UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence/Scientific Computing

Department of mathematics informatics and geosciences

Algorithmic Design

Lecturers:

Prof. Bernardini Giulia
Prof. Padoan Tommaso

Authors:

Christian Faccio
Carlos Velázquez
Andrea Spinelli

May 27, 2025

github.com/christianfaccio christianfaccio@outlook.it

Preface

As a student of the “Data Science and Artificial Intelligence” master’s degree at the University of Trieste, I have created these notes to study the course “Algorithmic Design” held by Prof. Giulia Bernardini. The course aims to provide students with a solid foundation in designing algorithms, analyzing their complexity, and understanding their applications in various domains. The course covers the following topics:

- Basics
- Sorting Algorithms
- Searching Algorithms

Note that this is only the first part of the course.

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in this field.

Contents

1 Introduction

Definition: *Algorithm*

An **algorithm** is a finite sequence of step-by-step, well-defined instructions that takes some value, or set of values, as input and produces some value, or set of values, as output.

It can be described by a **pseudocode**, which can then be converted in any programming language to obtain a working sofftware.

Algorithm 1 Example: Linear Search

Input: An array $A[1,\dots,n]$ of numbers and a number q
Output: An index i such that $A[i] = q$; or **FAIL** If no such index exists

```
1:  j ← 1                                          ▷ Variable initialization
2:  while j ≤ n do                                 ▷ Loop syntax, indentation required
3:      if A[j] = q then
4:          return j                               ▷ Return syntax
5:      end if
6:      j ← j + 1
7:  end while
8:  if j = n + 1 then                              ▷ If syntax, indentation required
9:      return FAIL                                ▷ Return syntax
10: end if
```

Definition: *Complexity*

The **complexity** of an algorithm is a function that gives the amount of resources (such as time, memory, communication bandwidth, etc.) that the algorithm requires to solve an instance of the problem as a function of the size of the instance. In simpler terms, it is the number of "steps" done by the algorithmas a function of the number of input elements.

We carry on a **worst-case analysis** of the complexity, which is the maximum number of steps that the algorithm can do for any input. This to guarantee to the user on the performance of the algorithm even with the most unfortunate input. So we want to compute the upper bounds on the running time.

We are interested in the **order of growth** of an algorithm: the fastest-growing term of the function that expresses the number of steps done by the algorithm in the worst case.

- Ignore machine-dependent constants
- Look at the growth of the number of instructions as a function of the input size n, when $n \to \infty$
- We will express the time complexity of algorithms using the asymptotic notation $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, $\omega(\cdot)$.

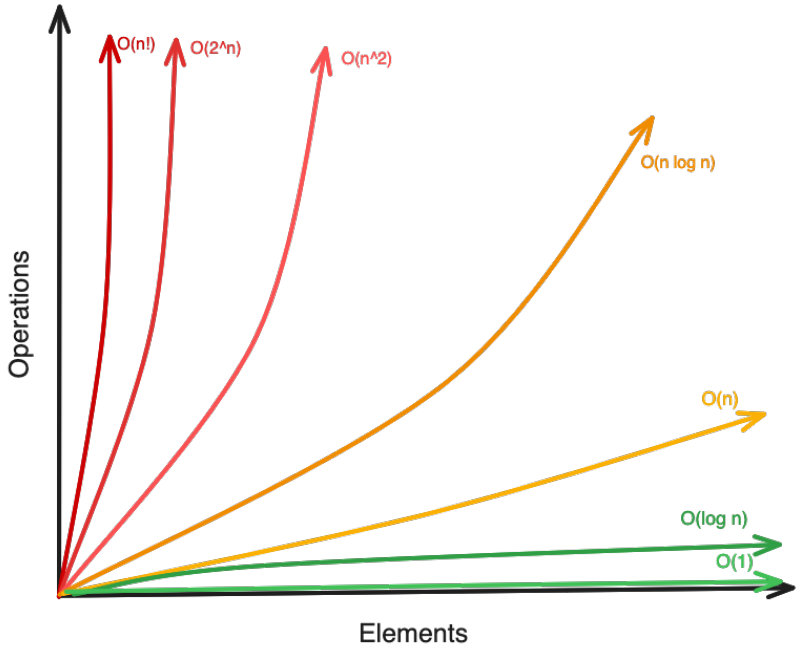


Figure 1.1: Complexity

Definition: *Time Complexity*

- **f(n) = O(g(n))**
  If $\exists c > 0,\ \text{int}\ n_0 > 0\ \text{ s.t. }\ 0 \leq f(n) \leq c \cdot g(n)\ \ \forall n > n_0$
- **f(n) = o(g(n))**
  If $\exists c > 0,\ \text{int}\ n_0 > 0\ \text{ s.t. }\ 0 \leq f(n) < c \cdot g(n)\ \ \forall n > n_0$
- **f(n) = Ω(g(n))**
  If $\exists c > 0,\ \text{int}\ n_0 > 0\ \text{ s.t. }\ 0 \leq c \cdot g(n) \leq f(n)\ \ \forall n > n_0$
- **f(n) = ω(g(n))**
  If $\exists c > 0,\ \text{int}\ n_0 > 0\ \text{ s.t. }\ 0 \leq c \cdot g(n) < f(n)\ \ \forall n > n_0$
- **f(n) = Θ(g(n))**
  If $\exists c_1, c_2 > 0,\ \text{int}\ \ n_0 > 0\ \text{ s.t. }\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \ \forall n > n_0$

We need now to fix a model of computation in order to analyze algorithms. We will start with the most widely used **RAM Model**.

- The model assumes a memory where each cell has a unique address, and accessing any memory cell takes the same constant time, regardless of its location.
- It defines a set of basic operations (e.g., addition, subtraction, comparison, assignment, etc.), each of which is assumed to take constant time.
- A single processing unit executes instructions one at a time in a sequential order (no parallelism).
- Each word in memory can store a value (e.g., an integer or a pointer), and operations are performed on these words. The model assumes that the size of a word is large enough to hold the required data.
- The model abstracts away the low-level details of a real computer, such as cache, paging, or specIfic hardware architecture.

2 Sorting Algorithms

2.1 RAM Model

2.1.1 Characteristics

Comparison-based vs Non-comparison-based

- **Comparison-based** algorithms are algorithms that sort a list by comparing elements of the list. These algorithms generally have lower bounds of $O(n \log n)$ due to the fundamental limitations of comparison-based sorting.
  The most common comparison-based sorting algorithms are:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
- **Non-comparison-based** algorithms don't rely on comparisons but instead use counting, hashing, or other methods to determine the order. They can sometimes achieve better time complexity (e.g., $O(n)$) by exploiting special properties of the input data.
  The most common non-comparison-based sorting algorithms are:
  - Counting Sort
  - Radix Sort
  - Bucket Sort

Space Complexity

How much extra memory is required by the algorithm relative to the size of the input. For example, an algorithm that requires $O(n)$ extra space is less efficient in terms of memory usage than one that requires only $O(1)$ space. The space complexity is particularly important when dealing with large datasets or memory-constrained environments.

Stability

A sorting algorithm is **stable** if it preserves the relative order of records with equal keys (i.e., elements that compare equal will retain their original order). This property is crucial when sorting records with multiple fields, where secondary ordering needs to be maintained.

The most common stable sorting algorithms are:

- Bubble Sort
- Insertion Sort
- Merge Sort

In-place vs Out-of-place

- **In-place**: The algorithm sorts the elements by modifying the input array directly, using only a constant amount of extra space (O(1) space complexity). This makes them memory efficient and suitable for large datasets. The most common in-place sorting algorithms are:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Heap Sort
- **Out-of-place**: The algorithm requires additional space proportional to the input size to hold a separate copy of the data (e.g., merge sort uses extra space for merging). While these algorithms may use more memory, they often offer other advantages like stability or better time complexity.

Online vs Offline

- **Online**: These algorithms can process the input in a piece-by-piece manner without needing the entire dataset upfront (e.g., insertion sort). This makes them suitable for streaming data or real-time applications.
- **Offline**: These algorithms require the entire dataset to be available before they can start sorting (e.g., merge sort). While this may be a limitation in some scenarios, offline algorithms often achieve better overall performance.

Divide and Conquer

Algorithms that use the divide-and-conquer paradigm divide the problem into smaller subproblems, solve each subproblem recursively, and then combine the results (e.g., merge sort, quicksort). This approach often leads to efficient algorithms with good time complexity.

- **Divide**: The algorithm divides the input into smaller, manageable subproblems.
- **Conquer**: The algorithm solves the subproblems recursively.
- **Combine**: The algorithm combines the solutions of the subproblems to solve the original problem.

Parallelism

Some sorting algorithms can take advantage of parallelism to speed up the sorting process. For example, merge sort can be parallelized by dividing the input into smaller subproblems and sorting them in parallel. This property becomes increasingly important with modern multi-core processors and distributed systems.

Recursive vs Iterative

- **Recursive**: Algorithms like quicksort and merge sort use recursion to break down the problem into smaller subproblems. While recursive implementations can be more elegant and easier to understand, they may have additional overhead due to function call stack.
- **Iterative**: Algorithms like bubble sort and insertion sort use loops and iteration to solve the problem without recursion. These implementations often have better space efficiency and may perform better in practice for small inputs.

2.1.2 Insertion Sort

Algorithm 2 Insertion Sort (A)

```
1: for j = 2 to n do
2:     key = A[j]                                   ▷ 1 × (n − 1)
3:     i = j - 1                                    ▷ 1 × (n − 1)
4:     while i > 0 and A[i] > key do
5:         A[i + 1] = A[i]                          ▷ 1 × n(n−1)/2
6:         i = i - 1                                ▷ 1 × n(n−1)/2
7:     end while
8:     A[i + 1] = key                               ▷ 1 × (n − 1)
9: end for
```

- **Best case**: $\theta(n)$
- **Worst case**: $\theta(n^2)$

Definition: ***Insertion Sort***

The basic concept of the Insertion Sort algorithm is to build a sorted array (or list) one element at a time by repeatedly taking the next element from the unsorted portion and inserting it into the correct position within the sorted portion. This process is similar to how one might sort playing cards in their hands.

Here's a step-by-step explanation:

- Start with the second element (index 1) of the array. This element is considered the "key" and will be compared with the elements before it.
- Compare the key with the element before it. If the key is smaller, shift the previous element to the right.
- Continue shifting elements to the right until you find the correct position for the key.
- Insert the key into its correct position.
- Move to the next element and repeat the process until the entire array is sorted.

