Indian Institute of Technology, Madras

Computational Tools: Algorithms, Data Structures and Programs - ID6105

Kanak Agarwal

August 27, 2025

1 Problem 1 - Space Complexity

The actual and the ideal space/time complexity are indicated in the respective plots of each algorithm. The subsections are inclusive of the code followed by their respective complexity plots.

1.1 Binary Search

```
import math
import matplotlib.pyplot as plt
import sys

# 1 Binary Search Implementation
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    steps = 0

    while low <= high:
        steps += 1
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid, steps
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1, steps

# 2 Collect data for scaling
sizes = [2**k for k in range(4, 20)]  # sizes from 16 to 2^19 (500k)
steps_list = []
log_values = []
space_list = []

for n in sizes:
    arr = list(range(n))
    target = arr[-1]  # worst-case scenario
    _, steps = binary_search(arr, target)
    steps_list.append(steps)
    log_values.append(math.log2(n))

    # measure space usage: list object + integers
    space_used = sys.getsizeof(arr) + sum(sys.getsizeof(x) for x in arr)
    space_list.append(space_used)

    print(f"n={n:<6} steps={steps:<3} log2(n)={math.log2(n):.2f} \
    space={space_used}")

# 3 Plot results: steps and space together
plt.figure(figsize=(8, 6))

# Steps scaling
plt.loglog(sizes, steps_list, 'o-', label="Binary Search Steps (time)")
plt.loglog(sizes, log_values, 's--', label="log_2(n) (time reference)")

# Space scaling
plt.loglog(sizes, space_list, 'd-', label="Space usage (measured)")
plt.loglog(sizes, sizes, 'k--', alpha=0.7, label="O(n) (space reference)")

plt.xlabel("Array size (n)")
plt.ylabel("Steps / Space (bytes)")
plt.title("Binary Search: Time and Space Complexity")
plt.legend()
plt.grid(True, which="both", linestyle="--", alpha=0.6)
plt.show()
```

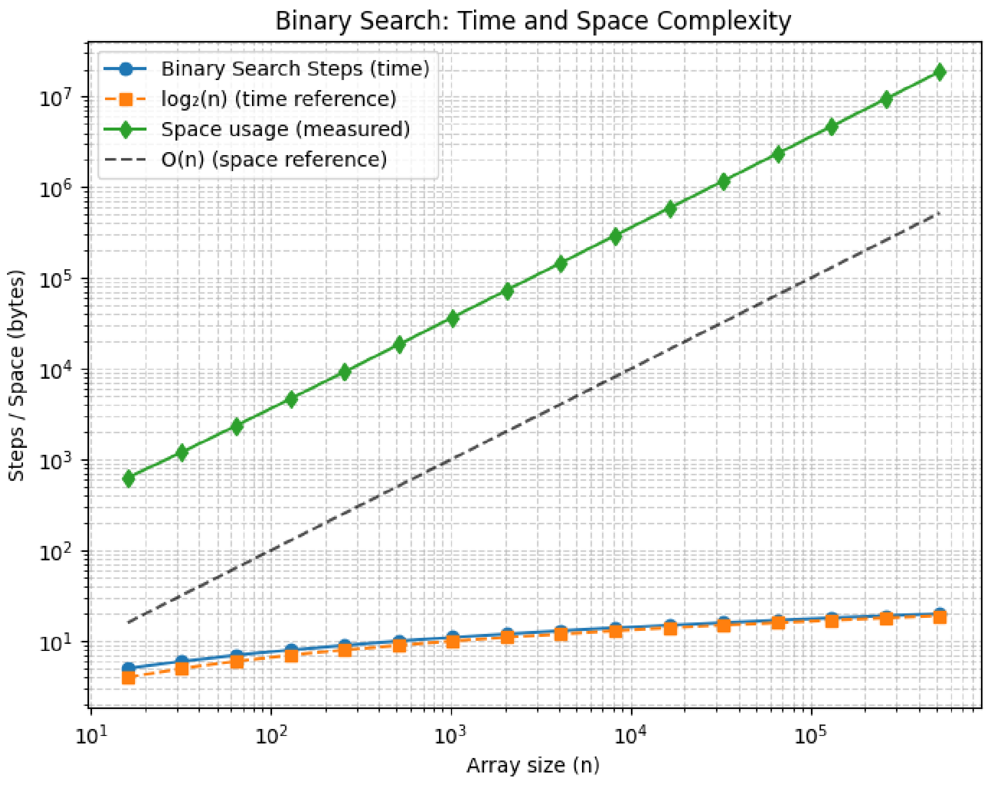

Figure 1: Time/Space Complexity for the Binary Search Algorithm

1.2 Bubble Sort

```
import time
import random
import matplotlib.pyplot as plt
import numpy as np
import sys

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Measure time and space for different input sizes
input_sizes = list(range(100, 3100, 100))
times = []
space_list = []

for n in input_sizes:
    test_array = [random.randint(0, 1000) for _ in range(n)]

    # Measure runtime
    start = time.time()
    bubble_sort(test_array)
    end = time.time()
    times.append(end - start)

    # Measure space: list object + sum of integers
    space_used = sys.getsizeof(test_array) + sum(sys.getsizeof(x) \
    for x in test_array)
    space_list.append(space_used)

# Plotting time + space together
plt.figure(figsize=(8,6))

# Time Complexity
plt.loglog(input_sizes, times, marker='o', label="Runtime (measured)")
plt.loglog(input_sizes, 1e-8*np.array(input_sizes)**2, linestyle='--', \
label=r"$O(n^2)$ (time reference)")

# Space Complexity
plt.loglog(input_sizes, space_list, marker='s', label="Space (measured)")
plt.loglog(input_sizes, input_sizes, linestyle='--', \
label=r"$O(n)$ (space reference)")

plt.title("Bubble Sort: Time and Space Complexity")
plt.xlabel("Input Size (n)")
plt.ylabel("Time (seconds) / Space (bytes)")
plt.legend()
plt.grid(True, which="both", linestyle="--", alpha=0.6)
plt.show()
```

