CS303 - Algorithms and Data Structures

Lecture 9

- REVIEW-

Professor : Mahmut Unan – UAB CS

Agenda

- REVIEW

- **Exam 1:** 02/09/2023 Thursday,
- Lecture Time

- Topics
  - Sorting
  -
  - Data Structures

Sorting Problem

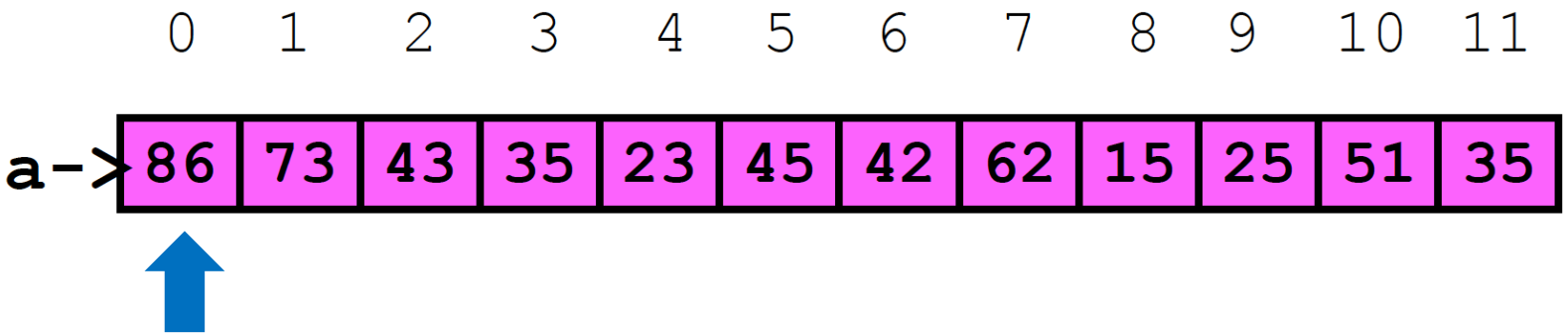
- **Input**: A sequence of number $<a_1, a_2, \ldots, a_n>$ (also commonly referred to as ***keys***)
- **Output**: A sequence of number $<a'_1, a'_2, \ldots, a'_n>$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

- In this class we will consider different algorithms for solving the sorting problem
- We will assume that the input sequence is an array of N elements

Linear Search

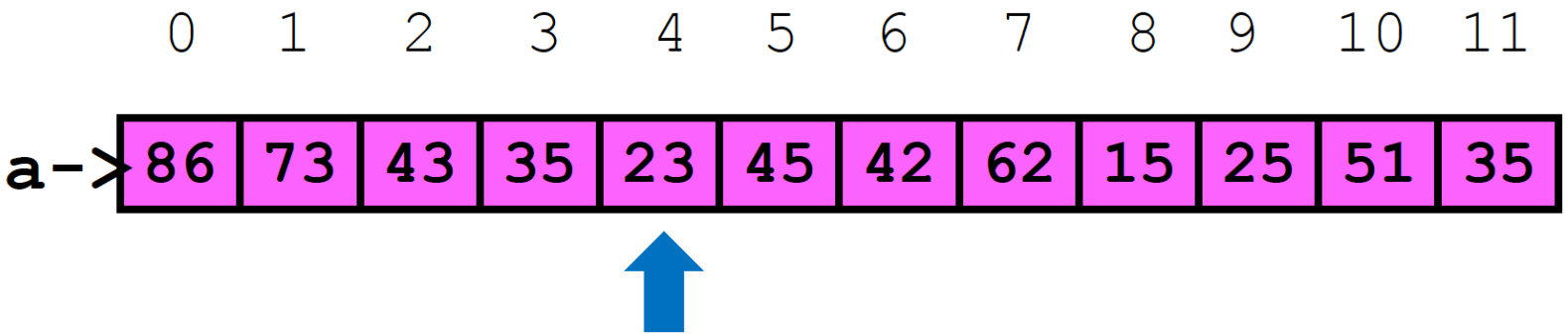
- Start from the first element of the array
- Compare each element of the array one by one
- If you find the element, return the index
- If you can not find it, return -1

Sequential (Linear) Search



x → 23

Sequential (Linear) Search



x → 23

How many comparisons?

Binary Search

Key Idea: Repeated Halving

- To find the phone number of James Harden…
- B = phone book
- While B is larger than one page:
    1. p = page in the middle
    2. q = name on first name on p
    3. if ‘Harden’ comes before q:

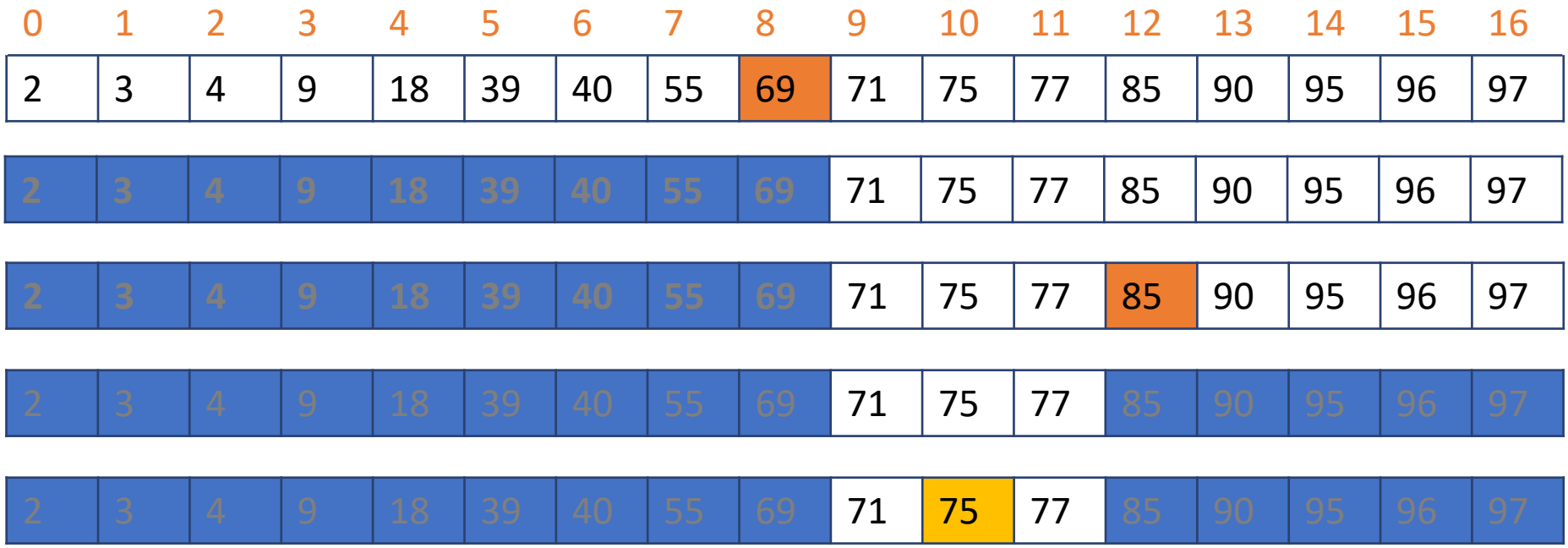
    Rip away second half of B

    else:

    Rip away first half of B
- Scan p line by line for ‘James Harden’

Binary Search – Recursive Calls

Suppose we search for target 75



Algorithm:

1. Let mid be the middle index of list lst
2. Compare target with lst[mid]
   - If target > lst[mid] continue search of target in sublist lst[mid+1:]
   - If target < lst[mid] continue search of target in sublist lst[? : ?]
   - If target == lst[mid] return mid

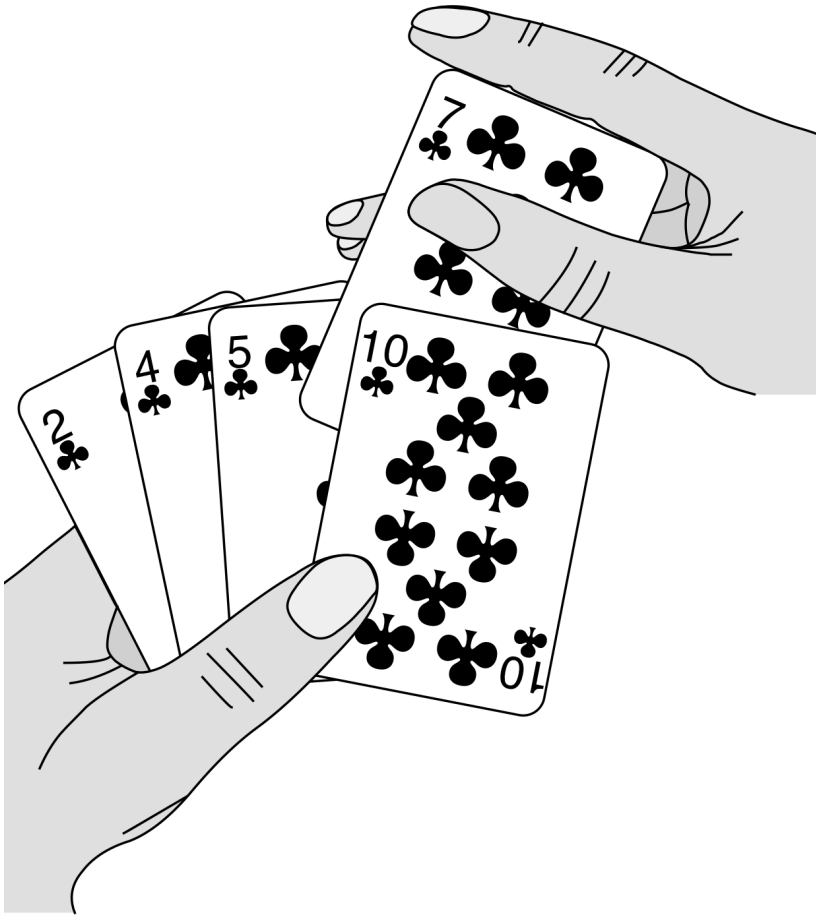
This is a recursive algorithm.
We're calling the same process on sublists of the original list

Binary Search

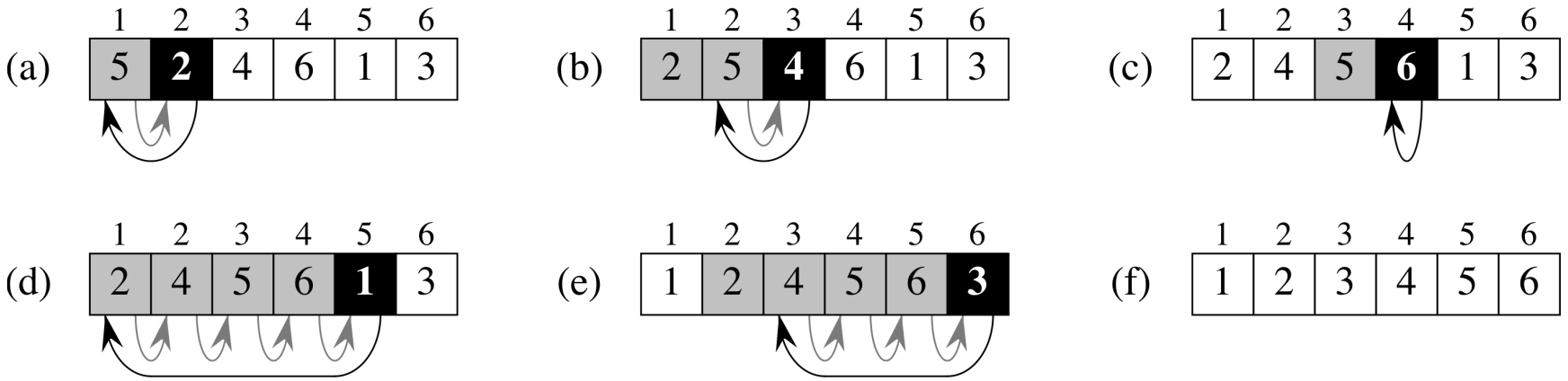
- The concept of repeated halving the size of the “search space” is the main idea of binary search.
- An item in a sorted collection of items can be found in approximately $\log_2 n$ comparisons.

Insertion Sort – General Approach

- Basic idea is similar to sorting a hand of playing cards
- Start with an empty left hand and cards face down on the table
- Remove one card at a time from the table and insert the card at the correct position in the left hand (till there are no cards left on the table)
- To find the correct position compare the card in right hand with the cards in the left hand from right to left



Insertion Sort – Example



Insertion Sort – Algorithm

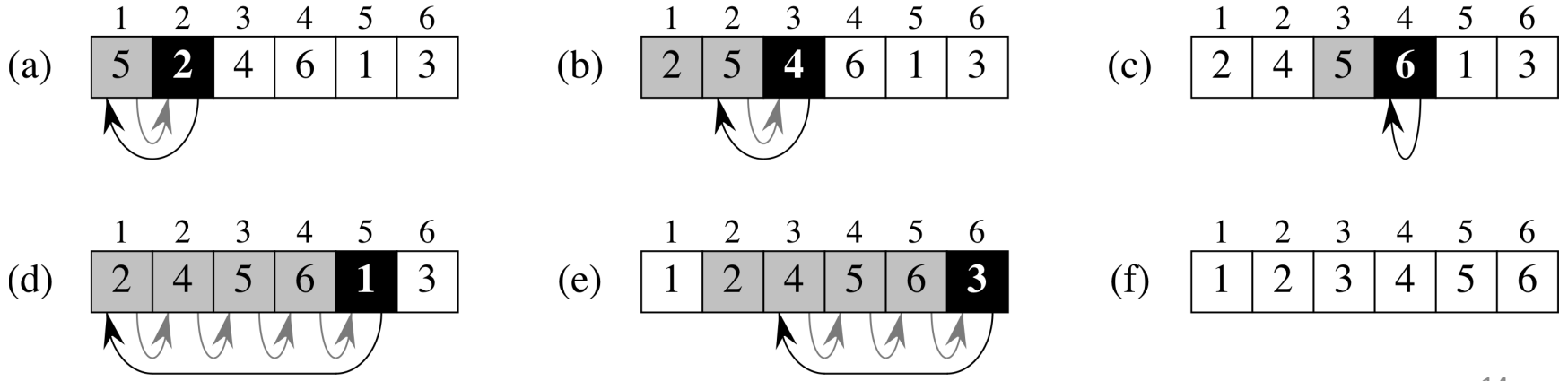
```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

****In pseudocode array indexing starts at **1** NOT **0****

Insertion Sort – Tracing

INSERTION-SORT(A, n)

```
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```



Insertion Sort – Observations

- This algorithms sorts the input values ***in place***, *i.e.,* it rearranges the values using the input array A by using a constant number of temporary locations (the input array is replaced with sorted values – no extra memory required)
- At any step *j* in the iteration elements *A[1..j-1]* represent the first *j* elements in the original array *A* that are in the sorted order

Algorithm Analysis

- Analyzing algorithms – predicting resources required by the algorithm
- Resources: **memory**, **computational time**, bandwidth, latency, etc.
- Computational time is often dependent on the problem input size
- Also for the same input size the computational time could vary depending on the actual input values
- Since computational time is machine-dependent we use the number of primitive operations or “steps” executed by an algorithm (***running time***) to compare performance of different algorithms

Insertion Sort – Complexity Analysis

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

t_j = number of times the **while** loop test is executed for that value of j

Best-case running time: when the array is already sorted

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

$t_j = 1$, for j = 2, 3, 4, …. n

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

T(n) = an + b, where a and b are constants

Worst-case running time: when the array is in reverse sorted order

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

$t_j = j$, for $j = 2, 3, 4, \ldots n$

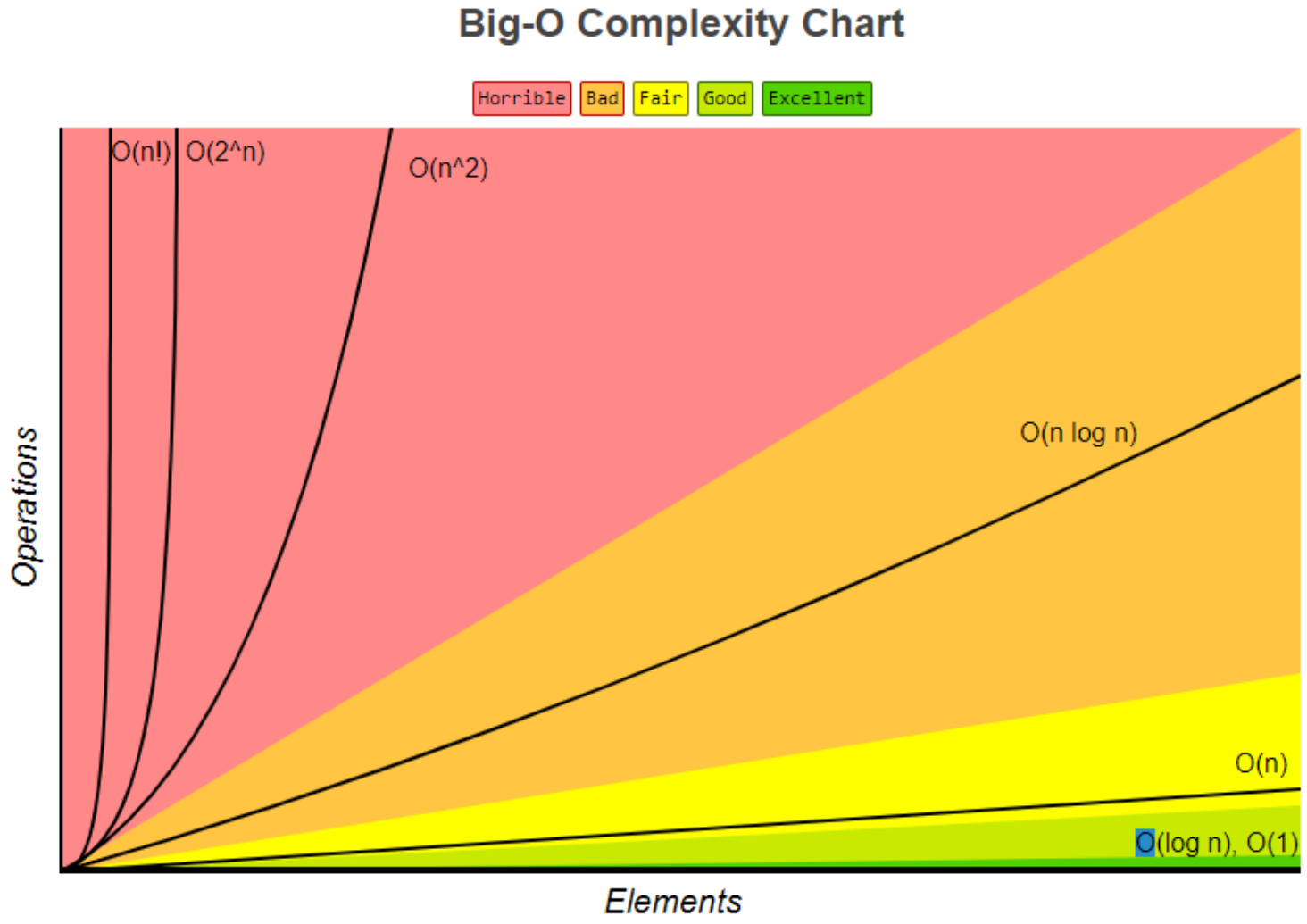
$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

$T(n) = an^2 + bn + c$, where a, b, and c are constants

Insertion Sort – Complexity Analysis

- Best case
  - when the array is already sorted
  - $T(n) = an + b$, where a and b are constants
  - provides the lower bound on the running time
- Worst case
  - when the array is in reverse sorted order
  - $T(n) = an^2 + bn + c$, where a, b, and c are constants
  - provides the upper bound on the running time
- Average case
  - assumes half of the elements needs to be rearranged
  - $T(n) = an^2 + bn + c$, where a, b, and c are constants
  - often as bad as the worst case

Big 0 Complexity Chart



Ref: http://bigocheatsheet.com/

Divide-and-conquer approach

Merge Sort

Recall - Recursion

- What is recursion ?
  - a recursive function calls itself one or more times in its body.
- Example?
  - Factorial function

