CS 245

Notes

Author
Steven Cao
University of Waterloo
Fall 2023

Contents

1 Asymptotic Analysis

1.1 Algorithm design

Definition 1.1 (Problem)**.** A **problem** is a description of input and require output.

For example, given an input array, rearrange elements in non-decreasing order.

Definition 1.2 (Problem instance)**.** A **problem instance** is one possible input of a specified problem.

A problem instance for sorting is $I = [5, 2, 1, 8, 2]$.

Definition 1.3 (Size of problem instance)**.** The **size of problem instance** I, Size(I), is a positive integer measuring size of problem instance I wrt some attribute of input.

If the input is an array, the instance size can be the size of the array.

Definition 1.4 (Algorithm)**.** An **algorithm** is a finite step-by-step process for carrying out a series of computations, given an arbitrary instance I.

Definition 1.5 (Solving a problem)**.** Algorithm A **solves** problem Π if for every instance I of Pi, A computes a valid output for I in finite time.

Definition 1.6 (Program)**.** A **program** is an **implementation** of an algorithm using a specified langauge.

To go from problem Π to a program that solves it, we need to take these steps:

1. **Algorithm design**: design algorithms that solve Π
2. **Algorithm analysis**: determine **correctness** and **efficiency** of algorithms
3. **Implementation**: implement the best algorithm (correct and efficient)

1.2 Pseudocode

Definition 1.7 (Pseudocode)**. Pseudocode** is a method of communicating algorithms to a human.
Pseudocode uses natural language descriptions and omits some unnecessary technical details (e.g. initialization).

Use indentation to depict scope instead of braces.

Example 1.1. Write pseudocode for insertion sort.

Algorithm 1 Insertion sort

```
Require: A, n
  for i ← 1 to n − 1 do
      j ← i
      while j > 0 and A[j] < A[j − 1] do
          swap A[j] and A[j − 1]
          j ← j − 1
```

1.3 Measuring efficiency

There are two types of efficiency.

Definition 1.8 (Running time)**. Running time** is the amount of time it takes for the program to run.

Definition 1.9 (Auxiliary space)**. Auxiliary space** is the amount of additional memory the program requires.

Efficiency is a function of input. We can depict this as $T(I)$, where I is a problem instance.

To derive $T(I)$, we group all instances of size n into set I_n. Then, we *measure* over I_n. We can take either the average case, the best case, or the worst case. In any case, $T(I)$ usually depends on instance size and instance composition.

One way to measure running time is to use experimental studies. That is, run the program implementing the algorithm with inputs of varying size and composition. We can plot the results in a graph wrt size.

However, this method requires implementing the algorithm and depends on software/hardware factors. To determine running time more concretely, we need a theoretical framework for algorithm analysis.

We first need an **idealized memory model**.

Definition 1.10 (RAM model)**.** In the **Random Access Memory (RAM) model**:

- There is a set of **memory cells**, each storing a data item.
- Any **primitive operation** (memory access, variable assignment, arithmetic operators, method return, etc.) takes the same constant time c.

We only need the pseudocode of the algorithm to analyze the algorithm under the RAM model. To find time efficiency, count the number of primitive operations. To find space efficiency, count the number of memory cells in use.

Example 1.2. Determine the running time of the following algorithm.

Algorithm 2 ArraySum

Require: A, n
  $sum \leftarrow A[0]$
  for $i \leftarrow 1$ to $n-1$ **do**
    $sum \leftarrow sum + A[i]$

Count the number of primitive operations for each statement and find the sum (depending on n). Focus on the **growth rate** of n, ignoring constant factors and lower order terms.

1.4 Asymptotic analysis

Asymptotic analysis (order notation) gives us tools to formally depict growth rate.

Definition 1.11 (O-notation)**.** $f(n) \in O(g(n))$ if there exist constants $c > 0$, $n_0 \geq 0$ such that

$$|f(n)| \leq c\,|g(n)|$$

for all $n \geq n_0$. That is, $f(n)$ is asymptotically (eventually) bounded from above by $g(n)$.

Example 1.3. Given $f(n) = 75n + 500$, $g(n) = 5n^2$, show that $f(n) \in O(g(n))$.

First set $f(n) = g(n)$ and solve the equation. We get $n = 82$. Take $c = 1$, $n_0 = 82$.

Alternatively, notice for all $n \geq 1$, we have

$$75n \leq 75n^2$$

$$500 \leq 500n^2$$

Therefore,

$$75n + 500 \leq 75n^2 + 500n^2 = 575n^2$$

Take $c = 575$, $n_0 = 1$. This gives a tighter bound than before.

What if we want an asymptotic notation that guarantees a tight bound? We also need a lower bound.

Definition 1.12 (Ω-notation)**.** $f(n) \in \Omega(g(n))$ if there exists constants $c > 0$, $n_0 \geq 0$ such that

$$|f(n)| \geq c\,|g(n)|$$

for all $n \geq n_0$. That is, $f(n)$ is asymptotically bounded from below by $g(n)$.

Definition 1.13 (Θ-notation)**.** $f(n) \in \Theta(g(n))$ if there exists constants $c_1, c_2 > 0$, $n_0 \geq 0$ such that

$$c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$$

for all $n \geq n_0$. That is, $f(n)$, $g(n)$ have equal growth rates.

Theorem 1.1 (Tight bound)**.** $f(n) \in \Theta(g(n))$ iff $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Ideally, Θ should be used to depict growth rate of algorithms. This is sometimes hard, so big O is used.

Growth rate	Name	Effect when problem size doubles
$\Theta(1)$	constant	$T(n) = c \implies T(2n) = c$
$\Theta(\log n)$	logarithmic	$T(n) = c \log n \implies T(2n) = T(n) + c$
$\Theta(n)$	linear	$T(n) = cn \implies T(2n) = 2T(n)$
$\Theta(n \log n)$	linearithmic	$T(n) = cn \log n \implies T(2n) = 2T(n) + 2cn$
$\Theta(n \log^k n)$	quasi-linear	
$\Theta(n^2)$	quadratic	$T(n) = cn^2 \implies T(2n) = 4T(n)$
$\Theta(n^3)$	cubic	$T(n) = cn^3 \implies T(2n) = 8T(n)$
$\Theta(2^n)$	exponential	$T(n) = c2^n \implies T(2n) = c^{-1}T^2(n)$

Table 1: Common growth rates, in increasing order

There are also **strict** asymptotic bounds.