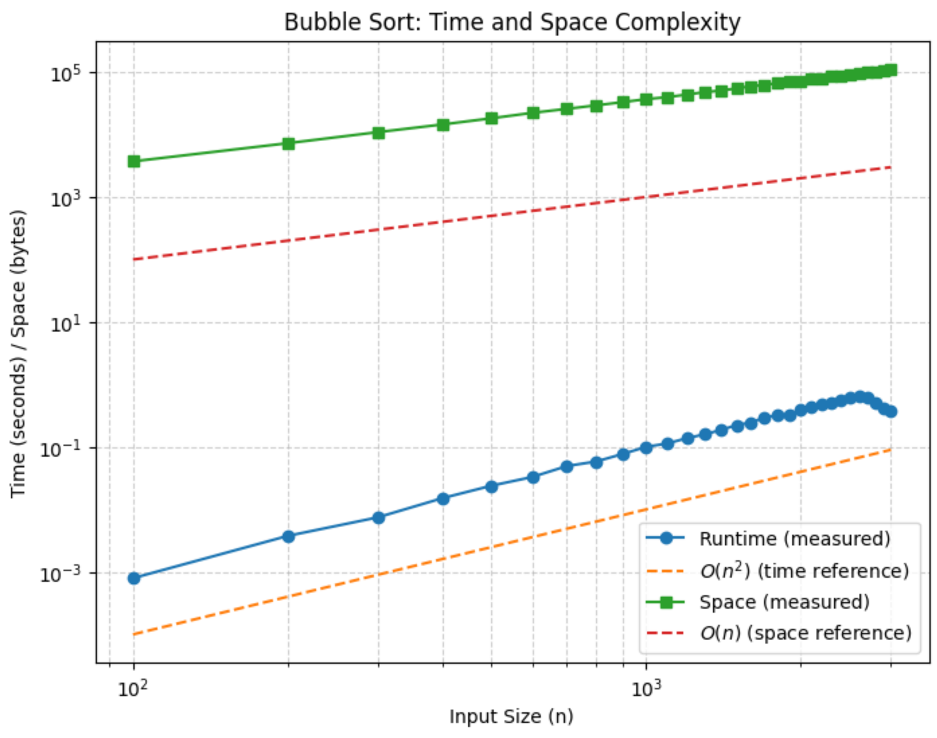

Figure 2: Time/Space Complexity for the Bubble Sort Algorithm

1.3 Fast Fourier Transform (FFT)

```
import numpy as np
import matplotlib.pyplot as plt
import time
import sys

# Input sizes: powers of 2 from 2^8 to 2^20
sizes = 2 ** np.arange(8, 22)
times = []
space_list = []

# Measure FFT time and space for each size
for N in sizes:
    x = np.random.rand(N)

    # Measure runtime
    start = time.perf_counter()
    fft_result = np.fft.fft(x)
    end = time.perf_counter()
    times.append(end - start)

    # Measure space: input array + output array + small constants
    space_used = sys.getsizeof(x) + sys.getsizeof(fft_result)
    space_list.append(space_used)

# Plot results: runtime + space
plt.figure(figsize=(8, 6))
```

```
# Time Complexity
plt.loglog(sizes, times, 'o-', label="FFT runtime (measured)")
plt.loglog(sizes, times[0] * sizes * np.log2(sizes) / (sizes[0] * \
np.log2(sizes[0])), 'r--', label=r"$O(N \log N)$ (time reference)")

# Space Complexity
plt.loglog(sizes, space_list, 's-', label="Space usage (measured)")
plt.loglog(sizes, sizes*sys.getsizeof(float()), 'g--', label=r"$O(N)$ \
(space reference)")

plt.xlabel("Input size N")
plt.ylabel("Time (s) / Space (bytes)")
plt.title("1D FFT: Time and Space Complexity")
plt.legend()
plt.grid(True, which="both", ls="--")
plt.tight_layout()
plt.show()
```

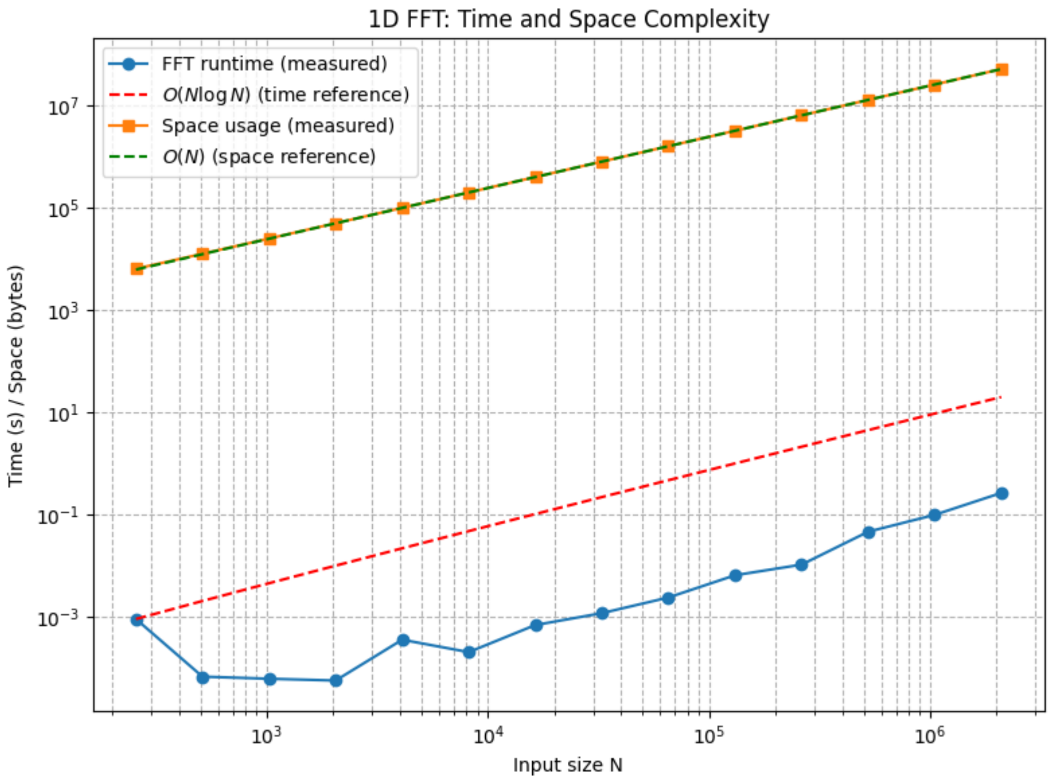

Figure 3: Time/Space Complexity for the FFT Algorithm

1.4 Matrix Multiplication

```
import numpy as np
import time
import matplotlib.pyplot as plt
import sys

# Matrix sizes to test
sizes = list(range(100, 1000, 100))
times = []
```

```
space_list = []
# Measure time and space for multiplying n x n matrices
for n in sizes:
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    start = time.time()
    # Standard matrix multiplication
    C = np.dot(A, B)
    end = time.time()
    times.append(end - start)
    # Measure space: A + B + C
    space_used = sys.getsizeof(A) + sys.getsizeof(B) + sys.getsizeof(C)
    space_list.append(space_used)

# Plotting time + space together
plt.figure(figsize=(8,6))

# Time Complexity
plt.loglog(sizes, times, marker='o', label="Runtime (measured)")
plt.loglog(sizes, 1e-10*np.array(sizes)**3, \
linestyle='--', label=r"$O(n^3)$ (time reference)")

# Space Complexity
plt.loglog(sizes, space_list, marker='s', label="Space (measured)")
plt.loglog(sizes, np.array(sizes)**2 * sys.getsizeof(float()), 'k--', \
label=r"$O(n^2)$ (space reference)")
plt.title("Matrix Multiplication: Time and Space Complexity")
plt.xlabel("Matrix size (n x n)")
plt.ylabel("Time (s) / Space (bytes)")
plt.legend()
plt.grid(True, which="both", linestyle="--", alpha=0.6)
plt.show()
```