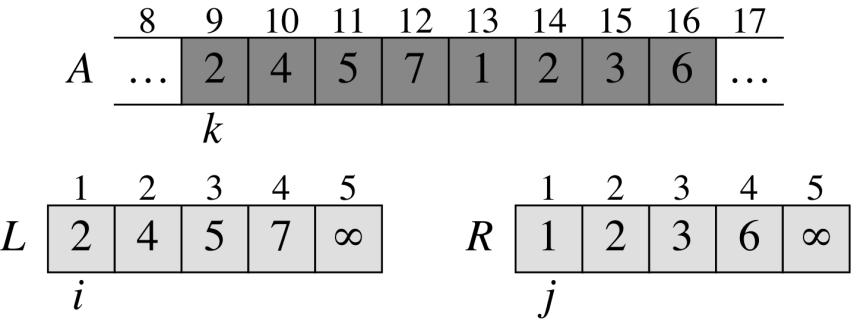
```
def factorial(n):
    if n == 1:
        return n
    else:
        return n*factorial(n-1)
```

Divide-and-conquer paradigm

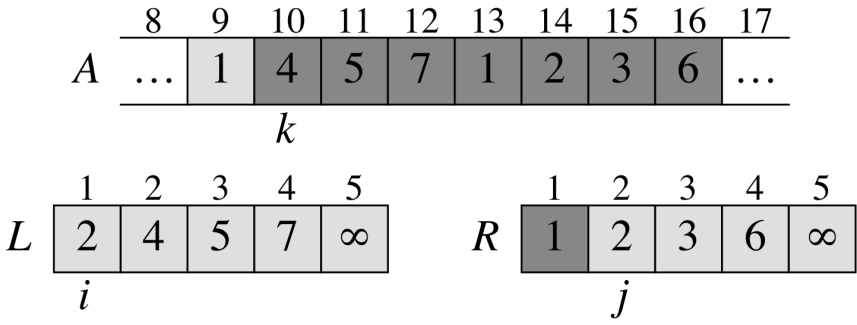
- **Divide** the problem into a number of sub-problems that are smaller instances of the same problem
- **Conquer** the sub-problems by solving them recursively
- **Combine** the solutions to the sub-problems into the solution for the original problem

Merge Sort

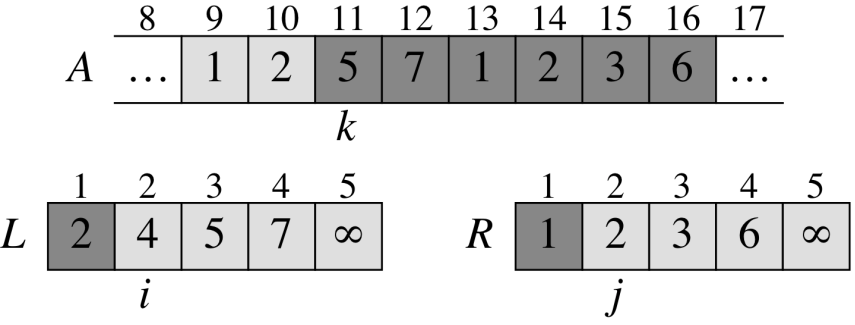
- **Divide** the N-element sequence to be sorted into two subsequences of N/2 elements each
- **Sort** the two subsequences recursively using merge sort
- **Merge** the two sorted subsequences to produce the sorted answer


8 9 10 11 12 13 14 15 16 17
A … 2 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

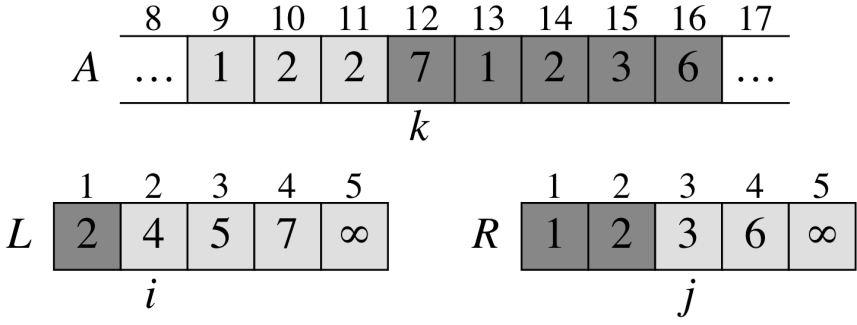
(a)


8 9 10 11 12 13 14 15 16 17
A … 1 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

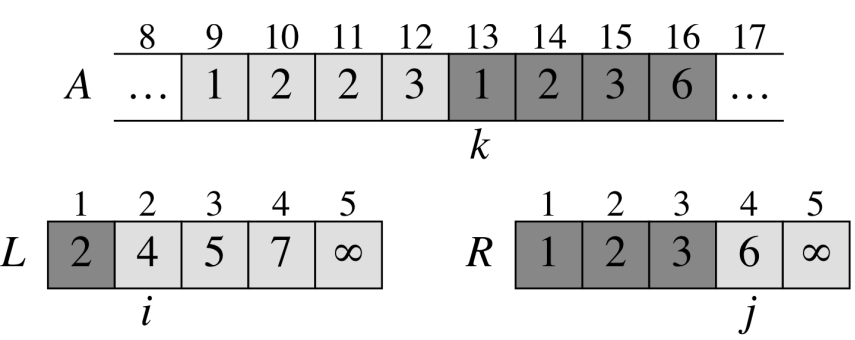
(b)


8 9 10 11 12 13 14 15 16 17
A … 1 2 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

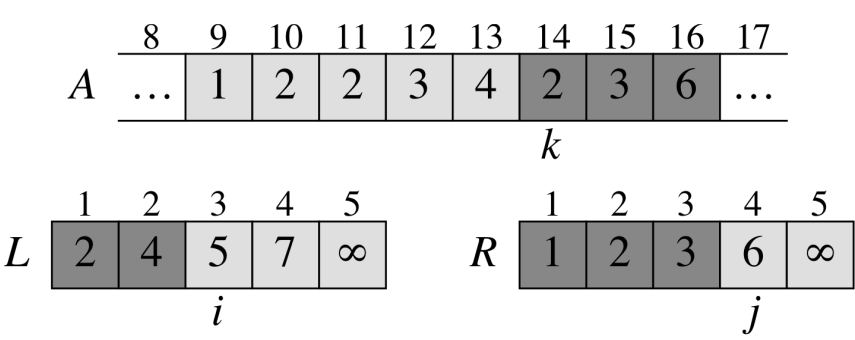
(c)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

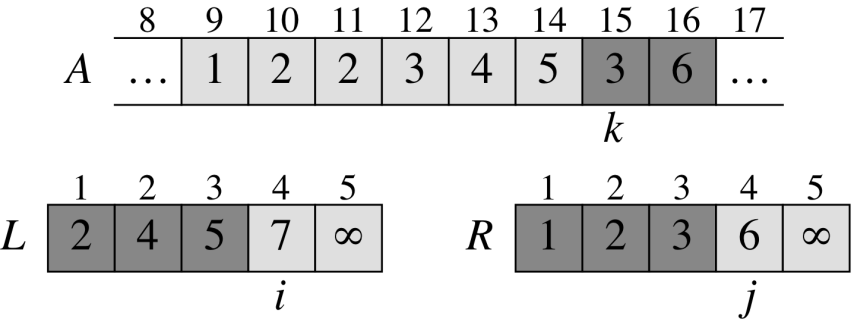
(d)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

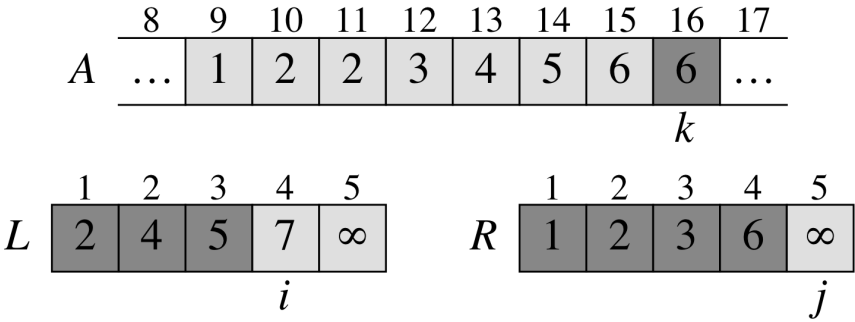
(e)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

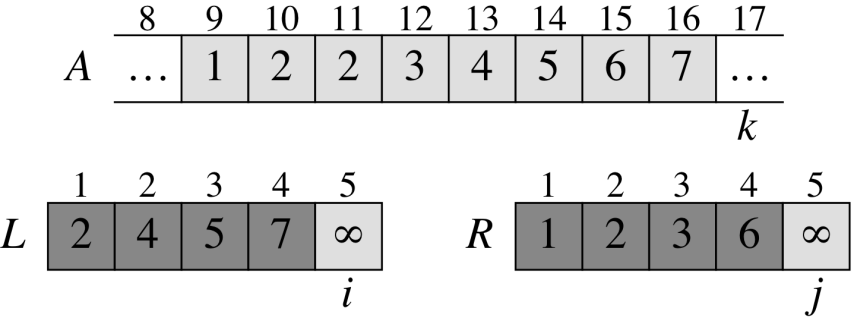
(f)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

Pseudo Code (from textbook)

```
MERGE-SORT(A, p, r)
  if p < r                                  // check for base case
      q = ⌊(p + r)/2⌋                       // divide
      MERGE-SORT(A, p, q)                   // conquer
      MERGE-SORT(A, q + 1, r)               // conquer
      MERGE(A, p, q, r)                     // combine
```

```
MERGE(A, p, q, r)
  n_1 = q − p + 1
  n_2 = r − q
  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
  for i = 1 to n_1
      L[i] = A[p + i − 1]
  for j = 1 to n_2
      R[j] = A[q + j]
  L[n_1 + 1] = ∞
  R[n_2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

Pseudo Code (revised)

```
MERGE-SORT (A, temp, p , r)
    if p < r
        q = |_ (p + r) / 2 _|
        MERGE-SORT (A, temp, p , q)
        MERGE-SORT (A, temp, q + 1, r)
        MERGE (A, temp, p, q, r)
```

```
MERGE (A, temp, p, q, r)
// merge A[p..q] with A[q+1..r]
        i = p
        j = q + 1

        // copy A[p..r] to temp[p..r]
        for k = p to r
                temp[k] = A[k]

        //merge back to A[p..r]
        for k = p to r
                if i > q            // left half empty, copy from the right
                        A[k] = temp[j]
                        j = j + 1
                else if j > r       // right half empty, copy from the left
                        A[k] = temp[i]
                        i = i + 1
                else if temp[j] < temp[i]         // copy from the right
                        A[k] = temp[j]
                        j = j + 1
                else
                        A[k] = temp[i]            // copy from the left
                        i = i + 1
```

Example 3

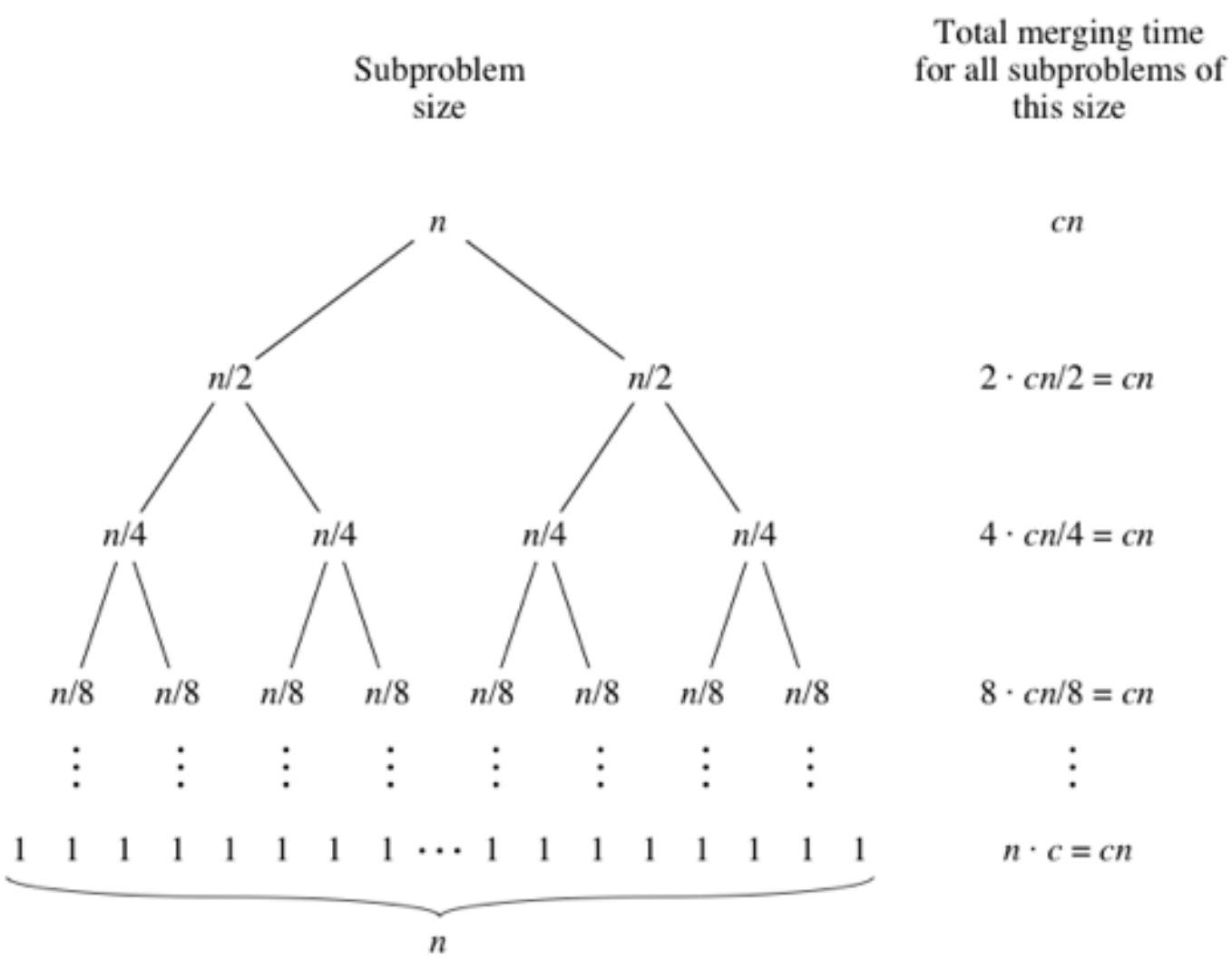
- Arr = [90,75,82,35,22,1]

How about non recursive merge sort?

Merge Sort – Time Complexity

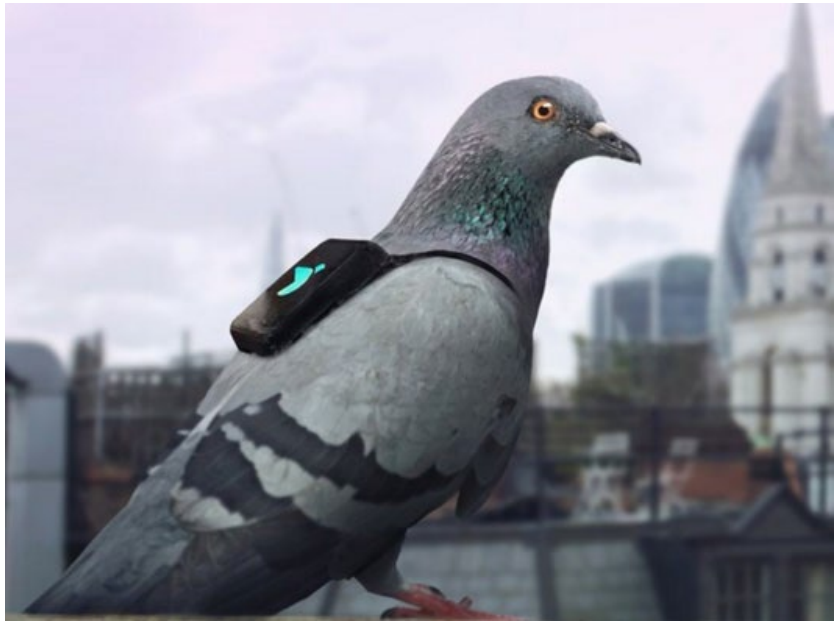
- Merge Sort is a recursive algorithm

- Worst case
- Average Case
- Best Case


Subproblem size
Total merging time for all subproblems of this size
n
cn
n/2
n/2
2 · cn/2 = cn
n/4
n/4
n/4
n/4
4 · cn/4 = cn
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
8 · cn/8 = cn
1 1 1 1 1 1 1 1 ··· 1 1 1 1 1 1 1 1
n · c = cn
n

- **Worst case** O (n logn)
- **Average Case** O (n logn)
- **Best Case** O (n logn)

Why Time Complexity



ADSL vs Pigeon
4GB data
60 mile
Which one to choose?

The RAM Model of Computation

- Random Access Machine
  - Each simple operation (+ - * / + if …) takes exactly one time step
  - Loops are not simple operations
  - Each memory access takes one step
- We will calculate best, worse and average case complexities

If-Then-Else

```
if (cond) then
   block 1 (sequence of statements)
else
   block 2 (sequence of statements)
end if;
```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time is the slower of the two possibilities:

$$\max(\text{time(block 1)}, \text{time(block 2)})$$

If block 1 takes O(1) and block 2 takes O(N), the if-then-else statement would be O(N).

Statements with function/ procedure calls

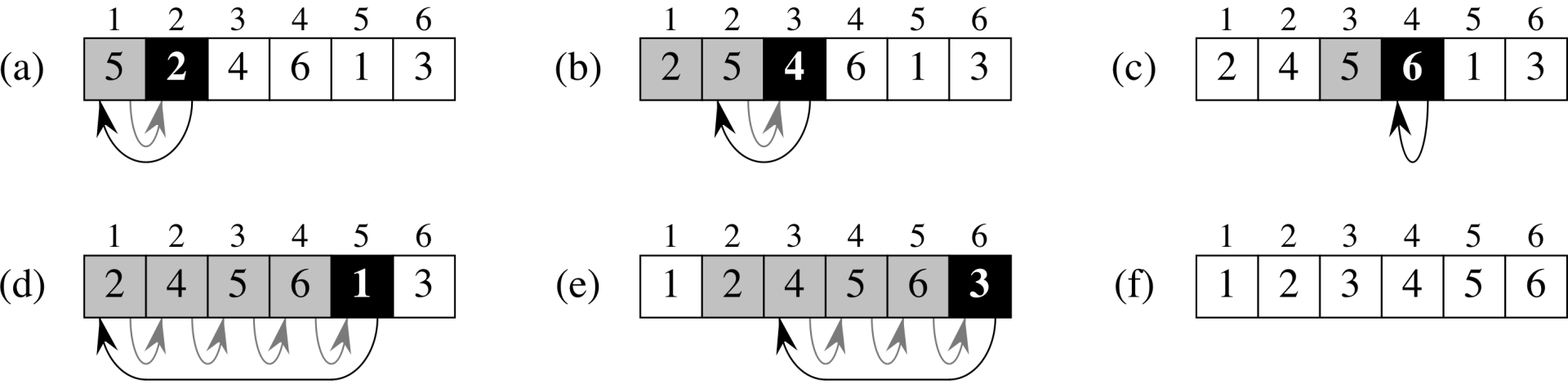
When a statement involves a function/ procedure call, the complexity of the statement includes the complexity of the function/ procedure. Assume that you know that function/ procedure *f* takes constant time, and that function/procedure *g* takes time proportional to (linear in) the value of its parameter *k*. Then the statements below have the time complexities indicated.

f(k) has O(1)
g(k) has O(k)

When a loop is involved, the same rule applies. For example:

```
for J in 1 .. N loop
    g(J);
end loop;
```

Insertion Sort Analysis



INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Best case – when array is already sorted, $t_j = 1$, for j=2..n

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Worst case – when array is in reverse order, $t_j = j$, for j=2..n

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Order of Growth

- It is how the time of execution depends on the length of the input.

Big – O Notation

- Big O notation is used in Computer Science to describe the performance or complexity of an algorithm.
- Big O specifically describes the worst-case scenario, and can be used to describe the execution time required

O-notation:

To denote asymptotic upper bound, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n") the set of functions:

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0 \}$

Ω-notation:

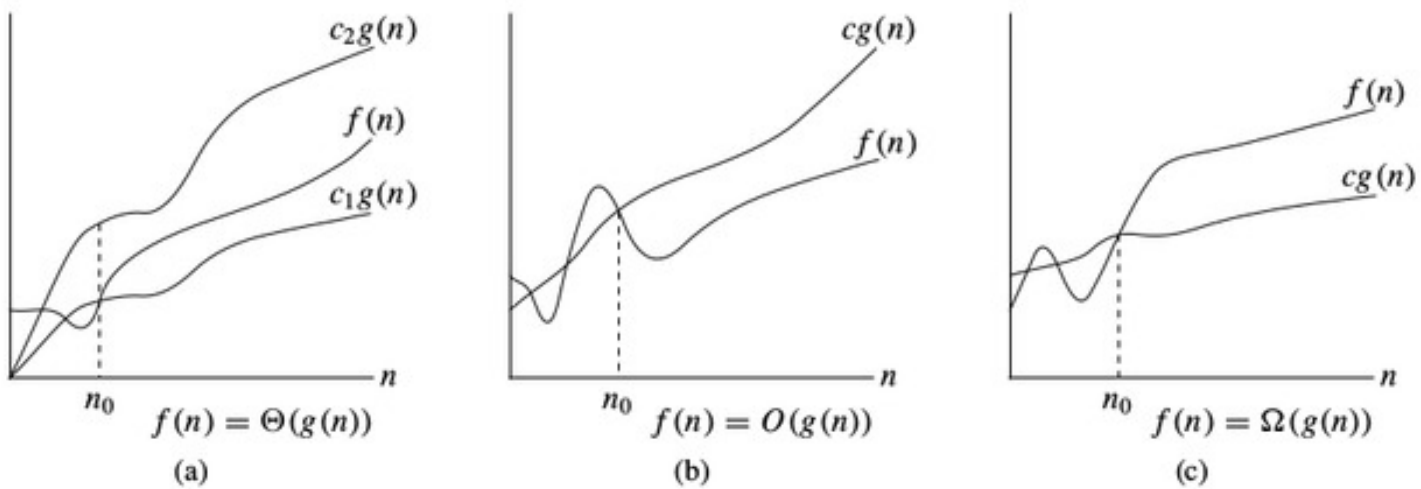
To denote asymptotic lower bound, we use Ω-notation. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n") the set of functions:

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq c * g(n) \leq f(n)$ for all $n \geq n_0 \}$

Θ-notation:

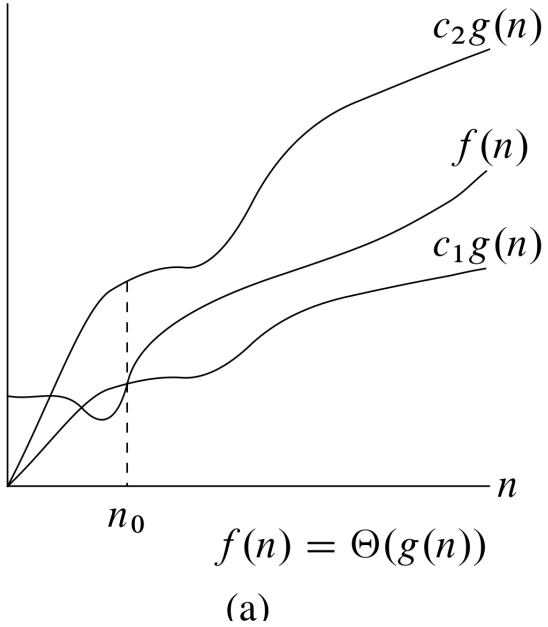
To denote asymptotic tight bound, we use Θ-notation. For a given function $g(n)$, we denote by $\Theta(g(n))$ (pronounced "big-theta of g of n") the set of functions:

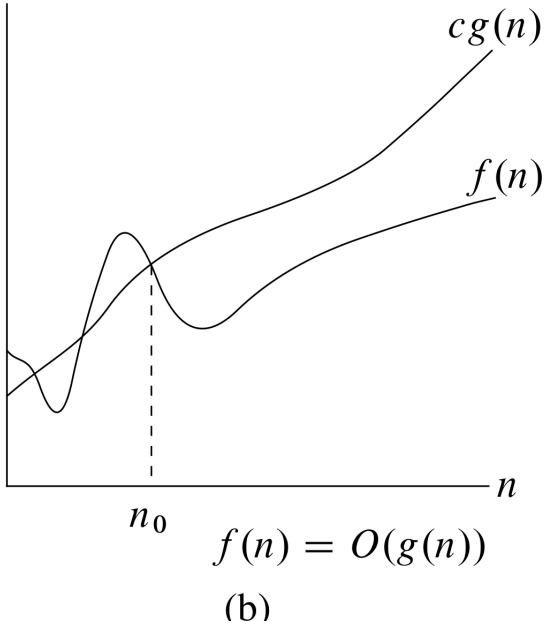
$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n > n_0 \}$

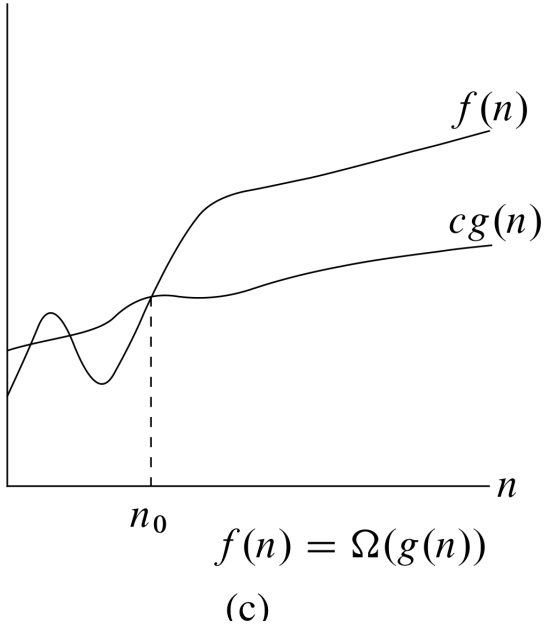


https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/

Asymptotic Analysis







$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

```
MERGE-SORT(A, p, r)
  if p < r                                  // check for base case
      q = ⌊(p + r)/2⌋                       // divide
      MERGE-SORT(A, p, q)                   // conquer
      MERGE-SORT(A, q + 1, r)               // conquer
      MERGE(A, p, q, r)                     // combine
```

```
MERGE(A, p, q, r)
  n_1 = q − p + 1
  n_2 = r − q
  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
  for i = 1 to n_1
      L[i] = A[p + i − 1]
  for j = 1 to n_2
      R[j] = A[q + j]
  L[n_1 + 1] = ∞
  R[n_2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

Analysis of divide and conquer algorithms

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

Common Order of Growth

- O(1)
- O(N)
- O(N^2)
- O (2^N)
- O(log N)
- O (N* log N)
- ….

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

http://web.mit.edu/16.070/www/lecture/big_o.pdf

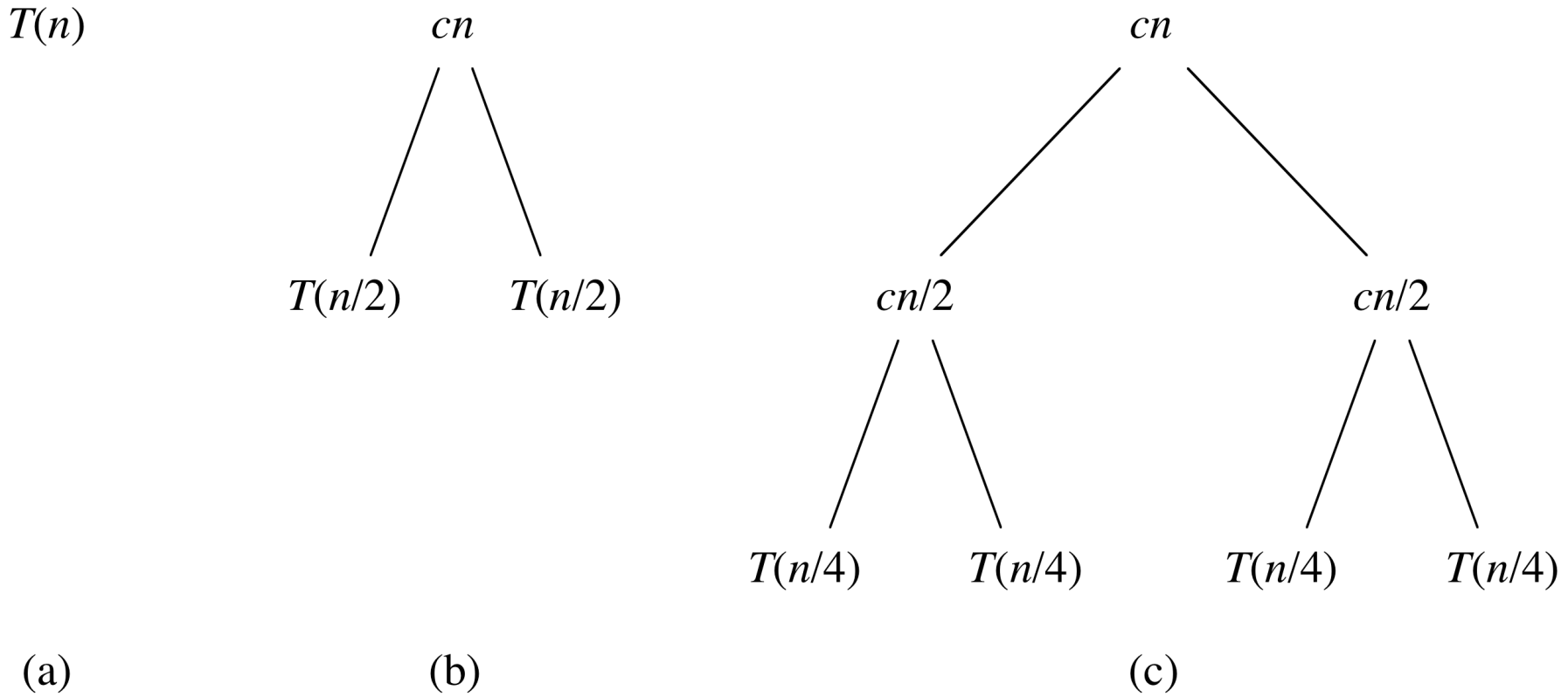
Working with the Big –O notation

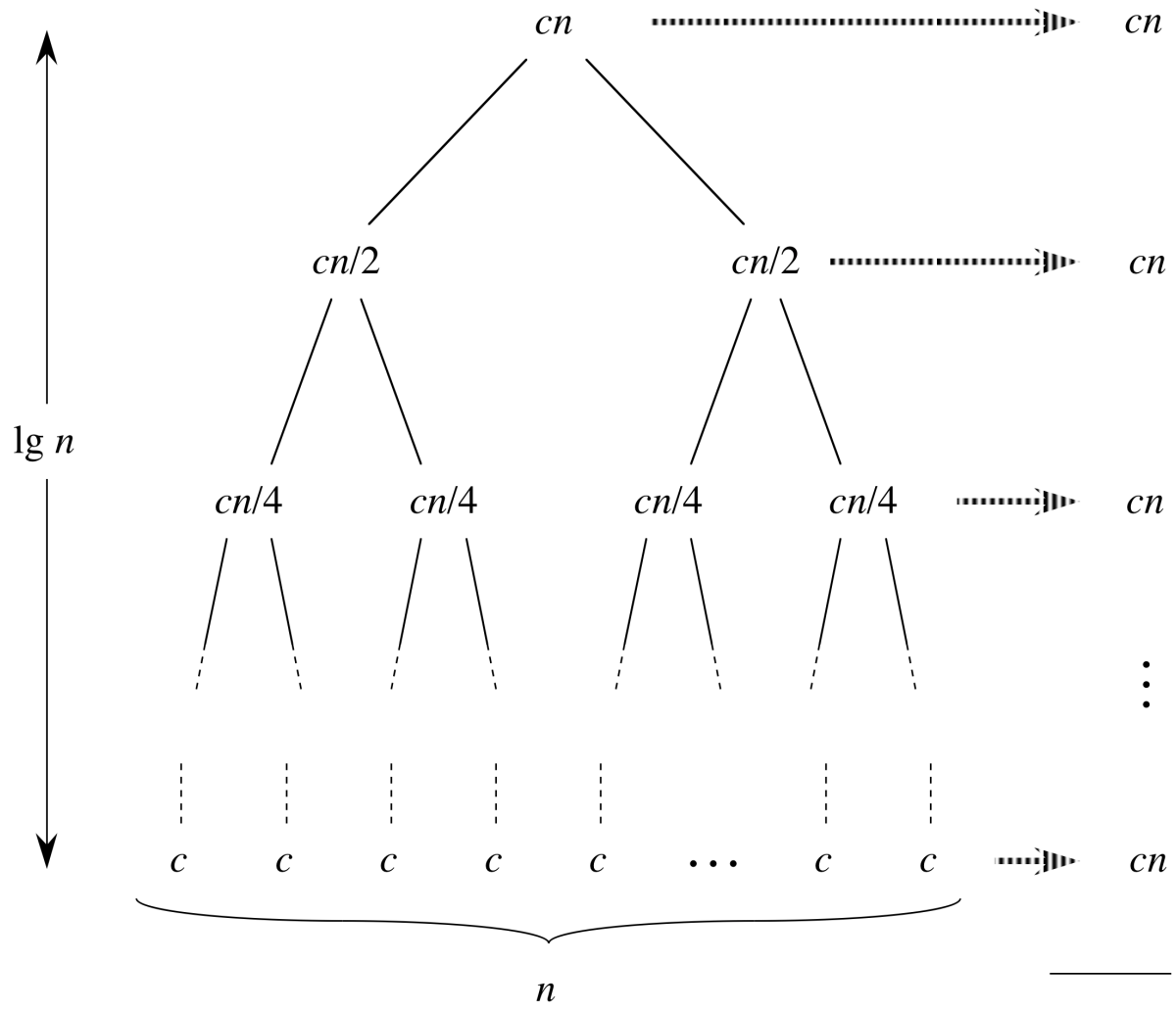
- **O (f(n)) +** O (g(n)) = O (max(f(n), g(n)))
- O(c*f(n)) =O(f(n))
- O (f(n)) * O (g(n)) = O (f(n) *g(n))
- ….

Solving Recurrences

- substitution method
- recursive-tree method
- master method

Recursion tree for solving recurrence equation *T(n) = 2T(n/2) + cn*

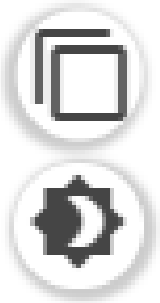




(d)

Total: $cn \lg n + cn$

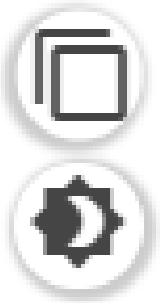
1. What is the time, space complexity of following code:



```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```


https://www.geeksforgeeks.org/practice-questions-time-complexity-analysis/

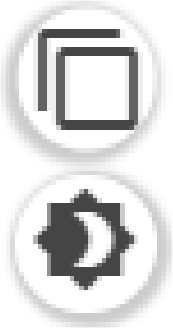
1. What is the time, space complexity of following code:



```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

O(N + M) ➔ O(n) time, O(1) space

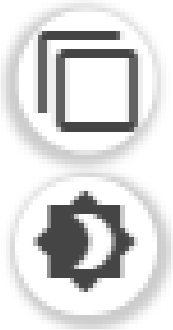
2. What is the time complexity of following code:



```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

1. O(N)
2. O(N*log(N))
3. O(N * Sqrt(N))
4. O(N*N)

2. What is the time complexity of following code:

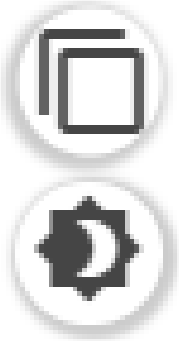


```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

1. O(N)
2. O(N*log(N))
3. O(N * Sqrt(N))
4. O(N*N)

O(n^2)

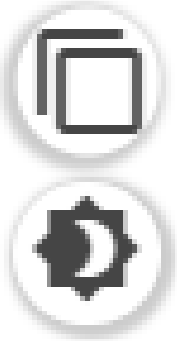
3. What is the time complexity of following code:



```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

1. O(n)
2. O(nLogn)
3. O(n^2)
4. O(n^2Logn)

3. What is the time complexity of following code:



```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

1. O(n)
2. O(nLogn)
3. O(n^2)
4. O(n^2Logn)

O(n logn)

```
int a = 0, i = N;
while (i > 0) {
        a += i;
        i /= 2;
}
```

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. O(log N)

```
int a = 0, i = N;
while (i > 0) {
        a += i;
        i /= 2;
}
```

O(logn)

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. O(log N)

What are Divide and Conquer algorithms? Describe how they work. Can you give any common examples of the types of problems where this approach might be used?

What are the key advantages of Insertion Sort, Quicksort, Heapsort and Mergesort? Discuss best, average, and worst case time and memory complexity.

https://www.interviewbit.com/courses/programming/topics/time-complexity/

1) Performance of traversing given integer array with the help of for loop ..

Code:

```
int a[] = {23,45,66,67,78,90};

for(int i=0;i<a.length;i++ {
    System.out.println("  value :"+a[i]);
}
```


http://javainterviewquestionswithanswer.blogspot.com/2016/11/calculate-time-complexity-of-below-code.html

```
function getMinMaxRange( array ) {
    let max = -Infinity;
    let min = Infinity;

    for ( let value of array ) {
        if ( value > max ) max = value;
    }
    for ( let value of array ) {
        if ( value < min ) min = value;
    }

    return max - min;
}
```


http://www.zsoltnagy.eu/big-o-notation-for-beginner-tech-interview-candidates/

```
for ( let i = 0; i < array.length; ++i ) {
    for ( let j = i; j < array.length; ++j ) {
        for ( let k = j; k < array.length; ++k ) {
            for ( let l = k; l < array.length; ++l ) {
                // O(N**4) algorithm
            }
        }
    }
}
```


http://www.zsoltnagy.eu/big-o-notation-for-beginner-tech-interview-candidates/

Trees

- **General tree** – hierarchical data structure
  - containing k ≥ 1 nodes N = {n1, n2,…, nk}
  - Connected by exactly (k – 1) links
- E= {e1, e2,…, ek-1}
  - **Root** node has no predecessor
  - **Leaves** have no successor
  - All other nodes are **internal nodes**
  - **Nodes** store information
  - Links often referred to as **edges**

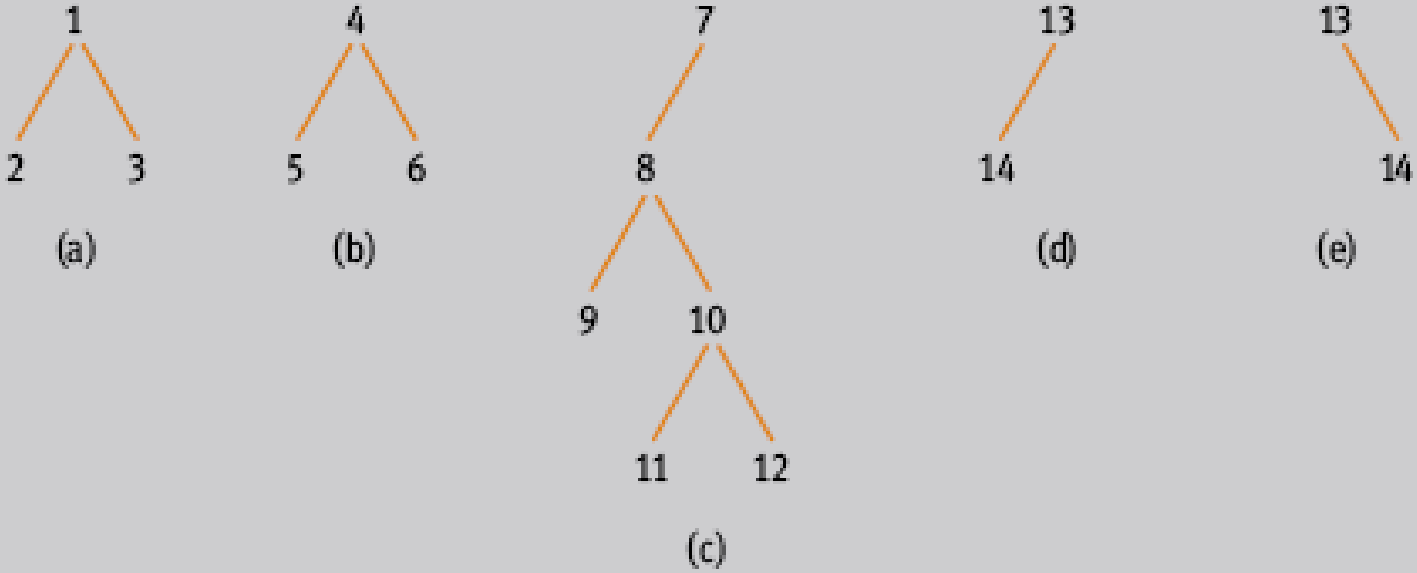
Trees (continued)

