Project Report

Prepared by: Warul Kumar Sinha (2023201045)
Under the advisory of: Dr. Kshitij Gajjar

A variety of algorithms exist to solve the same problem, but nuances often affect practical applicability. This project aims to analyse, benchmark, compare, and document algorithms alongside such nuances.
Information on benchmarking, optimizations and references can be found under Docs.

Contents

1 Hashing

Hashing is often considered to be a magical domain that allows a great degree of optimization in practice.
While there exist a bunch of hashing algorithms, it is oftentimes difficult to reason about the quality of the hasher.
Certain analytical techniques allow us to study various characteristics such as pairwise/k-wise independence of inputs.[5]
Here, we focus instead on the avalanche test,[2] a property that postulates that a hash function is typically considered good if for a single bit change in the input, half the bits in the output change.

1.1 Hashing Algorithms

- **Polynomial Rolling Hash.[3]**
  Let s be a string of length n. Define

  $$h_s = 0, \quad p = 31, \quad m = 10^9 + 9,$$

  and then for $i = 0, \ldots, n-1$:

  $$h_s \leftarrow (h_s + \text{ascii}(s[i]) \cdot p^i) \bmod m.$$

- `std::hash` **(FNV-1a).[4]**

  ```
  define fnv-1a:
    hash := FNV_offset_basis
    for each byte_of_data to be hashed do
      hash := hash XOR byte_of_data
      hash := hash × FNV_prime
    return hash
  ```

- **XOR Hash.**
  A simple hash function where the hash is the bitwise XOR of all 32-bit disjoint components (a string broken into 4-character sets).

- **Random-Hash.**
  A simple hash function where the hash is a random hash chosen from a family of hashes where each hash function is represented by a distinct mask of 32 bits (corresponding to selection of bits from input).
  The mask is applied to the input and the result is taken modulo m.

1.2 Results

Refer to Section 2.

2 Hashing Benchmarks

Each algorithm is tested against 5,000 randomly generated alphabetical strings of length 256, where one random index has one bit flipped.
For Random-Hash, we use 5,000 random 32-bit unsigned integers and compute the average number of bits flipped on hashing.

Algorithm	**Average Bits Flipped**
Rolling Hash	13.13
`std::hash`	14.30
XOR Hash	0.99
Random-Hash	1.90

Table 1: Average number of bits flipped by different hashing algorithms.

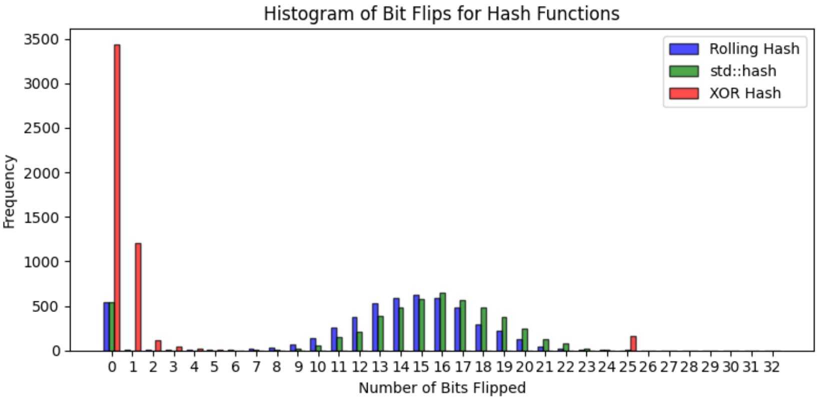

Figure 1: No. of bits flipped on average by different hashing algorithms.

It is observed that among the given implementations, `std::hash` performs best, with the rolling hash close behind.
The nature of XOR hash and Random-Hash leads to poor performance on the avalanche test.
XOR hash performs poorly particularly because for the given algorithm, one bit flip in the input, implies one bit flip in the output.

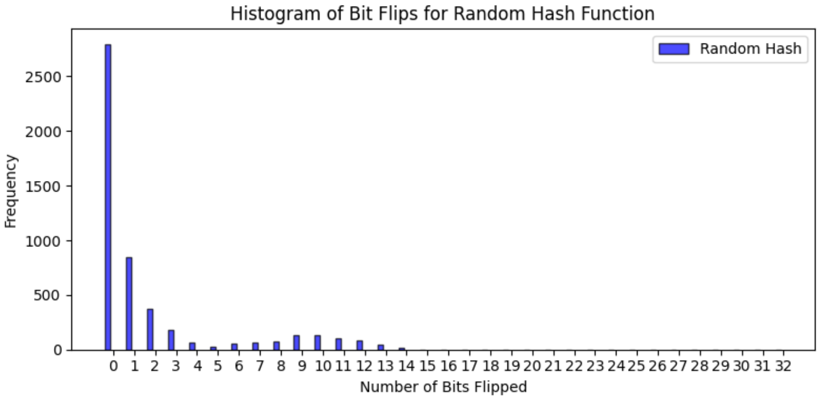

Figure 2: No. of bits flipped on average by the Random-Hash algorithm.

Random-Hash performs poorly because a flipped bit only affects the output if that bit is among the chosen bits for the mask (each chosen with probability 0.5), and because of the modulo distribution.

