Algorithms & Data Structures

Fabian Bosshard

June 17, 2025

Contents

1 Introduction

$T(n) :=$ number of **basic steps** needed to compute the result of a problem of size n

Example 1. How many $1,2$-beats can one compose over a total of n beats?

Idea: the number of $1,2$-beats over n beats is the sum of the number of $1,2$-beats over $n-1$ beats and the number of $1,2$-beats over $n-2$ beats

Algorithm 1 Pingala

```
1: function PINGALA(n)                          ▷ Number of 1,2-beats in n beats
2:     if n ≤ 2 then
3:         return n
4:     return PINGALA(n − 1) + PINGALA(n − 2)
```

$$T(1) = 1, \quad T(2) = 1, \quad T(n) = T(n-1) + T(n-2) + 2$$
$$T(n) \geq \underbrace{T(n-1)}_{\geq T(n-2)} + T(n-2) \geq 2T(n-2)$$
$$T(n) \geq 2T(n-2) \geq 2(2T(n-4)) = 2^2 T(n-4) \geq \ldots \geq 2^{n/2} T(0) = \sqrt{2}^n T(0) \quad \blacktriangleleft$$

2 Basics

Definition 1. We define the following families of functions:

- $O(g(n)) := \{f(n) \mid \exists c > 0, n_0 \in \mathbb{N} \mid 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$
- $\Omega(g(n)) := \{f(n) \mid \exists c > 0, n_0 \in \mathbb{N} \mid 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$
- $\Theta(g(n)) := \{f(n) \mid \exists c_1, c_2 > 0, n_0 \in \mathbb{N} \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$
- $o(g(n)) := \{f(n) \mid \forall c > 0, \exists n_0 \in \mathbb{N} \mid 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$
- $\omega(g(n)) := \{f(n) \mid \forall c > 0, \exists n_0 \in \mathbb{N} \mid 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$ ✍

The notation '$f(n) =$' is used to denote that $f(n)$ is an element of the set of functions on the right-hand side.

Example 2. Let $\pi(n)$ be the number of primes less than or equal to n. Then

$$\pi(n) = \Theta\left(\frac{n}{\log n}\right). \quad \blacktriangleleft$$

The Θ-notation, Ω-notation, and O-notation can be viewed as the "asymptotic" $=$, $\geq$, and $\leq$ relations for functions. The o-notation and ω-notation can be viewed as asymptotic $<$ and $>$.

Theorem 1. $f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)) \Leftrightarrow f(n) = \Theta(g(n))$ $\triangleleft$

The above theorem can be interpreted as saying

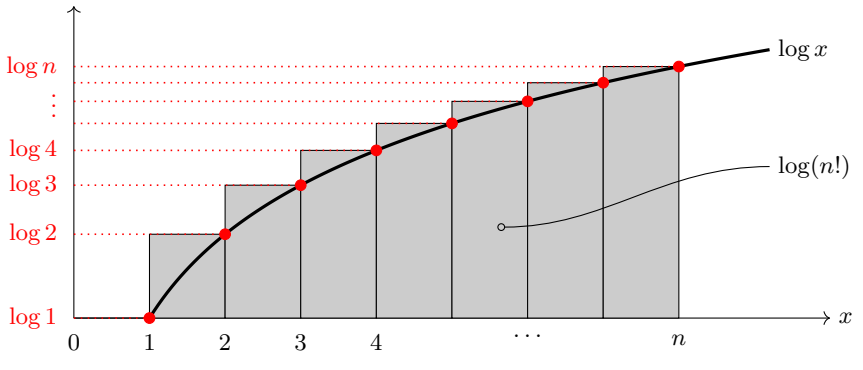
$$f \geq g \wedge f \leq g \Leftrightarrow f = g$$

Example 3. $\log(n!) \in \Theta(n \log n)$. We can rewrite $\log(n!)$ as

$$\log(n!) = \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log i \tag{1}$$

Clearly, $\log(n!) \in O(n \log n)$, since $n \log n = \log(n^n) = \sum_{i=1}^{n} \log n \geq \sum_{i=1}^{n} \log i$.

One way to understand why $\log(n!) \in \Omega(n \log n)$ is to interpret (1) as a sum of areas, each rectangle has width 1 and height $\log i$:



We can see that the area of the gray rectangles is bounded from below by the area under the curve $y = \log x$, i.e.

$$\int_1^n \log x \, \mathrm{d}x \leq \log(n!) \tag{2}$$

Using integration by parts, we can express the left-hand side of (2) as

$$\int_1^n \log x \, \mathrm{d}x = \int_1^n \underset{\uparrow}{1} \cdot \underset{\downarrow}{\log x} \, \mathrm{d}x = \left[x \log x - \int x \cdot \frac{1}{x} \, \mathrm{d}x\right]_1^n = [x \log x - x]_1^n$$
$$= n \log n - n + 1 \in \Omega(n \log n)$$

Therefore, $\log(n!) \in \Omega(n \log n) \wedge \log(n!) \in O(n \log n)$. Using Theorem 1, we conclude that $\log(n!) \in \Theta(n \log n)$. $\blacktriangleleft$

When $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** for $f(n)$, and that $g(n)$ **dominates** $f(n)$.

When $f(n) = \Omega(g(n))$, we say that $g(n)$ is a **lower bound** for $f(n)$.

When $f(n) = \Theta(g(n))$, we say that $g(n)$ is a **tight bound** for $f(n)$.

We use the o-notation to denote an upper bound that is not asymptotically tight, and the ω-notation to denote a lower bound that is not asymptotically tight. The following two implications hold:

$$f(n) = o(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad f(n) = \omega(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Notation	Name	Example
$O(1)$	constant	Finding median of *sorted* array; computing $(-1)^n$; using a fixed-size lookup table.
$O(\alpha(n))$	inverse Ackermann	Amortized cost per operation in a disjoint-set data structure.
$O(\log^* n)$	iterated logarithmic	Distributed coloring of cycles (Cole-Vishkin algorithm).
$O(\log \log n)$	double logarithmic	Interpolation search on uniformly distributed data.
$O(\log n)$	logarithmic	Binary search; operations in balanced trees or a binomial heap.
$O(\log^c n)$, $c > 1$	polylogarithmic	Matrix-chain ordering on a PRAM.
$O(n^c)$, $0 < c < 1$	fractional power	Searching in a k-d tree.
$O(\frac{n}{\log n})$		#Primes $\leq n$ (Example 1).
$O(n)$	linear	Scanning an unsorted list; ripple-carry addition of two n-bit integers.
$O(n \log^* n)$		Seidel's polygon-triangulation algorithm.
$O(n \log n) = O(\log n!)$	linearithmic	Fastest comparison sorts; Fast Fourier transform.
$O(n^2)$	quadratic	Schoolbook multiplication; bubble/selection/insertion sort; worst-case quicksort; direct convolution.
$O(n^3)$	cubic	Naive $n \times n$ matrix multiplication; partial correlation.
$O(n^c)$, $c > 1$	polynomial	TAG parsing; bipartite matching; determinant via LU.
$L_n[\alpha, c]$	sub-exponential	Factoring via number-field sieve.
$O(c^n)$, $c > 1$	exponential	Exact TSP by DP; brute-force logical equivalence checking.
$O(n!)$	factorial	Brute-force TSP; enumerating permutations or partitions; determinant by Laplace expansion.

2.1 Incremental Algorithms

Example 4 (Hand of cards)**.** Insertion sort (Algorithm 3) uses an algorithm design technique called **incremental** method: for each element $A[i]$, it inserts it into its proper place in the subarray $A[1 : i]$, having already sorted $A[1 : i - 1]$. This is reminiscent of how one might sort a hand of cards, where you pick up a card and insert it into the correct position in the already sorted hand.

At the start of each iteration of the for loop, the subarray $A[1 : i - 1]$ consists of the elements originally in $A[1 : i - 1]$, but in sorted order. This is a loop invariant (Section 2.2). ◀

2.2 Loop Invariant

When using a **loop invariante**, 3 things need to be shown:

1. **Initialization:** It is true prior to the first iteration of the loop.
2. **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination:** When the loop terminates, the invariant gives a useful property that helps show that the algorithm is correct.

A loop-invariant proof is a form of **mathematical induction**, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step. The third property is perhaps the most important one, since you are using the loop invariant to show correctness. Typically, you use the loop invariant along with the condition that caused the loop to terminate. Mathematical induction typically applies the inductive step infinitely, but in a loop invariant the "induction" stops when the loop terminates.

2.3 Correctness

You are given a problem P and an algorithm A. P formally defines a **correctness** condition. Assume, for simplicity, that A consists of one loop.

1. Formulate an invariant C
2. **Initialization**: prove that C holds right before the first execution of the first instruction of the loop
3. **Management**: prove that if C holds right before the first instruction of the loop, then it holds also at the end of the loop
4. **Termination**: prove that the loop terminates, with some exit condition X
5. Prove that $X \wedge C \Rightarrow P$, which means that A is correct

2.4 Divide-and-Conquer Algorithms

Many useful algorithms are **recursive**: they **recurse** (call themselves) one or more times to handle closely related subproblems. These algorithms typically follow the **divide-and-conquer** method: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

In the divide-and-conquer method, if the problem is small enough (the **base case**), you just solve it directly without recursing. Otherwise (the **recursive case**), you perform three characteristic steps:

1. **Divide** the problem into one or more subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the subproblem solutions to form a solution to the original problem.

