Lecture 10

Final exam - Today night
July 26 - Due Date.

Similar to midterm (Exactly same rules).

Merge - Sort

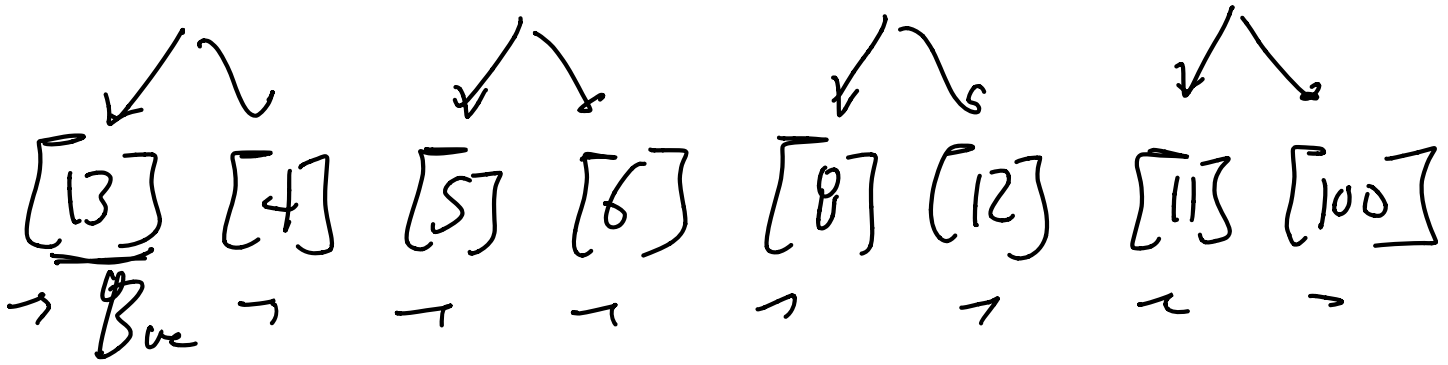
1st

[13, 4, 5, 6, 8, 12, 11, 100]

2nd

[13, 4, 5, 6] [8, 12, 11, 100].

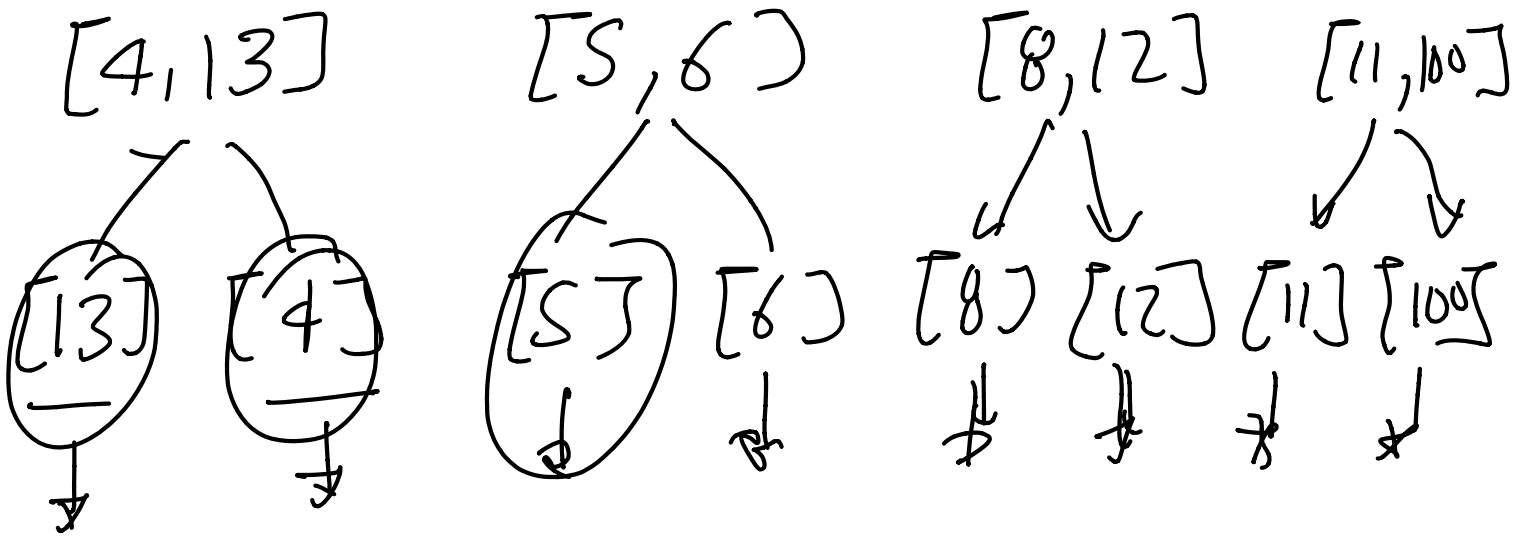
[13, 4] [5, 6] [8, 12] [11, 100]



