Binary Search (review)

Linear Search Recall

Linear search

- Keep the array in any order
- Examine the array starting from first entry until key is found or not found.

```
public static int linearSearch ( String key, String[] a )
{
    for ( int i = 0; i < a.length; i++ )
        if ( a[i].compareTo(key) == 0 ) return i;
    return -1;
}
```

Linear search

i	a[i]
0	alice
1	bob
2	carlos
3	carol
4	craig
5	dave
6	erin
7	eve
8	frank
9	mallory
10	oscar
11	peggy
12	trent
13	walter
14	wendy

oscar?

Match found.
Return 10

2

Binary Search Recall

Binary search

- Keep the array in sorted order
- Examine the middle key.
- If it matches, return its index.
- If it is larger, search the half with lower indices.
- If it is smaller, search the half with upper indices.

i	a[i]
0	alice
1	bob
2	carlos
3	carol
4	craig
5	dave
6	erin
7	eve
8	frank
9	mallory
10	oscar
11	peggy
12	trent
13	walter
14	wendy

oscar?

Binary search arithmetic

Search in a[lo,hi)
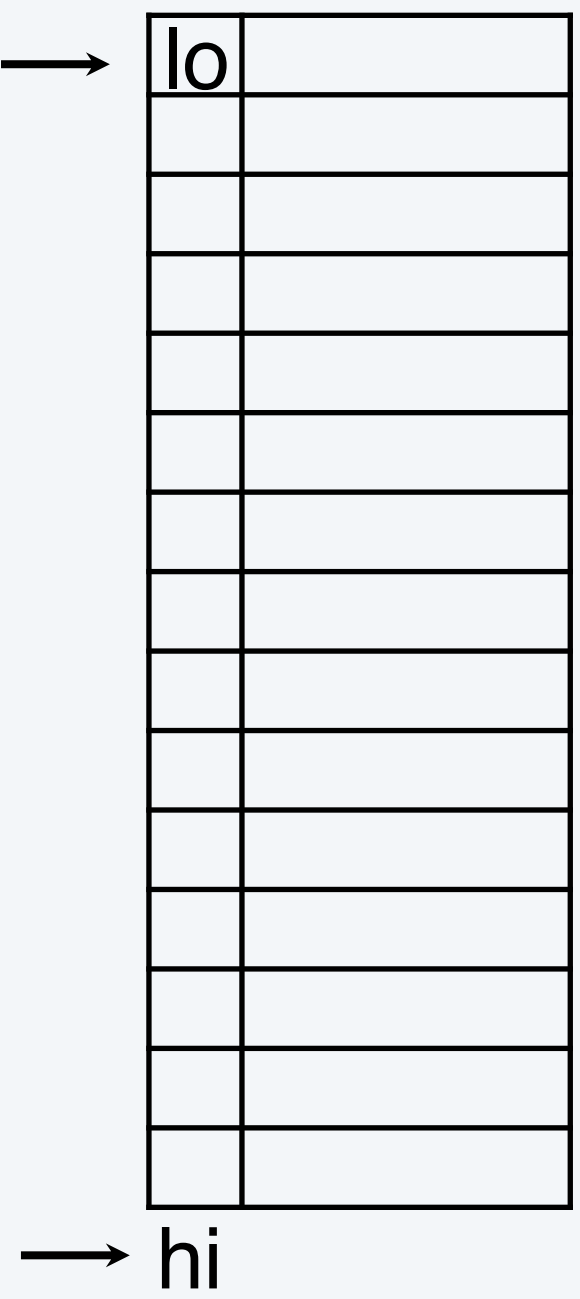
mid = lo + (hi-lo)/2
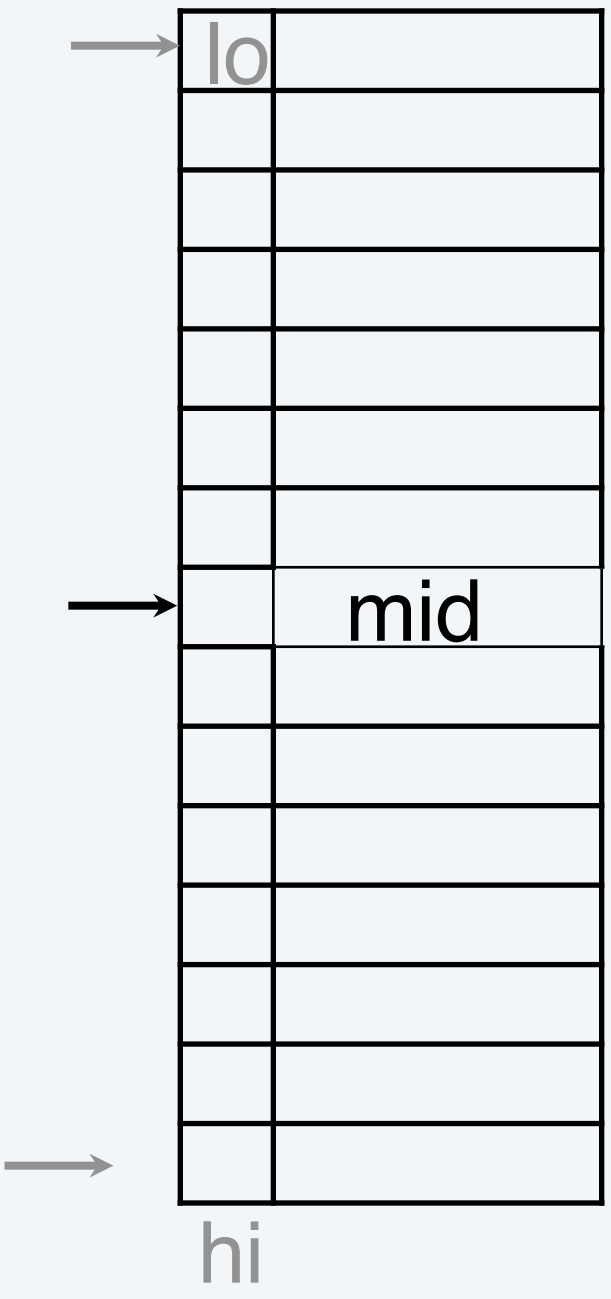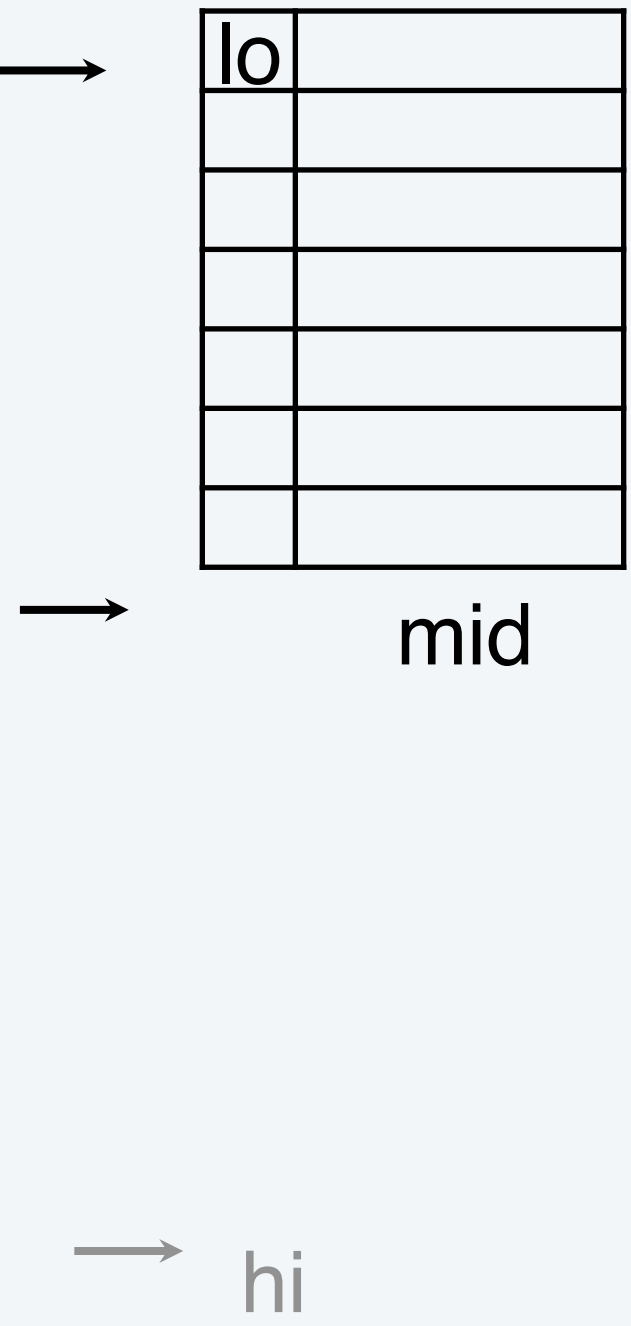
Lower half: a[lo,mid)