3 Range Queries

3.1 Fenwick Trees (Binary Indexed Trees)[6]

Given an array of integers, $A[0 \ldots N-1]$, a Fenwick tree is just an array, $T[0 \ldots N-1]$, where each element is equal to the sum of elements of A in the range, $[g(i), i]$:

$$T_i = \sum_{j=g(i)}^{i} A_j$$

where

$$g(i) = i \ \& \ (i+1),$$

```
define sum(int r):
  res := 0
  while (r >= 0):
      res := res + t[r]
      r := g(r) - 1
  return res
```

The above adds $[g(r), r]$, $[g(g(r)-1), g(r)-1]$, ..., $[g(-1), -1]$ to the result.

```
define increase(int i, int delta):
  for all j with g(j) <= i <= j:
      t[j] := t[j] + delta
```

Here jumps happen in the opposite direction; we need a way to iterate over all j's, such that $g(j) \leq i \leq j$. We can find all such j's by starting with i and flipping the last unset bit. We will call this operation $h(j)$.

$$h(j) = j \mid (j+1)$$

Using this, we get the following:

```
define increase(int i, int delta):
  while (i < n):
    t[i] := t[i] + delta
    i = h(i)
```

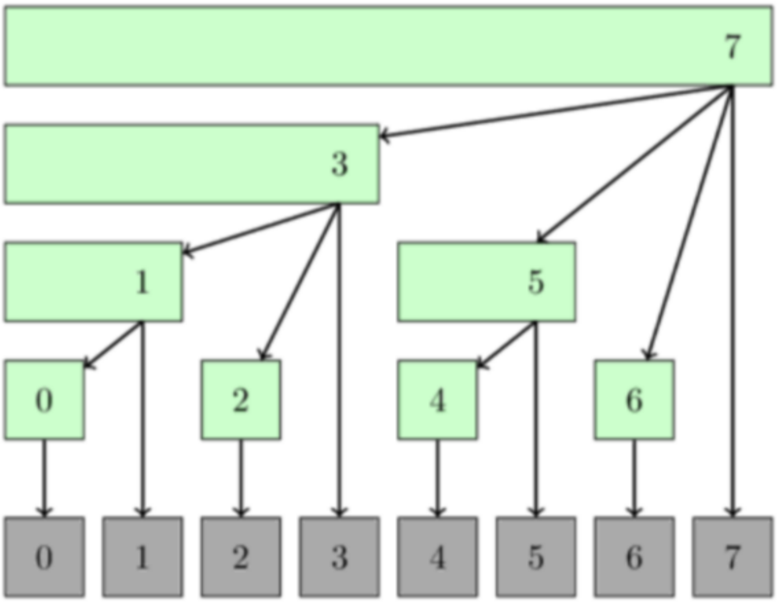

Figure 3: Nodes of a Fenwick tree and covered ranges.[6]

Time Complexity: While a standard construction may take $O(n \log n)$ time, it is possible to do so in linear time.

- Precompute: $O(n)$
- Query: $O(\log n)$

3.2 Segment Trees[7]

A Segment Tree stores information about array intervals in a tree, allowing efficient range queries and updates.
The idea is to create a binary tree, where the root covers aggregate over the entire range, leaf nodes only cover a single element and any internal node has two children, each covering half its range.
The above enforces logarithmic depth since at each level, ranges get halved. The logarithmic depth allows for logarithmic query times for both updation and interrogation.
This holds as at each level only $O(1)$ nodes are relevant to any query (since anything to the left/right of a range can be ignored for computation).

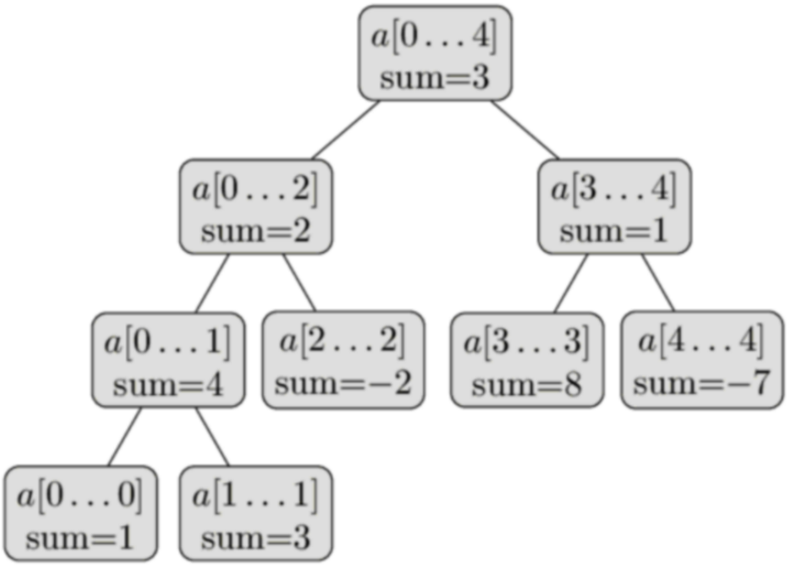

Figure 4: Nodes of a Segment tree constructed over $[1, 3, -2, 8, -7]$.[7]

Time Complexity:

- Precompute: $O(n)$
- Query: $O(\log n)$

The iterative variant[8] of segment trees instead follows a bottom-up approach, where the tree is treated as an array and starting from the leaf-level, relevant nodes are visited only using a loop and index arithmetic.

3.3 Results

Refer to Section 4.

4 Range Queries Benchmarks

Each size is tested with 10 instances per algorithm (evenly split update/sum queries).

Array length, query count categories:

- Small-1: 10–100, 10–100
- Small-2: 10–100, 100–1,000
- Medium-1: 1,000–10,000, 1,000–10,000
- Medium-2: 1,000–10,000, 10,000–100,000
- Large: 100,000–1,000,000, 100,000–1,000,000

	Fenwick Tree	Segment Tree	Iterative Seg. Tree
Small-1	(1,255, 3,454)	(3,853, 10,065)	(1,205, 4,383)
Small-2	(2,357, 16,172)	(6,359, 49,646)	(3,123, 21,445)
Medium-1	(29,236, 161,666)	(126,403, 844,780)	(44,588, 313,366)
Medium-2	(430,298, 1,735,205)	(2,341,528, 10,133,545)	(843,497, 3,598,143)
Large	(8,258,384, 25,595,156)	(38,811,311, 173,135,398)	(11,716,570, 54,932,104)

Table 2: Least and highest execution times (in nanoseconds) across inputs.

