LEARN CODING

ale66

github.com/ale66/learn-coding

SORTING

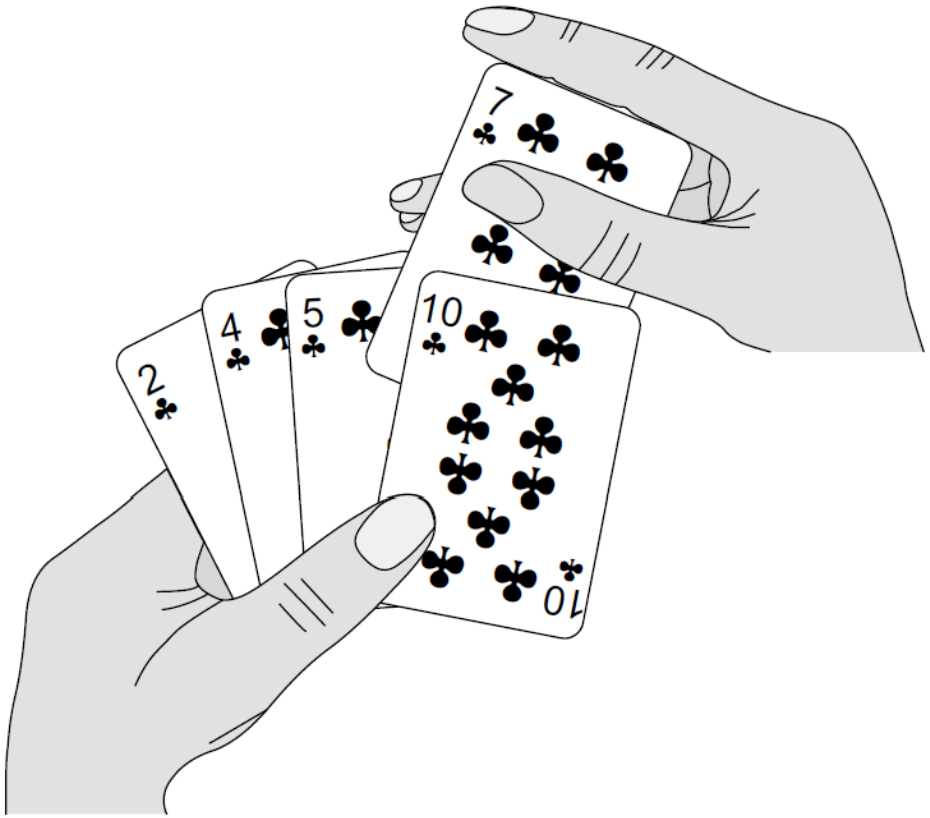


Figure 2.1 Sorting a hand of cards using insertion sort.

STATEMENT

Instance:

a sequence of *n* integers: $A = a_1, a_2 \ldots a_n$

Solution:

A permutation of the values $\pi : [1..n] \rightarrow [1..n]$

Constraint

values never decrease: $a_1 \leq a_2 \ldots a_n$

Let's assume that there are no repeated values and Python notation (first elem. is in pos. 0)

```
1  A = [11, 6, 8, 2, 22, 16, 25]
2
3  sort(A) = [?, ?, ?, ?, ?, ?]
```

$\pi(0) = ?$

$\pi(1) = ?$

$\pi(2) = ?$

$\pi(3) = ?$

...

Let's assume that there are no repeated values

```
1  A = [11, 6, 8, 2, 22, 16, 25]
2
3  sort(A) = [2, 6, 8, 11, 16, 22, 25]
```

$\pi(0) = 3$: the elem. in position 0 now goes to pos. 3

$\pi(1) = 1$: the elem. in pos. 1 remains there

$\pi(2) = 2$: so does the elem. in pos. 2

$\pi(3) = 0$: the elem. in pos. 3 now goes to pos. 0

...