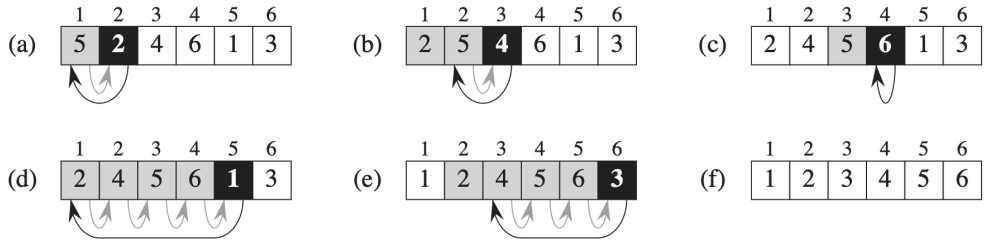


Figure 2.1: Insertion Sort [1]

Characteristics

- **Stable**: Yes
- **In-place**: Yes
- **Online**: Yes
- **Divide and Conquer**: No
- **Parallelism**: No
- **Recursive vs Iterative**: Iterative
- **Space Complexity**: O(1)
- **Comparison-based**: Yes

2.1.3 Merge Sort

Algorithm 3 Merge Sort (A, p, r)

```
1: if p < r then
2:     q = ⌊(p+r)/2⌋
3:     MergeSort(A, p, q)              ▷ Recursively sort the left half
4:     MergeSort(A, q + 1, r)          ▷ Recursively sort the right half
5:     Merge(A, p, q, r)               ▷ Merge the two sorted halves
6: end if
```

$1 + logn$ levels of the tree structure
Time complexity: $\theta(n \log n)$

The MERGE-SORT procedure splits the array into two halves, recursively sorts each half, and then merges the two sorted halves. The base case of the recursion occurs when the array to be sorted has length 1, in which case there is no work to be done, since every array of length 1 is already sorted.

Observation: *Stopping condition*

It stops when there is one element in the array, where $p = r$.

Algorithm 4 Merge (A, p, q, r)

```
1: n1 ← q − p + 1
2: n2 ← r − q
3: for i = 1 to n1 do                 ▷ Copy the new portions of the array into two new arrays
4:     L[i] ← A[p+i-1]
5: end for
6: for j = 1 to n2 do
7:     R[j] ← A[q+j]
8: end for
9: L[n1+1] ← ∞,  R[n2+1] ← ∞
10: i ← 1,  j ← 1
11: for k = p to r do                 ▷ Two-fingers merge algorithm
12:     if L[i] ≤ R[j] then
13:         A[k] ← L[i]
14:         i ← i + 1
15:     else
16:         A[k] ← R[j]
17:         j ← j + 1
18:     end if
19: end for
```

Time complexity: $\theta(n)$

Definition: *Merge Sort*

This algorithm is a **divide-and-conquer** algorithm that divides the input array into two halves, recursively sorts the two halves, and then merges the sorted halves. The merge operation combines two sorted arrays into a single sorted array. It is also **recursive** in its structure since it calls itself on subproblems.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order. The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure $MERGE(A, p, q, r)$, where A is an array and p, q, and r are indices into the array such that $p \leq q < r$. The procedure assumes that the subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray $A[p \ldots r]$.

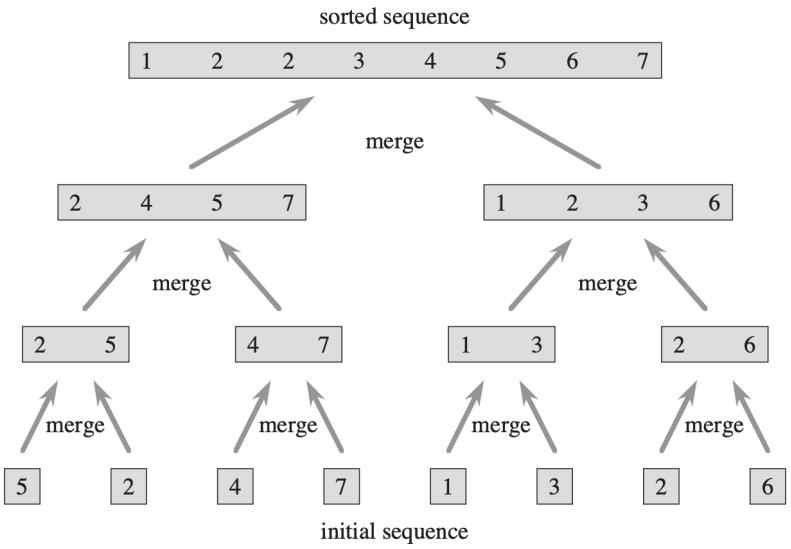


Figure 2.2: Merge Sort [1]

Time complexity analysis

We reason as follows to set up the recurrence for $\Theta(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

- **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**: The MERGE procedure on a subarray of size n takes time $\theta(n)$, and so $C(n) = \theta(n)$.

Where

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

And the solution to this recurrence is $T(n) = \theta(n \log n)$.

Characteristics

- **Stable**: Yes
- **In-place**: No
- **Online**: No
- **Divide and Conquer**: Yes
- **Parallelism**: Yes
- **Recursive vs Iterative**: Recursive
- **Space Complexity**: O(n)
- **Comparison-based**: Yes

2.1.4 Heapsort

The **(binary) heap** data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

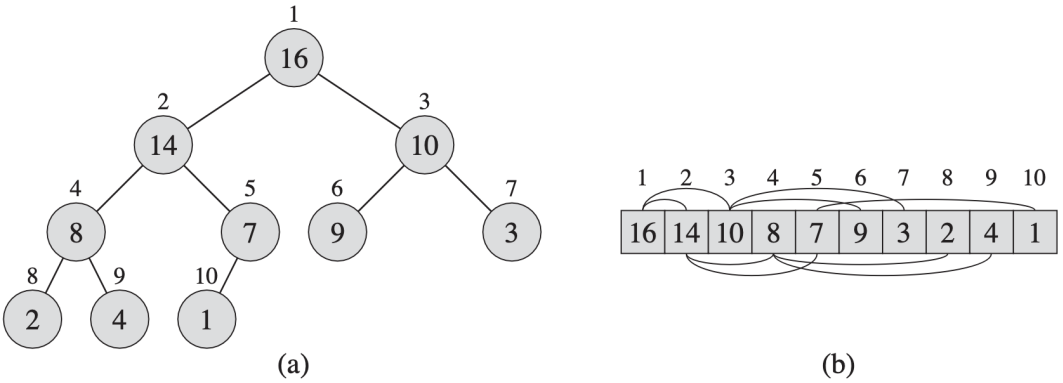


Figure 2.3: A max-heap viewed as a Binary Tree (a) and an array (b) [1]

The root of the tree is $A[1]$, and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

- **Parent**: $\lfloor i/2 \rfloor$
- **Left child**: $2i$
- **Right child**: $2i+1$

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a **heap property**, the specifics of which depend on the kind of heap.

Definition: ***Heap property***

- A **max-heap property** requires that for every node i other than the root:

$$A[PARENT(i)] \geq A[i]$$

  This means that:
  - The value of a node is at most the value of its parent
  - The root contains the maximum element
  - Any subtree is also a max-heap
- A **min-heap property** requires that for every node i other than the root:

$$A[PARENT(i)] \leq A[i]$$

  This means that:
  - The value of a node is at least the value of its parent
  - The root contains the minimum element
  - Any subtree is also a min-heap

These properties ensure that a path from any node to the root is sorted (non-increasing for max-heap, non-decreasing for min-heap).

Algorithm 5 Max-Heapify (A, i)

```
1: l ← LEFT(i)
2: r ← RIGHT(i)
3: if l ≤ heap-size[A] and A[l] > A[i] then
4:     largest ← l
5: else
6:     largest ← i
7: end if
8: if r ≤ heap-size[A] and A[r] > A[largest] then
9:     largest ← r
10: end if
11: if largest ≠ i then
12:     exchange A[i] with A[largest]
13:     Max-Heapify(A, largest)
14: end if
```

Levels: $\log n$ Time complexity: $\theta(\log n)$

Algorithm 6 Build-Max-Heap (A)

```
1: heap-size[A] ← length[A]
2: for i = ⌊ length[A]/2 ⌋ downto 1 do
3:     Max-Heapify(A, i)
4: end for
```

Time complexity: $O(n \log n)$

Algorithm 7 Heap-Sort (A)

```
1: Build-Max-Heap(A)                          ▷ O(n log n)
2: for i = length[A] downto 2 do              ▷ n times
3:     exchange A[1] with A[i]
4:     heap-size[A] ← heap-size[A] - 1
5:     Max-Heapify(A, 1)                      ▷ O(log n)
6: end for
```

Time complexity: $O(n \log n)$

In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY. Its inputs are an array A and an index i into the array. When it is called, **MAX- HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max- heaps**, but that A[i] might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Note that:

- **A.length** is the number of elements in the array.
- **A.heap-size** is the number of elements in the heap stored in array A.

Observation: *Choosing i*

The procedure MAX-HEAPIFY is used in a bottom-up manner to convert an array A[1..n], where n = A.length, into a max-heap. The elements in the subarray A[$\lfloor n/2 \rfloor$+1..n] are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\theta(\log n)$.

We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\log n)$ time.

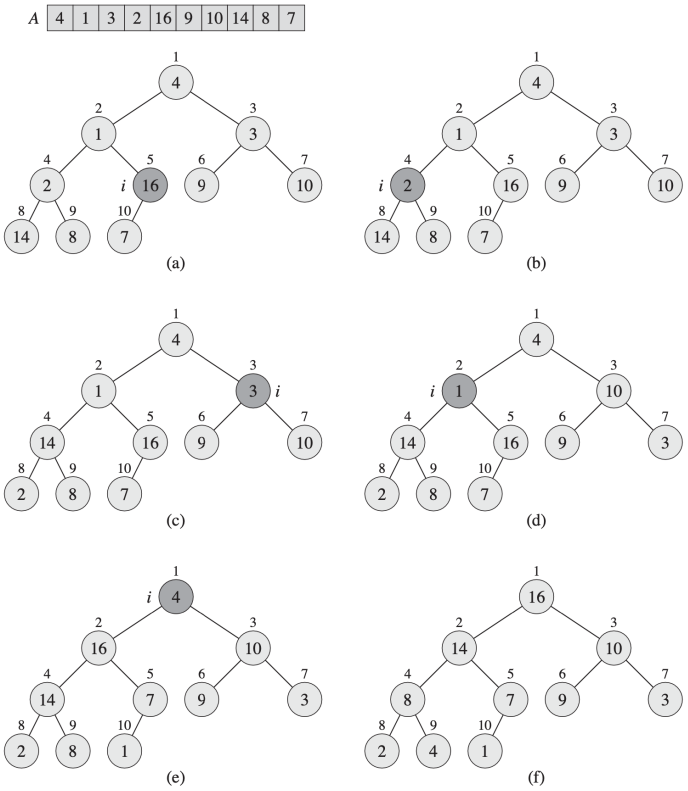


Figure 2.4: BUILD-MAX-HEAP [1]

Characteristics

- **Stable**: No
- **In-place**: Yes
- **Online**: No
- **Divide and Conquer**: No
- **Parallelism**: No
- **Recursive vs Iterative**: Recursive
- **Space Complexity**: O(1)
- **Comparison-based**: Yes

2.1.5 Quicksort

The quicksort algorithm has a worst-case running time of $\theta(n^2)$ on an input array of n numbers.

Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $\theta(n \log n)$, and the constant factors hidden in the $\theta(n \log n)$ notation are quite small. It also has the advantage of sorting in place.
It uses the divide-and-conquer paradigm. Here is the basic idea:

- **Divide**: Partition (rearrange) the array $A[p \ldots r]$ into two (possibly empty) subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that each element of $A[p \ldots q-e]$ is less than or equal to A[q] , which is, in turn, less than or equal to each element of $A[q+1 \ldots r]$. Compute the index q as part of this partitioning procedure.
- **Conquer**: Sort the two subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ by recursive calls to quicksort.
- **Combine**: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p \ldots r]$ is now sorted.