Upper half: a[mid+1,hi)

lo

hi

lo

mid

hi

lo

mid

hi

lo

mid+1

mid

hi
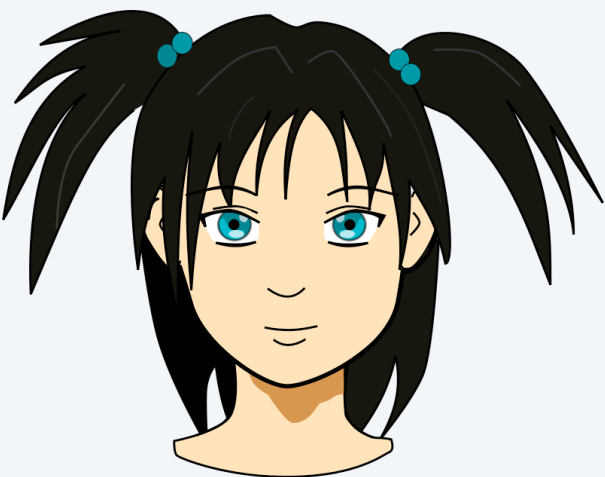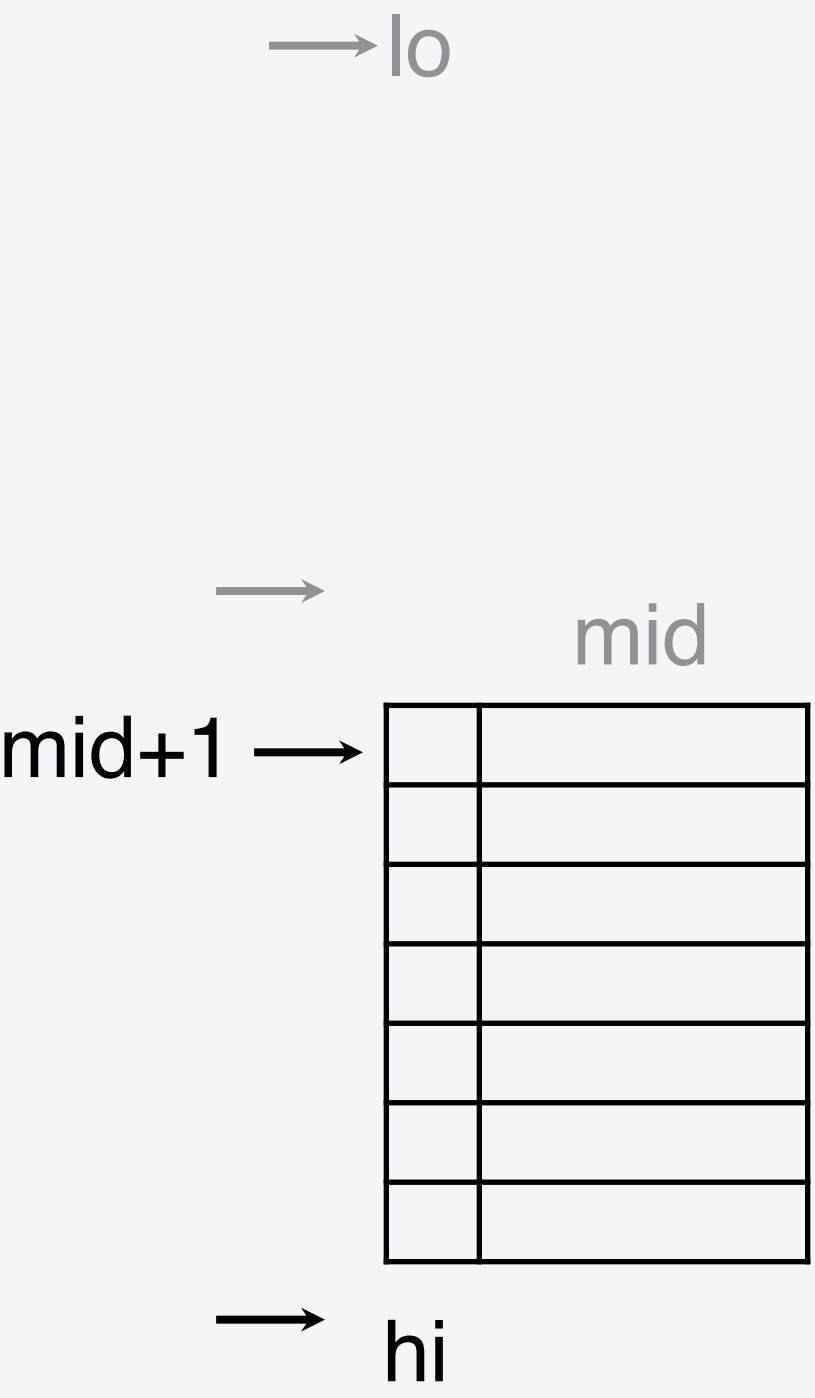
Tricky! Needs study...

```java
public static int search(String key, String[] a)

{

    return search(key, a, 0, a.length);

}
```

```java
public static int search(String key, String[] a, int lo, int hi)

{

    if (hi <= lo) return -1;

    int mid = lo + (hi - lo) / 2;

    int cmp = a[mid].compareTo(key);


    if (cmp == 0) return mid;

    else if  (cmp > 0) return search (key, a, lo, mid);

    else return search (key, a, mid+1, hi);

}
```

a.**compareTo**(b) returns:

0 if a equals to b

< 0 if a is less than b

> 0 if a is greater than b

Still, this was easier than I thought!

Understanding binary search

Identify. All elements that are inspected during binary search

comparisons. 4 names inspected – 4 array accesses

i	a[i]
0	alice
1	bob
2	carlos
3	carol
4	craig
5	dave
6	erin
7	eve
8	frank
9	mallory
10	oscar
11	peggy
12	trent
13	walter
14	wendy

oscar?