GOOD NEWS ABOUT SORTING

Solvable within $Kn \log_2 n$ steps

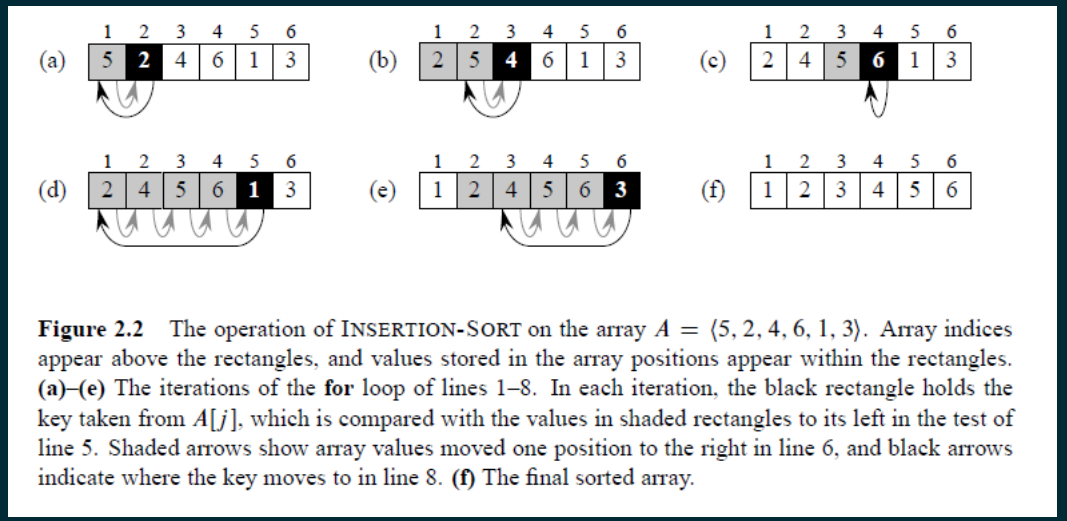


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

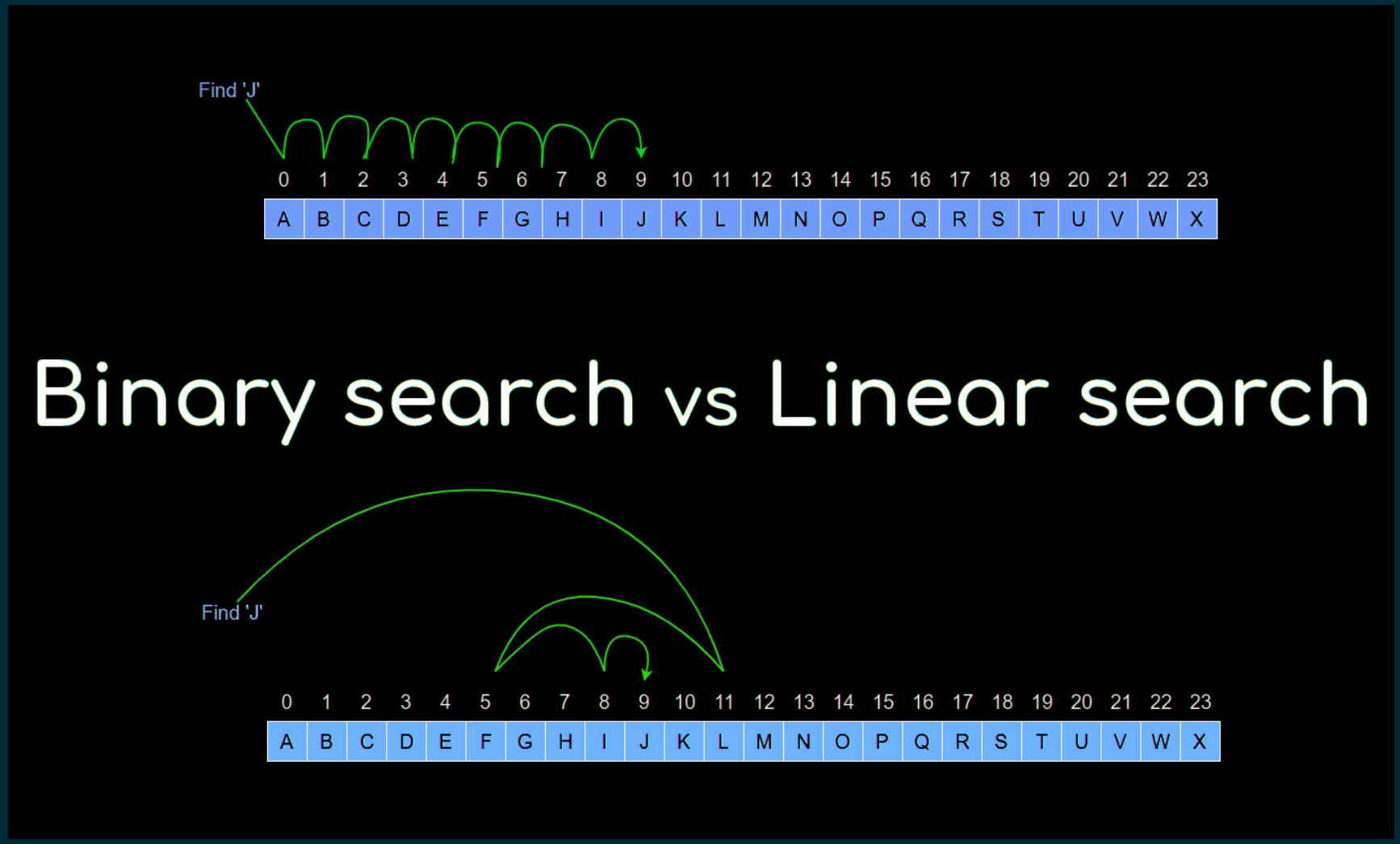
Even quicker when A is already half-sorted