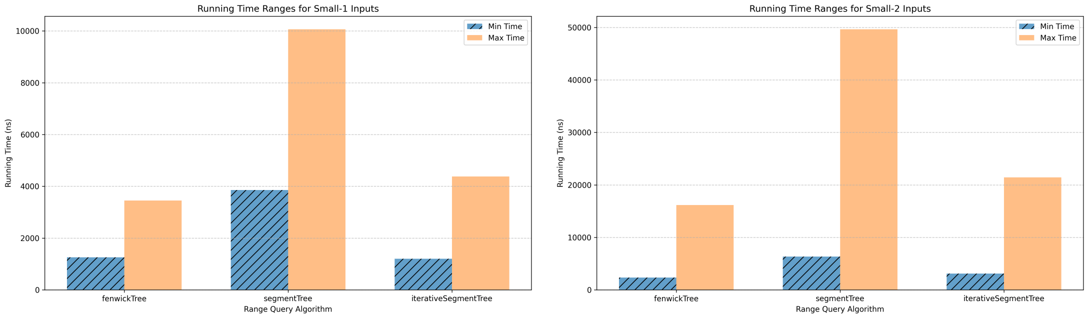

Figure 5: Top: Small-1 (few queries) & Small-2 (many queries).

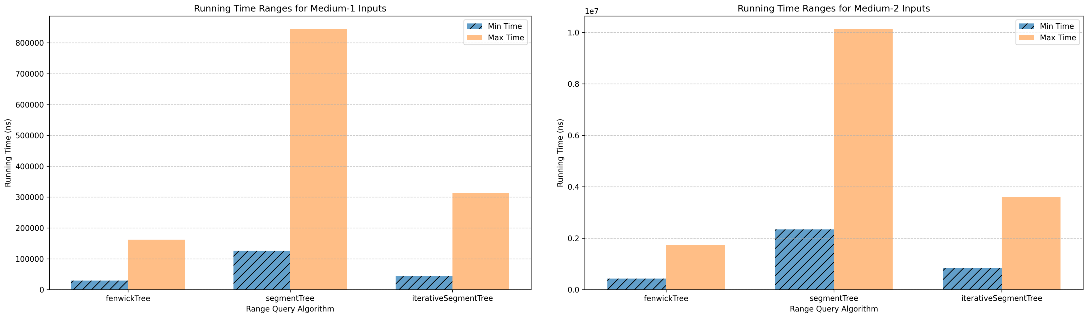

Figure 6: Medium-1 (few queries) & Medium-2 (many queries).

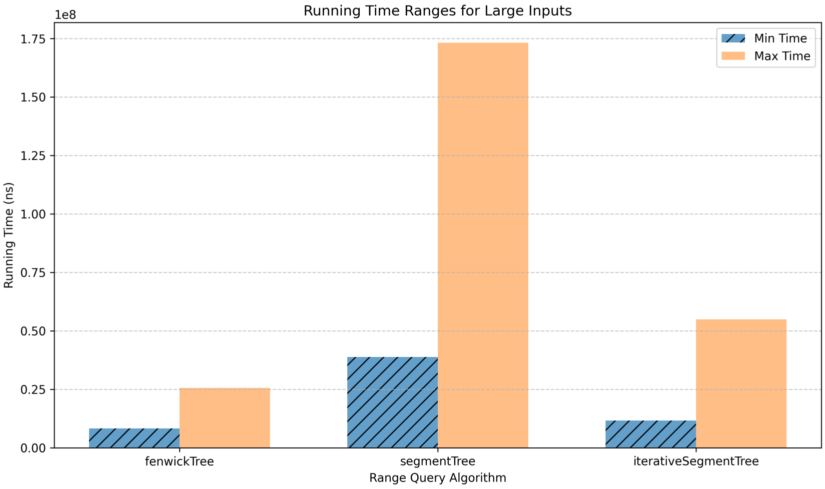

Figure 7: Large inputs.

It is observed that across array sizes and query counts, fenwick trees typically perform best among the three, with the iterative version of segment tree following it closely, and the recursive segment tree lagging behind. In practice, the versatility of segment trees brings them to use above fenwick trees. For instance, even to handle range minimum queries, the standard fenwick trees fail, and we must use two in conjunction.[9]

5 Sorting

5.1 Insertion Sort[10]

Standard insertion sort with $O(n^2)$ running time for an n length array.

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

5.2 Merge Sort[11]

Standard merge sort with $O(n \log n)$ running time for an n length array.

```
define merge_sort(m):
    if length(m) <= 1 then return m
    split m into left/right halves
    left  := merge_sort(left)
    right := merge_sort(right)
    return merge(left, right)

define merge(left, right):
  var result := empty list

  while left is not empty and right is not empty do
    if first(left) <= first(right) then
      append first(left) to result
      left := rest(left)
    else
      append first(right) to result
      right := rest(right)

  while left is not empty do
    append first(left) to result
    left := rest(left)

  while right is not empty do
```

```
    append first(right) to result
    right := rest(right)
```

5.3 QuickSort[12]

Standard quick sort using random-pivot with worst case $O(n^2)$ and average case $O(n \log n)$ running time for an n length array.

```
define quicksort(A, lo, hi):
    if lo >= hi then return
    p := partition(A, lo, hi)
    quicksort(A, lo, p-1)
    quicksort(A, p+1, hi)

define partition(A, lo, hi):
  pivot_idx = random(lo, hi)
  swap A[pivot_idx] and A[hi]
  pivot := A[hi]
  i := lo

  for j := lo to hi - 1 do
    if A[j] <= pivot then
      swap A[i] with A[j]
      i := i + 1

  swap A[i] with A[hi]
  return i
```

5.4 Hybrid Sort

While a variety of hybrid sorting algorithms exist, we study a mix of quicksort and insertion sort for its practical efficiency.
A mix of the quick-sort above and insertion sort, with $INSERTION\_SORT\_LIMIT = 65$.[13]
A subarray s of length $slen$ is sorted using insertion sort if $slen \leq INSERTION\_SORT\_LIMIT$.
Otherwise, the mechanics of quick sort as above are applied.
Algorithm runs with worst case $O(n^2)$ and average case $O(n \log n)$ running time for an n length array.
Faster in practice considering the practical improvements through insertion sort's efficiency over small subarrays.
Other noteworthy hybrids:

- Merge Sort + Insertion Sort
  Same as above, applying insertion sort on small subarrays. An interesting variant of the same can be found under Timsort.