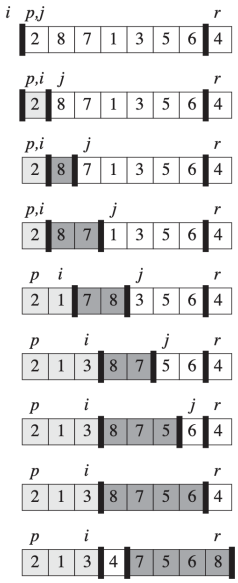


Figure 2.5: Quicksort [1]

Algorithm 8 Quicksort (A, p, r)

```
1: if p < r then
2:     q ← Partition(A, p, r)
3:     Quicksort(A, p, q-1)
4:     Quicksort(A, q+1, r)
5: end if
```

Best case: $\theta(n \log n)$
Worst case: $\theta(n^2)$

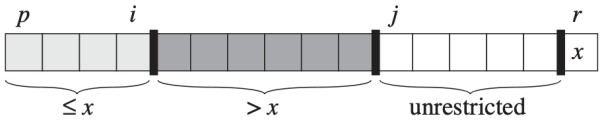


Figure 2.6: Regions maintained by PARTITION [1]

Algorithm 9 Partition (A, p, r)

```
1: x ← A[r]                                   ▷ Pivot element
2: i ← p - 1
3: for j = p to r-1 do
4:     if A[j] ≤ x then
5:         i ← i + 1
6:         exchange A[i] with A[j]
7:     end if
8: end for
9: exchange A[i+1] with A[r]
10: return i + 1
```

Repeat n times

Observation: *i, j values*

The indices i and j are used to keep track of the elements in the array. The index i is used to keep track of the elements that are less than or equal to the pivot element, while the index j is used to iterate through the array.

Performance

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.

- **Best case**: $\theta(n \log n)$
  In the most even possible split, PARTITION produces two subproblems, each of size no more than n=2, since one is of size bn=2cand one of size dn=2e 1. In this case, quicksort runs much faster. The recurrence for the running time is then

$$T(n) = 2T(n/2) + \theta(n)$$

  where we tolerate the sloppiness from ignoring the floor and ceiling and from subtracting 1.
- **Worst case**: $\theta(n^2)$
  The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n 1 elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\theta(n)$ time. Since the recursive call on an array of size 0 just returns, $T(0) = \theta(1)$, and the recurrence for the running time is:

$$T(n) = T(n-1) + \theta(n)$$

- **Average case**: $\theta(n \log n)$
  When we run quicksort on a random input array, the partitioning is highly unlikely to happen in the same way at every level, we expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. The average-case running time of quicksort is much closer to the best case than to the worst case. The total cost of quicksort is therefore $O(n \log n)$.

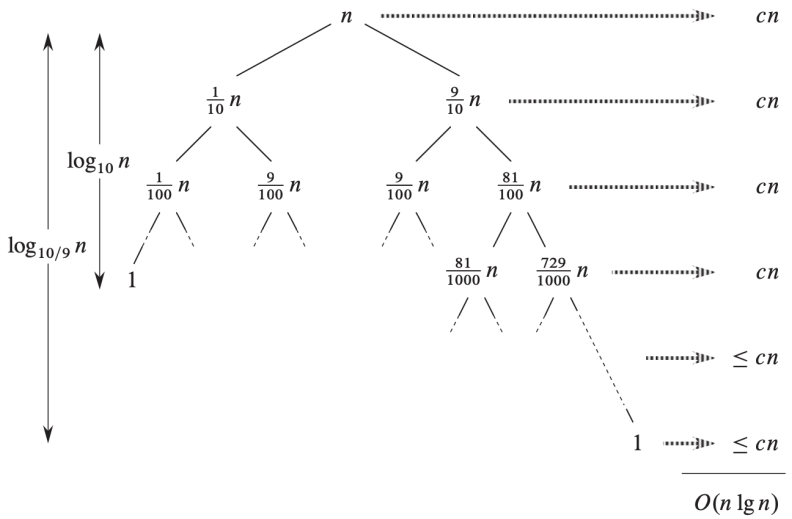


Figure 2.7: PARTITION always of 9 to 1 [1]

Characteristics

- **Stable**: No
- **In-place**: Yes
- **Online**: No
- **Divide and Conquer**: Yes
- **Parallelism**: Yes
- **Recursive vs Iterative**: Recursive
- **Space Complexity**: O($\log n$)
- **Comparison-based**: Yes

2.2 Sorting in Linear Time

Algorithms that can run on $\omega(n \log n)$ time complexity are considered to be linear time sorting algorithms. They share the property of determining the order of elements only by comparing them. For this they are called **comparison sorts**.

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order.

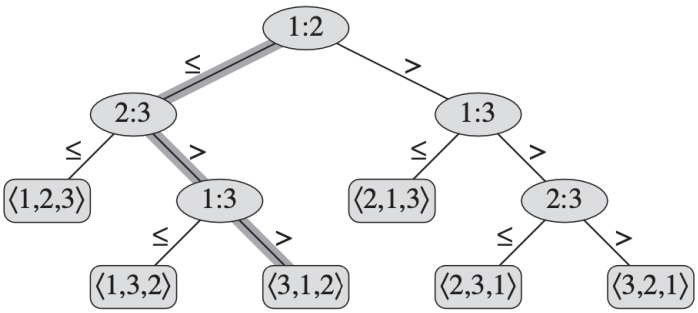


Figure 2.8: Decision Tree Model [1]

Definition: ***Decision Tree Model***

The decision tree model is a way to analyze comparison-based sorting algorithms. In this model, the algorithm is viewed as a binary tree, where each internal node represents a comparison between two elements, and each leaf represents a permutation of the input. The height of the tree represents the worst-case running time of the algorithm.

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

Theorem 2.2.1. *Any comparison sort algorithm requires* $\Omega(n \log n)$ *comparisons in the worst case.*

Proof. There are n! permutations of n elements, and any comparison sort algorithm must be able to distinguish among them to sort correctly. We can use a decision tree to model the algorithm. Each internal node of the tree corresponds to a comparison between two elements, and each leaf corresponds to a permutation of the input. Since the algorithm must be able to distinguish among n! permutations, the decision tree must have at least n! leaves. A binary tree of height h can have at most 2^h leaves. Therefore, we must have $2^h \geq n!$, which implies that $h = \Omega(\log n!) = \Omega(n \log n)$. □

2.2.1 Counting Sort

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than x, then x belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

In the code for counting sort, we assume that the input is an array $A[1,\ldots,n]$, and thus A:lengthDn. We require two other arrays: the array $B[1,\ldots,n]$ holds the sorted output, and the array $C[1,\ldots,k]$ provides temporary working storage.

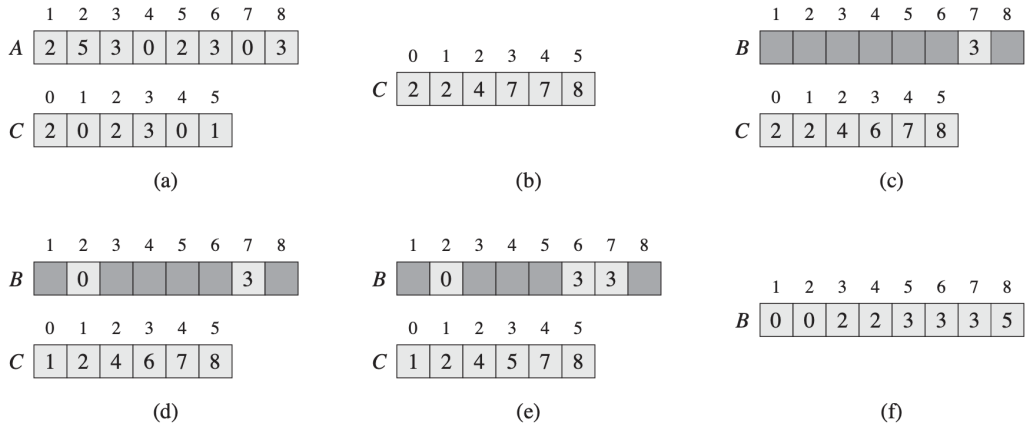


Figure 2.9: Counting Sort [1]

Algorithm 10 Counting Sort (A, B, k)

```
1:  for i = 1 to k do
2:      C[i] ← 0
3:  end for
4:  for j = 1 to A.length do
5:      C[A[j]] ← C[A[j]] + 1
6:                                  ▷ C[i] now contains the number of elements equal to i
7:  end for
8:  for i = 2 to k do
9:      C[i] ← C[i] + C[i-1]
10:                                 ▷ C[i] now contains the number of elements less than or equal to i
11: end for
12: for j = A.length downto 1 do
13:     B[C[A[j]]] ← A[j]
14:     C[A[j]] ← C[A[j]] - 1
15: end for
```

Time complexity: $\theta(n+k)$ Thus, the time complexity is $\theta(n)$ if $k = O(n)$

Observation: k

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $\theta(n)$ time.

The array B holds the sorted output, and the array C is used for temporary working storage for the number of elements encountered.

An important property of counting sort is that it is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. For radix sort to work correctly, counting sort must be stable.

Characteristics

- **Stable**: Yes
- **In-place**: No
- **Online**: No
- **Divide and Conquer**: No
- **Parallelism**: No
- **Recursive vs Iterative**: Iterative
- **Space Complexity**: O(n + k)
- **Comparison-based**: No

2.2.2 Radix Sort

Radix sort processes numbers digit by digit without direct number comparisons. It sorts numbers by repeatedly ordering them based on each digit position, starting from the least significant digit and moving towards the most significant one. For d-digit numbers where each digit has k possible values ($1 \leq k \leq d$), radix sort achieves a time complexity of $\theta(n)$.

Intuitively, radix sort breaks down each input integer into "columns" when they are written in some base b. Here d is the number of columns, and each column is a digit in the base-b representation of the number.In order for radix sort to work correctly, the digit sorts must be **stable**.

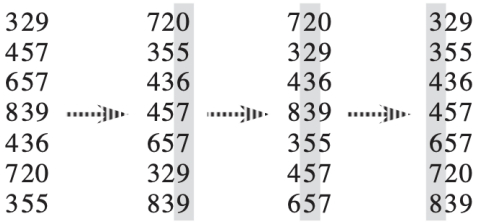


Figure 2.10: Radix Sort [1]

Algorithm 11 Radix Sort (A, d)

```
1: for i = 1 to d do
2:     Use a stable sort to sort array A on digit i
3: end for
4:                                   ▷ Time complexity: θ(d(n+k))
```

Lemma 1. *Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts them in $\theta(d(n+k))$ time if the stable sort it uses takes $\theta(n+k)$.*

Proof. The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to k 1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\theta(n+k)$. There are d passes, and so the total time for radix sort is $\theta(d(n+k))$. □

Characteristics

- **Stable**: Yes
- **In-place**: No
- **Online**: No
- **Divide and Conquer**: No
- **Parallelism**: No
- **Recursive vs Iterative**: Iterative
- **Space Complexity**: O(n + k)
- **Comparison-based**: No

3 Searching Algorithms

The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, we can use a search tree both as a dictionary and as a priority queue. Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\log n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same oper- ations take $\Theta(n)$ worst-case time. We shall see that the expected height of a randomly built binary search tree is only $\Theta(\log n)$, so that the **expected** time for these operations is $\Theta(\log n)$.

