CS – 600 – Advanced Algorithms - Homework 1

Komal Wavhal (20034443)

Table of Contents

Reinforcement: 1.6.7, 1.6.9, 1.6.22, 1.6.23, 1.6.24, 1.6.32

Creativity: 1.6.36, 1.6.39, 1.6.52, 1.6.62

Application: 1.6.70, 1.6.77

Exercise 1.6.7

Order the following list of functions by Big-O notation. Group together (for example, by underlining) those functions that are Big-Θ of one another.

$$6n \log n,\ 2^{100},\ \log \log n,\ \log^2 n,\ 2^{\log n},$$
$$2^{2^n},\ \lceil \sqrt{n} \rceil,\ n^{0.01},\ \frac{1}{n},\ 4n^{3/2},$$
$$3n^{0.5},\ 5n,\ \lfloor 2n \log^2 n \rfloor,\ 2^n,\ n \log_4 n,$$
$$4^n,\ n^3,\ n^2 \log n,\ 4^{\log n},\ \sqrt{\log n}.$$

Answer:

The functions ordered by their asymptotic growth rates (Big-O notation) and grouped by Big-Θ are as follows:

1. Smallest Growth Rate ($O(1)$):

$$\frac{1}{n},\ n^{0.01},\ 2^{100}$$

2. Logarithmic Growth:

$$\log\log n,\ \sqrt{\log n}$$

3. Logarithmic-Squared Growth:

$$\log^2 n$$

4. Linear Growth:

$$3n^{0.5},\ 5n,\ \lceil\sqrt{n}\rceil$$

5. Linearithmic Growth ($n\log n$):

$$6n\log n,\ n\log_4 n,\ \lfloor 2n\log^2 n\rfloor$$

6. Quadratic Growth:

$$n^2\log n$$

7. Polynomial Growth:

$$n^3,\ 4n^{3/2}$$

8. Exponential Growth:

$$2^n,\ 4^{\log n},\ 2^{\log n}$$

9. Larger Exponential Growth:

$$4^n$$

10. Dominant Growth ($O(2^{2^n})$):

$$2^{2^n}$$

$$\frac{1}{n} < n^{0.01},\ 2^{100} < \log\log n < \sqrt{\log n} < \log^2 n < 5n,\ 3n^{0.5},\ \lceil\sqrt{n}\rceil <$$

$$6n\log n,\ n\log_4 n,\ \lfloor 2n\log^2 n\rfloor < n^2\log n < n^3,\ 4n^{3/2} < 2^n,\ 4^{\log n},\ 2^{\log n} <$$

$$4^n < 2^{2^n}.$$

Exercise 1.6.9

Bill has an algorithm, find2D, to find an element x in an n × n array A. The algorithm find2D iterates over the rows of A and calls the algorithm arrayFind, of Algorithm 1.3.1, on each one, until x is found or it has searched all rows of A. What is the worst-case running time of find2D in terms of n? Is this a linear-time algorithm? Why or why not?

Algorithm 1.3.1: `arrayFind`
`arrayFind(x, A)`
Input: An element x and an n-element array, A.
Output: The index i such that $x = A[i]$, or -1 if no element of A is equal to x.

```
1: i ← 0
2: while i < n do
3:     if x = A[i] then
4:         return i
5:     else
6:         i ← i + 1
7:     end if
8: end while
9: return −1
```

The worst-case running time of find2D is O(n2). This is because the algorithm goes through all n rows of the n × n array. For each row, it uses array Find, which takes O(n) time in the worst case. So, the total time for find2D is:

n × O(n) = O(n2).

find2D is not a linear-time algorithm. A linear-time algorithm means the time taken should be proportional to the total number of elements in the array (n2). However, find2D takes time proportional to n2 because it performs a quadratic amount of work based on the array's size n (the number of rows or columns).

In simple terms, find2D does not process all elements in one straight pass—it processes each row one by one, which makes it slower than a linear-time algorithm.

Exercise 1.6.22

Show that n is o(n log n).