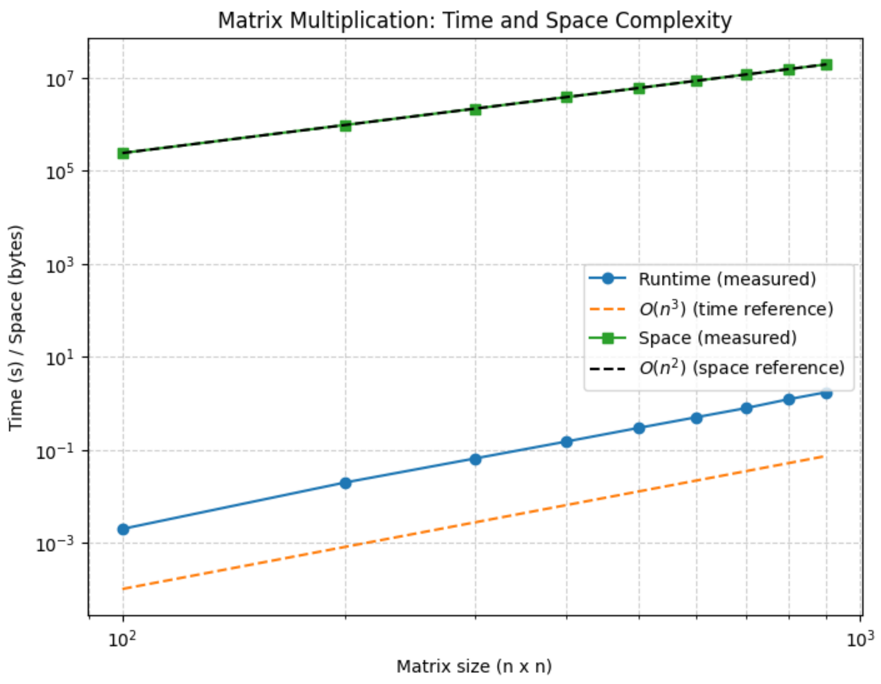

Figure 4: Time/Space Complexity for the Matrix Multiplication Algorithm

1.5 Subset Sum

```
import time
import matplotlib.pyplot as plt
import sys

# 1 Naive recursive subset sum
def subset_sum(nums, target, i=0):
    # Base cases
    if target == 0:
        return True
    if i >= len(nums) or target < 0:
        return False

    # Recursive: include or exclude nums[i]
    include = subset_sum(nums, target - nums[i], i + 1)
    exclude = subset_sum(nums, target, i + 1)
    return include or exclude

# 2 Measure scaling
sizes = list(range(5, 25))  # number of elements (keep small to avoid \
huge runtimes)
times = []
space_list = []

for n in sizes:
    nums = list(range(1, n + 1))  # simple sequence
    target = sum(nums) // 2       # typical hard target

    # Measure runtime
    start = time.time()
    subset_sum(nums, target)
    end = time.time()
    elapsed = end - start
    times.append(elapsed)

    # Measure space: list + recursion stack
    # Worst-case recursion depth = n
    space_used = sys.getsizeof(nums) + n * 64  # assume 64 bytes per \
    call frame
    space_list.append(space_used)

    print(f"n={n}  time={elapsed:.5f} sec  space={space_used} bytes")

# 3 Plot results
plt.figure(figsize=(8, 6))

# Time Complexity
plt.plot(sizes, times, 'o-', label="Naive Subset Sum")
plt.plot(sizes, [1e-7*2**n for n in sizes], 's--', label="2^n (reference)", \
alpha=0.7)
plt.yscale("log")  # exponential growth
```

```
# Space Complexity
plt.plot(sizes, space_list, 'd-', label="Space (measured)")
plt.plot(sizes, [sys.getsizeof(list(range(n))) + n*64 for n in sizes], \
'k--', label="O(n) (space reference)", alpha=0.7)

plt.xlabel("Number of elements (n)")
plt.ylabel("Time (s) / Space (bytes)")
plt.title("Naive Subset Sum: Time and Space Complexity")
plt.grid(True, which="both", linestyle="--", alpha=0.6)
plt.legend()
plt.show()
```

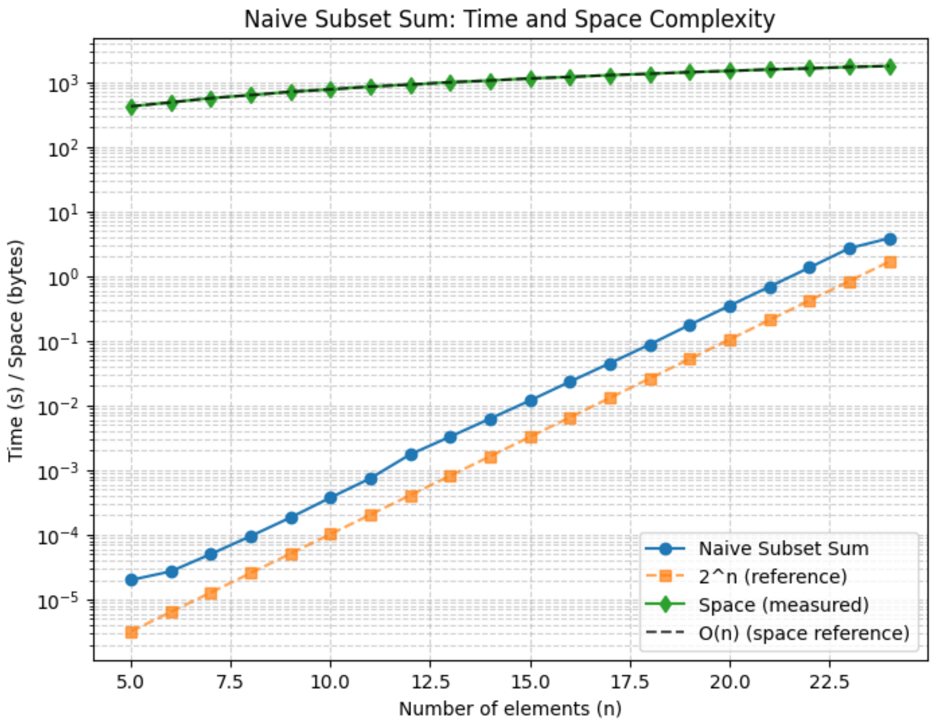

Figure 5: Time/Space Complexity for the Subset Sum Algorithm

2 Problem 2 - Time Complexity Proofs

1. **Prove:** $53n$ is $\Theta(n)$.

