Computer Algorithms

A. Senthil Thilak

Department of Mathematical and Computational Sciences
National Institute of Technology Karnataka

Computer Algorithms

1. **Foundations of Computer Algorithms**
   - Introduction to algorithms
   - Efficiency of an algorithm
   - Order of growth and Asymptotic notations

Outline

What is an Algorithm ?

- Algorithm is a step-by-step finite sequence of instructions to solve a well-defined problem.
- Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output** in a finite amount of time.

Example

- **Problem**: Sort a sequence of numbers into a monotonically increasing order.
- **Input**: A sequence of n numbers $\{a_1, a_2, \ldots a_n\}$.
- **Output**: Sorted sequence of numbers $\{a'_1, a'_2 \ldots a'_n\}$ such that $\{a'_1 \leq a'_2 \leq \cdots \leq a'_n\}$

An algorithm is a sequence of instructions that takes the given input and produces the desired output. We call the above problem as Sorting problem and the corresponding algorithm that solves it as **Sorting algorithm**.

Outline

Efficiency of an Algorithm

How do you measure the efficiency of an Algorithm?

- **Running time** or **Computational time**: The number of primitive operations or steps executed in the instructions of an algorithm.
- Every step in the algorithm takes a certain amount of time.
- It is obvious to note that the larger the input size, the greater the running time.
- Therefore, running time is measured in terms of the size of the input. (Eg: In the case of Sorting n numbers, the running time is expressed in terms of n.)
- For instance, there is a sorting algorithm whose running time is $an^2 + bn + c$, for some constants a, b and c.

Outline

Order of growth

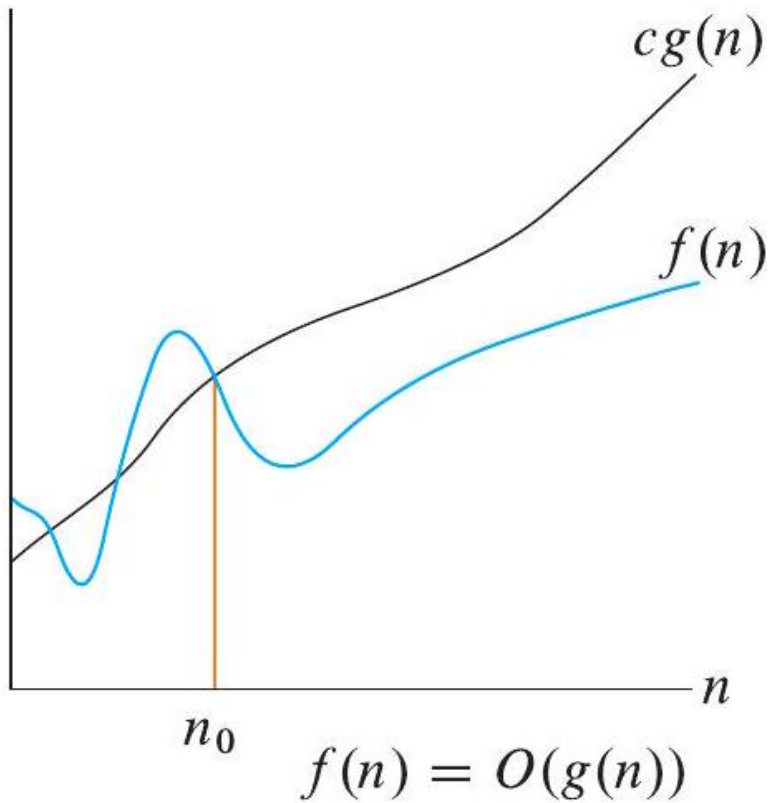
- The **order of growth** or **rate of growth** of an algorithm is an approximation of the time required to run a computer program as the input size increases.
- The order of growth ignores the constant factor needed for fixed operations and focuses instead on how the running time increases with proportional to the increase in the input size.
- If the running time of an algorithm is $an^2 + bn + c$, we consider only the leading term of a running time (an^2) since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading terms coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.
- That is, an algorithm's running time grows in proportion to the square of the input size (n^2).

Assymptotic notations

- Although we can sometimes determine the exact running time of an algorithm, the extra precision is rarely worth the effort of computing it.
- For the large enough input sizes, we study the asymptotic efficiency of algorithms.
- That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound.
- Usually, an algorithm that is asymptotically more efficient is the the best choice for all but very small inputs!!!

O-notation

- For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") is the set of functions $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq \mathrm{cg}(n)$ for all $n \geq n_0\}$



