










Sorting Algorithms

Number of compares of common sorting algorithms

ALGORITHM	CODE	IN PLACE	STABLE	BEST	AVERAGE	WORST	REMARKS
selection sort	Selection.java 	✓		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges; quadratic in best case
insertion sort	Insertion.java 	✓	✓	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small or partially-sorted arrays
bubble sort	Bubble.java 	✓	✓	n	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	rarely useful; dominated by insertion sort
shellsort	Shell.java 	✓		$n \log_3 n$	unknown	$c n^{3/2}$	tight code; subquadratic
mergesort	Merge.java 		✓	$\frac{1}{2} n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$n \log n$ guarantee; stable
quicksort	Quick.java 	✓		$n \log_2 n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
heapsort	Heap.java 	✓		n^\dagger	$2 n \log_2 n$	$2 n \log_2 n$	$n \log n$ guarantee; in place

$^\dagger n \log_2 n$ if all keys are distinct

It includes leading constants but ignores lower-order terms

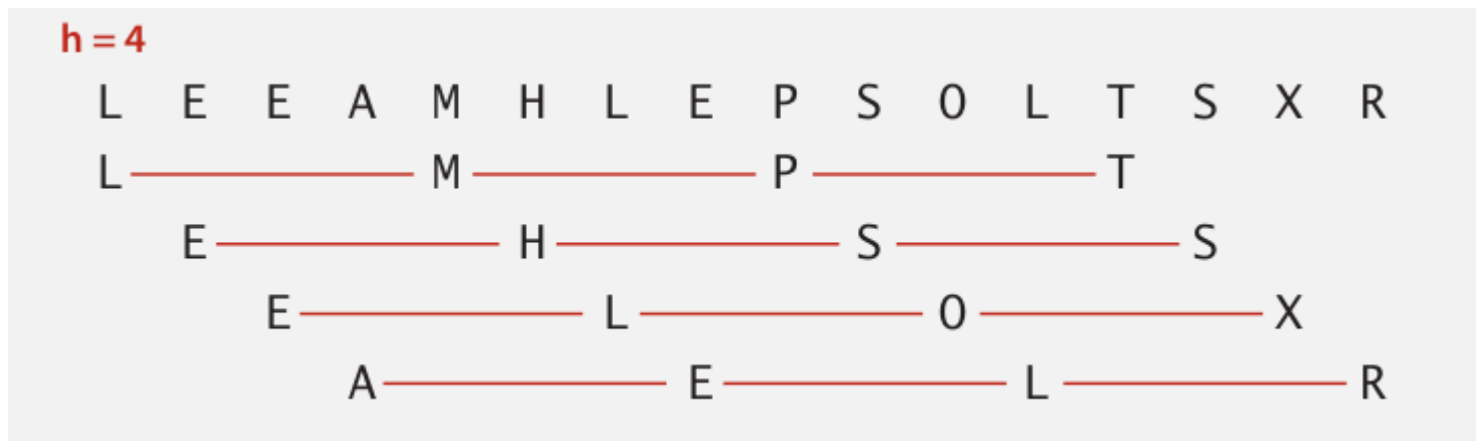
Insertion Sort

```
def insertion_sort(arr):  
    for k in range(1, len(arr)): # from 1 to n-1  
        cur = arr[k] # current element to be inserted  
        j = k # find correct index j for current  
        while j > 0 and arr[j-1] > cur: # element A[j-1] must be after current  
            arr[j] = arr[j-1]  
            j -= 1  
        arr[j] = cur #final position for the current value  
    return arr
```

Shell sort

Idea: Move entries more than one position at a time by h-sorting the array.