- Set of nodes, except the root, can be partitioned into zero or more disjoint subsets
  - Each partition is a **subtree**
  - Partitioning property holds for all subtrees
- Number of successors of a node is the **degree**
  - **Parent** – predecessor of a node
  - **Child** – successor of a node
  - Nodes with same parent are **siblings**

Binary Trees

- Binary trees are distinct from general trees
  - Binary trees can be empty
  - Degree no greater than two
- **Ordered trees:** every node is explicitly identifies as being either the left child or the right child of its parent
- **Binary tree**
  - Finite set of nodes, possibly empty
  - Consists of a root and two disjoint binary trees
  - Called left and right subtrees

Binary Trees (continued)



Heaps

- **Heap** is a binary tree satisfying two conditions:

  – **Order property** – data value in a node is no greater than data values stored in descendants

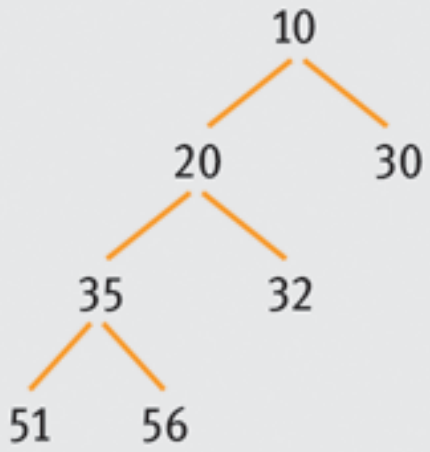
- – **Structure property** – **nearly complete binary tree (complete except last level)**
- Two important mutator methods

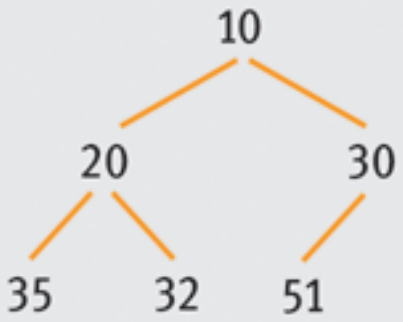
  – Insert a new value

  – Remove, return smallest/largest value (remove the root)

- Insertions must maintain order and structure properties

Heaps (continued)



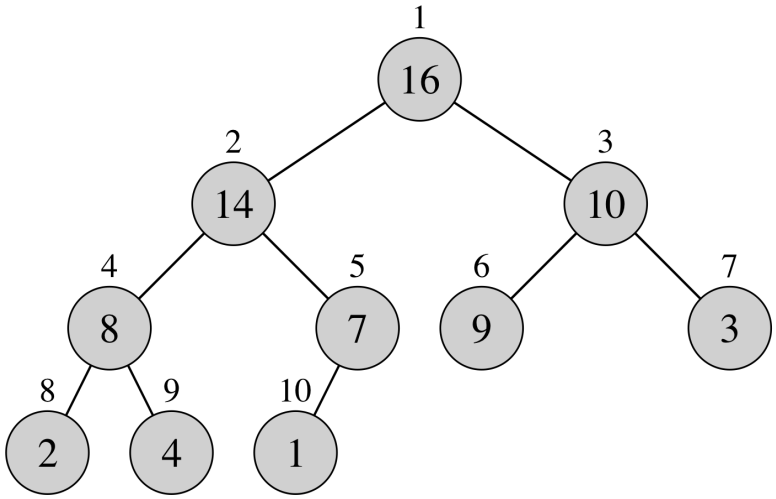


Which of the above binary trees is a complete tree or nearly complete binary tree?

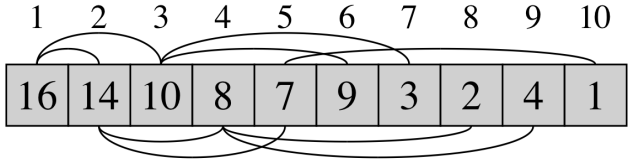
Implementation of Heaps Using One-Dimensional Arrays

- Heaps stored in one-dimensional array in strict left-to-right **level order**
- One-dimensional array representation of a heap is called a **heapform**
  - Do not need pointers
  - If node in position *i* then left child is in position 2*i*+1
- Insert a new value – place value in a unique location that maintains structure property
  - Corresponds to h[size] in heapform array

Implementation of Heaps Using One-Dimensional Arrays (continued)



(a)



(b)

PARENT(i) return |_i / 2_|
LEFT_CHILD (i) return 2i
RIGHT_CHILD(i) return 2i + 1
(remember, in pseudo code indexing starts from 1)

Implementation of Heaps Using One-Dimensional Arrays (continued)

- Heap is balanced tree by definition
  - Height is O(log *n*)
- Remove smallest/largest element in the heap
  - By definition, the root of the tree
- Replace the far-right node on lowest level *i*
  - Determine the correct location for the value moved to the root
- Maximum number of times to exchange a node with its smaller child is equal to the height

Heap sort

- Heap sort algorithm

  – Build a heap structure that contains *n* elements to be sorted

  – Remove the root, print it, and rebuild the heap

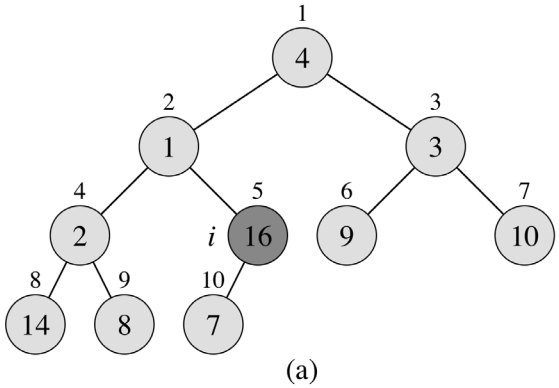
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

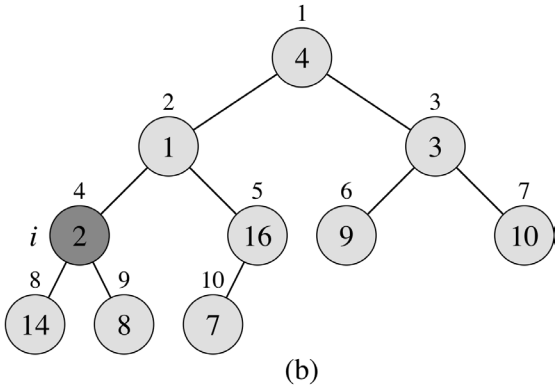
```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

BUILD-MAX-HEAP at MAX-HEAPIFY(A,5)

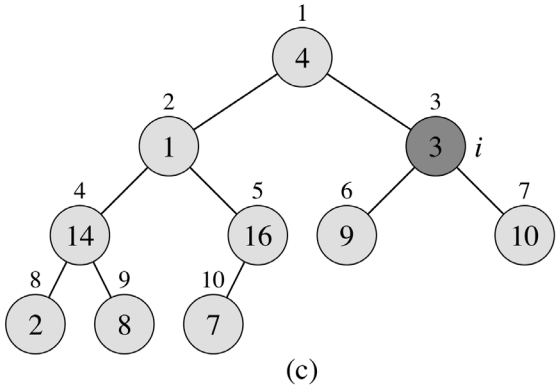
A	4	1	3	2	16	9	10	14	8	7



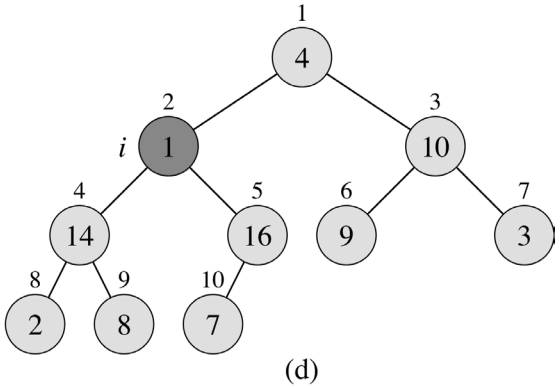
(a)



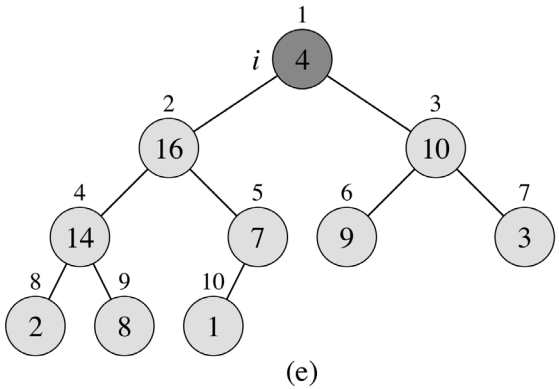
(b)



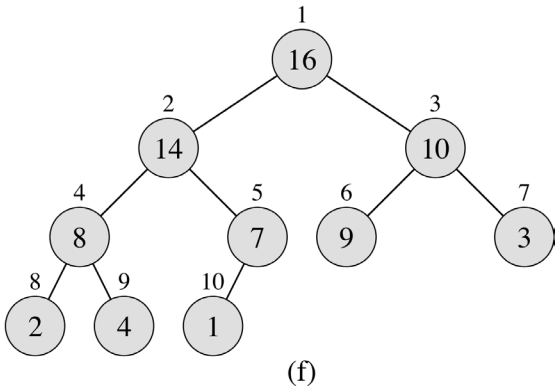
(c)



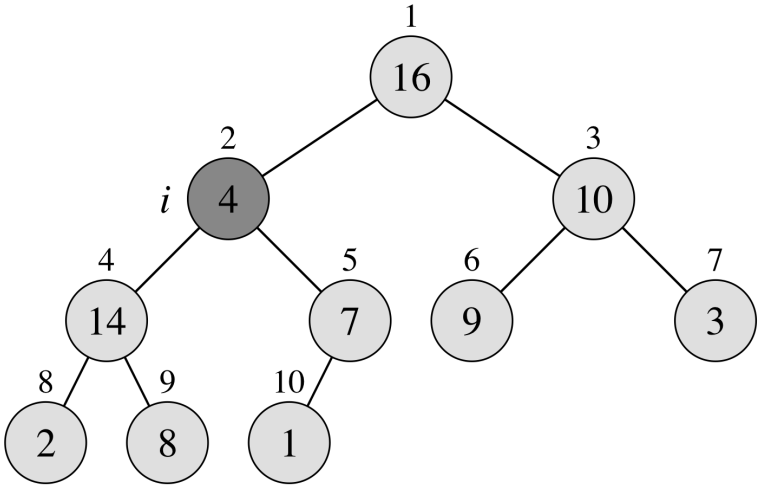
(d)



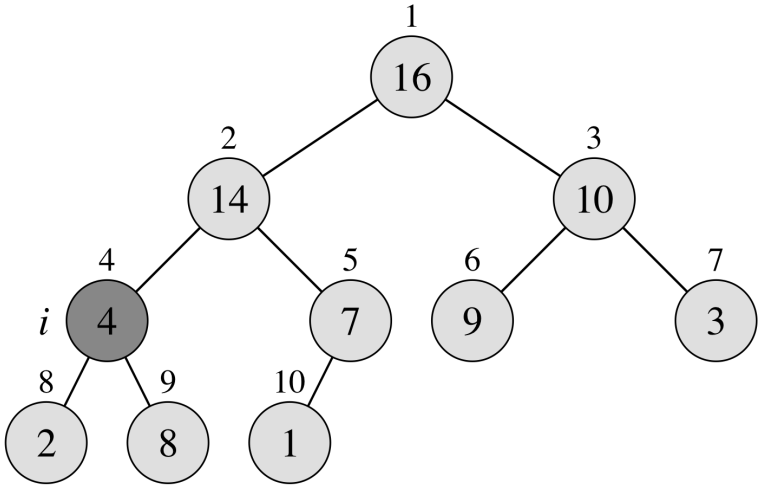
(e)



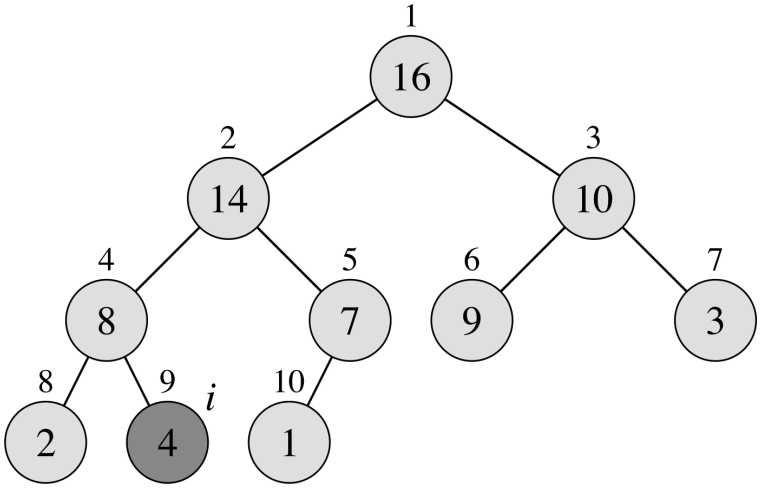
(f)



(a)

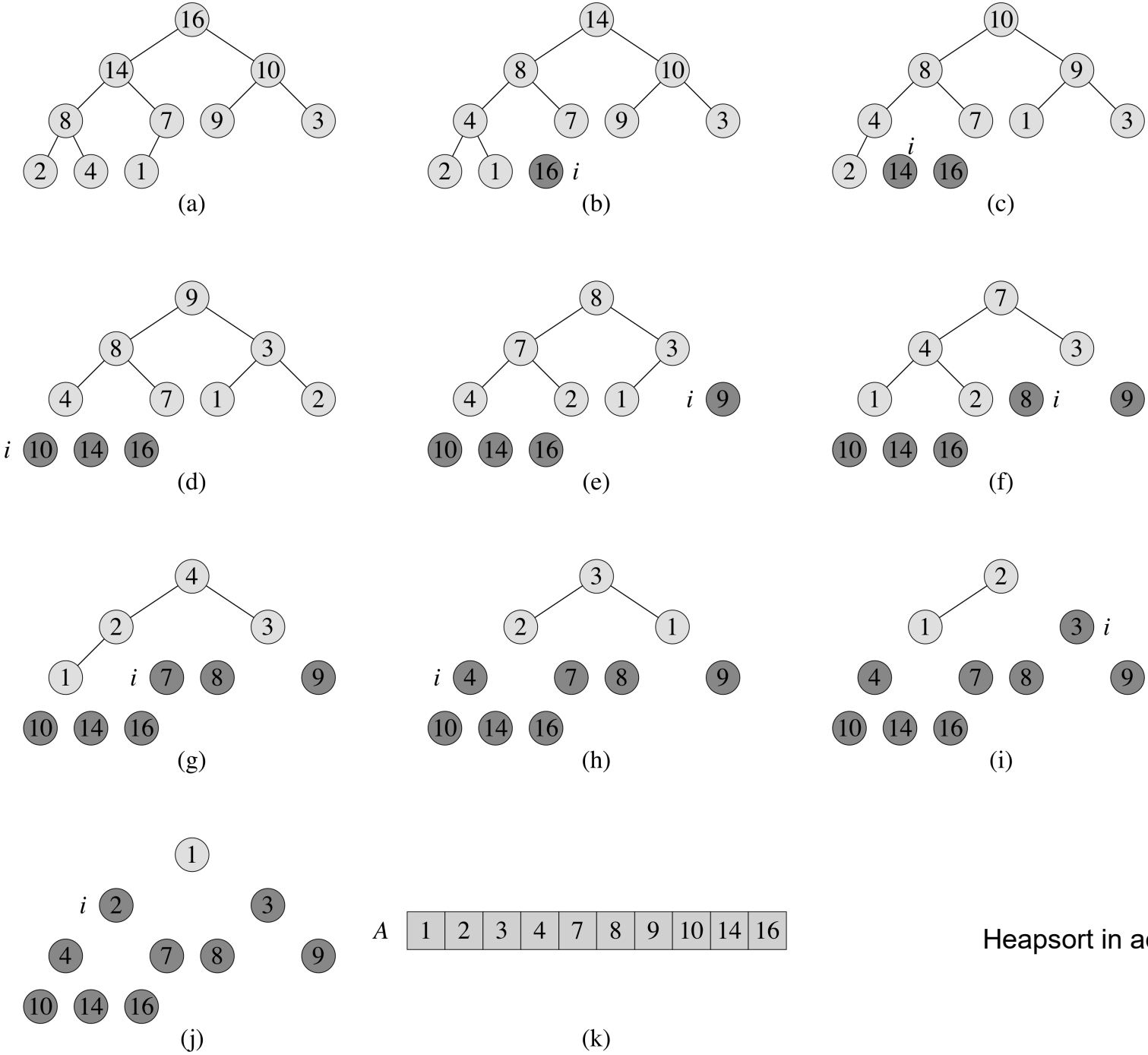


(b)



(c)

Illustration of MAX-HEAPIFY(A,2$_1$)$_4$



Heapsort in action

Example / 3

- Array = [6, 22, 4, 8, 2]

Application of Heaps

- Heap sort
  – Both phases of heap sort are O(log *n*) executed *n* times; thus, heap sort is O(*n* log *n*)

Running time: After **n** iterations the Heap is empty every iteration involves a **swap** and a **max_heapify** operation; hence it takes **O(log n)** time

Overall **O(n log n)**

Application of Heaps

- Behavior same for average and worst case
- Unlike merge sort, sorts in place
- Unlike insertion sort, running time is O(*n log n*)
- Priority queue – Heap implementation uses priority as key field
- Heaps – *garbage collected storage* used in OS and programming languages

Time Complexity Calculation

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

- Build Heap → O(n)

Time Complexity Calculation / 2

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

- Heapify → O(log(n))

Time Complexity Calculation / 2

- Heap Sort → O(n*log(n))

Heap Sort has **O(nlog(n))** time complexities for all the cases (best case, average case and worst case).

- The **MAX - HEAPIFY** procedure, which runs in O(lg n) time, is the key to maintaining the max -heap property.
- The BUI**LD - MAX - HEAP** procedure, which runs in O(n) time, produces a max -heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in O(nlg n) time, sorts an array in place.

Divide-and-conquer approach
Quick Sort

- Quick Sort is a fast sorting algorithm and it is widely applied in practice.

- Like merge sort, quicksort uses divide-and-conquer, and so it's a recursive algorithm.

- It consists 3 main steps ;
  - Choose a pivot value
  - Partition
  - Sort both parts

Divide-and-conquer paradigm

• **Divide:** Partition (rearrange) the array A[p..r] into

two subarrays A[p..q-1] and A[q+1..r] such

– each element of A[p..q-1] is less than or equal to A[q]

– A[q] is less than or equal to each element of A[q+1..r]

– A[q] is called **pivot** - the element around which to partition

(Note that element A[q] is in it's final position in the array)

• **Conquer:** Sort the two subarrays A[p..q-1] and

A[q+1..r] by recursive calls to quicksort

• **Combine:** Since the subarrays are already sorted,

there is no work needed to combine the subarrays,

the entire array A[p..r] is now sorted

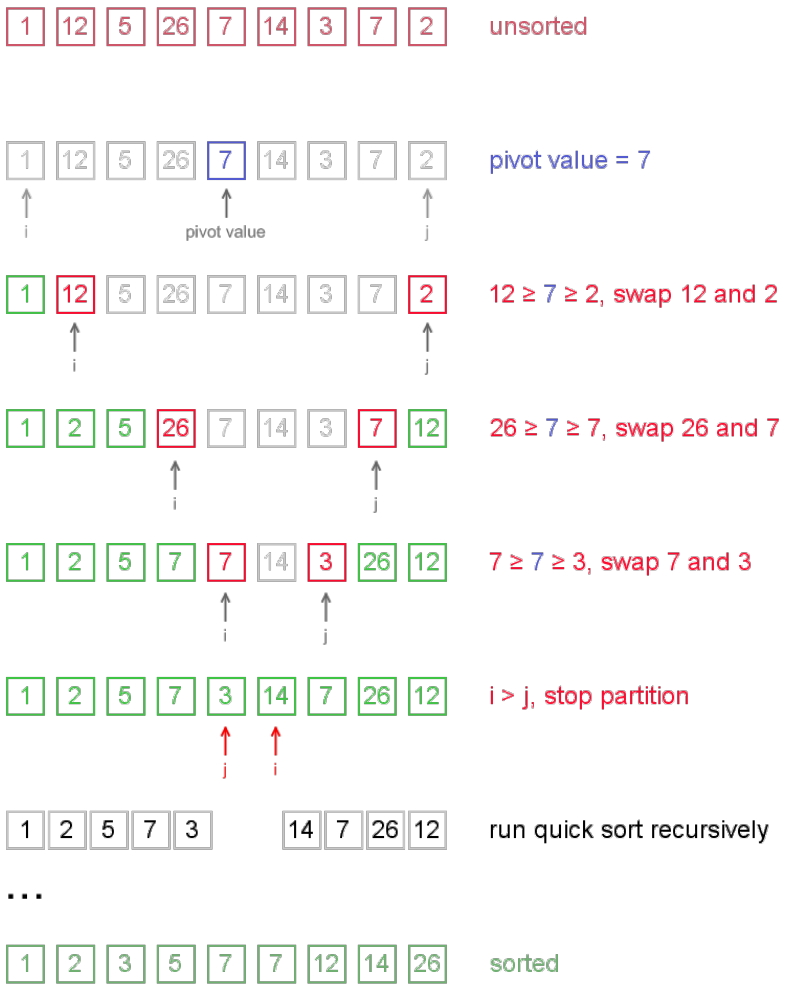
```
QUICKSORT(A, p, r)
  if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
        if A[j] ≤ x
              i = i + 1
              exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