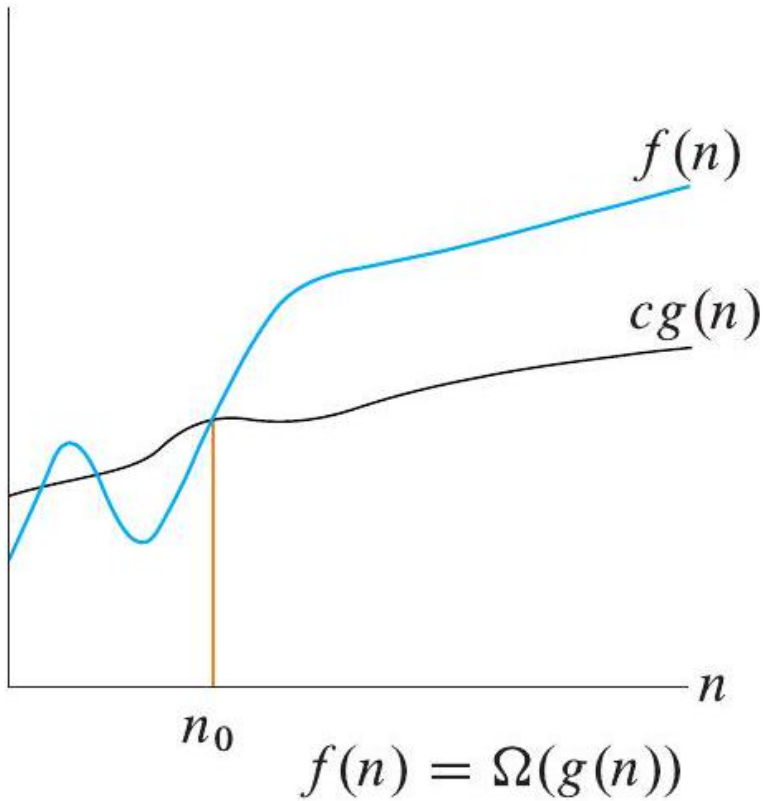
O-notation

- A function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that $f(n) \leq cg(n)$ for sufficiently large n.
- For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $\mathrm{cg}(n)$.
- O-notation describes an asymptotic upper bound. We use O-notation to give an upper bound on a function, within a constant factor.

Ω-notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

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



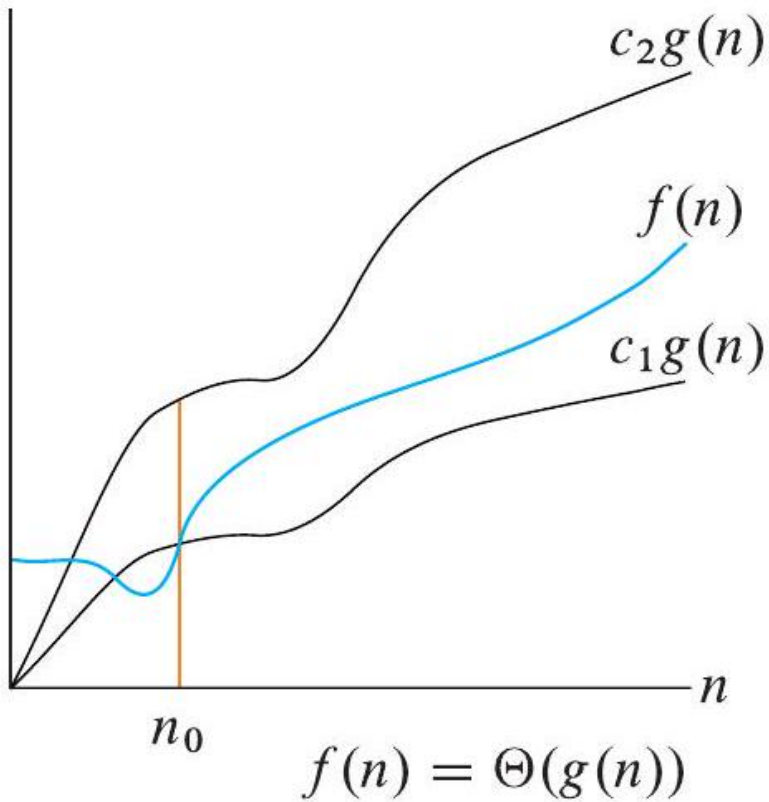
Ω-notation

- For all values n at or to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.
- Ω-notation provides an asymptotic lower bound.

Θ-notation

- For a given function $g(n)$, we denote by $\Theta(g(n))$ ("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 $n \geq n_0\}$.



Θ-notation

- For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ within constant factors.
- Θ-notation describes asymptotically tight bounds (upper and lower).
- For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

o-notation

- We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set $o(g(n)) = \{f(n) :$ for **any** positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < \mathrm{cg}(n)$ for all $n \geq n_0\}$.
- The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for some constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for all constants $c > 0$. Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n gets large:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, \text{ if it exists.}$$

ω-notation

- We use ω-notation to denote a lower bound that is not asymptotically tight. We formally define $\omega(g(n))$ ("little-omega of g of n") as the set $\omega(g(n)) = \{f(n) :$ for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq \mathrm{cg}(n) < f(n)$ for all $n \geq n_0\}$.
- The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty, \text{ if the limit exists.}$$

That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n gets larger.

Properties of Asymptotic notations

1. **Reflexivity:**

$$f(n) = O(f(n)); \qquad f(n) = \Omega(f(n)); \qquad f(n) = \Theta(f(n))$$

2. **Symmetry:**

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

3. **Transitivity:**

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \implies f(n) = O(h(n))$$

4. Does transitivity hold for Θ and Ω notations too? Yes!!!

5. **Transpose Symmetry:**

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

Some more observations on Asymptotic notations

Lemma

Let $f(n)$ *and* $g(n)$ *be two asymptotic non-negative functions. Then,* $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Lemma

For two asymptotic functions $f(n)$ *and* $g(n)$, $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$.

Some more observations on Asymptotic notations (contd...)

Remark

1. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, $c \in \mathbb{R}^+$ *then* $f(n) = \Theta(g(n))$. Characterizes tight bounds (Θ)
2. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$, $c \in \mathbb{R}$ *(c can be zero) then* $f(n) = O(g(n))$. Characterizes all upper bounds (O)
3. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *then* $f(n) = o(g(n))$ *and* $g(n) \neq o(f(n))$. Characterizes loose upper bounds (Little o)
4. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$, $c \in \mathbb{R}$ *(c can be ∞) then* $f(n) = \Omega(g(n))$. Characterizes all lower bounds (Ω)
5. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, *then* $f(n) = \omega(g(n))$ *and* $g(n) \neq \omega(f(n))$. Characterizes loose lower bounds (ω)
6. **L'Hôpital Rule:** *If $f(n)$ and $g(n)$ are both differentiable with derivates $f'(n)$ and $g'(n)$ respectively and if* $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, *then*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)},$$

whenever the limit on the right exists.

Some more observations on Asymptotic notations (contd...)

Exercise

Show that $\log n = o(\sqrt{n})$, *however* $\sqrt{(n)} \neq o(\log n)$.