   Proof: There exist constants $a = 1$ and $b = 53$ such that for all $n \geq a$,

   $$53n \leq b\,n,$$

   thus, $53n = \mathcal{O}(n)$. There exists the constant $c = 1$ such that for all $n \geq a$,

   $$53n \geq c\,n,$$

   so $53n = \Omega(n)$. Hence, $53n = \Theta(n)$.

2. **Prove:** $n^2\sqrt{n}$ is not $\Omega(n^3)$.

   Proof: By definition, for $f(n)$ to be $\Omega(g(n))$, there must exist a constant $c > 0$ such that,

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

Here, $f(n) = n^2\sqrt{n} = n^{2.5}$ and $g(n) = n^3$. Consider the ratio

$$\frac{f(n)}{g(n)} = \frac{n^{2.5}}{n^3} = \frac{1}{\sqrt{n}}.$$

As $n \to \infty$,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0.$$

Since the ratio tends to 0, for any constant $c > 0$,

$$n^{2.5} < c\,n^3$$

for sufficiently large n. Thus, no such c can satisfy the Ω condition.

3. **Prove:** n^2 is $\Omega(20n)$.

   Proof: By definition, for $f(n)$ to be $\Omega(g(n))$, there must exist a constant $c > 0$ such that,

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

   Here, $f(n) = n^2$ and $g(n) = 20n$. Consider

   $$\frac{f(n)}{g(n)} = \frac{n^2}{20n} = \frac{n}{20}.$$

   As $n \to \infty$, $\frac{n}{20} \to \infty$. Thus, for all $c > 0$,

   $$n^2 \geq 20n$$

4. **Prove:** $2n^4 - 3n^2 + 32n - 5n + 60$ is $\Theta(n^4)$.

   Proof:
   Let $f(n) = 2n^4 - 3n^2 + 27n + 60$ and $g(n) = n^4$.
   For all $n \geq 1$, we have

   $$f(n) = 2n^4 - 3n^2 + 27n + 60 \leq 2n^4 + 3n^2 + 27n + 60 \leq 3n^4$$

   since for large n, $3n^2 + 27n + 60 \leq n^4$. Choose $a = 1$ and $b = 3$, then for all $n \geq a$,

   $$f(n) \leq b\,g(n),$$

   so $f(n) = O(n^4)$. The function $f(n) \to 2n^4 - 3n^2 + 27n + 60 \geq n^4$ for infinitely many integers n. For example, for all $n \geq 1$, $2n^4 - 3n^2 + 27n + 60 \geq n^4$. Thus, $c = 1$, then for infinitely many n,

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

   so $f(n) = \Omega(n^4)$. Since $f(n) = O(n^4)$ and $f(n) = \Omega(n^4)$, by definition,

   $$f(n) = \Theta(n^4).$$

5. **Prove:** $100n^2$ is $O(n^3)$.

   Proof:
   By definition, $f(n)$ is $O(g(n))$ if there exist constants $a, b > 0$ such that for all $n \geq a$,

   $$f(n) \leq b\,g(n).$$

   Here, $f(n) = 100n^2$ and $g(n) = n^3$. For all $n \geq 1$,

   $$100n^2 \leq 100n^3.$$

   Thus, choose $b = 100$ and $a = 1$. Then for all $n \geq a$,

   $$f(n) \leq b\,g(n),$$

   Hence, $100n^2 = O(n^3)$.

6. **Prove:** $2n^3$ is $O(2^n)$.

   Proof:
   By definition, $f(n)$ is $O(g(n))$ if there exist constants $a, b > 0$ such that for all $n \geq a$,
   $$f(n) \leq b\, g(n).$$
   Here, $f(n) = 2n^3$ and $g(n) = 2^n$. Consider the ratio
   $$\frac{f(n)}{g(n)} = \frac{2n^3}{2^n} = \frac{n^3}{2^{n-1}}.$$
   As $n \to \infty$, 2^{n-1} grows exponentially while n^3 grows polynomially, so
   $$\lim_{n\to\infty} \frac{n^3}{2^{n-1}} = 0.$$
   Hence, there exists a constant $b > 0$ (for example, $b = 1$) and an integer a large enough (e.g., $a = 10$) such that for all $n \geq a$,
   $$2n^3 \leq b\, 2^n.$$
   Therefore, $2n^3 = O(2^n)$.

7. **Prove:** $10n^{10}$ is $O(2^n)$.

   Proof:
   By definition, $f(n)$ is $O(g(n))$ if there exist constants $a, b > 0$ such that for all $n \geq a$,
   $$f(n) \leq b\, g(n).$$
   Here, $f(n) = 10n^{10}$ and $g(n) = 2^n$. Consider the ratio
   $$\frac{f(n)}{g(n)} = \frac{10n^{10}}{2^n}.$$
   As $n \to \infty$, 2^n grows exponentially while $10n^{10}$ grows polynomially, so
   $$\lim_{n\to\infty} \frac{10n^{10}}{2^n} = 0.$$
   Hence, there exists a constant $b > 0$ (for example, $b = 1$) and an integer a large enough (e.g., $a = 50$) such that for all $n \geq a$,
   $$10n^{10} \leq b\, 2^n.$$
   Therefore, $10n^{10} = O(2^n)$.

8. **Prove:** n^{10} is $\Omega(1000n)$.

   Proof:
   By definition, $f(n)$ is $\Omega(g(n))$ if there exists a constant $c > 0$ such that
   $$f(n) \geq c\, g(n)$$
   for infinitely many integers n. Here, $f(n) = n^{10}$ and $g(n) = 1000n$. Consider
   $$\frac{f(n)}{g(n)} = \frac{n^{10}}{1000n} = \frac{n^9}{1000}.$$
   As $n \to \infty$, $\frac{n^9}{1000} \to \infty$. Hence, choose $c = 1$. Then
   $$n^{10} \geq 1000n$$
   for all $n \geq 1000^{1/9} \approx 2$, which is infinitely many integers. Therefore, n^{10} is $\Omega(1000n)$.