When an algorithm contains a recursive call, we can often describe its running time with a **recurrence relation**, which expresses the overall running time of a problem of size n in terms of the running time of the same algorithm on smaller inputs.

Let $T(n)$ be the worst case running time on an input of size n. If the problem is small enough, say $n \leq n_0$, for some constant n_0, the straightforward solution takes constant time $\Theta(1)$. Suppose that the division of the problem yields a subproblems, each of size n/b.[1] If it takes $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_0 \\ D(n) + aT(n/b) + C(n) & \text{if } n > n_0 \end{cases} \tag{3}$$

Theorem 2 (Master Theorem)**.** Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n)$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta\left(n^{\log_b a} \lg^k n\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$, and if $f(n)$ additionally satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ◁

2.5 Binary Search

Algorithm 2 is an efficient method for finding an element x in a sorted array A. By repeatedly halving the search interval, it reduces the problem size exponentially: at each step, it compares x to the middle element of the current interval and discards the half in which x cannot lie. This yields a worst-case running time of $O(\log n)$, a dramatic improvement over a linear search's $O(n)$ behavior.

Algorithm 2 Binary Search

```
1:  function BinarySearch(A, x)
2:      l ← 1                                   ▷ leftmost index
3:      r ← len(A)                              ▷ rightmost index
4:      while l ≤ r do
5:          m ← ⌊(l + r)/2⌋                     ▷ midpoint of A[l : r]
6:          if A[m] < x then
7:              l ← m + 1                       ▷ search in right half
8:          else if A[m] > x then
9:              r ← m − 1                       ▷ search in left half
10:         else
11:             return m                        ▷ found x at index m
12:     return 0                                ▷ not found: 0 is not a valid index
```

3 Sorting

3.1 Insertion Sort

Was already used to sort cards in Section 2.1, Example 4.

Algorithm 3 Insertion Sort

```
1: function InsertionSort(A)
2:     for i = 2, ..., len(A) do
3:         j ← i
4:         while j > 1 ∧ A[j − 1] > A[j] do
5:             swap A[j] and A[j − 1]
6:             j ← j − 1
```

3.2 Merge Sort

Algorithm 4 Merge Sort

```
1: function MergeSort(A)
2:     if len(A) ≤ 1 then                      ▷ base case (array is trivially sorted)
3:         return A
4:     m = ⌊len(A)/2⌋                          ▷ midpoint of A
5:     A_L ← MergeSort(A[1 : m])               ▷ recursively sort A[1 : m]
6:     A_R ← MergeSort(A[m+1 : |A|])           ▷ recursively sort A[m+1 : |A|]
7:     return Merge(A_L, A_R)                  ▷ merge the two sorted arrays
```

We describe the running time of merge sort (Algorithm 4) as follows:

[1] For merge sort (Algorithm 4), $a = b = 2$, but there are other divide-and-conquer algorithms in which $a \neq b$.

Algorithm 5 Merge

```
1:  function MERGE(A, B)
2:      i, j ← 1
3:      C ← []
4:      while i ≤ len(A) ∨ j ≤ len(B) do
5:          if i ≤ len(A) ∧ (j > len(B) ∨ A[i] < B[j]) then
6:              append A[i] to C
7:              i ← i + 1
8:          else
9:              append B[j] to C
10:             j ← j + 1
11:     return C
```

1. **Divide:** compute the middle of the array (Algorithm 4, Line 4), which takes constant time, $D(n) = \Theta(1)$
2. **Conquer:** recursively solve two subproblems (Algorithm 4, Line 5, 6), each of size $n/2$, contributes $2T(n/2)$
3. **Combine:** merge (Algorithm 5) the two sorted subarrays, which takes $\Theta(n)$ time, $C(n) = \Theta(n)$

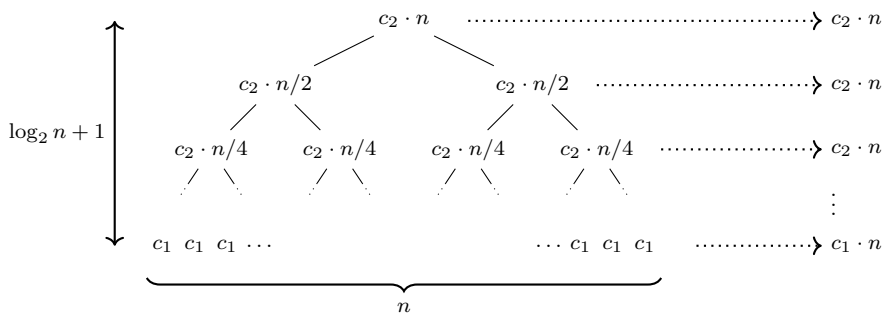
Using the so called 'master theorem':

$$T(n) = 2T(n/2) + \Theta(n) \xRightarrow{\text{master theorem}} T(n) = \Theta(n \log_2 n)$$

Intuitively we can also understand why that is the case without the master theorem. Assume for simplicity that n is an exact power of 2 and that the implicit base case is $n = 1$:

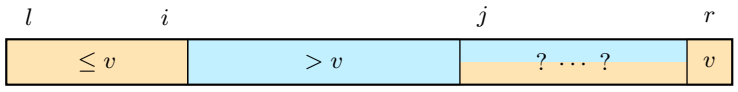
$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$$

where $c_1 > 0$ represents the time to solve the base case ($n = 1$) and $c_2 > 0$ is the time per element of the divide and combine steps.



$$\begin{aligned}
T(n) &= 2T(n/2) + c_2 n && = 2(2T((n/2)/2) + c_2(n/2)) + c_2 n \\
&= 2^2 T(n/2^2) + 2c_2 n && = 2^2(2T((n/2^2)/2) + c_2(n/2^2)) + 2c_2 n \\
&= 2^3 T(n/2^3) + 3c_2 n && = 2^3(2T((n/2^3)/2) + c_2(n/2^3)) + 3c_2 n \\
&\vdots \\
&= 2^{\log_2(n)} T(n/2^{\log_2(n)}) + \log_2(n) c_2 n \\
&= nT(1) + \log_2(n) c_2 n \\
&= c_1 n + c_2 n \log_2 n \\
&= \Theta(n \log n)
\end{aligned}$$

3.3 Quick Sort (with Lomuto Partitioning)



Lomuto partitioning (Algorithm 7) divides an array A into two subarrays $A[l : q-1]$ and $A[q+1 : r]$ such that all elements in $A[l : q-1]$ are less than or equal to $A[q]$ and all elements in $A[q+1 : r]$ are greater than $A[q]$.

Algorithm 6 Quick Sort

```
1:  function QUICKSORT(A, l, r)
2:      if l < r then
3:          q = PARTITION(A, l, r)
4:          QUICKSORT(A, l, q − 1)
5:          QUICKSORT(A, q + 1, r)
```

Algorithm 7 Lomuto Partitioning

```
1:  function PARTITION(A, l, r)
2:      v = A[r]                          ▷ pick last element as pivot
3:      i = l − 1                         ▷ highest index into the less-than-or-equal–partition
4:      for j = l, . . . , r do
5:          if A[j] ≤ v then
6:              i = i + 1
7:              swap A[i] and A[j]
8:      return i                          ▷ index of pivot
```

3.4 Heap Sort

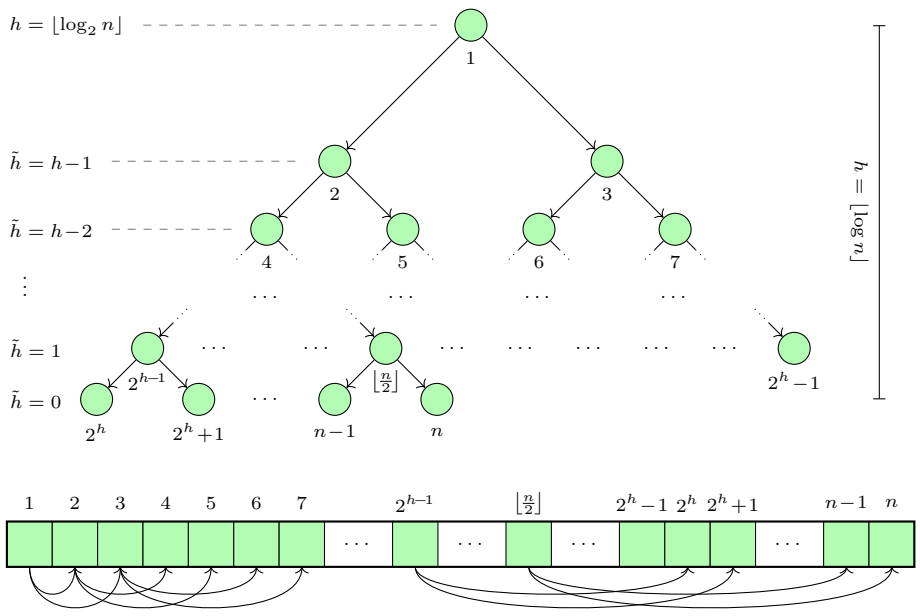
Definition 2 (Heap)**.** A *binary heap* is a nearly-complete binary tree stored in an array $A[1 : n]$ satisfying the *max-heap property*:

$$\forall i > 1 : \quad A[\text{Parent}(i)] \geq A[i]$$

The relationship between the indices of a binary heap is as follows:

$$\text{Parent}(i) = \lfloor i/2 \rfloor \qquad \text{Left}(i) = 2i \qquad \text{Right}(i) = 2i + 1$$

Furthermore, an n-element heap has height $h = \lfloor \log_2 n \rfloor$ and at most $\lceil n/2^{\tilde{h}+1} \rceil$ nodes at any given height $\tilde{h}$, where $\tilde{h}$ is defined as the longest path from the current node to a leaf node, measured in number of edges.