- QuickSort + HeapSort
  This algorithm applies quicksort till a recursive-depth of $O(\log n)$ for an n length array, and switches to heapsort thereafter if still unsorted. Doing so allows the algorithm to guarantee worst case $O(n \log n)$ running time while benefitting from general efficiency of quicksort.

5.5 Results

Refer to Section 6.

6 Sorting Benchmarks

Tested on 10 instances per algorithm in three size categories:

- Small: 10–200
- Medium: 1,000–20,000
- Large: 100,000–2,000,000

	InsertionSort	QuickSort	MergeSort	HybridSort
Small	(709, 7,473)	(768, 6,078)	(933, 7,776)	(763, 6,022)
Medium	(141,312, 10,548,489)	(81,225, 880,327)	(94,431, 1,043,705)	(72,792, 769,667)
Large	(NA, NA)	(9,733,001, 98,435,984)	(11,691,272, 120,194,603)	(8,779,871, 90,966,402)

Table 3: Sorting algorithms: least and highest times (nanoseconds).

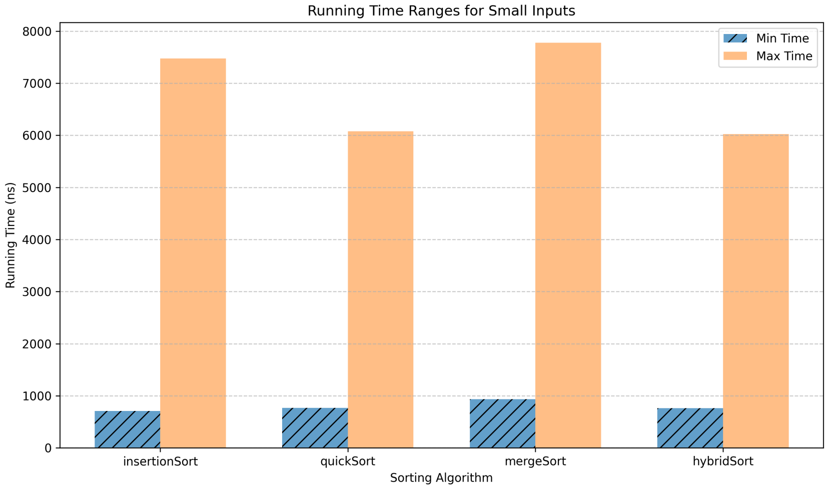

Figure 8: Least and highest running times on small inputs.

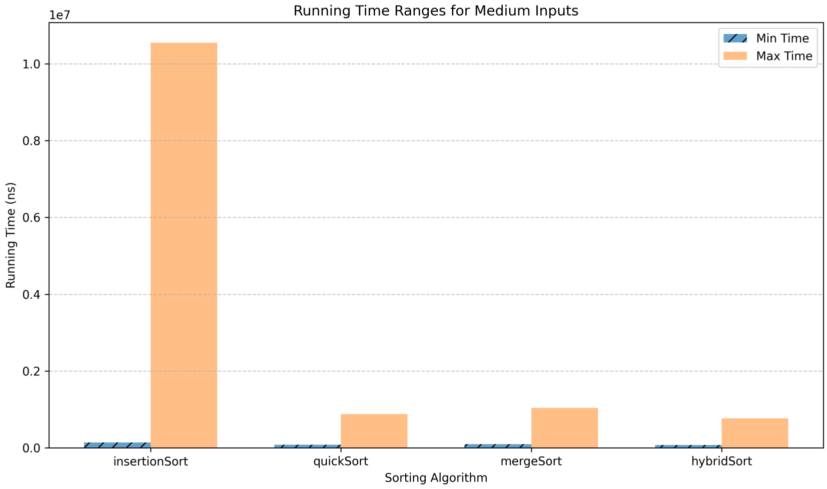

Figure 9: Least and highest running times on medium sized inputs.

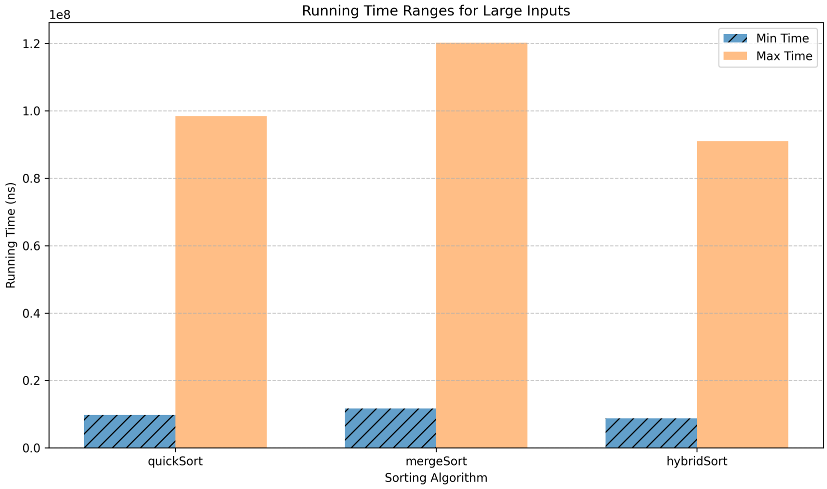

Figure 10: Least and highest running times on large inputs.