3.1 Binary Search Trees

We can represent a binary tree by a linked data structure in which each node is an object. In addition to a **key** and **satellite** data, each node contains attributes **left**, **right**, and **p** that point to the nodes corresponding to its left child, its right child, and its parent, respectively.

- **Input**: a SORTED sequence of n keys
- **Output**: a position i in the sequence where the key is located

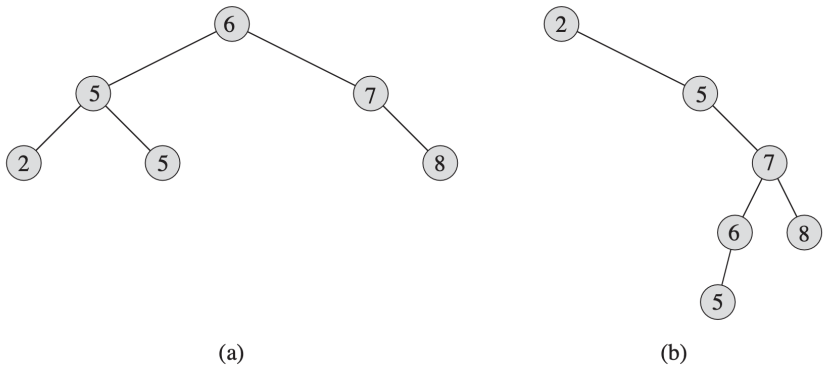


Figure 3.1: Binary Search Tree

The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**:

Definition: ***Binary Search Tree Property***

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk**. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a **preorder tree walk** prints the root before the values in either subtree, and a **postorder tree walk** prints the root after the values in its subtrees.)

Algorithm 12 Inorder Tree Walk

```
1: if x ≠ NIL then
2:     INORDER-TREE-WALK(x.left)
3:     print x.key
4:     INORDER-TREE-WALK(x.right)
5: end if
```

It takes $\theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree: once for its left child and once for its right child.

Theorem 3.1.1. *The complexity of INORDER-TREE-WALK on a n nodes binary search tree is* $\Theta(n)$.

Proof.

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c, \end{aligned}$$

Where c and d are constants that represent the time per call to INORDER-TREE-WALK and the time per print statement, respectively. k is the number of nodes in the left subtree of x, while $n-k-1$ is the number of nodes in the right subtree of x. The base case occurs when $n = 0$, in which case the running time is constant. The recurrence is linear, and so the running time of INORDER-TREE-WALK is $\Theta(n)$. □

3.1.1 Quering a Binary Search Tree

We often need to search for a key stored in a binary search tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine these operations and show how to support each one in time $O(h)$ on any binary search tree of height h.

Searching

Definition: ***Searching***

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

Algorithm 13 TREE-SEARCH(x, k)

```
1: while x ≠ NIL and k ≠ x.key do
2:     if k < x.key then
3:         x ← x.left
4:     else
5:         x ← x.right
6:     end if
7: end while
8: return x
```

Minimum and Maximum

We can always find an element in a binary search tree whose key is a minimum by following left child pointers from the root until we encounter a NIL.

Algorithm 14 TREE-MINIMUM(x)

```
1: while x.left ≠ NIL do
2:     x ← x.left
3: end while
4: return x
```

Same for the maximum:

Algorithm 15 TREE-MAXIMUM(x)

```
1: while x.right ≠ NIL do
2:     x ← x.right
3: end while
4: return x
```

Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Successor and Predecessor

Given a node in a binary search tree, sometimes we need to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than x:key. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree:

Algorithm 16 TREE-SUCCESSOR(x)

```
1: if x.right ≠ NIL then
2:     return TREE-MINIMUM(x.right)
3: end if
4: y ← x.p
5: while y ≠ NIL  and  x = y.right do
6:     x ← y
7:     y ← y.p
8: end while
9: return y
```

3.1.2 Insertion and Deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold.

Insertion

The procedure takes a node z for which $z.key = v, z.left = NIL$ and $z.right = NIL$. It modifies the tree T and some of the attributes of z in such a way that it inserts z into the appropriate position in the tree.

Algorithm 17 TREE-INSERT(T, z)

```
 1: y ← NIL
 2: x ← T.root
 3: while x ≠ NIL do
 4:     y ← x
 5:     if z.key < x.key then
 6:         x ← x.left
 7:     else
 8:         x ← x.right
 9:     end if
10: end while
11: z.p ← y
12: if y = NIL then
13:     T.root ← z
14: else if z.key < y.key then
15:     y.left ← z
16: else
17:     y.right ← z
18: end if
```

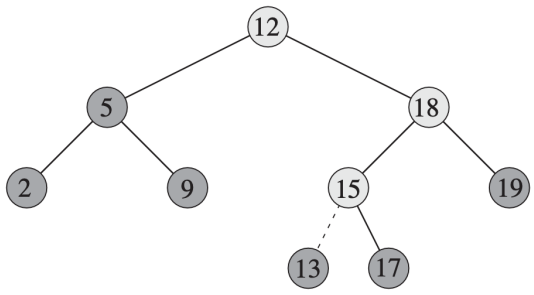


Figure 3.2: Insertion in a Binary Search Tree

Deletion

- **Case 1**: If node z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- **Case 2**: If node z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- **Case 3**: If node z has two children, then we find z's successor y, which lies in z's right subtree and has no left child. We want to splice y out of its current location and have it replace z in the tree.

Algorithm 18 TREE-DELETE(T, z)

```
 1: if z.left = NIL then
 2:     TRANSPLANT(T, z, z.right)
 3: else if z.right = NIL then
 4:     TRANSPLANT(T, z, z.left)
 5: else
 6:     y ← TREE-MINIMUM(z.right)
 7:     if y.p ≠ z then
 8:         TRANSPLANT(T, y, y.right)
 9:         y.right ← z.right
10:         y.right.p ← y
11:     end if
12:     TRANSPLANT(T, z, y)
13:     y.left ← z.left
14:     y.left.p ← y
15: end if
```

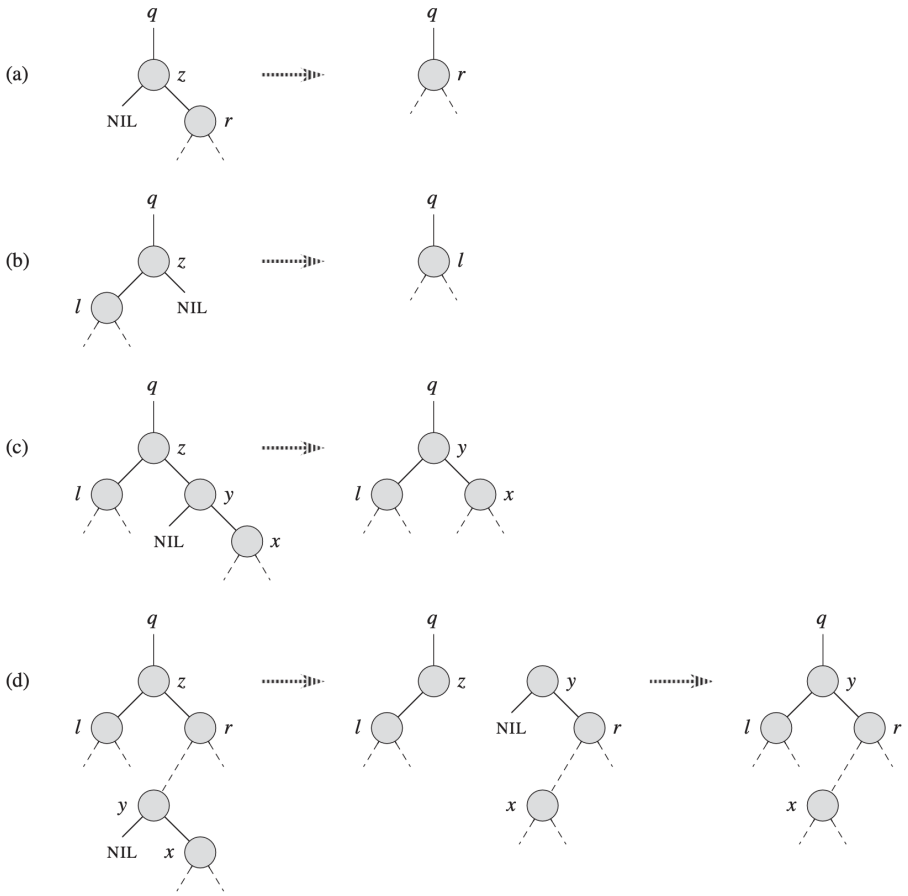


Figure 3.3: Deletion in a Binary Search Tree.
(a) Node z has no left child.
(b) Node z has a left child but not a right child.
(c) Node z has two children.
(d) Node z has two children, and its successor $y \neq r$ lies between the subtree rooted at r.

Algorithm 19 TRANSPLANT(T, u, v)

```
1:  if u.p = NIL then
2:      T.root ← v
3:  else if u = u.p.left then
4:      u.p.left ← v
5:  else
6:      u.p.right ← v
7:  end if
8:  if v ≠ NIL then
9:      v.p ← u.p
10: end if
```

Sorting

Algorithm 20 BST-SORT(T)

```
1: T ← ∅
2: for i = 1 to n do
3:     T.root ← TREE-INSERT(T, i)
4: end for
5: INORDER-TREE-WALK(T.root)
6:                                   ▷ Time complexity: Θ(n log n)
```

INORDER-TREE-WALK takes $\Theta(n)$ time, and we call it n times, so the total time is $\Theta(n \log n)$.

3.2 Red-Black Trees

It is a BST that enjoys the following properties:

- Every node is colored, either **red** or **black**.
- The root is **black**.
- Every leaf (NIL) is **black**.
- If a **red** node has children, then the children are **black**.
- All paths from a node to its leaves have the same **black** height.

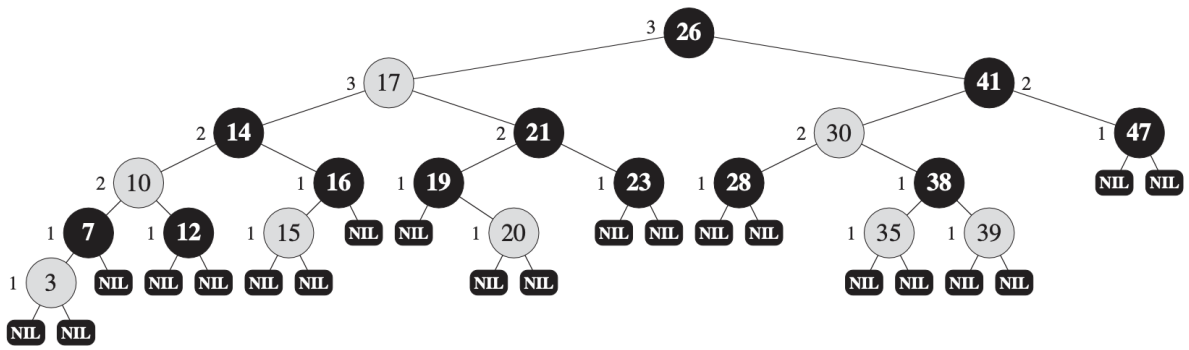


Figure 3.4: Red-Black Tree

Theorem 3.2.1. *Any Red-Black Tree with n keys has the height* $h \leq 2\log(n+1)$

Proof. We start by showing that the subtree rooted at any node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf (T.nil), and the subtree rooted at x indeed contains at least

$$2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$$

internal nodes.