Definition 1.14 (o-notation)**.** $f(n) \in o(g(n))$ if for all $c > 0$, there exists $n_0 \geq 0$ such that

$$|f(n)| \leq c |g(n)|$$

for all $n \geq n_0$.

Definition 1.15 (ω-notation)**.** $f(n) \in \omega(g(n))$ if for all $c > 0$, there exists $n_0 \geq 0$ such that

$$|f(n)| \geq c |g(n)|$$

for all $n \geq n_0$.

Theorem 1.2 (Relationships between order notations)**.**

$$\begin{aligned}
f(n) \in \Theta(g(n)) &\iff g(n) \in \Theta(f(n)) \\
f(n) \in O(g(n)) &\iff g(n) \in \Omega(f(n)) \\
f(n) \in o(g(n)) &\iff g(n) \in \omega(f(n)) \\
f(n) \in o(g(n)) &\implies f(n) \in O(g(n)) \\
f(n) \in o(g(n)) &\implies f(n) \notin \Omega(g(n)) \\
f(n) \in \omega(g(n)) &\implies f(n) \in \Omega(g(n)) \\
f(n) \in \omega(g(n)) &\implies f(n) \notin O(g(n))
\end{aligned}$$

We can perform algebra on order notation.

Theorem 1.3 (Identity of order notation)**.**

$$f(n) \in \Theta(f(n))$$

Order notation is **transitive**, similar to comparison operators.

Theorem 1.4 (Transitivity of order notation)**.**

$$\begin{aligned}
f(n) \in O(g(n)) \land g(n) \in O(h(n)) &\implies f(n) \in O(h(n)) \\
f(n) \in \Omega(g(n)) \land g(n) \in \Omega(h(n)) &\implies f(n) \in \Omega(h(n)) \\
f(n) \in O(g(n)) \land g(n) \in o(h(n)) &\implies f(n) \in o(h(n))
\end{aligned}$$

When adding multiple functions, take the max of all functions when analyzing.

Theorem 1.5 (Maximum rules of order notation)**.** Suppose $f(n) > 0$, $g(n) > 0$ for all $n \geq n_0$. Then,

$$\begin{aligned}
f(n) + g(n) &\in \Omega(\max\{f(n), g(n)\}) \\
f(n) + g(n) &\in O(\max\{f(n), g(n)\})
\end{aligned}$$

Theorem 1.6 (Limit theorem for order notation)**.** Suppose for all $n \geq n_0$, $f(n) > 0$, $g(n) > 0$, $L = \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$. Then,

$$f(n) \in \begin{cases} o(g(n)) & L = 0 \\ \Theta(g(n)) & 0 < L < \infty \\ \omega(g(n)) & L = \infty \end{cases}$$

1.5 Analysis of algorithms

The goal is to use asymptotic analysis to simplify run-time analysis.

We can identify primitive operations, which take constant time.

The complexity of a loop can be expressed as the sum of the complexities of each iteration. Nested summation may be needed for nested loops.

There are two general strategies:

1. Use Θ-bounds throughout the analysis and obtain the Θ-bound for the complexity of the algorithm.
2. Prove O-bound and Ω-bound separately.

Definition 1.16 (Worst-case time complexity)**.**

$$T_{\text{worst}}(n) = \max_{I \in I_n}(T(I))$$

For insertion sort, the worst case running time is $\Theta(n^2)$.

Definition 1.17 (Best-case time complexity)**.**

$$T_{\text{best}}(n) = \min I \in I_n(T(I))$$

For insertion sort, the best case running time is $\Theta(n)$.

Definition 1.18 (Best-case time complexity)**.**

$$T_{\text{avg}}(n) = \frac{1}{|I_n|} \sum_{I \in I_n} T(I)$$

Do not compare algorithms based on solely O-notation. Θ-notation should be used to compare algorithms.

Algorithms with the same runtime efficiency may not have the same speed. Further, algorithms with worse runtime efficiency may be faster than algorithms with better time efficiency for small inputs.

To find the space used by an algorithm, count total number of memory cells accessed wrt n. We are mostly interested in **auxiliary space**, the space used in addition to input.

1.6 Analysis of recursive algorithms

Consider merge sort, a recursive algorithm.

Algorithm 3 MergeSort

```
A, n, l ← 0, r ← n − 1, S ← ∅
if S is empty then
    initialize as array S[0..n − 1]
if r ≤ l then
    return
else
    m ← ⌊(r + l)/2⌋
    MergeSort(A, n, l, m, S)
    MergeSort(A, n, m + 1, r, S)
    Merge(A, l, m, r, S)
```

For each step, merge sort divides the array into two halves, sorts both recursively, then merges the two arrays. Merge is $\Theta(n)$ for merging n elements.

This produces the recurrence relation:

$$T(n) = \begin{cases} T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn & n > 1 \\ c & n = 1 \end{cases}$$

First consider the **sloppy recurrence** with floors and ceilings removed:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & n > 1 \\ c & n = 1 \end{cases}$$

This resolves to $T(n) \in \Theta(n \log n)$.