Note: There are asymptotic functions which cannot be compared using any of the above notation. For example, the following two functions $f(n)$ and $g(n)$ are such that $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$:

- $f(n) = n$ and $g(n) = n^{1+\sin n}$
- $f(n) = n\cos^2(n)$ and $g(n) = n\sin^2(n)$

Data structures

- A data structure refers to a way of organizing and storing data in a computer so that it can be accessed and used efficiently.
- Using the appropriate data structure or structures is an important part of algorithm design. No single data structure works well for all purposes

Arrays

- An **array** is a collection of homogeneous data elements.
- An array is stored as a contiguous sequence of bytes in memory. If the first element of an array has index s (say $s = 1$) and the array starts at memory address a with each array element occupying b bytes, then the i^{th} element occupies bytes from $a + b(i - 1)$ to $a + ib$.

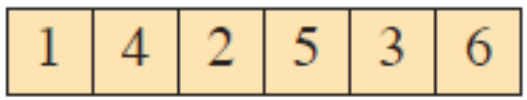


Figure 1: Array A

$A[1] = 1$, $A[2] = 4$, and so on. (Note that indexing may start from 0 depending on convenience. That is $A[0] = 1$, $A[1] = 4$).

Matrix

- A matrix is a two-dimensional array.

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

$M[i][j]$ represents an element in i^{th} row and j^{th} column. (Eg: $M[1][2] = 2$)

Stacks and Queues

- Dynamic sets like stacks and queues allow insertion and deletion of elements in a specific order at any time.
- In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or $\boldsymbol{LIFO}$, policy.
- Similarly, in a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or $\boldsymbol{FIFO}$, policy.
- In a stack, the INSERT operation is often called **PUSH**, and the DELETE operation, which does not take an element argument, is often called **POP**.
- In a Queue, the INSERT operation is often called **ENQUEUE**, and the DELETE operation, which does not take an element argument, is often called **DEQUEUE**.

Implementation of stacks

- For a given stack S, $S.top$ holds the address of the topmost element(Initially, the topmost element is 9 and its address is stored in $S.top$. That is $S.top = 4$.
- PUSH$(S, 17)$, PUSH$(S, 3)$ operations are performed. $(S.top = 6)$.
- POP(S) is executed.$(S.top = 5)$