Algorithm 8 Max-Heapify

```
1:  function MAXHEAPIFY(A, i)
2:      l ← Left(i)
3:      r ← Right(i)
4:      m ← i                          ▷ index of largest element among {A[i], A[l], A[r]}
5:      if l ≤ A.heap-size ∧ A[l] > A[m] then
6:          m ← l
7:      if r ≤ A.heap-size ∧ A[r] > A[m] then
8:          m ← r
9:      if m ≠ i then
10:         swap A[i] and A[m]
11:         MAXHEAPIFY(A, m)
```

Algorithm 9 Build-Max-Heap

```
1:  function BUILDMAXHEAP(A)
2:      A.heap-size ← |A|
3:      for i = ⌊|A|/2⌋, . . . , 1 do      ▷ elements after ⌊|A|/2⌋ are leaves
4:          MAXHEAPIFY(A, i)
```

Algorithm 10 Heap Sort

```
1:  function HEAPSORT(A)
2:      BUILDMAXHEAP(A)
3:      for i = |A|, . . . , 2 do
4:          swap A[1] and A[i]
5:          A.heap-size ← A.heap-size − 1
6:          MAXHEAPIFY(A, 1)
```

Algorithm 10 sorts an array A in place by first building a max-heap from the input array and then repeatedly extracting the maximum element (the root of the heap) and placing it at the end of the array. The complexities of Algorithms 8, 9, 10 are:

$$T_{\text{MAXHEAPIFY}}(n) = \Theta(\log n) \qquad T_{\text{BUILDMAXHEAP}}(n) = \Theta(n)$$

$$T_{\text{HEAPSORT}}(n) = \Theta(n \log n)$$

The complexity of Algorithm 8 is determined by the height $\tilde{h}$ of the node to be heapified, which is given by $\tilde{h} = \lfloor \log i \rfloor$ for a node at index i.

Analyzing the complexity of Algorithm 9 is more involved. A simple upper bound on the running time is $O(n \lg n)$, since each call to MAXHEAPIFY costs $O(\log n)$ and BUILDMAXHEAP makes $O(n)$ such calls. This upper bound is correct but not asymptotically tight.

We can derive a tighter bound by recalling that the time for MAXHEAPIFY to run at a node i depends on the height $\tilde{h}$ of that node in the tree, and that the height of most nodes is small.

Calling MAXHEAPIFY on a node of height $\tilde{h}$ costs $c\tilde{h}$, and there are at most $\lceil n/2^{\tilde{h}+1} \rceil$ nodes at that height. We can sum over all heights, starting from the leaves (with height 0) up to the root (with height $\lfloor \log n \rfloor$);

$$T(n) = \sum_{\tilde{h}=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{\tilde{h}+1}} \right\rceil c\tilde{h} \leq cn \sum_{\tilde{h}=0}^{\lfloor \log n \rfloor} \frac{n}{2^{\tilde{h}}} \tilde{h} \leq cn \sum_{\tilde{h}=0}^{\infty} \frac{\tilde{h}}{2^{\tilde{h}}} \tag{4}$$

Recall the geometric series:

$$\sum_{\tilde{h}=0}^{\infty} x^{\tilde{h}} = \frac{1}{1-x}, \quad |x| < 1$$

Differentiating both sides with respect to x gives:

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{\tilde{h}=0}^{\infty} x^{\tilde{h}}\right) = \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{1-x}\right) \quad \Longrightarrow \quad \sum_{\tilde{h}=0}^{\infty} \tilde{h}\, x^{\tilde{h}-1} = \frac{1}{(1-x)^2}$$

Multiply through by x to shift the exponent back:

$$\sum_{\tilde{h}=0}^{\infty} \tilde{h}\, x^{\tilde{h}} = x \sum_{\tilde{h}=0}^{\infty} \tilde{h}\, x^{\tilde{h}-1} = \frac{x}{(1-x)^2}, \quad |x| < 1, \tag{5}$$

Setting $x = \frac{1}{2}$ in (5) and substituting into (4) gives

$$T(n) \le c\,n \sum_{\tilde{h}=0}^{\infty} \tilde{h} \left(\frac{1}{2}\right)^{\tilde{h}} = c\,n \cdot \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = O(n) \tag{6}$$

Thus BuildMaxHeap builds a max-heap from an array in $O(n)$ time.

For Algorithm 10, we have a similar situation, but different from BuildMaxHeap, where the majority of the calls to MaxHeapify are done on nodes at the bottom of the heap (where the height is small), in HeapSort, the calls to MaxHeapify are always done on the root of the heap, and the height is large for the majority of the calls.

After BuildMaxHeap has finished (line 2), the array A is a valid max-heap of size n. The HeapSort loop (lines 3-6) then performs exactly $n-1$ iterations. At the k^{th} iteration the heap contains $n-k+1$ elements, so the call to MaxHeapify costs $\Theta\big(\log(n-k+1)\big)$. The total time spent in the loop is therefore

$$\sum_{k=1}^{n-1} \Theta\left(\log(n-k+1)\right) \;=\; \Theta\left(\sum_{l=1}^{n-1} \log l\right) \;=\; \Theta\left(\log(n!)\right) \overset{\text{Ex. 3}}{=} \Theta(n \log n)$$

where the change of index $l = n-k+1$ rewrites the sum in increasing order.

3.5 k-Smallest Element

Algorithm 11 is a randomized "divide-and-conquer" method for finding the k-th smallest element in an unsorted array in expected $O(n)$ time. At each step it chooses a pivot uniformly at random, partitions the input into three subsets – those less than, equal to, and greater than the pivot – and then recurses only on the subset that must contain the desired element.

Algorithm 11 Quick Select

```
1:  function QuickSelect(A, k)        ▷ A is an unordered multiset ('bag') of elements
2:      v ← A[randint(1, |A|)]        ▷ pick a random pivot
3:      A_L, A_M, A_R ← {}_m          ▷ three empty multisets
4:      for all a ∈ A do
5:          if a < v then
6:              add a to A_L
7:          else if a = v then
8:              add a to A_M
9:          else
10:             add a to A_R
11:     if k ≤ |A_L| then
12:         return QuickSelect(A_L, k)
13:     else if k ≤ |A_L| + |A_M| then
14:         return v
15:     else
16:         return QuickSelect(A_R, k − (|A_L| + |A_M|))
```

Partitioning around the pivot takes $\Theta(n)$ time, and since the pivot is random the expected size of the recursive call is at most a constant fraction of n, yielding an overall expected running time of $O(n)$.

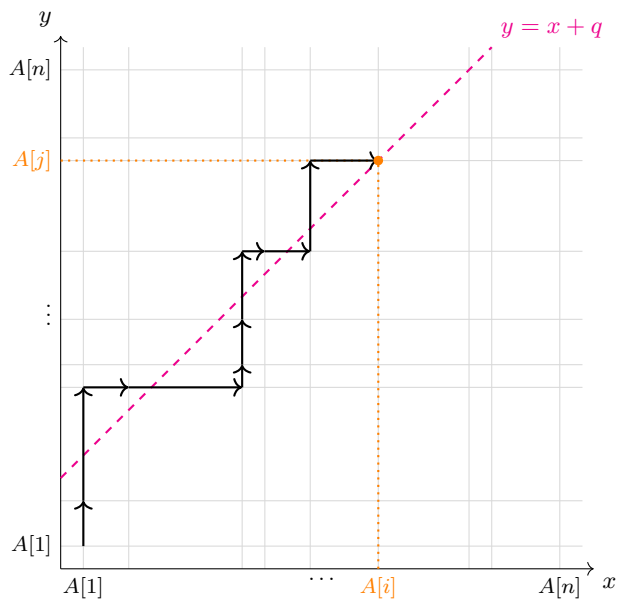
3.6 Overview of Sorting Algorithms

Algorithm	Time Complexity *worst*	*average*	*best*	in place?
InsertionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	✔
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✔
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✘
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✔
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✔

3.7 Application

When an array is sorted, many operations and queries (e.g. the one in Example 5) can be performed much more efficiently than in an unsorted array.

Example 5 (Caterpillar Method)**.** Checking if there a sorted array contains two elements $A[i]$ and $A[j]$ such that $A[i] + q = A[j]$, can be done in linear running time using the 🐛 method (sometimes also called *two-pointer* or *sliding-window* method).



Algorithm 12 Checking for two elements with difference q

```
1:  function Caterpillar(A, q)
2:      l, r ← 1
3:      while r ≤ n do
4:          if A[r] < A[l] + q then
5:              r ← r + 1
6:          else if A[r] > A[l] + q then
7:              l ← l + 1
8:          else
9:              return true
10:     return false
```

Algorithm 12 maintains two indices l and r, both starting at the left end of A (Line 2). At each step, it compares $A[r]$ and $A[l]$ and checks if the difference $A[r] - A[l]$ is less than, greater than, or equal to q:

- If $A[r] - A[l] < q$, move r one step to the right (to increase the difference).
- If $A[r] - A[l] > q$, move l one step to the right (to decrease the difference).
- If $A[r] - A[l] = q$, we found a valid pair and stop.

