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

June 2, 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 $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, $\omega(\cdot)$.

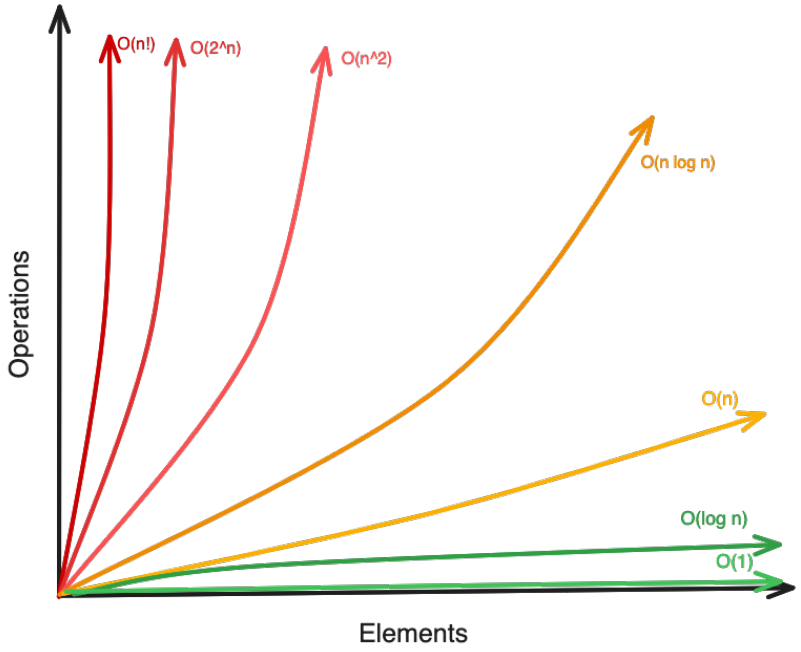


Figure 1.1: Complexity

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.

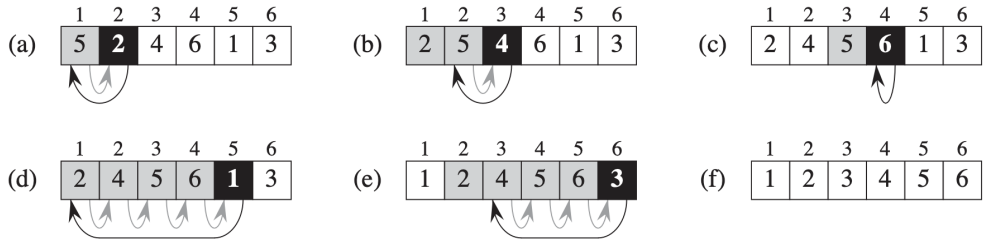


Figure 2.1: Insertion Sort [1]

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 $A[p \ldots r]$.

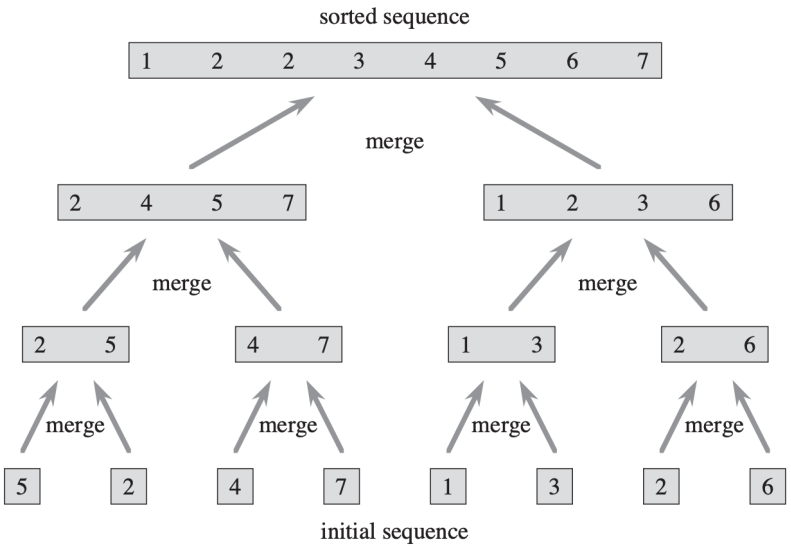


Figure 2.2: Merge Sort [1]

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.

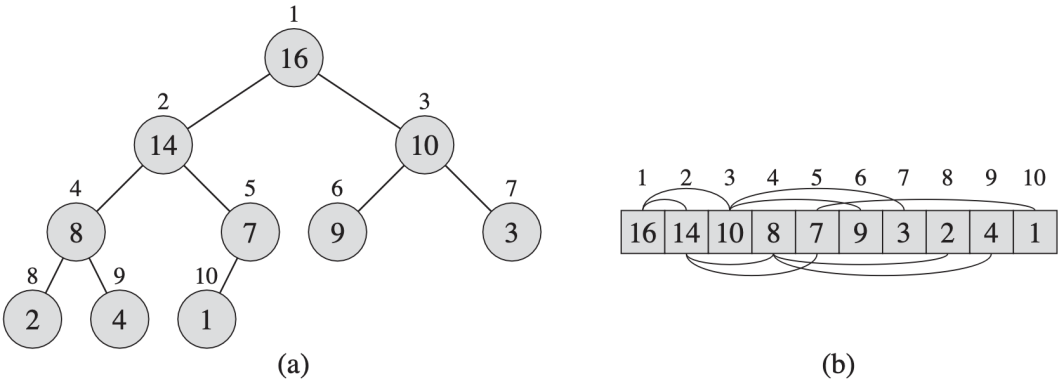


Figure 2.3: A max-heap viewed as a Binary Tree (a) and an array (b) [1]

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.

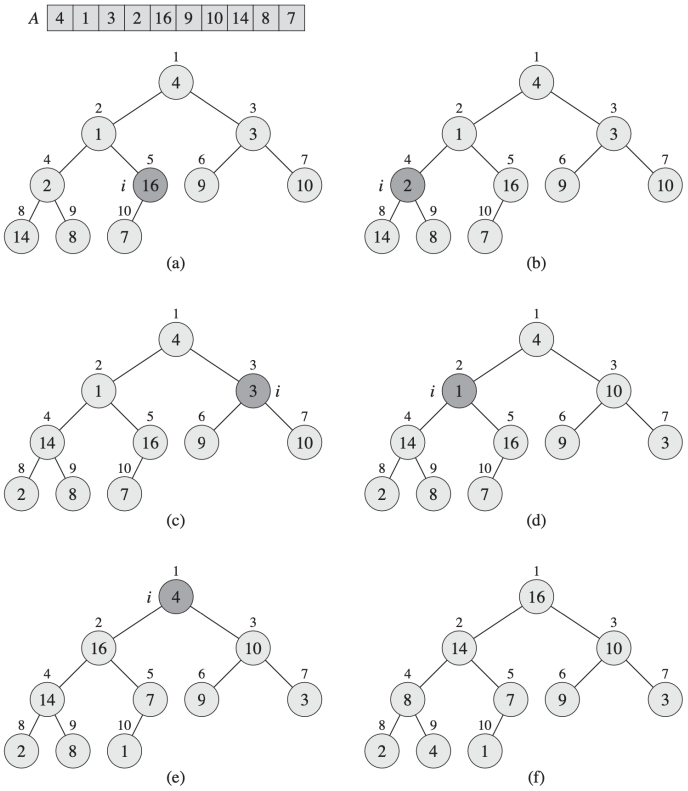


Figure 2.4: BUILD-MAX-HEAP [1]

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.

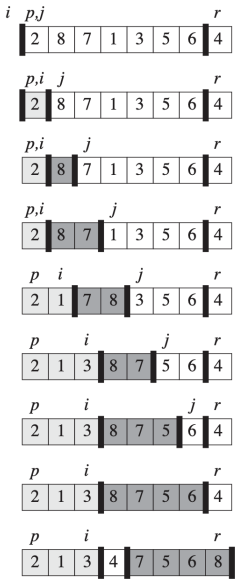


Figure 2.5: Quicksort [1]

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: $\theta(n^2)$

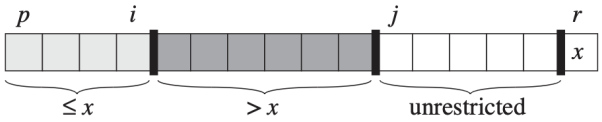


Figure 2.6: Regions maintained by PARTITION [1]

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 $O(n \log n)$.

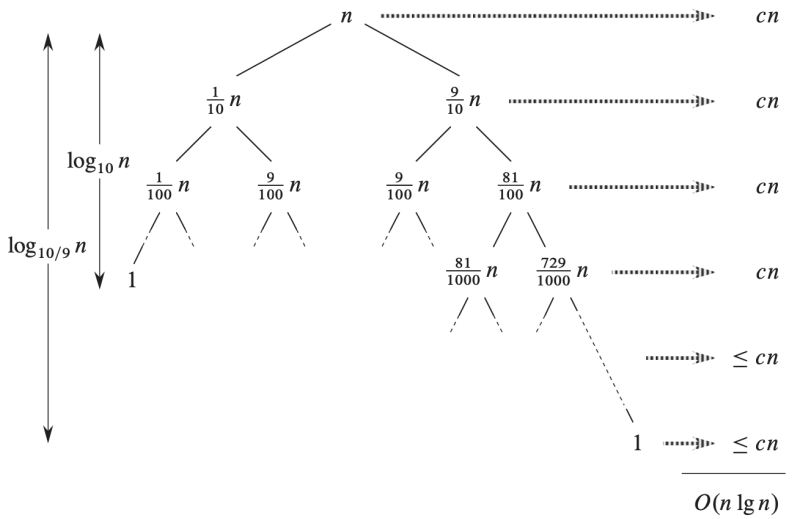


Figure 2.7: PARTITION always of 9 to 1 [1]

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.

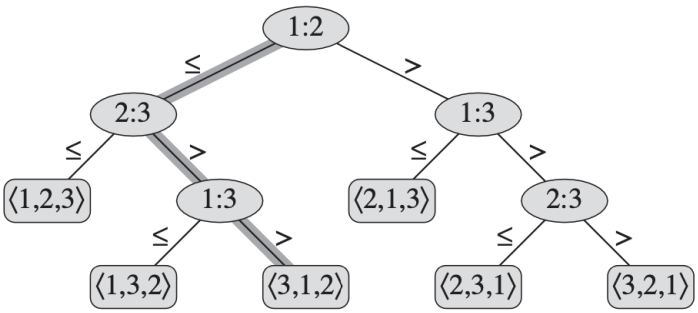


Figure 2.8: Decision Tree Model [1]

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.

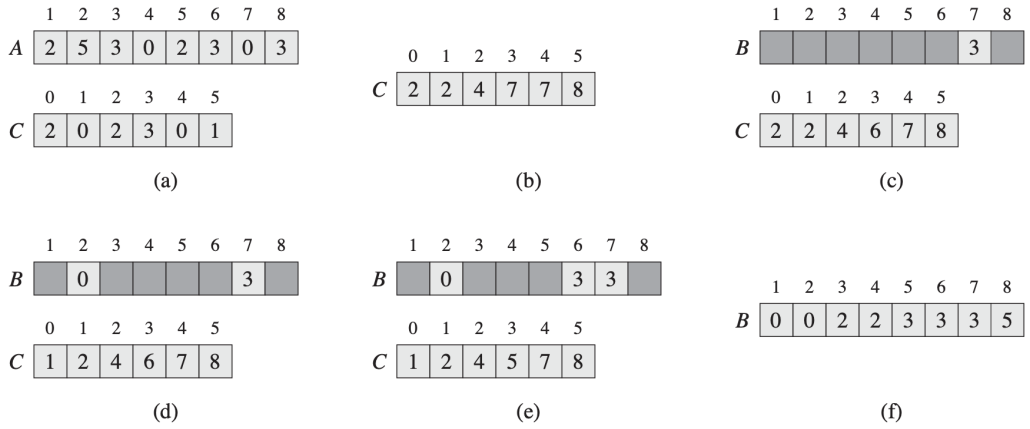


Figure 2.9: Counting Sort [1]

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**.

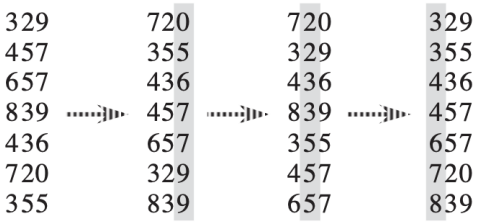


Figure 2.10: Radix Sort [1]

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

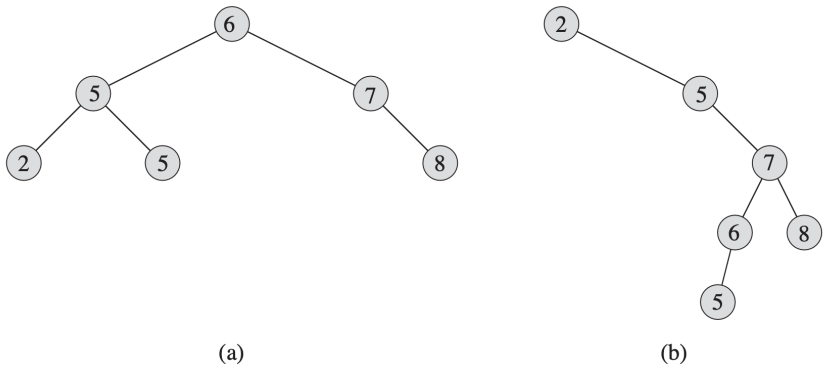


Figure 3.1: Binary Search Tree

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
```

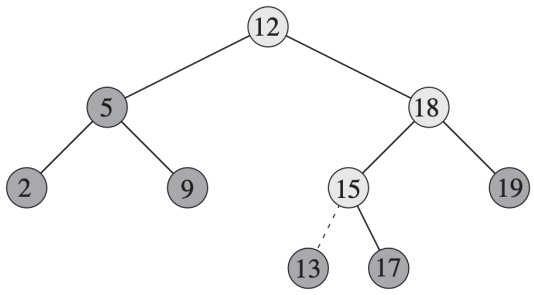


Figure 3.2: Insertion in a Binary Search Tree

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
```

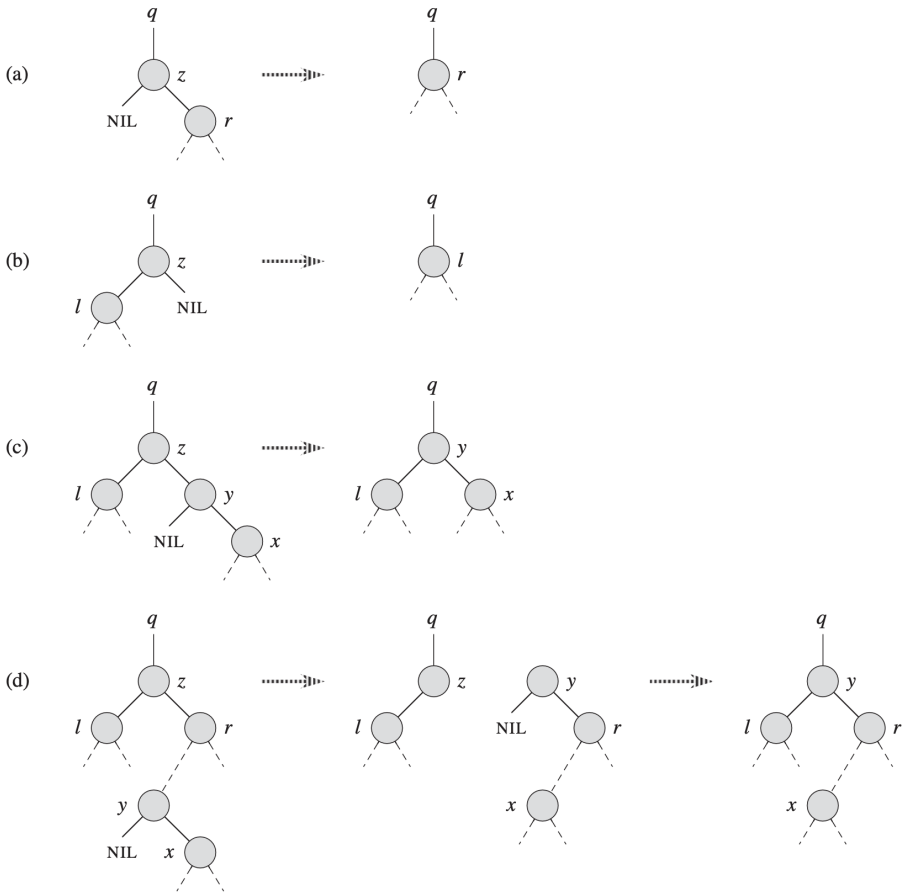


Figure 3.3: Deletion in a Binary Search Tree.
(a) Node z has no left child.
(b) Node z has a left child but not a right child.
(c) Node z has two children.
(d) Node z has two children, and its successor $y \neq r$ lies between the subtree rooted at r.

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.