Python runs `powersort()`, an optimised version of TimSort

```
1  a = [11, 6, 8, 2, 22, 16, 25]
2
3  a.sort()
```

github.com/ale66/learn-coding

SEARCHING



github.com/ale66/learn-coding

Istance:

- a collection of *n* integers $A = a_1, a_2, \ldots a_n$
- an integer k

Question: does k belong to the collection?

OBSERVATIONS

Strings and texts are searchable: alphanumerical ordering

The textbook algo. takes up to n comparisons

Search is everywhere: implementation must be

- quick, and
- reliable, hands negatives, 0s, `NaN`s, `None`s

CASE STUDY: EGO NETWORKS [FERRARA ET AL., 2011]

- while less than N profiles collected
  - generate random FB ids (a fixed-lenght, 32-digits integer)
  - test the random id: does it land on an open FB profile?
    - yes: expand the visit to the neighborood
    - no: go back to generating random ids

Cost: at least N generations+FB accesses; dN comparisons to check for duplicate ids

VANILLA SOLUTION: LINEAR SEARCH

```
 1  def linSearch(data_list, search_key):
 2    '''Linear search'''
 3
 4    found = False
 5
 6    n = len(data_list)
 7
 8    for elem in data_list:
 9
10        if elem == search_key:
11            found = True
12
13    return found
```

EXERCISE

Apply `while` instead of `for` to stop operations as soon as the key value is found

Return the position at which the key was found

Model call:

```
my_data = [11, 6, 8, 2, 22, 16, 25]

my_key = 42

outcome = yourSolution(my_data, my_key)
```

ORDERED SEQUENCES

Special case: the input sequence is already sorted, either in increasing or decreasing order

a *much more efficient* algorithm is available: Binary Search

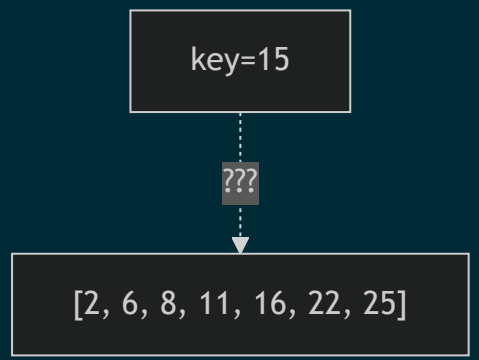
- only works for sorted sequences
- top speed: *at most* $\log_2 n$ comparisons

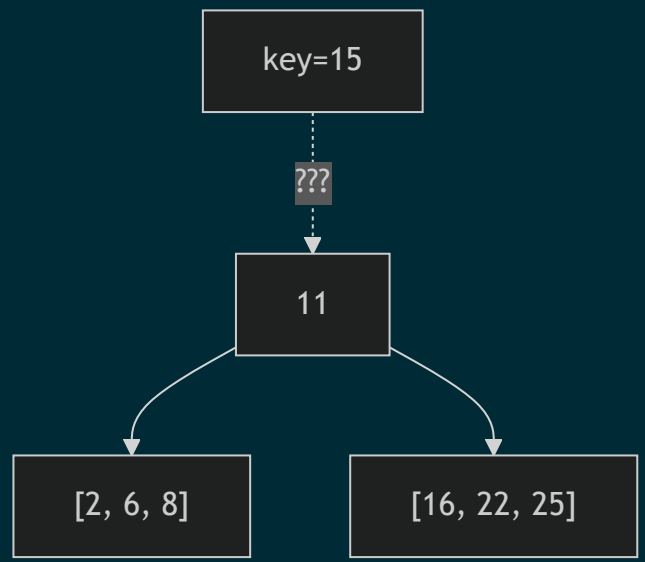
items	Vanilla (linear)	BS
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

