number

Searching

Binary Search

- The elements of a sorted `lon` are numbered by the valid indices into the list
- If the list is of length `n` then the interval of valid indices is `[0..(n - 1)]`
- The `lon` defined by this interval must be searched for the given number
- This means that the problem of searching a sorted `lon` can be cast as an interval-processing problem

Searching

Binary Search

- The elements of a sorted `lon` are numbered by the valid indices into the list
- If the list is of length `n` then the interval of valid indices is `[0..(n - 1)]`
- The `lon` defined by this interval must be searched for the given number
- This means that the problem of searching a sorted `lon` can be cast as an interval-processing problem
- Sorted sample `lons`:

```
(define L1S (sort L1 <))
(define L2S (sort L2 <))
(define L3S (sort L3 <))
```

Searching

Binary Search

- ```
  ;; Sample expressions for binary-search
  (define BS-L0-VAL  (bin-search 25 0 (sub1 (length L0))  L0))
  (define BS-L1-VAL1 (bin-search 88 0 (sub1 (length L1S)) L1S))
  (define BS-L2-VAL1 (bin-search  9 0 (sub1 (length L2S)) L2S))
  (define BS-L1-VAL2 (bin-search -9 0 (sub1 (length L1S)) L1S))
  (define BS-L2-VAL2 (bin-search 54 0 (sub1 (length L2S)) L2S))
  (define BS-L1-VAL3 (bin-search -7 0 (sub1 (length L1S)) L1S))
  (define BS-L2-VAL3 (bin-search  2 0 (sub1 (length L2S)) L2S))
  ```
  ```

Searching

Binary Search

- ```
  ;; Sample expressions for binary-search
  (define BS-L0-VAL  (bin-search 25 0 (sub1 (length L0))  L0))
  (define BS-L1-VAL1 (bin-search 88 0 (sub1 (length L1S)) L1S))
  (define BS-L2-VAL1 (bin-search  9 0 (sub1 (length L2S)) L2S))
  (define BS-L1-VAL2 (bin-search -9 0 (sub1 (length L1S)) L1S))
  (define BS-L2-VAL2 (bin-search 54 0 (sub1 (length L2S)) L2S))
  (define BS-L1-VAL3 (bin-search -7 0 (sub1 (length L1S)) L1S))
  (define BS-L2-VAL3 (bin-search  2 0 (sub1 (length L2S)) L2S))
  ```
- There are two differences among the sample expressions: the number searched for and the sorted list
  ```

Searching

Binary Search

- ```
  ;; number lon → res
  ;; Purpose: Return the index of the given number if it
  ;;          is a member of the given list. Otherwise,
  ;;          return #false
  ;; Assumption: The given lon is sorted in nondecreasing
  ;;             order
  (define (binary-search a-num a-lon)
  ```
  ```

Searching

Binary Search

- ```
  ;; number lon → res
  ;; Purpose: Return the index of the given number if it
  ;;          is a member of the given list. Otherwise,
  ;;          return #false
  ;; Assumption: The given lon is sorted in nondecreasing
  ;;             order
  (define (binary-search a-num a-lon)
  ```
- ```
  ;; Tests using sample computations for binary-search
  (check-expect (binary-search 25 L0)  BS-L0-VAL)
  (check-expect (binary-search 88 L1S) BS-L1-VAL1)
  (check-expect (binary-search  9 L2S) BS-L2-VAL1)
  (check-expect (binary-search -9 L1S) BS-L1-VAL2)
  (check-expect (binary-search 54 L2S) BS-L2-VAL2)
  (check-expect (binary-search -7 L1S) BS-L1-VAL3)
  (check-expect (binary-search  2 L2S) BS-L2-VAL3)
  ```

Searching

Binary Search

- ```
  ;; number lon → res
  ;; Purpose: Return the index of the given number if it
  ;;          is a member of the given list. Otherwise,
  ;;          return #false
  ;; Assumption: The given lon is sorted in nondecreasing
  ;;             order
  (define (binary-search a-num a-lon)
  ```
- ```
  ;; Tests using sample computations for binary-search
  (check-expect (binary-search 25 L0)  BS-L0-VAL)
  (check-expect (binary-search 88 L1S) BS-L1-VAL1)
  (check-expect (binary-search  9 L2S) BS-L2-VAL1)
  (check-expect (binary-search -9 L1S) BS-L1-VAL2)
  (check-expect (binary-search 54 L2S) BS-L2-VAL2)
  (check-expect (binary-search -7 L1S) BS-L1-VAL3)
  (check-expect (binary-search  2 L2S) BS-L2-VAL3)
  ```
- ```
  ;; Tests using sample values for binary-search
  (check-satisfied
   (binary-search 100 L3S)
   (λ (a-res) (or (false? a-res)
                  (= (list-ref L3 a-res) 100))))
  (check-expect (binary-search 2 '(1 2 3)) 1)
  (check-expect (binary-search 5 '(1 2 3)) #false)
  (check-expect (binary-search 2000000 L4) #false)
  (check-expect (binary-search 998999  L4) 998999)
  ```
  ```

Searching

The bin-search Function

```
(bin-search a-num 0 (sub1 (length a-lon)) a-lon)
```

Searching

The `bin-search` Function

- Searches a given list by traversing a given interval

Searching

The `bin-search` Function

- Searches a given list by traversing a given interval
- If the interval is empty then the answer is `#false`

Searching

The `bin-search` Function

- Searches a given list by traversing a given interval
- If the interval is empty then the answer is `#false`
- If the interval is not empty then the list element corresponding to the middle index in the given interval is compared with the number searched for
- If they are equal the middle index is returned as the answer
- If they are not equal then a decision must be made as to which new interval to search

Searching

The `bin-search` Function

- Searches a given list by traversing a given interval
- If the interval is empty then the answer is `#false`
- If the interval is not empty then the list element corresponding to the middle index in the given interval is compared with the number searched for
- If they are equal the middle index is returned as the answer
- If they are not equal then a decision must be made as to which new interval to search
- If the number searched for is less than the number at the middle index then the first half of the interval must be searched
- If the number searched for is greater than the number at the middle index then the second half of the interval must be searched

Searching

The `bin-search` Function

- Searches a given list by traversing a given interval
- If the interval is empty then the answer is `#false`
- If the interval is not empty then the list element corresponding to the middle index in the given interval is compared with the number searched for
- If they are equal the middle index is returned as the answer
- If they are not equal then a decision must be made as to which new interval to search
- If the number searched for is less than the number at the middle index then the first half of the interval must be searched
- If the number searched for is greater than the number at the middle index then the second half of the interval must be searched
- This is generative recursion. Why?

Searching

The bin-search Function

- The interval is empty

```
;; Sample expressions for bin-search
(define BINS-L0-VAL1  #false)
(define BINS-L3S-VAL1 #false)
```

Searching

The bin-search Function

- The interval is empty

```
;; Sample expressions for bin-search
(define BINS-L0-VAL1  #false)
(define BINS-L3S-VAL1 #false)
```

- The middle index list element equals the given number

```
(define BINS-L1S-VAL1
        (local [(define mid-index (quotient (+ 0  8) 2))]
          mid-index))
(define BINS-L2S-VAL1
        (local [(define mid-index (quotient (+ 3 7) 2))]
          mid-index))
```

Searching

The bin-search Function

- The interval is empty

```
;; Sample expressions for bin-search
(define BINS-L0-VAL1  #false)
(define BINS-L3S-VAL1 #false)
```

- The middle index list element equals the given number

```
(define BINS-L1S-VAL1
        (local [(define mid-index (quotient (+ 0  8) 2))]
          mid-index))
(define BINS-L2S-VAL1
        (local [(define mid-index (quotient (+ 3 7) 2))]
          mid-index))
```

- The middle index list element is greater than the given number

```
(define BINS-L1S-VAL2
        (local [(define mid-index (quotient (+ 0 8) 2))]
          (bin-search  0 0 (sub1 mid-index) L1S)))
(define BINS-L2S-VAL2
        (local [(define mid-index (quotient (+ 0 11) 2))]
          (bin-search -6 0 (sub1 mid-index) L2S)))
```

Searching

The bin-search Function

- The interval is empty

```
;; Sample expressions for bin-search
(define BINS-L0-VAL1  #false)
(define BINS-L3S-VAL1 #false)
```

- The middle index list element equals the given number

```
(define BINS-L1S-VAL1
        (local [(define mid-index (quotient (+ 0  8) 2))]
          mid-index))
(define BINS-L2S-VAL1
        (local [(define mid-index (quotient (+ 3 7) 2))]
          mid-index))
```

- The middle index list element is greater than the given number

```
(define BINS-L1S-VAL2
        (local [(define mid-index (quotient (+ 0 8) 2))]
          (bin-search  0 0 (sub1 mid-index) L1S)))
(define BINS-L2S-VAL2
        (local [(define mid-index (quotient (+ 0 11) 2))]
          (bin-search -6 0 (sub1 mid-index) L2S)))
```

- The middle index list element is less than the given number

```
(define BINS-L1S-VAL3
        (local [(define mid-index (quotient (+ 0 8) 2))]
          (bin-search  90 (add1 mid-index)  8 L1S)))
(define BINS-L2S-VAL3
```

Searching

The bin-search Function

- The interval is empty

```
;; Sample expressions for bin-search
(define BINS-L0-VAL1  #false)
(define BINS-L3S-VAL1 #false)
```

- The middle index list element equals the given number

```
(define BINS-L1S-VAL1
        (local [(define mid-index (quotient (+ 0  8) 2))]
          mid-index))
(define BINS-L2S-VAL1
        (local [(define mid-index (quotient (+ 3 7) 2))]
          mid-index))
```

- The middle index list element is greater than the given number

```
(define BINS-L1S-VAL2
        (local [(define mid-index (quotient (+ 0 8) 2))]
          (bin-search  0 0 (sub1 mid-index) L1S)))
(define BINS-L2S-VAL2
        (local [(define mid-index (quotient (+ 0 11) 2))]
          (bin-search -6 0 (sub1 mid-index) L2S)))
```

- The middle index list element is less than the given number

```
(define BINS-L1S-VAL3
        (local [(define mid-index (quotient (+ 0 8) 2))]
          (bin-search  90 (add1 mid-index)  8 L1S)))
(define BINS-L2S-VAL3
```

Searching

The bin-search Function

- ```
  ;; number [int>=0..int>=-1] lon → res
  ;; Purpose: Return an index for the given number if it
  ;;          is a member of the given list. Otherwise,
  ;;          return #false.
  ```
  ```

Searching

The bin-search Function

- ```
  ;; number [int>=0..int>=-1] lon → res
  ;; Purpose: Return an index for the given number if it
  ;;          is a member of the given list. Otherwise,
  ;;          return #false.
  ```
- ```
  ;; How: If the given interval is empty the given number
  ;;  is not in the given list and return #false. Otherwise,
  ;;  compute the middle index and return it if the given
  ;;  list has the given number at that index. If not
  ;;  search either the first or the second half of the
  ;;  given interval.
  ```

Searching

The bin-search Function

- ```
  ;; number [int>=0..int>=-1] lon → res
  ;; Purpose: Return an index for the given number if it
  ;;          is a member of the given list. Otherwise,
  ;;          return #false.
  ```
- ```
  ;; How: If the given interval is empty the given number
  ;;  is not in the given list and return #false. Otherwise,
  ;;  compute the middle index and return it if the given
  ;;  list has the given number at that index. If not
  ;;  search either the first or the second half of the
  ;;  given interval.
  ```
- ```
  ;; Assumption: The given lon is sorted in nondecreasing
  ;;  order and the given interval only contains valid
  ;;  indices into the given lon.
  (define (bin-search a-num low high a-lon)
  ```
  ```

Searching

The bin-search Function

- ```
  ;; Tests using sample computations for bin-search
  (check-expect (bin-search 65 0 -1 L0)  BINS-L0-VAL1)
  (check-expect (bin-search -9 5  4 L3S) BINS-L3S-VAL1)
  (check-expect (bin-search  7 0  8 L1S) BINS-L1S-VAL1)
  (check-expect (bin-search  3 3  7 L2S) BINS-L2S-VAL1)
  (check-expect (bin-search  0 0  8 L1S) BINS-L1S-VAL2)
  (check-expect (bin-search -6 0 11 L2S) BINS-L2S-VAL2)
  (check-expect (bin-search 90 0  8 L1S) BINS-L1S-VAL3)
  (check-expect (bin-search  8 0 11 L2S) BINS-L2S-VAL3)
  ```
  ```

Searching

The bin-search Function