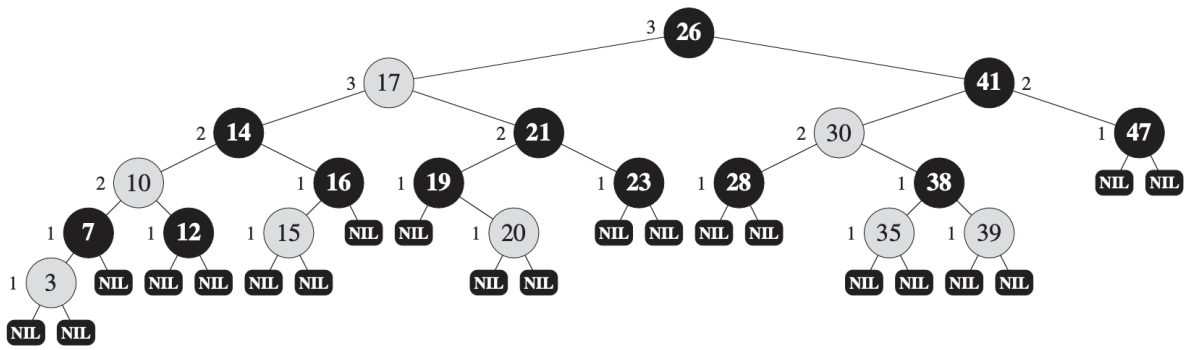


Figure 3.4: Red-Black Tree

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
```

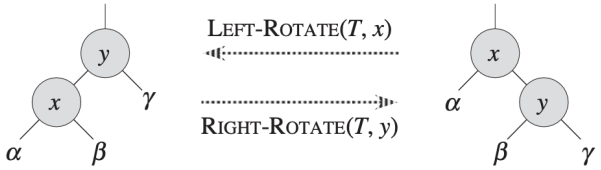


Figure 3.5: Left Rotation

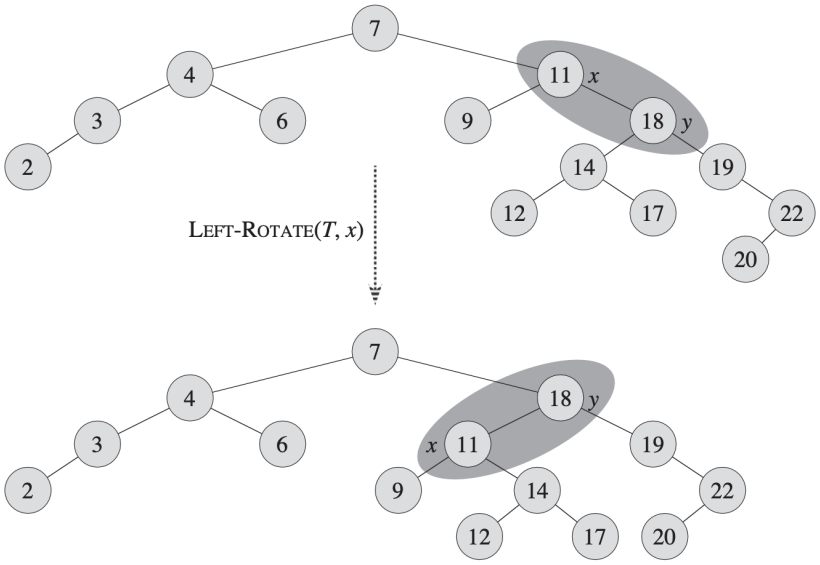


Figure 3.6: Example of rotation in a Red-Black Tree

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
```

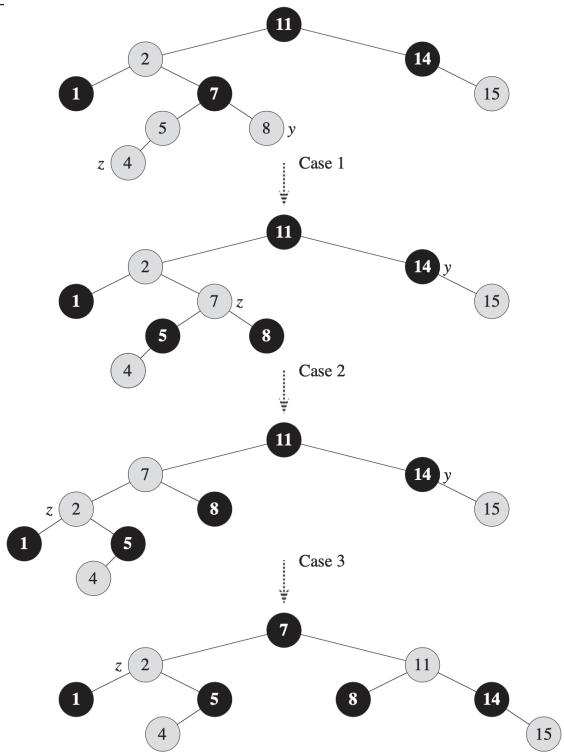


Figure 3.7: Insertion in a Red-Black Tree

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.

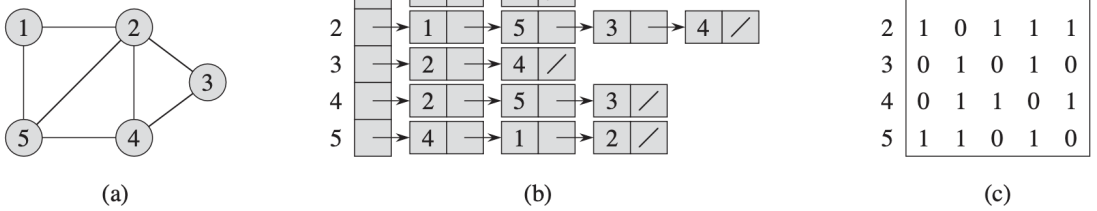


Figure 4.1: Undirected raph

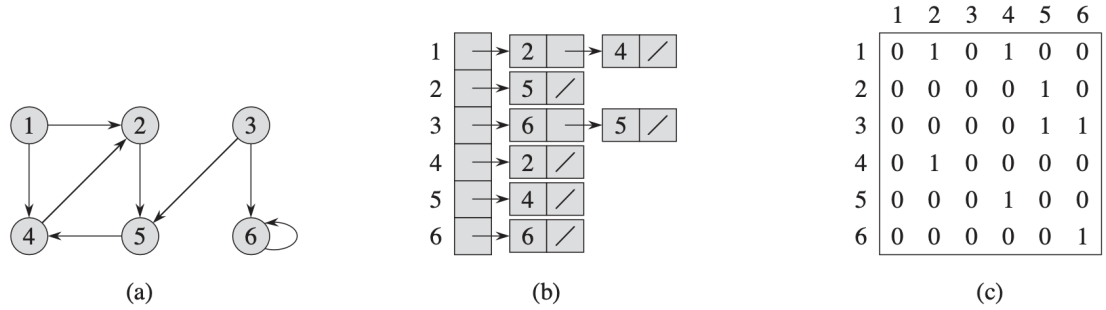


Figure 4.2: Directed graph

- **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.

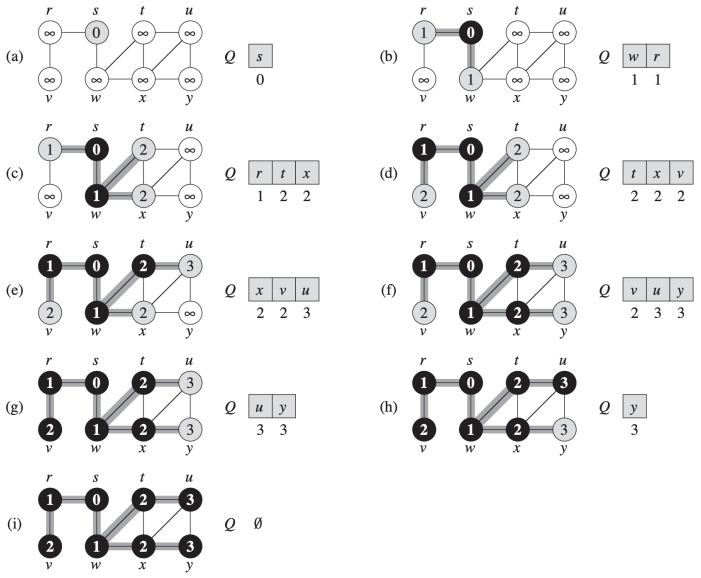


Figure 4.3: Breadth-First Search

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.

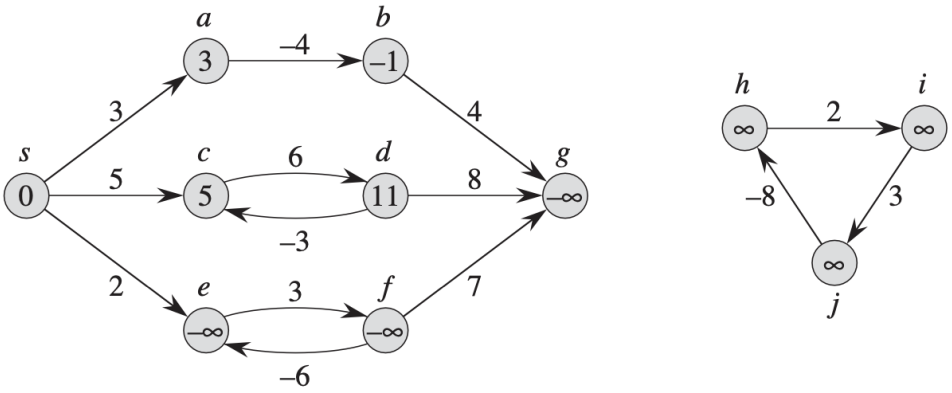


Figure 4.4: Negative cycle

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.