An h-sorted array is nothing but h-interleaved sorted sub-sequences



Shellsort: h-sort array for decreasing sequence of values of h

Shell sort - example

arr = [72, 50, 48, 41, 33, 23, 20, 19, 18, 16, 14, 13, 11, 9, 6, 5, 1]

For h = 13, [9, 6, 5, 1, 33, 23, 20, 19, 18, 16, 14, 13, 11, 72, 50, 48, 41]

For h = 4, [9, 6, 5, 1, 11, 16, 14, 13, 18, 23, 20, 19, 33, 72, 50, 48, 41]

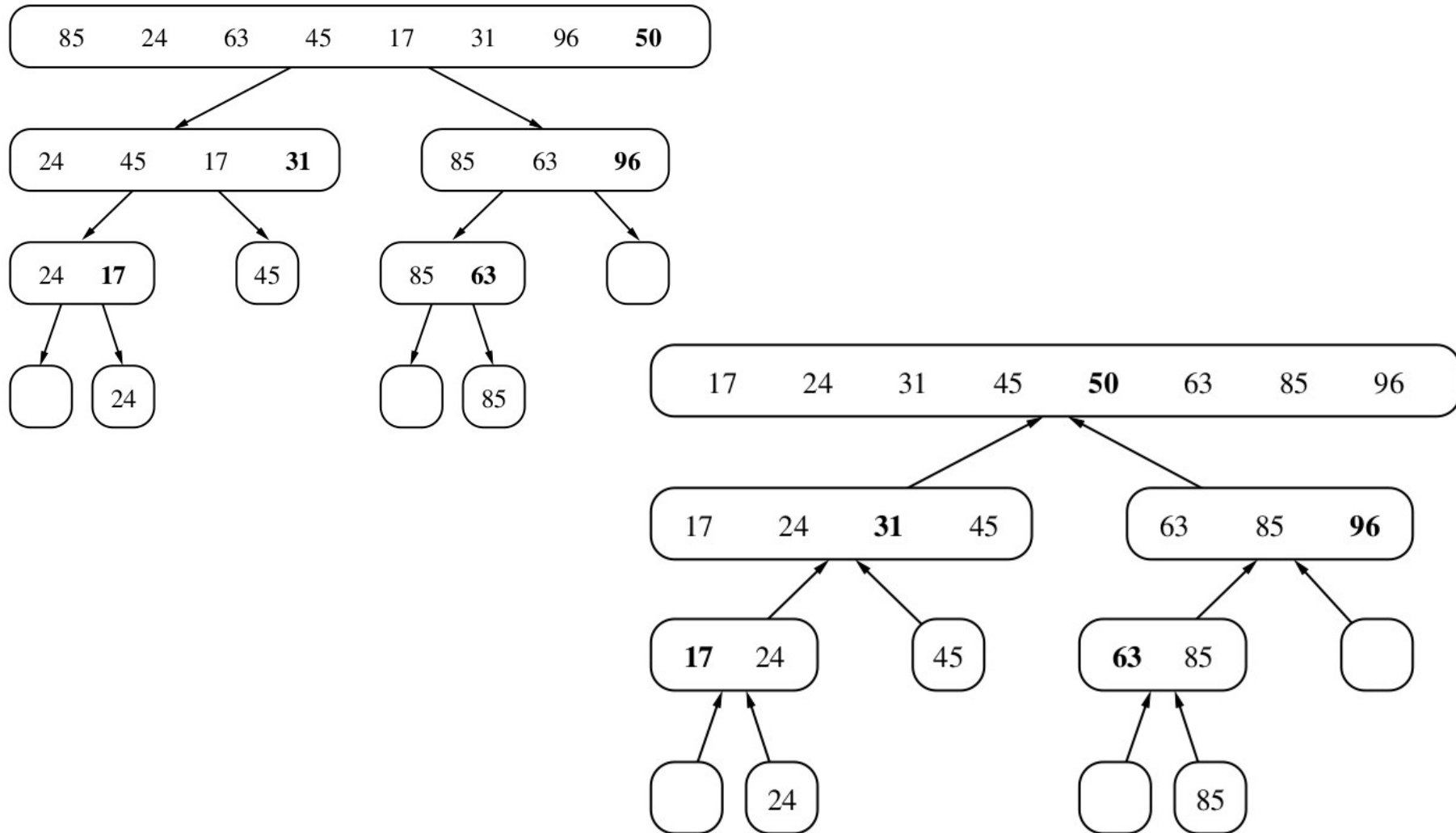
For h = 1, [1, 5, 6, 9, 11, 13, 14, 16, 18, 19, 20, 23, 33, 41, 48, 50, 72]