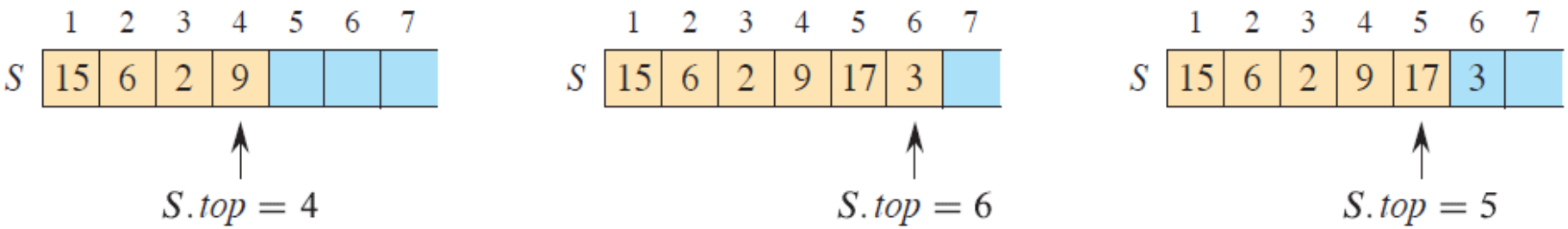


Figure 2: Array implementation of Stack

Pseudocode

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

PUSH(S, x)
1  if S.top == S.size
2      error "overflow"
3  else S.top = S.top + 1
4      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]
```

Figure 3: Pseudocode to implement Stacks

Implementation of Queues

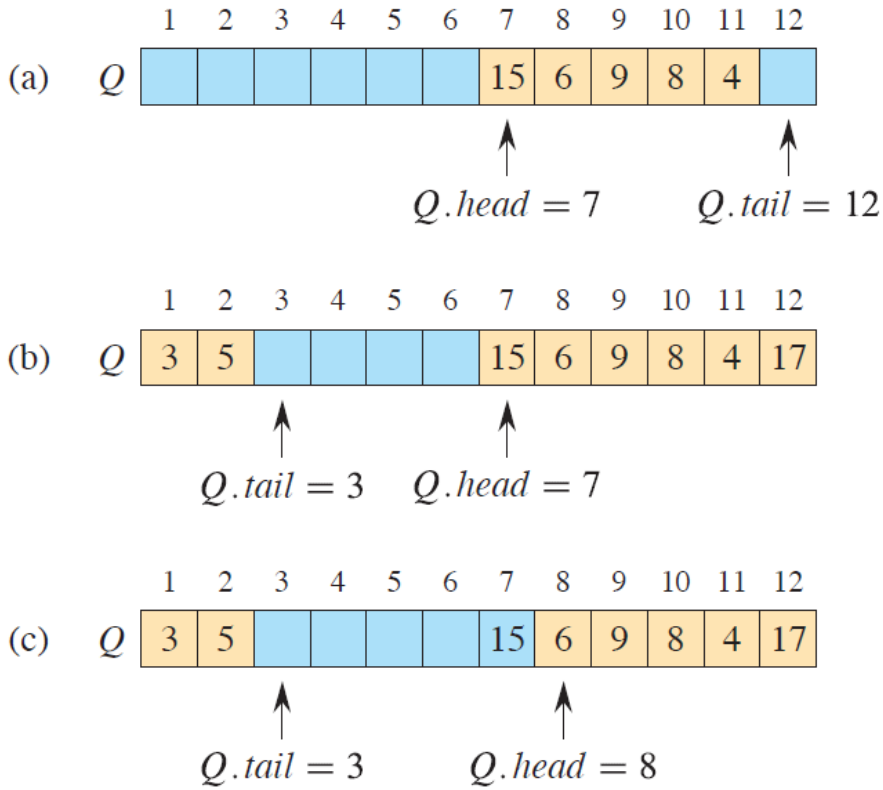
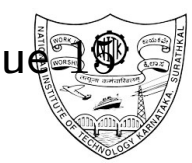


Figure 4: **(a)** The queue has 5 elements, in locations Q [7:11].
(b) The configuration of the queue after the calls $\text{Enqueue}(Q, 17), \text{Enqueue}(Q, 3)$, and $\text{Enqueue}(Q, 5)$.
(c) The configuration of the queue after the call $\text{DeQueue}(Q)$ returns the key value formerly at the head of the queue. The new head has key 6.



Pseudocode

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

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

Figure 5: Pseudocode to implement Queues

Linked Lists

- A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.
- Each element of a doubly linked list L is an object with an attribute key and two pointer attributes: next and prev containing the address of next and previous elements.

