

DSA Mini Textbook

Theo Park

Contents

Preface	3
1 Runtime Analysis	4
1.1 Power Law	4
1.2 Runtime Expressions	4
1.3 Asymptotic Runtime Analysis	4
1.4 Recursive Relationship	4
2 Intro to Data Structures	5
2.1 Array	5
2.2 Linked List	5
2.3 Stack	5
2.4 Queue	5
2.5 Binary Heap	5
2.5.1 Building a Heap – Top-down v.s. Bottom-up	5
2.6 Tree	5
3 Sorting Algorithms	6
3.1 Bubble, Selection, Insertion Sort	6
3.2 Shell Sort	7
3.3 Heap Sort	7
3.3.1 Sort Down Algorithm	8
3.4 Merge Sort	8
3.4.1 Merge Algorithm	8
3.5 Quick Sort	8
3.5.1 Partition and Pivot	9
3.6 Comparison Sorting Algo Lower Bound	9
3.7 Bucket Sort	10
3.8 Counting Sort	10
3.9 Radix Sort	11
3.9.1 Lexicographic Order	11
3.10 Chapter 3 Review	11
4 Hash Tables	12
4.1 Division Method	12
4.2 Multiplication Method	12
4.3 Collision	12
4.3.1 Chaining	12
4.3.2 Open Addressing	12

5	Search Tree	13
5.1	Binary Search Tree and Its Limit	13
5.2	2-3 Tree	13
5.3	Red-Black Tree	13
5.4	Left-Leaning Red-Black Tree	13
5.4.1	Deletion in LLRBT	13
6	Graph Traversal	14
6.1	Adjacency Matrix and List	14
6.2	DFS	14
6.3	BFS	14
7	Directed Graphs	15
7.1	Strong Connectivity	15
7.1.1	Brute-force Strong Connectivity Algorithm	15
7.1.2	Brute-force using Stack	15
7.1.3	Strongly Connected Components and Kosaraju's Algorithm	15
7.2	Directed Acyclic Graphs	15
7.2.1	Topological Sort	15
8	Weighted Graphs	16
8.1	Shortest Path	16
8.1.1	Dijkstra's Algorithm	16
8.1.2	Bellman-Ford Algorithm	16
8.2	Articulation Points	16
8.3	Minimum Spanning Tree	16
8.3.1	Cycle and Cut Properties	16
8.3.2	Prim's Algorithm	16
8.4	Union-Find	16
8.4.1	Kruskal MST Algorithm	16
9	Strings	17
9.1	Brute-force String Pattern Matching	17
9.2	KMP Algorithm	17
9.3	Trie	17
9.4	PATRICIA	17
9.5	Huffman Coding	17

Preface

Chapter 1

Runtime Analysis

Algorithms are any well-defined computational procedures that take some value(s) as input and produce more value(s) as output. They are **effective**, **precise**, and **finite**. There are several ways to analyze the runtime of an algorithm.

1.1 Power Law

1. For the algorithm, get a table for the input size n and the runtime $T(n)$.

n	$T(n)$
250	0.0
500	0.012
1000	0.0954
2000	0.7727
4000	6.1664

2. Make sure that the data plots:

- **have enough data plots.** For instance, if there are only two data plots, you should not make the power law conjecture.
- **fits the power law.** You can verify this by finding the ratio between data plots.

n	$T(n)$	ratio
250	0.0	–
500	0.012	–
1000	0.0954	$0.0954 / 0.012 = 7.95$
2000	0.7727	$0.7727 / 0.0954 = 8.10$
4000	6.1664	$6.1664 / 0.7727 = 7.98$

For the ratios we found, //TODO

1.2 Runtime Expressions

1.3 Asymptotic Runtime Analysis

1.4 Recursive Relationship

Chapter 2

Intro to Data Structures

Data structures are collections of data values, the relationships among them, and the functions or operations that can be applied to the data. All three characteristics need to be present.

2.1 Array

Array is a linear container of items.