Recursion	Resolves to	Example
$T(n) \leq T(n/2) + O(1)$	$T(n) \in O(\log n)$	Binary search
$T(n) \leq 2T(n/2) + O(n)$	$T(n) \in O(n \log n)$	Merge sort
$T(n) \leq 2T(n/2) + O(\log n)$	$T(n) \in O(n)$	Heapify
$T(n) \leq cT(n-1) + O(1), c < 1$	$T(n) \in O(1)$	Average-case analysis
$T(n) \leq 2T(n/4) + O(1)$	$T(n) \in O(\sqrt{n})$	Range search
$T(n) \leq T(\sqrt{n}) + O(\sqrt{n})$	$T(n) \in O(\sqrt{n})$	Interpolation search
$T(n) \leq T(\sqrt{n}) + O(1)$	$T(n) \in O(\log \log n)$	Interpolation search

Table 2: Common recurrence relations

1.7 Best, worst, average case

Algorithms can have two different running times on two instances with equal size.

Definition 1.19 (Worst-case time complexity)**.**

$$T_{\text{worst}}(n) = \max_{I \in I_n} \{T(I)\}$$

where I_n is the set of all instances with size n.

Definition 1.20 (Best-case time complexity)**.**

$$T_{\text{best}}(n) = \min I \in I_n \{T(I)\}$$

For *insertion-sort*, the best instance is a sorted array, so best case time complexity is $\Theta(n)$. The worst case time complexity is $\Theta(n^2)$.

Definition 1.21 (Average-case time complexity)**.**

$$T_{\text{avg}}(n) = \frac{1}{|I_n|} \sum_{I \in I_n} \{T(I)\}$$

2 Priority Queue

2.1 Abstract Data Types

Definition 2.1 (ADT)**.** An **abstract data type (ADT)** is a description of **information** and a collection of **operations** on the information.

ADTs merely provides an interface to access the information. ADTs do not describe how the information is stored.

There can be multiple ways to implement the same ADT. Information is realized as **data structures**, and operations are realized as **algorithms**.

Here are some common ADTs:

Definition 2.2 (Stack)**. Stack** is an ADT for a collection of items in LIFO order.
Operations:
- *push*: insert an item.
- *pop*: remove and return the most recently inserted item.

Optional operations:
- *size*
- *isEmpty*
- *top*

Possible implementations:
- Array
- Linked List

Stacks are used to store information in procedure/function calls.

Definition 2.3 (Queue)**. Queue** is an ADT for a collections of items in FIFO order.
Operations:
- *enqueue*: insert an item.
- *dequeue*: remove and return the least recently inserted item.

Optional operations:
- *size*
- *isEmpty*
- *peek*

Possible implementations:
- Circular array
- Linked list

2.2 Priority queue ADT

Definition 2.4 (Priority queue)**. Priority queue** is a collection of items in some order (priority).
Operations:
- *insert*: insert an item.
- *deleteMax*: remove and return the item with the highest priority.

We can use a priority queue to perform sorting.

This has time complexity $O(initialization + n \cdot insert + n \cdot deleteMax)$

Some ways to implement priority queues:

1. Unsorted arrays
   - *insert*: $O(1)$
   - *deleteMax*: $O(n)$

- *pq-sort*: $O(n^2)$

2. Sorted arrays
   - *insert*: $O(n)$
   - *deleteMax*: $O(1)$
   - *pq-sort*: $O(n^2)$

2.3 Binary heap

Definition 2.5 (Binary tree)**.** A **binary tree** is either:
- Empty
- Node, left subtree, right subtree

Theorem 2.1 (Binary tree minimum height)**.** Consider a binary tree with n nodes and height h. Then,

$$h \in \Omega(\log n)$$

The lower bound arises from a full binary tree, in which level $i < h$ has 2^i nodes, and level h has between 1 to 2^h nodes.

Definition 2.6 (Heap)**.** A **max-oriented binary heap** is a binary tree with the following properties:
1. Structural property:
   - All levels of the heap are full except possibly level h.
   - The last level is left-justified.
2. Heap-order property:
   - For any node i, key[parent(i)] $\geq$ key[i].

Heaps are ideal for implementing priority queues.

Theorem 2.2 (Heap height)**.** Given a heap with n nodes and height h,

$$h \in \Theta(\log n)$$

This comes from that heaps are "almost full" binary trees.

Since heaps are left-justified, we can store them in arrays in the following way:

- Store root in $A[0]$.
- Store subsequent levels one after the other, each level from left to right.

Then,

- Root is 0.

- Last node is $n-1$.
- Left child is $2i+1$.
- Right child is $2i+2$.
- Parent is $\lfloor (i-1)/2 \rfloor$

To perform insertion, first place the new key at the first free leaf, or the end of the array. Heap property may have been violated, so we need to perform a **fix-up**.

The time complexity is $O(h) = O(\log n)$

Heap sort is done by modifying *pq-sort* for heaps so that it is in-place (build heap on input array).

We can use *heapify* to build a heap from an array of known size.

This has complexity $\Theta(n)$. A heap can be build in linear time if we know all items in advance.

3 Sorting, average-case, randomization

For some algorithms, the best case and worst case differ too significantly, so average-case time complexity is most useful. Average-case analysis paints a more accurate picture of how an algorithm performs, given that all instances are equally likely.

3.1 Sorting permutations

Definition 3.1 (Sorting permutation)**.** The **sorting permutation** of array π is the array containing the index of each element when the array is sorted.

Arrays with the same relative order have the same sorting permutations.

There are $n!$ sorting permutations of an array with distinct numbers of size n. Thus, we can define average cost through permutations:

$$T_{\text{avg}}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

where Π_n is the set of all sorting permutations of size n.

We can group together instances with the same runtime to find the average runtime.

3.2 Randomized algorithms

Instances in the real world are often not random. In some cases the worst-case behavior can be consistently triggered, such as reversed arrays for insertion sort.

Randomization improves runtime in practice when instances are not equally likely. It makes sense to randomize algorithms which have better average-case than worst-case runtime.

Definition 3.2 (Randomized algorithm)**.** A **randomized algorithm** is an algorithm that relies on some random numbers in addition to input.
The runtime is dependent on both input I and random numbers R.