Implementation of Linked List

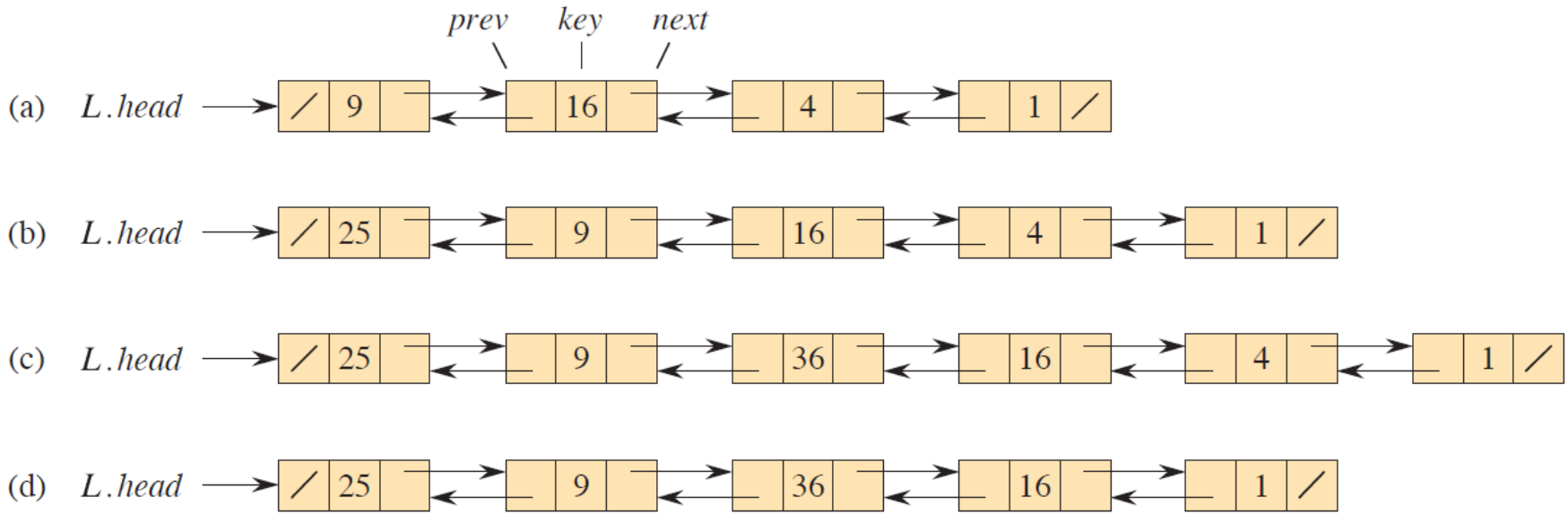


Figure 6: (a)Doubly linked list with each object(node) having attributes prev and next(pointers) and a key(value).
(b) Following the execution of List-PREPEND (L, x), where $x.key = 25$.
(c) The result of calling List-InSERT(x, y), where $x.key = 36$ and y points to the object with key 9.
(d) The result of the subsequent call List-Delete (L, x), where x points to the object with key 4.

Trees

- A tree is a hierarchical representation of a finite set of one or more data items
- There is a special node called the root of the tree and corresponding trees are called rooted trees.
- Linked lists are used to represent and implement trees.

Rooted tree

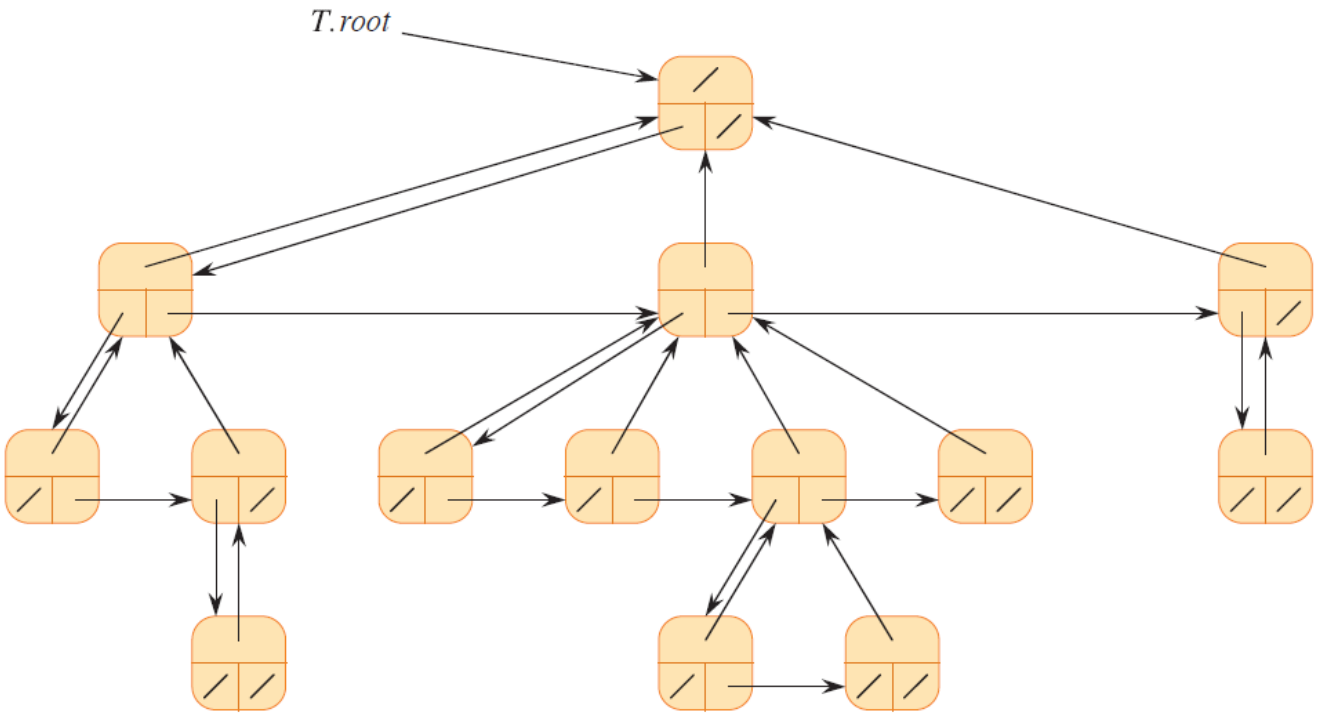


Figure 7: Rooted tree

Selection Sort

- The given list of numbers is divided into two sublists, *sorted* and *unsorted*.
- It finds the smallest element in the unsorted list and swaps the smallest element with the first element.
- Process repeats. That is, It finds the smallest element in the list starting from the second element

Selection Sort

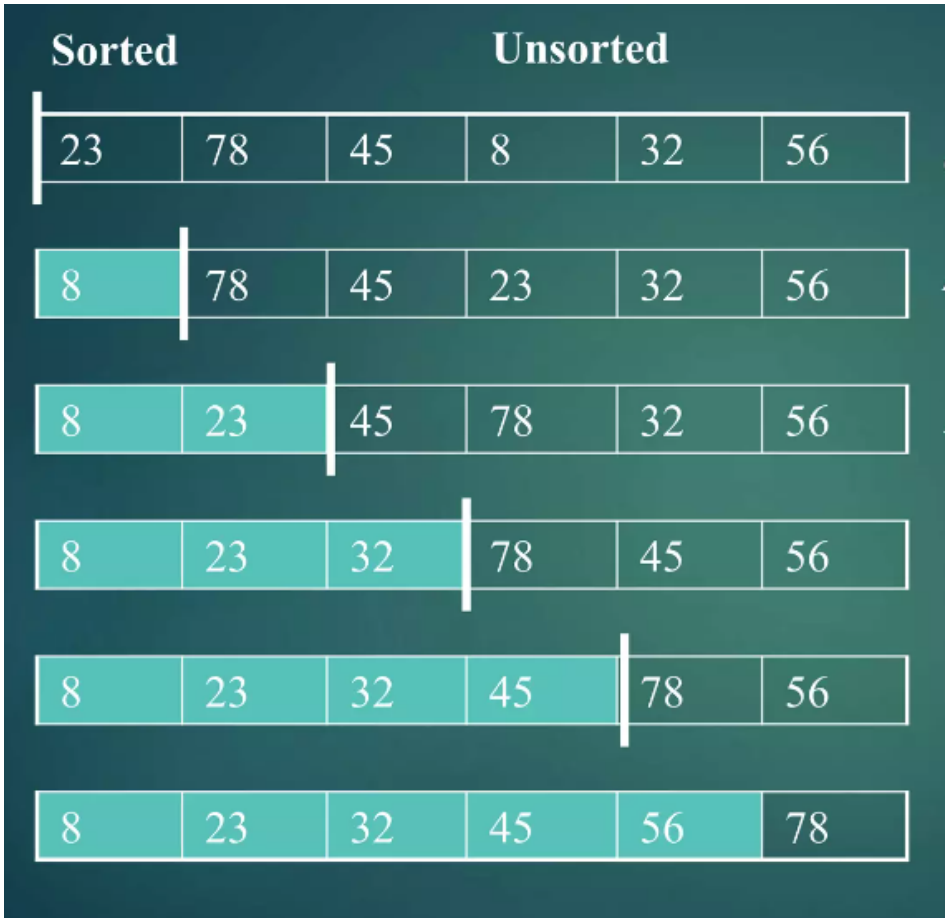


Figure 8: Selection Sort

Selection sort