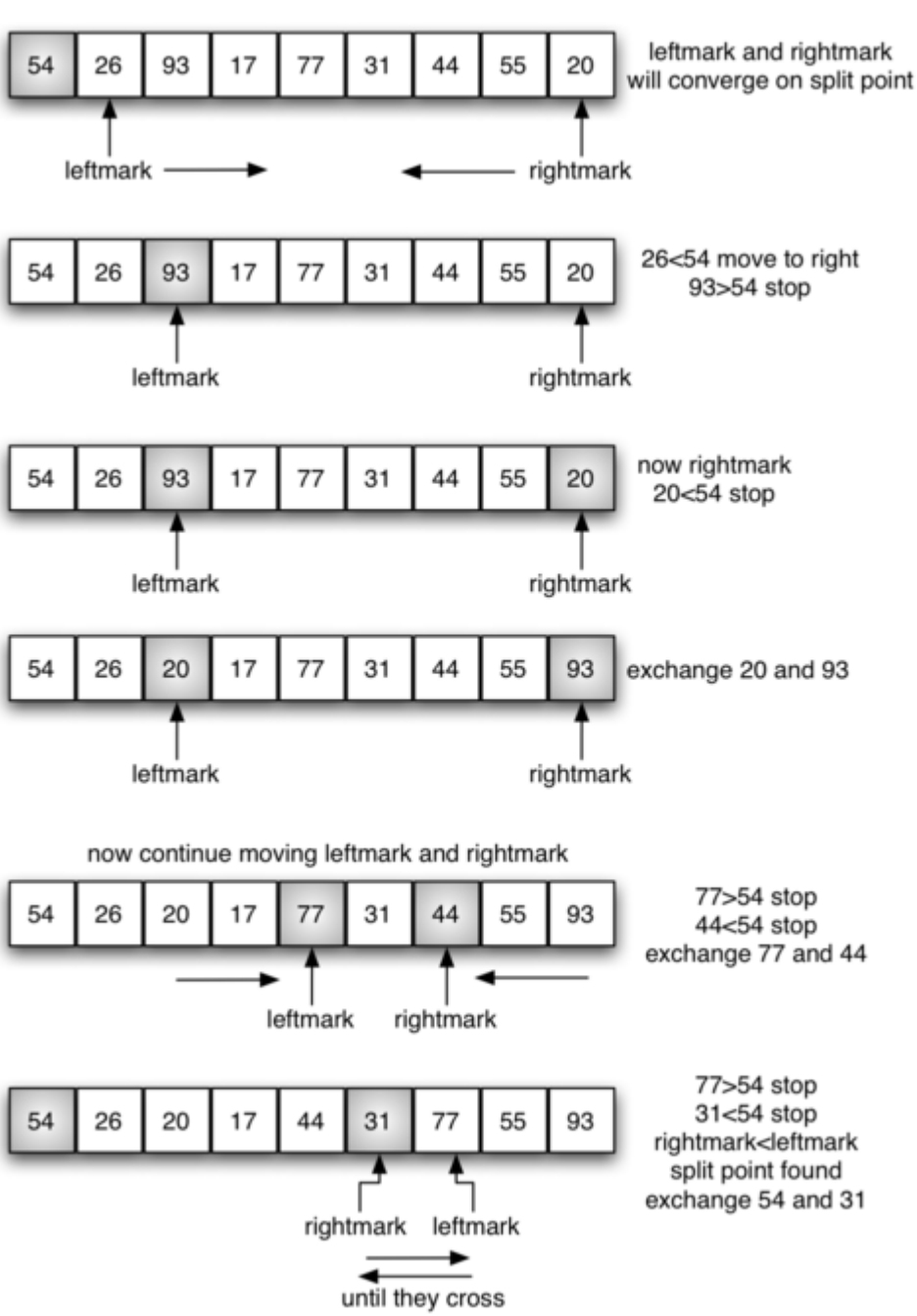
There are two indices **i** and **j** and at the very beginning of the partition algorithm **i** points to the first element in the array and **j** points to the last one.
Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found.
Index **j** is moved backward, until an element with value lesser or equal to the pivot is found.
If **i ≤ j** then they are swapped and i steps to the next position (**i + 1**), j steps to the previous one **(j - 1)**. Algorithm stops, when **i** becomes greater than **j**.
After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

Example 1

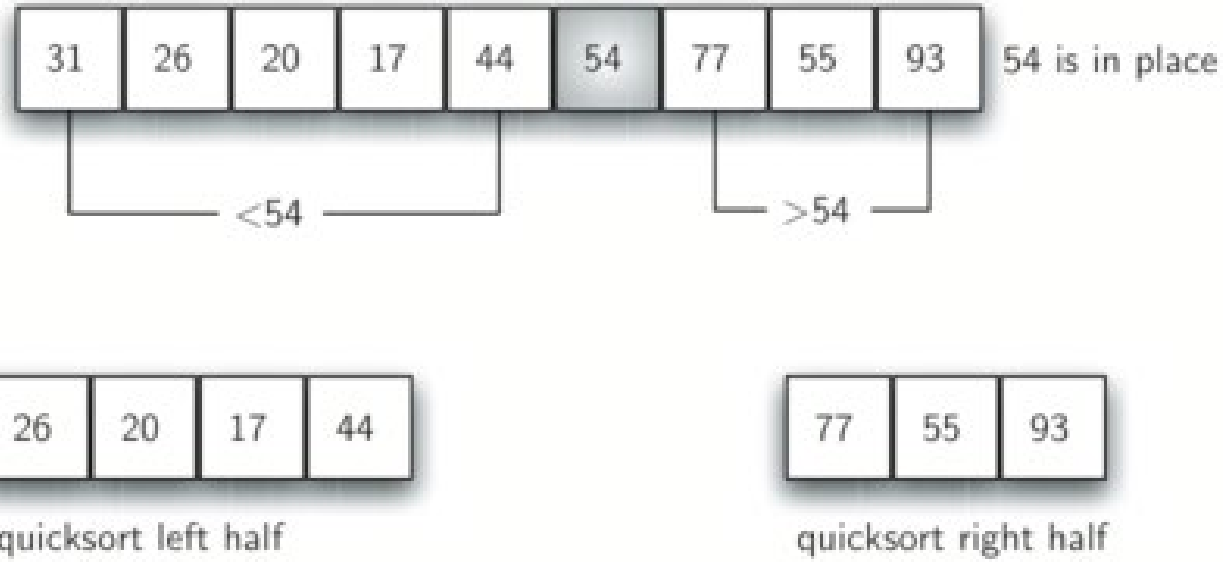
Arr = [1, 12, 5, 26, 7, 14, 3, 7, 2]



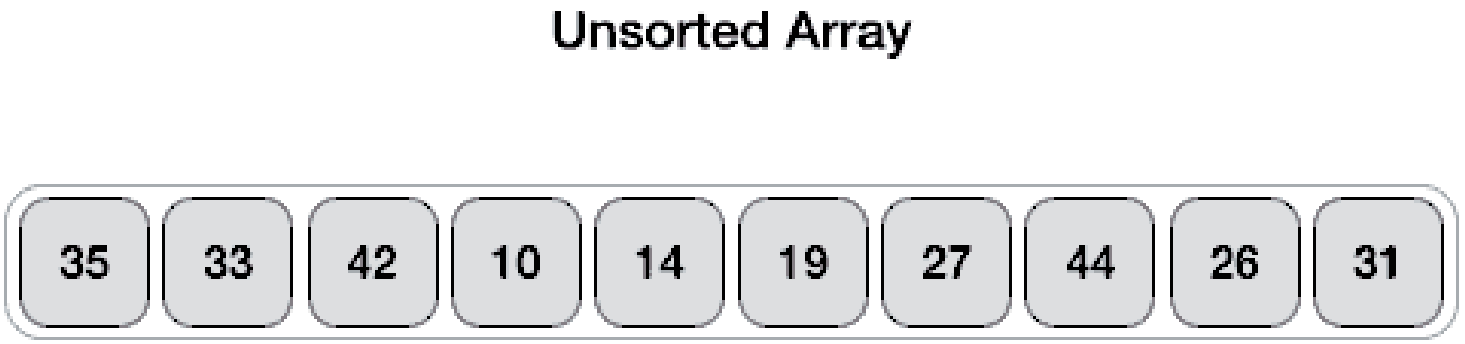
http://www.algolist.net/Algorithms/Sorting/Quicksort



https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html


31
26
20
17
44
54
77
55
93
54 is in place
<54
>54
31
26
20
17
44
quicksort left half
77
55
93
quicksort right half

Assign the last element as a pivot



https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm

Median-of-three Partitioning

- Pivot = median of the **left-most, right-most** and **center** element of the array.

- A = [**3** 6 8 2 **9** 10 1 21 **12**]
- median = [3, **9**, 12] → pivot = 9

- A2 = [**12** 10 5 **7** 19 21 **14**]
  - = [12 7 14] Sorted = [7 **12** 14] → pivot = 12

Median-of-three Partitioning

- Randomly choose three elements from the subarray, and take median of the three as the pivot….
- Median of five?
- Median of nine?
- ….
- Different versions of the quicksort is available

Time Complexity

- ***Worst Case:*** occurs when the partition process always picks greatest or smallest element as pivot. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

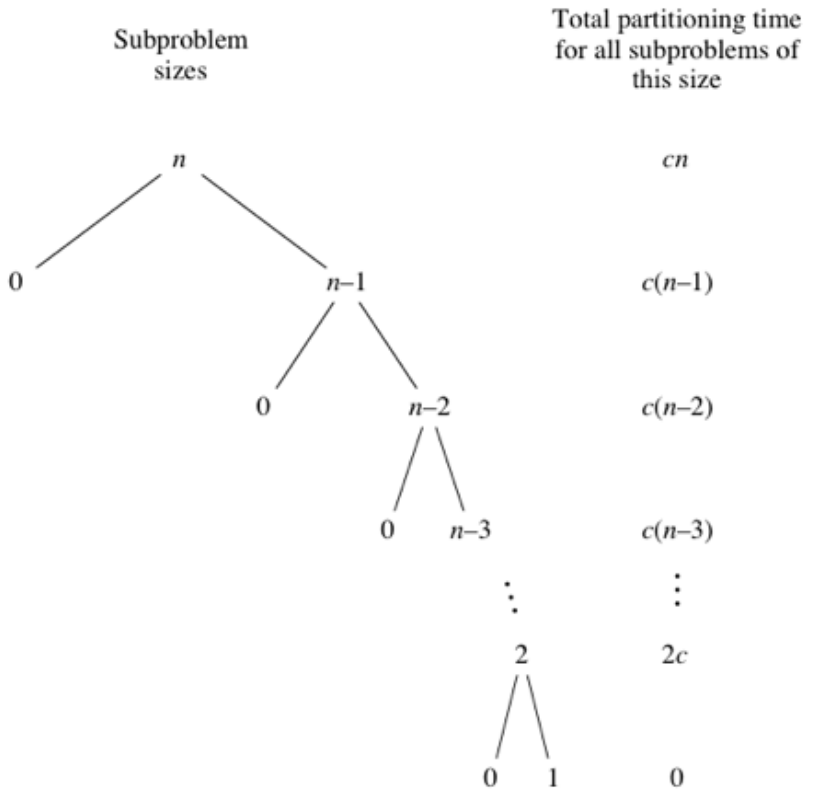
- ***Best Case:*** occurs when the partition process always picks the middle element as pivot.

Worst Case Running Time

- When quicksort always has the most unbalanced partitions possible,
  - then the original call takes *cn* time for some constant *c*,
  - the recursive call on *n-1* elements takes *c(n-1)* time,
  - the recursive call on *n-2* elements takes *c(n-2)* time,
  - and so on.

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Worst Case Running Time / 2



$cn + c(n-1) + c(n-2) + \cdots + 2c$
$= c\,(n + (n-1) + (n-2) + \cdots + 2)$
$= c\,(\,(n+1)\,(n/2) - 1\,)\,.$

$\Theta(n\text{^}2)$

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

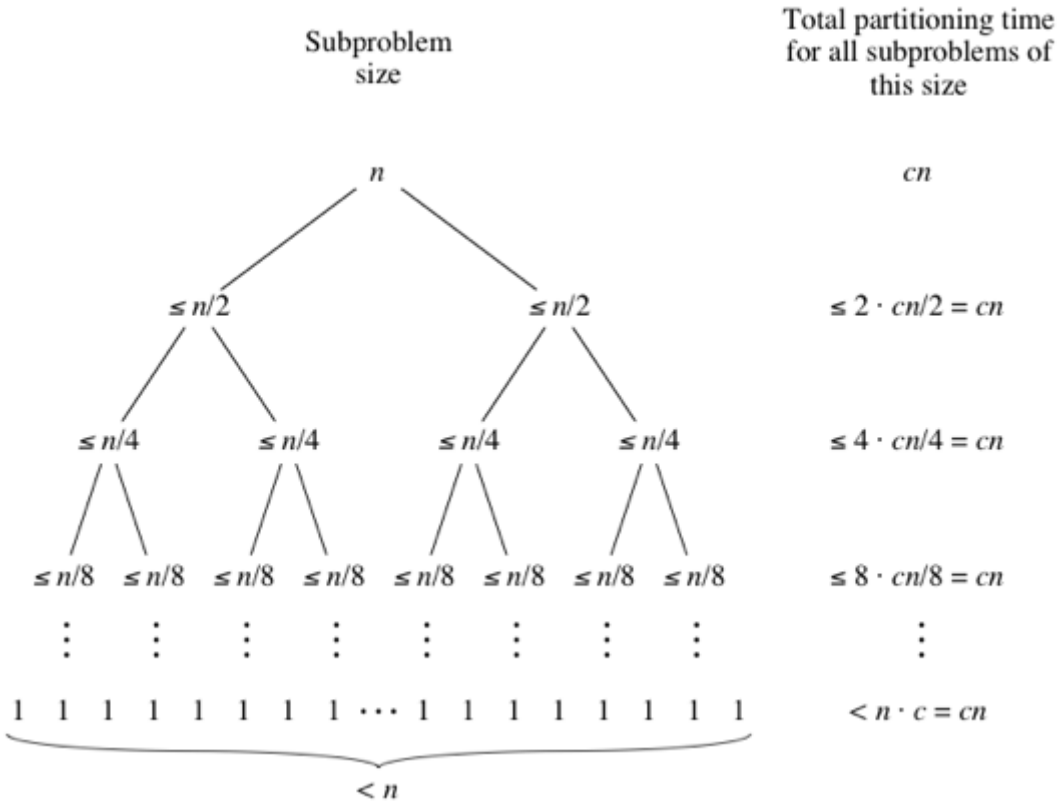
Best Case Running Time

Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other.

- The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has $(n-1) / 2$ elements.
- The latter case occurs if the subarray has an even number n of elements and one partition has $n / 2$ elements with the other having $n/2 - 1$

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Best Case Running Time / 2



https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

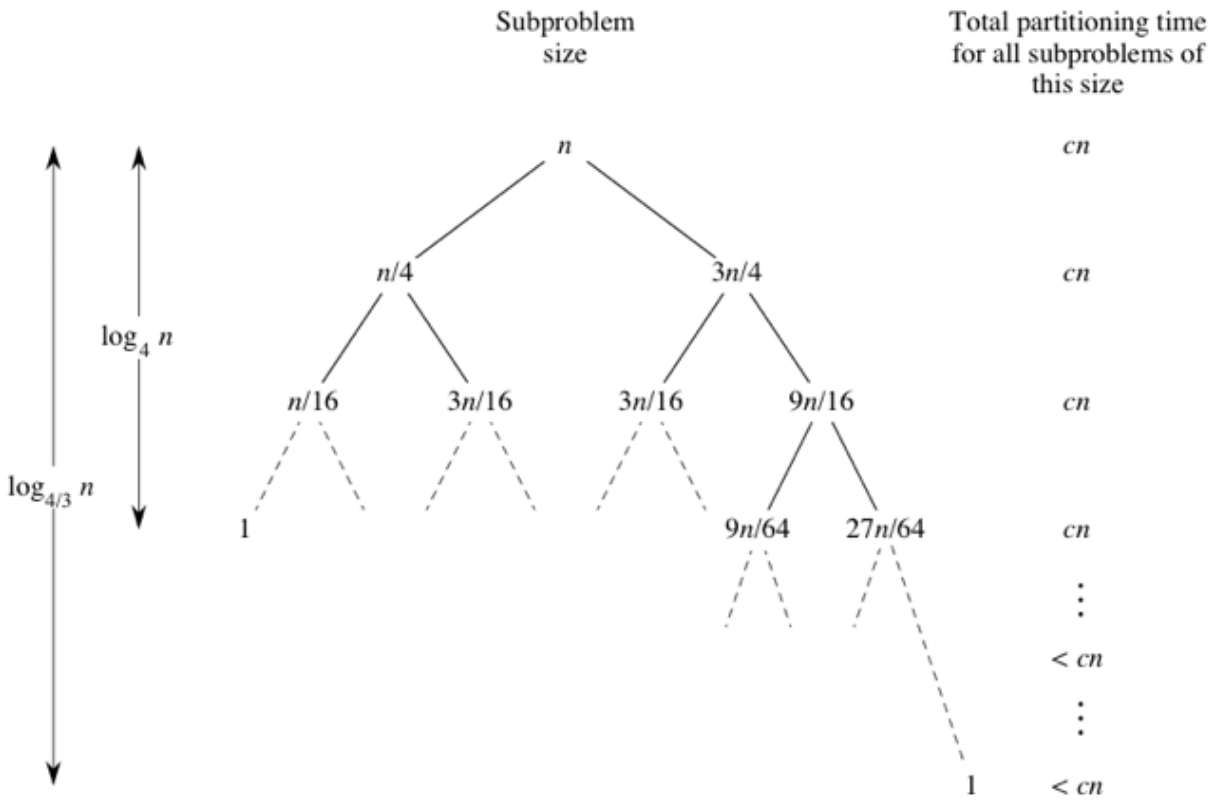
Average Case Running Time

The average-case running time is also $\Theta(n \log_2 n)$

Let's imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-to-1 split.
That is, imagine that each time we partition, one side gets $3n/4$ elements and the other side gets $n/4$. (To keep the math clean, let's not worry about the pivot.)