9. **Prove:** $n^3 + (\log_2 n)^{10}$ is $O(n^3)$.

   Proof:
   By definition, $f(n)$ is $O(g(n))$ if there exist constants $a, b > 0$ such that for all $n \geq a$,
   $$f(n) \leq b\, g(n).$$
   Here, $f(n) = n^3 + (\log_2 n)^{10}$ and $g(n) = n^3$. For all $n \geq 2$, we have
   $$(\log_2 n)^{10} \leq n^3,$$
   because $\log_2 n$ grows much slower than n and any fixed power of $\log_2 n$ is dominated by n^3 for sufficiently large n. Hence,
   $$f(n) = n^3 + (\log_2 n)^{10} \leq n^3 + n^3 = 2n^3.$$
   Choose $b = 2$ and $a = 2$. Then for all $n \geq a$,
   $$f(n) \leq b\, g(n),$$
   Thus, $f(n) = O(n^3)$.

10. **Prove:** $n\sqrt{n}\log_2(n)$ is $O(n^2)$.

   Proof:
   By definition, $f(n)$ is $O(g(n))$ if there exist constants $a, b > 0$ such that for all $n \geq a$,
   $$f(n) \leq b\, g(n).$$
   Here, $f(n) = n^{3/2}\log_2 n$ and $g(n) = n^2$. Consider the ratio
   $$\frac{f(n)}{g(n)} = \frac{n^{3/2}\log_2 n}{n^2} = \frac{\log_2 n}{\sqrt{n}}.$$
   As $n \to \infty$, $\frac{\log_2 n}{\sqrt{n}} \to 0$, because $\sqrt{n}$ grows faster than $\log_2 n$. Hence, there exists a constant $b > 0$ (e.g., $b = 1$) and an integer a large enough (e.g., $a = 2$) such that for all $n \geq a$,
   $$n\sqrt{n}\log_2(n) \leq b\, n^2.$$
   Therefore, $n\sqrt{n}\log_2(n) = O(n^2)$.

3 Problem 3 - *k*-nary Search

3.1 Ternary Search

```
import time
import random
import matplotlib.pyplot as plt
import math

def ternary_search(arr, target, left, right):
    if right >= left:
        # Divide the array into three parts
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3
        # Check if the target is at any mid
        if arr[mid1] == target:
            return mid1
        if arr[mid2] == target:
            return mid2

        # If target lies in the first part
        if target < arr[mid1]:
            return ternary_search(arr, target, left, mid1 - 1)

        # If target lies in the third part
        elif target > arr[mid2]:
            return ternary_search(arr, target, mid2 + 1, right)

        # Else it lies in the middle part
        else:
            return ternary_search(arr, target, mid1 + 1, mid2 - 1)

    return -1  # Target not found

sizes = [10**3, 10**4, 10**5, 10**6, 10**7, 10**8]  # array sizes
times = []

for n in sizes:
    arr = list(range(n))  # sorted array
    target = random.choice(arr)  # pick a random target

    start = time.time()
    index = ternary_search(arr, target, 0, n - 1)
    print(index,target)
    end = time.time()

    times.append(end - start)

log_n = [math.log(n) for n in sizes] # ideal scaling
scale = max(times) / max(log_n) # scale factor
log_n_scaled = [x * scale for x in log_n] # Normalize the log(n) values

plt.figure(figsize=(8, 5))
plt.plot(sizes, log_n_scaled, marker='s', linestyle='--', linewidth=2, \
label="O(log n) reference")
plt.plot(sizes, times, marker='o', linewidth=2, label="Measured Runtime")
plt.xscale("log")
plt.xlabel("Array size (n)")
plt.ylabel("Time (s))")
plt.title("Ternary Search Time Complexity")
plt.grid(True, which="both", linestyle="--", alpha=0.7)
plt.legend()
plt.show()
```