Definition 3.3 (Expected runtime)**.**

$$T_{\text{exp}}(I) = E[T(I,R)] = \sum_{\forall R} T(I,R)P(R)$$

Definition 3.4 (Worst-case expected runtime)**.**

$$T_{\text{w.exp}}(n) = \max_{I \in I_n} T_{\text{exp}}(I)$$

3.3 QuickSelect

The **selection problem** is as follows: given array A of n numbers and $k \in [0, n-1]$, find element that would be at index k if A is sorted.

A special case is median finding, which is equivalent to select($\lfloor n/2 \rfloor$).

Selection problem can be done in $O(n)$ average time using QuickSelect.

QuickSelect requires two subroutines:

1. ChoosePivot: select $p \in [0, n-1]$
2. Partition: rearrange A such that everything before $A[p]$ is less than $A[p]$, and everything after $A[p]$ is greater than $A[p]$.

ChoosePivot can be done by selecting the rightmost element of the array $(n-1)$.

Algorithm 4 StandardPartition

Require: A, p
  $S, E, L \leftarrow []$
  $v \leftarrow A[p]$
  for $x \in A$ **do**
    if $x < v$ **then** S.append(x)
    else if $x > v$ **then** L.append(x)
    else E.append(x)
  $A[0 \dots |S|-1] \leftarrow S$
  $A[|S| \dots |S|+|E|-1] \leftarrow E$
  $A[|S|+|E| \dots n-1] \leftarrow L$

Partitioning can also be done in-place, by continuously swapping outermost wrongly-positioned pairs.

Algorithm 5 HoarePartition

Require: A, p
  $A[n-1] \leftrightarrow A[p]$
  $i \leftarrow -1$
  $j \leftarrow n-1$
  $v \leftarrow A[n-1]$
  loop
    while $A[i] < v$ **do** $i \leftarrow i+1$
    while $j \geq i \wedge A[j] > v$ **do** $j \leftarrow j-1$
    if $i \geq j$ **then** break
    else $A[i] \leftrightarrow A[j]$
  $A[n-1] \leftrightarrow A[i]$

Algorithm 6 QuickSelect

Require: A, k
  $p \leftarrow \text{ChoosePivot}(A)$
  $i \leftarrow \text{Partition}(A, p)$
  if $i = k$ **then return** $A[i]$
  else if $i > k$ **then return** $\text{QuickSelect}(A[0 \ldots i-1], k)$
  else if $i < k$ **then return** $\text{QuickSelect}(A[i+1 \ldots n-1], k)$

QuickSelect has $\Theta(n)$ best case and $\Theta(n^2)$ worst case. The best case happens when the first chosen pivot has pivot index k. The worst case happens when the pivot value is always wrongly chosen.

We can show that the average runtime of QuickSelect is $\Theta(n)$.

To avoid bad instances, we can implementation a randomized version of QuickSelect:

- Shuffle the input A at the start
- Randomly select the pivot

Either method leads to $\Theta(n)$ expected time.

3.4 QuickSort

QuickSort is another algorithm developed by Hoare. It uses pivot and partitioning to sort an array.

Worse case is $\Theta(n^2)$, best case is $\Theta(n \log n)$.

By randomizing the pivot, the expected runtime is $\Theta(n \log n)$.

QuickSort can be improved in several ways:

1. Switch to insertion sort when n is small. This reduces worst case space to $\Theta(\log n)$.

Algorithm 7 QuickSort

Require: A, n
  if $n \leq 1$ **then return**
  $p \leftarrow \text{ChoosePivot}(A)$
  $i \leftarrow \text{Partition}(A, p)$
  $\text{QuickSort}(A[0 \dots i-1])$
  $\text{QuickSort}(A[i+1 \dots n-1])$

2. Switch to HeapSort when recursion depth is high. This reduces worst case time to $\Theta(n \log n)$.

3.5 Comparison-based sorting

Definition 3.5 (Comparison model)**.** In the **comparison model**, data can be accessed in two ways:
1. Compare two elements
2. Move elements around

The lower bound for comparison-based sorting is $\Theta(n \log n)$.

Using a **decision tree**, we can describe all decisions taken during the execution of an algorithm. The height of the tree (longest path to a leaf) represent the worst case of the algorithm.

Theorem 3.1. Comparison-based sorting algorithms require $\Omega(n \log n)$ comparisons.

Proof. Let SortAlg be a comparison-based sorting algorithm. Then, SortAlg has a decision tree where each leaf is a sorting permutation. This means given that the input array is of length n, the tree has $n!$ leaves.

Since a binary tree of height h has up to 2^h leaves, we get that

$$\begin{aligned}
2^h &\geq n! \\
h &\geq \log(n!) \\
&= \sum_{i=1}^{n} \log(i) \\
&= \sum_{i=\frac{n}{2}+1}^{n} \log(i) \\
&\geq \sum_{i=\frac{n}{2}+1}^{n} \frac{n}{2} \\
&= \frac{n}{2} \log \frac{n}{2} \\
&\in \Omega(n \log n)
\end{aligned}$$

□

3.6 Non-comparison-based sorting

Array of integers, strings can be sorted using non-comparison-based sorting.

Bucket sort is an example of a non-comparison-based sorting algorithm.

Algorithm 8 BucketSort

Require: A

```
B ← empty array of lists of size R
for e ∈ A do
    B[bucket of e].push(e)
i ← 0
for b ∈ B do
    for e ∈ b do
        A[i] ← e
        i ← i + 1
```

Bucket sort is stable and has $\Theta(n + R)$ time and $\Theta(n + R)$ space, where R is the number of buckets (radix).

We can use multiple stages of bucket sort to sort numbers by going to each digit.

MSD radix sort has $\Theta(mnR)$ time and $\Theta(m + n + R)$ space, where m is the number of digits.

LSD radix sort takes advantage of stability of bucket sort. It has $\Theta(m(n + R))$ time and $\Theta(n + R)$ space.