- ```
  ;; Tests using sample computations for bin-search
  (check-expect (bin-search 65 0 -1 L0)  BINS-L0-VAL1)
  (check-expect (bin-search -9 5  4 L3S) BINS-L3S-VAL1)
  (check-expect (bin-search  7 0  8 L1S) BINS-L1S-VAL1)
  (check-expect (bin-search  3 3  7 L2S) BINS-L2S-VAL1)
  (check-expect (bin-search  0 0  8 L1S) BINS-L1S-VAL2)
  (check-expect (bin-search -6 0 11 L2S) BINS-L2S-VAL2)
  (check-expect (bin-search 90 0  8 L1S) BINS-L1S-VAL3)
  (check-expect (bin-search  8 0 11 L2S) BINS-L2S-VAL3)
  ```
- ```
  ;; Tests using sample values for bin-search
  (check-satisfied
    (bin-search 100 0 (sub1 10000) L3S)
    (λ (a-res) (or (false? a-res)
                   (= (list-ref L3S a-res) 100))))
  (check-expect (bin-search 2 0 2 '(1 2 3)) 1)
  (check-expect (bin-search 5 0 5 '(1 2 3 4 6 7)) #false)
  ```

Searching

The bin-search Function

```
(if (< high low)
    #false
    (local [(define mid-index (quotient (+ low high) 2))]
      (cond
        [(= (list-ref a-lon mid-index) a-num) mid-index]
        [(> (list-ref a-lon mid-index) a-num)
         (bin-search a-num low (sub1 mid-index) a-lon)]
        [else
          (bin-search a-num (add1 mid-index) high a-lon)])))
```

Searching

Binary Search Performance

- Whenever `bin-search` is called a new interval of half the size is generated and recursively processed

Searching

Binary Search Performance

- Whenever `bin-search` is called a new interval of half the size is generated and recursively processed
- This means that with every recursive call the interval is getting smaller

Searching

Binary Search Performance

- Whenever `bin-search` is called a new interval of half the size is generated and recursively processed
- This means that with every recursive call the interval is getting smaller
- Eventually either the middle index element is equal to the given number or the interval becomes empty and the function halts.

Searching

Binary Search Performance

- ```
  (define BSL0 (time (binary-search 83333 L0)))
  (define BSL1 (time (binary-search 0 L1S)))
  (define BSL2 (time (binary-search 8 L2S)))
  (define BSL3 (time (binary-search (first L3) L3S)))
  (define BSL4 (time (binary-search 2000000 L4)))
  ```
  ```

Searching

Binary Search Performance

- ```
  (define BSL0 (time (binary-search 83333 L0)))
  (define BSL1 (time (binary-search 0 L1S)))
  (define BSL2 (time (binary-search 8 L2S)))
  (define BSL3 (time (binary-search (first L3) L3S)))
  (define BSL4 (time (binary-search 2000000 L4)))
  ```
-

	L0	L1	L2	L3	L4
linear-search	0	0	0	0	1359
binary-search	0	0	0	31	218
  ```

Searching

Binary Search Performance

- Why is binary search better?

Searching

Binary Search Performance

- Why is binary search better?
- What is the best running time for `binary-search`?

Searching

Binary Search Performance

- Why is binary search better?
- What is the best running time for `binary-search`?
- It is when a recursive call is not made.
- $O(k)$

Searching

Binary Search Performance

- Why is binary search better?
- What is the best running time for `binary-search`?
- It is when a recursive call is not made.
- $O(k)$
- What is the worst-case scenario for `bin-search`?

Searching

Binary Search Performance

- Why is binary search better?
- What is the best running time for `binary-search`?
- It is when a recursive call is not made.
- $O(k)$
- What is the worst-case scenario for `bin-search`?
- When the interval must be split the most
- The maximum number of times the interval can be split in half is proportional to $O(\lg(n))$

Searching

Binary Search Performance

- Why is binary search better?
- What is the best running time for `binary-search`?
- It is when a recursive call is not made.
- $O(k)$
- What is the worst-case scenario for `bin-search`?
- When the interval must be split the most
- The maximum number of times the interval can be split in half is proportional to $O(\lg(n))$
- How many operations are done for every recursive call?

Searching

Binary Search Performance

- Why is binary search better?
- What is the best running time for `binary-search`?
- It is when a recursive call is not made.
- $O(k)$
- What is the worst-case scenario for `bin-search`?
- When the interval must be split the most
- The maximum number of times the interval can be split in half is proportional to $O(\lg(n))$
- How many operations are done for every recursive call?
- In the worst case part of the list must be traversed twice (using `list-ref`) making the number of operations proportional to $2n = O(n)$.

Searching

Binary Search Performance

- Why is binary search better?
- What is the best running time for `binary-search`?
- It is when a recursive call is not made.
- $O(k)$
- What is the worst-case scenario for `bin-search`?
- When the interval must be split the most
- The maximum number of times the interval can be split in half is proportional to $O(\lg(n))$
- How many operations are done for every recursive call?
- In the worst case part of the list must be traversed twice (using `list-ref`) making the number of operations proportional to $2n = O(n)$.
- This makes `bin-search`'s complexity $O(n * \lg(n))$

Searching

HOMEWORK

- Problems: 2, 4

Trees

- Searching data linearly organized is straightforward

Trees

- Searching data linearly organized is straightforward
- Not all data is linear
- Searching a binary search tree: search one of the subtrees

Trees

- Searching data linearly organized is straightforward
- Not all data is linear
- Searching a binary search tree: search one of the subtrees
- The set of binary trees is a subtype of `tree`
- A tree is a nonlinear data structure in which every node has an arbitrary number of subtrees (or children)

Trees

- Searching data linearly organized is straightforward
- Not all data is linear
- Searching a binary search tree: search one of the subtrees
- The set of binary trees is a subtype of `tree`
- A tree is a nonlinear data structure in which every node has an arbitrary number of subtrees (or children)
- The top node is called the root of the tree and does not have a parent
- All other nodes have a single parent

Trees

- Searching data linearly organized is straightforward
- Not all data is linear
- Searching a binary search tree: search one of the subtrees
- The set of binary trees is a subtype of `tree`
- A tree is a nonlinear data structure in which every node has an arbitrary number of subtrees (or children)
- The top node is called the root of the tree and does not have a parent
- All other nodes have a single parent
- A tree is defined as follows:

```
;; A (treeof X) is either:
;;   1. '()
;;   2. (make-node X (listof node))

(define-struct node (val subtrees))
```

Trees

```
#| TEMPLATE FOR FUNCTIONS ON A (treeof X)
;;  Sample (treeof X)
(define TOX0 '())
(define TOX1 (make-node ... ...))
     ⋮
;; (treeof X) ... → ...    Purpose:
(define (f-on-tox a-tox ...)
  (if (empty? a-tox)
      ...
      (f-on-node a-tox ...)))

;; Sample expressions for f-on-tox
(define TOX0-VAL ...)
(define TOX1-VAL ...)
     ⋮
;; Tests using sample computations for f-on-tox
(check-expect (f-on-tox TOX0 ...) TOX0-VAL)
(check-expect (f-on-tox TOX1 ...) TOX1-VAL)
     ⋮
;; Tests using sample values for f-on-tox
(check-expect (f-on-tox ... ...) ...)
     ⋮
```

Trees

```
#| TEMPLATE FOR FUNCTIONS ON A node
;; Sample nodes
(define NODE0 (make-node ... ...))
      ⋮

;; node ... → ...
;; Purpose:
(define (f-on-node a-node ...)
  (...(f-on-X (node-val a-node ...)
   ...(f-on-lonode (node-subtrees a-node) ...)))

;; Sample expressions for f-on-node
(define NODE0-VAL ...)
      ⋮
;; Tests using sample computations for f-on-node
(check-expect (f-on-node NODE0 ...) NODE0-VAL)
      ⋮
;; Tests using sample values for f-on-node
(check-expect (f-on-node ... ...) ...)
      ⋮
                                                        |#
```

Trees

```
#| TEMPLATE FOR FUNCTIONS ON A (listof node)
;; Sample (listof node)
(define LONODE0 '())
(define LONODE1 ...)
      ⋮
;; a-lonode ... → ...
;; Purpose:
(define (f-on-lonode a-lonode ...)
  (if (empty? a-lonode)
      ...
      ...(f-on-node (first a-lox) ...)...(f-on-lonode (rest a-lox)) ...))
;; Sample expressions for f-on-lonode
(define LONODE0-VAL ...)
(define LONODE1-VAL ...)
      ⋮
;; Tests using sample computations for f-on-lonode
(check-expect (f-on-lox LONODE0 ...) LONODE0-VAL)
(check-expect (f-on-lox LONODE1 ...) LONODE1-VAL)
      ⋮
;; Tests using sample values for f-on-lonode
(check-expect (f-on-lonode ... ...) ...)
      ⋮
```

Trees

- How is a tree searched?

Trees

- How is a tree searched?
- To explore this problem we define the following sample nodes and trees (of numbers):

```
(define NODE10  (make-node 10  '()))
(define NODE3   (make-node  3  '()))
(define NODE87  (make-node 87  '()))
(define NODE-5  (make-node  -5 '()))
(define NODE0   (make-node   0 '()))
(define NODE66  (make-node  66 '()))
(define NODE44  (make-node  44 '()))
(define NODE47  (make-node  47 '()))
(define NODE850 (make-node 850 (list NODE10 NODE3)))
(define NODE235 (make-node 235 (list NODE87 NODE-5 NODE0)))
(define NODE23  (make-node  23 (list NODE44 NODE47)))
(define NODE-88 (make-node -88 (list NODE23)))
(define NODE600 (make-node
                  600 (list NODE850 NODE235 NODE66 NODE-88)))
(define T0 '())
(define T1 NODE10)
(define T2 NODE600)
```

Trees

- How is a tree searched?
- To explore this problem we define the following sample nodes and trees (of numbers):

```
(define NODE10  (make-node 10  '()))
(define NODE3   (make-node  3  '()))
(define NODE87  (make-node 87  '()))
(define NODE-5  (make-node  -5 '()))
(define NODE0   (make-node   0 '()))
(define NODE66  (make-node  66 '()))
(define NODE44  (make-node  44 '()))
(define NODE47  (make-node  47 '()))
(define NODE850 (make-node 850 (list NODE10 NODE3)))
(define NODE235 (make-node 235 (list NODE87 NODE-5 NODE0)))
(define NODE23  (make-node  23 (list NODE44 NODE47)))
(define NODE-88 (make-node -88 (list NODE23)))
(define NODE600 (make-node
                  600 (list NODE850 NODE235 NODE66 NODE-88)))
(define T0 '())
(define T1 NODE10)
(define T2 NODE600)
```

- Typing deep trees is a long, tedious, error-prone, and bias-prone process it is best to write a function to create nonempty trees

Trees

- To protect ourselves against any bias write a function to create a tree of random natural numbers with a maximum given depth

Trees

- To protect ourselves against any bias write a function to create a tree of random natural numbers with a maximum given depth
- ```
  (define RANDOM-NUM-RANGE 1000000)
  (define MAX-NUM-SUBTREES 10)
  ```
  ```

Trees

- To protect ourselves against any bias write a function to create a tree of random natural numbers with a maximum given depth
- ```
  (define RANDOM-NUM-RANGE 1000000)
  (define MAX-NUM-SUBTREES 10)
  ```
- Sample expressions for a random tree of depth 0 are:

```
;; Sample expressions for make-tonatnum
(define TON0
        (local [(define root-val (random RANDOM-NUM-RANGE))]
          (make-node root-val '())))
(define TON0-2
        (local [(define root-val (random RANDOM-NUM-RANGE))]
          (make-node root-val '())))
```
```

Trees

- To protect ourselves against any bias write a function to create a tree of random natural numbers with a maximum given depth
- ```
  (define RANDOM-NUM-RANGE 1000000)
  (define MAX-NUM-SUBTREES 10)
  ```
- Sample expressions for a random tree of depth 0 are:

```
;; Sample expressions for make-tonatnum
(define TON0
        (local [(define root-val (random RANDOM-NUM-RANGE))]
          (make-node root-val '())))
(define TON0-2
        (local [(define root-val (random RANDOM-NUM-RANGE))]
          (make-node root-val '())))
```

- Sample expressions for a tree of depth greater than 0 are:

```
(define TON1 (local [(define root-val (random RANDOM-NUM-RANGE))]
  (make-node root-val
             (build-list (random MAX-NUM-SUBTREES)
                         (λ (i)
                           (make-tonatnum (sub1 1)))))))
(define TON2 (local [(define root-val (random RANDOM-NUM-RANGE))]
               (make-node
                 root-val
                 (build-list (random MAX-NUM-SUBTREES)
                             (λ (i)
                               (make-tonatnum (sub1 2)))))))
```
```

Trees

- ```
  ;; natnum → (treeof number) Purpose:Create tree of given max depth
  (define (make-tonatnum d)
  ```
  ```

Trees