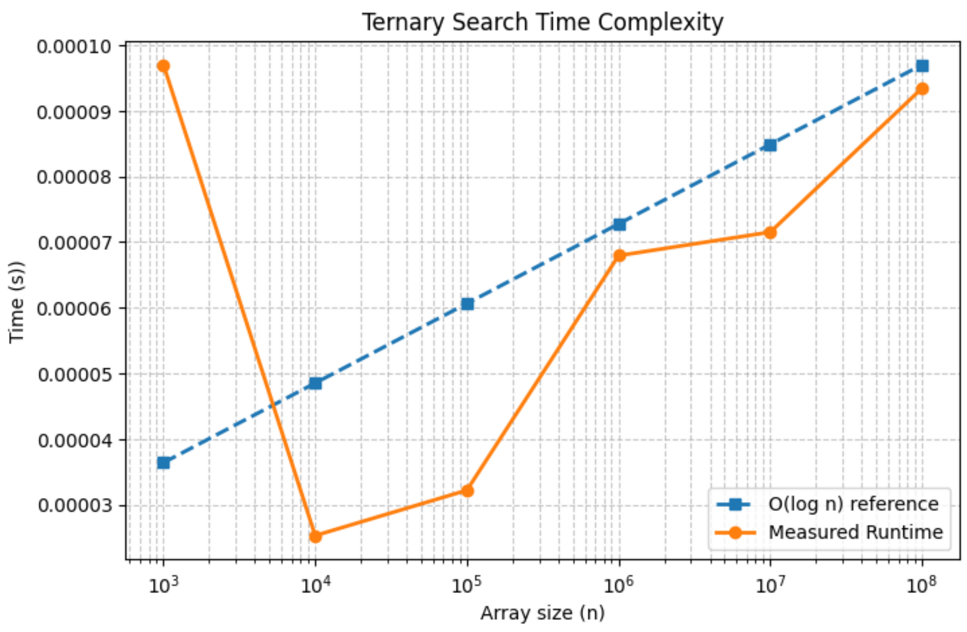

Figure 6: Time Complexity for the Ternary Search Algorithm

3.2 k-nary Search for Single k

```
import time
import random
import matplotlib.pyplot as plt
import math

def knary_search(arr, target, left, right,k):
    if right >= left:
        # Divide the array into k parts
        segment = (right - left) // k
        mids = [left + i * segment for i in range(1, k)] # k-1 midpoints

        # Check all midpoints
        for mid in mids:
            if arr[mid] == target:
                return mid

        # Find the segment where target lies
        if target < arr[mids[0]]:
            return knary_search(arr, target, left, mids[0]-1, k)
        for i in range(1, len(mids)):
            if target < arr[mids[i]]:
                return knary_search(arr, target, mids[i-1]+1, mids[i]-1, k)

        # Target in last segment
        return knary_search(arr, target, mids[-1]+1, right, k)

    return -1  # Target not found

sizes = [10**3, 10**4, 10**5, 10**6, 10**7, 10**8]  # array sizes
times = []
```

```
for n in sizes:
    arr = list(range(n))  # sorted array
    target = random.choice(arr)  # pick a random target

    start = time.time()
    index = knary_search(arr, target, 0, n - 1, 5)
    print(index,target)
    end = time.time()

    times.append(end - start)

log_n = [math.log(n) for n in sizes]
scale = max(times) / max(log_n)
log_n_scaled = [x * scale for x in log_n]

plt.figure(figsize=(8, 5))
plt.plot(sizes, log_n_scaled, marker='s', linestyle='--', linewidth=2, \
label="O(log n) reference")
plt.plot(sizes, times, marker='o', linewidth=2, label="Measured Runtime")
plt.xscale("log")
plt.xlabel("Array size (n)")
plt.ylabel("Time (s))")
plt.title("k-nary Search Time Complexity for k = 5")
plt.grid(True, which="both", linestyle="--", alpha=0.7)
plt.legend()
plt.show()
```

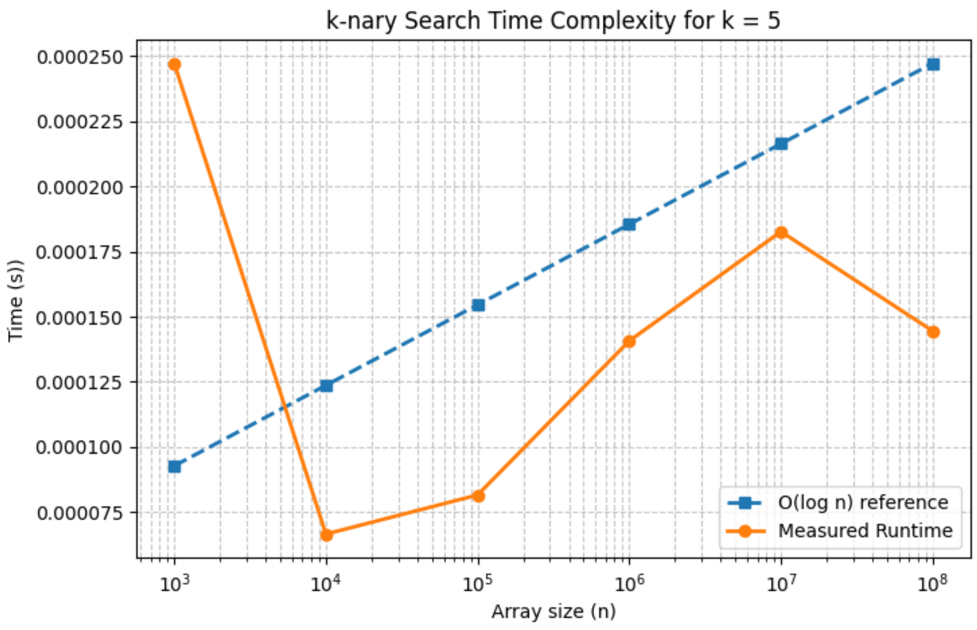

Figure 7: Time Complexity for the k-nary Search Algorithm for $k = 5$

3.3 k-nary Search for Multiple k

```
import time
import random
import matplotlib.pyplot as plt
import math

def knary_search(arr, target, left, right,k):
    if right >= left:
        # Divide the array into k parts
        segment = (right - left) // k
        mids = [left + i * segment for i in range(1, k)] # k-1 midpoints

        # Check all midpoints
        for mid in mids:
            if arr[mid] == target:
                return mid

        # Find the segment where target lies
        if target < arr[mids[0]]:
            return knary_search(arr, target, left, mids[0]-1, k)
        for i in range(1, len(mids)):
            if target < arr[mids[i]]:
                return knary_search(arr, target, mids[i-1]+1, mids[i]-1, k)

        # Target in last segment
        return knary_search(arr, target, mids[-1]+1, right, k)

    return -1  # Target not found

# Benchmark
sizes = [10**3, 10**4, 10**5, 10**6, 10**7, 10**8]
ks = [2, 3, 4, 5]  # different k values
results = {}

for k in ks:
    times = []
    for n in sizes:
        arr = list(range(n))
        target = random.choice(arr)
        start = time.time()
        knary_search(arr, target, 0, n-1, k)
        end = time.time()
        times.append(end - start)
    results[k] = times

# Plotting
plt.figure(figsize=(10,6))
for k in ks:
    plt.plot(sizes, results[k], marker='o', linewidth=2, label=f"k={k}")

# Plot O(log n) reference
log_n = [math.log(n) for n in sizes]
scale = max(max(results[k]) for k in ks) / max(log_n)
log_n_scaled = [x*scale for x in log_n]
plt.plot(sizes, log_n_scaled, 'k--', label='O(log n) reference')
plt.xscale('log')
plt.xlabel("Array size (n)")
plt.ylabel("Time (s)")
plt.title("k-nary Search Complexity for Different k")
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.legend()
plt.show()
```