Algorithm 9 MSDRadixSort

Require: $A, l \leftarrow 0, r \leftarrow n-1, d \leftarrow$ leading digit index
  if $l \geq r$ **then return**
  BucketSortDigit($A[l \dots r], d$)
  if there are no more digits **then return**
  $l' \leftarrow l$
  while $l' < r$ **do**
    $r' \leftarrow$ max value such that $A[l' \dots r']$ has the same d-th digit
    MSDRadixSort($A, l', r', d+1$)
    $l' \leftarrow r' + 1$

Algorithm 10 LSDRadixSort

Require: A
  for $d \in$ [digit index from least significant to most significant] **do**
    BucketSort(A, d)

4 Dictionaries

4.1 Binary search tree

Definition 4.1 (Binary search tree)**.** A **binary search tree** (BST) is a binary tree where each node is either:

1. Empty
2. Contains a KVP and two subtrees

A BST node has the property that every key to its left is smaller and every key to its right is bigger.

Operations search, insert, delete all have $\Theta(h)$ time, where h is the height of the BST. In the worst case, $h = n$ which leads to $\Theta(n)$ time for these operations. In the best case, the tree is balanced which leads to $\Theta(\log n)$ time.

Definition 4.2 (Height of tree node)**.** The height of node v is the height of the subtree rooted at v.

4.2 AVL tree

Definition 4.3. AVL tree An **AVL tree** is a BST with height-balance property: for any node v, the height of its left and right subtree differ by at most 1. The height or the difference in balance is stored for each node.

Theorem 4.1. An AVL tree of n nodes has $\Theta(\log n)$ height.

Insertion and deletion may cause an AVL tree to become unbalanced. In that case, we can rotate the tree to restore balance.

Algorithm 11 RotateRight

Require: z

$y \leftarrow z$.left
z.left $\leftarrow y$.right
y.right $\leftarrow z$
return y

Algorithm 12 RotateLeft

Require: z

$y \leftarrow z$.right
z.right $\leftarrow y$.left
y.left $\leftarrow z$
return y

Algorithm 13 Restructure

Require: z

$y \leftarrow$ child of z with greatest height
$x \leftarrow$ child of y with greatest height
if $x < y < z$ **then**
    return RotateRight(z)
else if $y < x < z$ **then**
    z.left $\leftarrow$ RotateLeft(y)
    return RotateRight(z)
else if $z < x < y$ **then**
    z.right $\leftarrow$ RotateRight(y)
    return RotateLeft(z)
else if $z < y < x$ **then**
    return RotateLeft(z)

When an AVL tree is modified, call Restructure on the inserted/replaced node.

4.3 Skip list

Definition 4.4 (Skip list)**.** A **skip list** is a hierarchy of linked lists. The lists are collected in layers. Each layer starts with a left sentinel ($-\infty$) and ends with a right sentienl (∞). A layer is half as dense as the layer below it. The bottom (base) layer is a linked list that contains every KVP. Each node may have a node (1/2 probability) in its layer above with the same key pointing to the node below and the next node in that layer.

To search through a skip list, we can take advantage of the sparser upper layers to skip over large chunks of list. Then, traverse through lower layers to narrow down the search.

Algorithm 14 getPredecessors

Require: k

```
p ← root
P ← empty stack of nodes
while p.below exists do
    p ← p.below
    while p.after.k < k do
        p ← p.after
    P.push(p)
```

The getPredecessors function can be used in skip list search, insert, delete.

Algorithm 15 SkipList search

Require: k

```
P ← getPredecessors(k)
q ← P.top
if q.after.k = k then return q.after
else return not found
```

To insert into a skip list, we need to randomly generate the height of the node stack for the inserted key. The height is determined by flipping a coin until a tail is flipped, and counting the number of heads.

To delete from a skip list, link the predecessors of the node stack to node after for each level. In both insert and delete operations, the number of levels needs to be adjusted so that the top layer is the only layer which contains only the two sentinels.

Let X_k be the height of a tower. Then, $X_k \sim \text{Geometric}(p = 1/2)$. And,

$$P(X_k \geq i) = \left(\frac{1}{2}\right)^i$$

$$P(X_k = i) = \left(\frac{1}{2}\right)^{i+1}$$

X_k is independent between towers.

Let S_i be the list of keys in layer i. We can determine expected length of S_i to be

$$E(|S_i|) = \frac{n}{2^i}$$

And,

$$E(h) = 2 + \log n$$

For search, insert, delete, the expected time determined by number of scan-forward operations is $O(\log n)$.

4.4 Biased search requests

Unordered lists/arrays can be made more efficient by using an optimal ordering if some items are accessed more frequently than others.

Definition 4.5 (Optimal static ordering)**.** An array in **optimal static ordering** is ordered by frequency of access known beforehand.

Sometimes the frequencies are not known beforehand.

Definition 4.6 (Move-to-front heuristic)**.** After search, move the accessed item to the front/back.
MTF is 2-competitive, meaning it is at most twice as expensive than optimal static ordering.

There is also transpose heuristic and frequency-count heuristic.

5 Dictionary for special keys

5.1 Interpolation search

There is a lower bound to comparison-based search.

Theorem 5.1. $\Omega(\log n)$ comparisons are required for search in comparison-based model.

When keys are close to evenly distributed, search can be done more optimally.

Algorithm 16 InterpolationSearch

Require: A, k
  $l \leftarrow 0$
  $r \leftarrow n-1$
  while $l \leq r$ **do**
    if $k < A[l]$ or $k > A[r]$ **then return** not found
    if $k = A[r]$ **then return** $A[r]$
    $m \leftarrow l + \left\lfloor \frac{k-A[l]}{A[r]-A[l]}(r-l) \right\rfloor$
    if $k = A[m]$ **then return** $A[m]$
    else if $A[m] < k$ **then** $l \leftarrow m+1$
    else $rm - 1$

Interpolation search has $O(\log \log n)$ average time.

5.2 Trie

In some cases we want to store words in a dictionary, where a word is a sequence of characters over alphabet Σ.

Definition 5.1 (Trie)**.** A **trie** (radix tree) is a tree where each edge correspond to a character in a string or the end of string (\$). Words are stored in leave nodes.