It is observed that for small inputs, InsertionSort is typically fast in practice, by virtue of its reliance on the same, HybridSort performs similarly, QuickSort lags behind a little and MergeSort is slightly slower still.
In comparison, for medium to large sized inputs, InsertionSort expectedly blows up (since it is quadratic in time complexity) and HybridSort performs best, QuickSort lags behind a little and MergeSort is slightly slower still.

7 Strings

7.1 KMP[15]

The KMP algorithm, named after Knuth, Morris, and Pratt, is an efficient implementation of the prefix function.
Given a string s, of length n, the prefix function is defined as an array p of length n, where $p[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$ which is also

a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $p[0] = 0$.
Mathematically, $p[i] = max(k : s[0 \dots k-1] = s[i-(k-1)..i])$, $k = 0 \dots i$
While a trivial implementation runs in $O(n^2)$ time, KMP runs in $O(n)$ time.
An obvious observation is that the values of the prefix function can only increase by 1 between neighbouring indices.
Secondly, if $s[i+1] = s[p[i]]$, $p[i+1] = p[i]+1$.
Next, we can use the fact that $s[p[i]+1 : i] = s[0 : p[i]-1]$ and find the largest prefix matching thus far, by looking at the value of p for the prefix as it is equivalent to the current substring.
This process is repeated until either the substring can be extended to match a prefix or there is no match.
From the above, since there can be at most n increments to the prefix value, and an increment/decrement takes $O(1)$ time, the running time is bounded by $O(n)$.

Applications:

- Search for a substring in a string
- Counting the number of occurrences of each prefix
- Compressing a string - representing a string s as $t + \cdots + t$ for some prefix t with the shortest length.

7.2 Manacher[16]

Manacher's algorithm provides an efficient way to find all sub-palindromes in a string s of length n in $O(n)$ time.
Formally, given a string s with length n, it finds all the pairs (i, j) such that substring $s[i \dots j]$ is a palindrome. Note that while there may be upto n^2 such substrings, we can instead store their information succintly, only keeping the following:

- for odd length palindromes, store for the central character, the length of the largest palindromic substring
- for even length palindromes, store for the central gap, the length of the longest palindromic substring

By only storing the longest around the center, we know all others as well since for any palindrome p of length k, $p[1 \dots k-2]$ (0 indexed) is also a palindrome.
The algorithm stores the right-most found subpalindrome (l, r). Since we know that (l, r) is a palindrome, we can find the mirror position of substrings within to check for extension with the current character.

Applications:

- Find largest subpalindrome
- Count no. of subpalindromes

7.3 Rabin-Karp[17]

Based on the idea of hashing, the Rabin-Karp algorithm compares the hash of substrings of a text t to that of a pattern p. The efficient implementation assumes that a matching hash implies (with high confidence) that the substring matches the pattern.
The hash of a substring can be computed by substring $s[i \dots j]$ can be found as follows:

$$hs_{ij} = (h[0 \dots j] - h[0 \dots i-1]) \cdot 2^{n-(j-i)} \bmod m$$

Where n is the length of the whole string. We multiply by the power of two to normalize hashes across differently positioned substrings. In practice, results from our tests agree that the algorithm rarely fails (with good hash functions). It did not fail over a single instance among 100+ test runs.

Applications:

- Search for a substring in a string
- Counting the number of different substrings
- With binary search - count no. of subpalindromes/find largest subpalindrome

7.4 Z-Function[18]

The Z-function calculates for each i, the greatest number of characters starting from the position i that coincide with the first characters of s. In other words, $z[i]$ is the length of the longest string that is, at the same time, a prefix of s and a prefix of the suffix of s starting at i.
The idea is to store the right-most prefix-match found, $[l, r)$. If the current index is outside this, we run the trivial algorithm, otherwise, similar to KMP, we can re-use already computed prefix information.

Applications:

- Search for a substring in a string
- Counting the number of occurrences of each prefix
- Compressing a string - representing a string s as $t + \dots + t$ for some prefix t with the shortest length.

7.5 Results

Refer to Section 8.

---

8 Strings Benchmarks

Tested 100 instances per algorithm in sizes Small (200–400), Medium (2,000–4,000), Large (20,000–40,000), Huge (200,000–400,000).

Size	Rabin–Karp	KMP	Z-Function
Small	(1,636, 45,676)	(601, 4,348)	(657, 5,191)
Medium	(13,937, 102,892)	(2,548, 23,085)	(3,598, 25,824)
Large	(148,434, 764,632)	(24,126, 206,257)	(37,017, 225,779)
Huge	(1,557,756, 4,086,762)	(244,870, 1,547,581)	(324,337, 1,749,875)

Table 4: Least and highest pattern matching times (nanoseconds).

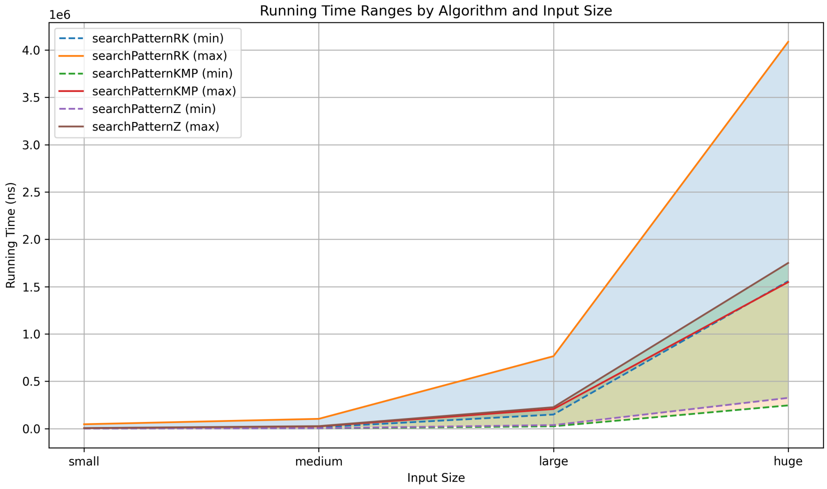

Figure 11: Pattern-matching execution times.