We say that n is $o(n \log n)$ if, for any constant $c > 0$,
there exists a constant $n_0 \geq 0$ such that:

$$n < c \cdot n \log n, \quad \text{for all } n \geq n_0.$$

Simplifying the inequality:

$$1 < c \cdot \log n, \quad \text{or equivalently, } \frac{1}{c} < \log n.$$

To satisfy this condition, n must be large enough such that $\log n > \frac{1}{c}$.
Using the properties of logarithms, we solve for n:

$$n > 2^{1/c}, \quad \text{(assuming that the base of the logarithm is 2).}$$

To ensure the inequality holds for all $n \geq n_0$, we choose:

$$n_0 = 2^{1/c} + 1.$$

This choice ensures that:

$$\log n_0 = \log(2^{1/c} + 1) > \log(2^{1/c}) = \frac{1}{c}.$$

Thus, for all $n \geq n_0$, the condition $n < c \cdot n \log n$ holds true.
Therefore, n is $o(n \log n)$ as per the formal definition of little-o notation.

Exercise 1.6.23 (8 points)

Show that n2 is ω(n).

We say that a function $f(n)$ is $\omega(g(n))$ if for any constant $c > 0$,
there exists a constant $n_0 \geq 0$ such that:

$$f(n) > c \cdot g(n), \quad \text{for all } n \geq n_0.$$

Here, let $f(n) = n^2$ and $g(n) = n$.
We need to show that for any constant $c > 0$,
there exists an n_0 such that:

$$n^2 > c \cdot n, \quad \text{for all } n \geq n_0.$$

1. Simplifying the inequality:

$$\frac{n^2}{n} > c, \quad \text{or equivalently, } n > c.$$

2. Choosing n_0: To ensure $n > c$, we choose:

$$n_0 = c + 1.$$

For all $n \geq n_0$, the inequality $n > c$ holds true, which implies:

$$n^2 > c \cdot n.$$

Therefore, By the definition of little-ω notation, n^2 grows asymptotically faster than n, and we conclude:

$$n^2 \text{ is } \omega(n).$$

Exercise 1.6.24

show that n3 log n is Ω(n3).

By the formal definition of Big-Ω notation:
A function $f(n)$ is $\Omega(g(n))$ if there exist positive constants $c > 0$ and $n_0 \geq 0$ such that:

$$f(n) \geq c \cdot g(n), \quad \text{for all } n \geq n_0.$$

Here, let $f(n) = n^3 \log n$ and $g(n) = n^3$.
We need to show that:

$$n^3 \log n \geq c \cdot n^3, \quad \text{for some } c > 0 \text{ and all } n \geq n_0.$$

1. Simplifying the inequality:

$$\frac{n^3 \log n}{n^3} \geq c, \quad \text{or equivalently, } \log n \geq c.$$

2. Choosing n_0: To satisfy $\log n \geq c$, choose:

$$n_0 = 2^c, \quad \text{(assuming the logarithm is base 2)}.$$

3. Verify the condition: For $n \geq n_0$, we have:

$$\log n \geq \log n_0 = c.$$

Thus:

$$n^3 \log n \geq c \cdot n^3.$$

Since we found constants $c > 0$ and $n_0 \geq 0$ such that $n^3 \log n \geq c \cdot n^3$ for all $n \geq n_0$, it follows that:

$$n^3 \log n \text{ is } \Omega(n^3).$$

Exercise 1.6.32

Suppose we have a set of n balls, and we choose each one independently with probability $1/n^{1/2}$ to go into a basket. Derive an upper bound on the probability that there are more than $3n^{1/2}$ balls in the basket.

Solution:

Let Y be a random ball that is chosen to go into the basket.

$\mu = E(Y) = n * n^{1/2} = n^{1/2}$

For $\delta=2$, the upper bound will be

$P(X >= (1+\delta)\ \mu) = P\ (X >= 3n^{1/2}) < [(e^{\delta})/(1+\delta)^{(1+\delta)}]^{\mu}$ (Using Chernoff bounds)

$= [e^2 / 3^3]^{\sqrt{n}}$

Therefore, the upper bound on the required probability is $[e^2 / 3^3]^{\sqrt{n}}$

Exercise 1.6.36

Question: Suppose we have a set of n balls and we choose each one independently with probability $\frac{1}{\sqrt{n}}$ to go into a basket. Derive an upper bound on the probability that there are more than $3\sqrt{n}$ balls in the basket.