- ```
  ;; natnum → (treeof number) Purpose:Create tree of given max depth
  (define (make-tonatnum d)
  ```

- test for tree of depth 0

  ```
  ;; Tests using sample computations for make-tonatnum
  (check-satisfied TONO-1 (λ (t) (and (integer? (node-val t))
                                      (>= (node-val t) 0)
                                      (empty? (node-subtrees t))))) .
  ```
  ```

Trees

- ```
  ;; natnum → (treeof number) Purpose:Create tree of given max depth
  (define (make-tonatnum d)
  ```

- test for tree of depth 0
  ```
  ;; Tests using sample computations for make-tonatnum
  (check-satisfied TON0-1 (λ (t) (and (integer? (node-val t))
                                      (>= (node-val t) 0)
                                      (empty? (node-subtrees t))))) .
  ```

- test for tree of depth 1
  ```
  (check-satisfied TON1
   (λ (t) (and  (integer? (node-val t))
                (>= (node-val t) 0)
                (< (length (node-subtrees t)) MAX-NUM-SUBTREES)
                (and (andmap (λ (n)
                               (and (integer? (node-val n))
                                    (>= (node-val t) 0)))
                             (node-subtrees t))
                     (andmap
                      (λ (n) (empty? (node-subtrees n)))
                      (node-subtrees t))))))
  ```
  ```

Trees

- ```
  ;; natnum → (treeof number) Purpose:Create tree of given max depth
  (define (make-tonatnum d)
  ```
- ```
    (local [(define root-val (random RANDOM-NUM-RANGE))]
      (cond  [(= d 0) (make-node root-val '())]
             [else (make-node root-val
                              (build-list
                                (random MAX-NUM-SUBTREES)
                                (λ (i) (make-tonatnum (sub1 d)))))]))
  ```
- test for tree of depth 0
  ```
  ;; Tests using sample computations for make-tonatnum
  (check-satisfied TON0-1 (λ (t) (and (integer? (node-val t))
                                      (>= (node-val t) 0)
                                      (empty? (node-subtrees t))))) .
  ```
- test for tree of depth 1
  ```
  (check-satisfied TON1
   (λ (t) (and  (integer? (node-val t))
                (>= (node-val t) 0)
                (< (length (node-subtrees t)) MAX-NUM-SUBTREES)
                (and (andmap (λ (n)
                               (and (integer? (node-val n))
                                    (>= (node-val t) 0)))
                             (node-subtrees t))
                     (andmap
                      (λ (n) (empty? (node-subtrees n)))
                      (node-subtrees t))))))
  ```

Trees

- Run the tests and make sure they all pass

Trees

- Run the tests and make sure they all pass
- We may now define a sample tree of arbitrary depth, say 7, as follows:

```
(define T3 (make-tonatnum 7))
```

- This tree is also used to test the tree searching programs developed

Trees

HOMEWORK

Problem 5

Trees

Depth-First Search

- Consider the problem of determining if a given number is a member of a given `(treeof number)`

Trees

Depth-First Search

- Consider the problem of determining if a given number is a member of a given `(treeof number)`
- The root value may be the number that is searched for
- The answer is `#true` and stop the search process

Trees

Depth-First Search

- Consider the problem of determining if a given number is a member of a given `(treeof number)`
- The root value may be the number that is searched for
- The answer is `#true` and stop the search process
- What if the root value is not the number searched for?

Trees

Depth-First Search

- Consider the problem of determining if a given number is a member of a given `(treeof number)`
- The root value may be the number that is searched for
- The answer is `#true` and stop the search process
- What if the root value is not the number searched for?
- The number may be in the first subtree and the answer is `#true`

Trees

Depth-First Search

- Consider the problem of determining if a given number is a member of a given `(treeof number)`
- The root value may be the number that is searched for
- The answer is `#true` and stop the search process
- What if the root value is not the number searched for?
- The number may be in the first subtree and the answer is `#true`
- What if it is not in the first subtree?

Trees

Depth-First Search

- Consider the problem of determining if a given number is a member of a given `(treeof number)`
- The root value may be the number that is searched for
- The answer is `#true` and stop the search process
- What if the root value is not the number searched for?
- The number may be in the first subtree and the answer is `#true`
- What if it is not in the first subtree?
- After a failed search of the first subtree it is necessary to *backtrack* (up the tree) to search the rest of the siblings
- The process is repeated for each subtree until one contains the number searched for or there are no more siblings to search

Trees

Depth-First Search

- Consider the problem of determining if a given number is a member of a given `(treeof number)`
- The root value may be the number that is searched for
- The answer is `#true` and stop the search process
- What if the root value is not the number searched for?
- The number may be in the first subtree and the answer is `#true`
- What if it is not in the first subtree?
- After a failed search of the first subtree it is necessary to *backtrack* (up the tree) to search the rest of the siblings
- The process is repeated for each subtree until one contains the number searched for or there are no more siblings to search
- This is known as *depth-first search*
- In depth-first search an avenue (like a subtree) is searched before any other search avenues (like other subtrees).

Trees

Depth-First Search

- A number is contained in a tree if the tree is not empty and the node contains the number

Trees

Depth-First Search

- A number is contained in a tree if the tree is not empty and the node contains the number
- ```
  ;; Sample expressions for ton-dfs-contains?
  (define T0-DFS-VAL (and (not (empty? T0))    Book typo
                          (node-dfs-contains? 77 T0)))
  (define T1-DFS-VAL (and (not (empty? T1))
                          (node-dfs-contains? 33 T1)))
  (define T2-DFS-VAL (and (not (empty? T2))
                          (node-dfs-contains? 23 T2)))
  (define T3-DFS-VAL (and (not (empty? T3))
                          (node-dfs-contains? 45 T3)))
  ```
  ```

Trees

Depth-First Search

- A number is contained in a tree if the tree is not empty and the node contains the number
- ```
  ;; Sample expressions for ton-dfs-contains?
  (define T0-DFS-VAL (and (not (empty? T0))   Book typo
                          (node-dfs-contains? 77 T0)))
  (define T1-DFS-VAL (and (not (empty? T1))
                          (node-dfs-contains? 33 T1)))
  (define T2-DFS-VAL (and (not (empty? T2))
                          (node-dfs-contains? 23 T2)))
  (define T3-DFS-VAL (and (not (empty? T3))
                          (node-dfs-contains? 45 T3)))
  ```
- Two differences: number searched for and tree searched
  ```

Trees

Depth-First Search

- ```
  ;; number (treeof number) → Boolean
  ;; Purpose: Determine if the given number is in the given tree
  (define (ton-dfs-contains? a-num a-ton)
  ```
  ```

Trees

Depth-First Search

- ```
  ;; number (treeof number) → Boolean
  ;; Purpose: Determine if the given number is in the given tree
  (define (ton-dfs-contains? a-num a-ton)
  ```
- ```
       ;; Tests using sample computations for ton-dfs-contains?
       (check-expect (ton-dfs-contains? 77 T0) T0-DFS-VAL)
       (check-expect (ton-dfs-contains? 33 T1) T1-DFS-VAL)
       (check-expect (ton-dfs-contains? 23 T2) T2-DFS-VAL)
       (check-expect (ton-dfs-contains? 45 T3) T3-DFS-VAL)
  ```

Trees

Depth-First Search

- ```
  ;; number (treeof number) → Boolean
  ;; Purpose: Determine if the given number is in the given tree
  (define (ton-dfs-contains? a-num a-ton)
  ```
- ```
  ;; Tests using sample computations for ton-dfs-contains?
  (check-expect (ton-dfs-contains? 77 T0) T0-DFS-VAL)
  (check-expect (ton-dfs-contains? 33 T1) T1-DFS-VAL)
  (check-expect (ton-dfs-contains? 23 T2) T2-DFS-VAL)
  (check-expect (ton-dfs-contains? 45 T3) T3-DFS-VAL)
  ```
- ```
  ;; Tests using sample values for ton-dfs-contains?
  (check-satisfied (make-node
                     307759
                     (list (make-node 816392 '())
                           (make-node 153333 '())
                           (make-node 684270 '())))
                   (λ (t) (ton-dfs-contains? 153333 t)))
  (check-satisfied (make-node
                     307759
                     (list (make-node 816392 '())
                           (make-node 153333 '())
                           (make-node 684270 '())))
                   (λ (t) (not (ton-dfs-contains? 6561 t))))
  ```
  ```

Trees

Depth-First Search

- ```
  ;; number (treeof number) → Boolean
  ;; Purpose: Determine if the given number is in the given tree
  (define (ton-dfs-contains? a-num a-ton)
  ```
- ```
        (and (not (empty? a-ton))
             (node-dfs-contains? a-num a-ton))
  ```
- ```
        ;; Tests using sample computations for ton-dfs-contains?
        (check-expect (ton-dfs-contains? 77 T0) T0-DFS-VAL)
        (check-expect (ton-dfs-contains? 33 T1) T1-DFS-VAL)
        (check-expect (ton-dfs-contains? 23 T2) T2-DFS-VAL)
        (check-expect (ton-dfs-contains? 45 T3) T3-DFS-VAL)
  ```
- ```
        ;; Tests using sample values for ton-dfs-contains?
        (check-satisfied (make-node
                           307759
                           (list (make-node 816392 '())
                                 (make-node 153333 '())
                                 (make-node 684270 '())))
                          (λ (t) (ton-dfs-contains? 153333 t)))
        (check-satisfied (make-node
                           307759
                           (list (make-node 816392 '())
                                 (make-node 153333 '())
                                 (make-node 684270 '())))
                          (λ (t) (not (ton-dfs-contains? 6561 t))))
  ```

Trees

The `node-dfs-contains?` Function

- This function must implement a depth-first search of a nonempty tree (i.e., a `node`)

Trees

The `node-dfs-contains?` Function

- This function must implement a depth-first search of a nonempty tree (i.e., a `node`)
- A node contains the given number if it is the root value or if any subtree contains the given value
- The former may be determined by comparing the given number and the root value of the given tree

Trees

The `node-dfs-contains?` Function

- This function must implement a depth-first search of a nonempty tree (i.e., a `node`)
- A node contains the given number if it is the root value or if any subtree contains the given value
- The former may be determined by comparing the given number and the root value of the given tree
- The latter must be determined by calling a function to process a list
  1. Mutually recursive with `node-dfs-contains?`
  2. Stop when a subtree that contains the given number is found
  3. Suggests `or`ing the results for each subtree as they are processed

Trees

The node-dfs-contains? Function

- ```
  ;; Sample expressions for node-dfs-contains?
  (define NODE10-VAL1 (or (= 33 (node-val NODE10))
                          (ormap (λ (t)
                                   (node-dfs-contains? 33 t))
                                 (node-subtrees NODE10))))
  (define NODE10-VAL2 (or (= 10 (node-val NODE10))
                          (ormap (λ (t)
                                   (node-dfs-contains? 10 t))
                                 (node-subtrees NODE10))))
  (define NODE600-VAL (or (= -5 (node-val NODE600))
                          (ormap (λ (t)
                                   (node-dfs-contains? -5 t))
                                 (node-subtrees NODE600))))
  ```
  ```

Trees

The node-dfs-contains? Function

- ```
  ;; Sample expressions for node-dfs-contains?
  (define NODE10-VAL1 (or (= 33 (node-val NODE10))
                          (ormap (λ (t)
                                   (node-dfs-contains? 33 t))
                                 (node-subtrees NODE10))))
  (define NODE10-VAL2 (or (= 10 (node-val NODE10))
                          (ormap (λ (t)
                                   (node-dfs-contains? 10 t))
                                 (node-subtrees NODE10))))
  (define NODE600-VAL (or (= -5 (node-val NODE600))
                          (ormap (λ (t)
                                   (node-dfs-contains? -5 t))
                                 (node-subtrees NODE600))))
  ```
- There are two differences: the number that is searched for and the node that is searched
  ```

Trees

The node-dfs-contains? Function

- ```
  ;; number node → Boolean
  ;; Purpose: Determine if given node contains given number
  (define (node-dfs-contains? a-num a-node)
  ```
  ```

Trees

The node-dfs-contains? Function

- ```
  ;; number node → Boolean
  ;; Purpose: Determine if given node contains given number
  (define (node-dfs-contains? a-num a-node)
  ```

- ```
  ;; Tests using sample computations for node-dfs-contains?
  (check-expect (node-dfs-contains? 33 NODE10)  NODE10-VAL1)
  (check-expect (node-dfs-contains? 10 NODE10)  NODE10-VAL2)
  (check-expect (node-dfs-contains? -5 NODE600) NODE600-VAL)
  ```