For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $\mathrm{bh}(x)$ or $\mathrm{bh}(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least

$$2^{\mathrm{bh}(x)-1} - 1$$

internal nodes. Thus, the subtree rooted at x contains at least

$$\left(2^{\mathrm{bh}(x)-1} - 1\right) + \left(2^{\mathrm{bh}(x)-1} - 1\right) + 1 = 2^{\mathrm{bh}(x)} - 1$$

internal nodes, which proves the claim. □

Rotations

The operations TREE-INSERT and TREE-DELETE can violate the properties of a red-black tree. To restore these properties, we use two operations: LEFT-ROTATE and RIGHT-ROTATE.

Algorithm 21 LEFT-ROTATE(T,x)

```
 1: y ← x.right
 2: x.right ← y.left
 3: if y.left ≠ T.nil then
 4:     y.left.p ← x
 5: end if
 6: y.p ← x.p
 7: if x.p = T.nil then
 8:     T.root ← y
 9: else if x = x.p.left then
10:     x.p.left ← y
11: else
12:     x.p.right ← y
13: end if
14: y.left ← x
15: x.p ← y
```

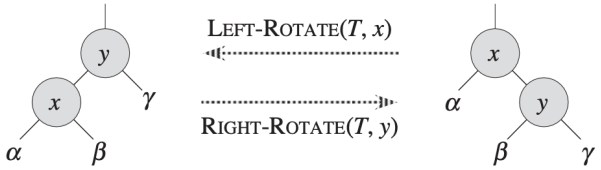


Figure 3.5: Left Rotation

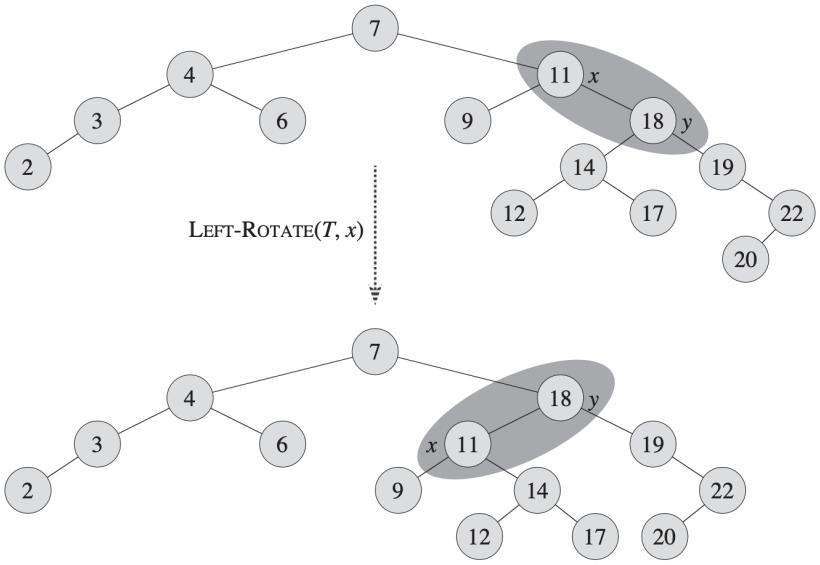


Figure 3.6: Example of rotation in a Red-Black Tree

Insertion

We can insert a new node into a red-black tree by using the following procedure. The procedure takes as input the tree T and a node z to insert into T. It modifies the tree as necessary to maintain the red-black properties.

Algorithm 22 RB-INSERT(T,z)

```
 1: y ← T.nil
 2: x ← T.root
 3: while x ≠ T.nil do
 4:     y ← x
 5:     if z.key < x.key then
 6:         x ← x.left
 7:     else
 8:         x ← x.right
 9:     end if
10: end while
11: z.p ← y
12: if y = T.nil then
13:     T.root ← z
14: else if z.key < y.key then
15:     y.left ← z
16: else
17:     y.right ← z
18: end if
19: z.left ← T.nil
20: z.right ← T.nil
21: z.color ← RED
22: RB-INSERT-FIXUP(T,z)
```

Time complexity: $\Theta(\log n)$

Why RB-INSERT-FIXUP is Necessary

The `RB-INSERT-FIXUP` algorithm is necessary to maintain the properties of a red-black tree after the insertion of a new node. When a new node is inserted, it is initially colored red. This can potentially violate the red-black tree properties, specifically:

- Property 2: The root must be black.
- Property 4: Both children of every red node must be black (no two red nodes can be adjacent).
- Property 5: Every path from a node to its descendant NIL nodes must have the same number of black nodes.

The `RB-INSERT-FIXUP` algorithm corrects any violations of these properties by performing a series of color changes and rotations. This ensures that the tree remains balanced, with a height of at most $2\log(n+1)$, which guarantees that the basic dynamic set operations (such as search, insert, and delete) can be performed in $O(\log n)$ time.

Algorithm 23 RB-INSERT-FIXUP(T,z)

```
1:  while z.p.color = RED do
2:      if z.p = z.p.p.left then
3:          y ← z.p.p.right
4:          if y.color = RED then
5:              z.p.color ← BLACK
6:              y.color ← BLACK
7:              z.p.p.color ← RED
8:              z ← z.p.p
9:          else
10:             if z = z.p.right then
11:                 z ← z.p
12:                 LEFT-ROTATE(T,z)
13:             end if
14:             z.p.color ← BLACK
15:             z.p.p.color ← RED
16:             RIGHT-ROTATE(T,z.p.p)
17:         end if
18:     else
19:         # Mirror case: perform same operations
20:         # with left/right reversed
21:     end if
22: end while
23: T.root.color ← BLACK
```

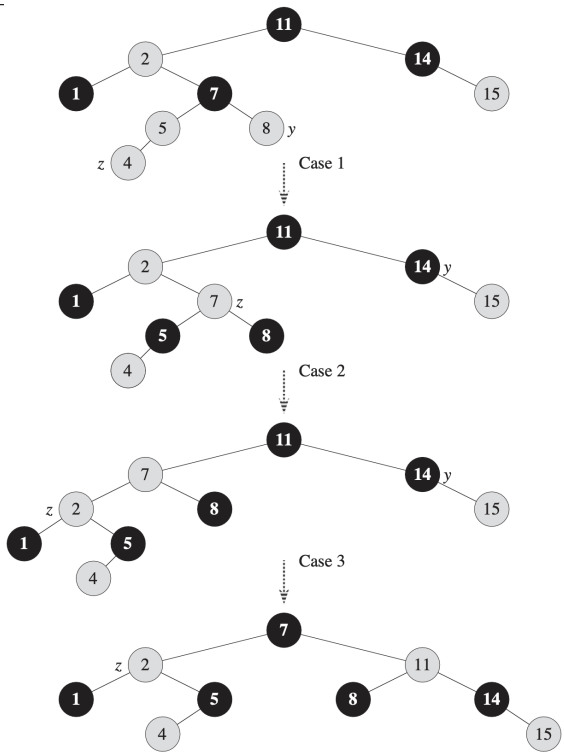


Figure 3.7: Insertion in a Red-Black Tree

Explanation of RB-INSERT-FIXUP Algorithm

The `RB-INSERT-FIXUP` algorithm is essential for maintaining the red-black tree properties after an insertion. Here is a step-by-step explanation of why it is necessary:

1. **Initial Insertion**: When a new node z is inserted, it is colored red. This is done to maintain property 5 (black-height property) without immediately violating it. However, this can lead to a violation of property 4 (no two red nodes can be adjacent).

2. **Fixing Violations**: It checks for violations of the red-black properties, specifically property 4. If z's parent $z.p$ is red, then there is a violation because z and $z.p$ are both red.
3. **Case Handling**: The algorithm handles violations through a series of cases:
   - **Case 1**: If z's uncle y is red, both $z.p$ and y are recolored to black, and $z.p.p$ is recolored to red. The algorithm then continues to check for violations up the tree.
   - **Case 2**: If z is a right child and $z.p$ is a left child, a left rotation is performed on $z.p$. This transforms the situation into Case 3.
   - **Case 3**: $z.p$ is recolored to black, $z.p.p$ is recolored to red, and a right rotation is performed on $z.p.p$.
4. **Termination**: The algorithm terminates when the root is reached or when no violations are found. Finally, the root is colored black to ensure property 2 (the root is black).

By performing these steps, the `RB-INSERT-FIXUP` algorithm ensures that all red-black tree properties are restored, maintaining the tree's balanced structure and guaranteeing efficient performance for subsequent operations.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\log n)$, lines 1-16 of RB-INSERT take $O(\log n)$ time. In RB-INSERT- FIXUP, the while loop repeats only if case 1 occurs, and then the pointer ´ moves two levels up the tree. The total number of times the while loop can be executed is therefore $O(\log n)$. Thus, RB-INSERT takes a total of $O(\log n)$ time.

Observation: ***Stopping condition***

It stops when there is one element in the array, where $p = r$.

4 Graph Algorithms

Definition: *Graph*

A graph G is a pair (V,E), where V is a set of vertices and E is a set of edges. Each edge is a pair of vertices.

We can choose between two standard ways to represent a graph $G = (V,E)$: as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed and undirected graphs. Because the adjacency-list representation provides a compact way to represent sparse graphs, it is usually the method of choice.

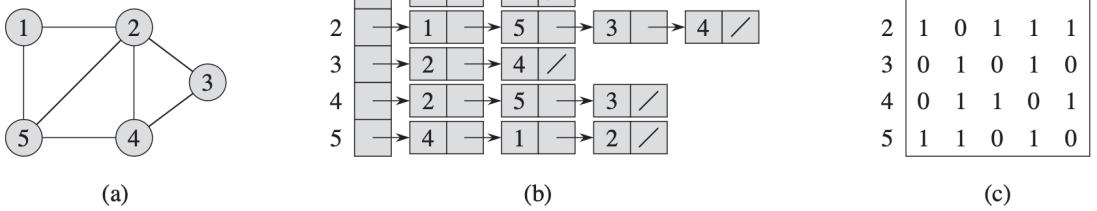


Figure 4.1: Undirected raph

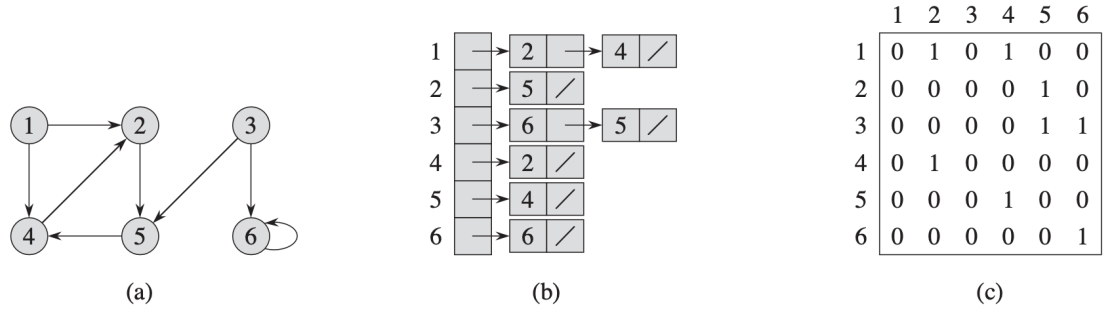


Figure 4.2: Directed graph

- **Path**: A path in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$. The length of the path is the number of edges in the path, which is $n-1$.
- **Walk**: A walk in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$. The length of the walk is the number of edges in the walk, which is $n-1$.
- **Cycle**: A cycle in a graph is a path of length at least 1, whose first and last vertices are the same.
- **Connected component**: A connected component of an undirected graph $G = (V,E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, there is a path from u to v.