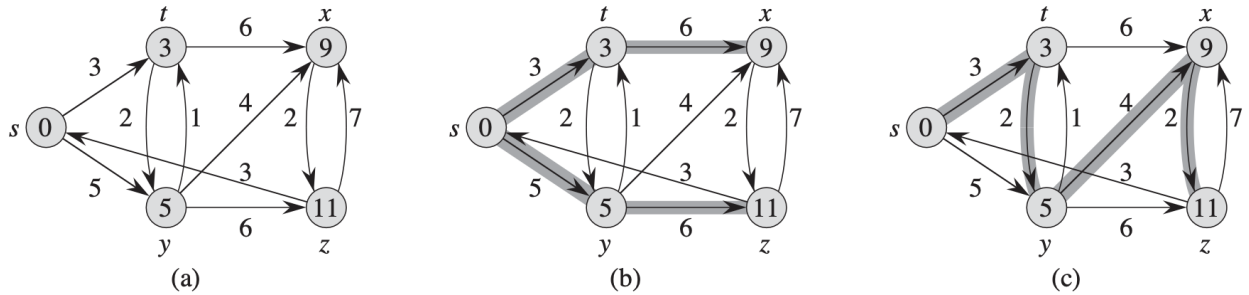


Figure 4.5: Shortest-path representation

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
```

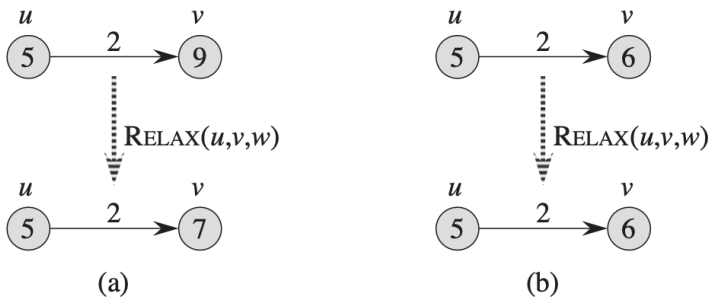


Figure 4.6: Relaxation

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).

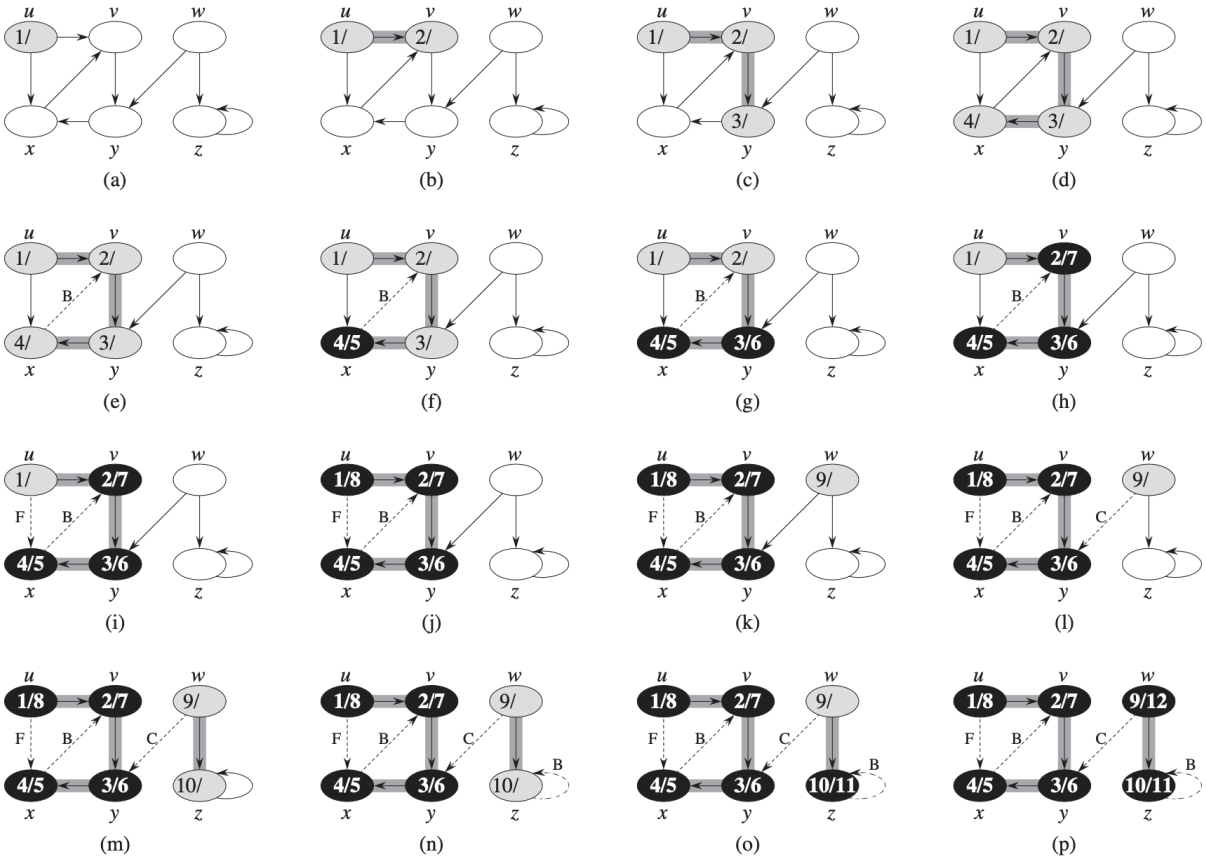


Figure 4.7: Depth-First Search

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 $u.d=\delta(s,u)$. □

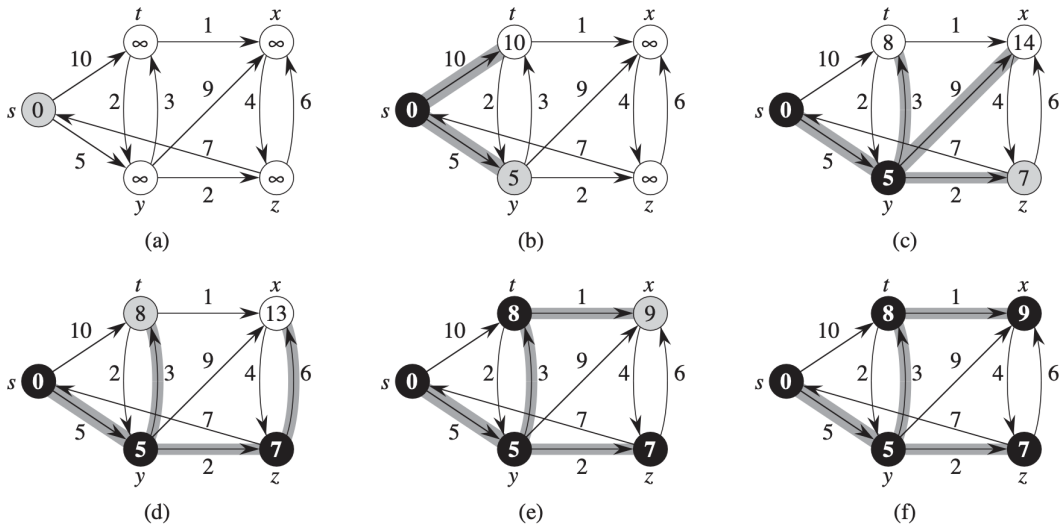


Figure 4.8: Dijkstra's Algorithm

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 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.).

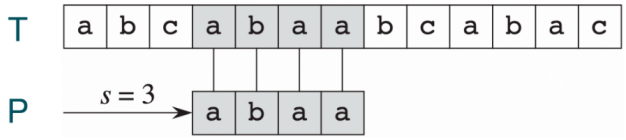
We say that pattern P occurs with shift s (equivalently, occurs at position $s+1$) in text T if:

- **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 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 $O(n \cdot m)$.

5.2 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                                   ▷ length of the current lowest prefix
4:  for q ← 2 to m do                       ▷ length of the substring we are currently checking
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:

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

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 n 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 = m then                               ▷ Found complete pattern match
12:         solution.append(i−m+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.

6 Multiple Pattern Matching

6.1 Multiple Pattern Matching Problem

Consider a scenario where instead of searching for a single pattern, we need to find multiple patterns simultaneously within a text.

Given a text T of length n and a collection of k patterns $P_1, \ldots, P_k$ with respective lengths $m_1, \ldots, m_k$, we want to find all occurrences of any pattern within the text.

A naive approach would be to apply the KMP algorithm k times, once for each pattern. This would yield a computational complexity of $O(\sum_{i=1}^{k} m_i + k \times n)$, where the first term accounts for preprocessing all patterns and the second term represents k separate scans of the text. Our goal is to reduce this complexity to $O(\sum_{i=1}^{k} m_i + n)$, eliminating the factor of k in the text scanning phase. This improvement can be achieved by preprocessing the text T instead of the individual patterns P_i, allowing us to search for all patterns simultaneously in a single pass.

Trie Data Structure

To efficiently handle multiple patterns, we introduce the concept of a **trie** (prefix tree). A trie is a specialized tree data structure that serves as an efficient dictionary for string storage and retrieval.

Formally, a trie is a rooted tree with the following properties:

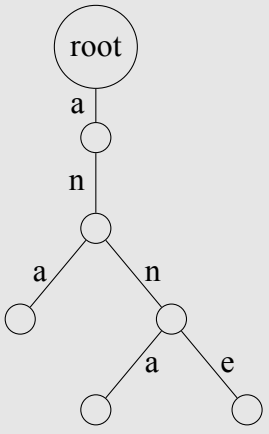
- **Edge labeling:** Each edge is labeled with a letter from alphabet Σ
- **Unique branching:** No two edges branching from the same node are labeled with the same letter
- **String representation:** Each path from root to leaf represents a string in our collection

Tries can compactly represent sets of strings and function as a type of dictionary data structure. For a collection of strings $T_1, \ldots, T_k$, the maximum number of nodes in the corresponding trie is $\Theta(\sum_{i=1}^{k} |T_i|)$, which equals the total number of characters across all strings.

Example: *Trie*

Let's define our set of strings as:

$$R = \{\text{ana}, \text{ann}, \text{anna}, \text{anne}\}$$



The trie structure shown on the right represents $Trie(R)$.
It demonstrates how all strings in R begin with the common prefix `an` before diverging into their distinct suffixes. The structure maps each string in R to a unique path from the root to a leaf.

The trie's efficiency comes from its ability to store shared prefixes only once, eliminating redundancy. This compression allows us to search for all patterns simultaneously by following the text's characters through the tree structure.

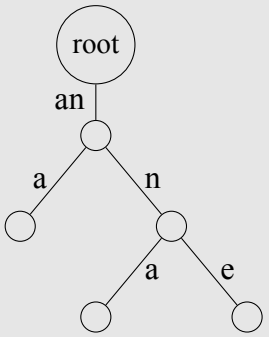
A **compacted trie** is a rooted tree whose edges are labeled by strings over alphabet Σ such that no two edges branching from the same node are labeled starting by the same letter. All nodes but the leaves have at least two children. The maximum number of nodes a compacted trie can have is $2k$, so $\Theta(k)$, where k is the number of words in the set.

Example: *Compacted Tries*

Let's define our set of strings as

$$T = \{\text{ana}, \text{ann}, \text{anna}, \text{anne}\}$$

Then, its compacted trie would be the one in the right. It demonstrates how all strings in T share the common prefix `an` before diverging into their distinct suffixes.
Also in this case, he structure maps each string in T to a unique path from the root to a leaf.



6.2 Suffix trees

6.2.1 Suffix Tree

A **suffix tree** represents a string T as a compacted trie containing all its suffixes. To ensure that each suffix terminates at a leaf node, we append a special end marker \$ (where $\$ \notin \Sigma$) to T. The suffix tree is then constructed from the extended string $T\$$. This modification ensures that there are no suffixes that are prefixes of other suffixes, which could otherwise appear in an internal node.

Example: *Suffix Trees*

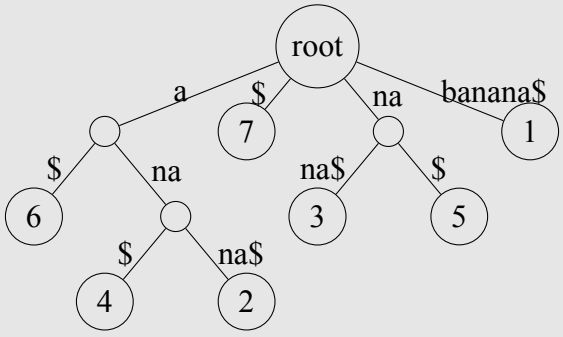
Let's define T as:

T	b	a	n	a	n	a	\$
Pos	1	2	3	4	5	6	7

where $\$ \notin \Sigma$. Then, we can define the list of suffixes of T as:

Suffixes						
b	a	n	a	n	a	\$
	a	n	a	n	a	\$
		n	a	n	a	\$
			a	n	a	\$
				n	a	\$
					a	\$
						\$

Then, its suffix tree would be:



where the labels of the leaves correspond to starting positions of suffixes in T.

Observation:

The number in each leaf represents the starting position of the suffix in the string.

To search for a pattern P in the suffix tree ST of T:

- Follow the path in ST that spells out the letters of P.
  - If you can spell the entire P, then all occurrences of P in T correspond to the leaf IDs below.
  - If you cannot spell all of P, then P does not occur in T.

For a string T of length n, the compacted suffix tree exhibits remarkable space efficiency with $O(n)$ nodes and $O(n)$ edges. While a naive representation would require $O(n^2)$ space due to potentially storing full substrings as edge labels.

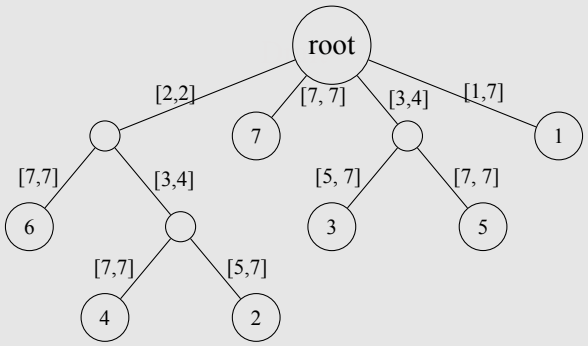
Recalling that every substring of T is a prefix of some suffix of T, meaning each substring maps to a unique path from the root in the suffix tree. This elegant property allows us to represent each edge label as a simple interval of positions over T, requiring only $O(1)$ space per edge. Since the tree contains $O(n)$ edges, this compact representation reduces the total space complexity to $O(n)$, making suffix trees both powerful and practical for large-scale string processing.

Example:

Let's define T as

T	b	a	n	a	n	a	$
Pos	1	2	3	4	5	6	7

Then, its new suffix tree would be



Note that each interval represents any occurrence in the string of its corresponding substring.

Suffix Tree Construction

The steps to build the suffix tree are represented in the following algorithm.

Algorithm 32 Naïve_construct_ST(T)

Input: A string T of length n
Output: The suffix tree ST of T

1: $ST \leftarrow \emptyset$
2: **for** $i = 1$ to $i = n+1$ **do**
3: Spell $T[i,\ldots,n+1]$ from the root of ST until arriving at a node u
4: Branch from u and add an edge with the remaining part of $T[i,...,n+1]$
5: **end for**
6: **return** ST

The worst case is when the text is a single character repeated n times, then the complexity of this algorithm is $O(n^2)$.

Suffix Tree Representations

There are different ways of representing a suffix tree, each one with its advantages and drawbacks.

Children	Time (pattern matching)	Space (ST)
List	$O(\sigma \times m + \|occurrences\|)$	$O(n)$
Array of size σ	$O(m + \|occurrences\|)$	$O(\sigma \times n)$
Balanced Binary Search Tree	$O(\log(\sigma) \times m + \|occurences\|)$	$O(n)$
Hash table	$O(m + \|occurrences\|)$	$O(n)$

Table 6.1: Used data structures to build suffix trees. $|T| = n$, $|P| = m$ and $|\Sigma| = \sigma$.

Even though hash tables may have the best combination of time and space complexity, their nature causes that children in each node are mixed and not ordered as in the other data structures.

- For the **list** data structure, we should store three elements per edge (next char, interval, pointer to node). For example, for an edge e_1 which points node H with label *banana*, its information would be $(b, [1,7], H)$.
- For the **array of size σ** data structure, we should store σ elements per node, being each element a tuple containing a pointer and an interval. For example, for a node A connected to node B with label *na* and to node C with label \$, then its information would be $[a, b, n, \$] = [Nil, Nil, (B, [3,4]), (C, [7,7])]$.

6.2.2 Suffix Array

$SA(T)$ is an array of integers of length $|T|+1$ where $SA[i] = j$ if and only if $T[j, \ldots, |T|+1]$ is the i-th lexicographically smallest suffix of T. In other words, it stores the starting positions of all suffixes of T sorted in lexicographical order.

Example: *Suffix Array*

Let's define T as:

T	b	a	n	a	n	a	\$
Pos	1	2	3	4	5	6	7

where $\$ < a < b < n$.
Then, we can define the list of suffixes of T as

Suffixes

b a n a n a \$
a n a n a \$
n a n a \$
a n a \$
n a \$
a \$
\$

The suffix array $SA(T)$ in this case would have the following form:

[7 6 4 2 1 5 3]
\$ a\$ ana\$ anana\$ banana\$ na\$ nana\$

where every integer is the starter position of a suffix.

In order to check if a pattern is in the text, We can apply binary search to look for it in the suffix array. If the pattern is not found, then it is not contained in the text. If a match is found, we need to check the neighbors (right and left elements) to search for more occurrences.

The time complexity is $\Theta(m \times \log n + m \times |occurrences|)$ where m is the number of character comparisons at each step, $\log n$ is the number of binary search steps and $m \times |occurrences|$ the steps used to find all occurrences.

6.2.3 Longest Repeating Factor Problem

The longest repeating factor of a text T is the longest substring that occurs at leas twice in T. It is represented by the deepest branching node in the suffix tree.

We explore the suffix tree using a DFS (depth-first-search) algorithm and storing in each node the length of the concatenations of strings read to reach that node.

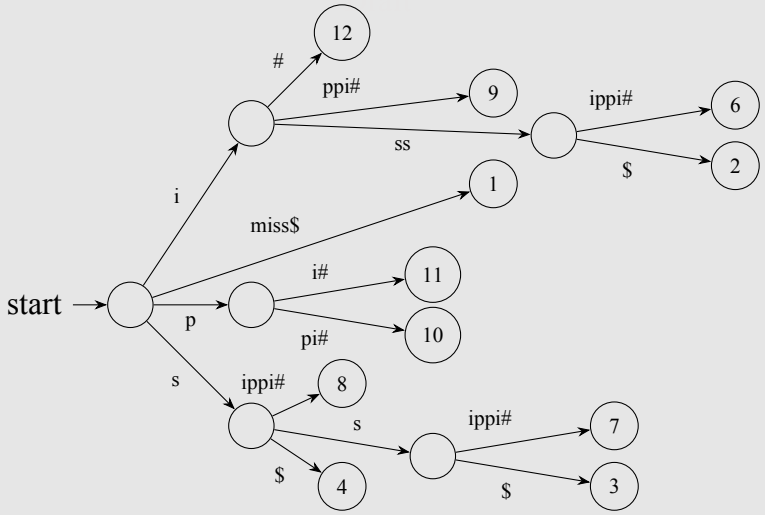
6.3 Generalized Suffix Tree

The generalized suffix tree of a set of strings $S_1, S_2, ..., S_k$ is the compacted trie of all suffixes of all strings in the set.

To build it, it suffices to built the suffix tree of their concatenation $S_1\$_1S_2\$_2 \ldots S_k\$_k$, where $\$_1 \ldots \$_k$ are distinct terminal symbols.

Example:

Given two strings $S_1 = miss\$$ and $S_2 = issippi\#$, then the generalized suffix tree is



6.3.1 Longest Common Substring Problem

The longest Common Substring (LCS) of two strings S and T is the longest substring that occurs both in S and T. It is represented by the deepest branching node in the suffix tree that have at least a descending leaf corresponding to S and at least a descending leaf corresponding to T.

When exploring the tree, you must check if the node has two branches ending in different terminal symbols in order to consider it a candidate. Explore the tree using a DFS and pushing the terminal symbols to their parent nodes when going back bottom-up. You can stop propagating the info once you reach the deepest node in the branch with both symbols.

7 Hash Tables

7.1 Static Hashing with Chaining

The model is based on the idea of having a universe of items, each with a distinct key.

We define a hash function $h : \mathscr{U} \to [0, m-1]$ which will be used to map the items to our table. However, there might be collisions: $\exists k_i \neq k_j$ s.t. $h(k_i) = h(k_j)$.

We need to make some assumptions in order to make it work:

- Each key of $\mathscr{U}$ is equally likely to be hashed to any slot of the hash table (**uniformity**). $Pr(h(k) = i) = \frac{1}{m} \forall k \in \mathscr{U}, \forall i \in [0, m-1]$
- Each key is mapped to a slot independently from the items already in the table (**independency**).
- $h(k)$ can be computed in $\Theta(1)$ time.

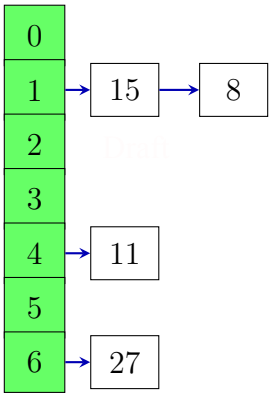


Figure 7.1: Hash table with chaining containing 7 buckets (0-6). The keys 15, 11, 27, and 8 are inserted in sequence.

Suppose we have stored n items into a table of size m. Then, the expected size of any list is $\frac{n}{m}$ (load factor α).

	Expected Time	Worst Case Time
Insert	$\Theta(1)$	$\Theta(1)$
Search	$\Theta(1+\alpha)$	$\Theta(n)$
Delete	$\Theta(1+\alpha)$	$\Theta(n)$

Table 7.1: Operation complexity in a static hash table.

The expected time to **search** for a key is $\Theta(1+\frac{n}{m}) = \Theta(1+\alpha)$, that is equal to $\Theta(1)$ if $n = O(m)$ ($\frac{n}{m} = O(1)$). Worst case is $\Theta(n)$ (all elements go to the same slot). The **insert** time is $\Theta(1)$ worst case because it is only required to compute the hash function.

The **delete** expected time is $\Theta(1+\frac{n}{m})$ (search + deletion). $\Theta(n)$ worst case.

7.2 Dynamic Hashing with Chaining

Dynamic hashing extends the static approach by allowing the hash table to grow and shrink dynamically based on the number of elements. This maintains optimal space utilization and performance.

The key invariant is maintaining $m \geq n$, or more precisely, $\frac{n}{m} = \Theta(1)$; when inserting the $(m+1)$-th element into a table of size m, we create a new table of size $m' > m$. This process involves three steps:

1. Allocate a new hash table of size m' in $\Theta(m')$ time
2. Generate a new hash function $h' : \mathscr{U} \to [0, m'-1]$ in $\Theta(m')$ time
3. Rehash all existing items into the new table using h' in $\Theta(m+n)$ time

For **growing** the table, we use a doubling strategy: $m' = 2m$. This choice of growth factor ensures optimal amortized performance.

Consider inserting n items sequentially into an initially empty table. The total cost is $\Theta(1+2+4+8+\cdots+n) = \Theta(\sum_{i=0}^{\log n} 2^i) = \Theta(n)$, yielding an amortized cost of $\Theta(1)$ per operation.

For **shrinking**, we follow a similar process:

1. Allocate a new table of size m'
2. Generate a new hash function $h' : \mathscr{U} \to [0, m'-1]$
3. Rehash all n items into the new table

We trigger shrinking when $n = \frac{m}{4}$, setting $m' = \frac{m}{2}$. This threshold of $\frac{1}{4}$ (rather than $\frac{1}{2}$) prevents thrashing between grow and shrink operations near the boundary.

The total cost for growing or shrinking is $\Theta(m'+m+n)$.

A crucial observation is that deletions can only occur for previously inserted items. Therefore, the total deletion cost is bounded by the total insertion cost of $\Theta(n)$, leading to an amortized cost of $\Theta(1)$ per operation.

	Expected Time	**Worst Case Time**
Insert	$\Theta(1)$	$\Theta(m'+m+n)$
Search	$\Theta(1+\alpha)$	$\Theta(n)$
Delete	$\Theta(1+\alpha)$	$\Theta(m'+m+n)$

Table 7.2: Operation complexity in a dynamic hash table.

8 Stream Model

8.1 Bloom Filters

A **Bloom filter** is a space-efficient probabilistic data structure designed to test whether an element is a member of a set. It is especially useful when the ***data is so large that it cannot fit in main memory***, motivating the use of a stream model. Bloom filters allow for false positives (reporting an element is present when it is not), but never false negatives.

Problem: Given a stream of keys, maintain a data structure to answer membership queries:

"Given any key, did k appear in the stream?"

8.1.1 Types of Bloom Filters

- ***1-binary array and 1-hash function.***
  We have an array B of m bits and a hash function $h : \mathscr{U} \to [0, m-1]$. For each key k in the stream, set $B[h(k)] \leftarrow 1$. If the cell is already 1, do nothing. Collisions can cause errors: we may answer *yes* to queries for which the real answer is *no* (false positives), but never the opposite. The probability of a false positive is roughly $\frac{n}{m}$.