Shell sort

```
def shell_sort(arr):  
    # Choose an appropriate increment sequence  
    h = 1  
    while(h < len(arr)/3):  
        h = 3*h + 1  
  
    while h > 0:  
        for i in range(h, len(arr)):  
            current = arr[i]  
            j = i - h  
            while j >= 0 and arr[j] > current:  
                arr[j + h] = arr[j]  
                j -= h  
            arr[j + h] = current  
  
        h //= 3 # Reduce the increment size  
  
    return arr
```

```
def insertion_sort(arr):  
    for k in range(1, len(arr)): # for each element  
        cur = arr[k] # current element to be inserted  
        j = k # find correct index j for current element  
        while j > 0 and arr[j-1] > cur:  
            arr[j] = arr[j-1]  
            j -= 1  
        arr[j] = cur  
  
    return arr
```

Quicksort



Quicksort – A divide & conquer approach

1. Divide by partitioning the array into two (possibly empty) subarrays based on a pivot. Compute the index of the pivot as part of this partitioning procedure.
2. Conquer by calling quicksort recursively to sort each of the subarrays
3. Combine by doing nothing: because the two subarrays are already sorted, no work is needed to combine them.

Quicksort – Pseudo code

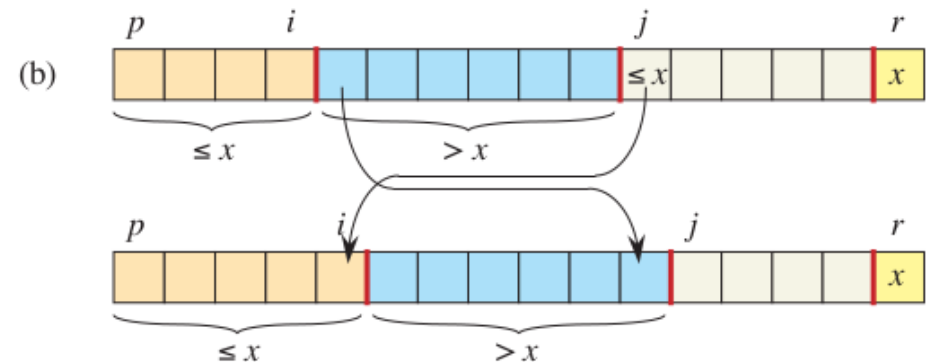
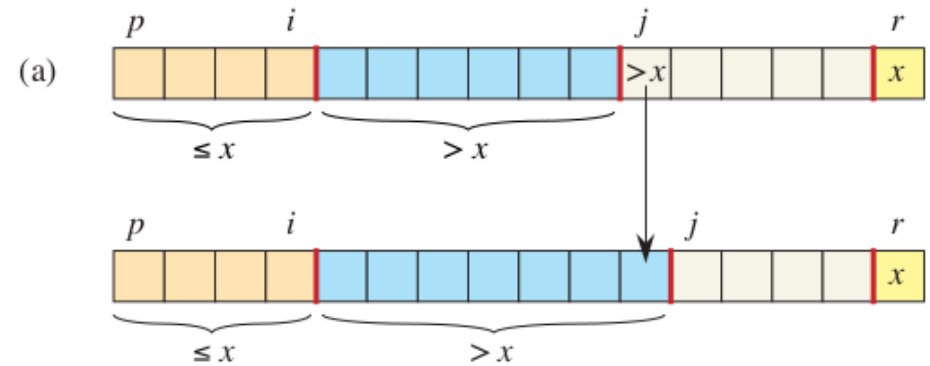
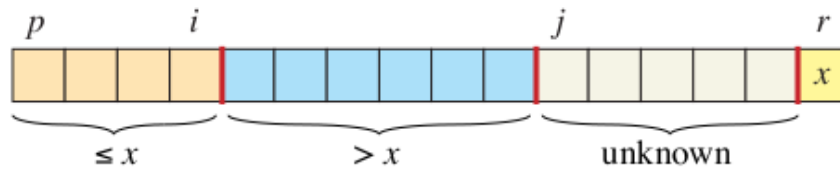
QUICKSORT(A, p, r)

```
1  if  $p < r$ 
2      // Partition the subarray around the pivot, which ends up in  $A[q]$ .
3       $q = \text{PARTITION}(A, p, r)$ 
4      QUICKSORT( $A, p, q - 1$ ) // recursively sort the low side
5      QUICKSORT( $A, q + 1, r$ ) // recursively sort the high side
```