```
public static void selectionSort(int[] list)
  { int min;
    int temp;
      for(int i = 0; i < list.length - 1; i++) {
          min = i;
          for(int j = i + 1; j < list.length; j++)
          if( list[j] < list[min] )
                min = j;
            temp = list[i];
            list[i] = list[min];
            list[min] = temp;
        }
  }
```

The average case time complexity Selection sort is $O(n^2)$.

Insertion Sort

- Let A be an unsorted array.
- Algorithm starts from the second element($A[2]$) and checks whether all the elements to the left of it are lesser than or equal to it.
- If not, elements are swapped to make sure elements to the left of it have lesser or equal value. This process repeats for each of the elements (from $A[2]$ to $A[n]$).

Insertion Sort

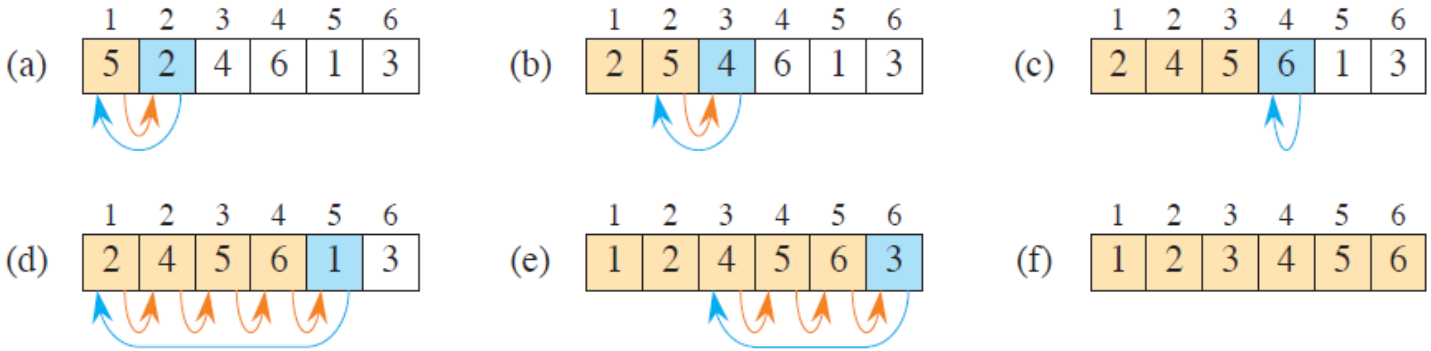


Figure 9: Insertion sort

Insertion Sort

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Figure 10: Insertion sort

The average case time complexity of the Insertion sort is $O(n^2)$.

Heapsort

Terminologies

- **Heap:** A *heap* is a binary tree with the data stored in the nodes. It has two important properties:
  - (i) **Balance/Shape property:** It is as much like a complete binary tree as possible. "Missing leaves", if any, are on the last level at the far right.
  - (ii) **Structure/Heap property:** The value (key) in each parent is greater than or equal to (for Max-heap) (or less than or equal to (for Min-heap)) the values at each of its children.

Overview of Heapsort

Heapsort is a sorting algorithm that organizes elements in an unsorted array into a binary heap by repeatedly moving the largest element from the heap and inserting it into the array being sorted.

The heapsort algorithm works in six major steps:

1. Transform the unsorted array into a binary tree by inserting each element as a node in a breadth-first manner.
2. Convert the binary tree into a Max-heap.
3. Swap the root node (largest element) with the last element in the heap.
4. Call *heapify*() function to restore the max-heap property.
5. Repeat steps 3 and 4 until the heap is sorted and exclude the last element from the heap on each iteration.
6. After each swap and *heapify*() call, ensure that the max heap property is satisfied.

Principle: Heapsort works based on *selection principle* and so *in-place*[1], but not *stable*[2].

[1] An *in-place* sorting needs no extra array.

[2] A *stable* sorting preserves the relative order of records with equal keys.

The Heapsort algorithm

Algorithm HeapSort(A)
1. BuildHeap(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. Swap($A[1], A[i]$)
4. $heapsize[A] \leftarrow heapsize[A] - 1$
5. Heapify($A, 1$)

Procedure BuildHeap(A)
1. $heapsize[A] \leftarrow length[A]$
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3. Heapify(A, i)

Procedure Heapify(A, i)
1. $l \leftarrow$ Left$[i]$,
   $r \leftarrow$ Right$[i]$
2. **if** $l \leq heapsize[A] \wedge A[l] > A[i]$
3. $largest \leftarrow l$
4. **else** $largest \leftarrow i$
5. **if** $r \leq heapsize[A] \wedge A[r] > A[largest]$
6. $largest \leftarrow r$
7. **if** $largest \neq i$
8. Swap($A[i], A[largest]$)
9. Heapify($A, largest$)

- *BUILDHEAP(A)* $\rightarrow$ Constructs the heap from $A[1 \ldots n]$ (STEP 1).
- Uses two variables: *length*$[A] = n$ (remains unchanged) and *heapsize*$[A]$, which is decremented as the heap is iteratively reduced in size (STEP 2), while swapping the root of (reducing) heap with its last node (STEP 3) and then deleting the last node (current max; STEP 4) so as to grow the sorted sublist at the rear end of A.