Trees

The node-dfs-contains? Function

- ```
  ;; number node → Boolean
  ;; Purpose: Determine if given node contains given number
  (define (node-dfs-contains? a-num a-node)
  ```

- ```
  ;; Tests using sample computations for node-dfs-contains?
  (check-expect (node-dfs-contains? 33 NODE10)  NODE10-VAL1)
  (check-expect (node-dfs-contains? 10 NODE10)  NODE10-VAL2)
  (check-expect (node-dfs-contains? -5 NODE600) NODE600-VAL)
  ```

- ```
  ;; Tests using sample values for node-dfs-contains?
  (check-satisfied (make-node 31
                              (list (make-node 45 '())
                                    (make-node 31 '())
                                    (make-node  7 '())))
                   (λ (t) (node-dfs-contains? 31 t)))
  (check-satisfied (make-node 67
                              (list (make-node 45 '())
                                    (make-node 31 '())
                                    (make-node  7 '())))
                   (λ (t) (not (node-dfs-contains? 87 t))))
  ```
  ```

Trees

The node-dfs-contains? Function

- ```
  ;; number node → Boolean
  ;; Purpose: Determine if given node contains given number
  (define (node-dfs-contains? a-num a-node)
  ```
- ```
    (or (= a-num (node-val a-node))
        (ormap (λ (t) (node-dfs-contains? a-num t))
               (node-subtrees a-node))))
  ```
- ```
  ;; Tests using sample computations for node-dfs-contains?
  (check-expect (node-dfs-contains? 33 NODE10)  NODE10-VAL1)
  (check-expect (node-dfs-contains? 10 NODE10)  NODE10-VAL2)
  (check-expect (node-dfs-contains? -5 NODE600) NODE600-VAL)
  ```
- ```
  ;; Tests using sample values for node-dfs-contains?
  (check-satisfied (make-node 31
                              (list (make-node 45 '())
                                    (make-node 31 '())
                                    (make-node  7 '())))
                   (λ (t) (node-dfs-contains? 31 t)))
  (check-satisfied (make-node 67
                              (list (make-node 45 '())
                                    (make-node 31 '())
                                    (make-node  7 '())))
                   (λ (t) (not (node-dfs-contains? 87 t))))
  ```

Trees

Performance

- Time three searches using `T3`
  1. Number in the first subtree 5 levels down
  2. Root value of the last subtree
  3. Number that is not in the tree

Trees

Performance

- Time three searches using T3
  1. Number in the first subtree 5 levels down
  2. Root value of the last subtree
  3. Number that is not in the tree
- get a root value 5 levels down

```
(time (ton-dfs-contains?
       (node-val (first
                  (node-subtrees
                   (first
                    (node-subtrees
                     (first
                      (node-subtrees
                       (first
                        (node-subtrees
                         (first
                          (node-subtrees (first (node-subtrees T3))
       T3))
(time
  (ton-dfs-contains?
    (list-ref (map (λ (t) (node-val t)) (node-subtrees T3))
              (sub1 (length (node-subtrees T3))))
    T3))
(time (ton-dfs-contains? -8 T3))
```

Trees

Performance

- Time three searches using T3
  1. Number in the first subtree 5 levels down
  2. Root value of the last subtree
  3. Number that is not in the tree
- get a root value 5 levels down

```
(time (ton-dfs-contains?
        (node-val (first
                    (node-subtrees
                     (first
                      (node-subtrees
                       (first
                        (node-subtrees
                         (first
                          (node-subtrees
                           (first
                            (node-subtrees (first (node-subtrees T3))
        T3))
(time
  (ton-dfs-contains?
    (list-ref (map (λ (t) (node-val t)) (node-subtrees T3))
              (sub1 (length (node-subtrees T3))))
    T3))
(time (ton-dfs-contains? -8 T3))
```

Trees

Performance

-

	$\texttt{Experiment}_1$	$\texttt{Experiment}_2$	$\texttt{Experiment}_3$
DFS	0	156	671

Trees

Performance

•

	$\texttt{Experiment}_1$	$\texttt{Experiment}_2$	$\texttt{Experiment}_3$
DFS	0	156	671

- DFS is very fast when the number searched for is in the first subtree

Trees

Performance

•

	$\texttt{Experiment}_1$	$\texttt{Experiment}_2$	$\texttt{Experiment}_3$
DFS	0	156	671

- DFS is very fast when the number searched for is in the first subtree
- Slower when several subtrees must searched like in the second experiment which is unfortunate given that the number searched for is at a depth of only 1 in the tree.

Trees

Performance

•

	`Experiment`$_1$	`Experiment`$_2$	`Experiment`$_3$
DFS	0	156	671

- DFS is very fast when the number searched for is in the first subtree
- Slower when several subtrees must searched like in the second experiment which is unfortunate given that the number searched for is at a depth of only 1 in the tree.
- The worst performance/scenario is seen/when the entire tree is searched like for the third experiment

Trees

Complexity

- Let `V` be the set of nodes in the tree
- Let `E` be the set of edges in the tree

Trees

Complexity

- Let `V` be the set of nodes in the tree
- Let `E` be the set of edges in the tree
- In the worst case (when the given number is not in the given tree) all the root values must be compared with the given number and all the edges must be traversed to reach every node
- This makes the work done by a depth-first search of a tree proportional to, `n = |V| + |E|` $= O(n)$
- It is a linear-time algorithm

Trees

HOMEWORK

- Problems 6 and 7

Trees

Breadth-First Search

- Performance is disappointing when the number searched for is at a shallow level in the tree
- Why search multiple subtrees when the number may be found only a few levels away from the root?
- Suggests that a tree be traversed level by level instead of by subtrees

Trees

Breadth-First Search

- The first number to check during the search is the root value because it has the lowest depth

Trees

Breadth-First Search

- The first number to check during the search is the root value because it has the lowest depth
- If the search must continue then the first values that need to be checked are the root values of its children
- If necessary the process continues with grandchildren and so on

Trees

Breadth-First Search

- The first number to check during the search is the root value because it has the lowest depth
- If the search must continue then the first values that need to be checked are the root values of its children
- If necessary the process continues with grandchildren and so on
- This may be achieved by keeping the trees that need to be traversed in a `first-in first-out` (`FIFO`) order

Trees

Breadth-First Search

- The first number to check during the search is the root value because it has the lowest depth
- If the search must continue then the first values that need to be checked are the root values of its children
- If necessary the process continues with grandchildren and so on
- This may be achieved by keeping the trees that need to be traversed in a `first-in first-out` (`FIFO`) order
- To start the given tree is stored in a `FIFO` manner
- If the root value is not equal to the given number then all the children are added to the set of trees that may still need to be searched in a `FIFO` manner
- This process repeats itself until the given number is found or the set of trees to search is empty

Trees

Breadth-First Search

- The first number to check during the search is the root value because it has the lowest depth
- If the search must continue then the first values that need to be checked are the root values of its children
- If necessary the process continues with grandchildren and so on
- This may be achieved by keeping the trees that need to be traversed in a `first-in first-out` (`FIFO`) order
- To start the given tree is stored in a `FIFO` manner
- If the root value is not equal to the given number then all the children are added to the set of trees that may still need to be searched in a `FIFO` manner
- This process repeats itself until the given number is found or the set of trees to search is empty
- How are trees kept in `FIFO` order?

Trees

Breadth-First Search

- The first number to check during the search is the root value because it has the lowest depth
- If the search must continue then the first values that need to be checked are the root values of its children
- If necessary the process continues with grandchildren and so on
- This may be achieved by keeping the trees that need to be traversed in a `first-in first-out` (`FIFO`) order
- To start the given tree is stored in a `FIFO` manner
- If the root value is not equal to the given number then all the children are added to the set of trees that may still need to be searched in a `FIFO` manner
- This process repeats itself until the given number is found or the set of trees to search is empty
- How are trees kept in `FIFO` order?
- Data of arbitrary size
- A data structure that keeps its elements in `FIFO` order is called a queue.

Trees

Queues

- How can we implement a queue?

Trees

Queues

- How can we implement a queue?
- `A (queueof X) is a (listof X)`
- Does this seem silly?

Trees

Queues

- How can we implement a queue?
- `A (queueof X) is a (listof X)`
- Does this seem silly?
- No, the interfaces for a list and a queue are not the same

Trees

Queues

- How can we implement a queue?
- `A (queueof X) is a (listof X)`
- Does this seem silly?
- No, the interfaces for a list and a queue are not the same
- Queue:

qempty?	This function tests if the given queue is empty.
qfirst	This function returns the first element of the queue.
enqueue	This function adds a set of elements to the end of the queue.
dequeue	This function removes the first element from the queue.

Trees

Queues

- ```
  (define E-QUEUE '())

  (define qempty? empty?)

  ;; Tests for qempty?
  (check-expect (qempty? '())      #true)
  (check-expect (qempty? '(a b c)) #false)
  ```
  ```

Trees

Queues

- ```
  (define E-QUEUE '())

  (define qempty? empty?)

  ;; Tests for qempty?
  (check-expect (qempty? '())      #true)
  (check-expect (qempty? '(a b c)) #false)
  ```
- ```
  ;; (qof X) → X throws error
  ;; Purpose: Return first X of the given queue
  (define (qfirst a-qox)
    (if (qempty? a-qox)
        (error "qfirst applied to an empty queue")
        (first a-qox)))

  ;; Tests for qfirst
  (check-error  (qfirst '())
                "qfirst applied to an empty queue")
  (check-expect (qfirst '(a b c)) 'a)
  ```

Trees

Queues

- ```
  ;; (listof X) (qof X) → (qof X)
  ;; Purpose: Add the given list of X to the given
  ;;          queue of X
  (define (enqueue a-lox a-qox) (append a-qox a-lox))

  ;; Tests for enqueue
  (check-expect (enqueue '(8 d) '()) '(8 d))
  (check-expect (enqueue '(d) '(a b c)) '(a b c d))
  (check-expect (enqueue '(6 5 4) '(7)) '(7 6 5 4))
  ```
  ```

Trees

Queues

- ```
  ;; (listof X) (qof X) → (qof X)
  ;; Purpose: Add the given list of X to the given
  ;;          queue of X
  (define (enqueue a-lox a-qox) (append a-qox a-lox))

  ;; Tests for enqueue
  (check-expect (enqueue '(8 d) '()) '(8 d))
  (check-expect (enqueue '(d) '(a b c)) '(a b c d))
  (check-expect (enqueue '(6 5 4) '(7)) '(7 6 5 4))
  ```
- ```
  ;; (qof X) → (qof X) throws error
  ;; Purpose: Return the rest of the given queue
  (define (dequeue a-qox)
    (if (qempty? a-qox)
        (error "dequeue applied to an empty queue")
        (rest a-qox)))

  ;; Tests for qfirst
  (check-error  (dequeue '())
                "dequeue applied to an empty queue")
  (check-expect (dequeue '(a b c)) '(b c))
  ```

Trees

Breadth-First Search

- ```
  ;;; Sample expressions for ton-bfs-contains?
  (define T0-BFS-VAL #false)
  ```
  ```

Trees

Breadth-First Search

- ```
  ;;; Sample expressions for ton-bfs-contains?
  (define T0-BFS-VAL #false)
  ```
- ```
  (define T1-BFS-VAL (bfs-helper
                       33
                       (enqueue (list T1) E-QUEUE)))
  (define T2-BFS-VAL (bfs-helper
                       23
                       (enqueue (list T2) E-QUEUE)))
  (define T3-BFS-VAL (bfs-helper
                       45
                       (enqueue (list T3) E-QUEUE)))
  ```

Trees

Breadth-First Search

- ```
  ;;; Sample expressions for ton-bfs-contains?
  (define T0-BFS-VAL #false)
  ```
- ```
  (define T1-BFS-VAL (bfs-helper
                       33
                       (enqueue (list T1) E-QUEUE)))
  (define T2-BFS-VAL (bfs-helper
                       23
                       (enqueue (list T2) E-QUEUE)))
  (define T3-BFS-VAL (bfs-helper
                       45
                       (enqueue (list T3) E-QUEUE)))
  ```
- Two differences among the sample expressions: a number and a tree

Trees

Breadth-First Search

- ```
  ;; number (treeof number) → Boolean
  ;; Purpose: Determine if the given number is in the
  ;;          given tree
  (define (ton-bfs-contains? a-num a-ton)
  ```
  ```

Trees

Breadth-First Search

- ```
  ;; number (treeof number) → Boolean
  ;; Purpose: Determine if the given number is in the
  ;;          given tree
  (define (ton-bfs-contains? a-num a-ton)
  ```

- ```
  ;;; Tests using sample values for ton-dfs-contains?
  (check-satisfied (make-node 307759
                              (list (make-node 816392 '())
                                    (make-node 153333 '())
                                    (make-node 684270 '())))
                   (λ (t)
                     (ton-bfs-contains? 153333 t)))
  (check-satisfied (make-node 307759
                              (list (make-node 816392 '())
                                    (make-node 153333 '())
                                    (make-node 684270 '())))
                   (λ
                     (t) (not (ton-bfs-contains? 6561 t))))
  ```

Trees

Breadth-First Search

- ```
  ;; number (treeof number) → Boolean
  ;; Purpose: Determine if the given number is in the
  ;;          given tree
  (define (ton-bfs-contains? a-num a-ton)
  ```
- ```
    (if (empty? a-ton)
        #false
        (bfs-helper a-num (enqueue (list a-ton) E-QUEUE))))
  ```
- ```
  ;;; Tests using sample values for ton-dfs-contains?
  (check-satisfied (make-node 307759
                              (list (make-node 816392 '())
                                    (make-node 153333 '())
                                    (make-node 684270 '())))
                   (λ (t)
                     (ton-bfs-contains? 153333 t)))
  (check-satisfied (make-node 307759
                              (list (make-node 816392 '())
                                    (make-node 153333 '())
                                    (make-node 684270 '())))
                   (λ
                     (t) (not (ton-bfs-contains? 6561 t))))
  ```
  ```

Trees

`bfs-helper`

- If the given queue is empty the answer is `#false`.