Graph Representation

The **adjacency-list representation** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each vertex $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ is a list of all the vertices adjacent to u in G. We can adapt the adjacency-list representation to represent weighted graphs by storing, for each vertex v, not only the vertices adjacent to v but also the weights of the edges incident on v.

The **adjacency-matrix representation** of a graph $G = (V, E)$ consists of a $|V| \times |V|$ matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency-matrix representation is particularly convenient when the graph is dense, that is, when $|E|$ is close to $|V|^2$.

4.1 Breadth-First Search

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a **breadth-first tree** with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges. The algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

Observation: *Data structure used*

The algorithm uses a first-in first-out queue Q to manage the set of gray vertices.

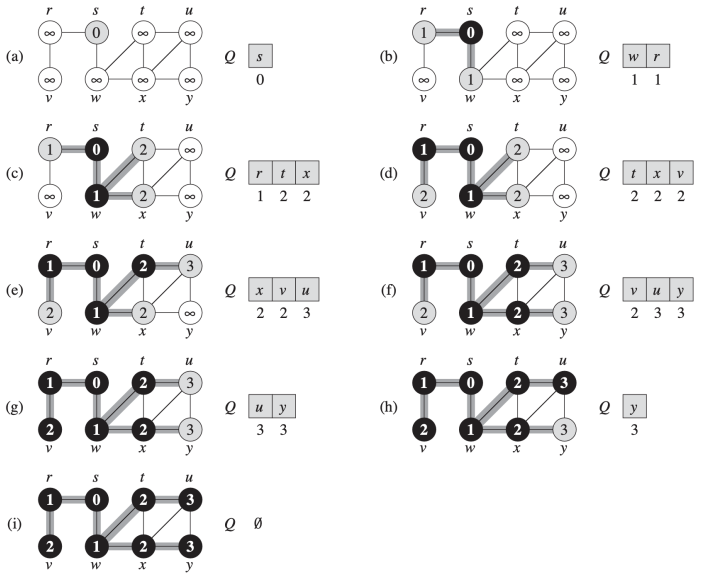


Figure 4.3: Breadth-First Search

Algorithm 24 BFS(G, s)

```
 1: for each vertex u ∈ V - {s} do                    ▷ Initialization: O(|V|)
 2:     color[u] ← WHITE
 3:     d[u] ← ∞
 4:     π[u] ← NIL
 5: end for
 6:
 7: color[s] ← GRAY                                   ▷ Visit of the source: O(1)
 8: d[s] ← 0
 9: π[s] ← NIL
10: Q ← ∅
11: ENQUEUE(Q, s)
12:
13: while Q ≠ ∅ do                                    ▷ Visit of the vertices: O(|E|)
14:     u ← DEQUEUE(Q)
15:     for each vertex v ∈ Adj[u] do
16:         if color[v] = WHITE then
17:             color[v] ← GRAY
18:             d[v] ← d[u] + 1
19:             π[v] ← u
20:             ENQUEUE(Q, v)
21:         end if
22:     end for
23:     color[u] ← BLACK
24: end while
```

Analysis

After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V+E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest Paths

Definition: *Shortest Path Distance*

Define the **shortest path distance** $\delta(s,v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v, then $\delta(s,v) = \infty$.

Theorem 4.1.1. *Let $G = (V,E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u,v) \in E$, $\delta(s,v) \leq \delta(s,u) + 1$.*

Proof. If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u,v), and thus the inequality holds. If u is not reachable from s, then $\delta(s,v) = \infty$, and the inequality holds. □

Theorem 4.1.2. *Let $G = (V,E)$ be a directed or undirected graph, and suppose BFS is run on G from a given source $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by the BFS satisfies $v.d \geq \delta(s,v)$.*

Proof. We prove this theorem by induction on the number of vertices in the breadth-first tree.

Base case: The base case is the source vertex s. Initially, $s.d = 0$ and $\delta(s,s) = 0$. Therefore, $s.d = \delta(s,s)$, and the base case holds.

Inductive step: Assume that for any vertex u in the breadth-first tree, $u.d \geq \delta(s,u)$. We need to show that for any vertex v adjacent to u, $v.d \geq \delta(s,v)$.

When v is first discovered, it is enqueued and $v.d$ is set to $u.d + 1$. By the inductive hypothesis, $u.d \geq \delta(s,u)$. Therefore, $v.d = u.d + 1 \geq \delta(s,u) + 1$. Since (u,v) is an edge in the graph, $\delta(s,v) \leq \delta(s,u) + 1$. Combining these inequalities, we get $v.d \geq \delta(s,u) + 1 \geq \delta(s,v)$.

Thus, by induction, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s,v)$. □

Theorem 4.1.3. *Let $G = (V,E)$ be a directed or undirected graph and suppose BFS is run on G from a given source $s \in V$. Then during its execution, BFS discovers every vertex $v \in V$ that is reachable from s, and upon termination, $v.d = \delta(s,v)$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.*

Proof. Assume, for the purpose of contradiction, that some vertex v receives a d-value not equal to its shortest-path distance $\delta(s,v)$. Let v be the vertex with the minimum $\delta(s,v)$ that receives such an incorrect d-value. Clearly, $v \neq s$. Since $v.d \geq \delta(s,v)$, we have $v.d > \delta(s,v)$. Vertex v must be reachable from s; otherwise, $\delta(s,v) = \infty > v.d$. Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s,v) = \delta(s,u) + 1$. Because $v.d > \delta(s,v)$, it follows that $v.d = \delta(s,u) + 1 = u.d + 1$.

Now consider when BFS dequeues u from Q. At this time, v can be white, gray, or black:

- If v is white, line 15 sets $v.d = u.d + 1$, contradicting $v.d > \delta(s,v)$.
- If v is black, v has already been removed from the queue, so $v.d \leq u.d$, again contradicting $v.d > \delta(s,v)$.
- If v is gray, it was painted gray during the dequeue of some vertex w, removed from Q earlier than u. Since $w.d \leq u.d$, it follows that $v.d = w.d + 1 \leq u.d + 1$, again contradicting $v.d > \delta(s,v)$.

Thus, $v.d = \delta(s,v)$ for all $v \in V$. Since all vertices reachable from s must be discovered, the proof is complete. Finally, if $v.\pi = u$, then $v.d = u.d + 1$, ensuring that a shortest path from s to v is obtained by traversing $(v.\pi, v)$. □

4.2 Single Source Shortest Paths

In a **shortest-path problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-values weights. the **weight** $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the **shortest-path weight** $\delta(u, v)$ from u to v as the minimum weight of any path from u to v:

$$\delta(u,v) = \begin{cases} \min\{w(p) : \text{p is a path from u to v}\} & \text{if there is a path from u to v,} \\ \infty & \text{otherwise.} \end{cases}$$

Warning: *BFS*

BFS is used to compute shortest paths in unweighted graphs. In weighted graphs, BFS can be used to compute shortest paths only when all edge weights are equal.

Theorem 4.2.1. *Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k. For any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.*

Proof. Decompose the path p into three segments: p_{0i} from v_0 to v_i, p_{ij} from v_i to v_j, and p_{jk} from v_j to v_k. Then, the weight of p is given by:

$$w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk}).$$

Assume there exists a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, the path $p' = \langle p_{0i}, p'_{ij}, p_{jk} \rangle$ would have weight:

$$w(p') = w(p_{0i}) + w(p'_{ij}) + w(p_{jk}),$$

which is less than $w(p)$. This contradicts the assumption that p is a shortest path from v_0 to v_k. Thus, p_{ij} must be a shortest path from v_i to v_j. □

Cycles

- **Positive cycle**: A cycle whose edges have a positive sum of weights.
- **Negative cycle**: A cycle whose edges have a negative sum of weights. It makes the cost of a path not well defined cause each cycle creates lower cost each time.

A shortest path cannot contain a negative cycle. If a negative cycle is reachable from the source vertex, then there is no shortest path, since the path can be made as short as desired by traversing the negative cycle arbitrarily many times. Nor it can contain a positive cycle, since the path can be made shorter by removing the cycle.

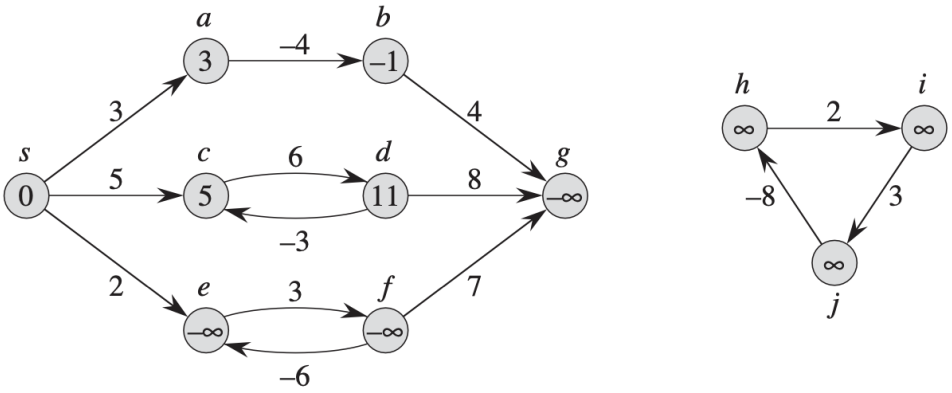


Figure 4.4: Negative cycle

Representation

To compute shortest paths, not only the weights but also the vertices on the paths are maintained. For a graph $G = (V, E)$, each vertex $v \in V$ has a predecessor $\pi(v)$, which is either another vertex or NIL. The π values represent the chain of predecessors that define the shortest path from a source vertex s to v. The procedure `PRINT-PATH` (G, s, v) can reconstruct this shortest path.

In shortest-path algorithms, the π values define a *predecessor subgraph* $G_\pi = (V_\pi, E_\pi)$:

$$V_\pi = \{v \in V : \pi(v) \neq \text{NIL}\} \cup \{s\}, E_\pi = \{(\pi(v), v) \in E : v \in V_\pi \setminus \{s\}\}.$$

At termination, G_π forms a *shortest-path tree*, which is a rooted tree with:

1. V', the set of vertices reachable from s in G.
2. A root at s and edges E' such that:
   - $V' \subseteq V$,
   - $E' \subseteq E$.
3. For all $v \in V'$, the unique simple path from s to v in G' is the shortest path in G.

A shortest-path tree extends the concept of a breadth-first tree to account for edge weights.