merge from

→ [4, 5, 6, 8, 11, 12, 13, 100] → Sorted

[4, 5, 6, 13] [8, 11, 12, 100]



Vectors

Ex:

0	1	2	3	4	5
0	20	5	4	7	100

You the need the 3rd elements

index 1 ≠ index 2.

[20, 5, 4, 7, 100].

Inputs - list.

- index

Output - list item at that index.

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

Time-complexity

$T(n, k) = 5 + T(n-1, k-1)$

↑ listsize ↑ idex

$T(n, k) = 5 + T(n-1, k-1)$

$T(n, k) = \text{constant} + T(n-1, k-1)$

$T(n, k) = O(\min(n, k))$

It's too slow

We want something faster

vector (array-

[10, 5, 6, 7, 10, 11]
3.

(vector-ref 3) ⇒ 7 O(1)

constant

Vectors are stored in contiguous memory location.

Example.

```
(: my-long-vec: (Vectorof Integer))
(define my-long-vec (vector 10 30 40))
```

0th 1st

```
(vector-ref my-long-vec 0)
```

→ 10 - output

```
(vector-ref my-long-vec 1)
```

→ 30 - output.

```
(vector-length my-long-vec)
(vector-append short-vec my-long-vec).
```

Time complexity of vector-length

it is O(1) - constant.

Till now we have only done functional programming.

my-list

[10, 30, 40, 60, 70]

(rest my-list)

→ This doesn't my-list rather it creates a new list

[30, 40, 60, 70].

Imperative programming.

Vector set

```
(: vec : (Vectorof Integer))
(define vec (vector 12 9 6))
(vec-set! vec 1 15)
```

It will change element at index 1 to 15.

[12, 9, 6]

↓ changed

[12, 15, 6]

(vector-ref vec 1) → 15.

What is the type annotation of vector-set!

```
(: vector-set! : (All (A) (Vectorof A)
  Integer A -> Void))
```

Output is void.
because there is
no output

There are functions map for vectors as well.
map- for vectors.

```
(vector-map add-one my-vec).
```

User defined function

which's add 1 to any integer

Output: Every element of my-vec will be incremented by 1.

Example (Swap)

[0, 1, 10, 5, 6] - input

i - input 2

j 2 - input 3

```
(: my-vec : (Vectorof Integer))
(define my-vec (vector 0 1 10 5 6))
```

Call swap! before doing checkexpect.

```
(swap! my-vec 1 2)
```

[0, 10, 1, 5, 6]

```
(check-expect (vector-ref my-vec
                          1) 10)

(check-expect (vector-ref my-vec
                          2) 1).
```

Check expect for list

```
(check-expect (f (list 1 2 3))
              5).
```

↑ This won't work here

```
(: swap! : (Vectorof Integer)
           Integer Integer
        -> Void)
(define (swap! vec i j)
  (local
    [(: tmp : Integer)
     (define tmp (vector-ref vec i))
     ]
  (begin
    (vector-set! vec i (vector-ref vec j))
    (vector-set! vec j tmp))))
```