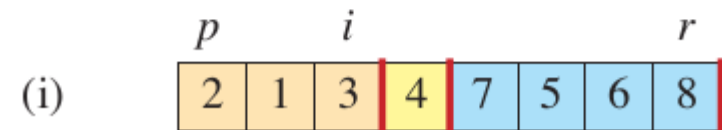
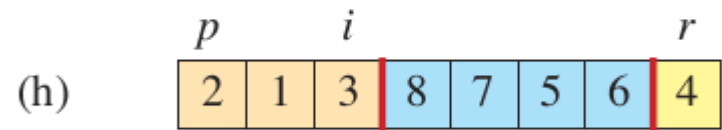
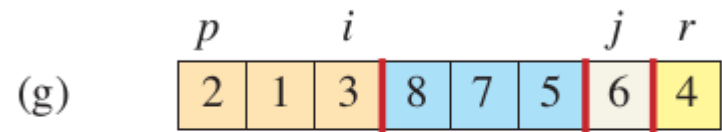
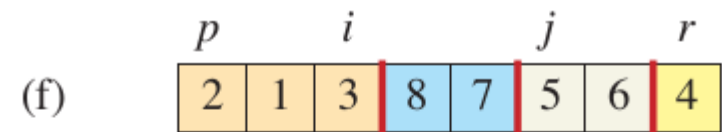
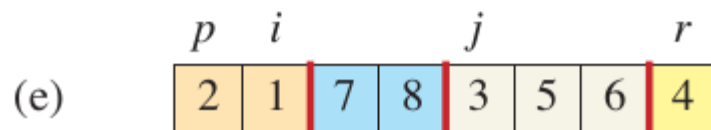
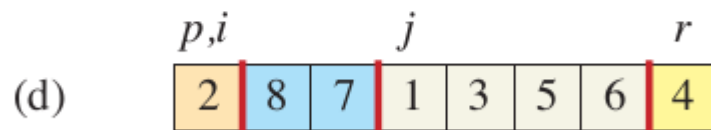
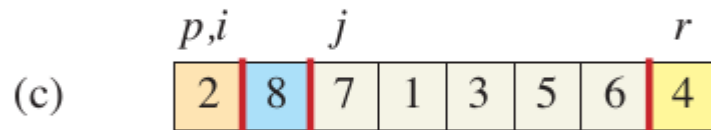
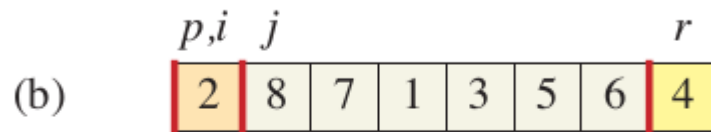
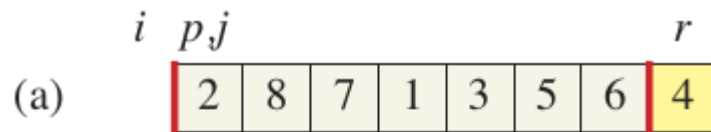
PARTITION(A, p, r)

```
1   $x = A[r]$  // the pivot
2   $i = p - 1$  // highest index into the low side
3  for  $j = p$  to  $r - 1$  // process each element other than the pivot
4      if  $A[j] \leq x$  // does this element belong on the low side?
5           $i = i + 1$  // index of a new slot in the low side
6          exchange  $A[i]$  with  $A[j]$  // put this element there
7  exchange  $A[i + 1]$  with  $A[r]$  // pivot goes just to the right of the low side
8  return  $i + 1$  // new index of the pivot
```