It is observed that among the given implementations, KMP is the fastest with Z-Function slightly behind and Rabin-Karp being much slower.

Subpalindrome Detection:

Size	Hash+Binary Search	Manacher
Small	(13,785, 84,621)	(3,518, 20,550)
Medium	(174,540, 983,490)	(30,515, 161,428)
Large	(2,382,988, 5,553,449)	(389,525, 1,050,119)
Huge	(28,819,973, 67,110,623)	(3,411,973, 10,409,500)

Table 5: Least and highest subpalindrome detection times (nanoseconds).

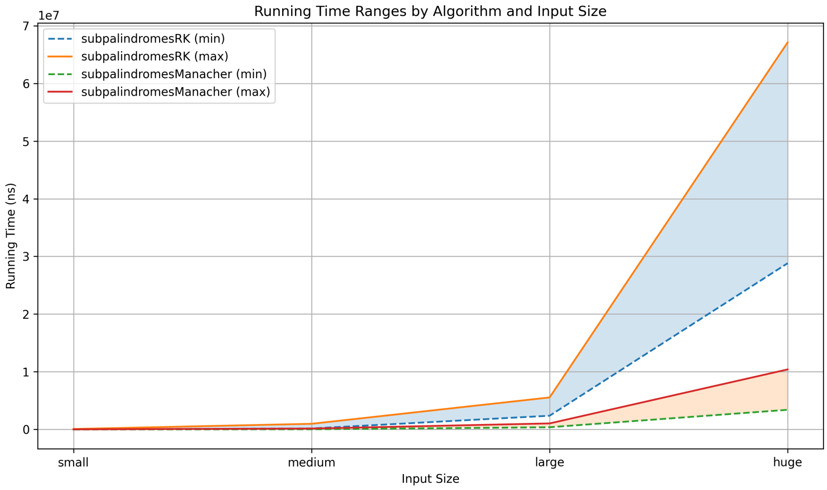

Figure 12: Subpalindrome execution times.

It is observed that Manacher's algorithm vastly outperforms the $O(n \log n)$ alternative, in line with its linear complexity.

9 Additional Algorithms

9.1 Sparse Tables[19]

Sparse Table is a data structure, that allows answering static range queries. The idea is to store for each range $[i : i + 2^j - 1]$, its result. This can be done efficiently using $O(n \log n)$ memory and time precomputation.

- Precompute: $O(n)$ for an n length array.
- Query: $O(\log n)$ or $O(1)$ for idempotent functions

9.2 Square Root Decomposition[24]

Consider the problam of range updates and queries over an array of length n. Let us break the array into $\lceil n/k \rceil$ blocks of length k each.[20] Maintain a range aggregate for each block. Doing so, allows us to performa an update/query in $O(n/k + k)$ time, going over block aggregates and a few extraneous elements to the right/left.
Setting $k = \sqrt{n}$, we can perform both, queries, and, updates in $O(\sqrt{n})$ time.

9.3 Cartesian Tree[22]

A Cartesian tree is a binary tree that follows the min-heap property. The root is the minimum element and the left/right children are built recursively over the left/right subarrays split. It can be constructed in $O(n)$ time using stacks.

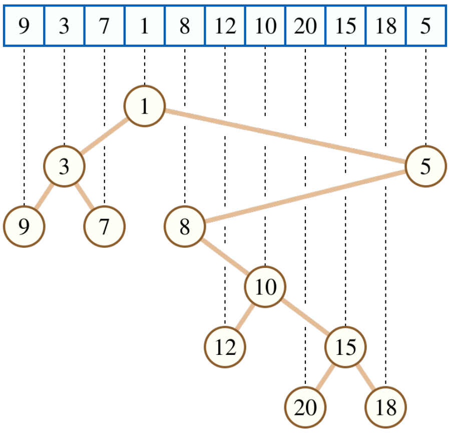

Figure 13: A Cartesian tree example.[22]

9.4 Method of Four Russians[23]

The Method of Four Russians is a technique for speeding up some algorithms over ranges. Particularly, when each cell may take on only a bounded number of possible values.

- Break the problem of size n into subproblems of size b, plus some top-level problem of size n/b.
- This is called a **macro/micro decomposition**.
- Solve all possible subproblems of size b.
- Solve the overall problem by combining solutions to the micro and macro problems.
- Think of it as **"divide, precompute, and conquer."**[21]

Typically, $b = O(\log n)$.

9.5 Lowest Common Ancestor (LCA)

The lowest common ancestor problem involves efficiently finding the lowest common ancestor of any two nodes v_1 and v_2 in a tree, assuming no changes to the tree structure. Trivial algorithms can solve this in $O(n^2)$ time by solving for each pair and storing results. We shall look at other approaches of solving this efficiently.

Binary Lifting[25] Here, for each node, we store it's 2^j distance ancestor. Given that the 2^0 ancestor is just the parent and the 2^{j+1} distance ancestor is the 2^j distance ancestor of the 2^j distance ancestor of the node, tranisitions are straightforward.
This allows for $O(N \log N)$ preprocessing for a tree with N nodes. Further, each query can be answered in $O(logN)$ time, as follows:

- Without loss of generality, let $height(v_1) = u$ ¡ $height(v_2) = v$.
- Using at most $\log N$ jumps, we can reach ancestor v_2' from v_2, such that $height(v_2')$ $= u$.