◀

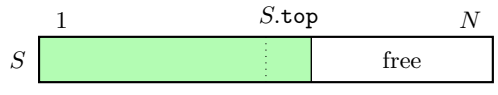
4 Data Structures

Operations on a dynamic set can be grouped into two categories, *queries*, which return information about the set, and *modifying operations*, which change the set:

- Search(S, k)
  query that, given a set S and a key value k, returns a pointer x to an element in S such that x.key $= k$, or nil if no such element belongs to S
- Insert(S, x)
  modifying operation that adds the element pointed to by x to the set S (we usually assume that any attributes in element x needed by the set implementation have already been initialized)
- Delete(S, x)
  modifying operation that, given a pointer x to an element in the set S, removes x from S (note that this operation takes a pointer to an element x, not a key value!)
- Minimum(S) and Maximum(S)
  queries on a totally ordered set S that return a pointer to the element of S with the smallest (for Minimum) or largest (for Maximum) key
- Successor(S, x)
  query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or nil if x is the maximum element
- Predecessor(S, x)
  query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or nil if x is the minimum element

4.1 Stacks

Definition 3 (Stack)**.** A *stack* is a LIFO container that supports constant-time insertion and deletion at one end. ✍



Interface (Algorithm 13):

- StackEmpty(S): returns `true` iff the stack S contains no elements.
- Push(S, x): places element x on the top of the stack S.
- Pop(S): removes and returns the top element of the stack S.

The stack has attributes S.`top`, indexing the most recently inserted element, and S.`length`, equaling the size N of the array.

Algorithm 13 Stack Operations (array-based)

```
1: function StackEmpty(S)
2:     return (S.top = 0)
3: function Push(S, x)
4:     if S.top = S.length then
5:         error "overflow"
6:     S.top ← S.top + 1
7:     S[S.top] ← x
8: function Pop(S)
9:     if StackEmpty(S) then
10:        error "underflow"
11:    S.top ← S.top − 1
12:    return S[S.top + 1]
```

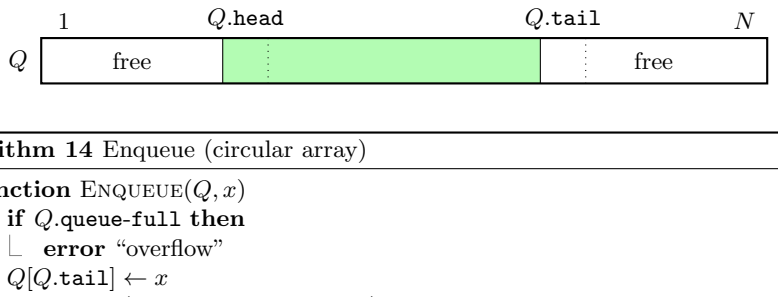
4.2 Queues

Definition 4 (Queue)**.** A *queue* is a FIFO container with constant-time insertion at the tail and deletion at the head. ✍

Interface (Algorithms 14 and 15):

- Enqueue(Q, x): insert x at the tail of Q.
- Dequeue(Q): remove and return the head element of Q.

The classic fixed-length *circular-array* implementation keeps two indices Q.head and Q.tail (Q.head points to the first element, Q.tail to the first free slot).



Algorithm 14 Enqueue (circular array)

```
1: function Enqueue(Q, x)
2:     if Q.queue-full then
3:         error "overflow"
4:     Q[Q.tail] ← x
5:     Q.tail ← (Q.tail mod Q.length) + 1
6:     Q.queue-empty ← false
7:     if Q.tail = Q.head then
8:         Q.queue-full ← true
```

Algorithm 15 Dequeue (circular array)

```
1: function Dequeue(Q)
2:     if Q.queue-empty then
3:         error "underflow"
4:     x ← Q[Q.head]
5:     Q.head ← (Q.head mod Q.length) + 1
6:     Q.queue-full ← false
7:     if Q.tail = Q.head then
8:         Q.queue-empty ← true
9:     return x
```

4.3 Linked Lists

Definition 5 (Doubly-Linked List)**.** A *doubly-linked list* L consists of nodes x that store a key x.key and two links x.prev, x.next. A special sentinel node L.nil simplifies boundary cases because the list is empty iff L.nil.next $= L$.nil. ✍

Algorithm 16 Doubly-Linked List Operations (with sentinel)

```
1: function ListInit(L)
2:     L.nil.prev ← L.nil
3:     L.nil.next ← L.nil
4: function ListInsert(L, x)                ▷ insert at front
5:     x.next ← L.nil.next
6:     L.nil.next.prev ← x
7:     L.nil.next ← x
8:     x.prev ← L.nil
9: function ListDelete(x)
10:    x.prev.next ← x.next
11:    x.next.prev ← x.prev
12: function ListSearch(L, k)
13:    x ← L.nil.next
14:    while x ≠ L.nil ∧ x.key ≠ k do
15:        x ← x.next
16:    return x
```

Algorithm 16 shows the basic operations on a doubly-linked list. ListInsert and ListDelete take $O(1)$ time, whereas ListSearch takes $\Theta(n)$ time in the worst case, with n the current length of L.

4.4 Dictionaries

Definition 6 (Dictionary)**.** A *dictionary* is an abstract data structure that represents a set of elements (or keys). It is a *dynamic set* that supports the following operations:

- Insert: insert element into the set
- Delete: delete element from the set
- Search: test memebership of an element in the set ✍

4.5 Direct-Address Tables

Definition 7 (Direct-Address Table)**.** A *direct-address table* implements a dictionary. Suppose the key universe is the set $U = \{1, \ldots, M\}$. A *direct-address table* is an array $T[1 : M]$ where slot $T[k]$ stores a Boolean indicating membership of key k in the represented set. ✍

Algorithm 17 Direct-Address Table Operations

```
1: function DirectAddressInsert(T, k)
2:     T[k] ← true
3: function DirectAddressDelete(T, k)
4:     T[k] ← false
5: function DirectAddressSearch(T, k)
6:     return T[k]
```

All direct-address table operations (Algorithm 17) cost $O(1)$ time, but the table occupies $\Theta(|U|)$ space, which is prohibitive when the universe is large and the actual set is sparse. i.e., direct-address tables usually waste a lot of space.

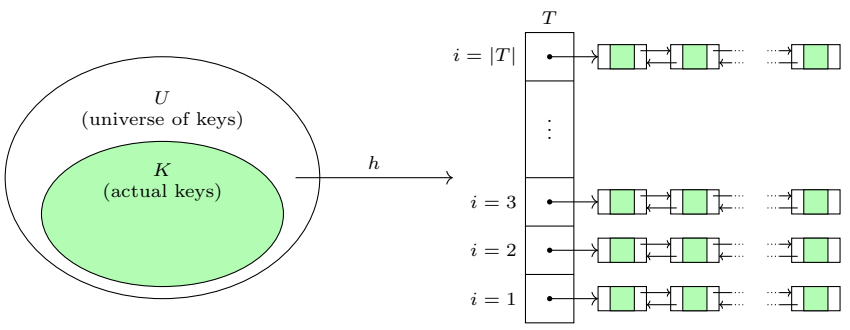
4.6 Hash Tables

To reduce the space overhead we use a smaller table T with $|T| \ll |U|$ and map each key $k \in U$ to a position in T using a *hash function* $h : U \to \{1, \ldots, |T|\}$.

Definition 8 (Load Factor)**.** For a table of size $|T|$ that currently stores n keys, the *load factor* is $\alpha = \frac{n}{|T|}$. ✍

4.6.1 Chaining

Each slot $T[i]$ stores a linked list of keys that hash to i.



Algorithm 18 Chained Hash Operations

```
1: function ChainedHashInsert(T, k)
2:     ListInsert(T[h(k)], k)
3: function ChainedHashSearch(T, k)
4:     return ListSearch(T[h(k)], k)
5: function ChainedHashDelete(T, k)
6:     x ← ChainedHashSearch(T, k)
7:     if x ≠ nil then
8:         ListDelete(x)
```

We assume *uniform hashing* i.e.

$$\mathrm{P}[h(k) = i] = \frac{1}{|T|} \quad \forall i \in \{1, \ldots, |T|\}$$

So, given n distinct keys, the expected length n_i of the linked list at position i is

$$\mathrm{E}[n_i] = \frac{n}{|T|} = \alpha$$

If we further assume $h(k)$ can be computed in $O(1)$, the expected running time of ChainedHashSearch is

$$\boxed{\Theta(1 + \alpha)}$$

4.6.2 Open Addressing

Instead of using linked lists, keys are stored directly in the array. On a collision we *probe* other slots in T using a permutation $h(k, 1), \ldots, h(k, |T|)$. So $h(k, i)$ is a function of both k and i, where i is the probe number.

$h(k, \cdot)$ must be a *permutation* of $\{1, \ldots, |T|\}$, i.e., $h(k, 1), \ldots, h(k, |T|)$ must cover all slots in T exactly once.