Match found.
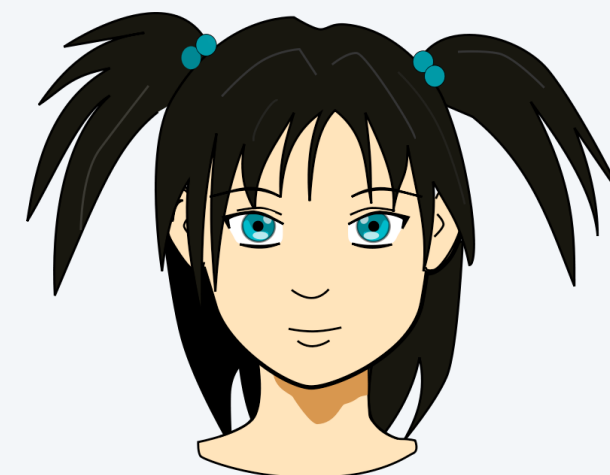Return 10

```java
public static int search(String key, String[] a)

{

    return search(key, a, 0, a.length);

}


public static int search(String key, String[] a, int lo, int hi)

{

    if (hi <= lo) return -1;

    int mid = lo + (hi - lo) / 2;

    int cmp = a[mid].compareTo(key);


    if (cmp == 0) return mid;

    else if (cmp > 0) return search (key, a, lo, mid);

    else return search (key, a, mid+1, hi);

}
```

```
search("oscar")
    return search(... 0, 15);


search("oscar", a, 0, 15)
    mid = 7;
    > "eve"
    return search(... 8, 15);

search("oscar", a, 8, 15)
    mid = 11;
    < "peggy"
    return search(... 8, 11);

search("oscar", a, 8, 11)
    mid = 9;
    > "mallory"
    return search(... 10, 11);

search("oscar", a, 10, 11)
    mid = 10;
    == "oscar"
    return 10;
```
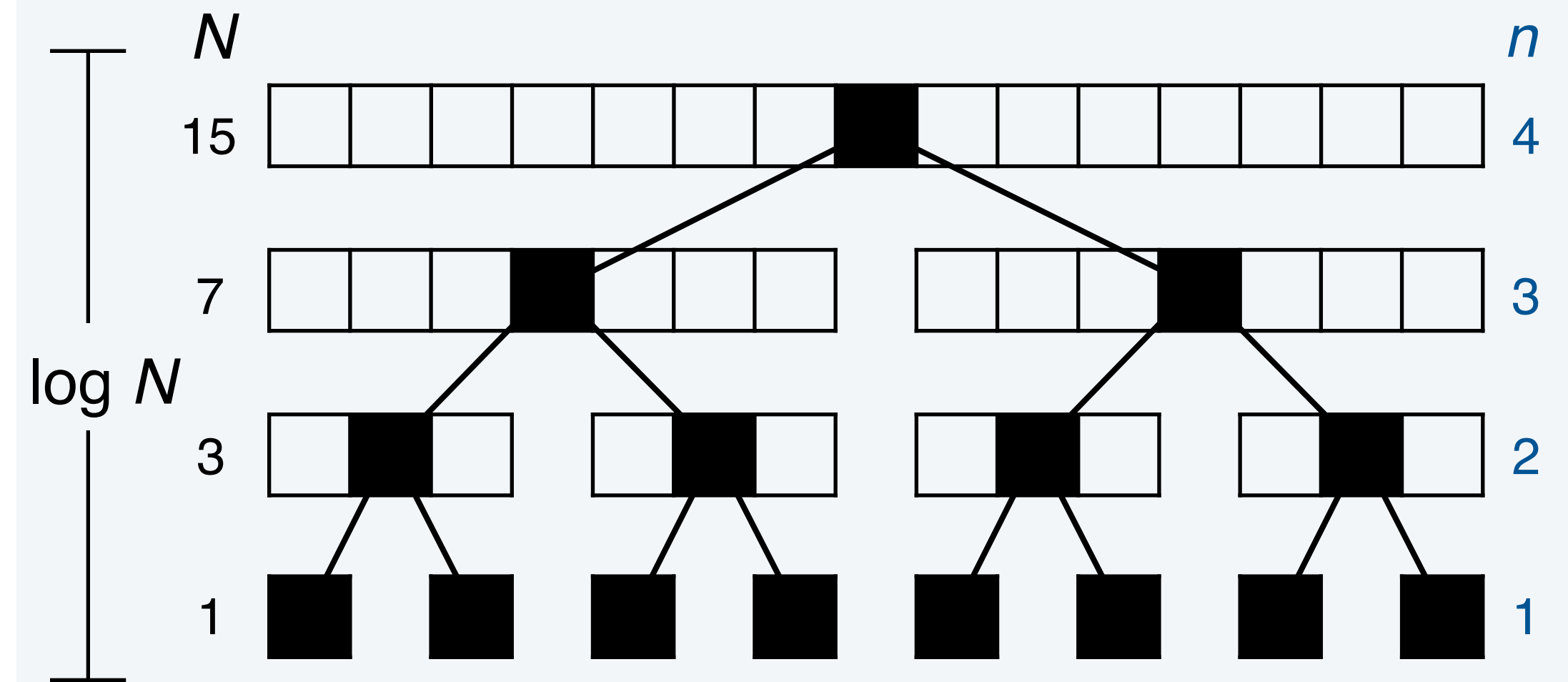
i	a[i]
0	alice
1	bob
2	carlos
3	carol
4	craig
5	dave
6	erin
7	eve
8	frank
9	mallory
10	oscar
11	peggy
12	trent
13	walter
14	wendy

Exact analysis for search miss for $N = 2^n - 1$

- Note that $n = \log(N+1) \sim \log N$.

- Subarray size for 1st call is $2^n - 1$.

- Subarray size for 2nd call is $2^{n-1} - 1$.

- Subarray size for 3rd call is $2^{n-2} - 1$.

- ...

- Subarray size for nth call is 1.

- Total # compares/array accesses (one per call): $n \sim \log N$.

Every search miss is a top-to-bottom path in this tree.

Proposition. Binary search uses $\sim\log N$ compares for a search miss.

Proof. An (easy) exercise in discrete math.

Proposition. Binary search uses $\sim\log N$ compares for a random search

Proof. A slightly more difficult exercise in discrete math.