OP 1 ← OP 1

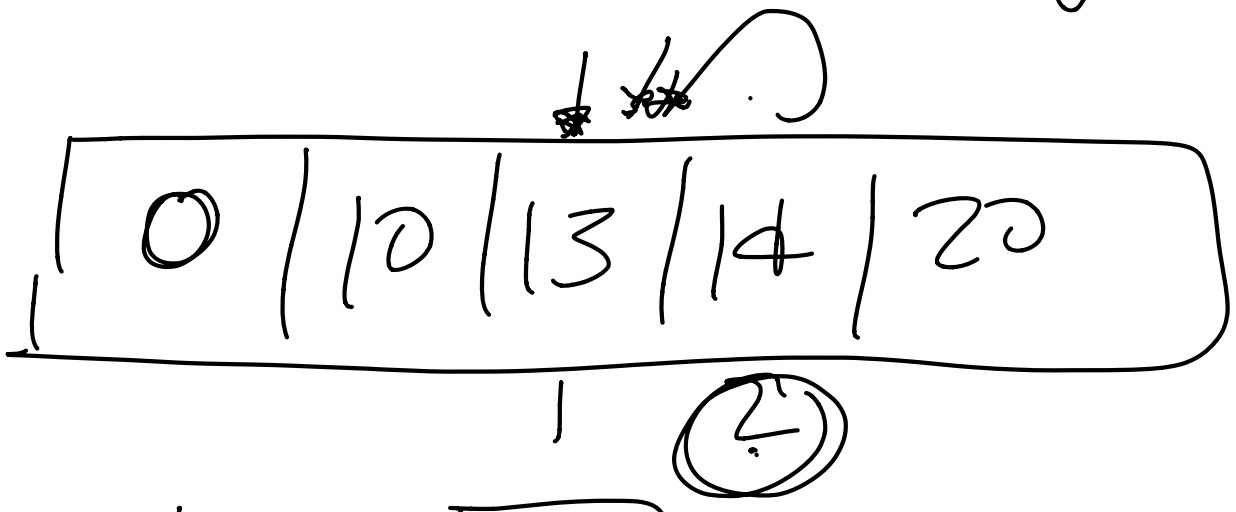
OP 1

Change the value at position i to change the value at position j

set the position.

i ← vec j

set the position at j ← tmp



vec[1] ← 44

vec[2] ← tmp.

(begin

allows you to combine
multiple updates in row.

Example

```
(: vec : (Vectorof Integer))
(define vec (Vec 100 200 300))

(begin (vector-set! vec 0 150)
       (vector-set! vec 1 100)
       (vector-ref! vec 1))
```

⇒ 100

100	200	300

After first vector-set!

150	200	300

After second vector-set!

