Algorithm Analysis

Abraham J. Reines

April 8, 2024

Task 6.3.1

Given the function $f(n) = 10^{12}n^3 + 10^6n^2 + n + 1$, we analyze its growth rates in terms of Big O, Small o, and Omega compared to $g(n) = n^4$ as n approaches infinity.

Big O Analysis

The Big O notation describes an upper bound of the algorithm, meaning it grows at most at the same rate as $g(n)$. The limit of $\frac{f(n)}{g(n)}$ as n approaches infinity has been found to be 0, which indicates that $f(n)$ grows no faster than $g(n)$ and is in fact slower. Therefore, $f(n)$ is $O(n^4)$.

Big O: True, because $f(n)$ is bounded above by $g(n)$ as n approaches infinity, and since the limit of $\frac{f(n)}{g(n)}$ is 0, $f(n)$ is indeed $O(n^4)$.

Small o Analysis

The Small o notation describes a stricter upper bound, meaning it grows strictly slower than $g(n)$. The analysis shows:

Small o: True, because the limit of $\frac{f(n)}{g(n)}$ as n approaches infinity is 0, which confirms $f(n)$ grows slower than $g(n)$. Thus, $f(n)$ is $o(n^4)$.

Omega Analysis

The Omega notation describes a lower bound, meaning the algorithm grows at least as fast as $g(n)$. The results of our analysis are:

Omega: False, because the limit of $\frac{f(n)}{g(n)}$ as n approaches infinity is 0, which signifies that $f(n)$ does not grow as fast as $g(n)$. Therefore, $f(n)$ is not $\omega(n^4)$.

6.3.2

Application Case:

- 250GB hard disk with 160GB of files.
- CPU with one core, 4GHz clock speed, and 64-bit registers.
- Capable of 4×10^9 8-character comparisons per second.

Search Time Analysis

1. **Single 1KB Signature:**

   The time required to search with a single 1KB signature is computed to be approximately 5 seconds. This duration represents the worst-case scenario, providing a concrete timeframe for completing a single search cycle. It is important to note that the Boyer-Moore algorithm will often perform faster than this worst-case estimate due to its pattern-skipping feature.

2. **Database of 1 Million 1KB Signatures:**

   Scaling up the search to account for a database of 1 million 1KB signatures, the total time estimated for the search is approximately 57.87 days, assuming a worst-case scenario where each signature search is independent and linear. The actual performance of the Boyer-Moore algorithm is expected to be better due to its sublinear average-case performance.

3. **Improving Efficiency:**

   To further improve the efficiency of malware/virus detection through string matching, we can utilize parallel processing by distributing the search workload across multiple CPU cores or even different machines. Additionally, implementing more advanced string matching algorithms such as Aho-Corasick, which is designed for searching multiple patterns simultaneously, can significantly reduce the time complexity. Optimizing the algorithm to take advantage of the preprocessing of the virus signatures and employing heuristic methods to skip unlikely sections of data can also enhance performance.

Scripting

```
"""
Author: Abraham Reines
Created: Mon Apr  1 12:55:40 PDT 2024
Modified: Wed Apr  3 13:29:07 PDT 2024
Name of file: CheckingLimits.py
"""

from sympy import symbols, limit, oo

n = symbols('n')
f = 10**12 * n**3 + 10**6 * n**2 + n + 1

def AsymptoteTime(f, g, n):
    """
    Analyze asymptote behavior of f(n) against g(n) using Big O, small o, and Omega

    Parameters:
    f (sympy expression): asymptotic f(n) should be analyzed
    g (sympy expression): g(n) comparison
    n (sympy symbol): variable with respect to limit computed
    """

    #  limit for conditions
    asymptoteLimit = limit(f/g, n, oo)

    #  Big O condition
    BigO = asymptoteLimit.is_finite
    BigO_result = "True" if BigO else "False"
    print(f"Big O: {BigO_result}, because the limit of f(n)/g(n) as n approaches
    ↪ infinity is {asymptoteLimit}, which is finite.")

    # small o condition
    SmallO = asymptoteLimit == 0
    SmallO_result = "True" if SmallO else "False"
    print(f"Small o: {SmallO_result}, because the limit of f(n)/g(n) as n approaches
    ↪ infinity is {asymptoteLimit}, which confirms f(n) grows slower than g(n).")

    # Omegas condition for inverse of limit
    OmegaLimit = limit(g/f, n, oo)
    omega = OmegaLimit == 0
    omega_result = "True" if omega else "False"
    print(f"Omega: {omega_result}, because the limit of g(n)/f(n) as n approaches
    ↪ infinity is {OmegaLimit}, which confirms f(n) does not grow as fast as g(n).")

# g(n) is n**4 for comparison
AsymptoteTime(f, n**4, n)
```

Listing 1: Content from CheckingLimits.py

```python
"""
Author: Abraham Reines
Created: Mon Apr  1 13:16:18 PDT 2024
Modified: Wed Apr  3 13:35:50 PDT 2024
Name of file: BackofEnvelope.py
"""

def What_is_the_search_time(BytesForFile, SignatureForBytes, HowManyComparisons):
    """
    time to search file with signature size using the Boyer-Moore algorithm

    Parameters:
    BytesForFile (int): bytes in the file
    SignatureForBytes (int): bytes for virus signature
    HowManyComparisons (int): 8-character comparisons of CPU in a second

    Returns:
    float: time for search the file (s)
    """
    # this is worst case
    Comparisons = BytesForFile
    TimeSec = Comparisons / (HowManyComparisons * 8)

    return TimeSec

def Readable_seconds(seconds):
    """
    Convert seconds to readable format

    Parameters:
    seconds (float): seconds

    Returns:
    str: days or years
    """
    # Constants
    SecondsForDay = 86400
    DaysForYear = 365.25

    if seconds < SecondsForDay:
        return f"{seconds} seconds"
    elif seconds < SecondsForDay * DaysForYear:
        days = seconds / SecondsForDay
        return f"{days:.2f} days"
    else:
        years = seconds / (SecondsForDay * DaysForYear)
        return f"{years:.2f} years"

# Constants
BytesForFile = 160 * (10**9)  # 160GB
SignatureForBytes = 1 * (2**10)  # 1KB
HowManyComparisons = 4 * (10**9)  # 8-character comparisons per second

#  1KB signature
OneSig = What_is_the_search_time(BytesForFile, SignatureForBytes, HowManyComparisons)
readable_OneSig = Readable_seconds(OneSig)

#  1 million signatures
MillSig = OneSig * 1_000_000
readable_MillSig = Readable_seconds(MillSig)

# results
print(f"Time for 1KB signature: {readable_OneSig}")
print(f"Time for 1 million 1KB signatures: {readable_MillSig}")
```

Listing 2: Content from BackofEnvelope.py

References

Academic Integrity Pledge

"This work complies with the JMU honor code. I did not give or receive unauthorized help on this assignment."