Algorithm 17 getPathTo

Require: w

```
z ← root
P ← [z]
d ← 0
while d ≤ |w| do
    if z[w[d]] exists then
        z ← z[w[d]]
        d ← d + 1
        P.push(z)
    else break
```

Algorithm 18 Trie search

Require: w

```
P ← getPathTo(w)
z ← P.top
if z is not a leaf then return not found
return z
```

For each node, a reference to the leaf containing the longest word in the subtree can be stored to speed up prefix search.

Search, prefix search, insert, delete all take $\Theta(|w|)$ time, independent of n.

5.3 Trie compression

Definition 5.2 (Pruned trie)**.** A **pruned trie** is a trie where subtrees with one key are represented as a single node.

Pruned trie saves space when there are few long strings. However, it can be compressed further by contracting chains in internal nodes.

Theorem 5.2 (Tree with no chains)**.** Let T be a tree with m leaves. If every internal node has at least 2 children, then the tree has at most $m - 1$ internal nodes.

Definition 5.3 (Compressed trie)**.** A **compressed trie** (Patricia trie) is a trie where each node stores the index of the next character to check during a search. A compressed trie with n keys always has $n-1$ internal nodes.

A compressed trie uses $O(n)$ space.

6 Dictionaries by hashing

6.1 Hashing

Definition 6.1 (Direct addressing)**.** If every key k is an integer and $k \in [0, M-1]$, with **direct addressing**, the key is stored in the corresponding k-th slot in the table.

Search, insert and delete are done by directly accessing $A[k]$. The drawback is that M needs to be sufficiently large which wastes space, and keys needs to be integers.

Definition 6.2 (Hash function)**.** A **hash function** is a function $h : U \to [0, M-1]$, used to determine the slot to store a KVP in a hash table.
$h(k)$ is the **hash value** of k.

Hash functions should be fast to compute ($O(1)$). If a hash function is not injective, collisions may occur. There are many strategies to handle collisions.

6.2 Hashing with chaining

Definition 6.3 (Hash chaining)**.** With **chaining**, multiple items are allowed in each slot, and are generally stored in linked lists.

Search looks for key k in the list at the slot $h(k)$. MTF heuristic can be applied. Insert adds the KVP to the front of the list at the slot.

Insert takes $O(1)$ time. However, search and delete can take up to $\Theta(n)$ time if all items hash to the same slot.

Definition 6.4 (Load factor)**. Load factor** represents the ratio of number of items to the number of slots in a hash table:

$$\alpha = \frac{n}{M}$$

Definition 6.5 (Uniform hashing assumption)**. Uniform hashing assumption** (UHA) assumes any possible hash function is equally likely to be chosen.

UHA is used to make average-case analysis possible for hash tables. With UHA, each item is equally likely to hash into any slot. That is,

$$P(h(k) = i) = \frac{1}{M}$$

for any key k and slot i.

Theorem 6.1. For any key k the expected size of the bucket at $h(k)$ is at most $1 + \alpha$.

When α become high enough, set $M \leftarrow 2M$ and perform a rehash to reduce collisions.

6.3 Open addressing

There are ways to resolve coliisions within the table without using additional space. Insert can be modified to follow a sequence of possible locations until an empty slot is found.

Definition 6.6 (Linear probing)**.** If $h(k)$ is occupied, place at the next available location. The probe sequence is $h(k, i) = h(k) + i \pmod{M}$

Delete requires the slot to be marked as deleted without clearing it, so that search can still find subsequent slots.

Linear probing may cause entries to be clustered into regions, which causes many probes to be needed for the operations. To avoid consecutive locations, a separate hash function can be used to determine the step size on probe collsion.

Definition 6.7 (Double hashing)**.** Open addressing with probe sequence $h(k, i) = (h_0(k) + ih_1(k)) \pmod{M}$

h_0, h_1 should be independent.

Definition 6.8 (Cuckoo hashing)**.** Use 2 tables T_0, T_1 with different hash functions h_0, h_1. A key k is at either $T_0[h_0(k)], T_1[h_1(k)]$.

If there is a collision during insertion, move the existing key into the other table determined by the other hash function. Repeat until the slow to be inserted is empty. If there is an infinite loop, perform a rehash.

6.4 Hash function

We want a hash function that is easy to compute and aims for $\frac{1}{M}$ probability of collision between two keys. Examples include:

- $h(k) = k \pmod{M}$ for some prime M.
- $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor, A \in [0, 1] \subset \mathbb{R}$. $A = \varphi$ is good.

7 Range search dictionary

7.1 Range search

Range search looks for all items in a given interval. This is simple for single dimensional data, which can be done in $O(\log n + s)$ time using sorted array of balanced BST. However, for multi-dimensional data, this is more difficult as the previous data structures can only sort the keys for one dimension.

7.2 Quadtree

Definition 7.1 (Quadtree)**.** A **quadtree** is a tree which recursively divides a 2D region. The root node represents a bounding box of size $2^k \times x^k$. Each internal node contains 4 children, splitting up the space into 4 regions. Points are stored in the leaves, which contains up to one point.

Quadtrees are similar to pruned tries in its operations.

Definition 7.2 (Spread factor)**.** The **spread factor** of a quadtree is defined by

$$\rho(S) = \frac{L}{d_{\min}}$$

where L is the side length of the quadtree, and $d_{\min}$ is the smallest distance between any two points in S.

In the worst case, $h \in \Omega(\log \rho(S))$.

Algorithm 19 Quadtree RangeSearch

Require: r : node, Q : query rectangle
  $R \leftarrow r$.region
  if $R \subseteq Q$ **then return** all points below r
  if $R \cap Q = \emptyset$ **then return** empty
  if r is a leaf **then**
    $p \leftarrow r$.point
    if $p \neq$ null and $p \notin Q$ **then return** p
    else return empty
  $S \leftarrow []$
  for $v \in r$.children **do**
    S.append(RangeSearch(v, Q))
  return S

Quad tree range search has $\Theta(nh)$ worst case time.