Trees

`bfs-helper`

- If the given queue is empty the answer is `#false`.
- If the given queue is not empty then the number searched for:
  - may be the root value of the first tree in the queue
  - may be found in any of the rest of the trees in the queue
  - may be found in any of the subtrees of the first tree in the queue

Trees

bfs-helper

- If the given queue is empty the answer is #false.
- If the given queue is not empty then the number searched for:
  - may be the root value of the first tree in the queue
  - may be found in any of the rest of the trees in the queue
  - may be found in any of the subtrees of the first tree in the queue
- The trees that may still be searched are the rest of the trees in the queue and the subtrees of the first tree in the queue

Trees

bfs-helper

- If the given queue is empty the answer is #false.
- If the given queue is not empty then the number searched for:
  - may be the root value of the first tree in the queue
  - may be found in any of the rest of the trees in the queue
  - may be found in any of the subtrees of the first tree in the queue
- The trees that may still be searched are the rest of the trees in the queue and the subtrees of the first tree in the queue
- Built a new queue by dequeuing the first element and enqueuing the children of the first tree
- This is generative recursion

Trees

bfs-helper

- If the given queue is empty the answer is #false.
- If the given queue is not empty then the number searched for:
  - may be the root value of the first tree in the queue
  - may be found in any of the rest of the trees in the queue
  - may be found in any of the subtrees of the first tree in the queue
- The trees that may still be searched are the rest of the trees in the queue and the subtrees of the first tree in the queue
- Built a new queue by dequeuing the first element and enqueuing the children of the first tree
- This is generative recursion
- Testing queues:

```
(define QTON0 '())
(define QTON1 (list T1))
(define QTON2 (list T2 T1))
```

Trees

bfs-helper

```
;; Sample expressions for bfs-helper
(define QTON0-VAL
  (and (not (qempty? QTON0))
       (or (= 89 (node-val (qfirst QTON0)))
           (local
               [(define newq (enqueue
                               (node-subtrees (qfirst QTON0))
                               (dequeue QTON0)))]
             (bfs-helper 89 newq)))))
(define QTON1-VAL
  (and (not (qempty? QTON1))
       (or (= 99 (node-val (qfirst QTON1)))
           (local
               [(define newq (enqueue
                               (node-subtrees (qfirst QTON1))
                               (dequeue QTON1)))]
             (bfs-helper 99 newq)))))
(define QTON2-VAL
  (and (not (qempty? QTON2))
       (or (= 47 (node-val (qfirst QTON2)))
           (local
               [(define newq (enqueue (node-subtrees (qfirst QTON2)
                                      (dequeue QTON2)))]
             (bfs-helper 47 newq)))))
```

Trees

bfs-helper

- ```
  ;; number (qof (treeof number)) → Boolean
  ;; Purpose: Search the trees in the given queue for the
  ;;          given number.
  ;; How: If the queue is empty or if the root value of
  ;;  the first tree in the queue equals the given number
  ;;  then stop. Otherwise, search for the number in a
  ;;  queue that contains all but the first tree in the
  ;;  given queue and the subtrees of the first tree in
  ;;  the given queue.
  (define (bfs-helper a-num a-qton)
  ```
  ```

Trees

bfs-helper

- ```
  ;; number (qof (treeof number)) → Boolean
  ;; Purpose: Search the trees in the given queue for the
  ;;          given number.
  ;; How: If the queue is empty or if the root value of
  ;;  the first tree in the queue equals the given number
  ;;  then stop. Otherwise, search for the number in a
  ;;  queue that contains all but the first tree in the
  ;;  given queue and the subtrees of the first tree in
  ;;  the given queue.
  (define (bfs-helper a-num a-qton)
  ```

- ```
  ;; Tests using sample computations for bfs-helper
  (check-expect (bfs-helper 89 QTON0) QTON0-VAL)
  (check-expect (bfs-helper 99 QTON1) QTON1-VAL)
  (check-expect (bfs-helper 47 QTON2) QTON2-VAL)
  ```

Trees

bfs-helper

- ```
  ;; number (qof (treeof number)) → Boolean
  ;; Purpose: Search the trees in the given queue for the
  ;;          given number.
  ;; How: If the queue is empty or if the root value of
  ;;  the first tree in the queue equals the given number
  ;;  then stop. Otherwise, search for the number in a
  ;;  queue that contains all but the first tree in the
  ;;  given queue and the subtrees of the first tree in
  ;;  the given queue.
  (define (bfs-helper a-num a-qton)
  ```

- ```
  ;; Tests using sample computations for bfs-helper
  (check-expect (bfs-helper 89 QTON0) QTON0-VAL)
  (check-expect (bfs-helper 99 QTON1) QTON1-VAL)
  (check-expect (bfs-helper 47 QTON2) QTON2-VAL)
  ```
- ```
  ;; Tests using sample values for bfs-helper
  (check-expect (bfs-helper 31 (list (make-node 768 '())))
                #false)
  ```
  ```

Trees

bfs-helper

- ```
  ;; number (qof (treeof number)) → Boolean
  ;; Purpose: Search the trees in the given queue for the
  ;;          given number.
  ;; How: If the queue is empty or if the root value of
  ;;  the first tree in the queue equals the given number
  ;;  then stop. Otherwise, search for the number in a
  ;;  queue that contains all but the first tree in the
  ;;  given queue and the subtrees of the first tree in
  ;;  the given queue.
  (define (bfs-helper a-num a-qton)
  ```
- ```
    (and (not (qempty? a-qton))
         (or (= a-num (node-val (qfirst a-qton)))
             (local [(define newq (enqueue
                                    (node-subtrees (qfirst a-qton))
                                    (dequeue a-qton)))]
               (bfs-helper a-num newq)))))
  ```
- ```
  ;; Tests using sample computations for bfs-helper
  (check-expect (bfs-helper 89 QTON0) QTON0-VAL)
  (check-expect (bfs-helper 99 QTON1) QTON1-VAL)
  (check-expect (bfs-helper 47 QTON2) QTON2-VAL)
  ```
- ```
  ;; Tests using sample values for bfs-helper
  (check-expect (bfs-helper 31 (list (make-node 768 '())))
                #false)
  ```

Trees

Performance

-

	Experiment$_1$	Experiment$_2$	Experiment$_3$
DFS	0	156	671
BFS	281	0	36531

Trees

Performance

-

	$\texttt{Experiment}_1$	$\texttt{Experiment}_2$	$\texttt{Experiment}_3$
DFS	0	156	671
BFS	281	0	36531

- Slower when the number searched for is deep in the first subtree
- Faster when the given number is shallow in one of the trees (e.g., the root value of the last subtree)
- In the worst-case scenario (when the given number is not in the given tree) we observe that depth-first search is faster WHY????

Trees

Complexity

- Recall that `ton-dfs-contains?`'s abstract running time is $O(n)$, where `n = |V| + |E|`

Trees

Complexity

- Recall that `ton-dfs-contains?`'s abstract running time is $O(n)$, where `n = |V| + |E|`
- In the worst case (when the given number is not in the given tree) all the root values must be compared with the given number and all the edges must be traversed to reach every node

Trees

Complexity

- Recall that `ton-dfs-contains?`'s abstract running time is $O(n)$, where `n = |V| + |E|`
- In the worst case (when the given number is not in the given tree) all the root values must be compared with the given number and all the edges must be traversed to reach every node
- Every node, `v`, is visited (not searched) multiple times: every time a set of nodes is added to the queue while `v` is in the queue (by `append`)
- How many times is `v` visited while in the queue?

Trees

Complexity

- Recall that `ton-dfs-contains?`'s abstract running time is $O(n)$, where `n = |V| + |E|`
- In the worst case (when the given number is not in the given tree) all the root values must be compared with the given number and all the edges must be traversed to reach every node
- Every node, `v`, is visited (not searched) multiple times: every time a set of nodes is added to the queue while `v` is in the queue (by `append`)
- How many times is `v` visited while in the queue?
- `MAX-NUM-SUBTREES` (once for the max number of siblings)

Trees

Complexity

- Recall that `ton-dfs-contains?`'s abstract running time is $O(n)$, where `n = |V| + |E|`
- In the worst case (when the given number is not in the given tree) all the root values must be compared with the given number and all the edges must be traversed to reach every node
- Every node, `v`, is visited (not searched) multiple times: every time a set of nodes is added to the queue while `v` is in the queue (by `append`)
- How many times is `v` visited while in the queue?
- `MAX-NUM-SUBTREES` (once for the max number of siblings)
- Abstract running time: $O(\texttt{MAX-NUM-SUBTREES} * |V|+|E|)$ $=O(\texttt{MAX-NUM-SUBTREES} * n) = O(n)$
- Breadth-first search has the same complexity as depth-first search

Trees

Complexity

- Recall that `ton-dfs-contains?`'s abstract running time is $O(n)$, where `n = |V| + |E|`
- In the worst case (when the given number is not in the given tree) all the root values must be compared with the given number and all the edges must be traversed to reach every node
- Every node, `v`, is visited (not searched) multiple times: every time a set of nodes is added to the queue while `v` is in the queue (by `append`)
- How many times is `v` visited while in the queue?
- `MAX-NUM-SUBTREES` (once for the max number of siblings)
- Abstract running time: $O(\texttt{MAX-NUM-SUBTREES} * |V|+|E|)$ $=O(\texttt{MAX-NUM-SUBTREES} * n) = O(n)$
- Breadth-first search has the same complexity as depth-first search
- Why then is breadth-first search slower than depth-first search in the worst-case scenario?

Trees

Complexity

- Recall that `ton-dfs-contains?`'s abstract running time is $O(n)$, where `n = |V| + |E|`
- In the worst case (when the given number is not in the given tree) all the root values must be compared with the given number and all the edges must be traversed to reach every node
- Every node, `v`, is visited (not searched) multiple times: every time a set of nodes is added to the queue while `v` is in the queue (by `append`)
- How many times is `v` visited while in the queue?
- `MAX-NUM-SUBTREES` (once for the max number of siblings)
- Abstract running time: $O(\texttt{MAX-NUM-SUBTREES} * |V|+|E|)$ $=O(\texttt{MAX-NUM-SUBTREES} * n) = O(n)$
- Breadth-first search has the same complexity as depth-first search
- Why then is breadth-first search slower than depth-first search in the worst-case scenario?
- The constant of proportionality is larger for breadth-first search.

Trees

HOMEWORK

- Problems: 10–12
- QUIZ: Problem 8 (due in one week)

N-Puzzle Version 2

- Can we do better than a random move for the player?

N-Puzzle Version 2

- Can we do better than a random move for the player?
- To make sure a useful move the puzzle needs to be solved by our program

N-Puzzle Version 2

- Can we do better than a random move for the player?
- To make sure a useful move the puzzle needs to be solved by our program
- What is a solution?

N-Puzzle Version 2

- Can we do better than a random move for the player?
- To make sure a useful move the puzzle needs to be solved by our program
- What is a solution?
- A solution is a sequence of moves that start with a given board and end in `WIN`
- Consider the player requesting help when the board is:

1	2	3
4		6
7	5	8

N-Puzzle Version 2

- Can we do better than a random move for the player?
- To make sure a useful move the puzzle needs to be solved by our program
- What is a solution?
- A solution is a sequence of moves that start with a given board and end in `WIN`
- Consider the player requesting help when the board is:

1	2	3
4		6
7	5	8

- A solution to the puzzle consists of two moves:

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7		8

1	2	3
4	5	6
7	8	

N-Puzzle Version 2

The make-move Function

- |

N-Puzzle Version 2

The `make-move` Function

- A solution is a `(listof world)`

N-Puzzle Version 2

The `make-move` Function

- A solution is a `(listof world)`
- Given a valid board (one that may be reached from `WIN` by making 0 or more moves) a solution always exists.