Array length 6	250	251	252	253	254	255
	0	1	2	3	4	5

- Access time: $\Theta(1)$
- Inserting n items in the *tail* for array size n : $\Theta(1)$ per item, $n \times \Theta(1) \in \Theta(1)$
- Inserting n items in the *tail* for array size *unknown*: $\Theta(n)$ per item, $n \times \Theta(n) \in \Theta(n)$

Lesson? **Keep track of the tail!**

2.2 Linked List

2.3 Stack

2.4 Queue

2.5 Binary Heap

2.5.1 Building a Heap – Top-down v.s. Bottom-up

2.6 Tree

Chapter 3

Sorting Algorithms

Once you store all the items in a data structure, you might want to organize them for the future use (such as selecting n th largest element). For this, you have to *sort* the data structure (in this book, array will be assumed). *Sorting* is deciding how to permute the array elements until they are sorted.

There are couple aspects of sorting algorithms you need to consider:

- Runtime: When analyzing a runtime of a sorting algorithm, both number of compares and number of swaps are considered. **Most sorting algorithms make more compares than swaps**, but if a sorting algorithm makes more swaps, it must be used for the asymptotic runtime analysis
- Stability: An algorithm is stable if it preserves the input ordering of equal items For example: //TODO
- In-place: An algorithm is in-place if it can directly sorts the items without making a copy or extra array(s)

3.1 Bubble, Selection, Insertion Sort

The first three sorting algorithms we will discuss are BUBBLE-SORT, SELECTION-SORT, and INSERTION-SORT. They are simple to understand and implement, however, they all have the worst case runtime of $O(n^2)$, which limits their uses.

BUBBLE-SORT goes through the array and swap elements that are out of place, and if such element is found, it repeats from the beginning.

```
1: function BUBBLE-SORT( $A$ )  $\triangleright A$  is an array size  $n$ 
2:    $repeat \leftarrow \text{True}$ 
3:   while  $repeat$  is True do
4:      $repeat \leftarrow \text{False}$ 
5:     for  $i = 0$  to  $n - 2$  do
6:       if  $A[i] > A[i + 1]$  then  $\triangleright$  Assume  $\text{SWAP}(A, i, j)$  swaps  $A[i]$  and  $A[j]$ 
7:          $\text{SWAP}(A, i, i + 1)$ 
8:          $repeat \leftarrow \text{True}$ 
9:   return  $A$ 
```

In-place?	Stable?
True	True

-	NumCompares	NumSwaps
Already Sorted	$n - 1$	0
Worst Case	$n^2 - n$	$\frac{1}{2}n^2 - \frac{1}{2}n$

SELECTION-SORT is a sorting algorithm closest to our “natural” thought of sorting an array. It makes the same number of comparisons no matter what.

```

1: function SELECTION-SORT( $A$ )                                     ▷  $A$  is an array size  $n$ 
2:   for  $i = 0$  to  $n - 2$  do
3:      $index \leftarrow i$ 
4:     for  $i = i + 1$  to  $n - 1$  do
5:       if  $[j] < A[index]$  then
6:          $index \leftarrow j$ 
7:       if  $i \neq index$  then
8:          $SWAP(A, i, index)$                                      ▷ Assume  $SWAP(A, i, j)$  swaps  $A[i]$  and  $A[j]$ 
9:   return  $A$ 

```

In-place?	Stable?
True	False

-	NumCompares	NumSwaps
Already Sorted	$\frac{1}{2}n^2 - \frac{1}{2}n$	0
Worst Case	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\lfloor \frac{1}{2}n \rfloor$

INSERTION-SORT, as the name suggests, take one element at a time and swaps it until it's at the correct index.

```

1: function INSERTION-SORT( $A$ )                                     ▷  $A$  is an array size  $n$ 
2:   for  $i = 1$  to  $n - 1$  do
3:      $j \leftarrow i - 1$ 
4:     while  $j \geq 0$  and  $A[j] > A[j + 1]$  do
5:        $SWAP(A, j, j + 1)$                                      ▷ Assume  $SWAP(A, i, j)$  swaps  $A[i]$  and  $A[j]$ 
6:        $j \leftarrow j - 1$ 
7:   return  $A$ 