150	100	300

Vector ref
Position 1 contains 100

And works on boolean values

(and #t #f) ⇒ #f

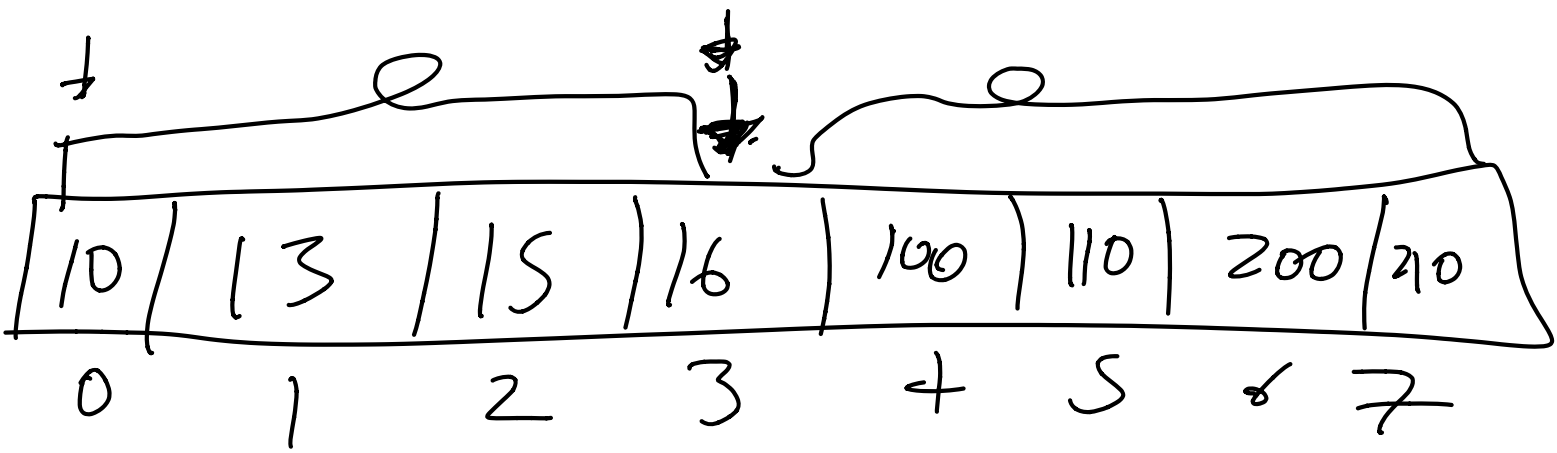
Time complexity of swap!

$T(n) = 4 = O(1)$

constant (Doesn't depend on the input size)

Binary-search

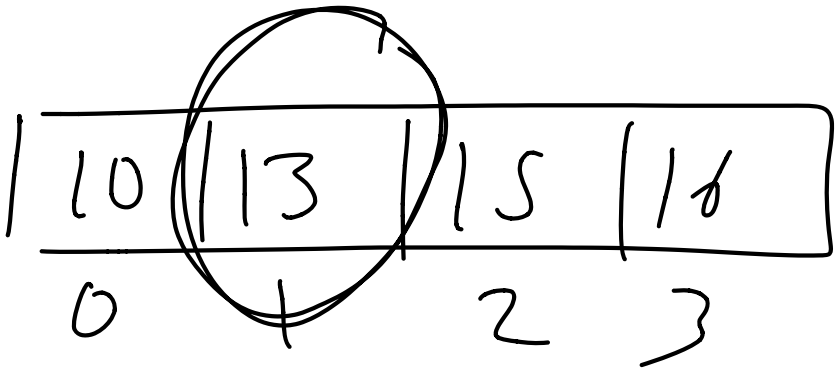
Sort vector



Check if 13 is in this array

Check if $13 \leq 16$ or is it $13 > 16$

if 13 exists it should be on the left-half of the vector.



13 = 13 the element is there?

Binary Search

```
(: bin-search? : (Vectorof Integer)
                 Integer → Boolean)

(define (bin-search? vec target)
  (
    local
    {
     (: in-interval? : Integer Integer
                     → Boolean)

  (cond
   [(= lo hi) #f] ← lop
   [(= (+ lo 1) hi) (= (vector-ref
   [else                vec lo) target)]
                ← top
   (local
```

{ (: mid : Integer) ← 2 OP
(define mid (quotient (+ lo hi) 2))
(: mid-elem : Inkeys) 1 OP
(define mid-elem (vector-ref vec mid)) }
(cond
[(= mid-elem target) #t] 1 OP
night half
↓ — T(n/2)
↓ — 1 OP
[(< mid-elem target) (in-interval
↓ left-half (+ mid 1) hi)]
→ [(> mid-elem target) (in-interval T(n/2)
lo mid)]
[else (error "Error")])) } ← local closes
(in-interval? 0 (vector-length vec)))))!
↑ Initial lo ↑ Initial hi