N-Puzzle Version 2

The `make-move` Function

- A solution is a `(listof world)`
- Given a valid board (one that may be reached from `WIN` by making 0 or more moves) a solution always exists.
- This informs us that there is no reason for this function to declare a failed search

N-Puzzle Version 2

The `make-move` Function

- A solution is a `(listof world)`
- Given a valid board (one that may be reached from `WIN` by making 0 or more moves) a solution always exists.
- This informs us that there is no reason for this function to declare a failed search
- If the given `world` is `WIN` no moves need to be made and the given world is returned

N-Puzzle Version 2

The `make-move` Function

- A solution is a `(listof world)`
- Given a valid board (one that may be reached from `WIN` by making 0 or more moves) a solution always exists.
- This informs us that there is no reason for this function to declare a failed search
- If the given `world` is `WIN` no moves need to be made and the given world is returned
- If the given `world` is not `WIN` then a solution starts with the given board and ends with `WIN`
- This means that a solution must have at least two boards
- The second `world` must be a successor of the given `world` and is returned

N-Puzzle Version 2

The `make-move` Function

- A solution is a `(listof world)`
- Given a valid board (one that may be reached from `WIN` by making 0 or more moves) a solution always exists.
- This informs us that there is no reason for this function to declare a failed search
- If the given `world` is `WIN` no moves need to be made and the given world is returned
- If the given `world` is not `WIN` then a solution starts with the given board and ends with `WIN`
- This means that a solution must have at least two boards
- The second `world` must be a successor of the given `world` and is returned
- Finding the solution is a different problem from making a move
- The auxiliary function needs the starting `world`

N-Puzzle Version 2

The `make-move` Function

- A solution is a `(listof world)`
- Given a valid board (one that may be reached from `WIN` by making 0 or more moves) a solution always exists.
- This informs us that there is no reason for this function to declare a failed search
- If the given `world` is `WIN` no moves need to be made and the given world is returned
- If the given `world` is not `WIN` then a solution starts with the given board and ends with `WIN`
- This means that a solution must have at least two boards
- The second `world` must be a successor of the given `world` and is returned
- Finding the solution is a different problem from making a move
- The auxiliary function needs the starting `world`
- Testing `worlds`:

```
;; Sample worlds
(define WRLD1 (make-world 1 2 3
                          4 5 6
                          7 0 8))

(define WRLD2 (make-world 1 0 3
                          4 2 6
                          7 5 8))
```

N-Puzzle Version 2

The make-move Function

- ```
  ;; Sample expressions for make-move
  (define MM-WIN-VAL   WIN)
  ```
  ```

N-Puzzle Version 2

The make-move Function

- ```
  ;; Sample expressions for make-move
  (define MM-WIN-VAL   WIN)
  ```
- ```
  (define MM-WRLD1-VAL (second (find-solution WRLD1)))
  (define MM-WRLD2-VAL (second (find-sol-solution WRLD2)))
  ```

N-Puzzle Version 2

The make-move Function

- ```
  ;; world → world   Purpose: Make a move for the player
  (define (make-move a-world)
  ```
  ```

N-Puzzle Version 2

The make-move Function

- ```
  ;; world → world   Purpose: Make a move for the player
  (define (make-move a-world)
  ```

- ```
  ;; Tests using sample computations for make-move
  (check-expect (make-move WIN)   MM-WIN-VAL)
  (check-expect (make-move WRLD1) MM-WRLD1-VAL)
  (check-expect (make-move WRLD2) MM-WRLD2-VAL)
  ```

N-Puzzle Version 2

The make-move Function

- ```
  ;; world → world   Purpose: Make a move for the player
  (define (make-move a-world)
  ```
- ```
  ;; Tests using sample computations for make-move
  (check-expect (make-move WIN)   MM-WIN-VAL)
  (check-expect (make-move WRLD1) MM-WRLD1-VAL)
  (check-expect (make-move WRLD2) MM-WRLD2-VAL)
  ```
- Assume find-solution processes empty tile's first neighbor in neighbors

  ```
  ;; Tests using sample values for make-move
  (check-expect (make-move (make-world 1 2 3
                                       4 0 6
                                       7 5 8))
                (make-world 1 2 3
                            4 5 6
                            7 0 8))
  (check-expect (make-move (make-world 1 2 3
                                       4 5 0
                                       7 8 6))
                (make-world 1 2 0
                            4 5 3
                            7 8 6))
  ```

N-Puzzle Version 2

The make-move Function

- ```
  ;; world → world   Purpose: Make a move for the player
  (define (make-move a-world)
  ```
- ```
  (if (equal? a-world WIN)
      a-world
      (second (find-solution a-world))))
  ```
- ```
  ;; Tests using sample computations for make-move
  (check-expect (make-move WIN)   MM-WIN-VAL)
  (check-expect (make-move WRLD1) MM-WRLD1-VAL)
  (check-expect (make-move WRLD2) MM-WRLD2-VAL)
  ```
- Assume find-solution processes empty tile's first neighbor in neighbors

```
;; Tests using sample values for make-move
(check-expect (make-move (make-world 1 2 3
                                     4 0 6
                                     7 5 8))
              (make-world 1 2 3
                          4 5 6
                          7 0 8))
(check-expect (make-move (make-world 1 2 3
                                     4 5 0
                                     7 8 6))
              (make-world 1 2 0
                          4 5 3
                          7 8 6))
```
```

N-Puzzle Version 2

The `find-solution` Function

- A solution starts with the given board and ends with a solution that starts from one of the successors of the given board
- Definitely recursive!

N-Puzzle Version 2

The `find-solution` Function

- A solution starts with the given board and ends with a solution that starts from one of the successors of the given board
- Definitely recursive!
- When does the search for a solution terminate?

N-Puzzle Version 2

The `find-solution` Function

- A solution starts with the given board and ends with a solution that starts from one of the successors of the given board
- Definitely recursive!
- When does the search for a solution terminate?
- When the given board is `WIN` a list containing `WIN` is returned

N-Puzzle Version 2

The `find-solution` Function

- A solution starts with the given board and ends with a solution that starts from one of the successors of the given board
- Definitely recursive!
- When does the search for a solution terminate?
- When the given board is `WIN` a list containing `WIN` is returned
- What if the given board is not `WIN`?

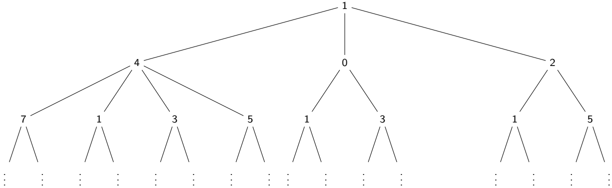
N-Puzzle Version 2

The `find-solution` Function

- A solution starts with the given board and ends with a solution that starts from one of the successors of the given board
- Definitely recursive!
- When does the search for a solution terminate?
- When the given board is `WIN` a list containing `WIN` is returned
- What if the given board is not `WIN`?
- A solution must be found from one of the successors of the given board
- How is this done?

N-Puzzle Version 2

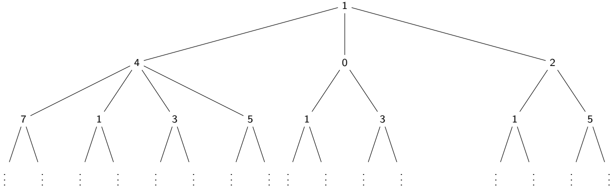
The `find-solution` Function

- 

- The is a tree (rooted at the `world` that has the empty tile space at `bpos` = 1)
- At level 1 we have the successors of the root
- At level 2 we have the successors of the root's successors and so on

N-Puzzle Version 2

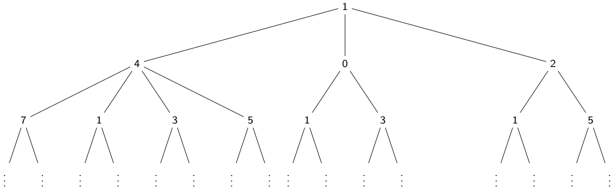
The `find-solution` Function

- 

- The is a tree (rooted at the `world` that has the empty tile space at `bpos` = 1)
- At level 1 we have the successors of the root
- At level 2 we have the successors of the root's successors and so on
- The search space being a tree is good news because we know how to search a tree

N-Puzzle Version 2

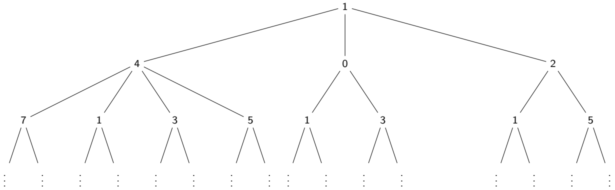
The `find-solution` Function



- The is a tree (rooted at the `world` that has the empty tile space at bpos = 1)
- At level 1 we have the successors of the root
- At level 2 we have the successors of the root's successors and so on
- The search space being a tree is good news because we know how to search a tree
- Let us explore using depth-first search

N-Puzzle Version 2

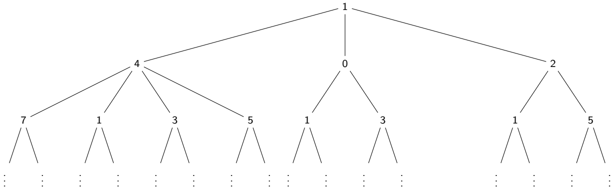
The `find-solution` Function

- 

- The is a tree (rooted at the `world` that has the empty tile space at `bpos` = 1)
- At level 1 we have the successors of the root
- At level 2 we have the successors of the root's successors and so on
- The search space being a tree is good news because we know how to search a tree
- Let us explore using depth-first search
- If the given `world` is `WIN` the solution is a list that only contains the given `world`

N-Puzzle Version 2

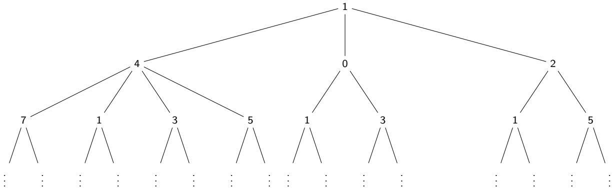
The `find-solution` Function

- 

- The is a tree (rooted at the `world` that has the empty tile space at `bpos` = 1)
- At level 1 we have the successors of the root
- At level 2 we have the successors of the root's successors and so on
- The search space being a tree is good news because we know how to search a tree
- Let us explore using depth-first search
- If the given `world` is `WIN` the solution is a list that only contains the given `world`
- If it is not `WIN` then the solution starts with the given board followed by a solution from the first successor of the given `world`

N-Puzzle Version 2

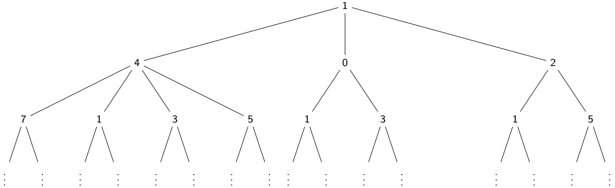
The `find-solution` Function

- 

- The is a tree (rooted at the `world` that has the empty tile space at `bpos` = 1)
- At level 1 we have the successors of the root
- At level 2 we have the successors of the root's successors and so on
- The search space being a tree is good news because we know how to search a tree
- Let us explore using depth-first search
- If the given `world` is `WIN` the solution is a list that only contains the given `world`
- If it is not `WIN` then the solution starts with the given board followed by a solution from the first successor of the given `world`
- Backtracking is not required given that a solution may be found starting from any valid board

N-Puzzle Version 2

The `find-solution` Function

- 

- The is a tree (rooted at the `world` that has the empty tile space at `bpos` = 1)
- At level 1 we have the successors of the root
- At level 2 we have the successors of the root's successors and so on
- The search space being a tree is good news because we know how to search a tree
- Let us explore using depth-first search
- If the given `world` is `WIN` the solution is a list that only contains the given `world`
- If it is not `WIN` then the solution starts with the given board followed by a solution from the first successor of the given `world`
- Backtracking is not required given that a solution may be found starting from any valid board
- It's generative recursion: recursively performs a search with a new instance of the problem

N-Puzzle Version 2

The find-solution Function

- ```
  ;; Sample expressions for find-solution
  (define FS-WIN-VAL (list WIN))
  ```
  ```

N-Puzzle Version 2

The find-solution Function

- ```
  ;; Sample expressions for find-solution
  (define FS-WIN-VAL (list WIN))
  ```
- ```
  (define FS-WRLD1-VAL
    (local
      [(define first-child
              (first (map (λ (neigh)
                             (swap-empty WRLD1 neigh))
                            (list-ref neighbors
                                      (blank-pos WRLD1)))))]
      (cons WRLD1 (find-solution first-child))))

  (define FS-WRLD2-VAL
    (local
      [(define first-child
              (first (map (λ (neigh)
                             (swap-empty WRLD2 neigh))
                            (list-ref neighbors
                                      (blank-pos WRLD2)))))]
      (cons WRLD2 (find-solution first-child))))
  ```

