Vectors and Binary Search

CS 151: Introduction to Computer Science I

J

Polymorphic Sorting

Built-in Racket function:

```
(: sort : (All (A) (Listof A) (A A -> Boolean) ->
           (Listof A)))
(define (sort my-list lte?)
  ...)
```

Sort `Strings` by length: plug in `lte?` comparing lengths
Sort `Activitys` by date or time: plug in `lte?` comparing date

Turns out our implementations of insertion sort and merge sort were pretty much polymorphic: just replace "$<$" with the given comparison function

Some rough estimates

One byte: a character or a small number

Kilo-: 1,000 (a .rkt file)

Mega-: 1,000,000 (a photo)

Giga-: 1,000,000,000 (a game, 500 gigs = one computer)

Tera-: 1,000,000,000,000 (a big hard drive)

Peta-: 1,000,000,000,000,000 (Fermilab, Hubble, big data)

Exa-: 1,000,000,000,000,000,000 (Internet = 5-300 exabytes)

Zetta-: 1,000,000,000,000,000,000,000 (all computers)

Some rough estimates

Number of people on Earth: 7.44 billion $\approx 7.4 \times 10^9$

A zettabyte (all computers): 10^{21} bytes

Number of particles in the universe: $\approx 10^{86}$

Runtime of our algorithm for computing `(fibo 500)`:
$\phi^{500} \approx 10^{104}$ recursive calls

100,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000

Some rough estimates

Number of operations a computer can do per second (flops): 10^9

In theory, could compute `(last my-list)` for lists of length up to about 10^9 in one second

In practice, lists of length a few million, 10^6, feasible, then starts to get iffy

Insertion sort won't work for 10^6

Merge sort will work for 10^6

list-ref: A Case Study in Slowness

```
(:  list-ref :  (All (A) (Listof A) Integer -> A))
(define (list-ref list index)
  (cond
    [(= index 0) (first list)]
    [else (list-ref (rest list) (- index 1))]))
```

$O(n)$ time in the worst case (getting the last few elements)

list-ref: A Case Study in Slowness

```
(:  my-long-list :  (Listof Integer))

(list-ref my-long-list 1000000)
(length my-long-list)
(cons -39 my-long-list)
```

For lists:

Accessing k-th element: $O(k)$ time

Getting length: $O(n)$ time

Adding elements (to front): $O(1)$ time

Vectors

```
(:  my-long-vec :  (Vectorof Integer))

(vector-ref my-long-vec 1000000)
(vector-length my-long-vec)
(vector-append short-vec my-long-vec)
```

For vectors:

Accessing k-th element: $O(1)$ time

Computing length: $O(1)$ time

Adding elements (to front): $O(n)$ time

Vectors

```
(:  some-primes :  (Vectorof Int))
(define some-primes (vector 2 3 5 7 9 11 13))
```

```
(vector-length some-primes) ⟹ 7
```

```
(vector-ref some-primes 3) ⟹ 7
```

```
(vector-ref some-primes 0) ⟹ 2
```

```
(vector-map add-one some-primes)
  ⟹ '# (3 4 6 8 10 12 14))
```

```
(vector-count odd?  some-primes) ⟹ 6
```

Vector Functions: last

Task: get the last element out of a vector

```
(:  vector-last :  (Vectorof Number) -> Number)
(define (vector-last vec)
  (vector-ref vec (- (vector-length vec) 1)))
```

$O(1)$ runtime!

Compared to $O(n)$ runtime for `last` of a list

Vector Functions: max-rain

Chicago gets some rain, but not that much, and for the past 50 years (starting from 1970), the number of inches of rain has been measured for that year. Given a `(Vector Int)` with this data, find the largest rainfall between 1990 and 2000.

```
(:  max-interval :  (Vectorof Int) Integer Integer
    -> Integer)
;; requires start <= end, and also that
;; the vector has nonnegative entries
(define (max-interval vec start end)
  (cond
    [(= start end) 0]
    [else (max (vector-ref vec start)
               (max-interval vec (+ start 1) end))]))

(max-interval rainfall 20 31)
```

Vector Functions: max-rain

Chicago gets some rain, but not that much, and for the past 50 years (starting from 1970), the number of inches of rain has been measured for that year. Given a `(Vector Int)` with this data, find the largest rainfall between 1990 and 2000.

Runtime analysis: $O(1)$ operations per iteration, but only `(end - start)` iterations.

$$T(n) = O(\texttt{end - start})$$

Compared to $O(n)$ runtime for list recursion

Binary Search

Q: Why are vectors useful?
A: Some interesting things can be done faster than lists!

Search problem: given a sorted list (or vector), find if a target number is in the list

Boring, normal recursive list search function:

```
(:  in?  :  (Listof Number) Number -> Boolean)
(define (in?  my-list target)
  (cond
    [(empty?  my-list) #f]
    [else (or (= target (first my-list))
              (in?  (rest my-list) target))]))
```

We wrote a recursive search function for `Tree` that was similar to this one

Binary Search

See Piazza for the code.

Runtime recurrence:

$$T(n) \leq T(n/2) + O(1)$$

Intuition for tree method: every iteration does $O(1)$ operations and halves the remaining vector to search in. Therefore the vector will be chopped down to size 1 after $\lceil \log_2 n \rceil$ steps

$$T(n) = O(\log_2 n)$$

Remark: $\lceil \log_2 n \rceil$ doesn't matter

What to know

- Merge sort
- Vectors
- Binary search