Answer:

Let X represent the total number of balls in the basket.
Each ball is chosen independently with probability $\frac{1}{\sqrt{n}}$.
The expected value of X is:

$$\mu = \mathbb{E}[X] = n \cdot \frac{1}{\sqrt{n}} = \sqrt{n}.$$

Using Chernoff bounds, we have:

$$\Pr(X > (1+\delta)\mu) \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu},$$

where $\mu = \sqrt{n}$ and $\delta > 0$ is the deviation factor.
Here, we are interested in $\Pr(X > 3\mu)$.
This corresponds to $1 + \delta = 3$, so $\delta = 2$.
Substituting these values into the Chernoff bound:

$$\Pr(X > 3\mu) \leq \left(\frac{e^2}{3^3}\right)^{\mu}.$$

Simplify further:

$$\Pr(X > 3\mu) \leq \left(\frac{e^2}{27}\right)^{\sqrt{n}}.$$

This bound demonstrates that the probability decreases exponentially with $\sqrt{n}$.

Exercise 1.6.36

What is the total running time of counting from 1 to n in binary if the time needed to add 1 to the current number i is proportional to the number of bits in the binary expansion of i that must change in going from i to i+1?

To calculate the total running time, we analyze how many bits flip during the counting process.
Each bit in the binary representation flips at a predictable rate:

- The rightmost bit flips every increment (e.g., $0 \to 1$, $1 \to 0$), so it flips n times.
- The second bit flips every 2 increments, so it flips $n/2$ times.
- The third bit flips every 4 increments, so it flips $n/4$ times.
- In general, the k-th bit flips $n/2^{k+1}$ times.

The total number of bit flips across all positions is:

$$\text{Total flips} = \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{k+1}}.$$

This sum is a geometric series:

$$\sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{1}{2^{k+1}} = 1 - \frac{1}{2^{\lfloor \log_2 n \rfloor + 1}} < 1.$$

Therefore:

$$\text{Total flips} \approx n.$$

Since the time required is proportional to the number of flips, the total running time is:

$$O(n).$$

Each bit flips less frequently as the bit position increases.
The cumulative effect is that the total number of flips is proportional to n.
Thus, the total running time of counting from 1 to n in binary is $O(n)$.

Exercise 1.6.39

Question: Consider the following recurrence equation, defining a function $T(n)$:

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 2 \cdot T(n-1) & \text{otherwise.} \end{cases}$$

Show, by induction, that $T(n) = 2^n$.

Answer:

We have to prove by induction that $T(n) = 2^n$ for all $n \geq 0$.
Base Case: For $n = 0$:

$$T(0) = 1.$$

The formula $T(n) = 2^n$ gives:

$$T(0) = 2^0 = 1.$$

Thus, the base case holds.

Inductive Step: Assume that $T(k) = 2^k$ holds for some $k \geq 0$.
This is the **inductive hypothesis**.
We need to show that $T(k+1) = 2^{k+1}$.
From the recurrence relation:

$$T(k+1) = 2 \cdot T(k).$$

Using the inductive hypothesis ($T(k) = 2^k$):

$$T(k+1) = 2 \cdot 2^k = 2^{k+1}.$$

Conclusion: By induction, $T(n) = 2^n$ holds for all $n \geq 0$.

Exercise 1.6.52

Question: Show that the summation $\sum_{i=1}^{n} \lceil \log_2 i \rceil$ is $O(n \log n)$.