We assume *independent uniform permutation hashing*: the probe sequence of each key is equally likely to be any of the $|T|!$ permutations of $\{1, \ldots, |T|\}$. Independent uniform permutation hashing generalizes the notion of independent uniform hashing introduced earlier to a hash function that produces not just a single slot number, but a whole probe sequence. True independent uniform permutation hashing is difficult to

implement, however, and in practice suitable approximations (such as double hashing, Example 6) are used.

Neither double hashing nor its special case, linear probing, meets the assumption of independent uniform permutation hashing. Double hashing cannot generate more than $|T|^2$ different probe sequences. Nonetheless, double hashing has a large number of possible probe sequences and seems to give good results. Linear probing is even more restricted, capable of generating only $|T|$ different probe sequences.

Example 6 (Double Hashing)**.** Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. Double hashing uses a hash function of the form

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod |T| \tag{7}$$

where h_1 and h_2 are two different hash functions. The second hash function $h_2(k)$ must be *relatively prime* to $|T|$ (i.e., $\gcd(h_2(k), |T|) = 1$) to ensure that all slots in T are probed. A convenient way to achieve this is to let $|T|$ be an exact power of 2 and to design $h_2(k)$ so that it always produces an odd number. Another way is to let $|T|$ be prime and to design $h_2(k)$ so that it always returns a positive integer less than $|T|$. In (7) we can also set $h_2(k) = 1$ for all k, in which case we get *linear probing*, a special case of double hashing. ◀

Algorithm 19 Open-Address Hash Insert (generic probing)

```
1: function HashInsert(T, k)
2:     for i = 1, ..., |T| do
3:         j ← h(k, i)
4:         if T[j] = NIL then
5:             T[j] ← k
6:             return j
7:     error "overflow"
```

To analyze the time complexity of Algorithm 19, we assume independent uniform permutation hashing for the hash function. We also assume that at least one slot is empty, i.e., $\alpha < 1$. Because deleting from an open-address hash table does not really free up a slot, we assume as well that no deletions occur.

If we denote by X the number of probes performed until an empty slot is found, then on each probe we hit an occupied slot with probability $\alpha = n/|T|$ and an empty slot with probability $1 - \alpha$. Hence

$$\mathrm{P}[X = i] = \alpha^{i-1}(1-\alpha), \quad i = 1, 2, \ldots$$

so that X is geometrically distributed with success probability $1 - \alpha$. Hence

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} i\alpha^{i-1}(1-\alpha) = \frac{1}{1-\alpha}$$

Thus an insertion (or an unsuccessful search) requires on average $\frac{1}{1-\alpha}$ probes; this grows rapidly as the load factor α approaches 1.

4.7 Binary Search Trees

Definition 9 (Binary Search Tree)**.** A *binary search tree* implements a dynamic set. It stores keys from a totally ordered domain in nodes linked by *left* and *right* child pointers such that for every node x

$$y \in \text{left-subtree}(x) \Leftrightarrow y.\texttt{key} \leq x.\texttt{key}, \quad z \in \text{right-subtree}(x) \Leftrightarrow z.\texttt{key} \geq x.\texttt{key}$$ ✍

Example 7 (Lower Bound)**.** Let t be the root of a binary search tree that represents a set S of numbers. The size of the tree is $|S| = n$. The height of the tree is h. Algorithm 20 returns the node containing the least element $y \in S$ such that $x \leq y$, or NIL if no such element exists.

Algorithm 20 Lower Bound for Binary Search Tree

```
1: function LowerBound(t, x)
2:     if t = NIL then                                   ▷ base case
3:         return NIL
4:     if t.key < x then
5:         return LowerBound(t.right, x)
6:     else
7:         y ← LowerBound(t.left, x)
8:         if y ≠ NIL then
9:             return y
10:        else
11:            return t
```

The complexity is $O(h)$, where h is the height of the tree. ◀

4.7.1 Traversals

Algorithm 21 Inorder Tree Walk (recursive)

```
1: function TreeWalk(x)
2:     if x ≠ NIL then
3:         TreeWalk(x.left)
4:         print x.key
5:         TreeWalk(x.right)
```

Algorithm 21 shows the InOrderTreeWalk algorithm. Three other variants can be obtained from Algorithm 21 by swapping the order of the recursive calls and the print statement (lines 3, 4, and 5). For PreOrderTreeWalk, we swap lines 3 and 4. For PostOrderTreeWalk, we swap lines 4 and 5. For ReverseOrderTreeWalk, we swap lines 3 and 5.

The general recurrence is

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

and all four variants run in $\Theta(n)$ time. This can be proven using the *substitution method*. Can we do better? No, because the length of the output is $\Theta(n)$.

4.7.2 Basic Queries

Algorithm 22 Searching a Binary Search Tree for a Key

```
1: function TreeSearch(x, k)
2:     if x = NIL ∨ k = x.key then
3:         return x
4:     if k < x.key then
5:         return TreeSearch(x.left, k)
6:     else
7:         return TreeSearch(x.right, k)
```

Algorithm 23 Minimum of a Binary Search Tree

```
1: function TreeMinimum(x)
2:     while x.left ≠ NIL do
3:         x ← x.left
4:     return x
```

To find the maximum, replace all occurences of `left` with `right` in Algorithm 23.

Definition 10 (Successor)**.** The *successor* of a node x is the minimum of the right subtree of x if it exists, otherwise it is the lowest ancestor a of x such that x falls in the left subtree of a. ✍

Algorithm 24 Successor of a Node in a Binary Search Tree

```
1: function TreeSuccessor(x)
2:     if x.right ≠ NIL then
3:         return TreeMinimum(x.right)          ▷ leftmost node in right subtree
4:     while x.parent ≠ NIL ∧ x.parent.right = x do
5:         x ← x.parent
6:     return x.parent      ▷ lowest ancestor of x whose left child is an ancestor of x
```

To find the predecessor, replace all occurences of `right` with `left` and use TreeMaximum instead of TreeMinimum in Algorithm 24.

4.7.3 Updates

Algorithm 25 Inserting a Node into a Binary Search Tree

```
1: function TreeInsert(T, z)
2:     y, x ← NIL, 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:     z.parent ← y
10:    if y = NIL then
11:        T.root ← z
12:    else if z.key < y.key then
13:        y.left ← z
14:    else
15:        y.right ← z
```

Deletion distinguishes three cases:

- z is a leaf node, i.e. it has no children:
  - simply remove the node z
- z has one child:
  - remove z
  - connect its child to its parent
- z has two children:
  - find the successor y of z (since z has two children, y is guaranteed to be the minimum of the right subtree of z and thus have at most one child)
  - copy the key of y into z
  - delete y, which is a leaf or has one right child, i.e. connect the child of y to the parent of y

Insertion, search and deletion operations have complexity $\Theta(h)$ where h is the height of the tree. In the average case, the height is $O(\log n)$ (i.e. with a random insertion order). In some particular cases, the height can be $O(n)$ (i.e. with ordered sequence).

The problem is that the 'worst case' is not that uncommon. One way to avoid this is to instead of inserting $A = [a_1, a_2, a_3, \ldots, a_n]$ in order, insert a random permutation of A. The problem is that A is usually not known in advance. It is the application that calls the insertion procedure. But we can also obtain a random permutation of A by using a randomized insertion algorithm (see 4.7.4).

Algorithm 26 Deleting a Node from a Binary Search Tree

```
1:  function TreeDelete(T, z)
2:      ▷ z has two children: find successor, copy its key, then delete successor ◁
3:      if z.left ≠ NIL ∧ z.right ≠ NIL then
4:          s ← TreeMinimum(z.right)
5:          z.key ← s.key          ▷ replace the key in z with the one from the successor
6:          return TreeDelete(T, s)
7:      ▷ z has at most one child: pick it (could be NIL) ◁
8:      if z.left ≠ NIL then
9:          c ← z.left
10:     else
11:         c ← z.right
12:     if c ≠ NIL then                    ▷ if child exists, update its parent pointer
13:         c.parent ← z.parent
14:     if z.parent = NIL then             ▷ if z was the root, make child the new root
15:         T.root ← c
16:     else                   ▷ otherwise, bypass z by connecting its child to its parent
17:         if z = z.parent.left then
18:             z.parent.left ← c
19:         else
20:             z.parent.right ← c
21:     return T
```

4.7.4 Randomized Insertion

In order to avoid the linear-height worst case one can insert into a tree using Algorithm 27. The idea behind the function TreeRandomizedInsert is as follows. We insert a new node z into the tree t as the new root of t with probability $1/(t.\texttt{size}+1)$. If z is not inserted as the new root, we recursively insert it into the appropriate subtree of t. The additional attribute $t.\texttt{size}$ represents the number of nodes in the subtree rooted at t.