- ***i-binary array and k-hash functions.***
  In order to decrease the probability of error, we could store i binary arrays of size m, such that for inserting an item i, we compute all its hash functions $B[h_i(k)] \leftarrow 1$. This way we can still have collisions but with a smaller probability. The probability of a false positive is:

$$\Pr(\text{FP}) \leq P_1^i$$

  where P_1 denotes, among all the binary arrays used, the highest fraction of cells set to 1 ($\frac{\#1s}{m}$). The probability of a false positive decreases exponentially as i increases.

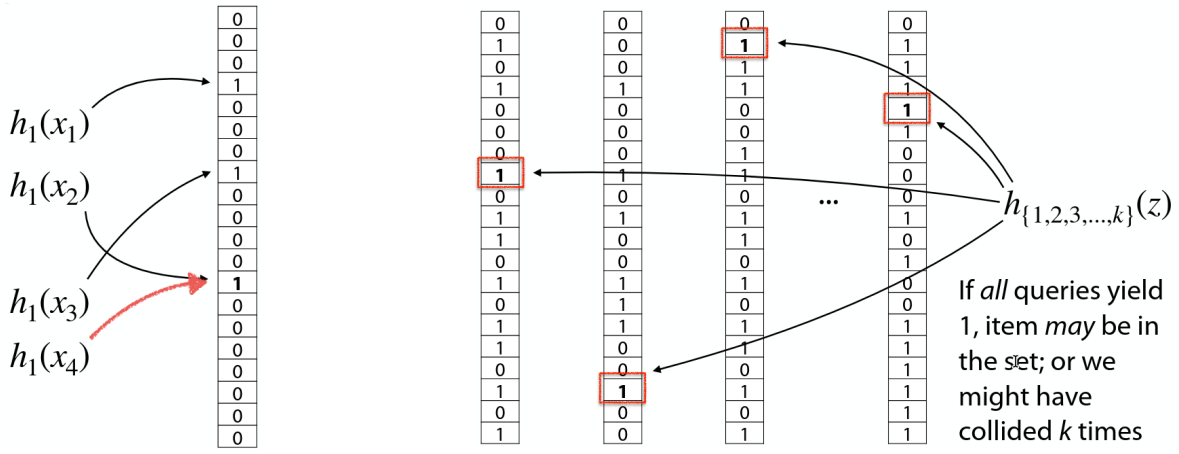


Figure 8.1: Bloom filters. 1-array and 1-hash functions (left) and k-array and k-hash functions (right).

- ***1-binary array and k-hash functions.***
  Use one binary array of size m and k independent hash functions. To insert k, set $B[h_j(k)] \leftarrow 1$ for all $j = 1, \ldots, k$. The probability that a bit is 0 or 1 after n insertions is:
  – With 1 hash function and $n = 1$, it is $P_0 = \left(1 - \frac{1}{m}\right)$, then $P_1 = 1 - P_0$.
  – With k hash functions and n insertions, it is $P_0 = (1 - \frac{1}{m})^{nk}$.
  Then, the probability of getting a false positive is:

$$\Pr(\text{FP}) = P_1^k = (1 - P_0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k = f \quad \Rightarrow \quad \Pr(\text{FP}) \sim \left(1 - e^{-\frac{nk}{m}}\right)^k = \tilde{f}$$

  If $\frac{n}{m} = c$ is constant, the optimal number of hash functions is:

$$k^* = \ln(2)\frac{m}{n}$$

  For example, if $\frac{m}{n} = 10$, $k^* \approx 7$.
  The probability of reading a 0 in the Bloom filter is $\tilde{P}_0 = e^{-\frac{nk}{m}}$. With the optimal $k^*k = \ln(2)\frac{m}{n}$:

$$\tilde{P}_0 = \frac{1}{2}$$

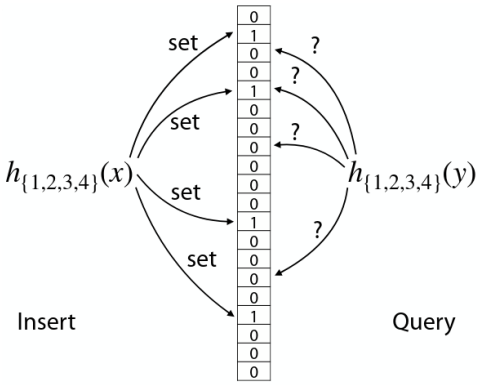


Figure 8.2: Bloom filter with 1 array and k hash functions.

How many items can we insert before the Bloom filter is 50% full?
A natural question is to estimate the number of insertions required before the Bloom filter's bit array becomes substantially full, for instance, 50%. Consider the process of setting bits in the m-bit array, and $k = 1$ hash function:
– **First insertion:** The probability that an attempt sets a new bit is $P_0 = \frac{m-0}{m} = 1$. The expected number of attempts is $1/P_0 = 1$.
– **Second insertion:** The probability that an attempt sets a new bit (one of the $m-1$ remaining zero bits) is $P_1 = \frac{m-1}{m}$. The expected number of additional attempts is $1/P_1 = \frac{m}{m-1}$.
– **General case:** The probability that an attempt sets a new bit is $P_j = \frac{m-j}{m}$. The expected number of additional attempts to achieve this is $1/P_j = \frac{m}{m-j}$.

Warning: *Notation P_j*

Note that we're using P_j here to represent the probability of setting a new bit during the j-th insertion. This is different from our earlier use of P_0, which represented the probability of finding a 0 in the Bloom filter after n insertions.

Let X be the random variable representing the total number of such bit-setting attempts required to set exactly $m/2$ distinct bits in the Bloom filter to '1'. The expected value of X, $E[X]$, is the sum of the expected number of attempts needed to set each successive new bit, from the 1st bit up to the $(m/2)$-th bit:

$$E[X] = \sum_{j=0}^{m/2-1} \frac{m}{m-j}$$

This sum can be rewritten by letting $i' = m - j$.

$$\begin{cases} i' = m & \text{if } j = 0 \\ i' = m/2+1 & \text{if } j = m/2-1 \\ i' = m-j & \text{otherwise} \end{cases} \quad \Rightarrow \quad E[X] = m \sum_{i'=m/2+1}^{m} \frac{1}{i'}$$

Using the Taylor approximation, we get:

$$E[X] \approx m(\ln m - \ln(m/2)) = m \ln\left(\frac{m}{m/2}\right) = m \ln 2$$

For example, if $m = 100$, $E[X] \approx 100 \ln 2 \approx 69.3$. So, we expect to make around 69 "effective bit-setting attempts" before half the bits in the Bloom filter are set to 1.

Set operations

Given BF_1 for S_1 and BF_2 for S_2 (same size m, same k hash functions), we can compute:

- **Union:** $B = B_1$ OR B_2 represents $S_1 \cup S_2$.
- **Intersection:** $B = B_1$ AND B_2 represents $S_1 \cap S_2$. The error rate is higher than a Bloom filter built directly for $S_1 \cap S_2$. Some 1s in the new BF may be from elements that are not in the intersection but match in both BFs.

Characteristics and Trade-offs

Bloom filters have key characteristics that make them valuable. Their **one-sided error profile** ensures no false negatives, only false positives, making them ideal when missing a true positive is worse than occasional false ones.

Performance-wise, **insertion and query operations are** $O(k)$, independent of the number of items. Hash operations can be parallelized for fast execution. Bloom filters are also **space-efficient** compared to traditional data structures, as shown in Table 8.1, making them perfect for large datasets or memory-constrained environments.

	BF	**HT**
item	m bits	$m+n$ pointers $+n$ integers

Table 8.1: Simplified space comparison between Bloom filter and hash table.

However, these benefits come with certain **limitations**:

- **do not support item deletion**: attempting to delete an item by clearing its corresponding bits could inadvertently affect other items that happen to hash to one or more of the same bits, potentially leading to false negatives—a violation of their core guarantee.
- **cannot store associated values with keys**: they are purely membership testing structures. If you need to retrieve data associated with an item, a Bloom filter can only serve as a preliminary check to avoid costly lookups for non-existent items.

8.2 Count-Min Sketch

While Bloom filters answer membership queries, the **Count-Min Sketch (CMS)** addresses a different problem in stream processing: estimating the **frequency** of items.

Problem: Given a stream of items $a_1, \ldots, a_N$ (where N is the total count of items in the stream, including repetitions) drawn from a universe $\mathscr{U}$, we want to maintain a compact data structure that can estimate the frequency $f(x)$ of any queried item $x \in \mathscr{U}$. The frequency $f(x)$ is the number of times x has appeared in the stream processed so far.

A CMS consists of a $d \times w$ matrix of counters, denoted C. Initially, all $C[i][j]$ are set to 0. The structure uses d independent hash functions $h_1, \ldots, h_d$. Each hash function $h_j : \mathscr{U} \to [0, w-1]$ maps items to one of the w columns in its respective row j.

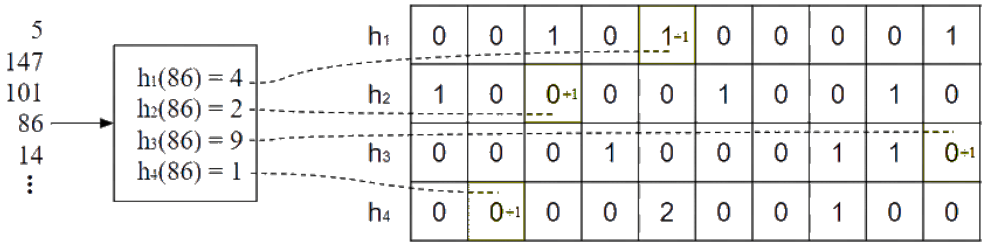


Figure 8.3: A $d \times w$ Count-Min Sketch. Here, $d = 4$ rows (hash functions) and $w = 10$ columns.

To **insert** an element a_i, we should apply all the hash functions (one per row) and increase by 1 the values of the corresponding cells.

To **query** an item x (estimate its frequency), denoted $\tilde{f}(x)$, we compute $h_j(x)$ for all $j = 1, \ldots, d$ and retrieve the values from the corresponding counters. The estimate is the minimum of these values:

$$\tilde{f}(x) = \min_{j \in [1,d]} \{C[j][h_j(x)]\}$$

Let $f(x)$ be the true frequency of x. A key property of CMS is that it **never underestimates** the frequency: $\tilde{f}(x) \geq f(x)$. Taking the minimum across different hash functions attempts to find the row with the least "collision noise."

Accuracy Guarantee (Claim):

The power of CMS lies in its probabilistic accuracy guarantee. If we choose the width $w = \lceil \frac{2}{\varepsilon} \rceil$ and depth $d = \lceil \log_2 \delta^{-1} \rceil$, then for any item $x \in \mathscr{U}$:

$$\Pr(\tilde{f}(x) \leq f(x) + \varepsilon n) \geq 1 - \delta$$

being n the number of items in the stream.

The parameters $\varepsilon > 0$ and $0 < \delta < 1$ control the trade-off between accuracy and space:

- ε determines the **error factor**: the overestimation is bounded by εN. Smaller ε requires larger w (more columns).
- δ determines the **failure probability**: the probability that the error bound is violated. Smaller δ requires larger d (more rows/hash functions).

In simpler terms, with probability at least $1 - \delta$, the estimated frequency $\tilde{f}(x)$ overestimates the true frequency $f(x)$ by at most εN.

Proof:

1. Fix item $x \in S$
2. For each row i, define d random variables $Z_1, \ldots, Z_d$ such that $Z_i = C_{i,h(x)} - f(x)$
3. Define also random variables $Y_{i,y} = \begin{cases} 1 & \text{if } h_i(x) = h_i(y) \\ 0 & \text{otherwise} \end{cases} \qquad \forall y \neq x, y \in S$
4. The error Z_i can be expressed as:
$$Z_i = \sum_{y \neq x} (Y_{i,y} f(y))$$
5. The expected value of Z_i is bounded by:
$$E(Z_i) = \sum_{x \neq y} E(Y_{i,y} f(y)) = \sum_{x \neq y} f(y) E(Y_{i,y}) = \sum_{x \neq y} f(y) \Pr(h_i(x) = h_i(y)) \leq \frac{n}{w}$$
6. Since $Z_i \geq 0$, applying Markov's inequality yields:
$$\Pr(Z_i \geq b\, E(Z)) \leq \frac{1}{b} \Rightarrow \Pr(Z_i \geq \frac{b\, n}{w}) \leq \frac{1}{b}$$
7. Let $b = w\, \varepsilon$
$$\Pr(Z_i \geq \varepsilon\, n) \leq \frac{1}{w\varepsilon} = \frac{1}{2}$$
That bounds the probability of making an error greater than some quantity in a given row.
8. The probability of overestimation is bounded by:
$$\Pr(Z_i + f(x) \geq f(x) + \varepsilon\, n) \leq \frac{1}{2}$$
9. The final probability bound is:
$$\Pr(\forall i \in [0, d-1], Z_i \geq \varepsilon n) \leq \left(\frac{1}{2}\right)^d = \left(\frac{1}{2}\right)^{\log_2 \delta^{-1}} = 2^{-\log_2 \delta^{-1}} = \delta$$
$$\Rightarrow \quad \Pr(\exists i \in [0, d-1] \text{ s.t. } Z_i < \varepsilon\, n) \geq 1 - \delta$$

CMS of dimensions $O(\varepsilon^{-1} \log \delta^{-1})$ achieves performance $\forall \varepsilon, \delta$.

ε	δ	$\lceil 2/\varepsilon \rceil \cdot \lceil \log_2 \delta^{-1} \rceil$
10%	0.1	80
1%	0.01	1,400
0.1%	0.001	20,000
0.0001%	0.01	1,400,000

Table 8.2: Number of counters required for various error (ε) and failure probability (δ) parameters.

Remember that ε multiplies m, and a counter requires many (maybe 32 or 64) bits.

Tip: ***CMS use case - heavy hitters***

A common and important use of the Count-Min Sketch (CMS) is to identify **heavy hitters** (items whose frequency exceeds a given threshold).

9 Cardinality Estimation

The problem of **cardinality estimation** (also known as **distinct count**) in a stream model is to determine the number of *distinct* elements that have appeared in a sequence of items.

Problem: Given a stream of items $S = a_1, a_2, \ldots, a_M$, where M is the total number of items seen, count N, the number of unique items in S.

Example: ***Cardinality of a set***

Let's consider the following set:

$$S = x\,y\,x\,y\,z\,x\,z\,y\,z\,x\,z\,z\,y\,x \qquad x \neq y \neq z$$

The cardinality of S is 3.

A common probabilistic approach relies on hashing items to a uniform distribution and observing properties of the resulting hash values.

Toy Problem (Hat Problem): Imagine a universe $\mathscr{U}$ of integers, say from 1 to 1000. Someone secretly selects a random subset of N distinct integers from $\mathscr{U}$ and hides them in a hat. Your task is to estimate N by observing only *one* representative value from the hidden set (e.g., its minimum, maximum, median).

If we choose to observe the **minimum** value, say m_{obs}, from the subset, we can intuitively reason about N. If the N numbers are uniformly and randomly scattered in the range $[1, 1000]$, they divide the range into approximately $N+1$ segments. The average length of such a segment would be $1000/(N+1)$. If m_{obs} is the minimum, it represents the length of the first segment (from 0 to m_{obs}). So, we can estimate:

$$m_{obs} \approx \frac{1000}{N+1} \implies N+1 \approx \frac{1000}{m_{obs}} \implies N \approx \frac{1000}{m_{obs}} - 1$$

For example, if the minimum observed is 95, then $N \approx \frac{1000}{95} - 1 \approx 10.52 - 1 \approx 9.5$.

To formalize this and make it general, we typically hash items to the real interval $[0, 1]$ using a good hash function $h : \mathscr{U} \to [0, 1]$ that distributes items uniformly. Let $x_1, x_2, \ldots, x_N$ be the hash values of the N distinct items seen in the stream. Let M_{min} be the minimum of these hash values: $M_{min} = \min\{h(a) \mid a \text{ is a distinct item in } S\}$. Our estimation equation becomes:

$$M_{min} \approx \frac{1}{N+1} \implies N \approx \frac{1}{M_{min}} - 1$$

Let $X_1, \ldots, X_N$ be N independent random variables drawn uniformly from $[0, 1]$ which model the hash values for the N items. Let $M = \min(X_1, \ldots, X_N)$. Then, we desire that:

$$E[M] = \frac{1}{N+1}$$

Let's define an indicator variable I_i that captures whether x_i is the minimum value:

$$I_i = \begin{cases} 1 & \text{if } x_i < \min_{j \neq i} x_j \\ 0 & \text{otherwise} \end{cases}$$

The expected value of each indicator is $\frac{1}{N+1}$ since each value has equal probability of being the minimum. This leads to:

$$E(I_i) = \frac{1}{N+1} \quad \forall i$$

Therefore, for a new value X_{N+1}:

$$E(I_{N+1}) = \frac{1}{N+1} = \Pr(X_{N+1} < \min_{1 \leq i \leq N} X_i) = E(M)$$

However, this approach has a drawback: if any value is very close to 0, the estimator can have high variance and potentially overestimate the true cardinality.

To improve robustness, we can use the k-th order statistic instead of just the minimum. Let M_k denote the k-th smallest value among the hash values. Then:

$$E(M_k) = \frac{k}{N+1} \implies \frac{M_k}{k} = \frac{1}{N+1}$$

Another effective strategy is to partition the $[0,1]$ interval into k equal subintervals and track the minimum value in each partition. For each new element:

1. Determine its partition in $\Theta(1)$ time
2. Compare with the current minimum of that partition in $\Theta(1)$ time
3. Update the minimum if necessary

The final cardinality estimate is obtained by:

- Computing separate estimates of N for each of the k partitions
- Taking the average of these estimates

Taking the average of the estimates helps reduce variance and improve accuracy.

10 Parallel Computation

10.1 Introduction and Notation

The design and efficacy of **parallel algorithms** are intrinsically linked to the underlying computational architecture. When devising a parallel program, one must consider not only the availability of multiple processing resources but also their ability to communicate and synchronize, if required.

Definition: ***Parallelize a Program***

To **parallelize a program** means to structure its operations such that they can be carried out simultaneously, or concurrently, on multiple processing units.

The extent to which a program can be parallelized, its **parallelizability**, is a characteristic determined by both the intrinsic nature of the problem being solved and the specific design of the program itself. A program that is inherently sequential, where each step depends on the completion of the preceding one, offers little to no opportunity for parallelization.

A program is deemed **sequential** if there exists a strict dependence between its execution steps, mandating a specific order. The process of **dependence analysis** is crucial for identifying these ordering constraints. Dependencies can be broadly categorized into two types:

- **Data dependency:** Occurs when an instruction refers to data that has been accessed or modified by a previous instruction.
- **Control dependency:** Occurs when the execution or non-execution of an instruction is contingent upon the outcome of a preceding instruction, often a conditional branch.

Parallelism can be exploited in different ways. **Data parallelism** involves distributing distinct data elements across multiple processing units, where each unit performs similar operations on its assigned data subset. Conversely, **instruction parallelism** (or task parallelism) focuses on executing different operations or instructions concurrently on multiple processing units.

Flynn's taxonomy provides a classical categorization of parallel computer architectures based on instruction and data streams:

- **SISD *(Single Instruction, Single Data):*** Traditional sequential von Neumann architecture.
- **SIMD *(Single Instruction, Multiple Data):*** A single instruction is executed synchronously by multiple processors operating on different data elements. Commonly used for data parallelism.