- *HEAPIFY*(A, i) $\rightarrow$ Returns a binary tree of A rooted at i satisfying the max-heap property. That is, it is used to rebuild the heap (STEP 5), which has lost is *heap property* due to the aforesaid swapping. Identifies the largest of $A[i]$, $A[LEFT[i]]$, $A[RIGHT[i]]$ and its index is stored in largest.

The divide-and-conquer approach - A short overview

Many useful algorithms are *recursive* in structure. That is, to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. Such algorithms are said to follow a **divide-and-conquer** approach - break the problem into several subproblems that are similar to the original problem but smaller in size, sove the subproblems recursively and then combine these solutions to obtain a solution to the original problem.

The divide-and-conquer (DandC) paradigm

The divide-and-conquer approach works in 3 steps at each level of the recursion:

1. **Divide:** Break the problem into a number of subproblems that are similar to the original one, but smaller in size.
2. **Conquer:** Conquer the subproblems by calling them recursively. If the subproblem sizes are small enough, just solve them in a straightforward manner.
3. **Combine:** Combine the solutions of subproblems and generate the solution of the original problem

Quick sort - Using DandC approach

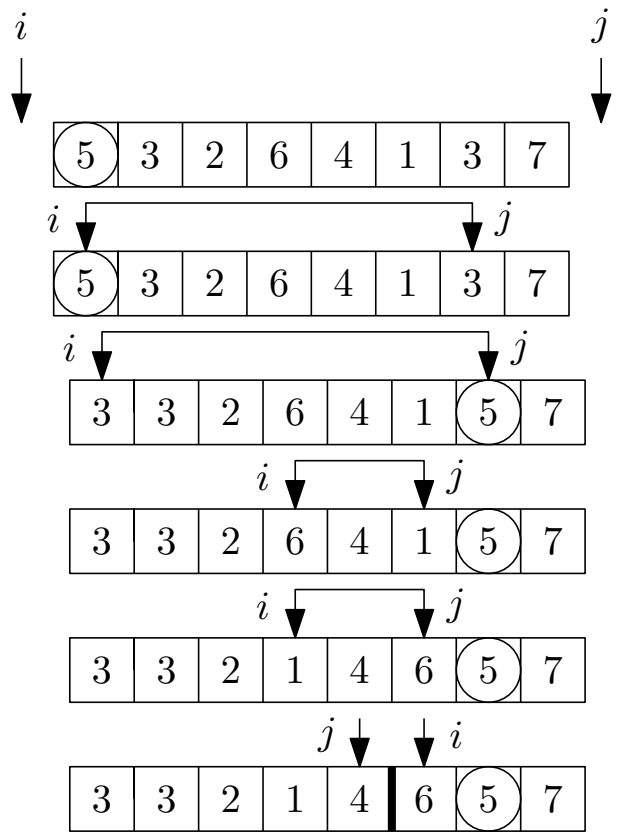
Principle: Works based on *divide-and-conquer*, *in-pace*, but not *stable*.

- **Divide:** The input list $L[1 \ldots n]$ is partitioned at an index $\mathbf{q}$ into two non-empty sublists, $L_1 := L[1 \ldots q]$ and $L_2 := L[q+1 \ldots n]$, such that each element of L_1 is **less than or equal to** every element of L_2. The index $\mathbf{q}$ is returned by the **PARTITION** procedure.
- **Conquer:** Conquer by calling quicksort recursively to sort each of the sublists/subarrays.
- **Combine:** Combine the two sorted subarrays to get the required array.

Quick Sort

Algorithm QuickSort(L, p, r)
1. **if** $p < r$
2. $\quad q \leftarrow$ Partition(L, p, r)
3. $\quad$ QuickSort(L, p, q)
4. $\quad$ QuickSort$(L, q + 1, r)$

Procedure Partition(L, p, r)
1. $x \leftarrow L[p]$
2. $i \leftarrow p - 1$
3. $j \leftarrow r + 1$
4. **while** True
5. $\quad$ **do** $j \leftarrow j - 1$
6. $\quad\quad$ **till** $L[j] > x$
7. $\quad$ **do** $i \leftarrow i + 1$
8. $\quad\quad$ **till** $L[i] < x$
9. $\quad$ **if** $i < j$
10. $\quad\quad$ Swap$(L[i], L[j])$
11. $\quad$ **else return** j



Correctness and Complexity Analysis I

CORRECTNESS: (Can be proved by induction)

- **PARTITION procedure:** Based on the following facts:

$$(i)\ |L_1| > 0 \text{ and } |L_2| > 0;\ (ii)\ L_1 \cup L_2 = L;\ (iii)\ L_1 \leq x \leq L_2.$$

- **Sorting procedure:** Follows from inductive reasoning. For one element, the algorithm leaves the data unchanged. Otherwise it produces the concatenation of L_1 and L_2, which are themselves recursively sorted by the inductive hypothesis.

TIME COMPLEXITY:

- **Best case:** Occurs when the list is partitioned at its median. So, the recursive rule for time complexity is,

$$T(n) = 2T(n/2) + \Theta(n),$$

which solves to

$$T(n) = \Theta(n \log n).$$

- **Worst case:** Arises when the list is already sorted and $\max(|L_1|, |L_2|) = n - 1$ in every step of the recursion, leading to the recurrence relation