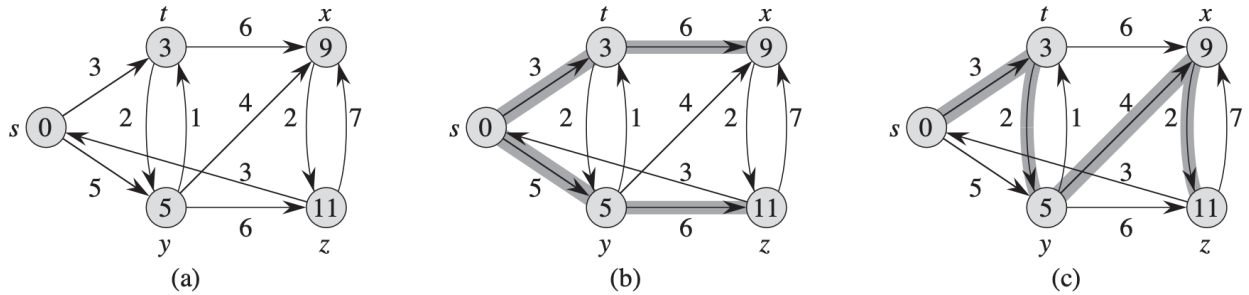


Figure 4.5: Shortest-path representation

Relaxation

For each vertex $v \in V$, we maintain an attribute $v.d$ that is an upper bound on the weight of a shortest path from source s to v. We call $v.d$ a **shortest-path estimate**. We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$-time procedure:

Algorithm 25 INITIALIZE-SINGLE-SOURCE(G, s)

```
1: for each vertex v ∈ V do
2:     v.d ← ∞
3:     π(v) ← NIL
4: end for
5: s.d ← 0
```

The process of **relaxing** an edge (u, v) consists of testing whether we can improve the shortest path to found so far by going through u and, if so, updating $v.d$ and $v.\pi$. A relaxation step may decrease the value of $v.d$ and update $v.\pi$. The following code performs a relaxation step on edge (u, v):

Algorithm 26 RELAX(u, v, w)

```
1: if v.d > u.d + w(u, v) then
2:     v.d ← u.d + w(u, v)
3:     v.π ← u
4: end if
```

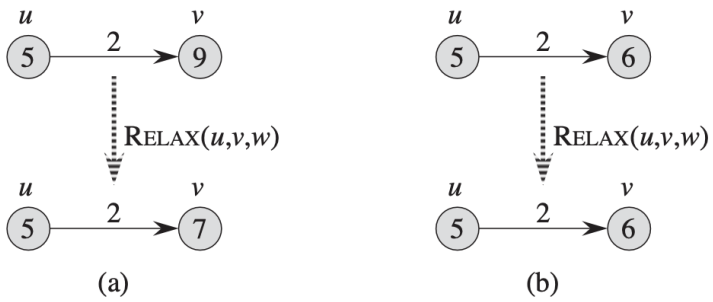


Figure 4.6: Relaxation

Bellman-Ford Algorithm

Algorithm 27 Bellman-Ford(G, w, s)

```
1: INITIALIZE-SINGLE-SOURCE(G, s)
2: for i = 1 to |V| - 1 do
3:     for each edge (u, v) ∈ E do
4:         RELAX(u, v, w)
5:     end for
6: end for
7: for each edge (u, v) ∈ E do
8:     if v.d > u.d + w(u, v) then
9:         return FALSE
10:    end if
11: end for
12: return TRUE
```

Warning: *Bellman-Ford*

This algorithm can be used to detect negative cycles. If the algorithm returns `FALSE`, then there is a negative cycle reachable from the source vertex. This means that it **does not** work with negaive cycles.

4.3 Depth-First Search

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of 's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Unlike breadth-first search,whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources. Therefore, we define the predecessor subgraph of a depth-first search slightly differently from that of a breadth-first search: we let $G_\pi = (V, E_\pi)$, where $E_\pi = \{(\pi(v), v) : v \in V \text{ and } \pi(v) \neq \text{NIL}\}$.

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees.

As in breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely.

Besides creating a depth-first forest, depth-first search also assigns each vertex v two timestamps: a discovery time $v.d$ when v is first visited (colored gray), and a finishing time $v.f$ when v's adjacency list is fully examined (colored black).

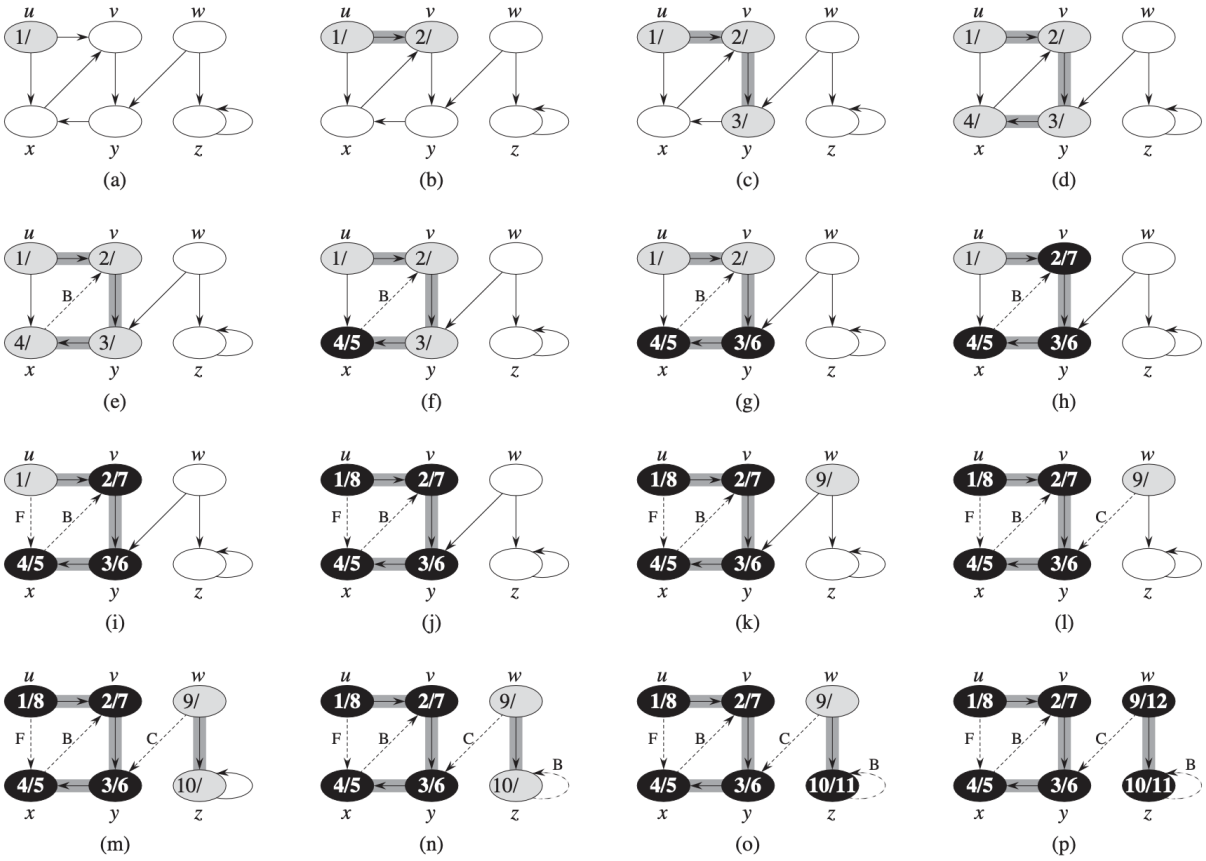


Figure 4.7: Depth-First Search

Analysis

The running time of depth-first search is $O(V+E)$, which is the same as the running time of breadth-first search. The time is proportional to the sum of the lengths of the adjacency lists of all the vertices. The depth-first search algorithm is a simple algorithm that is easy to implement and efficient when the graph is represented by the adjacency-list structure. The running time of DFS consists of two main parts:

- The loops on lines 1-3 and lines 5-7 of DFS take time $\Theta(V)$, where V is the number of vertices in the graph.
- The time for executing the `DFS-VISIT` procedure.

Using aggregate analysis:

- The procedure `DFS-VISIT` is called exactly once for each vertex $v \in V$, since a vertex is painted gray the first time it is visited.
- During an execution of `DFS-VISIT` (G,v), the loop on lines 4-7 is executed $|Adj[v]|$ times, where $Adj[v]$ is the adjacency list of vertex v.

The total cost of all executions of lines 4-7 is:

$$\sum_{v \in V} |Adj[v]| = \Theta(E),$$

where E is the number of edges in the graph.

Thus, the overall running time of DFS is:

$$\Theta(V+E).$$

4.4 Topological Sort

A **topological sort** of a directed acyclic graph $G=(V,E)$ is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering. A topological sort of a directed acyclic graph is not unique. If the graph has a cycle, then no linear ordering is possible.

Algorithm 28 TOPOLOGICAL-SORT(G)

call DFS(G) to compute finishing times $v.f$ for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices

We can perform a topological sort in time $O(V+E)$, since the time to call DFS is $O(V+E)$ and the time to insert each of the V vertices onto the front of the linked list is $O(1)$.

4.5 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G=(V,E)$ for the case in which all edge weights are nonnegative. It maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V-S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

Because it always chooses the "lightest" or "closest" vertex in $V-S$ to add to set S, we say that it uses a greedy strategy.

Theorem 4.5.1. *Dijkstra's algorithm, run on a weighted, directed graph $G=(V,E)$ with non-negative weight function w and source s, terminates with $u.d=\delta(s,u) \forall u \in V$.*

Proof. We prove this theorem by induction on the number of vertices in the set S.

Base case: Initially, $S=\emptyset$ and $s.d=0=\delta(s,s)$. For all other vertices $v \in V \setminus \{s\}$, $v.d=\infty \geq \delta(s,v)$. Thus, the base case holds.

Inductive step: Assume that for any vertex $u \in S$, $u.d=\delta(s,u)$. We need to show that when a vertex v is added to S, $v.d=\delta(s,v)$.

Let v be the next vertex added to S. By the inductive hypothesis, for all vertices $u \in S$, $u.d=\delta(s,u)$. Since v is chosen as the vertex with the minimum shortest-path estimate, $v.d \leq u.d+w(u,v)$ for all edges $(u,v) \in E$ where $u \in S$.

Consider any path p from s to v. Let u be the last vertex on the path p that is in S. Then, the subpath from s to u is a shortest path, and $u.d=\delta(s,u)$. The weight of the subpath from u to v is at least $w(u,v)$. Therefore, the total weight of the path p is at least $\delta(s,u)+w(u,v)=u.d+w(u,v) \geq v.d$. Thus, $v.d \leq \delta(s,v)$.

Since $v.d$ is a shortest-path estimate and cannot be less than the actual shortest-path distance, $v.d \geq \delta(s,v)$. Combining these inequalities, we get $v.d=\delta(s,v)$.