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Average Case Running Time / 2



https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Average Case Running Time / 3

$$\log_a n = \frac{\log_b n}{\log_b a}$$

for all positive numbers a, b, and n. Letting $a = 4/3$ and $b = 2$, we get that

$$\log_{4/3} n = \frac{\log_2 n}{\log_2(4/3)} \ ,$$

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Time Complexity of Quick Sort

- Best Case - Θ(n log n)
- Average Case - Θ(n log n)
- Worst Case - Θ(n^2)

Median of Three

- Aims to improve worst case scenario
  - ~(n log n)

- If you are interested in, read the following research paper;
- https://github.com/brucelilly/quickselect/blob/master/lib/libmedian/doc/pub/generic/paper.pdf

Comparison Sorts

- The only operation that may be used to gain order information about a sequence is comparison of pairs of elements.
- All sorts seen so far are comparison sorts: insertion sort, merge sort, heapsort, quicksort
- Comparison sorts use $\Omega(n \lg n)$ comparisons in the best case to sort n elements
- Merge sort and Heapsort are asymptotically optimal comparison sorts

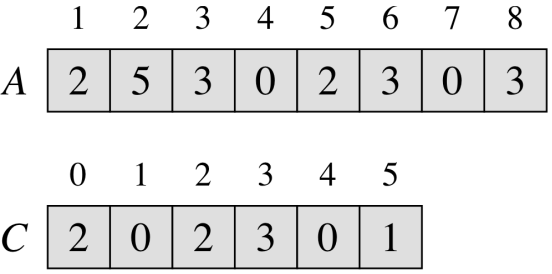
Sorting in Linear Time

- Sorting algorithms that run in linear time
  - counting sort
  - radix sort
  - bucket sort
- These algorithms use operations other than comparisons to determine the sorted order

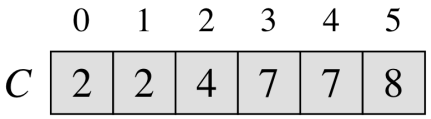
Counting Sort

- Depends on a key assumption: numbers to be sorted are integers in the range 0 to k
- Input: A[1..n] where A[j] is an integer in the range 0 to k for j = 1..n
- Output: B[1..n] – sorted
- Auxiliary Storage: C[0..k]

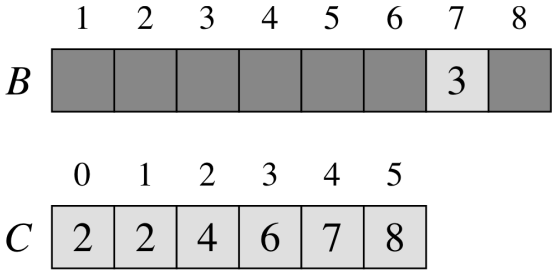
```
COUNTING-SORT(A, B, n, k)
  let C[0..k] be a new array
  for i = 0 to k
      C[i] = 0
  for j = 1 to n
      C[A[j]] = C[A[j]] + 1
  for i = 1 to k
      C[i] = C[i] + C[i − 1]
  for j = n downto 1
      B[C[A[j]]] = A[j]
      C[A[j]] = C[A[j]] − 1
```


1 2 3 4 5 6 7 8
A 2 5 3 0 2 3 0 3
0 1 2 3 4 5
C 2 0 2 3 0 1

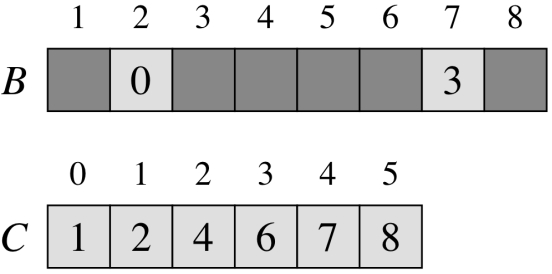
(a)


0 1 2 3 4 5
C 2 2 4 7 7 8

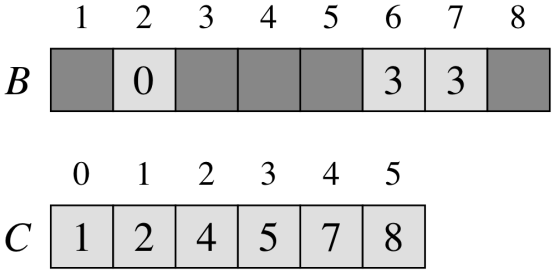
(b)


1 2 3 4 5 6 7 8
B 3
0 1 2 3 4 5
C 2 2 4 6 7 8

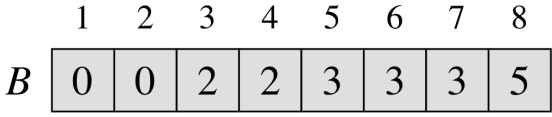
(c)


1 2 3 4 5 6 7 8
B 0 3
0 1 2 3 4 5
C 1 2 4 6 7 8

(d)


1 2 3 4 5 6 7 8
B 0 3 3
0 1 2 3 4 5
C 1 2 4 5 7 8

(e)


1 2 3 4 5 6 7 8
B 0 0 2 2 3 3 3 5

(f)

Counting Sort

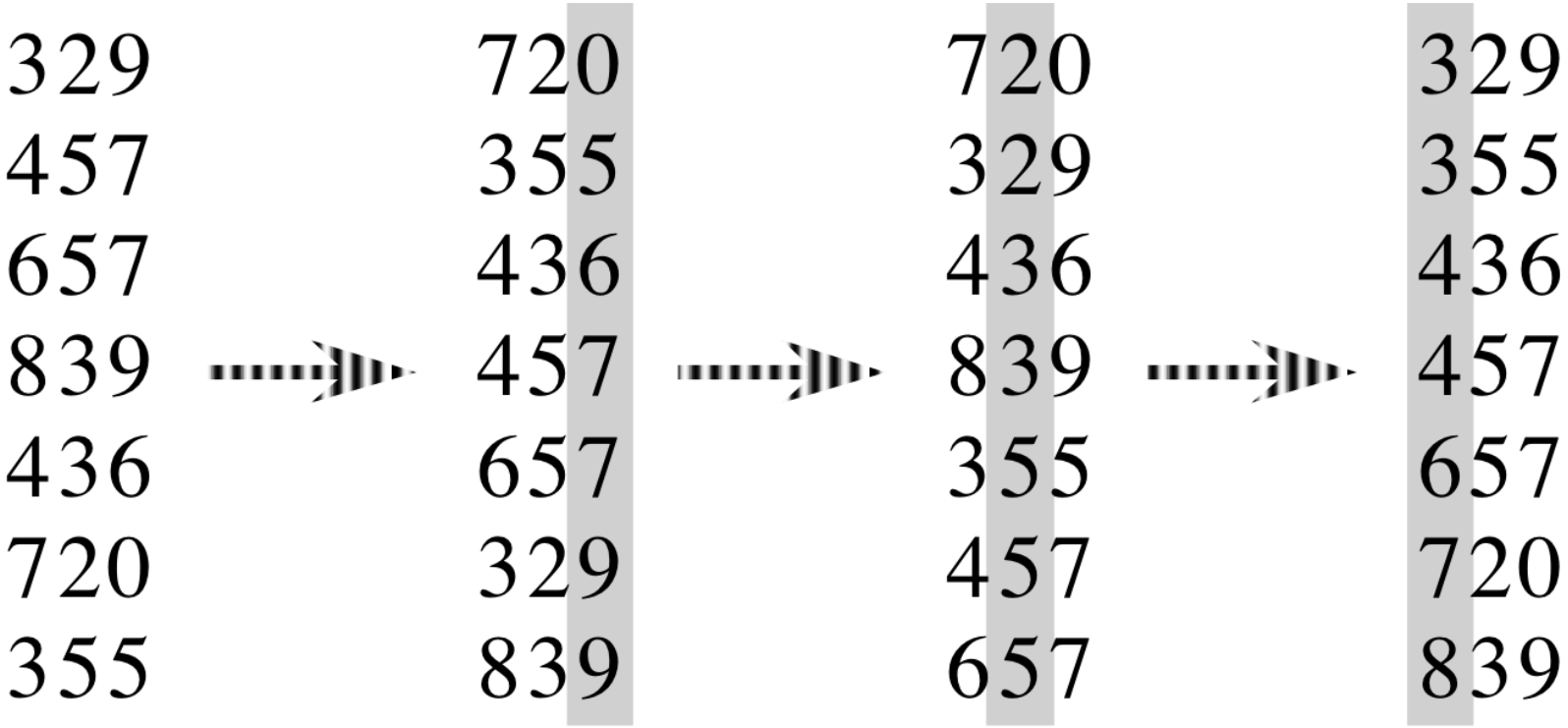
- Counting sort is ***stable*** (numbers with the same value appear in the output array in the same order as they do in the input array)
- Overall time is $\Theta(k+n) = \Theta(n)$, when $k = O(n)$
- Counting sort is often used in radix sort due to it's stability property

Radix Sort

- Algorithm originally used by the card-sorting machines
- Key idea: sort least significant digits first
- To sort *d* digits:

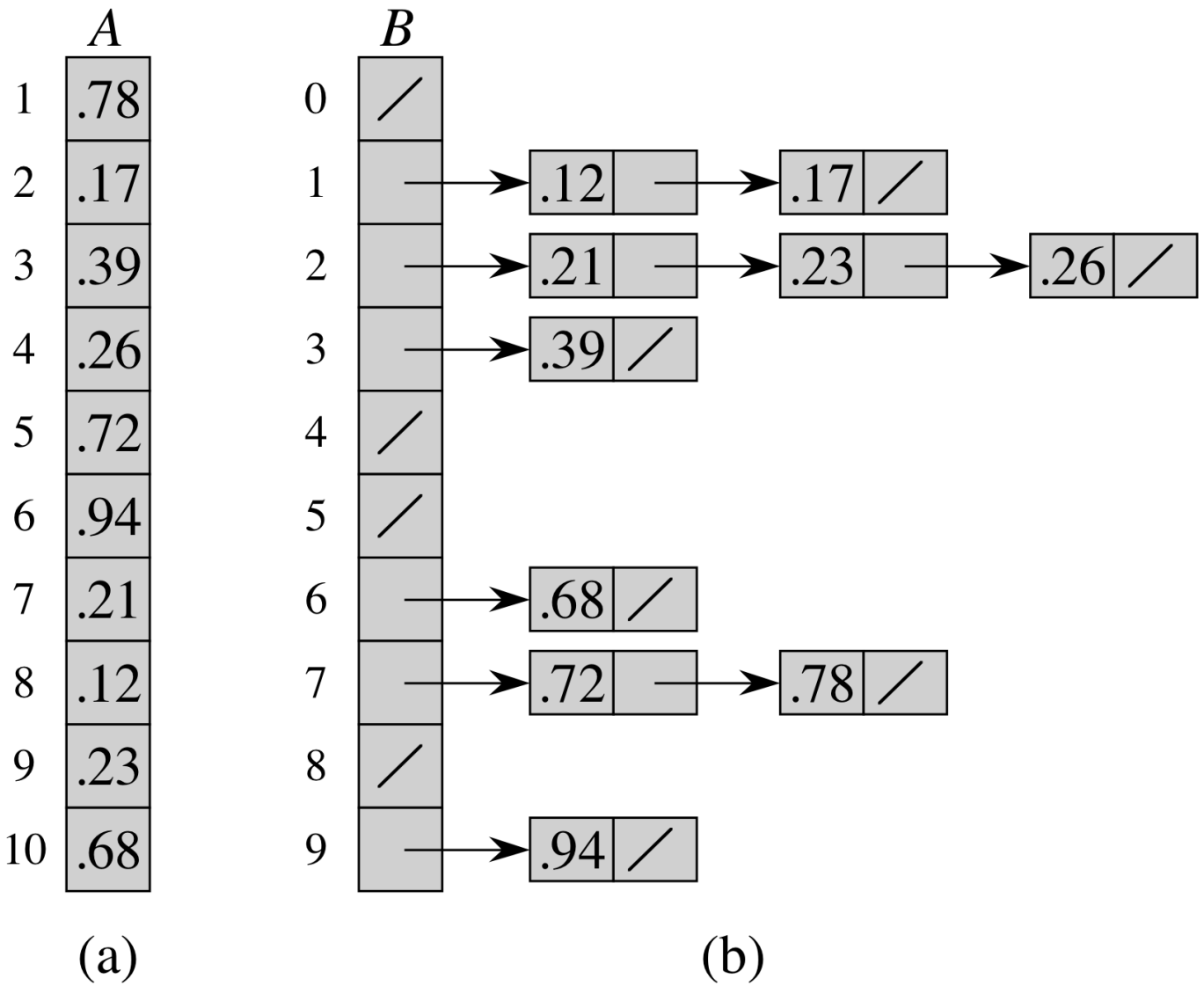
```
RADIX_SORT (A, d)
        for i = 1 to d
                use a stable sort to sort array A on digit i
```

- Time taken to sort *n* *d*-digit numbers in the range 0 to k is $\Theta(d(n+k))$ if the stable sort used takes $\Theta(n+k)$ time


329
457
657
839
436
720
355
720
355
436
457
657
329
839
720
329
436
839
355
457
657
329
355
436
457
657
720
839

Bucket Sort

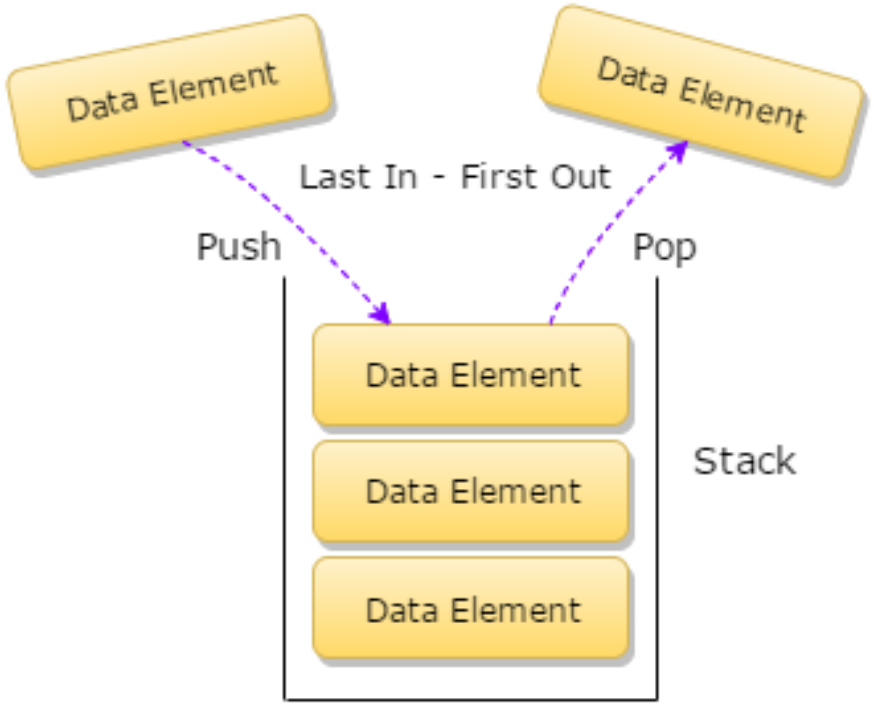
- Assumes the input is generated by a random process that distributes elements uniformly over the interval [0,1)
- *Algorithm*
  - Divide the interval [0,1) into *n* equal-sized *buckets.*
  - Distribute the *n* input values into the buckets.
  - Sort each bucket.
  - Then go through buckets in order, listing elements in each one.


A
1
.78
2
.17
3
.39
4
.26
5
.72
6
.94
7
.21
8
.12
9
.23
10
.68
(a)
B
0
1
.12
.17
2
.21
.23
.26
3
.39
4
5
6
.68
7
.72
.78
8
9
.94
(b)

```
BUCKET-SORT(A, n)
  let B[0..n − 1] be a new array
  for i = 1 to n − 1
      make B[i] an empty list
  for i = 1 to n
      insert A[i] into list B[⌊n · A[i]⌋]
  for i = 0 to n − 1
      sort list B[i] with insertion sort
  concatenate lists B[0], B[1], ..., B[n − 1] together in order
  return the concatenated lists
```

Stacks and Queues

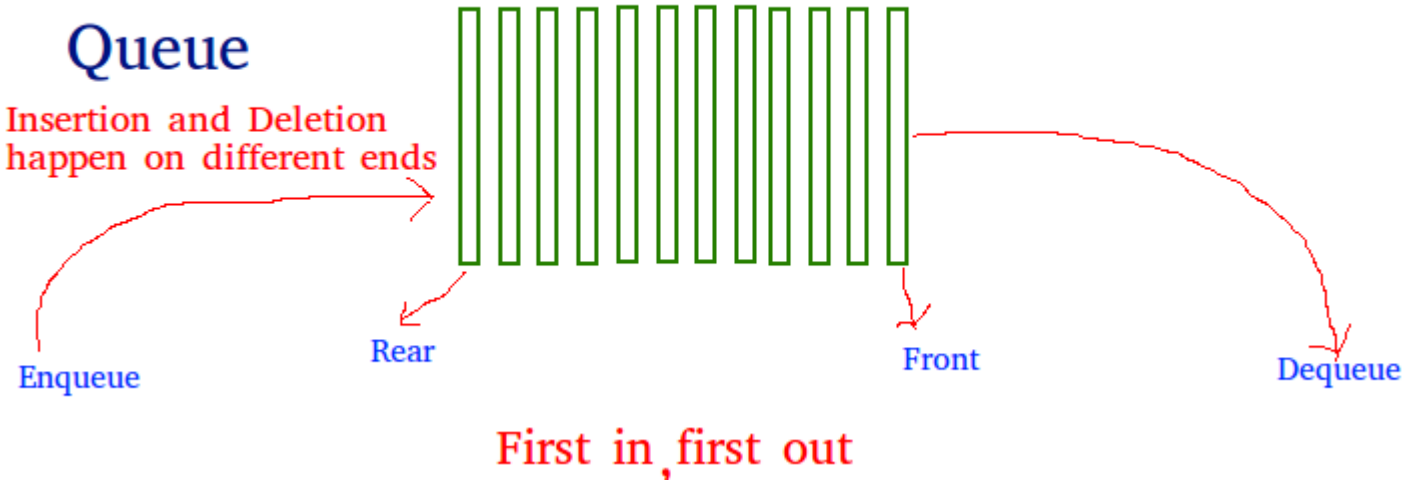
- Dynamic sets in which the element removed from the set by the DELETE operation is prespecified
- Stack
  - element deleted is the one most recently inserted
  - implements a **last-in, first-out (LIFO)** policy
  - INSERT operation is often called PUSH
  - DELETE operation is often called POP



https://www.w3schools.in/data-structures-tutorial/stack/

Stacks and Queues

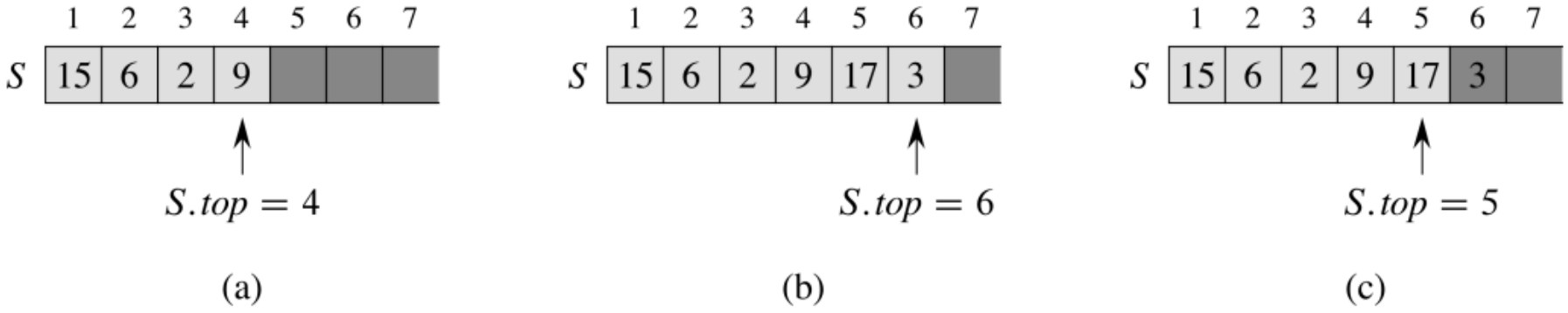
- Queue
  - element deleted is the one that has been in the set for the longest time
  - implements a first-in, first-out (FIFO) policy
  - INSERT operation is often called ENQUEUE
  - DELETE operation is often called DEQUEUE
- Deque – double-ended queue
  - a queue that allows insertion and deletion at both ends



https://www.geeksforgeeks.org/queue-data-structure/

Implementing a stack using arrays

S[1..n] - array



After
PUSH(S, 17)
PUSH(S, 3)