N-Puzzle Version 2

The find-solution Function

- ```
  ;; Sample expressions for find-solution
  (define FS-WIN-VAL (list WIN))
  ```
- ```
  (define FS-WRLD1-VAL
    (local
      [(define first-child
               (first (map (λ (neigh)
                              (swap-empty WRLD1 neigh))
                            (list-ref neighbors
                                      (blank-pos WRLD1)))))]
      (cons WRLD1 (find-solution first-child))))

  (define FS-WRLD2-VAL
    (local
      [(define first-child
               (first (map (λ (neigh)
                              (swap-empty WRLD2 neigh))
                            (list-ref neighbors
                                      (blank-pos WRLD2)))))]
      (cons WRLD2 (find-solution first-child))))
  ```
- Only difference: the `world` processed

N-Puzzle Version 2

The find-solution Function

- ```
  ;; world → (listof world) Purpose: Return a NP solution
  ;; How: The solution is built using the given world
  ;;      and the solution found starting from the
  ;;      first successor of the given world.
  (define (find-solution a-world)
  ```
  ```

N-Puzzle Version 2

The find-solution Function

- ```
  ;; world → (listof world) Purpose: Return a NP solution
  ;; How: The solution is built using the given world
  ;;      and the solution found starting from the
  ;;      first successor of the given world.
  (define (find-solution a-world)
  ```

- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution WIN)   FS-WIN-VAL)
  (check-expect (find-solution WRLD1) FS-WRLD1-VAL) ...
  ```

N-Puzzle Version 2

The find-solution Function

- ```
  ;; world → (listof world) Purpose: Return a NP solution
  ;; How: The solution is built using the given world
  ;;      and the solution found starting from the
  ;;      first successor of the given world.
  (define (find-solution a-world)
  ```

- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution WIN)   FS-WIN-VAL)
  (check-expect (find-solution WRLD1) FS-WRLD1-VAL) ...
  ```
- Trace algorithm by hand to write tests
  ```
  ;; Tests using sample values for find-solution
  (check-expect (find-solution (make-world 1 2 3 4 0 6 7 5 8)))
                (list (make-world 1 2 3 4 0 6 7 5 8)
                      (make-world 1 2 3 4 5 6 7 0 8)
                      (make-world 1 2 3 4 5 6 7 8 0))) ...
  ```

N-Puzzle Version 2

The find-solution Function

- ```
  ;; world → (listof world) Purpose: Return a NP solution
  ;; How: The solution is built using the given world
  ;;      and the solution found starting from the
  ;;      first successor of the given world.
  (define (find-solution a-world)
  ```
- ```
   (if (equal? a-world WIN)
       (list a-world)
       (local
         [(define first-child
                  (first (map (λ (neigh)
                                 (swap-empty a-world neigh))
                              (list-ref neighbors
                                        (blank-pos a-world)))))]
        (cons a-world (find-solution first-child))))
  ```
- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution WIN)   FS-WIN-VAL)
  (check-expect (find-solution WRLD1) FS-WRLD1-VAL) ...
  ```
- Trace algorithm by hand to write tests
  ```
  ;; Tests using sample values for find-solution
  (check-expect (find-solution (make-world 1 2 3 4 0 6 7 5 8)))
                (list (make-world 1 2 3 4 0 6 7 5 8)
                      (make-world 1 2 3 4 5 6 7 0 8)
                      (make-world 1 2 3 4 5 6 7 8 0))) ...
  ```
  ```

N-Puzzle Version 2

The `find-solution` Function

- Every time this function is recursively called it is given a successor of the given board as input
- Since a solution that starts with any valid board always exists eventually the given successor is `WIN` and the function halts.

N-Puzzle Version 2

The `find-solution` Function

- Every time this function is recursively called it is given a successor of the given board as input
- Since a solution that starts with any valid board always exists eventually the given successor is `WIN` and the function halts.
- Before running the tests we shall add tests for `process-key` In the textbook

N-Puzzle Version 2

A Bug: Infinite Recursion

- ```
  (process-key (make-world 2 3 0
                            1 5 6
                            4 7 8)
               HKEY)
  ```
- Never returns a value because it never terminates!
- How is that possible if we have a termination argument?
  ```

N-Puzzle Version 2

A Bug: Infinite Recursion

- ```
  (process-key (make-world 2 3 0
                           1 5 6
                           4 7 8)
               HKEY)
  ```
- Never returns a value because it never terminates!
- How is that possible if we have a termination argument?
- Sloppy work: did not think carefully about the termination argument
  ```

N-Puzzle Version 2

A Bug: Infinite Recursion

```
(find-solution (make-world 2 3 0 1 5 6 4 7 8))
→ (find-solution (make-world 2 0 3 1 5 6 4 7 8))
→ (find-solution (make-world 2 5 3 1 0 6 4 7 8))
→ (find-solution (make-world 2 5 3 1 7 6 4 0 8))
→ (find-solution (make-world 2 5 3 1 7 6 4 8 0))
→ (find-solution (make-world 2 5 3 1 7 0 4 8 6))
→ (find-solution (make-world 2 5 0 1 7 3 4 8 6))
→ (find-solution (make-world 2 0 5 1 7 3 4 8 6))
→ (find-solution (make-world 2 7 5 1 0 3 4 8 6))
→ (find-solution (make-world 2 7 5 1 8 3 4 0 6))
→ (find-solution (make-world 2 7 5 1 8 3 4 6 0))
→ (find-solution (make-world 2 7 5 1 8 0 4 6 3))
→ (find-solution (make-world 2 7 0 1 8 5 4 6 3))
→ (find-solution (make-world 2 0 7 1 8 5 4 6 3))
→ (find-solution (make-world 2 8 7 1 0 5 4 6 3))
→ (find-solution (make-world 2 8 7 1 6 5 4 0 3))
→ (find-solution (make-world 2 8 7 1 6 5 4 3 0))
→ (find-solution (make-world 2 8 7 1 6 0 4 3 5))
→ (find-solution (make-world 2 8 0 1 6 7 4 3 5))
→ (find-solution (make-world 2 0 8 1 6 7 4 3 5))
→ (find-solution (make-world 2 6 8 1 0 7 4 3 5))
→ (find-solution (make-world 2 6 8 1 3 7 4 0 5))
→ (find-solution (make-world 2 6 8 1 3 7 4 5 0))
→ (find-solution (make-world 2 6 8 1 3 0 4 5 7))
→ (find-solution (make-world 2 6 0 1 3 8 4 5 7))
→ (find-solution (make-world 2 0 6 1 3 8 4 5 7))
→ (find-solution (make-world 2 3 6 1 0 8 4 5 7))
→ (find-solution (make-world 2 3 6 1 5 8 4 0 7))
→ (find-solution (make-world 2 3 6 1 5 8 4 7 0))
→ (find-solution (make-world 2 3 6 1 5 0 4 7 8))
→ (find-solution (make-world 2 3 0 1 5 6 4 7 8))
```

N-Puzzle Version 2

Important Lessons

- Termination arguments are extremely important

N-Puzzle Version 2

Important Lessons

- Termination arguments are extremely important
- It would be nice:

```
(check-expect (halts? find-solution
                      (make-world 2 3 0 1 5 6 4 7 8))
              #true)
```

N-Puzzle Version 2

Important Lessons

- Termination arguments are extremely important
- It would be nice:

```
(check-expect (halts? find-solution
                      (make-world 2 3 0 1 5 6 4 7 8))
              #true)
```

- Determining if a given arbitrary program halts on a given arbitrary input is called *The Halting Problem*

N-Puzzle Version 2

Important Lessons

- Termination arguments are extremely important
- It would be nice:

```
(check-expect (halts? find-solution
                      (make-world 2 3 0 1 5 6 4 7 8))
              #true)
```

- Determining if a given arbitrary program halts on a given arbitrary input is called *The Halting Problem*
- Alas, `halts?` does not and cannot exist
- There is no general algorithm that can tell us that if a generative recursive program halts on a given input

N-Puzzle Version 2

Important Lessons

- Termination arguments are extremely important
- It would be nice:

```
(check-expect (halts? find-solution
                      (make-world 2 3 0 1 5 6 4 7 8))
              #true)
```

- Determining if a given arbitrary program halts on a given arbitrary input is called *The Halting Problem*
- Alas, `halts?` does not and cannot exist
- There is no general algorithm that can tell us that if a generative recursive program halts on a given input
- Another important lesson to take away is that an infinite recursion bug does not always manifest itself.
- A test for `process-key` revealed the infinite recursion bug but the tests for `find-solution` did not reveal the bug
- This is why thorough testing of programs that use generative recursion is so important

N-Puzzle Version 2

Important Lessons

- Termination arguments are extremely important
- It would be nice:

```
(check-expect (halts? find-solution
                      (make-world 2 3 0 1 5 6 4 7 8))
              #true)
```

- Determining if a given arbitrary program halts on a given arbitrary input is called *The Halting Problem*
- Alas, `halts?` does not and cannot exist
- There is no general algorithm that can tell us that if a generative recursive program halts on a given input
- Another important lesson to take away is that an infinite recursion bug does not always manifest itself.
- A test for `process-key` revealed the infinite recursion bug but the tests for `find-solution` did not reveal the bug
- This is why thorough testing of programs that use generative recursion is so important
- Our quest to find a solution to the problem of providing the player with help must continue. Any ideas on how to solve this problem?

N-Puzzle Version 3

- Depth-first search may get caught in an infinite loop
- Arises because depth-first search explores a single search path

N-Puzzle Version 3

- Depth-first search may get caught in an infinite loop
- Arises because depth-first search explores a single search path
- Either:
  - Multiple search paths must be explored simultaneously
  - Repeatedly exploring the same problem instance must be avoided

N-Puzzle Version 3

- Depth-first search may get caught in an infinite loop
- Arises because depth-first search explores a single search path
- Either:
  - Multiple search paths must be explored simultaneously
  - Repeatedly exploring the same problem instance must be avoided
- We explore a design based on the former because we know how to explore multiple paths in a tree simultaneously: breadth-first search and a queue of paths
- Goal is to redesign `make-move` to perform a breadth-first search instead of a depth-first search

N-Puzzle Version 3

The Design of `make-move`

- Purpose: solve the puzzle
- If the given `world` is `WIN` no moves are needed and the given `world` is returned

N-Puzzle Version 3

The Design of `make-move`

- Purpose: solve the puzzle
- If the given `world` is `WIN` no moves are needed and the given `world` is returned
- If the given `world` is not `WIN` then a solution must be computed using breath-first search

N-Puzzle Version 3

The Design of `make-move`

- Purpose: solve the puzzle
- If the given `world` is `WIN` no moves are needed and the given `world` is returned
- If the given `world` is not `WIN` then a solution must be computed using breath-first search
- A solution is represented as a `(listof world)`

N-Puzzle Version 3

The Design of make-move

- ```
  ;; Sample expressions for make-move
  (define MM-WIN-VAL   WIN)
  ```
  ```

N-Puzzle Version 3

The Design of make-move

- ```
  ;; Sample expressions for make-move
  (define MM-WIN-VAL   WIN)
  ```
- ```
  (define MM-WRLD1-VAL (second (find-solution-bfs
                                 (enqueue (list (list WRLD1))
                                          E-QUEUE))))
  (define MM-WRLD2-VAL (second (find-solution-bfs
                                 (enqueue (list (list WRLD2))
                                          E-QUEUE))))
  ```

N-Puzzle Version 3

The Design of make-move

- ```
  ;; world → world
  ;; Purpose: Make a move for the player
  (define (make-move a-world)
  ```
  ```

N-Puzzle Version 3

The Design of make-move