```

In-place?	Stable?
True	True

-	NumCompares	NumSwaps
Already Sorted	$n - 1$	0
Worst Case	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$

3.2 Shell Sort

3.3 Heap Sort

HEAP-SORT uses binary max-heap to sort an array. While it's the first sorting algorithm to utilize a data structure, it's not preferred in real life due to cache issue.

```

1: function HEAP-SORT( $A$ )                                         ▷  $A$  is an array size  $n$ 
2:    $A \leftarrow \text{BUILD-HEAP}(A)$ 
3:   for  $i = n - 1$  down to 0 do
4:      $\text{SORT-DOWN}(A, i)$ 
5:   return  $A$ 

```

The algorithm first builds the heap from the array elements (refer to section 2.5 for methods for building a heap). BOTTOM-UP is used for its runtime. Then the algorithm calls SORT-DOWN from the last heap elements down to the first.

3.3.1 Sort Down Algorithm

3.4 Merge Sort

MERGE-SORT is an algorithm //TODO

```

1: function MERGE-SORT( $A, l, r$ )                                      $\triangleright A$  is an array size  $n$ 
2:   if  $l < r$  then
3:      $m \leftarrow (l + r)/2$ 
4:     MERGE-SORT( $A, l, m$ )
5:     MERGE-SORT( $A, m + 1, r$ )
6:     MERGE( $A, l, m, r$ )
7:   return  $A$ 

```

3.4.1 Merge Algorithm

```

.....
1: function MERGE( $A, l, m, r$ )                                      $\triangleright A$  is an array size  $n$ 
2:    $n1 \leftarrow m - l + 1$ 
3:    $n2 \leftarrow r - m$ 
4:    $L \leftarrow$  array size of  $(n1 + 1)$ 
5:    $R \leftarrow$  array size of  $(n2 + 1)$ 
6:    $\triangleright$  Assign elements to each array                                 $\triangleleft$ 
7:   for  $i = 0$  to  $n1 - 1$  do
8:      $L[i] \leftarrow A[l + i]$ 
9:   for  $i = 0$  to  $n2 - 1$  do
10:     $R[i] \leftarrow A[m + i + 1]$ 
11:    $L[n1], R[n2] \leftarrow \infty$ 
12:    $i, j \leftarrow 0$ 
13:   for  $k = l$  to  $r$  do
14:     if  $L[i] \leq R[j]$  then
15:        $A[k] \leftarrow L[i]$ 
16:        $i \leftarrow i + 1$ 
17:     else
18:        $A[k] \leftarrow R[j]$ 
19:        $j \leftarrow j + 1$ 
20:   return  $A$ 
.....

```

3.5 Quick Sort

QUICK-SORT is another divide-and-conquer sorting algorithm.

```

function QUICK-SORT( $A, l, r$ )                                      $\triangleright A$  is an array size  $n$ 

```

```

if  $l < r$  then
     $m \leftarrow \text{PARTITION}(A, l, r)$ 
     $\text{QUICK-SORT}(A, l, m - 1)$ 
     $\text{QUICK-SORT}(A, m + 1, r)$ 
return  $A$ 

```

3.5.1 Partition and Pivot

```

function PARTITION( $A, l, r$ )

```

$\triangleright A$ is an array size n

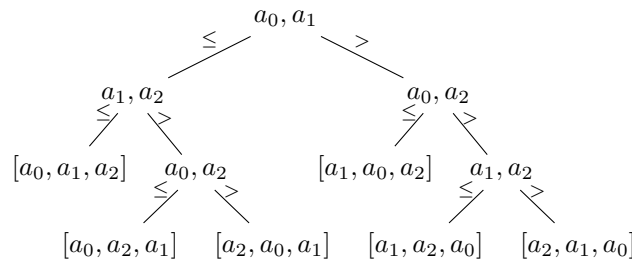
```

     $p = A[r]$ 
     $i = l - 1$ 
    for  $j = 1$  to  $r$  do
        if  $A[j] < p$  then
             $i \leftarrow i + 1$ 
            SWAP( $A, i, j$ )
     $i \leftarrow i + 1$ 
    SWAP( $A, i, r$ )
    return  $i$ 

```

3.6 Decision Tree and the Lower Bound for Comparison Sorting Algorithm

So far, we have been discussing *comparison sorting algorithms* (sorting algorithms that only reads array elements through $>, =, <$ comparison). We can draw a decision tree with all the permutations of how a comparison sorting algorithm would compare and sort the array $[a_0, a_1, a_2]$.