Quicksort – more details



Quicksort – more details



Performance of quicksort

Worst-case partitioning:
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Best-case partitioning:

produces two sub problems, each of size no more than $n/2$,

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

Randomised quicksort

RANDOMIZED-PARTITION(A, p, r)

```
1  $i = \text{RANDOM}(p, r)$   
2 exchange  $A[r]$  with  $A[i]$   
3 return PARTITION( $A, p, r$ )
```

RANDOMIZED-QUICKSORT(A, p, r)

```
1 if  $p < r$   
2      $q = \text{RANDOMIZED-PARTITION}(A, p, r)$   
3     RANDOMIZED-QUICKSORT( $A, p, q - 1$ )  
4     RANDOMIZED-QUICKSORT( $A, q + 1, r$ )
```

Lower Bound for Sorting

The running time of any comparison-based algorithm for sorting an n -element sequence is $\Omega(n \log n)$ in the worst case.

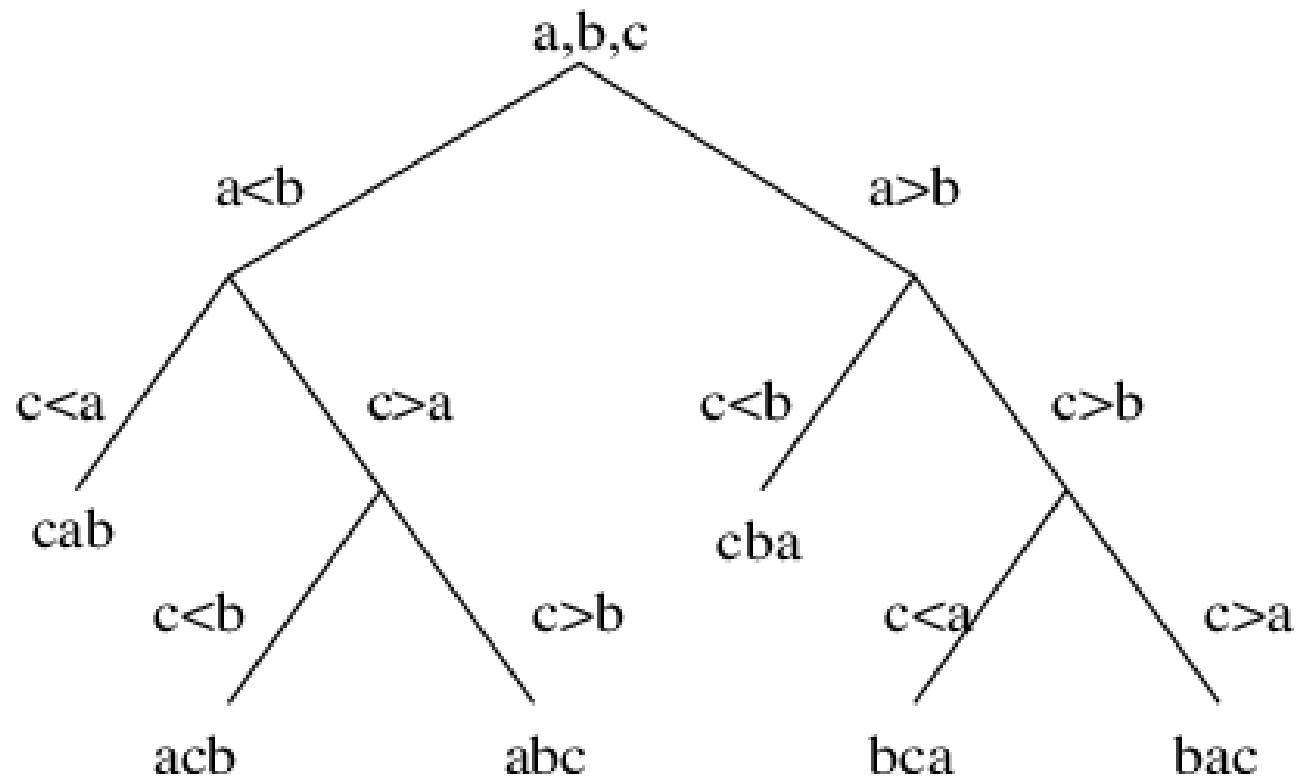
W.K.T, a sorting algorithm essentially outputs a permutation of the input