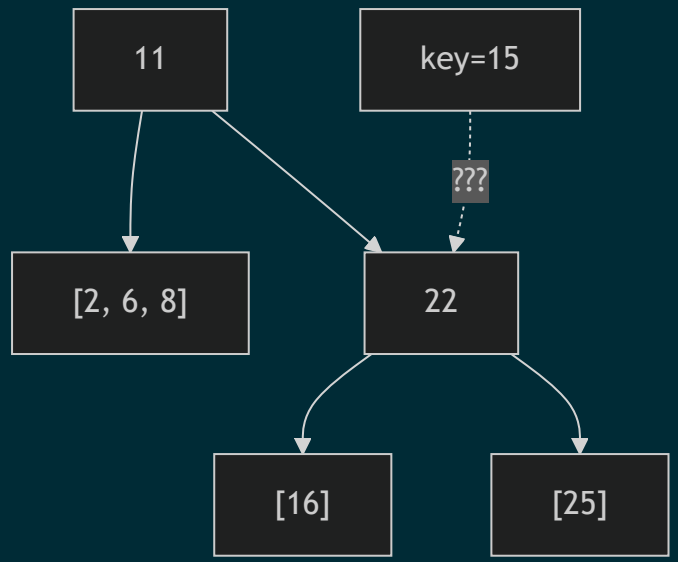
IDEA

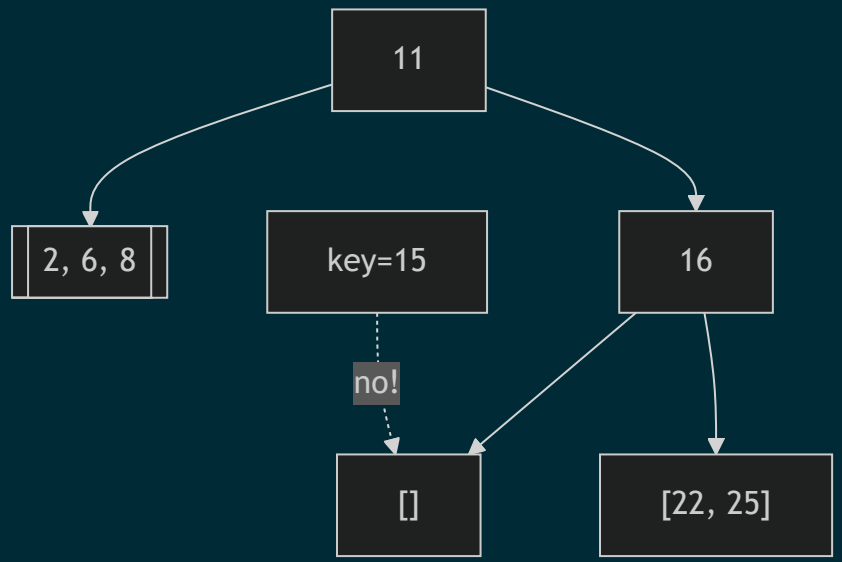
Data are sorted: exploit this property to cut down the size of the list *segment* to be checked

```
1  my_sorted_list = [2, 6, 8, 11, 16, 22, 25]
2
3  key = 15
```






11
key=15
???
[2, 6, 8]
22
[16]
[25]


11
2, 6, 8
key=15
16
no!
[]
[22, 25]

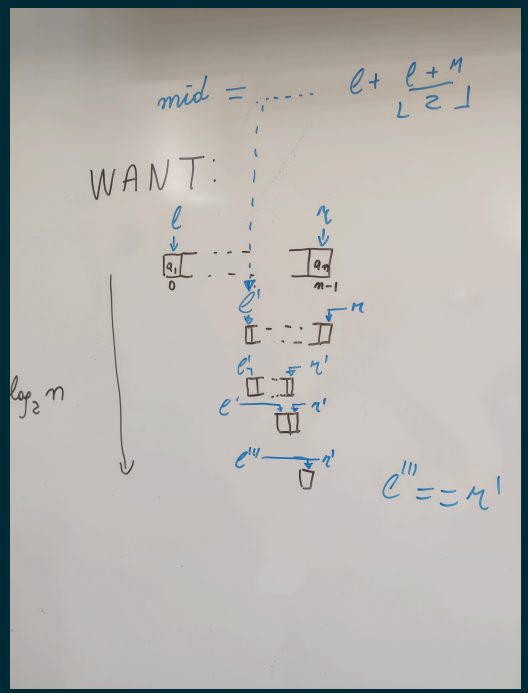
STEPS

- input the ordered list and the key value to be searched
- find the *median* value (here, it's right in the middle!)
- if the median $==$ the key value then stop and say 'yes'
- but if the search key $>$ median then the value, if it exists, can only be in second half of the list
- otherwise, the key value, if it exists, can only be in the first half of the list
- depending on the result of the comparison, continue searching on the 'right' half of the list.