Answer:

We analyze the summation:

$$\sum_{i=1}^{n} \lceil \log_2 i \rceil.$$

To approximate it, we first note that:

$$\lceil \log_2 i \rceil \leq \log_2 i + 1.$$

Thus:

$$\sum_{i=1}^{n} \lceil \log_2 i \rceil \leq \sum_{i=1}^{n} \log_2 i + \sum_{i=1}^{n} 1.$$

The term $\sum_{i=1}^{n} 1 = n$.
The more significant term is $\sum_{i=1}^{n} \log_2 i$.
To approximate this summation, we use the integral:

$$\sum_{i=1}^{n} \log_2 i \approx \int_{1}^{n} \log_2 x \, dx.$$

We evaluate the integral:

$$\int \log_2 x \, dx = \frac{1}{\ln 2} \int \ln x \, dx = \frac{1}{\ln 2} \cdot x(\ln x - 1).$$

Now evaluate from 1 to n:

$$\int_{1}^{n} \log_2 x \, dx = \frac{1}{\ln 2} \left[n(\ln n - 1) - 1(\ln 1 - 1) \right].$$

Simplifying:

$$\int_{1}^{n} \log_2 x \, dx = \frac{1}{\ln 2} \left(n \ln n - n + 1 \right).$$

Thus, the summation $\sum_{i=1}^{n} \log_2 i$ is tightly approximated as:

$$\sum_{i=1}^{n} \log_2 i \approx \frac{1}{\ln 2} (n \ln n - n + 1).$$

For asymptotic analysis, the $n \log_2 n$ term dominates, so:

$$\sum_{i=1}^{n} \lceil \log_2 i \rceil = O(n \log n).$$

Exercise 1.6.62

Question: Consider an implementation of the extendable table, but instead of copying the elements of the table into an array of double the size (that is, from n to $2N$) when its capacity is reached, we copy the elements into an array with $\lceil\sqrt{N}\rceil$ additional cells, going from capacity N to $N+\lceil\sqrt{N}\rceil$. Show that performing a sequence of n add operations (that is, insertions at the end) runs in $\Theta(n^{3/2})$ time in this case.

Answer:
We analyze the total cost of n insertions using amortization with cyber dollars.

1. Cost of Insertions:
- Each insertion into the table costs 1 cyber dollar.
- Thus, the total cost for n insertions is n cyber dollars.

2. Cost of Resizing:
- When the table is resized from capacity N to $N+\lceil\sqrt{N}\rceil$, the cost of copying all N elements is N cyber dollars.
- The total resizing cost depends on the number of resizes, k, needed to perform n insertions.

3. Number of Resizes:
The table size after k-th resize is approximately:

$$\sum_{i=1}^{k}\sqrt{i}.$$

Using integration to approximate:

$$\sum_{i=1}^{k}\sqrt{i}\approx\int_{1}^{k}\sqrt{x}\,dx=\frac{2}{3}k^{3/2}.$$

Thus:

$$n\approx\frac{2}{3}k^{3/2},\quad\text{or equivalently, }k\approx\left(\frac{3n}{2}\right)^{2/3}.$$

4. Total Cost:
- The cost of resizing is:

$$\sum_{i=1}^{k}i\approx\int_{1}^{k}x\,dx=\frac{1}{2}k^{2}.$$

Substituting $k\approx n^{2/3}$:

$$\text{Resizing cost}\approx\frac{1}{2}n^{4/3}.$$

5. Final Time Complexity:
The total cost of n operations is:

$$\text{Total cost}=\text{Insertion cost}+\text{Resizing cost}=n+\Theta(n^{3/2}).$$

For large n, the $\Theta(n^{3/2})$ term dominates, so the total running time is:

$$\Theta(n^{3/2}).$$

Exercise 1.6.70

Given an array A, describe an efficient algorithm for reversing A. For example, if A = [3, 4, 1, 5], then its reversal is A = [5, 1, 4, 3]. You can only use O(1) memory in addition to that used by A itself. What is the running time of your algorithm?

1. Two-Pointer Method

The two-pointer method uses two indices, one starting at the beginning of the array and the other at the end. The elements at these indices are swapped, and the pointers are moved toward the center.

Algorithm:

1. Initialize two pointers: $i = 0$ (start of the array) and $j = n - 1$ (end of the array).
2. While $i < j$:
   - Swap $A[i]$ and $A[j]$.
   - Increment i by 1 and decrement j by 1.
3. Stop when $i \geq j$.