Let $S = (x_0, x_1, \dots, x_{n-1})$ be an input sequence to be sorted

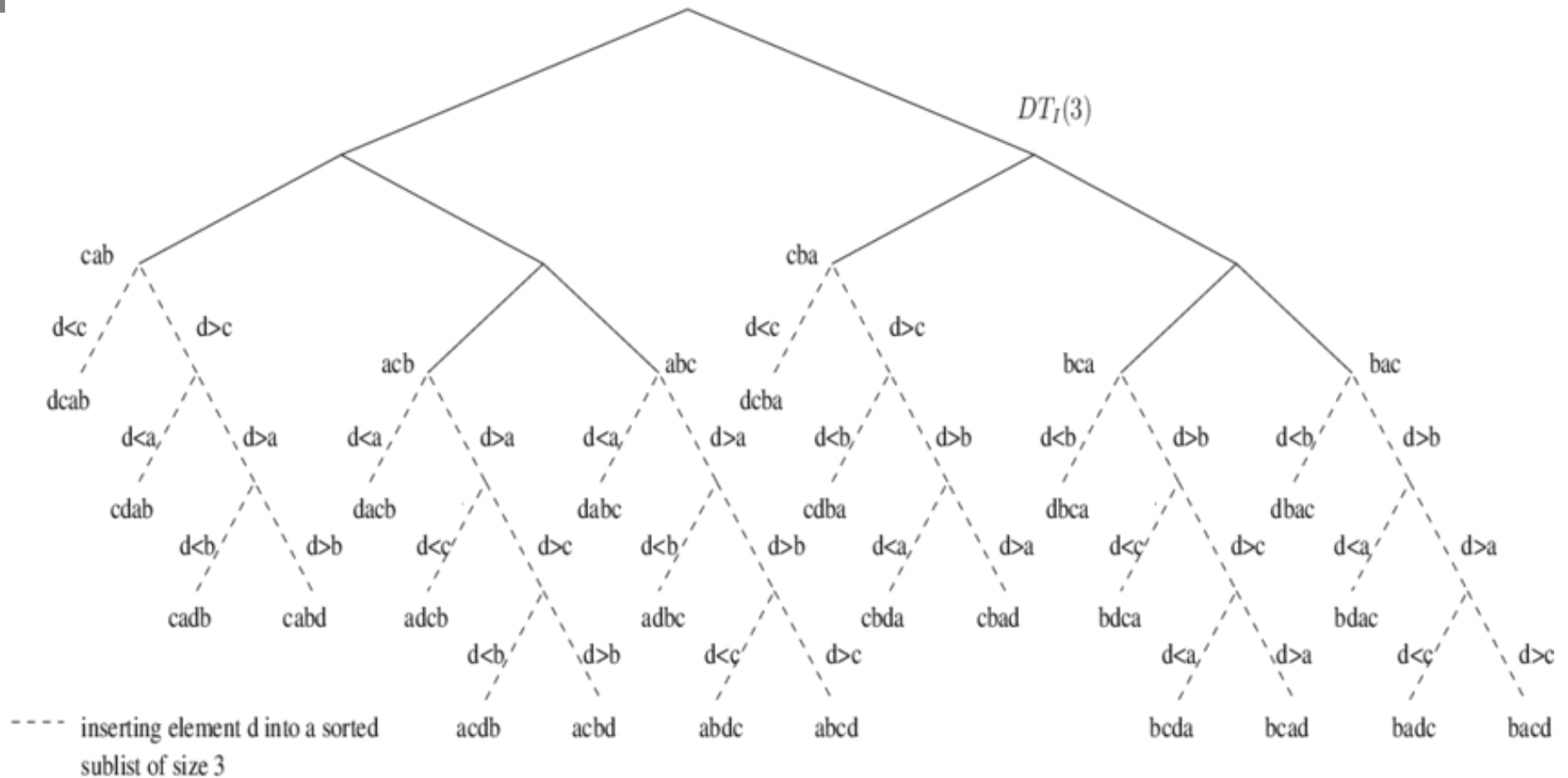
Key observations:

(i) there are $n!$ different possible permutations the algorithm might output

Lower Bound for Sorting



Lower Bound for Sorting



Lower Bound for Sorting

(ii) for each of these permutations, there exists an input for which that permutation is the only correct answer.

e.g., $[x_3, x_2, x_0, x_1]$ is the only correct answer for sorting the input $[7, 8, 4, 3]$

ie. we can fix some set of $n!$ inputs (e.g., all orderings of $\{1, 2, \dots, n\}$), one for each of the $n!$ output permutations.

Lower Bound for Sorting

Each external node in T must be associated with one permutation of S . Moreover, each permutation of S must result in a different external node of T .

Thus, T must have at least $n!$ external nodes. For a proper binary tree having n nodes, we know that $\log(n+1) - 1 \leq h \leq (n-1)/2$, the height of T is at least $\log(n!)$

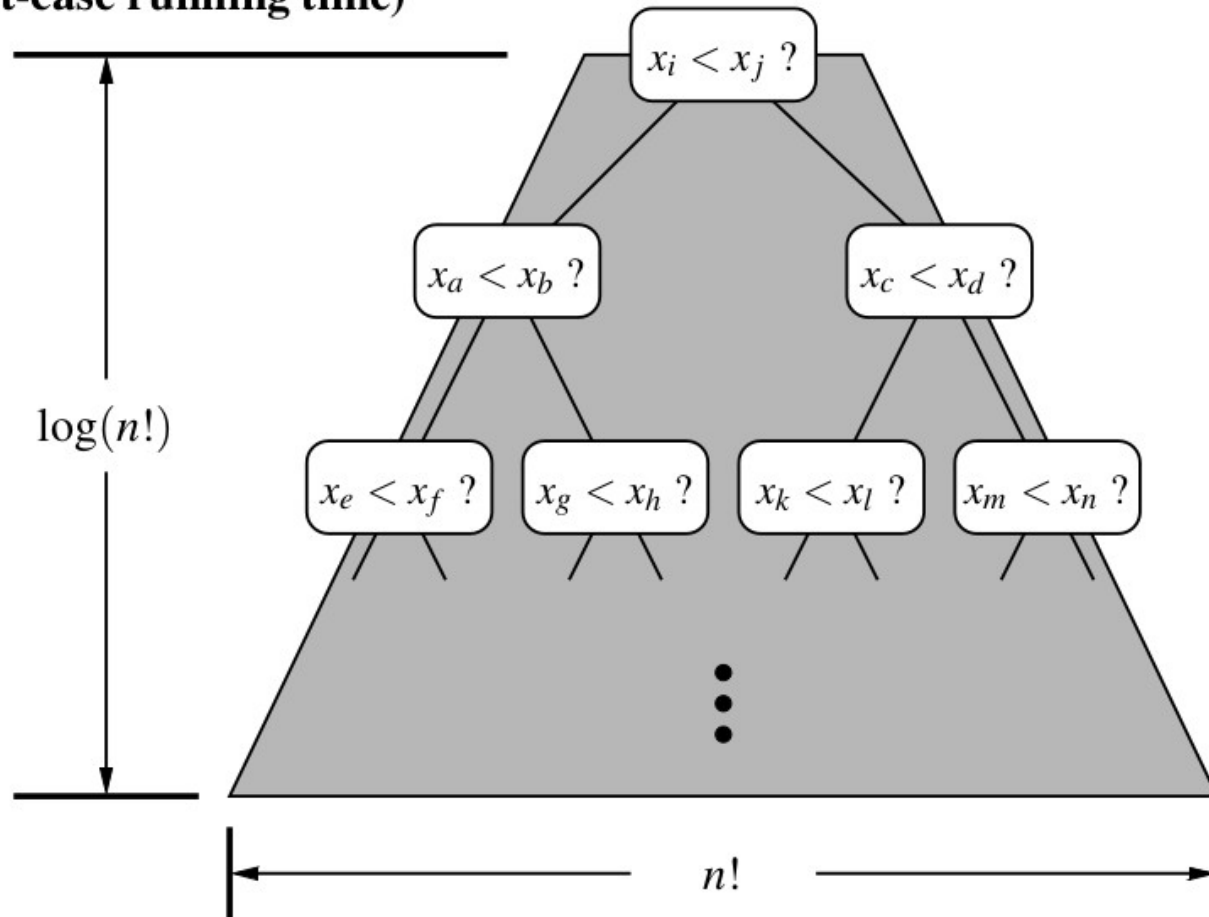
$$\log(n!) \geq \log \left(\left(\frac{n}{2} \right)^{\frac{n}{2}} \right) = \frac{n}{2} \log \frac{n}{2},$$