lo ↓ 2 ↓lo 6 ↓hi ↓hi hi ↓

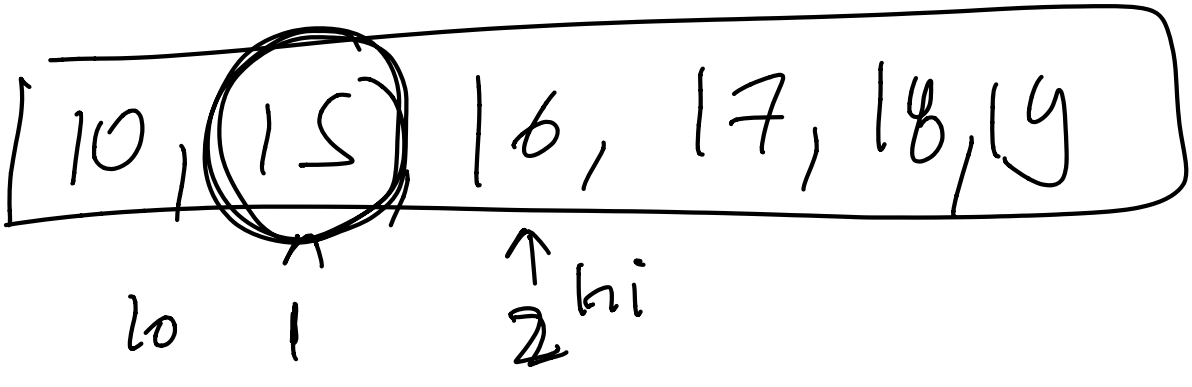
[10, 13, 14, 15, 16, 20, 21]

(in-interval? 2 5) ⇒ #f false

target is 20

(in-interval? 2 6) ⇒ #true

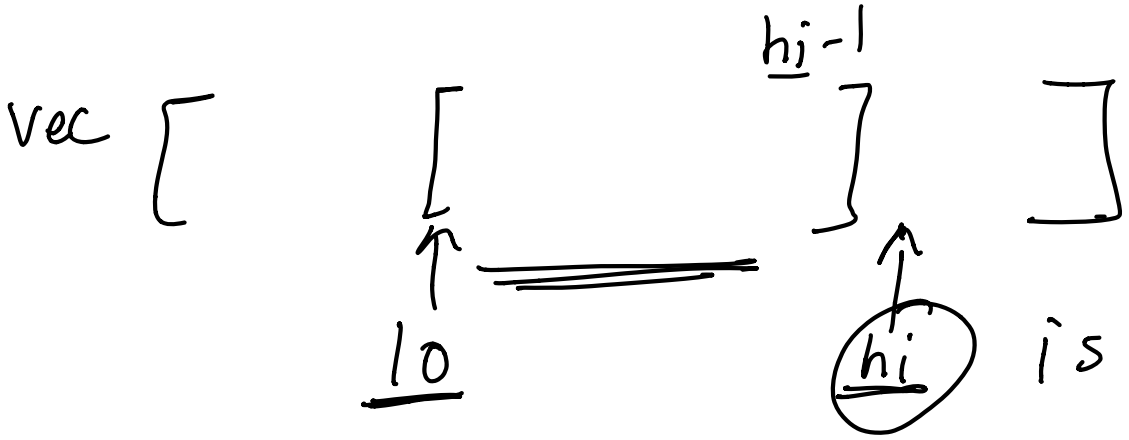
[2, 2]



(= (vector-ref vec 10) 15)