- We then find the LCA of v_1 and v_2', by finding the largest 2^j ancestor of each such that they differ, and repeat this until finially the parent of the two nodes found is common.
- This node is the LCA.

Farach-Colton & Bender[26] The algorithm borrows from the Method of Four Russians to break the Eulerian tour of the tree array (of heights) into blocks. We divide the array A into blocks of size $K = 0.5 \log N$. There are $\lceil N/K \rceil$ blocks, for each block, store its aggregate, i.e., the minimum, and create a sparse table over these aggregates. The complexity to construct the sparse table is:

$$\frac{N}{K}\log\left(\frac{N}{K}\right) = \frac{2N}{\log(N)}\log\left(\frac{2N}{\log(N)}\right) =$$

$$= \frac{2N}{\log(N)}\left(1 + \log\left(\frac{N}{\log(N)}\right)\right) \leq \frac{2N}{\log(N)} + 2N = O(N)$$

Figure 14: Sparse Table Complexity[26]

Minimum over blocks can be solved using the above sparse table. For minimum within blocks, we observe that the values in the array being heights computed using the Euler tour of the tree, only differ by 1, i.e., $+1$ or -1, let us represent these by 0, 1 respectively. Consider subtracting from each element of a block, its first element, this does not change the result of queries (index of minima) inside the block. Because the blocks are so small, there are only a handful of possible sequences:

$$2^{K-1} = 2^{0.5\log(N)-1} = 0.5\left(2^{\log(N)}\right)^{0.5} = 0.5\sqrt{N}$$

Figure 15: Block Possibilities Count[26]

Thus, the number of different blocks is at most $O(\sqrt{N})$, and therefore results of range minimum queries over them can be precomputed in time:

$$O(\sqrt{N} \cdot K^2) = O(\sqrt{N} \cdot \log^2 N)) = O(N)$$

Mapping between blocks and results can be stored in an array $block[bitmask][left][right]$, storing the results of the block over its subarray $[left, right]$.

Note An interesting implication of the above is a different method for the static range-minium-query (RMQ) problem.[27] By constructing a cartesian tree on the input array (of length n), we can solve the RMQ problem with $O(n)$ precomputation and $O(1)$ time per query. Query resolution is equivalent here to finding the LCA over a range.

10 Docs

10.1 Analysis

Algorithms' algorithmic complexity analysis and correctness guarantees where relevant are included.

10.2 Benchmarking

Real world performance of algorithms/programs depends on a variety of factors such as inputs, infrastructure capabilities, environment, program optimizations, etc. Theoretically optimal algorithms aren't necessarily the best solutions in practice. To understand and study these differences, we benchmark programs as follows:
Program running times may vary, and are considered to be random variables.

- We would like to find statistical guarantees about the expected running times.
- Since expectation and variance are unknown, concentration inequalities cannot be used to estimate tight bounds on expectation.
- Considering the above, we instead study the n-th percentile performance with strict statistical guarantees, particularly the median.

Benchmarking Pipeline Assuming program running times are random variables sampled from a distribution, the following ensues.

- Run the program T times and note the running times, $t_1, \ldots, t_n$.
- Find the least and highest running times:

$$\begin{aligned} lrt &= \min_{1 \leq i \leq n} (t_i) \\ hrt &= \max_{1 \leq i \leq n} (t_i) \end{aligned}$$

- *Claim:* The median running time lies in the range $[lrt, hrt]$ with probability $1 - 2^{-(T-1)}$.
- *Proof:*

$$\begin{aligned} \Pr[\text{median} \in [lrt, hrt]] &= 1 - \Pr[\text{median} < lrt \mid hrt < \text{median}] \\ &= 1 - \Pr[\text{median} < lrt] - \Pr[hrt < \text{median}] \quad \text{(since the events are disjoint} \\ &= 1 - 2^{-T} - 2^{-T} \quad \text{(since probability that a single execution is on either si} \\ &= 1 - 2^{-(T-1)} \end{aligned}$$

- Similar arguments can be used to find n^{th} percentile statistics.

Comparing Algorithms While exact comparisons between algorithms on the basis of the overlapping bounds are not feasible, a direct comparison can instead be made as follows:

- For each pair of algorithms, run both on the same set of inputs and compare running times.
- If either comes ahead by at least a margin (say, 5%), we can attribute that to the algorithm's efficiency.
- If not, the result for the same can be marked inconclusive.

While the above is more direct, we have used the bounds as indicative of performance and instead performed comparison across all algorithms rather than pairwise analysis.

10.3 Optimizations

All C++ benchmarks compiled with `-Ofast -march=native` flags.

10.4 System Information

- Processor: Intel(R) Core(TM) Ultra 9 185H
- Operating System: Linux 6.14.5
- Language Versions: C++23 (gcc 15.1.1), Python 3.13.3, Bash 5.2.37

11 Conclusion

Through this study, we benchmarked and analyzed the nuances between algorithms, where theoretical complexity didn't always imply algorithmic superiority.
The input size and values also contributed in turn to performance. We studied the running times under a statistical lens with mathematical guarantees on bounds.

11.1 Results

- Random-hash or xor-hash were poor in comparison to polynomial hashing and fnv-1a hash.
- Fenwick trees outperformed both iterative and recursive versions of Segment trees, where the iterative version was only slightly behind.
- The insertion+quicksort hybrid performed best across all input sizes with insertion sort performing equally well on small inputs and quicksort only lagging behind slightly on larger inputs and merge sort lagging by a margin.
- KMP and Z-Function performed similarly on pattern matching with Rabin-Karp lagging behind, while Manacher's algorithm outperformed Rabin-Karp+Binary-Search by a margin.