BS halves the searched data at each iteration

Soon, the halving will shrink the list down to just one value, so we check and finish

How soon? It will take at most $\log_2 n$ 'cuts' to shrink the list down to 1

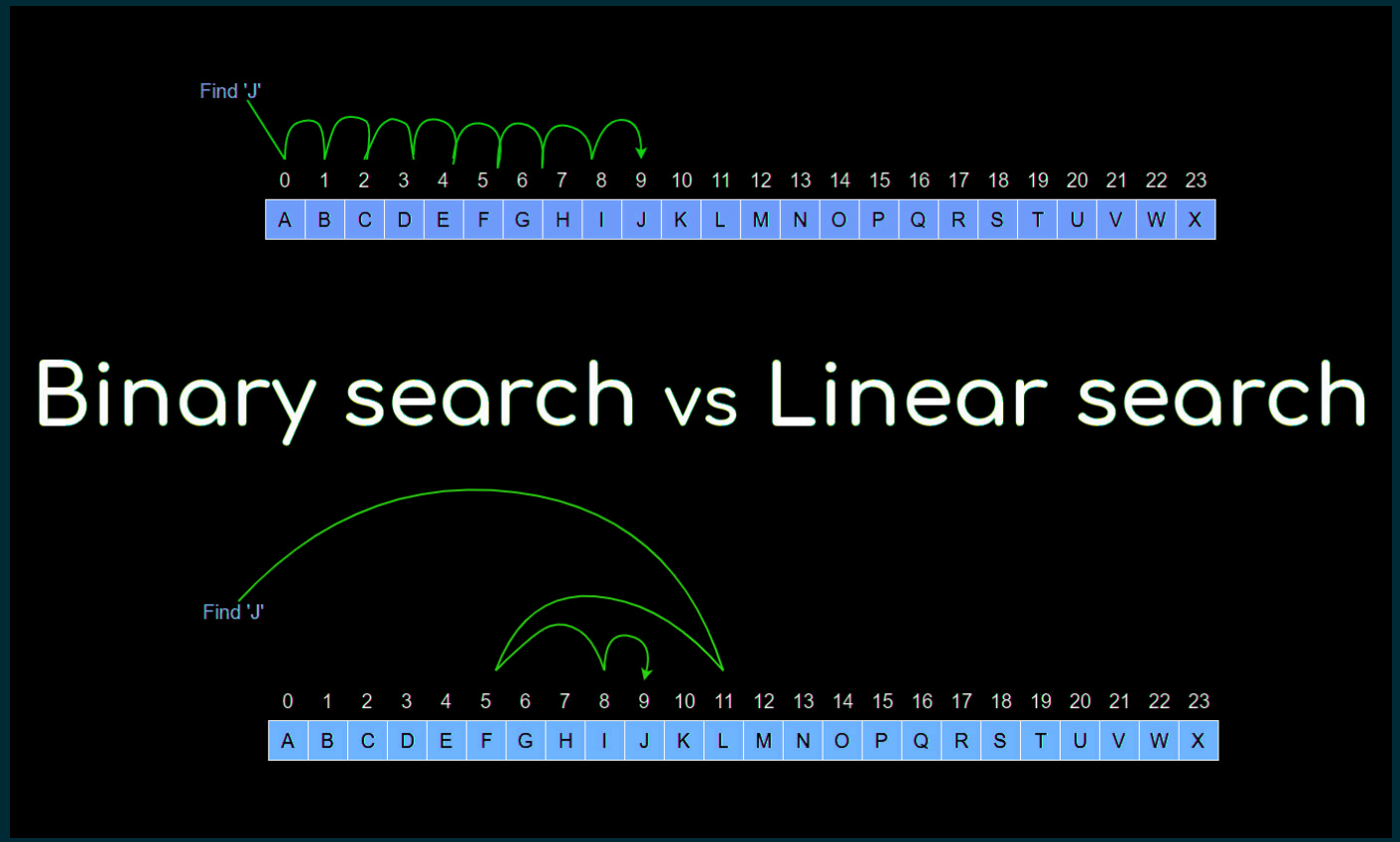


```python
def bs(a, k):
    '''Simple Binary search implementation: a is a list of integers, k is a search

    found = False
    n = len(a)

    # the boundaries of our search
    l = 0
    r = n
```

```python
    while ((found == False) and (l < r)):

        mid = l + int((r-l)/2)

        if k < a[mid]:
            r = mid

        elif k > a[mid]:
            l = mid + 1

        else:
            found = True
            print('found in position ', mid)

    return found
```

VISUALISATION

Instance: an ordered list of 24 uppercase chars, `key = 'J'`



$$\lceil \log_2 24 \rceil = \lceil 4.5849 \rceil = 5 \text{ comparisons}$$