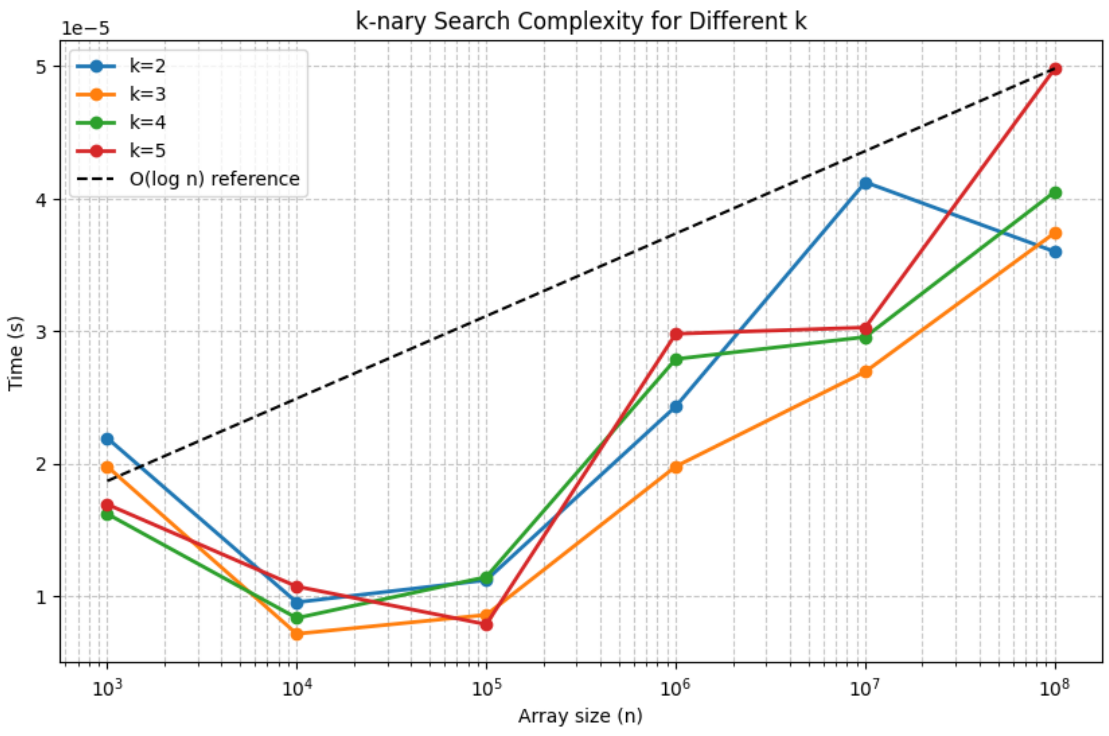

Figure 8: Time Complexity for the k-nary Search Algorithm for different values of k

3.4 Complexity of the 5-nary Search Algorithm

Proof:

The recurrence relation is given by,

$$T(n) = T\left(\frac{n}{k}\right) + \mathcal{O}(k-1) \tag{1}$$

where,

$$T\left(\frac{n}{k}\right) \rightarrow \text{Splitting of the Array} \tag{2}$$

and,

$$\mathcal{O}(k-1) \approx \mathcal{O}(k) \rightarrow \text{Checking the Midpoints} \tag{3}$$

The master theorem states,

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{4}$$

therefore,

$$\begin{gathered} a = 1 \rightarrow \text{Select one Segment and Recurse into it!} \\ b = 5 \rightarrow \text{Number of Divisions, } k = 5 \\ f(n) = \mathcal{O}(k) = \mathcal{O}(1) \rightarrow \text{for fixed } k \end{gathered} \tag{5}$$

thus, from the master theorem,

$$T(n) = \mathcal{O}(\log_5 n) \approx \mathcal{O}(\log n) \tag{6}$$

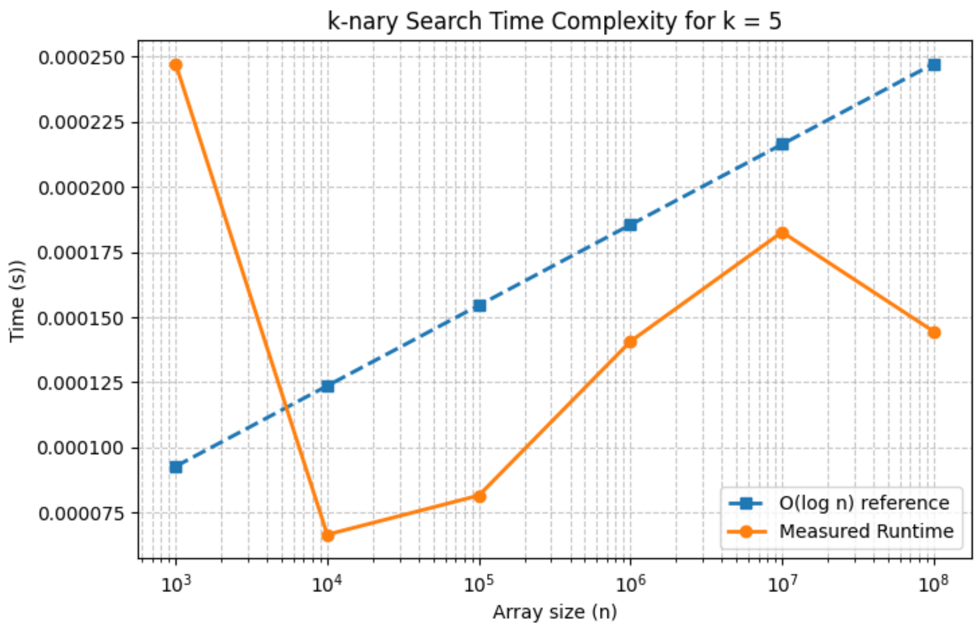

Figure 9: Time Complexity for the k-nary Search Algorithm for $k = 5$

3.5 Number of Operations (N) required for the k-nary Search Algorithm

Proof:

The number of operations per recursion is given by,

$$N(\text{Comparison}) = k - 1 \rightarrow \text{Comparison at } k - 1 \text{ Midpoints} \tag{7}$$

The number of recursive steps are given by,

$$N(\text{Recursion}) = \log_k n \rightarrow \text{Recursion Depth} \tag{8}$$

Thus, total number of operations,

$$N(\text{Total}) = (k - 1)\log_k n \rightarrow \text{Worst Case!} \tag{9}$$

4 Problem 4 - Perfect Square Dissection

4.1 Algorithm

```
1: Input: order, a
2: Output: A list of ‘order‘ distinct integers whose squares sum to $a^2$, or None
3: solution ← []
4: used ← {}
5: function BACKTRACK(remaining_area, remaining_count, max_side)
6:     if remaining_count = 0 then
7:         if remaining_area = 0 then
8:             return True
9:         else
10:            return False
11:        end if
```

```
12:     end if
13:     for $s = \min(\text{max\_side}, \lfloor\sqrt{remaining\_area}\rfloor)$ downto 1 do
14:         if $s \notin used$ and $s < a$ then
15:             Append $s$ to solution
16:             Add $s$ to used
17:             if BACKTRACK(remaining_area - $s^2$, remaining_count - 1, $s - 1$) then
18:                 return True
19:             end if
20:             Remove last element from solution
21:             Remove $s$ from used
22:         end if
23:     end for
24:     return False
25: end function
26: if BACKTRACK(()$a^2$, order, $a - 1$) then
27:     return solution
28: else
29:     return None
30: end if
```