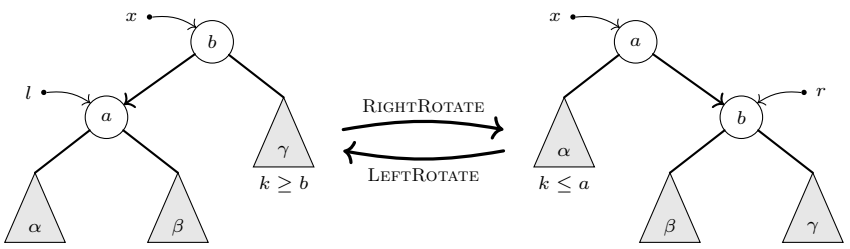
Algorithm 27 Randomized Insertion into a Binary Search Tree

```
1:  function TreeRandomizedInsert(t, z)
2:      if t = NIL then
3:          return z
4:      r ← randint(1, t.size + 1)
5:      if r = 1 then        ▷ P(r = 1) = P(z is inserted as the new root of t) = 1/(t.size+1)
6:          z.size ← t.size + 1
7:          return TreeRootInsert(t, z)                          ▷ see 4.7.6
8:      if z.key < t.key then
9:          t.left ← TreeRandomizedInsert(t.left, z)
10:     else
11:         t.right ← TreeRandomizedInsert(t.right, z)
12:     t.size ← t.size + 1
13:     return t
```

With Algorithm 27 every insertion order is equally likely; the expected height of the tree is $O(\log n)$ and all operations run in expected $O(\log n)$ time.

4.7.5 Rotations

Definition 11 (Rotation)**.** A *rotation* is a local restructuring of a BST that exchanges the relative position of a node x and one of its children while preserving the in-order sequence of keys. Rotations do *not* change the in-order ordering of the keys and they are the basic tool used by self-balancing trees. ✍



Algorithm 28 Rotations in a Binary Search Tree

```
1:  function RightRotate(x)
2:      l ← x.left
3:      x.left ← l.right
4:      l.right ← x
5:      return l
6:  function LeftRotate(x)
7:      r ← x.right
8:      x.right ← r.left
9:      r.left ← x
10:     return r
```

Rotations are very useful operations. For example, we can use it to perform

4.7.6 Root Insertion

General strategies to deal with complexity in the worst case:

- *Randomization*: can make the scenario $h = n$ highly unlikely
- *Amortized Maintenance*: relatively expensive but 'amortized' operations
- *Self-Balancing*: e.g. Red-Black trees (see 4.8)

Algorithm 29 Root Insertion into a Binary Search Tree

```
1:  function TreeRootInsert(x, z)
2:      if x = NIL then
3:          return z
4:      if z.key < x.key then
5:          x.left ← TreeRootInsert(x.left, z)
6:          return RightRotate(x)
7:      else
8:          x.right ← TreeRootInsert(x.right, z)
9:          return LeftRotate(x)
```

4.8 Red-Black Trees

A *red-black tree* is a binary search tree with one extra bit of storage per node: its `color`, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Definition 12 (Red-Black Tree)**.** A red-black tree is a binary search tree that satisfies the following *red-black properties*:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. ✍

Definition 13 (Black-Height)**.** The number of black nodes on any simple path from, but not including, a node x down to, and including, a leaf is called the *black-height* of the node x:

$$\mathrm{bh}(x) \coloneqq \#\,\text{black nodes on path from (excluding) } x \text{ to (including) leafs}$$

By property 5, the notion of black-height is well defined, since all descending paths have the same number of black nodes. ✍

Lemma 3 (Height of a Red-Black Tree)**.** The height $h(x)$ of a red-black tree with $n = \mathrm{size}(x)$ internal nodes is at most $2\log(n+1)$. ◁

Proof. First, we prove (by induction) that the subtree rooted at any node x contains at least

$$2^{\mathrm{bh}(x)} - 1 \tag{8}$$

internal nodes.

(i) base case: x is a leaf, so $\mathrm{size}(x) = 0$ and $\mathrm{bh}(x) = 0$, fulfilling (8) ✔

(ii) induction step: consider x, y_1, y_2 such that $x = y_1.\texttt{parent} = y_2.\texttt{parent}$

$$\mathrm{size}(x) = \mathrm{size}(y_1) + \mathrm{size}(y_2) + 1 \geq (2^{\mathrm{bh}(y_1)} - 1) + (2^{\mathrm{bh}(y_2)} - 1) + 1$$

By rule 5, both children must have the same black-height. Let $\mathrm{bh}(y) = \mathrm{bh}(y_1) = \mathrm{bh}(y_2)$. Then

$$\mathrm{size}(x) \geq 2 \cdot (2^{\mathrm{bh}(y)} - 1) + 1 = 2^{\mathrm{bh}(y)+1} - 1$$

The black-height of a child y differs at most by one from the black-height of the parent x, i.e. $\mathrm{bh}(y) \in \{\mathrm{bh}(x), \mathrm{bh}(x) - 1\}$. In both cases, we have $\mathrm{size}(x) \geq 2^{\mathrm{bh}(x)} - 1$, fulfilling (8) ✔

By property 4, the black-height of a node x is at least half the height of the tree, i.e. $\mathrm{bh}(x) \geq \frac{h(x)}{2}$. Therefore,

$$n = \mathrm{size}(x) \geq 2^{\mathrm{bh}(x)} - 1 \geq 2^{\frac{h(x)}{2}} - 1$$

which can be rearranged to

$$h(x) \leq 2\log(n+1)$$ □

5 Graphs

5.1 Representations of Graphs

There are two standard ways to represent a graph: as a collection of adjacency lists or as an adjacency matrix. Because the adjacency-list representation provides a compact way to represent *sparse* graphs (those for which $|E| \ll |V|^2$), it is usually the method of choice. The adjacency-matrix representation might be preferred when the graph is *dense* (i.e., $|E| \approx |V|^2$), or you need to be able to tell quickly wether there is an edge connecting two given vertices.

The space required for the adjacency-list representation is $\Theta(|V| + |E|)$, and finding each edge in the graph also takes $\Theta(|V| + |E|)$ time, since each of the $|V|$ adjacency lists must be examined.

The space required for the adjacency-matrix representation is $\Theta(|V|^2)$, and finding each edge in the graph takes $\Theta(|V|^2)$ time, since the entire adjacency matrix must be examined. For an undirected graph, the adjacency matrix is symmetric.

The complexity for different operations is summarized in the following table.

Operation	Representation	
	Adjacency List	Adjacency Matrix
accessing vertex u	$O(1)$ optimal	$O(1)$ optimal
iteration through V	$\Theta(\|V\|)$ optimal	$\Theta(\|V\|)$ optimal
iteration through E	$\Theta(\|V\| + \|E\|)$ okay (not optimal)	$\Theta(\|V\|^2)$ possibly very bad
checking $(u, v) \in E$	$O(\|V\|)$ bad	$O(1)$ optimal
space complexity	$\Theta(\|V\| + \|E\|)$ optimal	$\Theta(\|V\|^2)$ possibly very bad

5.2 Breadth-First Search

Breadth-First Search is one of the simplest but also a fundamental algorithm, as it is the archetype of many important algorithms.

Algorithm 30 Breadth-First Search

```
1:  function BFS(G, s)                                   ▷ s is the source
2:      for all u ∈ V \ {s} do
3:          u.color, u.δ, u.π ← WHITE, ∞, NIL
4:      s.color, s.δ, s.π ← GRAY, 0, NIL
5:      Q ← ∅                          ▷ initialize empty queue for vertices to visit
6:      ENQUEUE(Q, s)
7:      while Q ≠ ∅ do
8:          u ← DEQUEUE(Q)
9:          for all v ∈ Γ(u) do
10:             if v.color = WHITE then
11:                 v.color ← GRAY
12:                 v.δ ← u.δ + 1
13:                 v.π ← u
14:                 ENQUEUE(Q, v)
15:         u.color ← BLACK
```

We enqueue a vertex only if is WHITE, and we immediately color it GRAY; thus, we enqueue every vertex at most once. So the (dequeue) while loop at Line 7 executes $O(|V|)$ times. For each vertex u, the inner loop at Line 9 executes $\Theta(|\Gamma(u)|)$ times, for a total of $O(|E|)$ steps. Thus, the total running time is $O(|V| + |E|)$.

The minimum number of edges in any path from s to v is called the shortest-path distance from s to v and is denoted by $\delta(s, v)$. If there is no path from s to v, then $\delta(s, v) = \infty$. We define the predecessor subgraph as $G_\pi = (V_\pi, E_\pi)$ where

$$V_\pi = \{v \in V \mid v.\pi \neq \text{NIL}\} \cup \{s\}$$

and

$$E_\pi = \{(v.\pi, v) \mid v \in V_\pi \setminus \{s\}\}$$

The predecessor subgraph G_π is a *breadth-first tree* if V_π consists of all vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G.

Breadth-First Search constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree rooted at s. Upon termination, $v.\delta = \delta(s, v)$ for all $v \in V$. For any vertex v reachable from s, the simple path in the breadth-first tree G_π from s to v corresponds to a shortest path (that is, a path containing the smallest number of edges) from s to v in the original graph G.

Assuming that Breadth-First Search has computed a breadth-first tree G_π, we can print the shortest path from s to v using (the recursive) Algorithm 31.

Algorithm 31 Print Shortest Path

```
1: function PRINTPATH(G, s, v)
2:     if v = s then
3:         PRINT(s)
4:     else if v.π = NIL then
5:         PRINT("No path exists")
6:     else
7:         PRINTPATH(G, s, v.π)
8:         PRINT(v)
```

5.3 Depth-First Search

As its name implies, Depth-First Search searches "deeper" in the graph whenever possible. Depth-First Search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search *backtracks* to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then Depth-First Search selects one of them as a new source, repeating the search from that source. This process is repeated until every vertex has been discovered.[2]

Since Depth-First Search always explores all vertices, we define the predecessor subgraph as $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(v.\pi, v) \mid v \in V \wedge v.\pi \neq \text{NIL}\}$$

[2] Although Breadth-First Search could proceed from multiple sources and Depth-First Search could be limited to one source source, the approach here reflects how they are typically used. Breadth-First Search usually serves to find shortest-path distances and the associated predecessor subgraph fro a given source, while Depth-First Search is often a subroutine in another algorithm.