Best case. Oscar is the a[mid] record (1 comparison)

Worst case. Oscar is end/begin key (log N comparisons)

Average case. Oscar is some key (log N comparisons)

i	a[i]
0	alice
1	bob
2	carlos
3	carol
4	craig
5	dave
6	erin
7	eve
8	frank
9	mallory
10	oscar
11	peggy
12	trent
13	walter
14	wendy

oscar?

Empirical tests of binary search

Whitelist filter scenario

- Whitelist of size N.
- $10N$ transactions.

N	T_N (seconds)	$T_N/T_{N/2}$	transactions per second
100,000	1		
200,000	3		
400,000	6	2	67,000
800,000	14	2.35	57,000
1,600,000	33	2.33	48,000
10.28 million	264	2	48,000

% java Generator 100000 ...
1 seconds
% java Generator 200000 ...
3 seconds
% java Generator 400000 ...
6 seconds
% java Generator 800000 ...
14 seconds
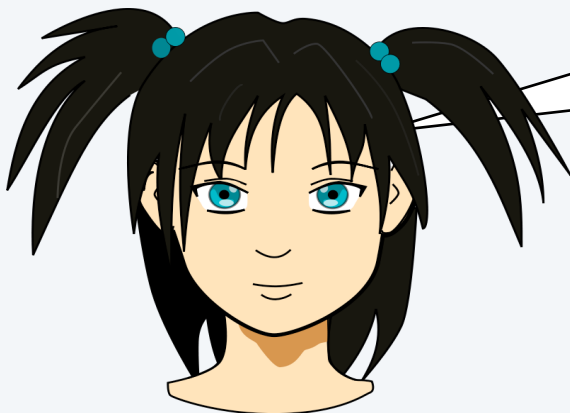% java Generator 1600000 ...
33 seconds

... = 10 a-z I java TestBS
a-z = abcdefghijklmnopqrstuvwxyz

nearly 50,000 transactions per second, and holding

Validates hypothesis that order of growth is $N\log N$.

Will scale.

Great! But how do I get the list into sorted order at the beginning?

Building the Decision Tree

Decision tree is a theoretical tool used to analyze the running time of algorithms. It illustrates the possible executions on inputs.

Use the following implementation of binary search to build the tree for any size array.

Search for every item in the array until the tree is fully built.

```java
public static int search (int key, int [] a)
{
    return search (key, a, 0, a.length);
}


public static int search (int key, int [] a, int lo, int hi)
{
    if ( hi <= lo ) return -1; // key is not present in array a
    int mid = lo + (hi - lo) / 2;

    int cmp = a[mid].compareTo(key);

    if      ( cmp == 0 ) return mid;
    else if ( cmp > 0 )  return search (key, a, lo, mid);
     else return search (key, a, mid+1, hi);
}
```
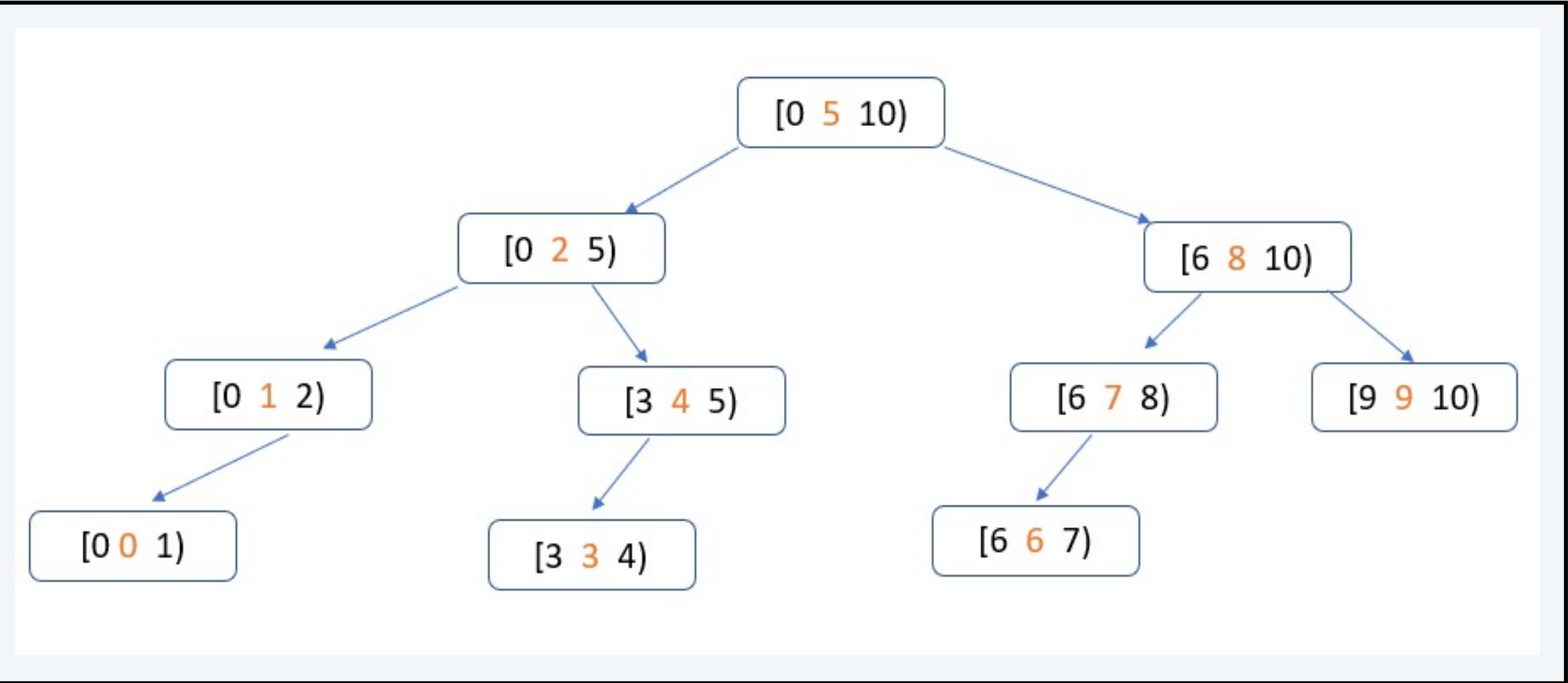
0	1	2	3	4	5	6	7	8	9
11	12	17	19	26	38	45	62	69	83