4.2 Code

```
import math
import time
import sys
import matplotlib.pyplot as plt

def perfect_square_partition(order, a):
    solution = []
    used = set()

    def backtrack(remaining_area, remaining_count, max_side):
        if remaining_count == 0:
            return remaining_area == 0
        for s in range(min(max_side, int(math.isqrt(remaining_area))), 0, -1):
            if s not in used and s < a:
                solution.append(s)
                used.add(s)
                if backtrack(remaining_area - s*s, remaining_count - 1, s-1):
                    return True
                solution.pop()
                used.remove(s)
        return False

    if backtrack(a*a, order, a-1):
        return solution
    else:
        return None

orders = [3, 5, 6, 8, 10, 12, 14, 16, 18, 20]
a = 100
times = []
```

```
space_usage = []

for order in orders:
    start_time = time.time()
    partition = perfect_square_partition(order, a)
    end_time = time.time()
    elapsed = end_time - start_time
    times.append(elapsed)

    mem_solution = sys.getsizeof(partition) if partition else 0 \
    # Memory of solution
    mem_used = sys.getsizeof(set(partition)) if partition else 0 \
    # Memory of used
    total_mem = mem_solution + mem_used + sys.getsizeof(order) + \
    sys.getsizeof(a)
    space_usage.append(total_mem)

    print(f"Order: {order}, Partition: {partition}, Time: {elapsed:.4f} s,\
    Memory: {total_mem} bytes")

# Time Complexity
plt.figure(figsize=(8,5))
plt.plot(orders, times, marker='o', linestyle='-', linewidth=2, \
label="Measured time")
plt.plot(orders, [1e-5*(2**o) for o in orders], marker='s', linestyle='--', \
label="Reference O(2^n)")
plt.yscale("log")
plt.xlabel("Order (n)")
plt.ylabel("Time (s)")
plt.title(f"Time Complexity (a={a})")
plt.grid(True, which="both", linestyle="--", alpha=0.7)
plt.legend()
plt.show()

# Space Complexity
plt.figure(figsize=(8,5))
plt.plot(orders, space_usage, marker='o', linestyle='-', linewidth=2, \
label="Measured space")
plt.plot(orders, [5*(a)*math.log(o) for o in orders], marker='s', \
linestyle='--', label="Reference O(a log n)")
plt.yscale("log")
plt.xlabel("Order (n)")
plt.ylabel("Memory (bytes)")
plt.title(f"Space Complexity (a={a})")
plt.grid(True, which="both", linestyle="--", alpha=0.7)
plt.legend()
plt.show()
```

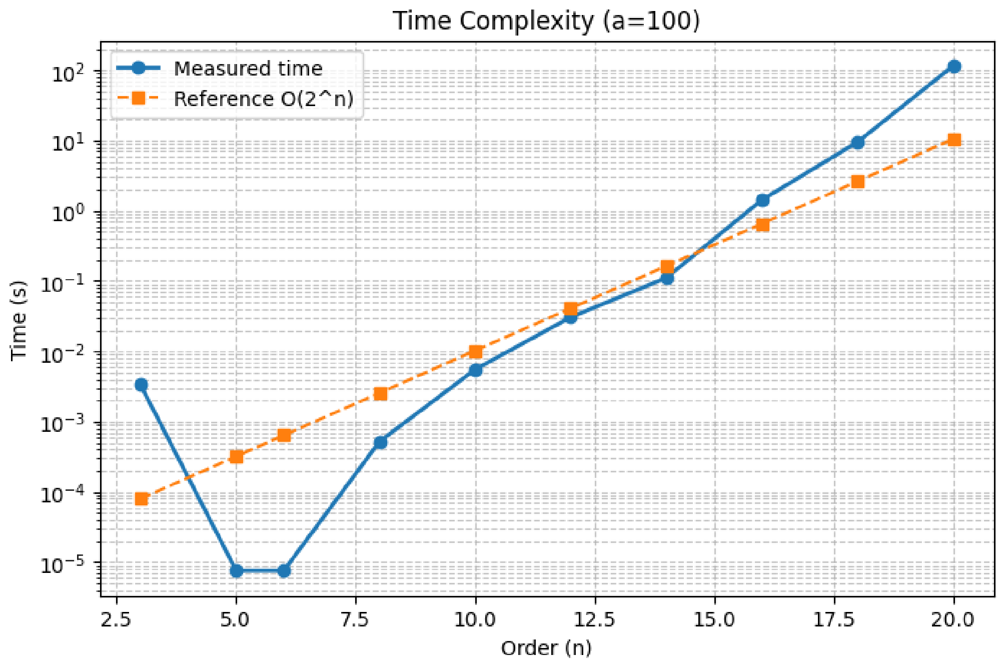

Figure 10: Time Complexity for the Perfect Square Dissection Algorithm

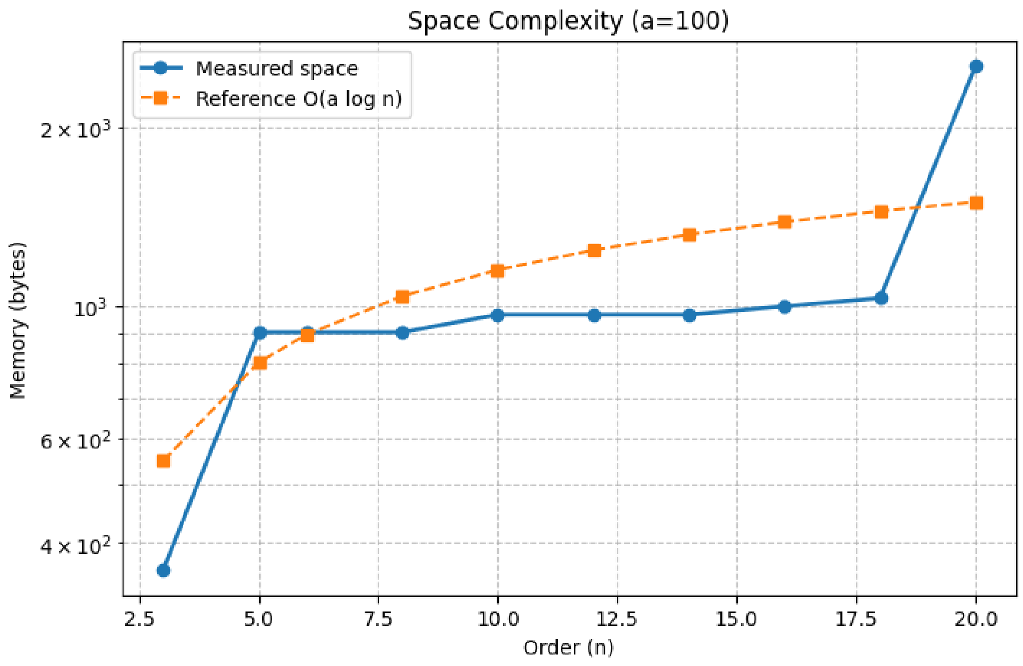

Figure 11: Space Complexity for the Perfect Square Dissection Algorithm