Pseudocode:

$\texttt{ReverseArray}(A)$:

Input: An array A of size n.

Output: The reversed array A.

1: $i \leftarrow 0$, $j \leftarrow n - 1$
2: **while** $i < j$ **do**
3: Swap $A[i]$ and $A[j]$
4: $i \leftarrow i + 1$
5: $j \leftarrow j - 1$
6: **end while**
7: **return** A

Time Complexity: $O(n)$ (each swap takes constant time, and there are $n/2$ swaps).
Space Complexity: $O(1)$.

2. XOR Swapping Method

This method eliminates the need for a temporary variable during swaps by using XOR bitwise operations. Two elements can be swapped in place with three XOR operations.

Algorithm:

1. Initialize two pointers: $i = 0$ (start of the array) and $j = n - 1$ (end of the array).
2. While $i < j$:
   - Perform XOR swapping:

$$A[i] = A[i] \oplus A[j], \quad A[j] = A[i] \oplus A[j], \quad A[i] = A[i] \oplus A[j].$$

- Increment i by 1 and decrement j by 1.

3. Stop when $i \geq j$.

Pseudocode:

$\texttt{ReverseArrayXOR}(A)$:

Input: An array A of size n.

Output: The reversed array A.

1: $i \leftarrow 0$, $j \leftarrow n-1$
2: **while** $i < j$ **do**
3: $\quad A[i] \leftarrow A[i] \oplus A[j]$
4: $\quad A[j] \leftarrow A[i] \oplus A[j]$
5: $\quad A[i] \leftarrow A[i] \oplus A[j]$
6: $\quad i \leftarrow i+1$
7: $\quad j \leftarrow j-1$
8: **end while**
9: **return** A

Time Complexity: $O(n)$ (each XOR operation is constant time, and there are $n/2$ swaps).
Space Complexity: $O(1)$.

Exercise 1.6.77

Given an integer k > 0 and an array A of n bits, describe an efficient algorithm for finding the shortest subarray of A that contains k ones. What is the running time of your method?

Algorithm Description:

1. **Initialize Variables**:
   - Let start $= 0$, count $= 0$, minLength $= n+1$, and result $= (-1, -1)$.
   - start and end represent the current subarray boundaries.
   - count tracks the number of ones in the current subarray.

2. **Expand the Window**:
   - Iterate through the array with end from 0 to $n-1$:
   - Add A[end] to count.
   - If count $= k$, shrink the window.

3. **Shrink the Window**:
   - While count $= k$:
   - Check if the current subarray length (end $-$ start $+ 1$) is smaller than minLength.
   If yes, update minLength and result.
   - Subtract A[start] from count and increment start.

4. **Return the Result**:
   - If minLength remains $n+1$, no subarray with k ones exists. Otherwise, return result and minLength.

Pseudocode:

```
1: function FindShortestSubarrayWithKOnes(A, n, k)
2:     start ← 0, count ← 0, minLength ← n + 1
3:     result ← (−1, −1)
4:     for end ← 0 to n − 1 do
5:         count ← count + A[end]
6:         while count = k do
7:             if (end − start + 1) < minLength then
8:                 minLength ← end − start + 1
9:                 result ← (start, end)
10:            end if
11:            count ← count − A[start]
12:            start ← start + 1
13:        end while
14:    end for
15:    if minLength = n + 1 then return "No subarray found"
16:    elsereturn result, minLength
17:    end if
18: end function
```

Example:

Input:

A = [1,0,1,1,0,1,0,1],

k = 3

Steps:

- end $= 0$, count $= 1$
- end $= 2$, count $= 2$
- end $= 3$, count $= 3$: Shrink window to find $(2, 5)$.

Output:

Result: $(2, 5)$, Length: 4

Complexity Analysis:

Time Complexity: $O(n)$

- Each element is processed at most twice: once when expanding and once when shrinking.

Space Complexity: $O(1)$

- Only pointers and counters are used.