We'll use an array of size 10 as an example.

An array of size 10



Mid point in each interval considered marked orange

We assume the interval [lo hi) and left interval [lo mid) and right interval [mid+1, hi) are considered in each step.

The tree depicts all possible search paths for binary search for keys that may or may not be present.

Q. What is the maximum number of comparisons to find (or not find) an element?
A. Proportional to height of the tree

Full trees of various size			Number of Comparisons
N = 1	N = 3	N = 7	

	1
	2
	3
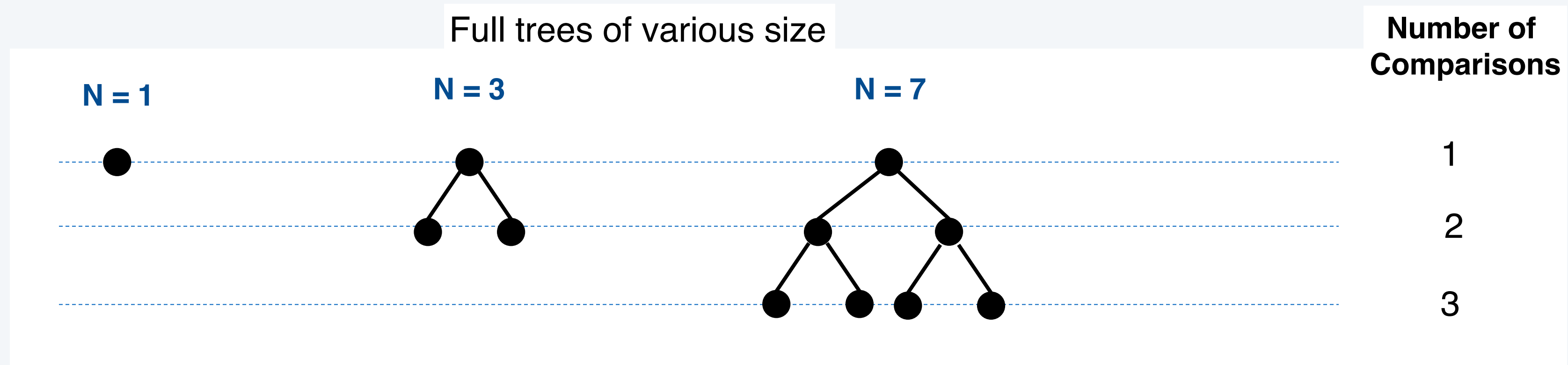
Assume a full tree, array sizes 1, 3, 7, 15, 31, … ← these are all $2^{height} - 1 = N$

The worst-case number of compares for an array of size N is proportional to the height.
height = log (N + 1)