- ```
  ;; world → world
  ;; Purpose: Make a move for the player
  (define (make-move a-world)
  ```
-
- ```
  ;; Tests using sample computations for make-move
  (check-expect (make-move WIN)   MM-WIN-VAL)
  (check-expect (make-move WRLD1) MM-WRLD1-VAL)
  (check-expect (make-move WRLD2) MM-WRLD2-VAL)
  ```

N-Puzzle Version 3

The Design of `make-move`

- How are tests using sample values developed?

N-Puzzle Version 3

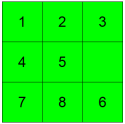
The Design of `make-move`

- How are tests using sample values developed?
- Draw the part of the search space explored by breadth-first search until a solution is returned

N-Puzzle Version 3

The Design of `make-move`

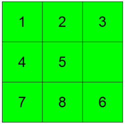
- How are tests using sample values developed?
- Draw the part of the search space explored by breadth-first search until a solution is returned
- Consider starting the search with the `world` for this image:

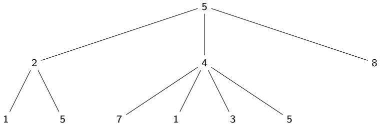


N-Puzzle Version 3

The Design of `make-move`

- How are tests using sample values developed?
- Draw the part of the search space explored by breadth-first search until a solution is returned
- Consider starting the search with the `world` for this image:

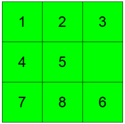


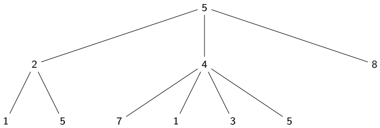
- 

N-Puzzle Version 3

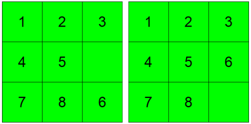
The Design of make-move

- How are tests using sample values developed?
- Draw the part of the search space explored by breadth-first search until a solution is returned
- Consider starting the search with the `world` for this image:



- 

- Path returned:



N-Puzzle Version 3

The Design of make-move

- ```
  ;; world → world
  ;; Purpose: Make a move for the player
  (define (make-move a-world)
  ```

- ```
  ;; Tests using sample computations for make-move
  (check-expect (make-move WIN)   MM-WIN-VAL)
  (check-expect (make-move WRLD1) MM-WRLD1-VAL)
  (check-expect (make-move WRLD2) MM-WRLD2-VAL)
  ```
- ```
  ;; Tests using sample values for make-move
  (check-expect (make-move (make-world 1 2 3
                                       4 5 0
                                       7 8 6))
                (make-world 1 2 3
                            4 5 6
                            7 8 0))
  (check-expect (make-move (make-world 1 2 3
                                       4 0 6
                                       7 5 8))
                (make-world 1 2 3
                            4 5 6
                            7 0 8))
  ```
  ```

N-Puzzle Version 3

The Design of make-move

- ```
  ;; world → world
  ;; Purpose: Make a move for the player
  (define (make-move a-world)
  ```
- ```
    (if (equal? a-world WIN)
        a-world
        (second (find-solution-bfs
                   (enqueue (list (list a-world)) E-QUEUE)))))
  ```
- ```
  ;; Tests using sample computations for make-move
  (check-expect (make-move WIN)   MM-WIN-VAL)
  (check-expect (make-move WRLD1) MM-WRLD1-VAL)
  (check-expect (make-move WRLD2) MM-WRLD2-VAL)
  ```
- ```
  ;; Tests using sample values for make-move
  (check-expect (make-move (make-world 1 2 3
                                       4 5 0
                                       7 8 6))
                (make-world 1 2 3
                            4 5 6
                            7 8 0))
  (check-expect (make-move (make-world 1 2 3
                                       4 0 6
                                       7 5 8))
                (make-world 1 2 3
                            4 5 6
                            7 0 8))
  ```

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Find a solution to the puzzle using breadth-first search given a queue of paths

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Find a solution to the puzzle using breadth-first search given a queue of paths
- `make-move` calls this function with an nonempty queue
- New paths are always added to the queue when a recursive call is made
- Not necessary to test if the given queue is empty in this function

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Find a solution to the puzzle using breadth-first search given a queue of paths
- `make-move` calls this function with an nonempty queue
- New paths are always added to the queue when a recursive call is made
- Not necessary to test if the given queue is empty in this function
- If the first path in the queue has reached `WIN` return it

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Find a solution to the puzzle using breadth-first search given a queue of paths
- `make-move` calls this function with an nonempty queue
- New paths are always added to the queue when a recursive call is made
- Not necessary to test if the given queue is empty in this function
- If the first path in the queue has reached `WIN` return it
- If the first path has not reached `WIN` use successors of the last world in the first path to create new paths
- First path is removed from the given queue and the new paths are added to the queue

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Find a solution to the puzzle using breadth-first search given a queue of paths
- `make-move` calls this function with an nonempty queue
- New paths are always added to the queue when a recursive call is made
- Not necessary to test if the given queue is empty in this function
- If the first path in the queue has reached `WIN` return it
- If the first path has not reached `WIN` use successors of the last world in the first path to create new paths
- First path is removed from the given queue and the new paths are added to the queue
- Testing queues (of reversed paths):

```
;; Sample (qof (listof world))
(define QLOW1 (enqueue (list (list WIN)) E-QUEUE))
(define QLOW2 (enqueue (list (list WIN
                                   (make-world 1 2 3 4 5 0 7 8 6)))
                       E-QUEUE))
(define QLOW3 (enqueue (list (list (make-world 1 2 0 4 5 3 7 8 6)
                                   (make-world 1 2 3 4 5 0 7 8 6))
                             (list (make-world 1 2 3 4 0 5 7 8 6)
                                   (make-world 1 2 3 4 5 0 7 8 6))
                             (list (make-world 1 2 3 4 5 6 7 8 0)
                                   (make-world 1 2 3 4 5 0 7 8 6)))
                       E-QUEUE))
```

N-Puzzle Version 3

The Design of find-solution-bfs

- WIN has been reached

```
;; Sample expressions for find-solution-bfs
(define FS-QLOW1-VAL
        (local [(define first-path (qfirst QLOW1))]
          (reverse first-path)))
(define FS-QLOW2-VAL
        (local [(define first-path (qfirst QLOW2))]
          (reverse first-path)))
```

N-Puzzle Version 3

The Design of find-solution-bfs

- WIN has been reached

```
;; Sample expressions for find-solution-bfs
(define FS-QLOW1-VAL
        (local [(define first-path (qfirst QLOW1))]
          (reverse first-path)))
(define FS-QLOW2-VAL
        (local [(define first-path (qfirst QLOW2))]
          (reverse first-path)))
```

- WIN has not been reached

```
(define FS-QLOW3-VAL
  (local [(define first-path  (qfirst QLOW3))
          (define first-world (first first-path))
          (define successors
                  (map (λ (neigh) (swap-empty first-world neigh))
                       (list-ref neighbors
                                 (blank-pos first-world))))
          (define new-paths (map (λ (w) (cons w first-path))
                                 successors))
          (define new-q (enqueue new-paths (dequeue QLOW3)))]
    (find-solution-bfs new-q)))
(define FS-QLOW4-VAL ...)
```

N-Puzzle Version 3

The Design of find-solution-bfs

- ```
  ;; (qof (listof world)) → (listof world)
  ;; Purpose: Return sequence of moves to WIN
  ;; How: ...
  (define (find-solution-bfs a-qlow)
  ```
  ```

N-Puzzle Version 3

The Design of find-solution-bfs

- ```
  ;; (qof (listof world)) → (listof world)
  ;; Purpose: Return sequence of moves to WIN
  ;; How: ...
  (define (find-solution-bfs a-qlow)
  ```

- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution-bfs QLOW1) FS-QLOW1-VAL) ...
  ```

N-Puzzle Version 3

The Design of find-solution-bfs

- ```
  ;; (qof (listof world)) → (listof world)
  ;; Purpose: Return sequence of moves to WIN
  ;; How: ...
  (define (find-solution-bfs a-qlow)
  ```

- ```
  ;; Tests using sample computations for find-solution
  (check-expect (find-solution-bfs QLOW1) FS-QLOW1-VAL) ...
  ```
- ```
  ;; Tests using sample values for find-solution
  (check-expect (find-solution-bfs
                  (list (list (make-world 0 2 3 1 5 6 4 7 8))))
                (list
                 (make-world 0 2 3 1 5 6 4 7 8)
                 (make-world 1 2 3 0 5 6 4 7 8)
  ```
  ```

N-Puzzle Version 3

The Design of find-solution-bfs

```
• ;; (qof (listof world)) → (listof world)
  ;; Purpose: Return sequence of moves to WIN
  ;; How: ...
  (define (find-solution-bfs a-qlow)
•   (local [(define first-path  (qfirst a-qlow))
            (define first-world (first first-path))]
     (if (equal? first-world WIN)
         (reverse first-path)
         (local
          [(define successors
                   (map (λ (neigh) (swap-empty first-world neigh))
                        (list-ref neighbors (blank-pos first-world))))
           (define new-paths (map (λ (w) (cons w first-path))
                                  successors))
           (define new-q (enqueue new-paths (dequeue a-qlow)))]
     (find-solution-bfs new-q))))
• ;; Tests using sample computations for find-solution
  (check-expect (find-solution-bfs QLOW1) FS-QLOW1-VAL) ...
• ;; Tests using sample values for find-solution
  (check-expect (find-solution-bfs
                  (list (list (make-world 0 2 3 1 5 6 4 7 8))))
                (list
                 (make-world 0 2 3 1 5 6 4 7 8)
                 (make-world 1 2 3 0 5 6 4 7 8)
```

N-Puzzle Version 3

The Design of `find-solution-bfs`

- When the first path's first `world` in the given queue is `WIN` `find-solution-bfs` halts

N-Puzzle Version 3

The Design of `find-solution-bfs`

- When the first path's first `world` in the given queue is `WIN` `find-solution-bfs` halts
- For each recursive call one path is taken one step down in the search tree and the new paths to this tree level are added to the queue

N-Puzzle Version 3

The Design of `find-solution-bfs`

- When the first path's first `world` in the given queue is `WIN` `find-solution-bfs` halts
- For each recursive call one path is taken one step down in the search tree and the new paths to this tree level are added to the queue
- FIFO ordering: all existing paths reaching the search tree's level `h` are tested to determine if they are a solution before any path that reaches level `h + 1`

N-Puzzle Version 3

The Design of `find-solution-bfs`

- When the first path's first `world` in the given queue is `WIN` `find-solution-bfs` halts
- For each recursive call one path is taken one step down in the search tree and the new paths to this tree level are added to the queue
- FIFO ordering: all existing paths reaching the search tree's level `h` are tested to determine if they are a solution before any path that reaches level `h + 1`
- Given that a solution exists starting from any valid `world` eventually one of the paths at some height `h` reaches `WIN` and the function terminates

N-Puzzle Version 3

The Design of find-solution-bfs

- When the first path's first world in the given queue is WIN find-solution-bfs halts
- For each recursive call one path is taken one step down in the search tree and the new paths to this tree level are added to the queue
- FIFO ordering: all existing paths reaching the search tree's level h are tested to determine if they are a solution before any path that reaches level h + 1
- Given that a solution exists starting from any valid world eventually one of the paths at some height h reaches WIN and the function terminates
- Does not lead to an infinite recursion!!!

```
(check-expect (process-key (make-world 2 3 0
                                       1 5 6
                                       4 7 8)
                           HKEY)
              (make-world 2 0 3
                          1 5 6
                          4 7 8))
```

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Run the game with the following board and ask for help:

1	3	8
5	2	
4	6	7

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Run the game with the following board and ask for help:

1	3	8
5	2	
4	6	7

- Takes a relatively long time...why?

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Run the game with the following board and ask for help:

1	3	8
5	2	
4	6	7

- Takes a relatively long time...why?
- `(time (find-solution-bfs (make-world 1 3 8 5 2 0 4 6 7)))`

Execution Time
32734
33250
32546
33625
33906

- 33 seconds to find a solution only has 11 moves! Why?

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Let us approximate the number of paths that exists in a full tree of height `h`
- A lower bound for the number of paths in a tree of height `h` is the number of paths in a binary tree of height `h` $=$ to the number of leaves

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Let us approximate the number of paths that exists in a full tree of height `h`
- A lower bound for the number of paths in a tree of height `h` is the number of paths in a binary tree of height `h` = to the number of leaves
-

h	Number of Paths
0	$1 = 2^h$
1	$2 = 2^h$
2	$4 = 2^h$
3	$8 = 2^h$
4	$16 = 2^h$
$\vdots$	$\vdots$

- Invariant property for all the rows of the above table is that:

  `num-paths(h) =` 2^h

- This is an exponential function which means that the number of paths in the queue grows exponentially

N-Puzzle Version 3

The Design of `find-solution-bfs`

- Let us approximate the number of paths that exists in a full tree of height `h`
- A lower bound for the number of paths in a tree of height `h` is the number of paths in a binary tree of height `h` = to the number of leaves
-

h	Number of Paths
0	$1 = 2^h$
1	$2 = 2^h$
2	$4 = 2^h$
3	$8 = 2^h$
4	$16 = 2^h$
$\vdots$	$\vdots$

- Invariant property for all the rows of the above table is that:

  `num-paths(h) =` 2^h
- This is an exponential function which means that the number of paths in the queue grows exponentially
- We must face the music: there is nothing worse than a slow video game!
- Our search for a solution to providing help to the player needs to continue

N-Puzzle Version 3

HOMEWORK

- Problems: 1, 3