Algorithm 32 Depth-First Search

```
1:  function DFS(G)
2:      for all u ∈ V do
3:          u.color, u.π ← WHITE, NIL
4:      T ← 0                          ▷ global time variable, for timestamps
5:      for all u ∈ V do
6:          if u.color = WHITE then
7:              DFS-VISIT(u)                          ▷ new tree in forest
8:  function DFS-VISIT(u)
9:      u.color ← GRAY                          ▷ u has just been discovered
10:     T ← T + 1
11:     u.d ← T
12:     for all v ∈ Γ(u) do                    ▷ explore each edge leaving u
13:         if v.color = WHITE then
14:             v.π ← u
15:             DFS-VISIT(v)          ▷ recursively visit v if it undiscovered
16:     u.color ← BLACK                          ▷ blacken u; it is finished
17:     T ← T + 1
18:     u.f ← T
```

It is a *depth-first forest*, comprising several *depth-first trees*.

Each vertex is initially white, is grayed when it is discovered in the search (Line 9), and is blackened when it is finished, that is, when its neighborhood has been examined completely (Line 16).

Each vertex u has two timestamps: the first, $u.d$, records when u is first discovered (and grayed), and the second, $u.f$, records when the search finished examining u's neighborhood (and blackens u). For each vertex $u \in V$, we have

$$u.d, u.f \in \{1, \ldots, 2|V|\} \quad \text{and} \quad u.d < u.f$$

since there is one discovery and one finishing event for each of the $|V|$ vertices. u is WHITE before $u.d$, GRAY between $u.d$ and $u.f$, and BLACK after $u.f$.

Upon every call of DFS-VISIT(u) in Line 7, a new depth-first tree is created, rooted at u. In each call DFS-VISIT(u), u is initially WHITE. Lines 12–15 examine each vertex v adjacent to u and recursively visit v if it is WHITE.

The result depends on the order in which Line 5 examines the vertices and upon the order in which Line 12 visits the neighbors of u. Usually, these different visiation orders tend not to cause problems, because many applications can use any of those.

We call DFS-VISIT(u) exactly once (either in Line 7 or recursively in Line 15) for each vertex u, because we call it only if $u.\texttt{color} = \text{WHITE}$, but then we immediately set $u.\texttt{color} = \text{GRAY}$ in Line 9. The loop in Lines 12–15 executes $\Theta(|\Gamma(u)|)$ times. So, the total running time is $\Theta(|V| + |E|)$.

Properties of the Depth-First Forest G_π:

- v is a descendant of $u \iff v$ is discovered during the time in which u is gray
- discovery and finish times have *parenthesis structure*: if in DFS-VISIT we printed '(u' when we discovered u and 'u)' when we finished u, then the printed expression would be well formed in the sense that the parentheses are properly nested
- for any two vertices u_1 and u_2, exactly one of the following conditions holds
  - $[u_1.d, u_1.f] \cap [u_2.d, u_2.f] = \emptyset \iff$ neither is a descendant of the other
  - $[u_1.d, u_1.f] \subset [u_2.d, u_2.f] \iff u_1$ is a descendant of u_2
  - $[u_1.d, u_1.f] \supset [u_2.d, u_2.f] \iff u_2$ is a descendant of u_1
- v is a descendant of $u \iff$ at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices

Classification of Edges in E:

- *tree edges* are edges $(u, v) \in E_\pi$ in the depth-first forest G_π. (u, v) is a tree edge if v was first discovered by exploring the edge (u, v) in Line 12.
- *back edges* are edges $(u, v) \notin E_\pi$ connecting u to an ancestor v in the depth-first forest G_π.
- *forward edges* are edges $(u, v) \notin E_\pi$ connecting u to a descendant v in the depth-first forest G_π.
- *cross edges* are edges $(u, v) \notin E_\pi$ are all other edges.

Lemma 4. A directed graph is acyclic if and only if a Depth-First Search yields no back edges. ◁

Theorem 5. In a undirected graph, a Depth-First Search yields only tree edges and back edges. ◁

When during an Depth-First Search an edge (u, v) is first explored, the color of v yields information about the edge:

- if v is WHITE, then (u, v) is a tree edge
- if v is GRAY, then (u, v) is a back edge
- if v is BLACK, then (u, v) is a
  - forward edge if $u.d < v.d$
  - cross edge if $u.d > v.d$

Observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations.

5.3.1 Topological Sort

A *topological sort* of a directed acyclic graph ("dag") $G = (V, E)$ is a linear ordering of the vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

Algorithm 33 Topological Sort

```
1: function TopologicalSort(G)
2:     call DFS(G) to compute the finish time v.f for each v ∈ V
3:     as each vertex v is finished, insert it onto the front of a linked list L
4:     return L
```

5.3.2 Strongly Connected Components

Definition 14 (Strongly Connected Component)**.** A *strongly connected component* of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$, that is, u and v are reachable from each other. ✍

Algorithm 34 uses the transpose of G, $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$. That is, E^{T} consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^{T} is $\Theta(V + E)$. The graphs G and G^{T} have exactly the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^{T}.

The linear-time (i.e., $\Theta(V + E)$-time) Algorithm 34 computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^{T}.

Definition 15 (Component Graph)**.** Suppose that G has strongly connected components $C_1, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, \ldots, v_k\}$, and it contains one vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{\mathrm{SCC}}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$.

Looked at another way, if we contract all edges whose incident vertices are within the same strongly connected component of G so that only a single vertex remains, the resulting raph is G^{SCC}. ✍

Algorithm 34 Strongly Connected Components

```
1: function StronglyConnectedComponents(G)
2:     call DFS(G) to compute the finish time v.f for each v ∈ V
3:     create the transpose graph G^T
4:     call DFS(G^T), but in Line 5, consider the vertices in order of decreasing u.f
5:     return the depth-first trees of G_π formed in Line 4
```

The Strongly Connected Components, defined in Definition 14 and the Component Graph $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, defined in Definition 15 have the following properties:

If C_i and C_j are two distinct Strongly Connected Components in a directed graph $G = (V, E)$,...

1. ...and $u_i, v_j \in C_i$ and $u_j, v_j \in C_j$, and G contains a path $u_i \rightsquigarrow u_j$, then G cannot also contain a path $v_j \rightsquigarrow v_i$, i.e. the Component Graph G^{SCC} is acyclic.
2. ...and there is an edge $(u, v) \in E$ with $u \in C_j$ and $v \in C_i$, then $C_j.f > C_i.f$.
3. ...and suppose that $C_i.f > C_j.f$, then E^{T} contains no edge (v, u) such that $u \in C_j$ and $v \in C_i$.

Property 1 implies that the Component Graph G^{SCC} can be topologically sorted (see Section 5.3.1). And in fact, Algorithm 34 visits the vertices of the Component Graph in topologically sorted order, by considering vertices in the second depthfirst search (Line 4) in decreasing order of the finish times that were computed in the first depth-first search (Line 2).

In Properties 2 and 3, the finish time of a Strongly Connected Component, $C.f$, refers to the maximal finish time of any vertex in C as computed by Line 2 of Algorithm 34, i.e.,

$$C.f := \max_{v \in C} (v.f)$$

Property 3 provides the key idea behind Algorithm 34: When the Depth-First Search of G^{T} in Line 4 of Algorithm 34 visits any Strongly Connected Component, any edges out of that component must be to components that the search has already visited. Each depth-first tree produced by Line 4, therefore, corresponds to exactly one Strongly Connected Component of G.

Another way to see this is to consider the component graph $(G^{\mathrm{T}})^{\mathrm{SCC}}$ of G^{T}. If we map each strongly connected component visited in the second depth-first search to a vertex of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, the second depth-first search visits vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in the reverse of a topologically sorted order.

5.4 Minimum Spanning Tree

Given a graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ we want to find an acyclic subset $T \subseteq E$ such that T touches all vertices in V and

$$w(T) = \sum_{e \in T} w(e)$$

that is, the total weight of the tree T is minimal. This T is called a *minimum spanning tree* (MST) of G.

Conceptually, Kruskal's algorithm (1956) is similar to the Strongly Connected Components algorithm, while Prim's algorithm (1957) resembles Dijkstra's shortest-paths algorithm. Both run in $O(|E| \log |V|)$ time.

The general idea is to build T by adding one edge $e \in E$ at a time, such that e is the lightest edge that does not create a cycle. Both Kruskal and Prim are greedy algorithms (see 6.1), that is, in each step they make the choice that seems best at the moment. Even though such a strategy does not generally yield an optimal solution, for the MST problem one can prove that certain greedy strategies yield an optimal solution.

Algorithm 35 Generic Minimum Spanning Tree

```
1: function Generic-MST(G, w)
2:     A ← ∅                          ▷ invariant: A is part of a minimum spanning tree
3:     while A is not a spanning tree do
4:         find a safe edge e = (u, v)        ▷ a safe edge maintains the invariant
5:         A ← A ∪ {e}
```

The strategy is given in Algorithm 35. The invariant used is that A is a subset of a minimum spanning tree and a *safe edge* is an edge that maintains this invariant.