$$T(n) = T(n-1) + \Theta(n),$$

which solves to

$$T(n) = \Theta(n^2).$$

- Not better than *Insertion sort*!
- Takes $\Theta(n^2)$ time for a sorted list, in which case *insertion sort* runs in $O(n)$ time.

Correctness and Complexity Analysis II

- **Average case:** Based on the assumption that the pivot x is equally likely to the i^{th} min for $i \in \{1, 2 \ldots n\}$. If x is the 1^{st} min, then x is the sole element in L_1 so that $|L_1| > 0$; for all other cases, $x \in L_2$. So, $|L_1| = 1$ occurs when x is the 1^{st} or 2^{nd} min. Thus, $Prob(|L_1| = 1) = 2/n$ and $Prob(|L_1| = q) = 1/n$, for each q with $2 \leq q \leq n-1$. Hence, the average-case time complexity of quicksort is

$$
\begin{aligned}
T(n) &= \frac{1}{n}\left((T(1) + T(n-1)) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n) \qquad (2.1)\\
&\leq \frac{1}{n}\left(O(n^2) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n)\\
&= \frac{1}{n}\left(\sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n), \quad \text{as } \tfrac{1}{n}O(n^2) = O(n) \text{ and } O(n) + \Theta(n) = \Theta(n)\\
&\leq \frac{2}{n}\sum_{q=1}^{n-1} T(q) + \Theta(n) \qquad (2.2)
\end{aligned}
$$

We solve the above eequation using the method of substitution (induction), with the hypothesis that $R(q) \leq aq \log q$ for a suitable constant $a > 0$ and $\forall\ a < n$. Then, we get

Correctness and Complexity Analysis III

$$
\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{q=1}^{n-1} aq\log q + \Theta(n) \quad = \frac{2a}{n}\sum_{q=1}^{n-1}(q\log q) + \Theta(n) \\
&= \frac{2a}{n}\left(\sum_{q=1}^{\lceil n/2\rceil-1}(q\log q) + \sum_{q=\lceil n/2\rceil}^{n-1}(q\log q)\right) + \Theta(n) \\
&\leq \frac{2a}{n}\left(\log(n/2)\sum_{q=1}^{\lceil n/2\rceil-1} q \;+\; \log n\sum_{q=\lceil n/2\rceil}^{n-1} q\right) + \Theta(n) \\
&= \frac{2a}{n}\left((\log n-1)\sum_{q=1}^{\lceil n/2\rceil-1} q \;+\; \log n\sum_{q=\lceil n/2\rceil}^{n-1} q\right) + \Theta(n) \\
&= \frac{2a}{n}\left(\log n\sum_{q=1}^{n-1} q \;-\; \sum_{q=1}^{\lceil n/2\rceil-1} q\right) + \Theta(n)
\end{aligned}
$$

Correctness and Complexity Analysis IV

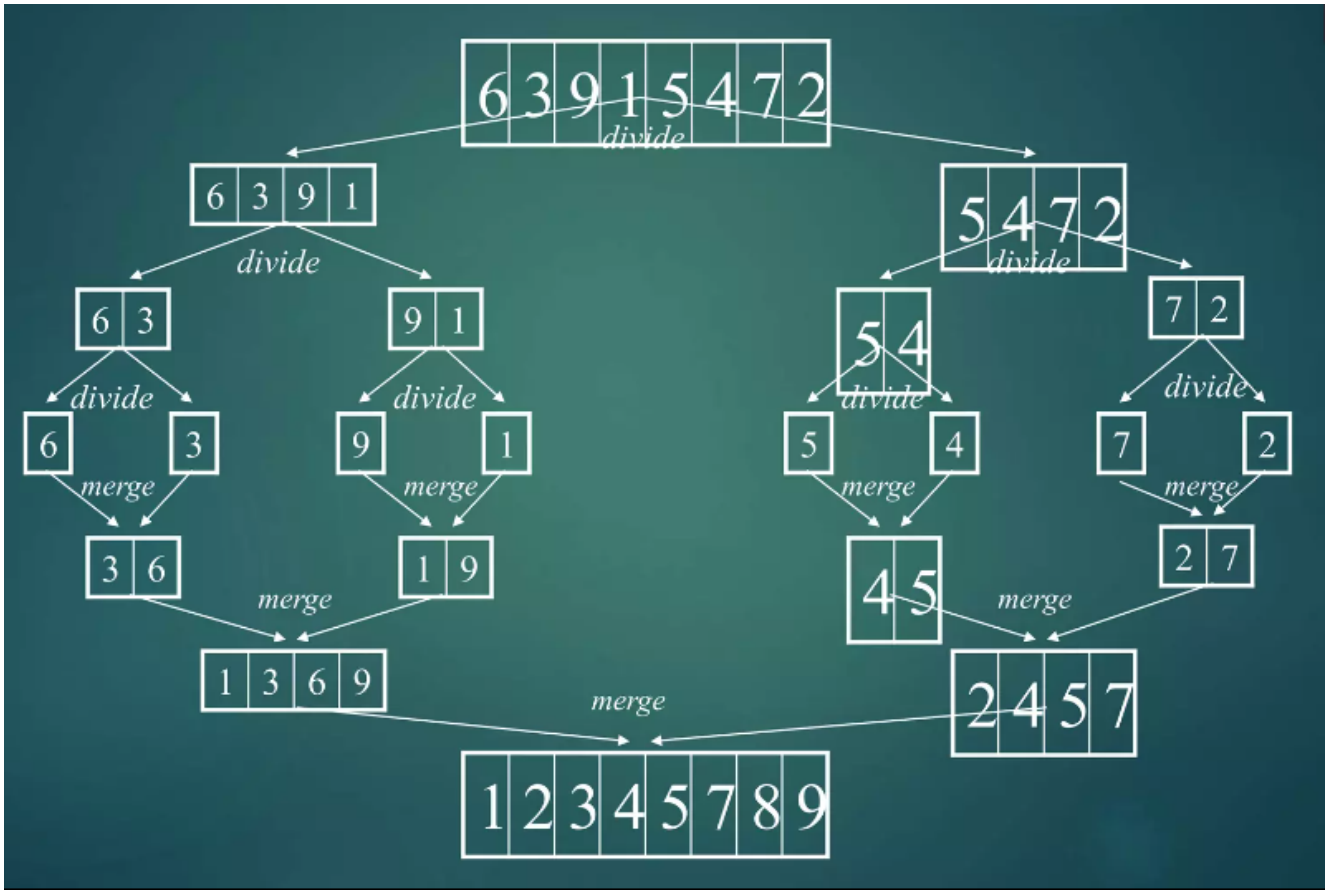
$$
\begin{aligned}
or, \quad T(n) &\leq \frac{2a}{n}\left(\frac{1}{2}n(n-1)\log n \; - \; \frac{1}{2}\left(\left\lceil\frac{n}{2}\right\rceil - 1\right)\left\lceil\frac{n}{2}\right\rceil\right) + \Theta(n) \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n(n-1)\log n \; - \; \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2}\right) + \Theta(n) \\
&= a(n-1)\log n - \frac{a}{2}\left(\frac{n}{2} - 1\right) + \Theta(n) \\
&= an\log n - \left(\frac{a}{2}\left(\frac{n}{2} - 1\right) + a\log n - \Theta(n)\right),
\end{aligned}
$$

which can be made smaller than $an\log n$ for a sufficiently large value of a, $\forall\ n \geq n_0$. Thus, $T(n) = O(n\log n)$.

Merge Sort

- Merge sort applies the divide and conquer method.
  - It divides or partitions the list into two subarrays each of half the size.
  - Conquer by sorting each subarray by calling mergesort recursively.
  - Sort and merge the final two subarrays

Merge Sort


6 3 9 1 5 4 7 2
divide
6 3 9 1
5 4 7 2
divide
divide
6 3
9 1
5 4
7 2
divide
divide
divide
divide
6
3
9
1
5
4
7
2
merge
merge
merge
merge
3 6
1 9
4 5
2 7
merge
merge
1 3 6 9
2 4 5 7
merge
1 2 3 4 5 7 8 9

Merge Sort

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

The average case time complexity of Merge Sort is $O(n * logn)$

Questions?