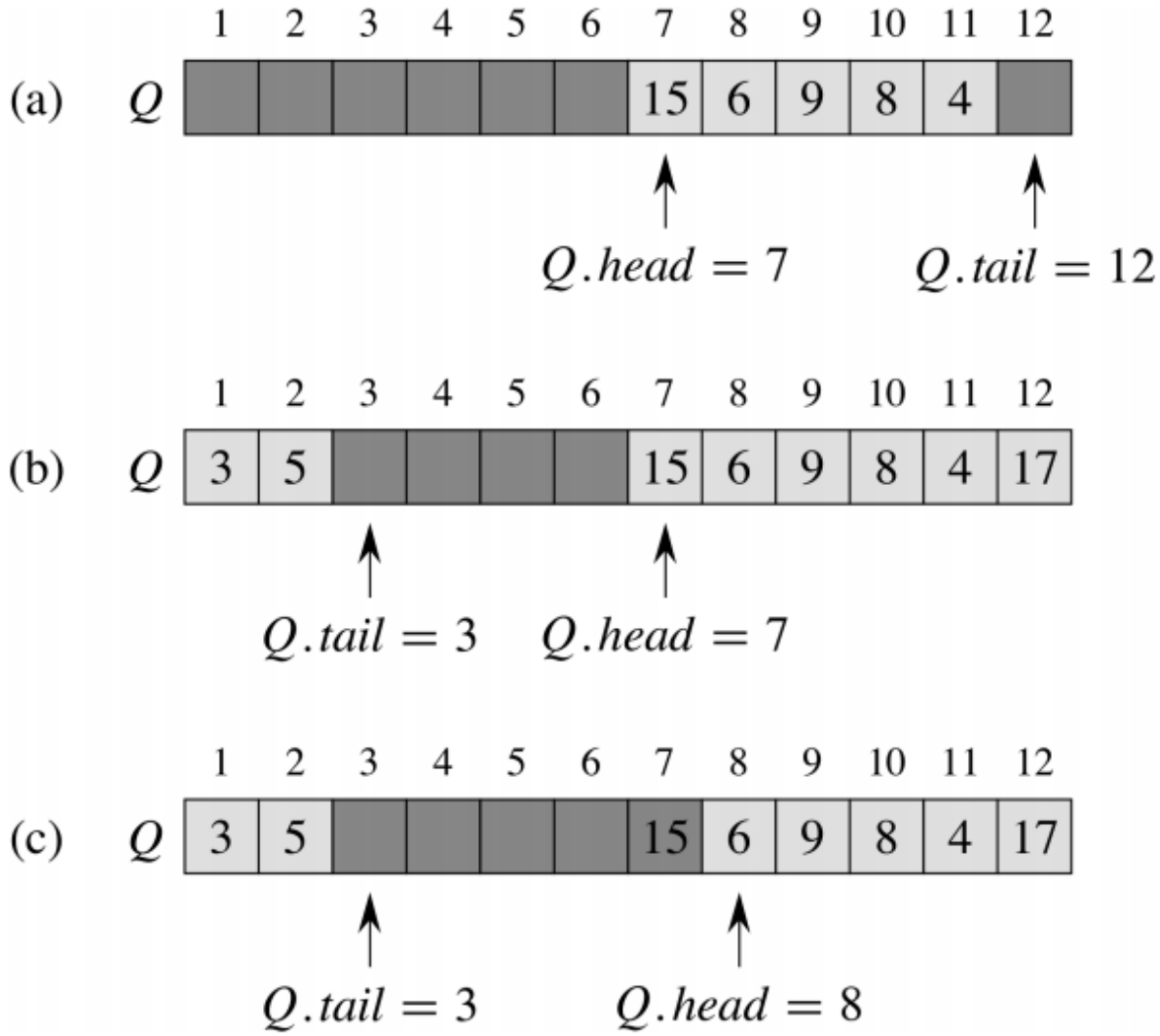
After
POP(S)

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1        // overflow not shown
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Implementing a queue using arrays



After
ENQUEUE(Q, 17)
ENQUEUE(Q, 3)

ENQUEUE(Q, 5)

After
DEQUEUE(Q)

// overflow and underflow not shown

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Python Stack Implementation

```
In [13]: stack=[]

In [14]: stack.append("UAB")
         stack.append("CS")
         stack.append("303")
         print(stack)

         ['UAB', 'CS', '303']

In [15]: print(stack.pop())

         303

In [16]: print(stack.pop())

         CS

In [17]: print(stack.pop())

         UAB
```

Python Queue

In [22]:
```
queue=[]
```

In [23]:
```
queue.append("UAB")
queue.append("CS")
queue.append("303")
print(queue)
```

```
['UAB', 'CS', '303']
```

In [25]:
```
print(queue.pop(0))
```

```
UAB
```

In [26]:
```
print(queue.pop(0))
```

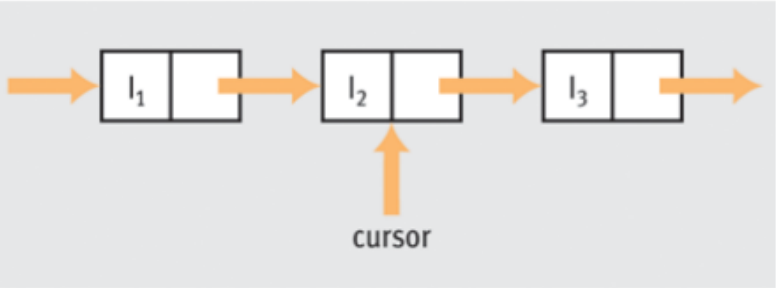
```
CS
```

In [27]:
```
print(queue.pop(0))
```

```
303
```

Linked Lists

- Linked Lists provide a simple, flexible representation for dynamic sets
  - the order in a linked list is determined by a pointer in each object
  - retrieval, insertion, deletion allowed anywhere within the structure
- **List** – ordered collection of zero or more **nodes**
- Nodes contain two **fields**
  - **Information** field (**data** field)
  - **Pointer** field (**next** field)



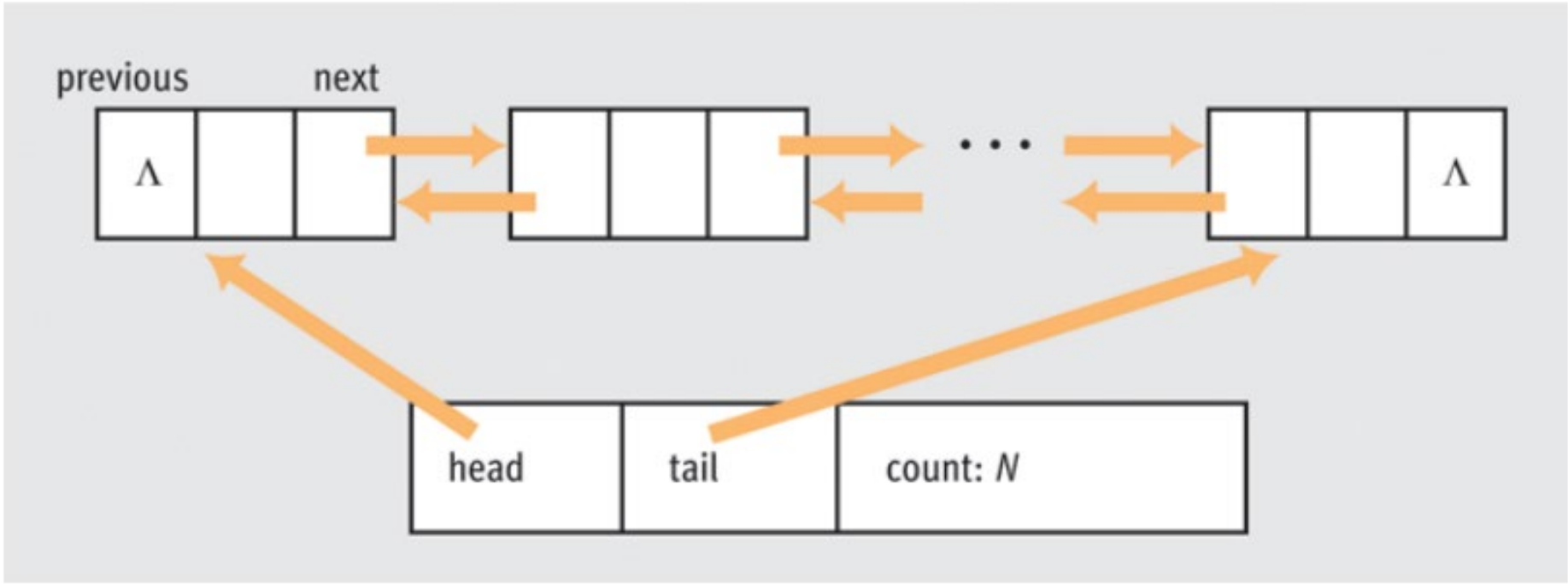
Implementations of Lists

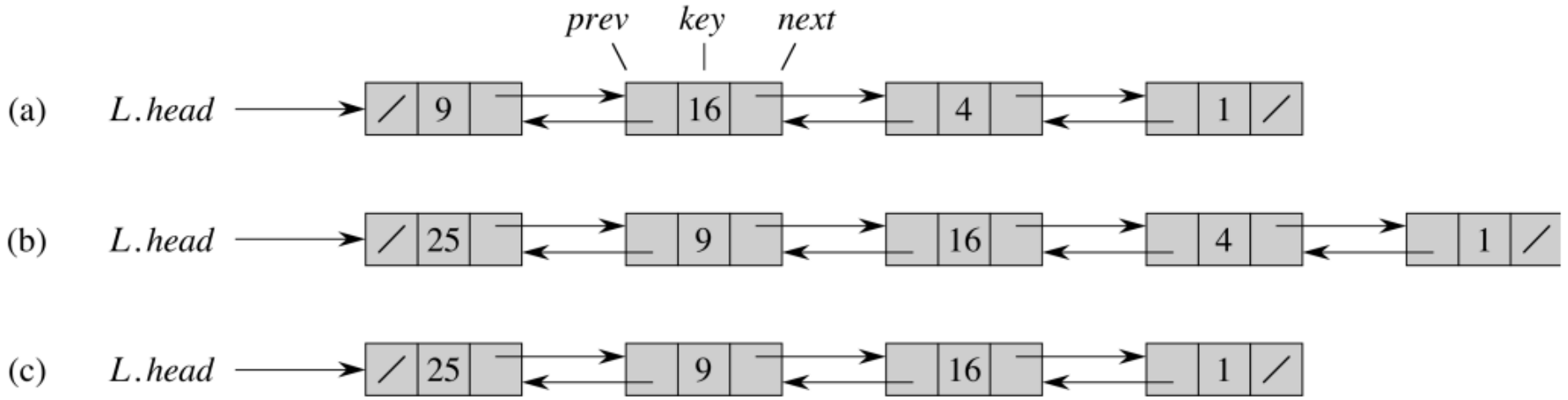
- Insert a node at the cursor:
  - Allocating space for the node
  - Assign the next field to the successor of the cursor
  - Assign the node to the next field of the cursor
  - Update total node count
- Adding a node is O(1) time complexity
- Many operations are O(1) because cursor points to the location, not using physical adjacency

Doubly Linked Lists

- **Singly linked list** – each node has a pointer to its successor
- **Doubly linked list** – nodes have a pointer to successor and predecessor
  - head and last nodes, cursor, and count
- Time somplexity to search an element O(n)
  - Doubly linked list, list insertion/deletion time complexity O(1)

Doubly linked list data structure





(b) After LIST_INSERT(L, 25)

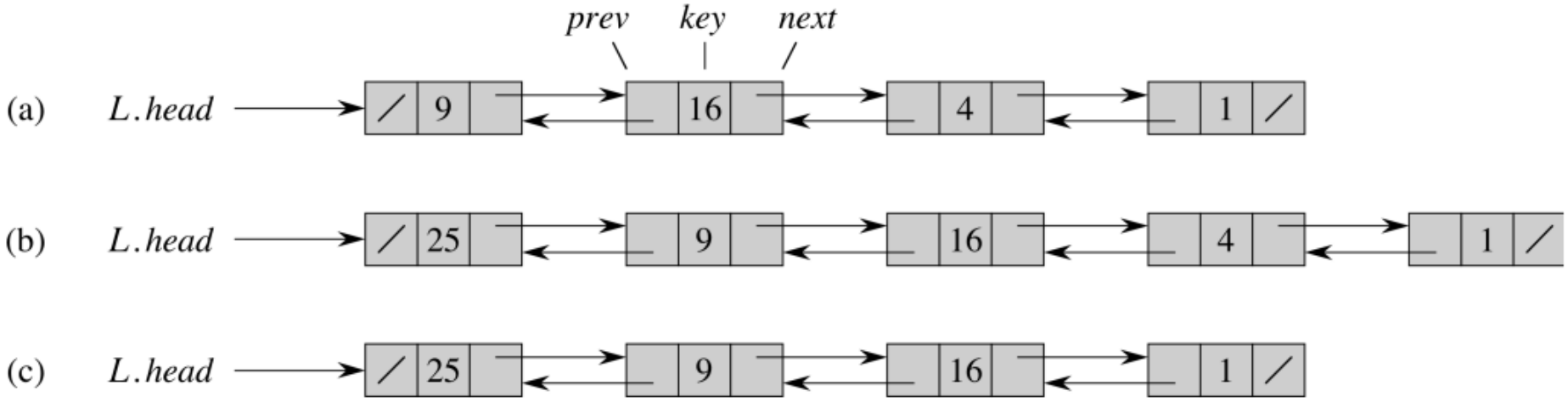
(c) After LIST_DELETE(L, 4)

LIST-INSERT(L, x)

```
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

LIST-DELETE(L, x)

```
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```



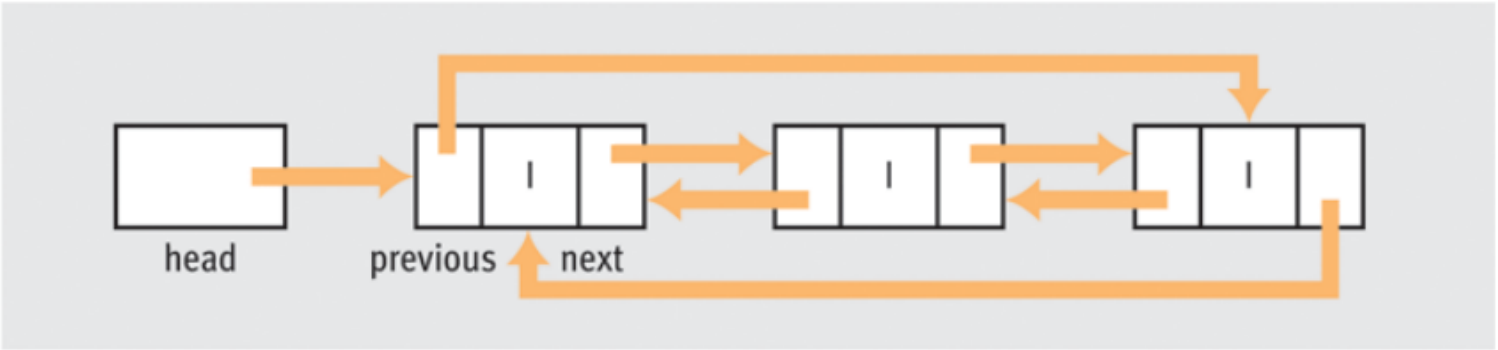
LIST-SEARCH(L, k)

1 $x = L.head$
2 **while** $x \neq$ NIL and $x.key \neq k$
3 $\quad x = x.next$
4 **return** x

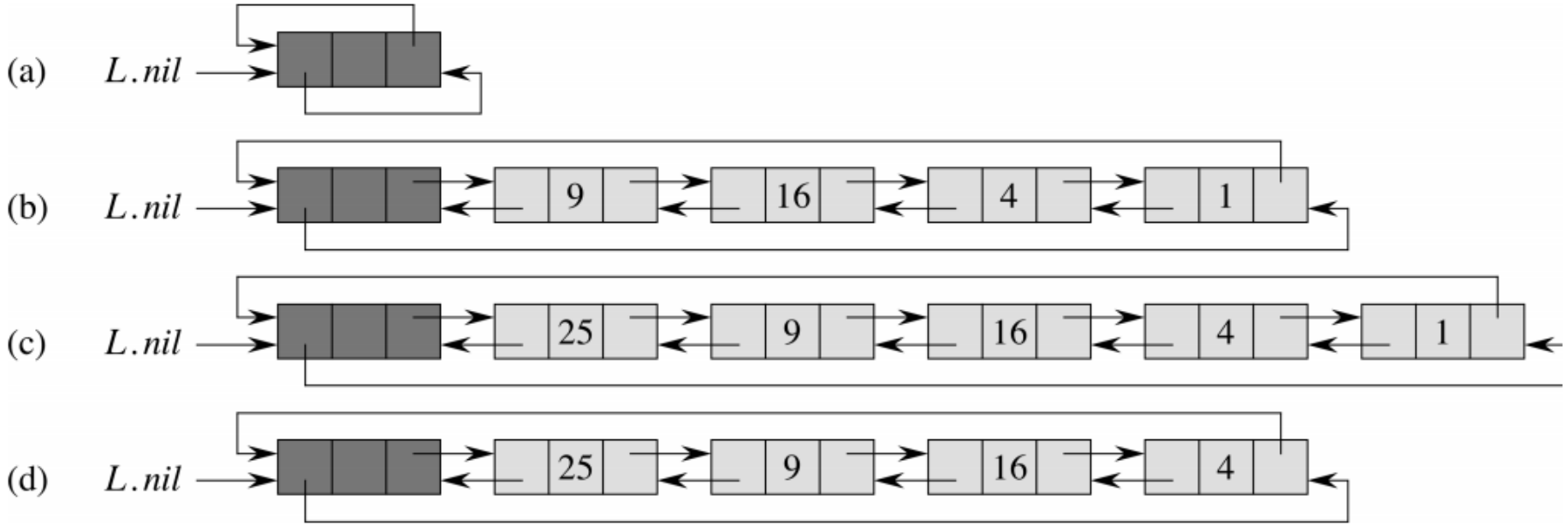
Circular Lists

- **Singly linked circular list (ring)**
  - last node next field points to the head of the list
  - No special case at ends of list
  - next() operation on last node returns first node
  - previous() operation on first node returns last node
- **Doubly linked circular list**
  - Last node successor points to the head node
  - Head node predecessor points to last node

Circular, doubly linked list



Circular, doubly linked list with a sentinel



Why Hash tables / Hashing ?

- Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:
  - Insert a phone number and corresponding information.
  - Search a phone number and fetch the information.
  - Delete a phone number and related information.

https://www.geeksforgeeks.org/hashing-set-1-introduction/

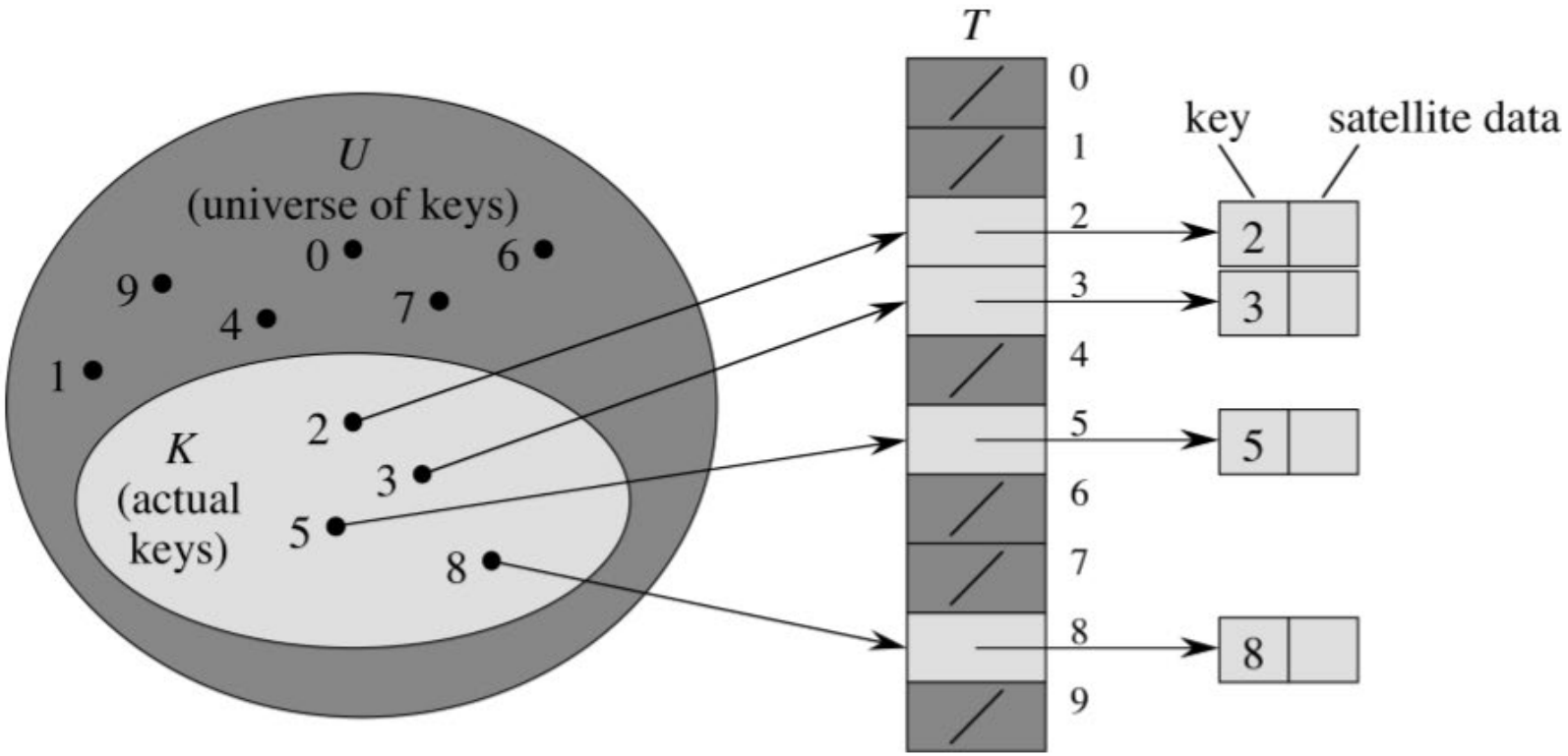
- We can think of using the following data structures to maintain information about different phone numbers.
  - Array of phone numbers and records.
  - Linked List of phone numbers and records.
  - Balanced binary search tree with phone numbers as keys.
  - Direct Access Table.

https://www.geeksforgeeks.org/hashing-set-1-introduction/

- **Direct Access Table:** here we make a big array and use phone numbers as index in the array.
- An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number.
- Time complexity wise this solution is the best among all, we can do all operations in O(1) time
- …..