Quadtrees are efficient when points are close to evenly distributed. Quadtrees generalize to higher dimensions, such as octrees for 3 dimensions.

7.3 kd-tree

Definition 7.3 (kd-tree)**.** A **kd-tree** is a binary tree which recursively splits points into two equal regions, alternating between vertical and horizontal.

To build a kd-tree, recursively split the nodes along the median. This can be done using QuickSelect (partition).

Range search with kd-tree is similar to quadtree.

Algorithm 20 kd-tree RangeSearch

Require: r : node, Q : query rectangle
  $R \leftarrow r$.region
  if $R \subseteq Q$ **then return** all points below r
  if $R \cap Q = \emptyset$ **then return** empty
  if r is a leaf **then**
    $p \leftarrow r$.point
    if $p \in Q$ **then return** p
    else return empty
  $S \leftarrow []$
  for $v \in r$.children **do**
    S.append(RangeSearch(v, Q))
  return S

Range search for kd-tree has $O(s + \sqrt{n})$ time.

At higher dimensions, range search can be done in $O\left(s + n^{1-\frac{1}{d}}\right)$ time. Other operations can be done in the same time independent of d.

7.4 Range tree

Balanced BST is efficient for one-dimensional dictionaries. Range search can be done in $O(s + \log n)$ time. To range search, find the paths leading to the the boundary nodes P_1, P_2. Then, report the descendents of all topmost inside nodes.

Definition 7.4 (Range tree)**.** A 2D **range tree** is a BST sorted by the x-coordinate key. Each node also contains an associate tree, which is its subtree sorted by the y-coordinate.

Range search works similarly to a BST. Begin by performing range search on the main tree by the x-coordinate. For topmost inside nodes, range search their associate tree by the y-coordinate.

Range trees require $O(n \log n)$ space.

Range trees can be generalized to higher dimensions using multiple layers of associate trees. In this case, range trees use $O(n(\log n)^{d-1})$ space, $O(n\log n^d)$ time to construct, $O(s + (\log n)^d)$ time to range search.

Range trees trade space for search and range search efficiency.

8 String matching

Definition 8.1 (String matching)**.** Search for a string (pattern) within a body of text.

A simple string matching algorithm is the brute force algorithm, which checks over all possible positions of the pattern until a match is found. This has $\Theta(m(n-m))$ time, where $m = |P|, n = |T|$. For medium sized m, this leads to $\Theta(n^2)$ time.

8.1 Karp-Rabin

Hash values (fingerprints) can be used to eliminate guesses faster. To calculate and update the hash efficiently, we can implement rolling hash which is updatable in constant time for the subsequent fingerprint.

Algorithm 21 KarpRabinMatch

Require: T, P
  $M \leftarrow$ some prime number
  $h_P \leftarrow h(P)$
  $h_T \leftarrow h(T[0 \dots m-1])$
  $s \leftarrow R^{m-1} \pmod M$
  for $i \in [0, n-m]$ **do**
    if $h_T = h_P$ and strcmp$(T[i \dots i+m], P)$ **then return** i
    if $i < n-m$ **then**
      $h_T \leftarrow R((h_T) - sT[i]) + T[i+m] \pmod M$
  return fail

8.2 KMP

KMP (Knuth-Morris-Pratt) algorithm starts similar to brute-force pattern matching. However, on mismatch, shift the pattern forward such that the longest prefix of the pattern matchs the longest suffix of the pattern before the mismatch character.

KMP uses a failure array to determine the pattern shift after failure. The failure array maps the index at failure to the index in pattern to shift to after mismatch.

Outside of computing failure array, KMP has $\Theta(n)$ time.

Fast computation of failure array can be done in $\Theta(m)$ time, which is done in a similar way to KMP itself. The failure array is computed as it is being used.

Algorithm 22 KMPMatch

Require: T, P
```
F ← FailureArray(P)
i ← 0
j ← 0
while i < n do
    if P[j] = T[i] then
        if j = m − 1 then
            return i − m + 1
        else
            i ← i + 1
            j ← j + 1
    else
        if j > 0 then
            j ← F[j − 1]
        else
            i ← i + 1
return fail
```

Algorithm 23 FailureArray

Require: P
```
F[0] ← 0
j ← 1
l ← 0
while j < m do
    if P[j] = P[l] then
        l ← l + 1
        F[j] ← l
        j ← j + 1
    else if l > 0 then
        l ← F[l − 1]
    else
        F[j] ← 0
        j ← j + 1
```

Together, KMP has $\Theta(n+m)$ time.

8.3 Boyer-Moore

Boyer-Moore algorithm provide fastest pattern matching in practice for English text. The algorithm search the pattern in reverse. When a mismatch occurs, the algorithm uses **bad character heuristic** and **good suffix heuristic** to eliminate unnecessary checks.

For bad character heuristic, first compute the last occurrence array $L(c)$ which maps a character to the index of last occurrence in P, or -1 if $c \notin P$. This can be done in $O(m+|\Sigma|)$ time. When a mismatch occurs at character $c \in T$, shift the pattern so that the last occurrence of the character in text in P lines up with c. If $c \notin P$, shift the pattern completely past the current window. If the shift causes the pattern to move backwards, instead shift the pattern forward by one character similar to brute-force.

Algorithm 24 BoyerMoore

Require: T, P
  $L \leftarrow \text{LastOccurrenceArray}(P)$
  $j \leftarrow m-1$
  $i \leftarrow m-1$
  while $i < n$ and $j \geq 0$ **do**
    if $T[i] = P[j]$ **then**
      $i \leftarrow i-1$
      $j \leftarrow j-1$
    else
      $i \leftarrow i+m-1-\min\{L(c), j-1\}$
      $j \leftarrow m-1$
  if $j = -1$ **then**
    return $i+1$
  else
    return fail

Good suffix heuristic is similar to KMP failure array, but applied to the suffix. On mismatch, the pattern is shifted so that the good suffix aligns with the last occurence of the suffix in P.