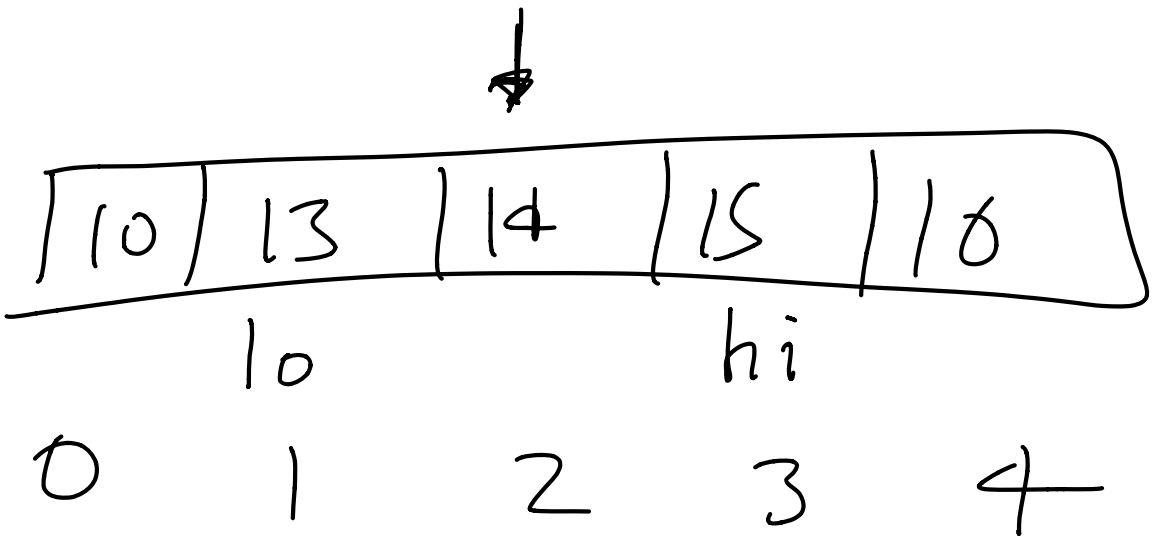
in-interval inside local.

Two inputs integer lo
integer. hi



return true if target is somewhere between lo and hi

5



mid = 2

target = 20

[lo, mid, hi]

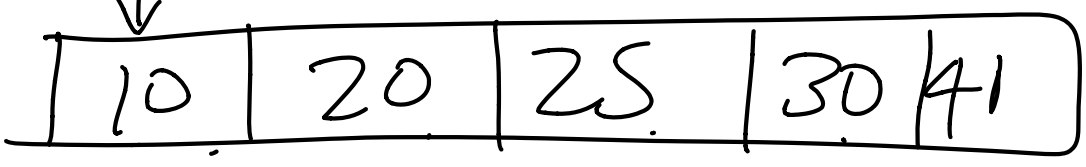
[mid+1, hi]

[lo, mid] [mid+1, hi]

Initial lo and hi value

lo = 0 initial

hi = 5 ↓ length of the vector.



Divide and conquer

We divide the vector in two -halves right-half and left-half

Then check in which half can your target may exist.

Time - complexity.

Time-complexity : T(n)