- Need a huge storage ☹

Direct-address tables

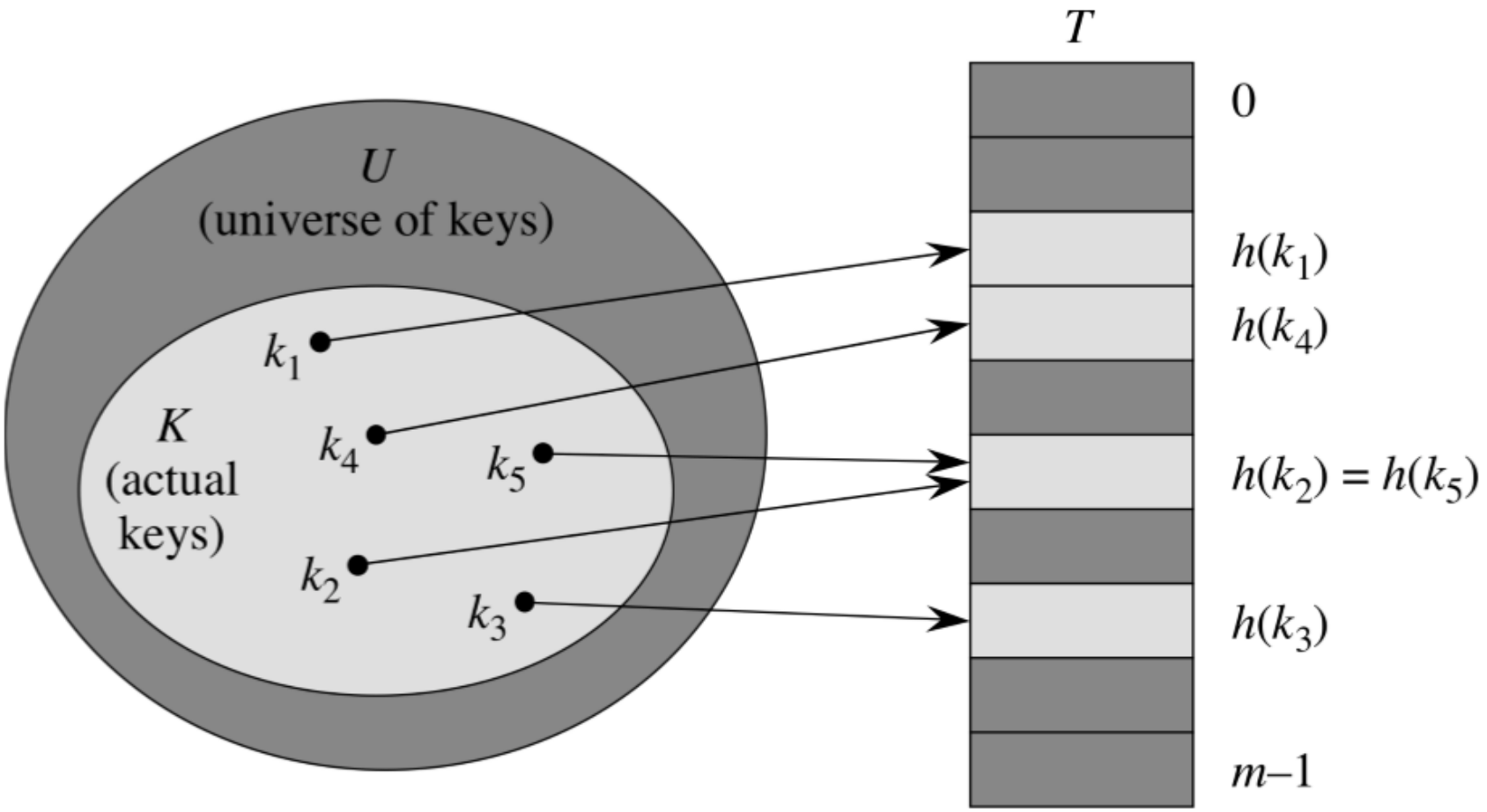


DIRECT-ADDRESS-SEARCH(T, k)
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
$T[key[x]] = x$

DIRECT-ADDRESS-DELETE(T, x)
$T[key[x]] = \text{NIL}$

Hash Tables



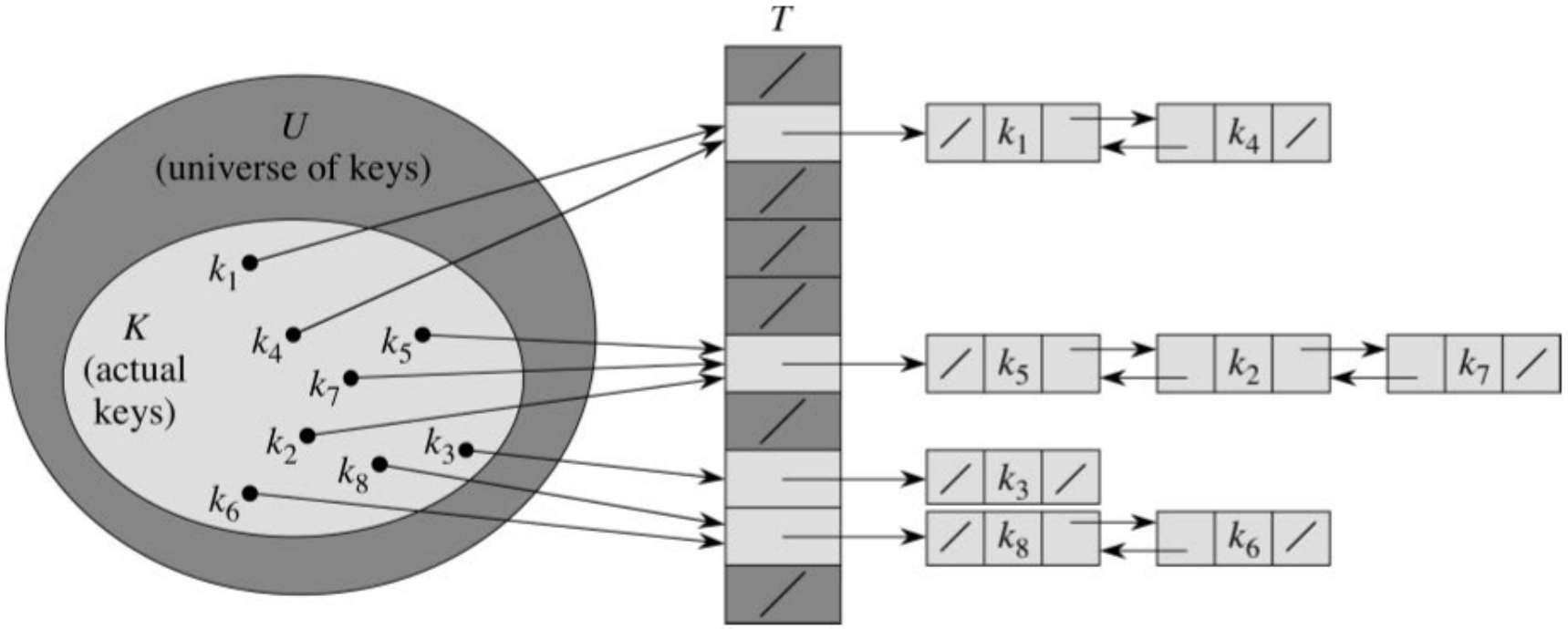
h : U ➔ {0, 1, ….., m-1}

Problem ?

- There is one hitch: two keys may hash the to the same slot, we call this situation a **collision**
- Fortunately, we have effective techniques for resolving the conflict created by collisions. For example; chaining

- **In chaining**: we place all elements that hash to the same slot into the same linked list.

Collision resolution by chaining



CHAINED-HASH-INSERT(T, x)

insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-DELETE(T, x)

delete x from the list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list $T[h(k)]$

Time Complexity

	Average	Worst Case
space	O(n)	O(n)
insert	O(1)	O(n)
lookup	O(1)	O(n)
delete	O(1)	O(n)

Hash Function

- A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
- A good hash function should have following properties

  1) Efficiently computable.

  2) Should uniformly distribute the keys (Each table position equally likely for each key)
- For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

https://www.geeksforgeeks.org/hashing-set-1-introduction/

Hash Functions

- A good hash function satisfies the assumption of a simple uniform hashing (each key is likely to hash to any of the *m* slots, independently of where any other key has hashed up)
- Heuristic approaches:
  - hashing by division
  - hashing by multiplication
- Randomization:
  - universal hashing, provides better performance on average

Hash Functions

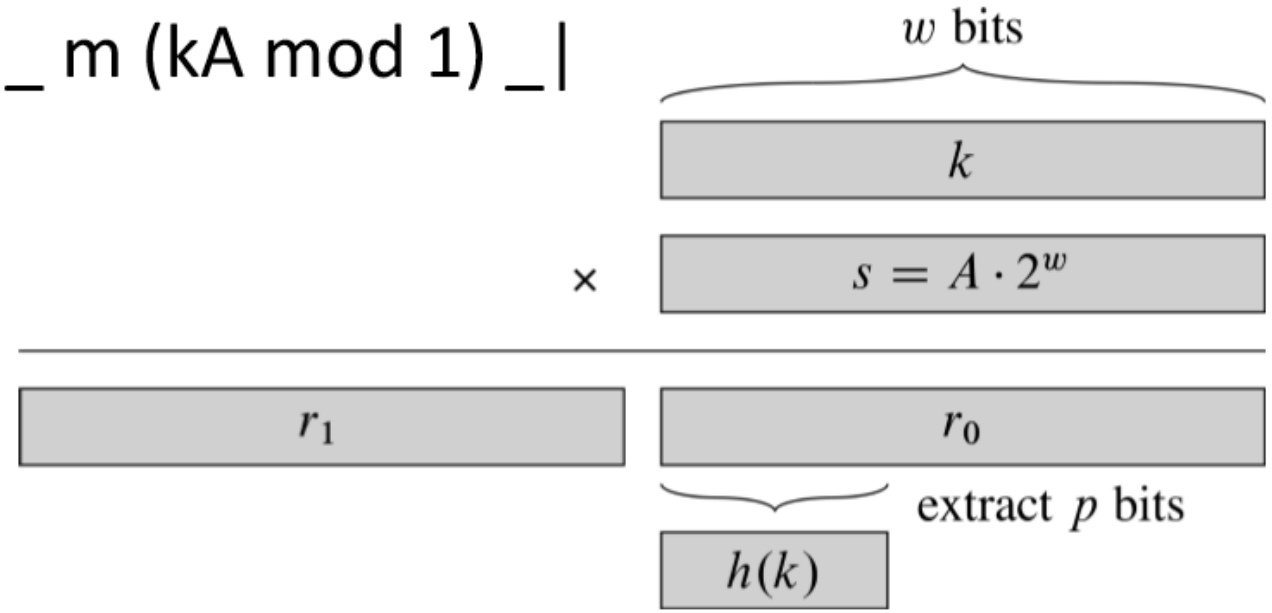
- A good hash function should satisfy the following three key requirements
  - Deterministic- equal keys should produce the same hash value
  - efficient to compute
  - Uniformly distribute the keys

Division Method

- Map a key *k* into one of *m* slots by taking the remainder of *k* divided by *m*: $h(k) = k \bmod m$
- A prime number not too close to an exact power of 2 is often a good choice for *m*
- What happens if m is a power of 2 ($m = 2^p$) ?
- Consider inserting the keys *10, 22, 31, 4, 15, 28, 17, 88, 59* into a hash table that has 11 slots. Demonstrate what happens when the keys are inserted into a hash table with collisions resolved by chaining and the hash function $h(k) = k \bmod 11$.

Multiplication Method of Hashing

- Step 1: multiply the key by a constant A (0 < A < 1) and extract the fractional part of kA
- Step 2: multiply the above value by m and take the floor of the result
- h(k) = |_ m (kA mod 1) _|



Universal hashing

- If a malicious adversary chooses the keys to be hashed by some fixed hash function, then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of Big Theta (n).
- Any fixed hash function is vulnerable to such terrible worst-case behavior; the only effective way to improve the situation is to choose the hash function **randomly** in a way that is **independent** of the keys that are actually going to be stored.
- This approach, called **universal hashing**, can yield provably good performance on average, no matter which keys the adversary chooses.

Open Addressing

- All elements occupy the hash table itself, no lists and no elements stored outside the table
- Avoids pointers altogether, instead of following pointers, the slots to be examined are ***computed***
- To insert a key into the hash table successively examine, or ***probe***, the hash table until an empty slot is found
- The sequence of positions probed *depends upon the key being inserted*

Open Addressing

- To determine which slots to probe, the hash function is extended to include the probe number (starting from 0) as a second input
- The extended hash function is:
  $h: U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- For every key k, the probe sequence
  $\langle h(k,0), h(k,1), \ldots, h(k, m-1) \rangle$
  will be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$
- Commonly used techniques to compute the probe sequences:
  - linear probing
  - quadratic probing
  - double hashing

Linear Probing

- Given an auxiliary hash function, $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, linear probing uses the hash function
  $h(k,i) = (h'(k) + i) \bmod m$, for i = 0, 1, …, m-1
- Given key k, linear probing works as follows:
  - first probe *T[h'(k)]*, *i.e.,* the slot given by the auxiliary hash function
  - next probe slot *T[h'(k)+1]*, and so on up to slot *T[m-1]*
  - then wrap around to slots *T[0]*, *T[1]*, … until we finally probe slot *T[h'(k)-1]*
- Disadvantage: primary clustering – long runs of occupied slots tend to get longer and the average search time increases

Quadratic Probing

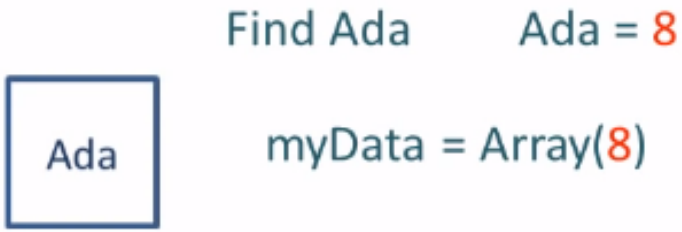
- Uses a hash function of the form
  $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$,
  where h' is an auxiliary hash function,
  c_1 and c_2 are positive auxiliary constants, and
  i=0,1,…,m-1
- The initial position probed is T[h'(k)], later positions probed are offset by amounts that depend in a quadratic manner on the probe number i
- As in linear probing, the initial probe determines the entire sequence, and so only m distinct probe sequences are used

Double hashing

- Uses a hash function of the form
  $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$,
  where h_1 and h_2 are auxiliary hash functions
- Initial probe goes to position $T[h_1(k)]$ (since i=0)
- Successive probe positions are offset from previous positions by the amount $h_2(k) \bmod m$
- Unlike linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary

Exercise

Search Ada ?



https://www.youtube.com/watch?v=KyUTuwz_b7Q&t=217s

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

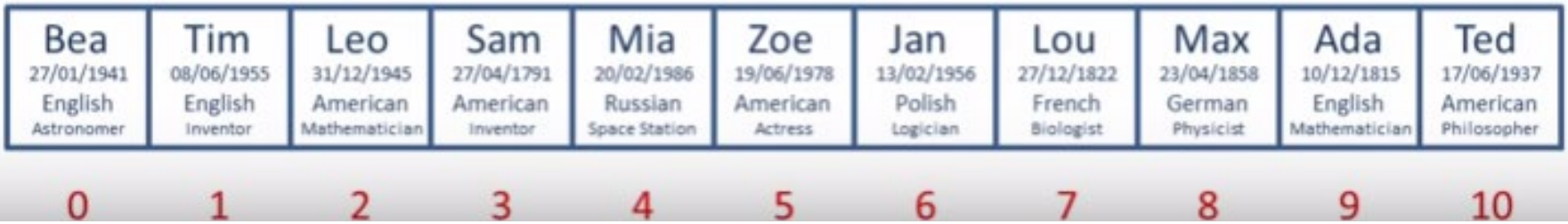
Index number = *sum ASCII codes* Mod *size of array*

Find Ada Ada = (65 + 100 + 97) = 262

Find Ada 262 Mod 11 = 9

myData = Array(9)

Hash Tables are used to store objects



Open Addressing – Linear Probing

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9

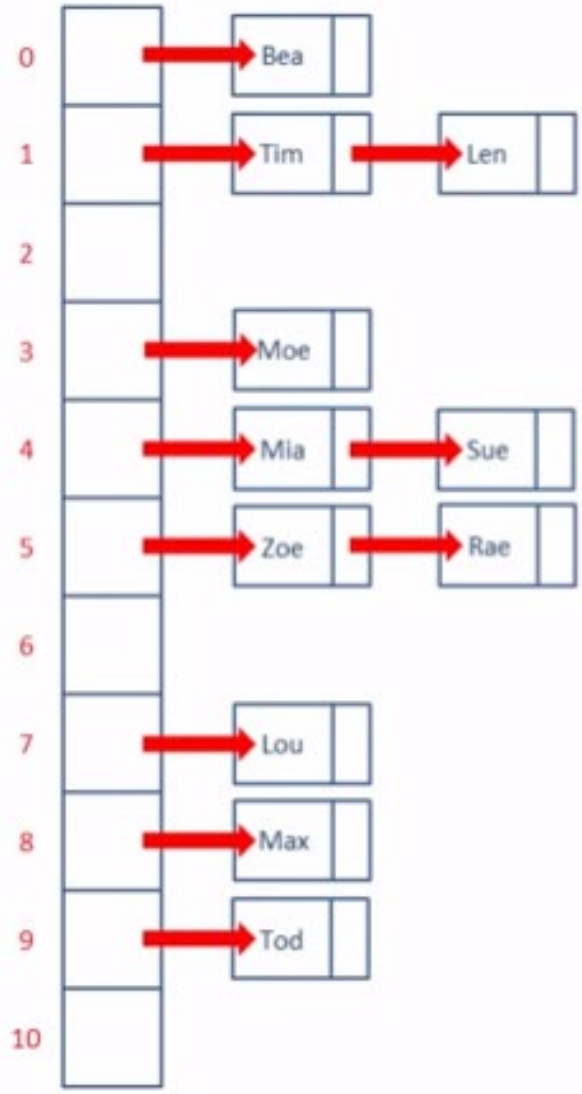
Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
0	1	2	3	4	5	6	7	8	9	10

Find Rae 280 Mod 11 = 5

Rae

myData = Array(5)

Chaining



Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9

Find Rae 280 Mod 11 = 5

myData = Array(5)

Rae