which is $\Omega(n \log n)$

Since there are at least $n/2$ terms that are greater than or equal to $n/2$ in the product $n!$

Lower Bound for Sorting

Minimum Height
(i.e., worst-case running time)



Non-comparison-based sorting algorithms

Non-comparison-based sorting algorithms such as Radix Sort and Bucket Sort don't rely on directly comparing elements to determine their order.

Bucket Sort: A sequence S of n entries whose keys are integers in the range $[0, N - 1]$, for some integer $N \geq 2$, can be sorted in $O(n + N)$ time.

Both Bucket and Radix sorting methods are stable

Bucket Sort

Algorithm bucketSort(S):

Input: Sequence S of entries with integer keys in the range $[0, N - 1]$

Output: Sequence S sorted in nondecreasing order of the keys

let B be an array of N sequences, each of which is initially empty

for each entry e in S **do**

$k =$ the key of e

 remove e from S and insert it at the end of bucket (sequence) $B[k]$

for $i = 0$ to $N - 1$ **do**

for each entry e in sequence $B[i]$ **do**

 remove e from $B[i]$ and insert it at the end of S

Radix Sort - example

How to sort a sequence

$S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2)).$

Ans:

Stably sort the 2nd component followed by the 1st (ie. start from rightmost and traverse to left side)

$S_2 = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7)).$

$S_{2,1} = ((1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3)),$

Radix Sort - another example

170,045,075,090,802,024,002,066

First we identify each right-most (least significant) digit:

170, 045, 075, 090, 802, 024, 002, 066

We sort the list according to that digit, preserving order within that group:

170, 090, 802, 002, 024, 045, 075, 066

Then we identify the next digit, moving leftwards:

170, 090, 802, 002, 024, 045, 075, 066

We sort the list according to that digit, preserving order within that group:

802, 002, 024, 045, 066, 170, 075, 090

Then we identify the next digit, moving leftwards:

802, 002, 024, 045, 066, 170, 075, 090

We sort the list according to that digit, preserving order within that group:

002, 024, 045, 066, 075, 090, 170, 802,

Timsort — a relatively new kid on the block

1. Timsort is a hybrid, stable sorting algorithm, derived from merge sort and insertion sort
2. Implemented by Tim Peters in 2002 for use in the Python programming language
3. The algorithm finds sub-sequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently.
4. provides good average-case time complexity of $O(n \log n)$, similar to Merge Sort. It also performs well for partially sorted data, making it a versatile choice for Python's default sorting mechanism.

References

Data Structures and Algorithms in Python

Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms

Leiserson, Stein, Rivest, Cormen

Algorithms, 4th Edition

Robert Sedgewick and Kevin Wayne

Few Images from the internet