Thus, by induction, for each vertex $u \in V$, the value $u.d$ computed by Dijkstra's algorithm satisfies $u.d=\delta(s,u)$. □

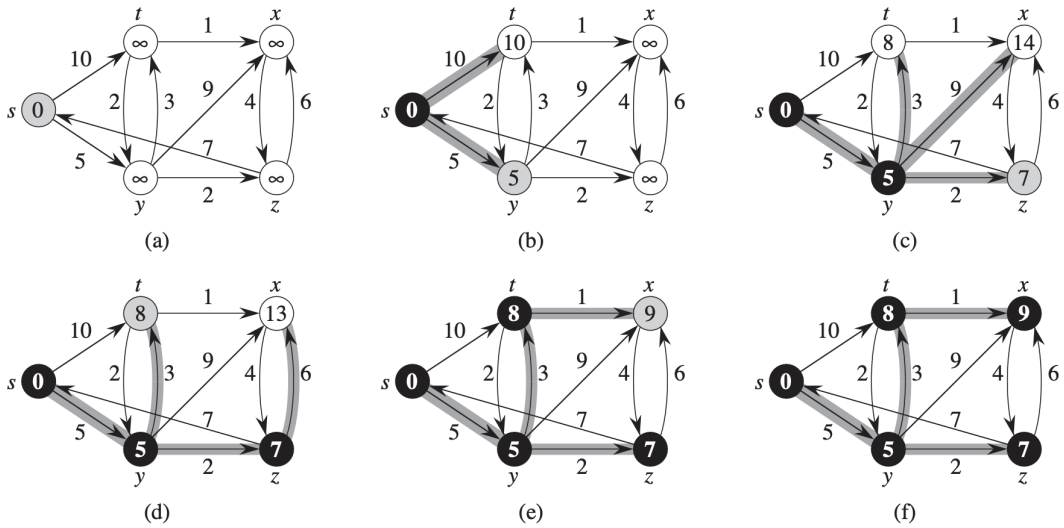


Figure 4.8: Dijkstra's Algorithm

Analysis

The running time of Dijkstra's algorithm depends on how the priority queue is implemented. Let's analyze the running time for different implementations of the priority queue.

- **Using a binary min-heap:**
  - The initialization of the single source takes $O(V)$ time.
  - The priority queue operations (insert, extract-min, and decrease-key) take $O(\log V)$ time each.
  - The algorithm performs $|V|$ extract-min operations, each taking $O(\log V)$ time, for a total of $O(V \log V)$.
  - The algorithm performs $|E|$ decrease-key operations, each taking $O(\log V)$ time, for a total of $O(E \log V)$.

  Therefore, the total running time using a binary min-heap is $O((V+E)\log V)$.
- **Using a Fibonacci heap:**
  - The initialization of the single source takes $O(V)$ time.
  - The insert and decrease-key operations take $O(1)$ amortized time each.
  - The extract-min operation takes $O(\log V)$ amortized time.
  - The algorithm performs $|V|$ extract-min operations, each taking $O(\log V)$ amortized time, for a total of $O(V \log V)$.
  - The algorithm performs $|E|$ decrease-key operations, each taking $O(1)$ amortized time, for a total of $O(E)$.

  Therefore, the total running time using a Fibonacci heap is $O(V \log V + E)$.

In summary, the running time of Dijkstra's algorithm is $O((V+E)\log V)$ when using a binary min-heap and $O(V \log V + E)$ when using a Fibonacci heap. The latter is more efficient for dense graphs where $|E|$ is large.

5 Data Mining

5.1 Exact Pattern Matching

Consider two strings: a **text** string $T[1,...,n]$ of length n and a **pattern** string $P[1,...,m]$ of length m. Both strings are defined over a finite alphabet Σ (e.g., ASCII characters, DNA nucleotides, etc.).

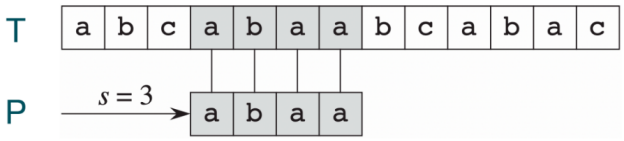
We say that pattern P occurs with shift s (equivalently, occurs at position $s+1$) in text T if:

- **the shift is within valid bounds:** $0 \leq s \leq n-m$
- **the substring matches exactly:** $T[s+1,...,s+m] = P[1,...,m]$

When P occurs with shift s in T, we call s a **valid shift**. Conversely, if P does not occur at shift s, we call it an **invalid shift**.

By convention, we refer to:

- T as the **text** (the longer string being searched)
- P as the **pattern** (the shorter string being sought)



5.1.1 Naive Algorithm (Shifts Version)

Algorithm 29 Naive Algorithm for Exact Pattern Matching

Input: a text T of length n and a pattern P of length m
Output: all the occurrences (or valid shifts) of P in T

```
1:  sol ← ∅
2:  for s ← 0 to n − m do
3:      i ← 1
4:      while i ≤ m and T[s + i] = P[i] do
5:          i ← i + 1                                  ▷ i scans the pattern
6:      end while
7:      if i > m then                                  ▷ if the pattern is found
8:          sol ← sol ∪ {s}
9:      end if
10: end for
11: return sol;
```

The internal `While loop` scans m chars of the text, with a complexity of $O(m)$, and the external `For loop` scans the whole text, with a complexity of $O(n)$. So the overall complexity is $O(n \cdot m)$.

5.1.2 Knuth-Morris-Pratt (KMP) Algorithm

The idea is to preprocess P to skip unnecessary comparisons computing how P matches against itself. This is achieved by computing an array $\pi = [1,...,|P|]$ such that $\pi[q]$ is the length of the longest proper suffix of $P[1,...,q]$ which is also a prefix of P.

First, consider one prefix of P at a time in increasing order of length. Second, for each prefix, check each proper suffix in decreasing order of length.

Example:

Example For a given pattern `P = ababaca` , then we compute π as:

P	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

Computing π is useful because we know that if the first q chars of P matched with some shift S the next potentially valid shift is $S' = S + (q - \pi[q])$. A string of length m has m proper suffixes/prefixes.

Algorithm 30 Compute_π(P)

Input: a pattern P of length m
Output: an array π of length m containing the length of the longest proper suffix of $P[1,...,q]$ which is also a prefix of P

```
1: allocate π[1,...,m]
2: π ← 0 for all i = 1,...,m
3: k ← 0
4: for q ← 2 to m do
5:     while k > 0 and P[k+1] ≠ P[q] do
6:         k ← π[k]
7:     end while
8:     if P[k+1] = P[q] then
9:         k ← k+1
10:    end if
11:    π[q] ← k
12: end for
13: return π
```

The complexity analysis of `Compute_π` is based on the following observations:

- The increase of k is at most $|P| - 1$ $(= m - 1)$ across all iterations
- k is always decreased in the While loop
- k is never negative

Adding all costs: $O(m)$ for initialization + $O(m)$ for For loop operations (excluding while) + $O(m)$ for amortized cost of while loop and condition checks + $O(1)$ for return.

Therefore, the total complexity is $\Theta(m)$. This type of analysis, where an operation's cost varies per iteration but has a bounded total cost across all iterations, is called ***amortized analysis***.

Prefix Function

π can also be seen as a prefix function such that:

$$\pi : [1,m] \to [0,m-1]$$

where

$$\pi[q] = max(k|k < q \wedge P[1,...,k] \text{ is a proper suffix of } P[1,...,q])$$

The **prefix function iteration** is the sequence:

$$\pi^*[q] = \{\pi[q], \pi^{(2)}[q] = \pi[\pi[q]], ..., \pi^{(t)}[q]\}$$

where t is the smallest value such that:

$$\pi^{(t)}[q] = 0 \wedge \pi^{(i)}[q] = \pi[\pi^{(i-1)}[q]]$$

$\pi^*[q]$ contains the lengths of every proper suffix of the prefix $P[1,...,q]$.

Example: *prefix function*

For a given pattern $P = abababab$ and $q = 7$, let's compute the prefix function iteration:

$\pi[7] = 5$ (since *ababa* is the longest proper suffix that matches a prefix)
$\pi^{(2)}[7] = \pi[5] = 3$ (next longest is *aba*)
$\pi^{(3)}[7] = \pi[3] = 1$ (then *a*)
$\pi^{(4)}[7] = \pi[1] = 0$ (finally empty string)

Therefore $t = 4$ iterations are needed to reach 0.
The complete prefix function iteration is:

$$\pi^*[7](abababab) = \{5 \text{ (ababab)}, 3 \text{ (abab)}, 1 \text{ (a)}, 0 \text{ (empty)}\}$$

This sequence contains all lengths of proper suffixes of $P[1,\ldots,7]$ that are also prefixes of P.

Lemma 2 (Prefix Function Property)**.**

For any position q in the pattern, if there exists a non-empty proper suffix that matches a prefix (i.e., $\pi[q] > 0$), then the length of that suffix minus 1 appears in the prefix function iteration of the previous position:

$$\forall q \in [1,m], if\ \pi[q] > 0 \quad \Rightarrow \quad \pi[q] - 1 \in \pi^*[q-1]$$

Proof. Let's prove this by following these steps:

1. Since $\pi[q] > 0$, we know that $P[1,\ldots,\pi[q]]$ is a proper suffix of $P[1,\ldots,q]$
2. By definition of proper suffix, $\pi[q] < q$
3. This implies $\pi[q] - 1 < q - 1$
4. Consider the substring $P[1,\ldots,\pi[q]-1]$:
   - It is a prefix of $P[1,\ldots,\pi[q]]$ (which matches a suffix of $P[1,\ldots,q]$)
   - Therefore, it must be a proper suffix of $P[1,\ldots,q-1]$
5. By the definition of prefix function iteration, $\pi[q]-1$ must be in $\pi^*[q-1]$

q.e.d.

The Knuth-Morris-Pratt Algorithm

Let's define the set E_{q-1} that helps us compute the prefix function values:
$E_{q-1} = \{k \in \pi^*[q-1] | P[k+1] = P[q]\} = \{k | k < q-1 \wedge P[1,...,k+1] \text{ is a suffix of } P[1,...,q]\}$

Intuitively, E_{q-1} contains all values k from the prefix function iteration $\pi^*[q-1]$ where:

- The next character after position k matches the current character at position q
- The substring $P[1,...,k+1]$ forms a proper suffix of $P[1,...,q]$

Using this set, we can formally define the prefix function value at position q as:

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset \quad \text{(no matching suffixes)} \\ 1 + \max(k \in E_{q-1}) & \text{otherwise} \quad \text{(take longest matching suffix)} \end{cases}$$

With these mathematical foundations, we can now define the Knuth-Morris-Pratt algorithm:

Algorithm 31 KMP(T, P)

Input: a text T of length n and a pattern P of length m
Output: all the occurrences (or valid shifts) of P in T

```
1:  π ← Compute_π(P)
2:  q ← 0                                          ▷ Number of characters matched so far
3:  solution ← ∅
4:  for i = 1 to |T| do do
5:      while q > 0 and P[q+1] ≠ T[i] do            ▷ Next character does not match
6:          q ← π[q]
7:      end while
8:      if P[q+1] = T[i] then                       ▷ Next character matches
9:          q ← q+1
10:     end if
11:     if q = |P| then                             ▷ Found complete pattern match
12:         solution.append(i − |P| + 1)
13:         q ← π[q]                                ▷ Look for next potential match
14:     end if
15: end for
16: return solution
```

The time complexity of the Knuth-Morris-Pratt algorithm is $\Theta(n+m)$, where n is the length of the text T and m is the length of the pattern P. This result follows from aggregate analysis of the algorithm's behavior. The key insight is that each character in the text is examined at most twice: once when extending a match and once when falling back through the prefix function values.

Bibliography

[1] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.