- **MISD *(Multiple Instruction, Single Data):*** Multiple instructions operate on a single data stream. This architecture is rare in practice.
- **MIMD *(Multiple Instruction, Multiple Data):*** Multiple processors execute different instruction streams on different data streams. This is the most general and flexible parallel architecture, encompassing multi-core processors and distributed systems.

For the algorithms discussed subsequently, we will often assume a **SIMD-like model** or a model where processing units can execute local programs, often synchronized at a high level.

Memory Architectures: Shared vs. Distributed

The organization of memory within a parallel computing system fundamentally shapes both the interaction patterns between processing units and the architectural decisions underlying parallel program design, with implications for performance, scalability, and programming complexity.

Shared Memory

In a **shared memory** architecture, all processing units have access to a common, global memory.

- Communication between processing units is implicitly achieved by reading from/writing to shared variables in this common memory.
- This model can simplify programming as data does not need to be explicitly sent between processing units.
- However, simultaneous access to shared data requires careful synchronization mechanisms (e.g., locks, semaphores) to prevent **race conditions** (where the outcome depends on the non-deterministic order of access) and ensure data consistency.

Distributed Memory

In a **distributed memory** architecture, each processing unit has its own private, local memory.

- A processing unit can only directly operate on data residing in its local memory.
- Communication and data sharing between processing units must be explicit, typically achieved through **message passing** over an interconnection network.
- While this model can be more complex to program due to the need for explicit communication, it often scales better to a very large number of processors as memory bandwidth is not a central bottleneck.

Key Architectural Characteristics for Algorithm Design

When designing a parallel algorithm, key characteristics of the architecture must be considered:

- ***Number of processing units (P):*** Influences the degree of parallelism achievable.
- ***Memory space organization:*** Whether memory is shared or distributed.
- ***IPC mechanisms:*** How PUs exchange data (e.g., shared variables, message passing).
- ***Control mechanism:*** Whether control is centralized (e.g., a global controller issuing instructions in SIMD) or distributed (each PU executes its own program, as in MIMD). This relates to instruction parallelism.
- ***Interconnection network topology:*** The physical organization and connections between processors (e.g., linear array, ring, mesh, hypercube). This dictates communication patterns, latency, and bandwidth.

Tip: ***Standard Notation***

Throughout our discussion of parallel algorithms, we will use the following notation:

- P: The number of processing units available.
- N: The size of the input problem (e.g., # elements in a vector, # nodes in a graph).
- M: The size of the local memory available to each processing unit. Unless otherwise specified, we will often assume $M = O(1)$, meaning each processor has a constant amount of local storage.

Input Distribution

The distribution of input data across processing units represents a fundamental design decision in parallel algorithms. This choice is influenced by several critical factors:

- ***Problem characteristics:*** The inherent nature of the problem and its computational demands, including data dependencies and access patterns
- ***Communication-computation tradeoff:*** The delicate balance between data replication and inter-processor communication costs, considering both bandwidth and latency constraints
- ***Implementation complexity:*** The need to maintain algorithm clarity, maintainability, and computational efficiency while managing synchronization overhead
- ***Load balancing:*** The distribution of computational work across processing units to maximize resource utilization

An alternative paradigm to static input distribution is **systolic computation**, characterized by streaming data through the system during execution. This approach is defined by a synchronized global clock that divides time into discrete **time steps**, with predictable data flow patterns through processor interconnections at each cycle. Each processing unit executes three fundamental operations per cycle, and consistent operations are performed synchronously across all units at each time step. This regular, rhythmic pattern of computation and communication enables efficient implementation of algorithms with regular data dependencies.

- Data reception from connected units, ensuring proper synchronization and data integrity.
- Local processing of received data, applying the algorithm's core computation
- Result transmission to subsequent units, maintaining the systolic flow pattern

Example: *Matrix Distribution*

For distributing an $N \times N$ matrix across P processing units, two common approaches are:

- **Block distribution:**
  The matrix is partitioned into contiguous blocks of size $\frac{N}{P} \times N$, with each processing unit receiving one complete block.
  This approach:
  – Minimizes communication for operations on local data
  – May lead to load imbalance if computation is not uniform
- **Cyclic distribution:**
  Matrix elements are assigned one at a time cyclically across all processing units.
  This approach:
  – Promotes better load balancing
  – May increase communication overhead for operations requiring neighboring elements

10.2 Performance Analysis

In parallel computing, we analyze algorithms through several key metrics. The space complexity is determined by the local memory size M required per processing unit, while the parallel execution time $T = T_P(N)$ measures how long it takes to process N elements using P processors. The total work W captures all computational effort, including both active processing and overhead costs like processor idle time and communication.

To understand the efficiency of data exchange, we examine the communication overhead ratio:

$$c = \frac{W^c}{W} \qquad \text{where } W^c = \text{total number of communications}$$

We assume that each physical connection can handle exactly one communication per time step. The speedup S tells us how much faster our parallel solution is compared to the sequential version:

$$S = \frac{\Gamma}{T} \leq \frac{T_1(N)}{T(N)} \leq \frac{W}{T} = P$$

where Γ represents the execution time of the fastest possible sequential algorithm. We can further measure how effectively we're using our processors through the parallel efficiency:

$$\varepsilon = \frac{S}{P} = \frac{\Gamma}{W} \leq 1$$

The goal is to design algorithms that achieve both minimal execution time and high efficiency - meaning we want T to be as small as possible while keeping ε close to 1.

Tip: ***Asymptotic notation***

Let's remark the asymptotic notation, for sufficiently large n:

- $f(n) = O(g(n))$ means that $f(n)$ is bounded above by a multiple of $g(n)$:
  $$f(n) = O(g(n)) \text{ if } \exists c \in R^+, n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0, f(n) \leq cg(n)$$
- $f(n) = \Omega(g(n))$ means that $f(n)$ is bounded below by a multiple of $g(n)$:
  $$f(n) = \Omega(g(n)) \text{ if } \exists c \in R^+, n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0, f(n) \geq cg(n)$$
- $f(n) = \Theta(g(n))$ means that $f(n)$ is bounded above and below by a multiple of $g(n)$:
  $$f(n) = \Theta(g(n)) \text{ if } \exists c_1, c_2 \in R^+, n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Networks of Processors

Processing units are physically organized in an interconnection network. Each unit is linked to a (susally small) subset of other units. Links can be unidirectional or bidirectional and are used by units to pass messages or data.

We can use graphs metrics to describe the network:

- **Distance:** $dist_G(u,v)$ of two nodes u and v is the length of the shortest path from u to v.
- **Diameter:** $diam(G)$ is the maximum distance between any two nodes.
- **Bandwidth:** $\delta_G(S)$ of a subset S of nodes is the maximum number of edges connecting the partitions S and $G \setminus S$.
  $$\delta_G(S) = |\{e \in E \text{ s.t. } e \in S \times (G \setminus S)\}|$$
  it provides an upper bound on the communications that can occur simultaneously between two subsets of nodes. The bandwidth of the entire network is given by its bisection bandwidth.

11 Linear Array and Ring Networks

The **linear array** and the **ring** are interconnection networks of processors organized in the shape of a "line". Each processor is physically linked only to each of its neighbours, which in the linear array can either be two or just one for the outmost processors, instead in the ring they are always two, since processors are connected in a circle. Examples, with $P = 6$ processors, of both kinds of networks are depicted below.

- **linear array**: links are ***bidirectional***, indicated by the straight lines.

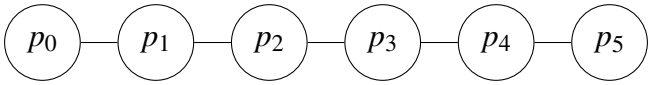


Figure 11.1: Linear array with 6 processors and bidirectional links.

- **ring**: links are ***unidirectional***, in which case they are represented using arrows, to specify the direction in which data can flow.

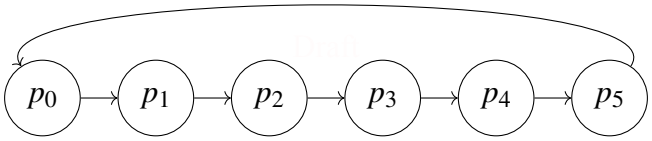


Figure 11.2: Ring with 6 processors and unidirectional links.

The number of processors and the kind of links adopted will be specified for each network depending on the use case.

Properties

Linear array	Ring
• ***Diameter*** $diam(L_P) = P - 1$, i.e., the distance between the two opposite outmost processors.	• ***Diameter*** $diam(R_P) = \lfloor P/2 \rfloor$, i.e., the maximum distance between two processors.
• ***Bisection bandwidth*** $b(L_P) = 1$, independent of the number P of processors.	• ***Bisection bandwidth*** $b(R_P) = 2$, independent of the number P of processors.

11.1 Odd-Even Transposition Sort

The **odd-even transposition sort** is a simple sorting algorithm that can be implemented on a linear array. Given N values, each in input to a different processing unit of a linear array with $P = N$, constant memory $M = 1$ and bidirectional links, the task will take at least $N - 1$ steps, since in the worst case it requires to move the value initially in the leftmost processor to the rightmost one, or the opposite. Despite being very simple, odd-even transposition sort is actually very efficient, because it always takes exactly N steps, so just one more than the previous lower bound.

The algorithm proceeds through N steps, where at each step $t = 0 \ldots N-1$, processors with indices matching the parity of t (i.e., both even or both odd) engage in comparison and potential value exchange with their right neighbor. Specifically, processor i compares its value with processor $i+1$'s value (if $i+1 < N$ exists) and swaps them if necessary to maintain ascending order. Each processor makes its decision based on its known index position in the linear array. This process continues until step $N-1$, at which point all values are guaranteed to be sorted in ascending order across the processor array.

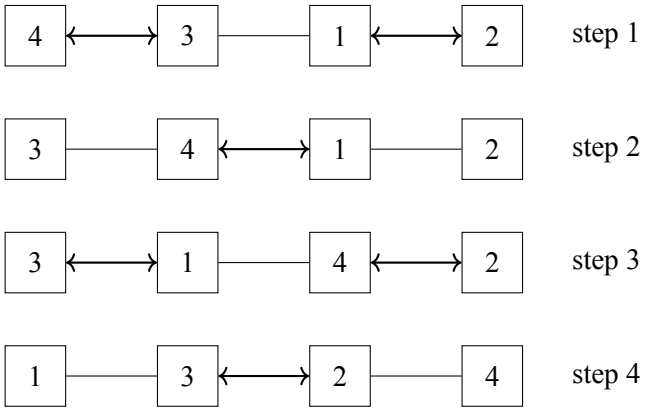


Figure 11.3: Odd-even transposition sort computation on a linear array with $P = N = 4$.

Odd-even transposition sort belongs to a class of sorting algorithms called ***oblivious comparison-exchange algorithms***. The comparison-exchange part means simply that the algorithms operates only by comparing pairs of values and possibly exchanging their positions. On the other hand, ***oblivious*** means that the behaviour of the algorithm does not depend on the input values, that is, the algorithm always performs the same sequence of comparison-exchange operations, on predetermined pairs, independently on the values to be sorted. To prove the correctness of this kind of sorting algorithms we can use the following lemma.

Lemma 3 (0-1 sorting)**.**

If an oblivious comparison-exchange algorithm sorts all input sets consisting solely of 0s and 1s, then it sorts all input sets with arbitrary values.

Proof. Assume by contradiction that the algorithm fails to sort some sequence of arbitrary values $x_0, x_1, \ldots, x_n$. Let π be a permutation such that $x_{\pi(0)}, x_{\pi(1)}, \ldots, x_{\pi(n)}$ is the output of the algorithm. Since the algorithm fails to sort the sequence, there must exist a pair of indices h and k such that $h < k$ but $x_{\pi(h)} > x_{\pi(k)}$. Now, for every $0 \leq i \leq n$ define

$$y_i = \begin{cases} 0 & \text{if } x_i \leq x_{\pi(k)} \\ 1 & \text{if } x_i > x_{\pi(k)} \end{cases}$$

Observe that, for every i and j, if $x_i \leq x_j$ then also $y_i \leq y_j$, because otherwise we would have $x_i \geq x_{\pi(k)} \geq x_j \geq x_i$. Therefore, the algorithm, being oblivious and comparison-based, would perform exactly the same exchanges when executed on either sequence, resulting in the same permutation π of before and output $y_{\pi(0)}, y_{\pi(1)}, \ldots, y_{\pi(n)}$. Since we know that $x_{\pi(h)} > x_{\pi(k)}$, then $y_{\pi(h)} = 1$ and clearly $y_{\pi(k)} = 0$. But since $h < k$, this contradicts the hypothesis that the algorithm correctly sorts all inputs with only 0s and 1s. *q.e.d.*

Using the previous lemma, in order to prove correctness of an oblivious comparison-exchange algorithm, it is enough to show that it can sort all inputs consisting of solely 0s and 1s.

For instance, in the case of odd-even transposition sort, let the input be any sequence of k 1s and $N-k$ 0s, in some order. We need to show that, within N steps, the 1s have all been moved to the k rightmost processors, that is, those indexed by $N-k, N-k+1, \ldots, N-1$. Observe that after any comparison performed between two adjacent processors, a 1 can only move to the right, or not move at all. Let k be the rightmost processor to initially have a 1. Then, all processors to its right will have 0s (since we have k 1s initially). So, depending on the parity of h, during either the first or the second step of the computation the rightmost 1 will start moving to the right, and it will continue doing so at every subsequent step until it reaches the last processor $N-1$. This also means that shortly after the second step, so at least from the third, the second rightmost 1 will also start moving to the right and will only stop once it reaches processor $N-2$. This reasoning extends to all the k-th step: in general, the i-th rightmost 1 will move at most i steps, after which it will start moving to the right and continue doing so until it reaches processor $N-i$.

Note that the i-th rightmost 1 initially can at most be located in processor $k-i$, which is distant $N-k$ from its final destination, that is, processor $N-i$. So, all is need to move at most $N-k$ times. Furthermore, since each of them will stay still for at most k steps, and will move at every step after that, this means that within $k+N-k=N$ steps all 1s will have reached their destination, and the sequence will, in fact, be sorted.

Performances

- The **parallel execution time** is $T_N(N) = \Theta(N)$, since the procedure takes exactly N steps. The total **work** performed is $W = \Theta(N^2)$, as each of the N processors performs $\Theta(N)$ operations.
- The **speedup** is $S = \Theta(N \log N) / \Theta(N) = \Theta(\log N)$ when compared to the fastest sequential sorting algorithm. Consequently, the **efficiency** is $\varepsilon = \Theta\left(\frac{\log N}{N}\right)$, indicating that the parallel implementation becomes less efficient as N grows.

11.2 Ranked Enumeration Sort

A different kind of sorting algorithms, counts for each value to be sorted how many precede it in the order. Clearly, these are not comparison-exchange algorithms. A simple algorithm of this kind is **rank enumeration sort**, meant to execute on a ring with $P = N$ processing units with constant memory $M = 2$, working as follows:

- The values, initially assigned each one to a different processing unit, are circulated around the ring, together with the index of their initial processor.
- Every time a processing unit i receives a pair (x_j, j), it adds 1 to its own counter r_i only if $x_j < x_i$, or if $x_j = x_i$ and $j < i$. Indeed, the order which is actually used is the so-called lexicographic order on the pairs (value, index).
- Since every index is unique, there can be no duplicates among the pairs.
- After every processor i had opportunity to see every other value, i.e. after $N-1$ steps, its counter r_i will be the rank of the value x_i that it originally received. So, the rank r_i is the number of values that precede x_i once the sequence is sorted, which is also the index of the processor where x_i should be at the end.
- Then, the pairs (x_i, r_i) stored in each processor i are circulated around the ring, before for another $N-1$ steps. This way, when processor r_i receives the pair (x_i, r_i), matching its own index, it will simply hold onto and store the value x_i, while continuing passing along all the others.

Thus we said once before about the lexicographic order, this method will distribute all values to their $2N-2$ total steps to obtain the sorted order. Note also that unidirectional links are enough to execute this algorithm.

Performances

- The **parallel execution time** is $T_N(N) = \Theta(N)$ since the procedure takes exactly $2N-2$ constant steps, then the work is $W = \Theta(N^2)$.
- The **speedup** is $S = \Theta(N \log N) / \Theta(N) = \Theta(\log N)$ w.r.t. the fastest sequential sorting algorithm, hence the efficiency is $\varepsilon = \Theta\left(\frac{\log N}{N}\right)$.

11.3 Discrete Convolution

The **discrete convolution** of two N-vectors $\vec{a} = (a_0, \ldots, a_{N-1})$ and $\vec{b} = (b_0, \ldots, b_{N-1})$ is the $(2N-1)$-vector $\vec{c} = (c_0, \ldots, c_{2N-2})$ where c_k, for every $0 \leq k \leq 2N-2$, is defined by

$$c_k = \sum_{\substack{i+j=k \\ 0 \leq i,j < N}} a_i b_j,$$

i.e., the sum of products of elements with indices that sum to k. Convolutions arise in a variety of applications, including signal processing and polynomial multiplication.

In order to compute every c_k on a systolic linear array with $P = 2N$ and $M = 1$, we need to find a flow of the data such that, for every i and j, the values a_i and b_j meet, simultaneously, during the same step at the processing unit computing c_{i+j}. This ensures that each component of each sum is accounted for in the right place, and that the constant memory space is not exceeded in the process.

One way to do this is to input data one value at a time from both ends of the linear array, $\vec{a}$ from the right and $\vec{b}$ from the left, and make them flow towards the opposite direction. Furthermore, to guarantee that all pairs correctly align with the corresponding processing units, the values are input in alternating orders: the first value of $\vec{a}$ at the end of $\vec{a}$ and $\vec{b}$ is input in reverse order $b_{N-1}, \ldots, b_0$, while $\vec{a}$ in the regular one.

The flow of data for the computation of the discrete convolution of two vectors of size $N = 3$ is shown in Figure 4. The computation of each c_k is done entirely by processing unit k, which updates its $c_k = c_k + a_i b_j$ every time it receives values a_i, b_j from both sides during the same step. Initially, $c_k = 0$ in every processor k.

As can be seen in the figure, each value a_i enters the linear array from the right at step $2i+1$, and then moves one to the left at every subsequent step. Therefore value a_i will be located in processor k at step $2i + 2N - k$.

Performances

- The **parallel execution time** is $T_{2N}(N) = \Theta(N)$ since the procedure takes exactly $3N-1$ constant steps, then the work is $W = \Theta(N^2)$.
- The **speedup** is $S = \Theta(N^2) / \Theta(N) = \Theta(N)$ w.r.t. naive sequential algorithms which would compute the product of each pair a_i, b_j; in such a case the efficiency is $\mathscr{E} = \Theta\left(\frac{N}{2N}\right) = \Theta(1)$, while the latter is not that informative it highlights that it is actually bounded by a constant, independent of N.

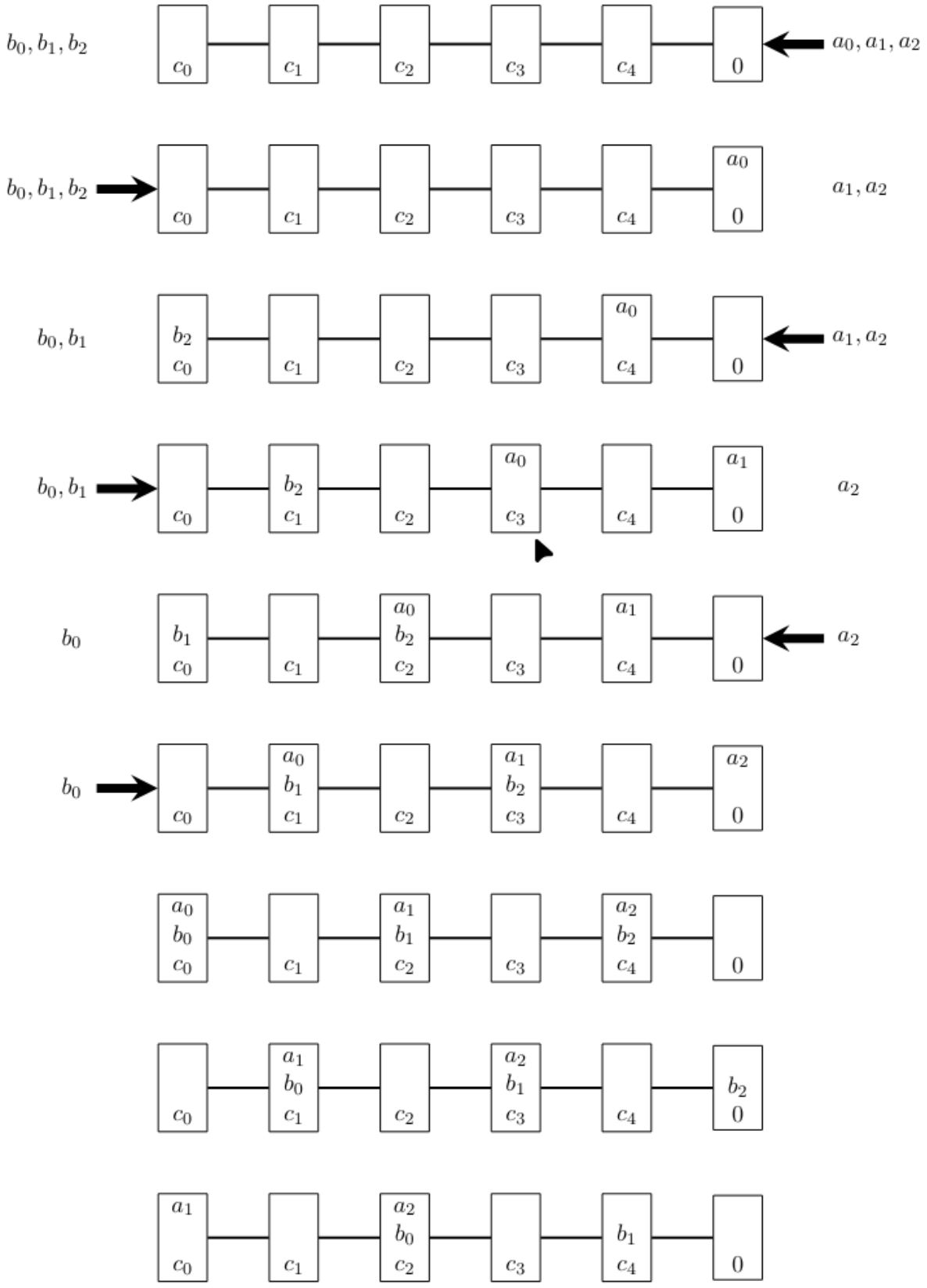


Figure 11.4: Data flow for the discrete convolution of two vectors of size 3 on a linear array.

11.4 Discrete Fourier Transform

The **Discrete Fourier Transform (DFT)** of a vector $\vec{x}$ of size N is the vector of the same size defined by

$$\hat{x}_k = \sum_{i=0}^{N-1} x_i \, \omega^{i \cdot k} \quad \text{for every } 0 \leq k \leq N-1,$$

where ω is a primitive N-th root of unity. The DFT has many uses in different fields. Among them, it can be used, along with its inverse operation (IDFT), to compute the discrete convolution.

Also in this case, we employ a parallel algorithm on a systolic linear array with $P = N$ and $M = 3$, aiming at computing each $\hat{x}_k$ in a different processing unit k. The three cells of the local memory of each processor k will be used, respectively:

- to store ω,
- to accumulate the sum $\tilde{x}_k$, and
- to store a value v_k corresponding to a power of ω depending on the current step of the sum.

The value v_k is not to be confused with the step of the execution.

We can assume that each processor has already stored ω in its first memory cell. If this were not the case, we could easily prepend ω to the input stream and have each processor store its first received value as ω. Additionally, we can initialize each processor k with $v_k = \omega^{-1}$ in its third memory cell and $\hat{x}_k = 0$ in its second memory cell.

The vector $\vec{x}$ is input to the leftmost processing unit one value at a time in the regular order. As in a standard systolic computation, values will flow towards the right from one processor to the next at each step, after going through some computation. Each time a processor k receives a value $\tilde{x}_i$ it updates its $\hat{x}_k = \hat{x}_k + \tilde{x}_i$ and $v_k = v_k \cdot \omega$, and then it sends the product $\tilde{x}_i \cdot v_k$ to the next processor to the right.

To prove that the algorithm works correctly, we have to show that each $\tilde{x}_k$ received by processor k and added to its $\tilde{x}_k$ is the correct $x_i \omega^{i \cdot k}$ component corresponding to the sum for $\tilde{x}_k$. To do this, it is enough to observe two facts.

1. After processing unit k has received the i-th value x_i and has updated its stored values, its $v_k = \omega^k$, hence it will relay the product $x_i \cdot \omega^k$.
2. At each step, the value x_i is multiplied by the appropriate power of ω as it moves through the array, so that when it reaches processor k, it is multiplied by $\omega^{i \cdot k}$.

Thus, after N steps, each processor k will have accumulated the sum $\tilde{x}_k = \sum_{i=0}^{N-1} x_i \omega^{i \cdot k}$, as required.

Performances

- The **parallel execution time** is $T_N(N) = \Theta(N)$, since each of the N values must traverse the array, and each processor performs N constant-time operations.
- The **work** is $W = \Theta(N^2)$, as each of the N processors performs N operations.
- The **speedup** is $S = \Theta(N^2) / \Theta(N \log N) = \Theta(N / \log N)$ with respect to the fastest sequential FFT algorithm, and the efficiency is $\mathscr{E} = \Theta\left(\frac{1}{\log N}\right)$.

Example: *DFT computation on a linear array*

Let's consider a concrete example with $N = 3$ to better understand how the DFT computation works on a linear array. We have three processors and want to compute the DFT of vector $\vec{x} = [x_0, x_1, x_2]$. The transformation can be written as:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \\ 1 & \omega^2 & \omega^4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_0 + x_1 + x_2 \\ x_0 + \omega x_1 + \omega^2 x_2 \\ x_0 + \omega^2 x_1 + \omega^4 x_2 \end{bmatrix}$$

Then, let's see how the computation progresses step by step:

Step	0	1	2	3	4	5
$\hat{x}_0$	0	x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	$x_0 + x_1 + x_2$	$x_0 + x_1 + x_2$
$\hat{x}_1$	0	0	x_0	$x_0 + \omega x_1$	$x_0 + \omega x_1 + \omega^2 x_2$	$x_0 + \omega x_1 + \omega^2 x_2$
$\hat{x}_2$	0	0	0	x_0	$x_0 + \omega^2 x_1$	$x_0 + \omega^2 x_1 + \omega^4 x_2$
v_0	ω^{-1}	1	ω	ω^2	ω^2	ω^2
v_1	ω^{-1}	ω^{-1}	1	ω	ω^2	ω^2
v_2	ω^{-1}	ω^{-1}	ω^{-1}	1	ω	ω^2

We can observe how each processor accumulates its partial sum over time. For instance, processor 0 computes $\hat{x}_0$ by simply adding all input values, while processor 1 multiplies each input by increasing powers of ω before adding them.

12
Mesh and Tours Networks

The *mesh* and the *torus* networks have processing units disposed as a grid. In a sense, they are the two dimensional versions of the linear array and ring, respectively. There are actually even more general mesh and torus architectures, defined for any number of dimensions, but we will focus on the basic 2-D versions.

A mesh (or torus) with P processing units can be drawn as a grid of side $\sqrt{P}$. In the mesh each inner processor has 4 links, to the four cardinal directions, while processors along the edges have fewer links, either 3 or 2 depending on their position. On the other hand, in the torus, similarly to the ring, all processors have 4 links, those along the edges simply have connections looping back to the opposite edge of the network. In this kind of network, it is often useful to use two indices, instead of just one, to identify processors, the first for the ”row” and the second for the ”column” of the network, as done for the elements of a matrix. Such indices will then range over the set $\{0,1,\ldots,\sqrt{P}-1\}$. Also in this case, we will assume that each processing unit knows its pair of indices.

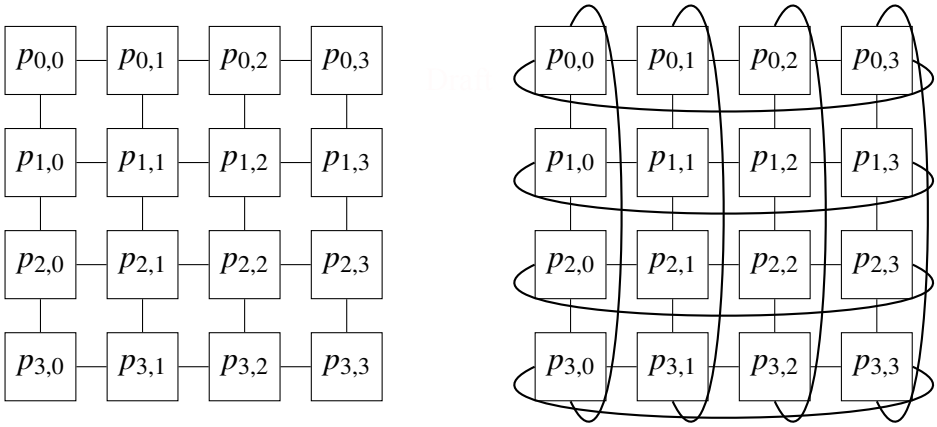


Figure 12.1: 4×4 mesh (left) and torus with 16 processing units (right)

Properties of the Mesh and Torus

A **mesh** M_P with P processors and bidirectional links has:

- **Diameter** $diam(M_P) = 2\sqrt{P} - 2$, i.e. the distance between two opposite corners.
- **Bisection bandwidth** $b(M_P) = \sqrt{P} + (\sqrt{P} \bmod 2)$.

A **torus** T_P with P processors and bidirectional links has:

- **Diameter** $diam(T_P) = 2\lfloor \sqrt{P}/2 \rfloor$, i.e. the distance between any two opposite (horizontally and vertically) processors.
- **Bisection bandwidth** $b(T_P) = 2\sqrt{P} + 2(\sqrt{P} \bmod 2)$.

12.1 Transitive Closure of a Graph

The **transitive closure** of a (directed) graph $G = (V,E)$ is denoted by $G^* = (V,E^*)$, where $E^* = \{(u,v) \in V \times V \mid \text{there is a (directed) path from } u \text{ to } v\}$. This fundamental operation is particularly useful when a graph represents relationships between objects, such as dependencies, reachability, or hierarchical structures.

Without loss of generality, we can represent the nodes of V as consecutive integers, i.e., $V = \{0,1,\ldots,|V|-1\}$. The adjacency matrix A of G is an $|V| \times |V|$ matrix where each element $A_{i,j}$ for $i,j \in V$ is defined as:

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

The transitive closure A^* of the adjacency matrix A, is simply the adjacency matrix of G^*.

Before diving into the parallel algorithm, let's outline the general procedure for computing the transitive closure. The algorithm proceeds through $|V|$ distinct phases, with each phase potentially augmenting the graph with new edges based on the current state. We represent the graph at the start of phase k as $G^k = (V,E^k)$ and at the end as $G^{k+1} = (V,E^{k+1})$. Starting with the original graph $G^0 = G$, the algorithm transforms it through all phases until reaching $G^{|V|}$, which we will prove is equivalent to the transitive closure G^*.

During each phase $0 \leq k < |V|$, we update the graph according to the rule: $E^{k+1} = E^k \cup \{(i,j) \in V \times V \mid (i,k),(k,j) \in E^k\}$. In simpler terms, if phase k finds edges from i to k and from k to j in G^k, it adds a new edge from i to j in G^{k+1}.

The key to proving that $G^{|V|}$ is indeed the transitive closure lies in this observation: for any $0 \leq k \leq |V|$, an edge (i,j) exists in G^k if and only if the original graph G contains a path from i to j that passes only through nodes in $\{0,1,\ldots,k-1\}$ (excluding i and j themselves). When we reach phase $|V|-1$, this set encompasses all nodes in V, making $G^{|V|}$ the complete transitive closure.

We can prove this observation through induction on k. For the base case $k = 0$, $G^0 = G$ contains only the original edges, which represent paths without intermediate nodes (since $\{0,\ldots,k-1\} = \emptyset$). For the inductive step, assume the property holds for some $k \geq 0$. In phase k, we add edges (i,j) whenever G^k contains edges (i,k) and (k,j). By our inductive hypothesis, this means G has paths from i to k and from k to j that only use nodes in $\{0,\ldots,k-1\}$. Combining these paths creates a path from i to j that uses nodes in $\{0,\ldots,k\}$, thus proving the property for $k+1$.

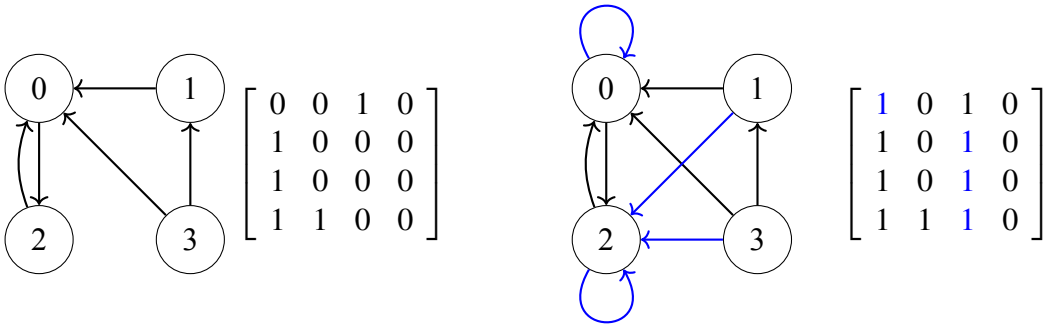


Figure 12.2: A graph and its adjacency matrix (left) and their transitive closures (right). The edges in blue are the new ones added in the transitive closure.

We can now explain the parallel algorithm to compute the transitive closure of a directed graph $G = (V,E)$ with $|V| = N$ nodes, on a mesh with the same size of the corresponding adjacency matrix A, that is, with $P = N^2$ processing units with constant memory.

The algorithm implements the N phases described above in a systolic fashion. Unlike previous systolic algorithms, each phase requires multiple steps, though some steps from different phases can be executed in parallel. The algorithm operates directly on the adjacency matrix A, which enters the mesh one row at a time from the top, with each element being assigned to its corresponding processing unit in the first row.

A key difference from the systolic algorithms we saw for linear array networks is that data doesn't automatically advance to the next row at each step. Instead, when a row A_i enters the mesh, it stops at the first available row. Since the matrix is input starting with row A_0, which stops at row 0, followed by subsequent rows, each row A_i naturally stops at row i of the mesh. This means row A_i will be in position at step $2i+1$, having waited i steps to enter the mesh and $i+1$ steps to reach its designated row.

The actual computation occurs when a row i of the mesh receives any row of the matrix other than A_i. After processing all inputs, row i of processors forwards its stored row A_i to the next row. This row has remained in place for N steps since its initial arrival, during which time all other matrix rows have passed through (with each row below taking one additional step).

Starting from the beginning, row A_i begins moving again after step $N+2i+1$. Once in motion, it flows downward through the mesh for $N-1-i$ steps until it reaches the last row of processors. The final row then outputs these values in the next step. Therefore, row A_i emerges from the last row of the mesh at step $2N+i+1$, producing the corresponding row of the transitive closure A^*.

The algorithm completes in $3N$ steps, as the last row A_{N-1} is output at step $2N+(N-1)+1=3N$.

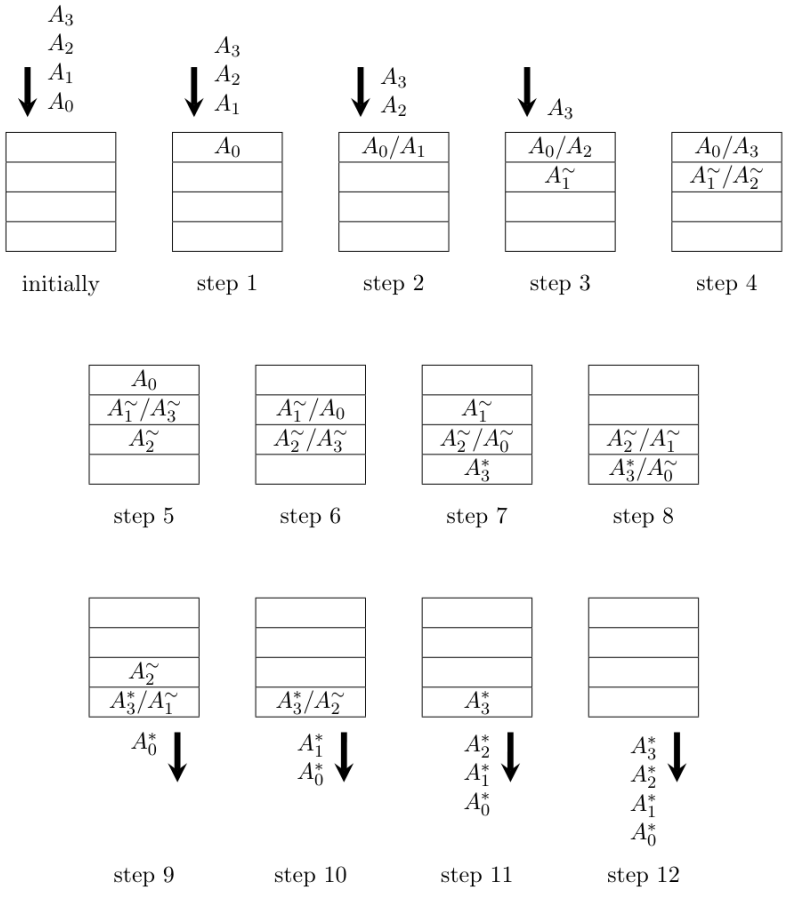


Figure 12.3: Flow of the rows of the adjacency matrix of a graph with 4 nodes for computing the transitive closure on a 4×4 mesh. A_i indicates a row that has undergone some computation but is not yet the final transitive closure, which is denoted by A^*.

Let us now examine in detail how the algorithm computes the transitive closure in N phases, as illustrated in Figure 12.3. For each node i, define $E_i = \{(u,v) \in E \mid u = i\}$ as the set of outgoing edges from i. At each phase k, we update E_i to $E_i^{k+1} = E_i^k \cup \{(i,j) \mid j \in V, (i,k) \in E_i^k, (k,j) \in E_k^k\}$. Initially, E_i^0 is given by row A_i of the adjacency matrix.

