COMP 3711 Course Notes

Design and Analysis of Algorithms

LIN, Xuanyu

A L G O R I T H M S

COMP 3711 Design and Analysis of Algorithms

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

September 20, 2023

Contents

1 Asymptotic Notation **2**

2 Introduction - The Sorting Problem **2**
- 2.1 Selection Sort . . . 2
- 2.2 Insertion Sort . . . 2
- 2.3 Wild-Guess Sort . . . 2
- 2.4 Worst-Case Analysis . . . 2

3 Divide & Conquer **4**
- 3.1 Binary Search . . . 4
- 3.2 Merge Sort . . . 5
- 3.3 Inversion Counting . . . 6
  - 3.3.1 Counting Inversions: Divide-and-Conquer . . . 6
  - 3.3.2 Implementation of the Algorithm . . . 6
- 3.4 Basic Summary of D&C: Problem Size & Number of Problems . . . 7
- 3.5 Maximum Contiguous Subarray . . . 7
  - 3.5.1 A D&C Algorithm . . . 8
  - 3.5.2 Kadane's Algorithm . . . 8
- 3.6 Integer and Matrix Multiplication . . . 9
  - 3.6.1 A Simple D&C Algorithm for Integer Multiplication . . . 9
  - 3.6.2 Karatsuba Multiplication . . . 10
- 3.7 Matrix Multiplication . . . 10
  - 3.7.1 A D&C Solution to Matrix Multiplication . . . 10
  - 3.7.2 Strassen's Matrix Multiplication Algorithm . . . 11
- 3.8 Master Theorem . . . 11

1 Asymptotic Notation

Upper Bounds $T(n) = O(f(n))$
if exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \leq c \cdot f(n)$.
Lower Bounds $T(n) = \Omega(f(n))$
if exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \geq c \cdot f(n)$.
Tight Bounds $T(n) = \Theta(f(n))$
if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.
Note: Here ”=” means ”is”, not equal.

2 Introduction - The Sorting Problem

2.1 Selection Sort

Algorithm 1: Selection Sort

```
Input: An array A[1...n] of elements
Output: Array A[1...n] of elements in sorted order (asending)
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        if A[i] > A[j] then
            swap A[i] and A[j]
        end
    end
end
```

Running Time: $\frac{n(n-1)}{2}$
Best-Case = Worst-Case: $T(n) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$

2.2 Insertion Sort

Algorithm 2: Insertion Sort

```
Input: An array A[1...n] of elements
Output: Array A[1...n] of elements in sorted order (asending)
for i ← 2 to n do
    j ← i − 1 while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
    end
    j ← j − 1
end
```

Running Time: Depends on the input array, ranges between $(n-1)$ and $\frac{n(n-1)}{2}$
Best-Case: $T(n) = n - 1 = \Theta(n)$ (Useless)
Worst-Case: $T(n) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$ (Commonly-Used)
Average-Case: $T(n) = \Theta(\sum_{i=2}^{n} \frac{i-1}{2}) = \Theta(\frac{n(n-1)}{4}) = \Theta(n^2)$ (Sometimes Used)

2.3 Wild-Guess Sort

Running Time: Depends on the random generation, could be faster than the insertion sort.

2.4 Worst-Case Analysis

The algorithm’ s worst case running time is $O(f(n)) \implies$ On all inputs of (large) size n, the running time of the algorithm is $\leq c \cdot f(n)$.

Algorithm 3: Wild-Guess Sort

Input: An array $A[1...n]$ of elements
Output: Array $A[1...n]$ of elements in sorted order (asending)
$\pi \leftarrow [4, 7, 1, 3, 8, 11, 5, ...]$ Create random permutation Check if $A[\pi[i]] \leq A[\pi[i+1]]$ for all $i = 1, 2, ..., n-1$ If yes, output A according to π and terminate else $Insertion - Sort(A)$

The algorithm' s worst case running time is $\Omega(f(n)) \implies$ There exists at least one input of (large) size n for which the running time of the algorithm is $\geq c \cdot f(n)$.

Thus, Insertion sort runs in $\Theta(n^2)$ time.

Notice

Selection sort, insertion sort, and wild-guess sort all have worst-case running time $\Theta(n^2)$. How to distinguish between them?

- Closer examination of hidden constants
- Careful analysis of typical expected inputs
- Other factors such as cache efficiency, parallelization are important
- Empirical comparison

Stirling's Formula

Prove that $\log(n!) = \Theta(n \log n)$

First $\log(n!) = O(n \log n)$ since:

$$\log(n!) = \sum_{i=1}^{n} \log i \leq n \times \log n = O(n \log n)$$

Second $\log(n!) = \Omega(n \log n)$ since:

$$\log(n!) = \sum_{i=1}^{n} \log i \geq \sum_{i=n/2}^{n} \log i \geq n/2 \times \log n/2 = n/2(\log n - \log 2) = \Omega(n \log n)$$

Thus, $\log(n!) = \Theta(n \log n)$

3 Divide & Conquer

Main idea of D & C: Solve a problem of size n by breaking it into one or more smaller problems of size less than n. Solve the smaller problems recursively and combine their solutions, to solve the large problem.