Q. What is the height of a decision tree for an array of size 10?

A. An array of height 10 has the height of 4.
   Assume a full tree, 10 items do not fit in an array N = 7, so we go to N = 15
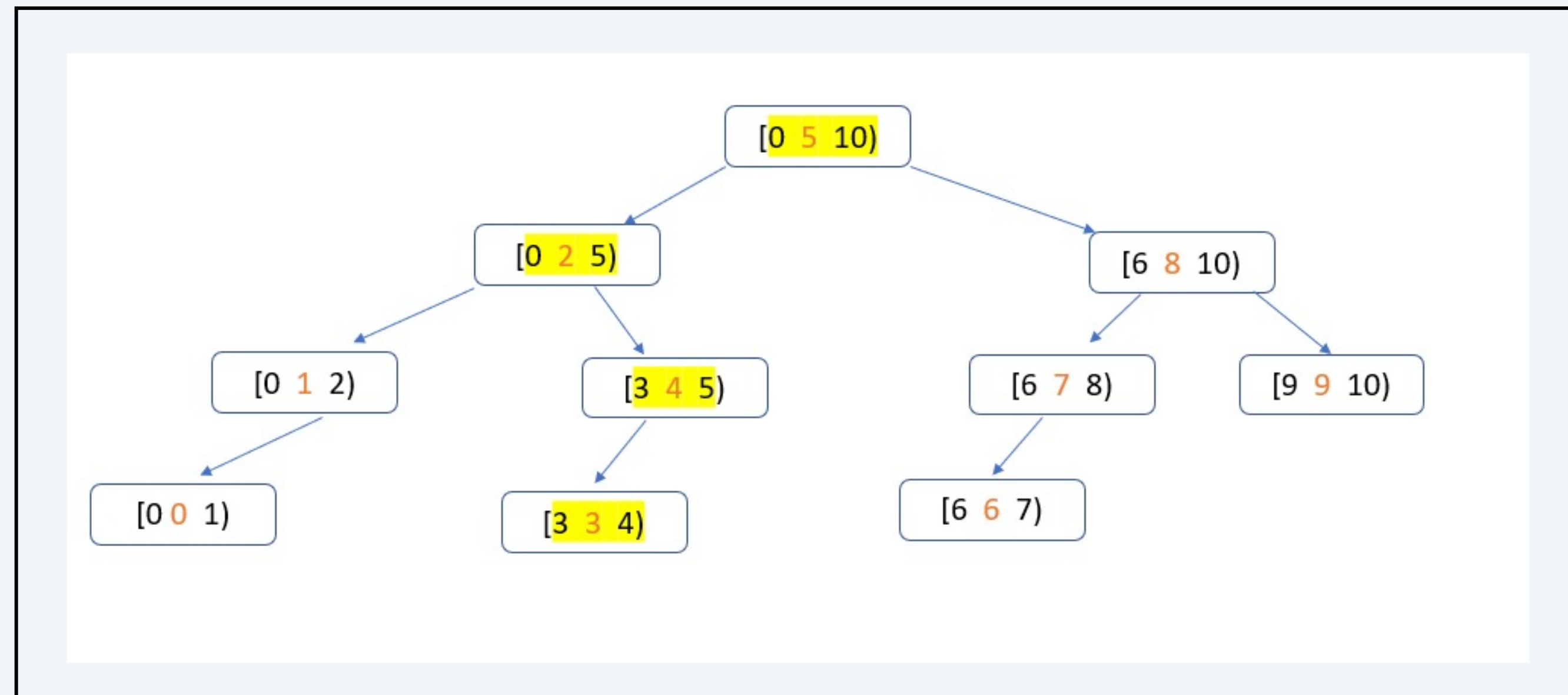   **height** = log (N + 1) = log (15 + 1) = 4

An Example

0	1	2	3	4	5	6	7	8	9
11	12	17	19	26	38	45	62	69	83

Searching for Target 19
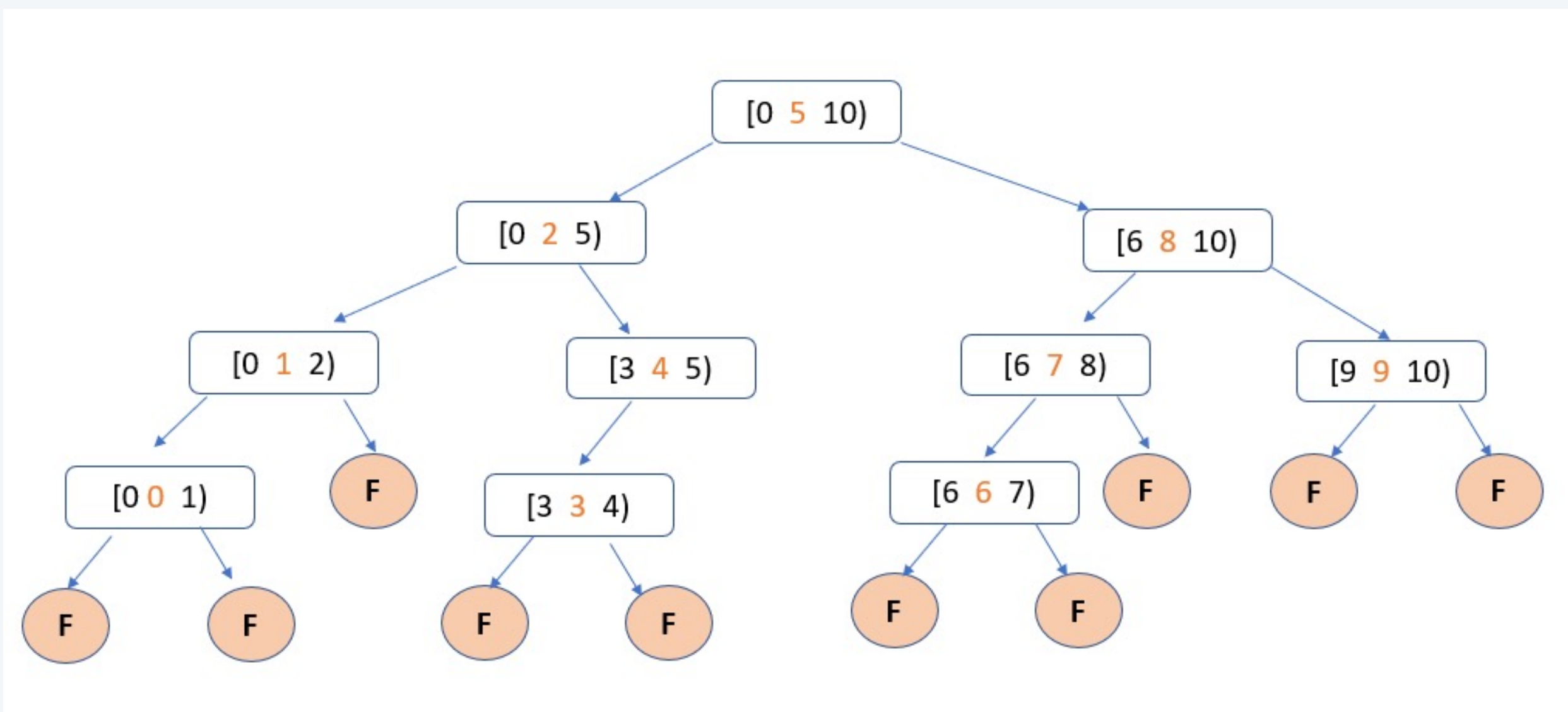## Found at array index 3

Count the calls along the search path to find the number of compares.