The systolic mesh computes these updates in parallel as the rows of the matrix flow through the mesh. When a row A_i (encoding E_i^0) passes over another row A_k (encoding E_k^0), the processors in row k perform the following operation: processor $p_{k,k}$ broadcasts $A_{i,k}$ horizontally to all processors in its row. Each processor $p_{k,j}$ then updates its value as $A_{i,j} \leftarrow A_{i,j} \vee (A_{i,k} \wedge A_{k,j})$. This operation checks whether there is a path from node i to node j passing through node k. This process is repeated for each $k = 0,1,\ldots,N-1$ as A_i traverses the mesh, so that by the time A_i reaches row i, it has incorporated all possible paths from i to any j passing through any subset of $\{0,\ldots,i-1\}$. At this point, A_i represents E_i^i. To complete the computation of the transitive closure, A_i must continue to traverse the remaining rows $k = i+1,\ldots,N-1$. In each of these rows, the same update is performed: processor $p_{k,k}$ broadcasts $A_{i,k}$, and each $p_{k,j}$ updates $A_{i,j} \leftarrow A_{i,j} \vee (A_{i,k} \wedge A_{k,j})$. After A_i has passed through all N rows, it encodes E_i^N, which is the set of all nodes reachable from i in the transitive closure.

It is important to note that, due to the pipelined nature of the mesh, each row A_i must wait until step $N+2i+1$ before it can begin moving again after reaching its designated row, to ensure all necessary updates from previous rows have been completed. Once this waiting period is over, A_i continues through the remaining rows, and upon exiting the mesh, the final version of A_i corresponds to the i-th row of the transitive closure matrix A^*.

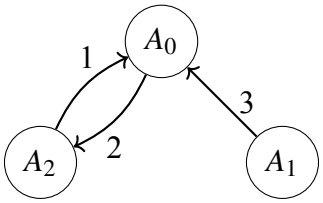
As previously noted, the algorithm requires $3N$ steps. However, a significant challenge arises from the need to broadcast values across mesh rows. In a straightforward implementation, each broadcast would require $N-1$ steps to propagate a value from one end of an N-processor row to the other. This would result in a quadratic time complexity of $O(N^2)$ constant steps. While this holds true for a naive approach, we can achieve better performance by carefully analyzing the operation dependencies. The key insight comes from applying a technique called ***retiming***, which eliminates the need to wait for broadcasts to complete. Retiming works by strategically inserting delays in some communications while advancing others, ensuring that each processor still receives its inputs in the correct order and combination. Through this optimization, we can perform all computations in $3N + 2N - 2 = 5N - 2$ constant steps. This is achieved by introducing a total delay of $2N-2$ steps, which maintains data dependencies while eliminating unnecessary broadcast waiting periods.

Performances

- The **parallel execution time** is $T_{N^2}(N) = \Theta(N)$ since with retiming the procedure takes $5N-2$ constant steps, then the work is $W = \Theta(N^3)$.
- The **speedup** is $S = \Theta(N^3)/\Theta(N) = \Theta(N^2)$ w.r.t. a naive sequential algorithm (faster sequential algorithms exist, but they are based on matrix multiplication, and so impractical due to the huge constants), then the **efficiency** is $\varepsilon = \Theta(N^2/N^2) = \Theta(1)$

12.2 Least-Weight Paths of a Graph

Most interestingly, the previous parallel algorithm for computing the transitive closure of a directed graph can be easily adapted to solve a number of graph problems. For example, the computation of the ***least-weight (directed) path*** between each pair of nodes i, j. In this setting, every edge $(i, j) \in E$ is labelled by a weight w_{ij}, and the adjacency matrix A becomes the weights matrix W, containing the corresponding weights instead of values in $\{0, 1\}$. Moreover, every possible edge will be assumed to be present in the graph, i.e. $E = V \times V$, but possibly with infinite weight. In order to ensure that a least-weight path exists for every pair of nodes, we will also assume that the graph does not contain negative-weight cycles.



$$A = \begin{bmatrix} \infty & \infty & 2 \\ 3 & \infty & \infty \\ 1 & \infty & \infty \end{bmatrix} \xrightarrow{\text{algorithm execution}} A^* = \begin{bmatrix} \infty & \infty & 2 \\ 3 & \infty & 5 \\ 1 & \infty & 3 \end{bmatrix}$$

The algorithm follows the same structure as the transitive closure computation, but with a crucial modification in the update operation. Instead of the Boolean OR operation used for reachability, each processor $p_{k,j}$ performs a distance minimization: after storing the local weight $W_{k,j}$ and receiving the flowing values $W_{i,j}$ and $W_{i,k}$, it updates $W_{i,j} \leftarrow \min(W_{i,j}, W_{i,k} + W_{k,j})$. This operation implements the core principle of dynamic programming for shortest paths: if there exists a path from i to j through intermediate node k with total weight $W_{i,k} + W_{k,j}$ that is shorter than the current known path weight $W_{i,j}$, then we update our estimate.

Upon completion, each matrix element $W_{i,j}$ contains the weight of the least-weight path from node i to node j. However, knowing only the path weights is often insufficient—we typically need to reconstruct the actual sequence of nodes that constitutes the optimal path.

Path Reconstruction. To enable path reconstruction, we maintain an auxiliary matrix X alongside the weight matrix W. For each pair of nodes (i, j), the entry x_{ij} stores the *next hop* in the optimal path from i to j. The rows of matrix X flow through the systolic mesh in parallel with the corresponding rows of W, and the broadcast operations include both weight and next-hop information.

The initialization and update procedures for the next-hop matrix are as follows:

- **Initialization:** Set $x_{ij} = j$ for all pairs (i, j), indicating that the direct edge (i, j) is initially considered the best path.
- **Update rule:** When processor $p_{k,j}$ updates the weight $W_{i,j} \leftarrow \min(W_{i,j}, W_{i,k} + W_{k,j})$, it simultaneously updates the next-hop information:

$$x_{ij} \leftarrow \begin{cases} x_{ij} & \text{if } W_{i,j} \leq W_{i,k} + W_{k,j} \\ x_{ik} & \text{if } W_{i,j} > W_{i,k} + W_{k,j} \end{cases}$$

  This ensures that x_{ij} always points to the first node on the current best path from i to j.

After the algorithm terminates, the complete least-weight path from node i to node j can be reconstructed by following the sequence of next-hop pointers: $i \to x_{ij} \to x_{x_{ij}j} \to x_{x_{x_{ij}j}j} \to \cdots$ until reaching the destination j. This reconstruction process requires at most $N - 1$ steps, where N is the number of nodes, since any simple path contains at most $N - 1$ intermediate nodes.

So, the same network and algorithm that we used to compute the transitive closure of a graph can be used to compute shortest (least-weight) paths between all pairs of nodes. Thus, the performances of the algorithm are still those given in the previous section.

12.3 Connected Components of a Graph

Another graph problem that can take advantage of the algorithm for the transitive closure is that of finding the connected components of an undirected graph. A ***connected component*** is a maximal subset $S \subseteq V$ of nodes such that for every pair of nodes $i, j \in S$, there is a path from i to j. To be maximal means that no other node $k \in V \setminus S$ in the graph is connected by a path to some node in S.

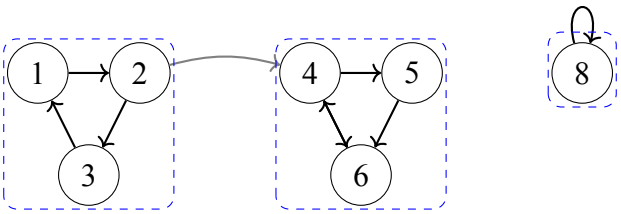


Figure 12.4: A graph with three connected components.

First, we compute the transitive closure A^* of the adjacency matrix. Then, $A^*_{ij} = 1$ if and only if nodes i and j are in the same connected component. So, it is enough to associate each node $i \in V$ with the index $k = \min\{j \mid A^*_{ij} = 1\}$, that is, the leftmost 1 in the row i of the matrix. Nodes associated with the same index k form a connected component, one for each different k.

Again, performances are similar to those of the transitive closure algorithm since the added steps, which take place after computing the transitive closure, can still be performed on the same mesh network in linear time (in parallel). However, note that the speedup and efficiency are not as good, because it exists a sequential algorithm which is able to compute the connected components in just $O(|E|)$ steps.

12.4 Matrix Multiplication

Matrix multiplication is a classical problem for which a number of algorithms have been proposed, both sequential and parallel. For simplicity we will restrict to the product of square matrices. Given two square matrices A and B of size $N \times N$, we want to compute their product $C = A \cdot B$, where each element is defined by $C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} B_{k,j}$, for every $0 \leq i, j \leq N-1$. Moreover, we want to do this on a torus of size $N \times N$, so with $P = N^2$ processors with each constant memory $M = 3$.

Since each processor $p_{i,j}$ of the network has only constant memory $M = 3$, the two input matrices must be both distributed (one element per processing unit). So, initially processor $p_{i,j}$ will contain elements $A_{i,j}$ and $B_{i,j}$ of the input matrices. The third memory cell will instead be used to store the corresponding element $C_{i,j}$ of the product. Initially, every $C_{i,j} = 0$. Observe that some processors cannot use their initial input values right away. For instance, processor $p_{0,1}$ cannot immediately compute any component of the sum $C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + \cdots + A_{0,N-1}B_{N-1,1}$, since its input values are $A_{0,1}$ and $B_{0,1}$. Therefore, to avoid idle waiting, the algorithm must first move the data to a configuration where each processing unit contains a pair of input values that can immediately use for computing one addend of the corresponding element of C. To understand what such a configuration could be, we first give an example. Suppose $N = 3$.

$$
\begin{aligned}
C_{0,0} &= A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0} & \qquad C_{1,0} &= A_{1,0}B_{0,0} + A_{1,1}B_{1,0} + A_{1,2}B_{2,0} \\
C_{0,1} &= A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1} & \qquad C_{1,1} &= A_{1,0}B_{0,1} + A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{0,2} &= A_{0,0}B_{0,2} + A_{0,1}B_{1,2} + A_{0,2}B_{2,2} & \qquad C_{1,2} &= A_{1,0}B_{0,2} + A_{1,1}B_{1,2} + A_{1,2}B_{2,2}
\end{aligned}
$$

$$
\begin{aligned}
C_{2,0} &= A_{2,0}B_{0,0} + A_{2,1}B_{1,0} + A_{2,2}B_{2,0} \\
C_{2,1} &= A_{2,0}B_{0,1} + A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_{2,2} &= A_{2,0}B_{0,2} + A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{aligned}
$$

■ Possible elements that can be computed immediately after initial data placement

Now, for each $C_{i,j}$, we need to find a pair of values $A_{i,k}$ and $B_{k,j}$ such that they will not be chosen for any other $C_{i',j'}$. This is due to the fact that all inputs must still be uniquely distributed, a different pair to each processing unit. There are many configurations that satisfy such conditions, even just by symmetry.

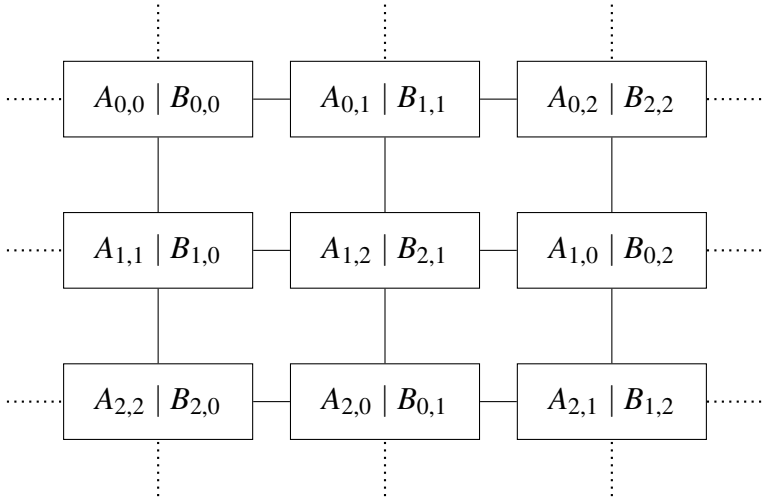


Figure 12.5: A distribution of the input allowing each processor to perform some computation.

Let us analyze the configuration shown in Figure 12.5. Initially, each processor $p_{i,j}$ holds elements $A_{i,j}$ and $B_{i,j}$. The data movement pattern reveals that each row i of matrix A undergoes a leftward rotation by i positions, while each column j of matrix B rotates upward by j positions. This pattern generalizes to any dimension N: after the rotations, processor $p_{i,j}$ contains elements $A_{i,k}$ and $B_{k,j}$, where $k = i + j \mod N$. The initial data movement phase requires $N-1$ steps to achieve this configuration, since the last row of A and last column of B must move $N-1$ positions. We assume this initial configuration is not provided, making this phase necessary.

The computation phase alternates between arithmetic operations and communication steps. In each step, processor $p_{i,j}$:

1. Updates its partial result: $C_{i,j} = C_{i,j} + A_{i,k} \cdot B_{k,j}$
2. Forwards $A_{i,k}$ to its left neighbor and $B_{k,j}$ upward
3. Receives new values $A_{i,k'}$ and $B_{k',j}$ from right and below, where $k' = k+1 \mod N$

This creates a continuous rotation pattern: rows of A shift leftward while columns of B shift upward. After N such steps, each processor has processed all required value pairs for its final computation. The total execution time is $2N-1$ steps ($N-1$ for initial configuration plus N for computation), after which the torus contains the complete product matrix C.

The algorithm extends naturally to cases where $P < N^2$ processors are available, forming a $\sqrt{P} \times \sqrt{P}$ torus (assuming N is a multiple of $\sqrt{P}$). Here, we employ block distribution: each processor $p_{i,j}$ manages $n \times n$ blocks $A(i,j)$ and $B(i,j)$, where $n = N/\sqrt{P}$. The block $A(i,j)$ has the following structure:

$$A(i,j) = \begin{bmatrix} A_{in,\,jn} & A_{in,\,jn+1} & \cdots & A_{in,\,(j+1)n-1} \\ A_{in+1,\,jn} & A_{in+1,\,jn+1} & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ A_{(i+1)n-1,\,jn} & \cdots & \cdots & A_{(i+1)n-1,\,(j+1)n-1} \end{bmatrix}$$

The block structure for $B(i,j)$ follows the same pattern as $A(i,j)$. In this block-based approach, each processor $p_{i,j}$ computes its corresponding block $C(i,j)$ of the product matrix C. This requires increased memory per processor: $M = 3N^2/P$ elements, which grows with the problem size. The algorithm maintains the same two-phase structure as before, but operates on blocks rather than individual elements.

The block-based algorithm proceeds as follows:

1. **Initial Configuration Phase:** Each row i rotates its blocks $A(i,\_)$ leftward by i positions, while each column j rotates its blocks $B(\_,j)$ upward by j positions.
2. **Computation Phase:** In each step, processor $p_{i,j}$:
   - Updates its block: $C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)$
   - Forwards blocks $A(i,k)$ leftward and $B(k,j)$ upward

Note that now the steps are not constant any more, not even the communications, since they involve blocks of N^2/P elements. Furthermore, the arithmetical operations require to compute a matrix product, since the blocks $A(i,k)$ and $B(k,j)$ are actually matrices. On the other hand, the two phases take just $\sqrt{P}-1$ and $\sqrt{P}$ steps, respectively.

Performances

- The **parallel execution time** is $T_{N^2}(N) = \Theta(N)$ since the procedure takes $2N-1$ constant steps, then the work is $W = \Theta(N^3)$.
- The **speedup** is $S = O(N^{2.37})/\Theta(N) = O(N^{1.37})$ w.r.t. the fastest sequential algorithm (although, as already mentioned, such an algorithm is actually impractical due to the huge constants), then the **efficiency** is $\varepsilon = O(N^{1.37}/N^2) = O(1/N^{0.63})$.

Tip: ***Matrix multiplication***

Element $A_{i,j}$ stays in row i but moves leftward by i positions, while element $B_{j,i}$ stays in column i but moves upward by j positions. This means that their new index will be:

$$A_{i,j} \to A_{i,(j-i)} \quad \text{and} \quad B_{j,i} \to B_{(j-i),i}$$

Thus, in processor $p_{i,j}$ we will have the elements:

$$A_{i,(j+i)} \quad \text{and} \quad B_{(i+j),j}$$

where all indices are taken modulo N.

13 Binary Tree Networks

The **binary tree** is one of the most fundamental data structures in computer science, and it naturally lends itself to parallel computing as an interconnection network. In this architecture, each processing unit can have up to three connections: one to its parent (above), and potentially two to its children (left and right). The network topology can be characterized by the position of each processor within the tree structure. Leaf processors have only one link (to their parent), the root processor has two links (to its children), and all other inner nodes maintain three connections. This hierarchical organization is typically sufficient for algorithm design, making explicit processor indexing unnecessary in most cases.