= Time complexity of in-interval

What is input size

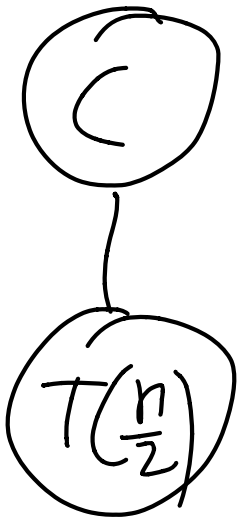
inputs for in-interval lo, hi

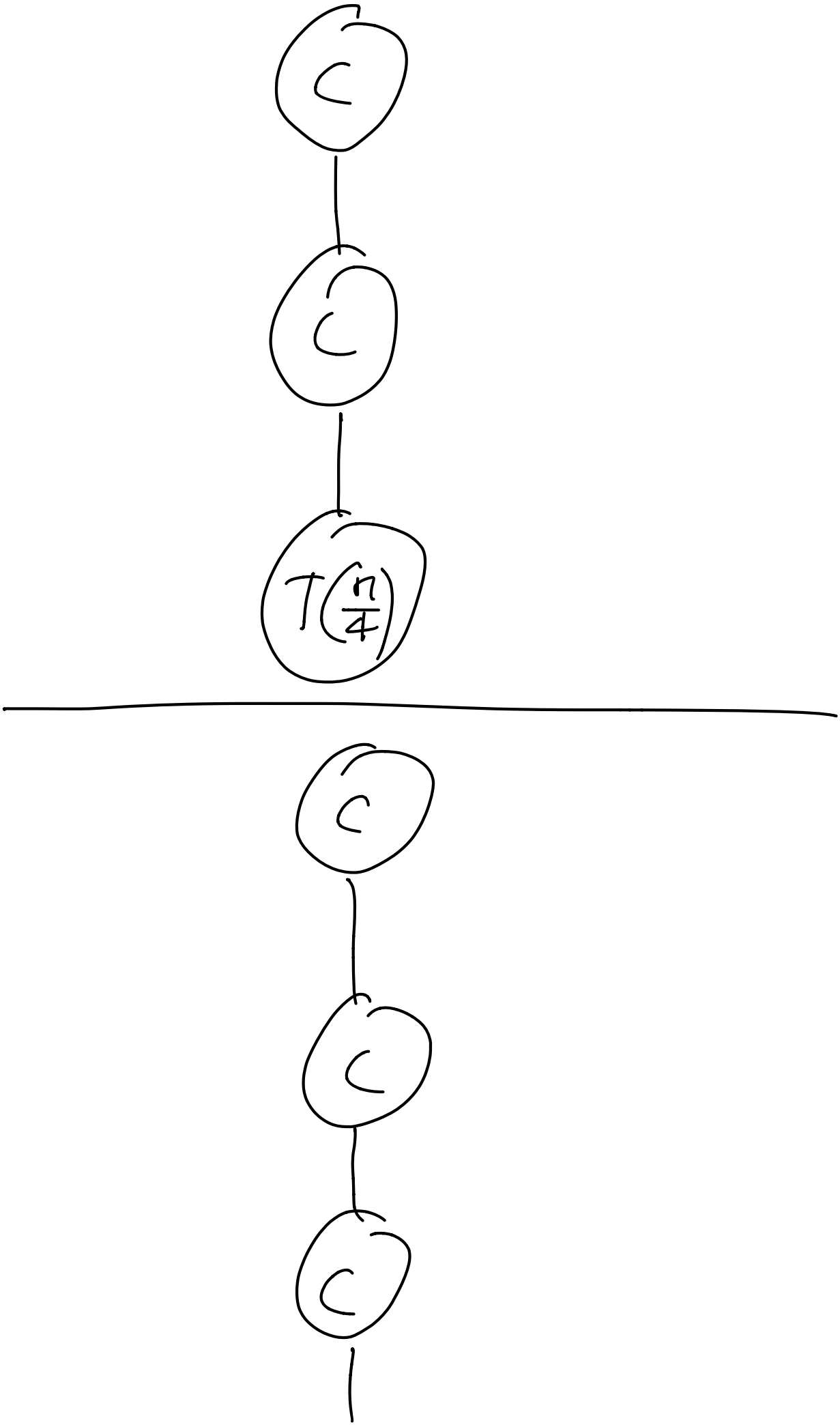
size of the range = hi - lo

n - size of the range

$$T(n) = \text{constant} + T\left(\frac{n}{2}\right)$$
$$T(n) = C + T\left(\frac{n}{2}\right)$$

Recursion Tree




C
C
T(n/4)
C
C
C
1

1

$T(\frac{n}{8})$

What of this tree.
If the height of this tree is i

$$\frac{n}{2^i} = 1.$$

$$2^i = n$$

$$i = \log_2 n$$

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

$$= O(\log_2 n)$$

Time complexity of Binary Search.

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

n = length of the vector (size of the initial range).