Q. What is the number of comparisons to find key 19 (at index 3)?

A. 4 compares

[0 5 10)

[0 2 5)   [6 8 10)

[0 1 2)   [3 4 5)   [6 7 8)   [9 9 10)

[0 0 1)   [3 3 4)   [6 6 7)

0	1	2	3	4	5	6	7	8	9
11	12	17	19	26	38	45	62	69	83

Successful Search – index of the search element is an inner node

Unsuccessful Search – occurs at the leaf nodes marked as F

At each node there is ONE comparison to decide which way to go.
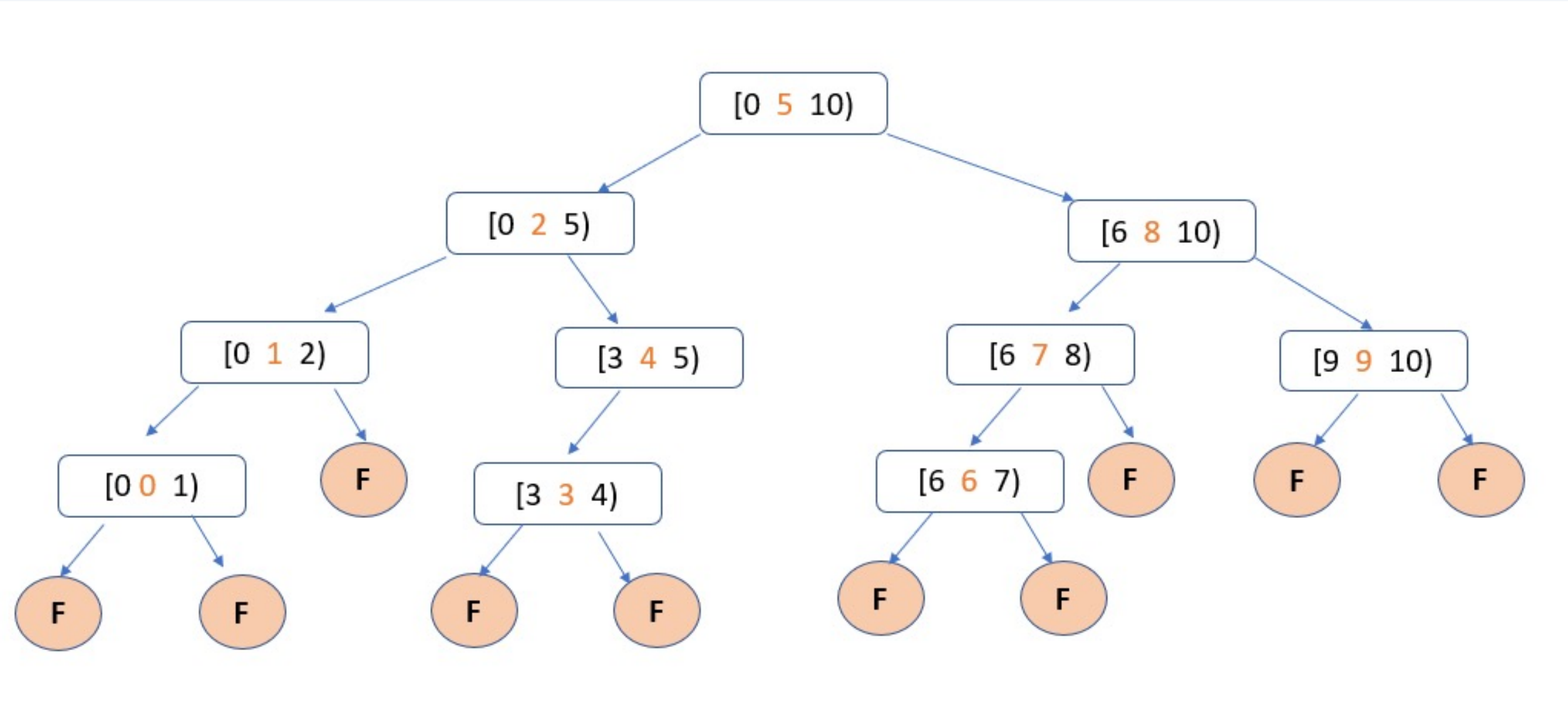
Worst-case for successful search

in this example the indices {0, 3, 6} have the longest path of 4 comparisons for successful search.

Worst-case for unsuccessful search

items that fail at the nodes that have the worst-case successful search.

Complexity of the search ~ log N

15

0	1	2	3	4	5	6	7	8	9
11	12	17	19	26	38	45	62	69	83

Successful Search – average the number of comparisons for successful searches.

Index	#comparisons
5	1
2	2
8	2
1	3
4	3
7	3
9	3
0	4
3	4
6	4
	29 / 10 = 2.9

Average number of compares for success

Unsuccessful Search – if we don't know how many keys are being searched, we can only affirm that the average will be between 3-4.

16

Binary Search Review

http://ds.cs.rutgers.edu