The decision tree for array size of 3 has $3! = 6$ leaves, and it is trivial that a decision tree for an **array with n elements will have $n!$ leaves**. The height of the decision tree represents the worst-case number of comparisons the algorithm has to make in order to sort the array.

A tree with the height h has at most 2^h leaves. Using this property, we can have the lower Big-omega bound for height of the decision tree for comparison sorting algorithms.

$$\begin{aligned}
2^h &\geq n! \therefore h \geq \log_2(n!) && (h \text{ is the height of the decision tree}) \\
h &\geq \log_2(n!) = \log_2(1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n) \\
&= \log_2(1) + \log_2(2) + \dots + \log_2(n-1) + \log_2(n) \\
&= \sum_{i=1}^n \log_2(i) = \sum_{i=1}^{\frac{n}{2}-1} \log_2(i) + \sum_{i=\frac{n}{2}}^n \log_2(i) \\
&\geq 0 + \sum_{i=\frac{n}{2}}^n \log_2(i) \geq 0 + \sum_{i=\frac{n}{2}}^n \log_2\left(\frac{n}{2}\right) && (i \text{ in the former expression is } \geq \frac{n}{2}) \\
&= \frac{n}{2} \log_2\left(\frac{n}{2}\right) && (i = \frac{n}{2} \text{ to } n \text{ is exactly } \frac{n}{2} \text{ iterations}) \\
&\in \Theta(n \log_2(n))
\end{aligned}$$

Because $h \geq \log_2(n!) \in \Theta(n \log_2(n))$, $h \in \Omega(n \log_2(n))$. Therefore, we can conclude that any **comparison sorting algorithms cannot run faster than $O(n \log_2(n))$ time** in the worst-case scenario.

3.7 Bucket Sort

BUCKET-SORT is a sorting algorithm where elements are divided into each "bucket" and a different sorting algorithm is called for each bucket. While BUCKET-SORT is a comparison sorting algorithm, it's an attempt to reduce the runtime by decreasing the number of elements that the sorting algorithm has to sort.

3.8 Counting Sort

COUNTING-SORT is a *non-comparison* sorting algorithm. It uses the extra array *count*, where its index initially represents the value of each element in *A* (e.g., if there are three 5's in *A*, $\text{count}[5] = 3$ before the "accumulation" step to determine the final index), to sort the array.

```

1: function COUNTING-SORT(A, k)                                ▷ A is an array size n, k is the max element of A
2:   count ← array size k + 1 filled with 0
3:   for i = 0 to n - 1 do                                       ▷ Num occurrence in each element in A,  $O(n)$ 
4:     count[A[i]] ← count[A[i]] + 1

5:   for i = 1 to k do                                           ▷ Accumulate the values in count from left to right,  $O(k)$ 
6:     count[i] ← count[i] + count[i - 1]

7:   out ← array size n
8:   for i = n - 1 down to 0 do                                   ▷ Use count values to determine the index for the elements in A,  $O(n)$ 
9:     out[count[A[i]] - 1] ← A[i]
10:    count[A[i]] ← count[A[i]] - 1

11: return out

```

1. Suppose we have an array $A = [2, 5, 3, 0, 2, 3, 0, 3]$. $k = \text{MAX}(A) = 5$.
2. Initialize *count*, the array size $5 + 1$, with 0's. $\text{count} = [0, 0, 0, 0, 0, 0]$.
3. Count number of occurrence. $\text{count} = [2, 0, 2, 3, 0, 1]$ (e.g., 2 occurred 2 times)
4. Accumulate values of *count* from left to right. $\text{count} = [2, 2, 4, 7, 7, 8]$ (e.g., $\text{count}[1] = 2 + 0$, $\text{count}[2] = 2 + 0 + 2, \dots$)

5. Initialize *out*, the array size $n = 8$, with nil's. $out = [\text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}]$.