11.2 Future Scope

The practical study of algorithms can be further extended to other algorithms, and particularly those with hardware specific optimizations.
Sorting[28] and Pattern matching,[29] problems we have considered are already known to benefit from the same.
In addition, work from Sergey Slotin on data structures[30] such as segment trees[31] is another area of study for practical performance.
Deep understanding of systems programming languages, assembly and hardware optimizations are expected to dive into these directions.

References

[1] "GCC Optimization Pragmas," `https://nor-blog.pages.dev/posts/2021-10-26-gcc-optimization-pragmas/`, [Online; accessed 15-May-2025].

[2] "Avalanche effect," *Wikipedia*, `https://en.wikipedia.org/wiki/Avalanche_effect`, [Online; accessed 15-May-2025].

[3] "String Hashing," *CP-Algorithms*, `https://cp-algorithms.com/string/string-hashing.html`, [Online; accessed 15-May-2025].

[4] "FNV hash function," *Wikipedia*, `https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function`, [Online; accessed 15-May-2025].

[5] "Universal hashing," *Wikipedia*, `https://en.wikipedia.org/wiki/Universal_hashing`, [Online; accessed 15-May-2025].

[6] "Fenwick Tree," *CP-Algorithms*, `https://cp-algorithms.com/data_structures/fenwick.html`, [Online; accessed 15-May-2025].

[7] "Segment Tree," *CP-Algorithms*, `https://cp-algorithms.com/data_structures/segment_tree.html`, [Online; accessed 15-May-2025].

[8] "Iterative Segment Tree," *Codeforces Blog*, `https://codeforces.com/blog/entry/18051`, [Online; accessed 15-May-2025].

[9] "Fenwick Tree - Range Minimum Queries," *IO Informatics*, vol. 9, pp. 39–44, 2015, `https://ioinformatics.org/journal/v9_2015_39_44.pdf`, [Online; accessed 15-May-2025].

[10] "Insertion sort," *Wikipedia*, `https://en.wikipedia.org/wiki/Insertion_sort`, [Online; accessed 15-May-2025].

[11] "Merge sort," *Wikipedia*, `https://en.wikipedia.org/wiki/Merge_sort`, [Online; accessed 15-May-2025].

[12] "Quicksort," *Wikipedia*, `https://en.wikipedia.org/wiki/Quicksort`, [Online; accessed 15-May-2025].

[13] "DualPivotQuicksort.java," *OpenJDK*, `https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/DualPivotQuicksort.java`, [Online; accessed 15-May-2025].

[14] "What's the difference of dual pivot quick sort and quick sort?" *Stack Overflow*, https://stackoverflow.com/questions/20917617/whats-the-difference-of-dual-pivot-quick-sort-and-quick-sort, [Online; accessed 15-May-2025].

[15] "KMP (Knuth-Morris-Pratt)," *CP-Algorithms*, https://cp-algorithms.com/string/prefix-function.html, [Online; accessed 15-May-2025].

[16] "Manacher's Algorithm - Finding all sub-palindromes in $O(N)$," *CP-Algorithms*, https://cp-algorithms.com/string/manacher.html, [Online; accessed 15-May-2025].

[17] "Rabin-Karp," *CP-Algorithms*, https://cp-algorithms.com/string/rabin-karp.html, [Online; accessed 15-May-2025].

[18] "Z-function," *CP-Algorithms*, https://cp-algorithms.com/string/z-function.html, [Online; accessed 15-May-2025].

[19] "Sparse Table," *CP-Algorithms*, https://cp-algorithms.com/data_structures/sparse-table.html, [Online; accessed 15-May-2025].

[20] "CS166: Data Structures (Lecture 0)," *Wayback Machine*, Aug. 12, 2023, https://web.archive.org/web/20230812194808/http://web.stanford.edu/class/cs166/lectures/00/Small00.pdf, [Online; accessed 15-May-2025].

[21] "CS166: Data Structures (Lecture 1)," *Wayback Machine*, Apr. 22, 2024, https://web.archive.org/web/20240422013245/https://web.stanford.edu/class/cs166/lectures/01/Small01.pdf, [Online; accessed 15-May-2025].

[22] "Cartesian Tree," *Wikipedia* https://en.wikipedia.org/wiki/Cartesian_tree, [Online; accessed 15-May-2025]

[23] "Method of Four Russians," *Wikipedia*, https://en.wikipedia.org/wiki/Method_of_Four_Russians, [Online; accessed 15-May-2025].

[24] "Sqrt Decomposition," *CP-Algorithms*, https://cp-algorithms.com/data_structures/sqrt_decomposition.html, [Online; accessed 15-May-2025].

[25] "LCA - Binary Lifting," *CP-Algorithms*, https://cp-algorithms.com/graph/lca_binary_lifting.html, [Online; accessed 15-May-2025].

[26] "LCA - Farach-Colton and Bender," *CP-Algorithms*, https://cp-algorithms.com/graph/lca_farachcoltonbender.html, [Online; accessed 15-May-2025].

[27] "LCA - Reduction to RMQ. Linear Algorithm," *CP-Algorithms*, https://cp-algorithms.com/graph/rmq_linear.html, [Online; accessed 15-May-2025].

[28] "Vectorized and performance portable Quicksort," *Google Open Source Blog*, June 2022, https://opensource.googleblog.com/2022/06/Vectorized%20and%20performance%20portable%20Quicksort.html, [Online; accessed 15-May-2025].

[29] "C++ string::find() complexity," *Stack Overflow*, Apr. 26, 2012, https://stackoverflow.com/questions/8869605/c-stringfind-complexity, [Online; accessed 15-May-2025].

[30] "SIMD Algorithms," *Codeforces*, `https://codeforces.com/blog/entry/99790`, [Online; accessed 15-May-2025].

[31] "Even more efficient but not so easy Segment Trees," *Codeforces*, `https://codeforces.com/blog/entry/100454`, [Online; accessed 15-May-2025].