We focus on **complete** binary trees, where all levels are fully populated, ensuring uniform distance from the root to all leaves. This distance defines the tree's **height** h. For a binary tree with P processing units, the height is given by $h = \log_2(P+1) - 1$. Conversely, a tree of height h contains $P = 2^{h+1} - 1$ processors, with 2^h of these being leaf nodes. Figure 13.1 illustrates a complete binary tree of height 3.

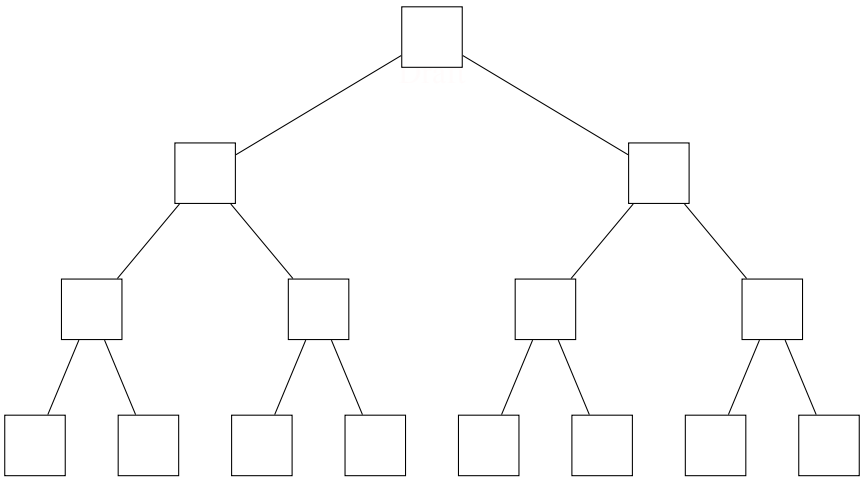


Figure 13.1: Binary tree of height 3 with $2^{3+1} - 1 = 15$ processing units.

Properties

A binary tree B_P of height $h = \log_2(P+1) - 1$ with P processing units and bidirectional links has the following properties:

- **Diameter** $diam(B_P) = 2h$, i.e. the distance between any two leaves on opposite sides w.r.t. the root.
- **Bisection bandwidth** $b(B_P) = 1$, independent of the number P of processors.

13.1 Associative Operations

The binary tree is especially suited for the computation of associative operations on some elements. Formally, this applies to any *semigroup* $(S, \otimes)$, where S is a set and $\otimes : (S \times S) \to S$ is an *associative* binary operation, i.e. such that $(x \otimes y) \otimes z = x \otimes (y \otimes z)$ for all $x, y, z \in S$. Examples of such operations are addition, multiplication, maximum and minimum, logical operations AND and OR, and many others. The "conquer" phases of some divide-and-conquer procedures can also be expressed by associative operations.

Consider a binary tree network of height $h = \log N$. This network can efficiently compute any associative operation $\otimes$ on N elements in just $h = \log N$ steps, assuming the operation itself takes constant time. The algorithm proceeds as follows:

1. Initially, each leaf processor receives one of the N input elements. Given the tree's height h, there are exactly $2^h = N$ leaves and $P = 2N - 1$ total processing units.
2. During computation, values propagate upward from children to parents. When an inner node receives values x and y from its children, it computes $x \otimes y$ and forwards the result to its parent.
3. The root node, being the last to receive values, computes the final result and terminates the computation.

Warning: ***Associativity does not imply commutativity***

Note that associativity does not imply commutativity. The operation $\otimes$ need not be commutative for this algorithm to work correctly.

The algorithm effectively computes the sequence $x_0 \otimes x_1 \otimes \cdots \otimes x_{N-1}$ by combining adjacent sub-results. This approach preserves the order of operations, ensuring the final result matches the sequential computation. Each processing unit requires only constant memory, provided that elements of the semigroup S have bounded size. Figure 13.2 illustrates this process for the addition.

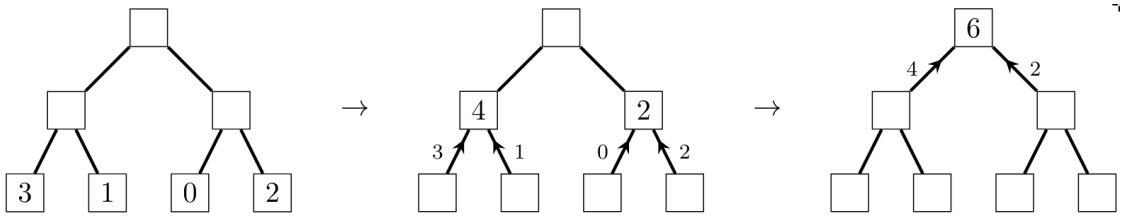


Figure 13.2: Addition of 4 values on a binary tree of height 2.

Performances

- The **parallel execution time** is $T_{2N-1}(N) = \Theta(\log N)$ since the procedure takes $h = \log N$ constant steps, then the work is $\mathscr{W} = \Theta(N \log N)$.
- The **speedup** is $\mathscr{S} = \Theta(N) / \Theta(\log N) = \Theta(N / \log N)$ w.r.t. the sequential iteration of the $N - 1$ binary operations, then the **efficiency** is $\mathscr{E} = \Theta\left(\frac{N / \log N}{2N-1}\right) = \Theta(1 / \log N)$.

13.2 Prefix Computation

Another common problem related to associative operations is to compute the **prefix operation** of a sequence of elements. Given an associative operation $\otimes$ and a sequence $x_0, x_1, \ldots, x_{N-1}$ of N elements, we want to apply $\otimes$ to every prefix of the sequence to obtain another sequence of the same length:

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 \otimes x_1 \\ y_2 &= x_0 \otimes x_1 \otimes x_2 \\ &\vdots \\ y_{N-1} &= x_0 \otimes x_1 \otimes \cdots \otimes x_{N-1} \end{aligned}$$

We will again use a binary tree with $2N-1$ processing units with constant memory, of which N are leaves where the input will initially be placed, one element in each leaf from left to right. The computation of the N prefixes is performed in two phases, at the end of which each leaf of the binary tree will contain the corresponding prefix, i.e., the result of the operation $\otimes$ applied to the sequence of elements initially contained in that leaf and those to its left.

The first phase is ascending, similar to the procedure described in the previous section, where all data is sent upwards and inner nodes forward the results $x_L \otimes x_R$ of the binary operation applied to the values x_L and x_R received from their children. However, in this case, inner nodes also store in their memory the value x_L that they received from their left child, only the left one. Eventually, the root of the binary tree will receive from its left child the result of the operation $\otimes$ applied to the values initially contained in the leaves of the left subtree. The same will happen for the right subtree from the right child, although this value is actually useless to the root.

At this point, the second phase begins, which, instead, is a descending one. First, every inner node will send its stored value only to its right child. Then, such values will be forwarded downwards to both children by every following node. Meanwhile, whenever a leaf receives a value z from above, it updates $x_i := z \otimes x_i$, where x_i initially was its given input.

The evolution of the state of the network through the whole computation with 4 input values is graphically represented in Figure 13.3, where $\otimes$ is the classical addition.

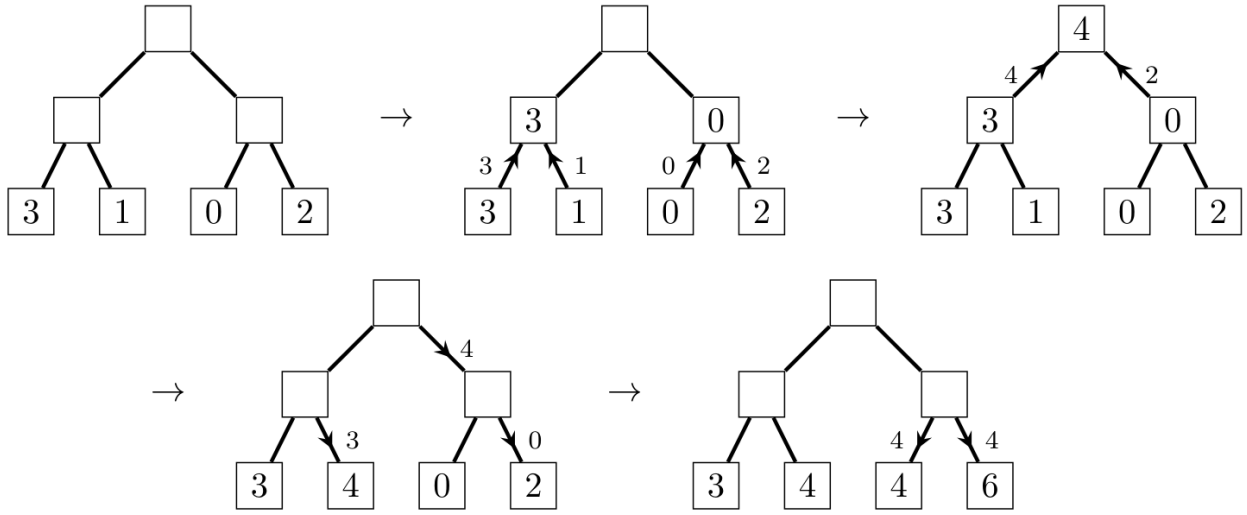


Figure 13.3: Prefix computation on a binary tree of height 2.

The two phases, described above as happening separately one after the other, actually can and should overlap as much as possible. This also makes sense with respect to the instructions assigned to the processing units of the network. Indeed, instead of globally separating the two phases, the algorithm/program would simply prescribe to act as follows. First, every leaf sends its input value x_i upwards. Then, every processing unit just acts in response to receiving a value from some of its links.

- The first time (and only one in fact) a processor receives values x_L and x_R from its left and right child, respectively, it forwards x_L to its right child, computes $x_L \otimes x_R$ and forwards the result upwards to its parent (the root will ignore the latter operation).
- Every time a processor receives a value x from above, if it is a leaf it updates its $x_i := x \otimes x_i$, otherwise it forwards such a value x to both its left and right children.

Since, at most, values will need to flow from the leaves up to the root and back, after $2h = 2\log N$ steps every leaf will have received all necessary values to compute the corresponding prefix.

The correctness of the results can be proved by induction on the height h of the binary tree. In the base case, when $h = 1$, the root, which is the only inner node, will just receive x_0 from the left leaf (ignoring the value coming from the right) and forward x_0 to the right leaf, which then will compute $x_0 \otimes x_1$, as desired. Now, to prove the correctness for a tree of height $h = k+1$, assume that the algorithm produces the correct results on the left and right subtrees of height k, which would be

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 \otimes x_1 \\ &\vdots \\ y_{N/2-1} &= x_0 \otimes x_1 \otimes \cdots \otimes x_{N/2-1} \end{aligned} \qquad \text{and} \qquad \begin{aligned} z_0 &= x_{N/2} \\ z_1 &= x_{N/2} \otimes x_{N/2+1} \\ &\vdots \\ z_{N/2-1} &= x_{N/2} \otimes x_{N/2+1} \otimes \cdots \otimes x_{N-1} \end{aligned}$$

respectively, since $N = 2^{k+1}$. Furthermore, it is easy to see that the root will receive value $y_{N/2-1}$ from its left child, that is the root of the left subtree, since every inner node forwards upwards the result of the operation applied to the sequence of input values in the subtree rooted in such a node. Then, the root of the binary tree will forward the received value $y_{N/2-1}$ only to its right child. Such a value will be forwarded downwards to all descendants, and so all the leaves, in the right subtree. We conclude by observing that, after receiving such a value, every leaf in the right subtree will compute the corresponding $y_{N/2+j} = x_0 \otimes \cdots \otimes x_{N/2-1} \otimes x_{N/2} \otimes \cdots \otimes x_{N/2+j} = y_{N/2-1} \otimes z_j$, for $0 \leq j \leq N/2-1$. And so, each leaf in general will finally contain the corresponding prefix $y_j = x_0 \otimes x_1 \otimes \cdots \otimes x_j$, for all $0 \leq j \leq N-1$ starting from the leftmost leaf.

Performances

- The **parallel execution time** is $T_{2N-1}(N) = \Theta(\log N)$ since the procedure takes $2h = 2\log N$ constant steps, then the work is $W = \Theta(N \log N)$.
- The **speedup** is $S = \Theta(N)/\Theta(\log N) = \Theta(N/\log N)$ w.r.t. the sequential iteration of the $N-1$ binary operations (recording every partial result corresponding to every prefix), then the efficiency is $\varepsilon = \Theta(1/\log N)$.

13.3 Selection

The logarithmic-time computation of cumulative operations enables efficient solutions to many complex problems. A particularly important example is the **selection** problem: finding the k-th smallest element in a set, also known as the k-th ***order statistic***. This generalizes finding the maximum, minimum, and especially the ***median*** of a set. The key insight is to use a divide-and-conquer approach: find a pivot element that partitions the set into two roughly equal parts, then recursively search only the relevant partition based on k and the partition sizes.

Consider a set of N values, each represented as B-bit binary numbers. We use a binary tree with N leaves (one per input value) and $2N-1$ total processors. The algorithm proceeds through B iterations, examining one bit position at a time from most to least significant. Each leaf processor holds one input value and is initially "active". The root stores the current problem size N and target index k.

For each iteration i (examining bit position i), the algorithm has two phases:

1. **Ascending phase:** Active leaves send their i-th bit upward. Inner nodes compute the sum of received bits and forward to their parents. The root receives the total count S of active values with bit i equal to 1.
2. **Descending phase:** The root determines whether the target element has bit i equal to 0 or 1:
   - If $k \leq N - S$: target has bit $i = 0$, update $N := N - S$
   - Otherwise: target has bit $i = 1$, update $k := k - (N - S)$ and $N := S$

   The root broadcasts this decision downward. Leaves with mismatching bit i deactivate.

After B iterations, all remaining active leaves contain the same value: the k-th smallest element.

To prove its correctness, we partition the remaining values based on bit i, creating an implicit pivot of the form $b_1 2^{B-1} + \cdots + b_{i-1} 2^{B-i+1} + 2^{B-i}$ where $b_1, \ldots, b_{i-1}$ are the previously determined bits. The algorithm correctly identifies which partition contains the target element and eliminates the other partition.

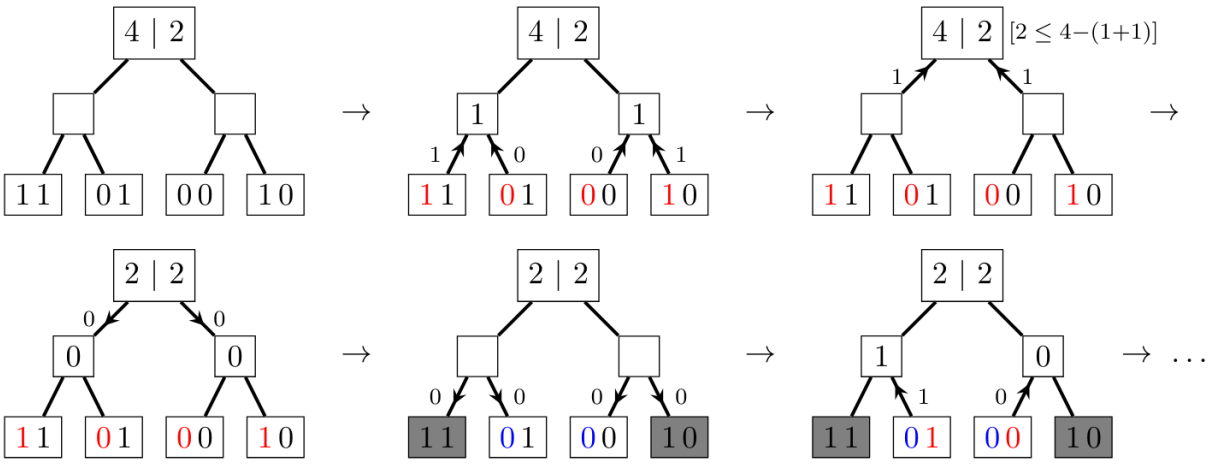


Figure 13.4: First part of the selection of the 2nd smallest of 4 binary values with 2 bits.

In general, the i-th iteration, for $1 \leq i \leq B$, works as follows. Assume that N' leaves are still active and that their inputs have identical bits from the most significant one up to the $(i-1)$-th bit (this invariant holds initially and is maintained after each iteration).

Let N' and k' be the values currently stored in the root. During the ascending phase, the N' active leaves send upward the i-th bit of their input. The inner nodes perform cumulative sum operations on these bits, with the total reaching the root. The cumulative sum S represents the exact number of remaining inputs with their i-th bit equal to 1. Consequently, there are $N' - S$ inputs with i-th bit equal to 0 and S inputs with i-th bit equal to 1.

Since all active values share the same $i-1$ leftmost bits $b_1, b_2, \ldots, b_{i-1}$, they can be partitioned by the pivot value $b_1 2^{B-1} + b_2 2^{B-2} + \cdots + b_{i-1} 2^{B-i+1} + 2^{B-i}$. The descending phase begins with the root determining whether the i-th bit of the target element is 0 or 1, based on the condition $k' \leq N' - S$. The root updates its stored values accordingly and broadcasts this decision to its children.

During the descending phase, the N' active leaves receive the bit information and decide whether to deactivate or remain active based on their input's i-th bit. After the i-th iteration, only leaves containing inputs whose i leftmost bits match exactly those of the target element (computed inductively during the previous i iterations) remain active.

Therefore, after all B iterations complete, all remaining active leaves contain the same value: the target k-th smallest element. A final ascending phase collects this result, with any node (including the root) outputting the answer. Alternatively, any remaining active leaf can directly output the result, since all contain identical values.

Performances

- The **parallel execution time** is $T_{2N-1}(N) = \Theta(B \log N)$ since the algorithm performs B iterations, each taking $\Theta(\log N)$ constant steps, and a final ascending phase for another $\log N$ steps, then the work is $W = \Theta(BN \log N)$.
- The **speedup** is $S = O(N)/\Theta(B \log N) = O\left(\frac{N}{B \log N}\right)$ w.r.t. a general sequential selection algorithm, such as the *median-of-medians* one, then the efficiency is $\varepsilon = O\left(\frac{1}{B \log N}\right)$.

Parallel Computing - Key concepts

The following table summarizes the primary metrics used in parallel algorithm analysis:

Metric	Definition	Meaning or Impact
N	Problem size	Number of elements in the input of the problem
P	Number of processors	Number of processors used to solve the problem
$T = T_p(n)$	Time complexity	Execution time by each task or best parallel execution time
M	Space complexity	Memory requirements for the parallel algorithm
$W = T_p \cdot P$	Work	Total operations in a task/analysis
$S = O(f(n))$	Space complexity	Memory requirements for the parallel algorithm
W^c	Communication cost	Cost of data transfer between processors
$C = \frac{W^c}{W}$	Communication overhead	Ratio of communication work to computational work
Γ	Sequential execution time	Execution time of the best sequential algorithm
$S = \frac{\Gamma}{T}$	Speedup	Ratio between sequential time and parallel time
$\varepsilon = \frac{S}{P}$	Efficiency	Measure of processor utilization (how effectively processors are used)
$S = \frac{T_1(n)}{T_p(n)} \leq \frac{W}{T_p(n)}$	Work-span formula	Theoretical bound relating work and parallel time

The fundamental relationship between these metrics can be expressed as:

$$S = \frac{\Gamma}{T} \leq \frac{T_1(n)}{T} \leq \frac{W}{T} = P$$

This inequality establishes the theoretical limits of parallel performance by showing that speedup is bounded by the ratio of work to parallel time. The efficiency metric follows directly:

$$\varepsilon = \frac{S}{P} = \frac{\Gamma}{T} \cdot \frac{T}{W} = \frac{\Gamma}{W} \leq 1$$

The bound $S \leq P$ holds because any P-processor algorithm running in T steps can be simulated sequentially in $T \cdot P$ steps by executing each parallel step serially. Therefore, $\Gamma \leq T \cdot P = W$, which establishes the fundamental work-time relationship in parallel computing.

The primary optimization goals in parallel algorithm design are to minimize execution time T (fast) while maximizing efficiency ε (efficient). These objectives often present trade-offs, as adding more processors may reduce T but can decrease ε due to increased communication costs and diminishing returns from parallelization.

Bibliography

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