6. Place each element to the *out* array using *count* array

(a) When $i = n - 1 = 7$: $A[7] = 3$ and $count[3] = 7 \Rightarrow out[7 - 1] := A[7] = 3$ and $count[3] := 7 - 1$

$out = [\text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, 3, \text{nil}]$
 $count = [2, 2, 4, 6, 7, 8]$

(b) When $i = n - 2 = 6$: $A[6] = 0$ and $count[0] = 2 \Rightarrow out[2 - 1] := A[6] = 0$ and $count[0] := 2 - 1$

$out = [\text{nil}, 0, \text{nil}, \text{nil}, \text{nil}, \text{nil}, 3, \text{nil}]$
 $count = [1, 2, 4, 6, 7, 8]$

(c) When $i = n - 3 = 5$: $A[5] = 3$ and $count[3] = 6 \Rightarrow out[6 - 1] := A[5] = 3$ and $count[3] := 6 - 1$

$out = [\text{nil}, 0, \text{nil}, \text{nil}, \text{nil}, 3, 3, \text{nil}]$
 $count = [1, 2, 4, 5, 7, 8]$

(d) ...

$out = [0, 0, 2, 2, 3, 3, 3, 5]$
 $count = [0, 2, 2, 4, 7, 7]$

In-place?	Stable?
False	True

Because of its use for RADIX-SORT, COUNTING-SORT must be stable, and it indeed is. If there are items with the same value, it will be moved to the *out* array in order in the last (third) for loop.

Runtime	Space Usage
$O(n + k)$	$O(n + k)$

As the algorithm iterates both the size of the array n and the maximum element in the array k , **the algorithm runs in $O(n + k)$ time and uses $O(n + k)$ space.**

3.9 Radix Sort

RADIX-SORT is a non-comparative sorting algorithm for elements with more than one significant digits. It utilizes a stable sorting algorithm such as COUNTING-SORT to sort elements lexicographically.

```

1: function RADIX-SORT( $A, k$ )                                 $\triangleright A$  is an array where the maximum dimension of an element is  $d$ 
2:   for  $i = d$  down to 1 do
3:      $\quad$  Call a stable sorting algorithm at dimension  $i$ 
4:   return  $A$ 

```

3.9.1 Lexicographic Order

$$(x_1, x_2, \dots, x_d) < (y_1, y_2, \dots, y_d) \Leftrightarrow (x_i < y_i) \vee (x_1 = y_1 \wedge (x_2, \dots, x_d) < (y_2, \dots, y_d))$$

3.10 Chapter 3 Review

Chapter 4

Hash Tables

4.1 Division Method

4.2 Multiplication Method

4.3 Collision

4.3.1 Chaining

4.3.2 Open Addressing

Chapter 5

Search Tree

5.1 Binary Search Tree and Its Limit

5.2 2-3 Tree

5.3 Red-Black Tree

5.4 Left-Leaning Red-Black Tree

5.4.1 Deletion in LLRBT

Chapter 6

Graph Traversal

6.1 Adjacency Matrix and List

6.2 DFS

6.3 BFS

Chapter 7

Directed Graphs

7.1 Strong Connectivity

7.1.1 Brute-force Strong Connectivity Algorithm

7.1.2 Brute-force using Stack

7.1.3 Strongly Connected Components and Kosaraju's Algorithm

7.2 Directed Acyclic Graphs

7.2.1 Topological Sort

Chapter 8

Weighted Graphs

8.1 Shortest Path

8.1.1 Dijkstra's Algorithm

8.1.2 Bellman-Ford Algorithm

8.2 Articulation Points

8.3 Minimum Spanning Tree

8.3.1 Cycle and Cut Properties

8.3.2 Prim's Algorithm

8.4 Union-Find

8.4.1 Kruskal MST Algorithm

Chapter 9

Strings

9.1 Brute-force String Pattern Matching

9.2 KMP Algorithm

9.3 Trie

9.4 PATRICIA

9.5 Huffman Coding