8.4 Suffix tree

Suffix tree stores all suffixes of T in a trie. This can be used for efficient string matching for multiple patterns on the same text. Suffix tree uses $\Theta(n)$ space if the trie is compressed.

Generally suffix tree can be built in $\Theta(|\Sigma|\, n^2)$ time. Pattern matching uses $O(m)$ time.

Suffix tree is good but is slow to build and takes up a lot of space.

8.5 Suffix array

Suffix array is an alternative to suffix tree. It stores all suffixes of T in lexicographic order.

To build the suffix array, use MSD radix sort on suffixes of T. This takes $\Theta((n + |\Sigma|) \log n)$ time.

To pattern match, perform binary search on the suffix array. This takes $\Theta(m \log n)$ time.

9 Data compression

Compression is the process of mapping the source text S to the coded text C. Lossless compression allows S to be exactly recovered from C for every input.

We want to minimize the size of C. This can be quantified using compression ratio.

Definition 9.1 (Compression ratio)**.**

$$\frac{|C| \log |\Sigma_C|}{|S| \log |\Sigma_S|}$$

Ideally compression ratio should be below 1.

Theorem 9.1 (Impossibility of compression)**.** No lossless encoding scheme can have CR < 1 for all input strings.

However, real-life data has patterns that can be compressed. We can design encoding schemes that work well for frequently occuring patterns.

If S and C are too large, they must be stored as **streams**. This means the algorithm has to process the input one character at a time.

9.1 Single-character encodings

Definition 9.2 (Character encoding)**.** Character encoding E maps each character in the source alphabet to a string in the coded alphabet. That is,

$$E : \Sigma_S \to \Sigma_C^*$$

Character encodings can be fixed-length or variable-length. For example, ASCII is fixed-length, and Morse code is variable-length.

Fixed-length codes do not compress. Variable-length codes can allocate shorter codes for more frequent characters, which can make the coded text shorter.

The decoding algorithm must uniquely map $\Sigma_C^* \to \Sigma_S$. **Prefix-free** codes can do this without an end character.

9.2 Huffman code

We can build the best trie given T in the following way:

1. Create tries with one node for each character in Σ_S, labelled with the frequency of the character.
2. Join two least frequent tries into a new tries, using the sum of frequencies of the two tries as the frequency.
3. Repeat until there is one trie remaining.

A min-heap can be used in this process.

Huffman encode takes $O(|\Sigma_S| \log |\Sigma_S| + |C|)$ time.

9.3 Run-length encoding (RLE)

RLE is an example of **multi-character encoding**. It encodes run of single characters into the length of the run.

Elias gamma code can be used to encode positive integers used for run length prefix-free. This is done by adding $m - 1$ zeros before the binary number, where m is the number of digits.

The best case of RLE is where the entire string is composed of long runs. If $S = 0 \dots 0$, then $|C| = O(\log n)$

9.4 Lempel-Ziv-Welch (LZW)

Sometimes we cannot determine the frequency substrings in a text beforehand. We can use a **adaptive dictionary**, which is a dictionary that is constructed during encoding/decoding.

To encode, start with dictionary D containing all single characters. For each step, find the longest string in S which is in D. The string is encoded with the corresponding code in D. Then, the string and the subsequent character is added into D as a new entry.

To decode, the dictionary D is reconstructed in a similar way to encoding. However, D is one step behind, which is problematic when a code requires the next entry in D which doesn't exist yet. In this case, the entry can be reconstructed by $s_{\text{prev}} + S_{\text{prev}}[0]$

Encoding and decoding is both $O(|S|)$ time.

9.5 Bzip2

Bzip2 is a compression scheme which uses **text transforms**. Text transforms map texts to another text, which may compress better. Bzip2 uses Burrows-Wheeler transoform (BWT), move-to-front transform, modified (zero-only) RLE, and Huffman coding.

MTF transform begins with an array of all characters L. Upon retrieving a character c from S, encode it as the index of the character in L and move that character to the front in L.

Algorithm 25 LZWEncode

Require: S, C

```
D ← initial dictionary
i ← new dictionary entry start (128)
while S is not empty do
    v ← D.root
    while S is not empty and c = v[S.top] exists do
        v ← c
        S.pop
    C.append(v)
    if S is not empty then
        v[S.top] = i
        i ← i + 1
```

To decode MTF transform, start with the same L and look up L to determine the original character.

To encode BWT:

1. List all cyclic shifts of S.
2. Sort the array of cyclic shifts lexicographically.
3. Combine the last character of each array into a single string and return the string.

Sorting can be done efficiently using MSD radix sort, with consideration that anything after the end character \$ does not matter.

To decode BWT:

1. Create the shift array where the last element corresponds to a character in C.
2. Sort C in lexicographic order and write the string to the first characters of the shift array.
3. Starting from the string with \$ at the end, let c be the character in front.
4. Determine the index i for which c occurs in front out of all strings.
5. For the i-th occurence of c in the last character of all arrays, place the

10 External memory

10.1 Stream-based algorithm

Definition 10.1 (External memory model (EMM))**.** In EMM, there is an unbounded external memory which stores the input with size n. The algorithm can fetch the memory one block at a time.

For stream-based algorithms, need to process input one block at a time, and push to output one block at a time. Runtime can be determined based on the number of block transfers.

Generally, $\frac{n}{B}$ block transfers are required to read input or write output of size n.

Merge sort can be implemented with $O\left(\frac{n}{B}\log n\right)$ block transfers.

10.2 External dictionary

Standard BST has poor memory locality. That is, nearby nodes are unlikely to be in the same block. There are data structures that can improve locality.

2-4 trees are BST where each node can store 1-3 nodes and 2-4 children. Nodes are ordered such that nodes to the left are smaller than nodes to the right. And, all subtrees are at the same level.

Searching for 2-4 trees is similar to BST.

To insert, find the path to the node where the key can be inserted. If inserting would cause a node overflow, split the node and give the parent one additional node.