A *cut* of a graph $G = (V, E)$ is a partition of V into two disjoint sets S and $V \setminus S$. An edge $e \in E$ *crosses* the cut if one of its endpoints is in S and the other is in $V \setminus S$. A cut *respects* a set of edges A if no edges in A cross the cut.

Theorem 6. Let $A \subseteq E$ be included in a minimum spanning tree and $(S, V \setminus S)$ be a cut that respects A. Then a minimum-weight edge e crossing the cut $(S, V \setminus S)$ is a safe edge for A. ◁

Theorem 6 provides a rule for recognizing safe edges and is the key to why Algorithm 35 works. At any point in the execution, the graph $G_A = (V, A)$ is a forest and each of the connected components of G_A is a tree. Any safe edge e for A connects two distinct components of G_A, since $A \cup \{e\}$ must be acyclic.

The loop in Lines 3–5 of Algorithm 35 executes $|V| - 1$ times, because it finds one of the $|V| - 1$ edges of a a minimum spanning tree in each iteration. Initially, when $A = \emptyset$, there are $|V|$ trees in G_A and each iteration reduces that number by one, until the forest contains only a single tree, upon which the method terminates.

Kruskal and Prim both use the following corollary of Theorem 6.

Corollary 7. Let $A \subseteq E$ be included in a minimum spanning tree and let $C = (V_C, E_C)$ be a connected component (tree) of $G_A = (V, A)$. Then a minimum-weight edge e connecting C to some other component in G_A is a safe edge for A. ◁

Kruskal and Prim use a different, specific rule to determine a safe edge in Line 4 of Algorithm 35. In Kruskal's algorithm, A is a forest whose vertices are all those of the given graph and the safe edge added is always a lowest-weight edge that connects two distinct components. In Prim's algorithm, A is a single tree and the safe edge added is always a lowest-weight edge connecting the tree to a vertex not in the tree.

For Kruskal, we need the *disjoint-set data structure*:

- MakeSet(x) creates a set containing only the element x
- Find(x) returns the representative of the set containing x
- Union(x, y) joins the sets containing x and y into a single set

Algorithm 36 Kruskal

```
1: function Kruskal(G, w : E → ℝ)
2:     A ← ∅
3:     for all v ∈ V do
4:         MakeSet(v)
5:     sort E in non-decreasing order by w
6:     for all (u, v) ∈ E in non-decreasing w-order do
7:         if Find(u) ≠ Find(v) then
8:             A ← A ∪ {(u, v)}
9:             Union(u, v)
```

Algorithm 37 Prim

```
1: function Prim(G, w : E → ℝ, r)
2:     T ← (∅, ∅)                                  ▷ minimal spanning tree
3:     for all v ∈ V(G) do
4:         v.weight ← ∞                            ▷ best known cost of connecting v to T
5:         v.π ← NIL ▷ u ∈ T such that (u, v) is the least-cost edge connecting v to T
6:     r.weight ← 0                                ▷ r is the root of the spanning tree
7:     while V(T) ≠ V(G) do                        ▷ while T does not span all vertices
8:         find u ∉ V(T) such that u.weight is minimal
9:         T ← T ∪ {u}                             ▷ add u to the spanning tree T
10:        for all v ∈ Γ(u) \ V(T) do
11:            if w(u, v) < v.weight then
12:                v.weight ← w(u, v)
13:                v.π ← u
```

5.5 Single-Source Shortest Paths

Dijkstra's algorithm is a greedy, label-setting method for nonnegative edge weights. At each step it finalizes the closest unreached vertex and relaxes its outgoing edges, yielding a time complexity of $O((|V| + |E|) \log |V|)$ with a binary-heap implementation.

Bellman-Ford algorithm is a dynamic-programming, label-correcting method that supports arbitrary (i.e. including negative) weights and detects negative-weight cycles. It relaxes all edges in up to $|V| - 1$ passes for $O(|V| \times |E|)$ time and uses one extra pass to check for cycles.

The Bellman-Ford equation:

$$\delta(s, v) = \min_{(u,v) \in E} (\delta(s, u) + w(u, v)) \tag{9}$$

Algorithm 38 Dijkstra

```
1: function Dijkstra(G, w : E → ℝ≥0, s)
2:     N ← ∅                          ▷ nodes of G whose least-cost path from s is definitely known
3:     for all v ∈ V(G) do
4:         v.δ ← ∞                                  ▷ best known cost from s to v
5:         v.π ← NIL                ▷ node preceding v on the least-cost path from s
6:     s.δ ← 0                                      ▷ s is the source
7:     while N ≠ V(G) do  ▷ while we do not know the least-cost path to all nodes
8:         find u ∉ N such that u.δ is minimal
9:         N ← N ∪ {u}                              ▷ add u to the nodes N
10:        for all v ∈ Γ(u) \ N do
11:            if u.δ + w(u, v) < v.δ then
12:                v.δ ← u.δ + w(u, v)
13:                v.π ← u
```

Algorithm 39 Bellman-Ford

```
1: function BellmanFord(G, w : E → ℝ, s)
2:     for all v ∈ V(G) do
3:         v.δ ← ∞
4:         v.π ← NIL
5:     s.δ ← 0
6:     for i = 1 to |V(G)| − 1 do
7:         for all (u, v) ∈ E(G) do
8:             if u.δ + w(u, v) < v.δ then
9:                 v.δ ← u.δ + w(u, v)                ▷ applying Equation (9)
10:                v.π ← u
11:    for all (u, v) ∈ E(G) do
12:        if u.δ + w(u, v) < v.δ then
13:            return false                  ▷ negative-weight cycle detected
14:    return true
```

6 Design Techniques

6.1 Greedy Algorithms

Greedy algorithms construct a solution by always choosing the option that looks best at the moment. That is, they make a locally optimal choice in the hope that this choice leads to a globally optimal solution.

At every step, we consider only what is best best in the current problem, not considering the results of the subproblems.

The key ingredients of a problem for a greedy strategy to work:

1. *Greedy-Choice*: One can always arrive at a globally optimal solution by making a locally optimal choice.
2. *Optimal Substructure*: An optimal solution to the problem contains within it optimal solutions to subproblems.

It is natural to prove property 2 by induction: if the solution to the subproblem is optimal, then combining the greedy choice with that solution yields an optimal solution.

The proof pattern when designing a greedy algorithm is:

(i) *Cast* the problem as one where we make a *greedy choice* and are left with a *subproblem*.

(ii) *Prove* that (at least) one optimal solution (not necessarily always the same one) contains the greedy choice (Property 1).

(iii) *Prove* that the remaining subproblem is such that combining the greedy choice with the optimal solution of the subproblem gives an optimal solution to the original problem.

Remark 1 (Designing Greedy Algorithms)**.**

- Inventing a greedy algorithm is easy (easy to come up with greedy choices)
- Proving it optimal may be hard (requires deep understanding of the structure of the problem) ◀

Remark 2 (Greedy vs. Dynamic Programming)**.** Greedy algorithms have many similarities to dynamic programming. In particular, problems for which dynamic programming works also need to have optimal substructure (Property 2).

One major difference is that instead of first finding optimal solutions to subproblems and then making an informed choice, greedy algorithms first make a greedy choice the choice that looks best at the time and then solve a resulting subproblem, without bothering to solve all possible related smaller subproblems. ◀

6.2 Dynamic Programming

Dynamic programming is a method for solving (typically optimization) problems by decomposing them into (slightly smaller) subproblems that share subsubproblems. To make such a recursive approach efficient, the optimal solutions to those subproblems are stored in an array/table of appropriate size and dimension, so that each susubbproblem is solved just once. Instead of solving the problem top-down (recursively, with memoization) it is also possible (and often slightly more efficient due to the overhead of recursion) to solve the problem bottom-up (iteratively filling the table, starting with the base cases).

The key ingredients of a problem for a dynamic programming method to work:

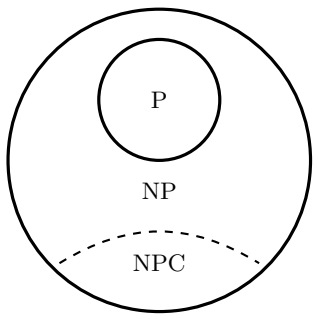
1. *Optimal Substructure*: An optimal solution to the problem contains within it optimal solutions to subproblems.
2. *Overlapping Subproblems*: The subproblems share subsubproblems. A recursive solution therefore revisits the same subsubproblem repeatedly.

To characterize the space of subproblems, a good rule of thumb is to try to keep it as simple as possible.

Remark 3 (Dynamic Programming vs. Divide and Conquer)**.** Both decompose a problem into subproblems. But in divide-and-conquer the subproblems are disjoint, that is, they do not share subsubproblems. The subproblems in divide-and-conquer are also typically much smaller than the original problem, while in dynamic programming the subproblems are typically only slightly smaller than the original problem (e.g. reducing $L(j)$ to $L(j-1)$) ◀

7 Complexity Theory

Complexity theory is about classifying computational problems by their computational difficulty.



P contains all problems that can be *solved* in polynomial time.

NP contains all problems for which one can *verify* a given solution in polynomial time, whereas the solution may not be found in polynomial time.