3.1 Binary Search

Example: Binary Search

Input: A sorted array $A[1, ..., n]$, and an element x

Output: Return the position of x, if it is in A; otherwise output nil

Idea of the binary search: Set $q \leftarrow$ middle of the array. If $x = A[q]$, return q. If $x < A[q]$, search $A[1, ..., q-1]$, else search $A[q+1, ..., n]$.

Algorithm 4: Binary Search

```
Input: Array A[1...n] of elements in sorted order
BinarySearch(A[], p, r, x)(p, r being the left & right iteration, x being the element being searched)
if p > r then
    return nil
end
q ← [(p + r)/2]
if x = A[q] then
    return q
end
if x < A[q] then
    BinarySearch(A[], p, q − 1, x)
end
else
    BinarySearch(A[], q + 1, r, x)
end
```

Recurrence of the algorithm, supposing $T(n)$ being the number of the comparisons needed for n elements: $T(n) = T(\frac{n}{2}) + 2$ if $n > 1$, with $T(1) = 2$.

$\implies T(n) = 2\log_2 n + 2 \implies O(\log n)$ algorithm

Example: Binary Search in Rotated Array

Suppose you are given a sorted array A of n distinct numbers that has been rotated k steps, for some unknown integer k between 1 and $n-1$. That is, $A[1...k]$ is sorted in increasing order, and $A[k+1...n]$ is also sorted in increasing order, and $A[n] < A[1]$.

Design an $O(\log n)$-time algorithm that for any given x, finds x in the rotated sorted array, or reports that it does not exist.

Algorithm:

First conduct a $O(\log n)$ algorithm to find the value of k, then search for the target value in either the first part or the second part.

$Find - x(A, x)$

$k \leftarrow Find - k(A, 1, n)$ (First find k)

$if\ x \geq A[1]\ then\ return\ BinarySearch(A, 1, k, x)$

$Else\ return\ BinarySearch(A, k + 1, n, x)$

Example: Finding the last 0

You are given an array $A[1...n]$ that contains a sequence of 0 followed by a sequence of 1 (e.g., 0001111111). A contains k 0(s) ($k > 0$ and $k << n$) and at least one 1.

Design an $O(\log k)$-time algorithm that finds the position k of the last 0.

Algorithm:

$i \leftarrow 1$
$while\ A[i] = 0$
  $i \leftarrow 2i$
$find - k(A[i/2...i])$

3.2 Merge Sort

Principle of the Merge Sort:

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

Algorithm 5: Merge Sort

```
MergeSort(A, p, r)(p, r being the left & right side of the array to be sorted)
if p = r then
    return
end
q ← [(p + r)/2]
MergeSort(A, p, q)
MergeSort(A, q + 1, r)
Merge(A, p, q, r)
First Call: MergeSort(A, 1, n)
```

Algorithm 6: Merge

```
Input: Two Arrays L ← A[p...q] and R ← A[q + 1...r] of elements in sorted order
Merge(A, p, q, r)
Append ∞ at the end of L and R
i ← 1, j ← 1
for k ← p to r do
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i + 1
    end
    else
        A[k] ← R[j]
        j ← j + 1
    end
end
```

Let $T(n)$ be the running time of the algorithm on an array of size n.

Merge Sort Recurrence:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n), \quad n > 1, \quad T(1) = O(1)$$

Simplification:

$$\implies T(n) = 2T(n/2) + n, \quad n > 1, \quad T(1) = 1$$

Result:

$$T(n) = n \log_2 n + n = O(n \log n)$$

3.3 Inversion Counting

Definition of the Inversion Numbers: Given array $A[1...n]$, two elements $A[i]$ and $A[j]$ are inverted if $i < j$ but $A[i] > A[j]$. The inversion number of A is the number of inverted pairs.

Theorem:
The number of swaps used by Insertion Sort = Inversion Number (Proved by induction on the size of the array)
Algorithm to Compute Inversion Number:
Algorithm 1: Check all $\Theta(n^2)$ pairs.
Algorithm 2: Run Insertion Sort and count the number of swaps -Also $\Theta(n^2)$ time.
Algorithm 3: Divide and Conquer

3.3.1 Counting Inversions: Divide-and-Conquer

Principle of the Algorithm:
- Divide: divide array into two halves
- Conquer: recursively count inversions in each half
- Conbine: count inversions where a_i and a_j are in different halves, and return sum of three quantities

Inversion counting during the combine step is very similar to the Merge Algorithm (Algorithm 6), by counting the sum of each inversion number of the right array (indicated by $I[j]$) comparing to the left array.

Algorithm 7: Inversion Count during Combination

Input: Two Arrays $L \leftarrow A[p...q]$ and $R \leftarrow A[q+1...r]$ of elements in sorted order
`Count(`A, p, q, r`)`
$i \leftarrow 1,\ j \leftarrow 1,\ c \leftarrow 0$
while $(i \leq q - p + 1) \&\& (j \leq r - q)$ **do**
  if $L[i] \leq R[j]$ **then**
    $i \leftarrow i + 1$
  end
  else
    $I[j] = q - p - i + 2$
    $c \leftarrow c + I[j]$
    $j \leftarrow j + 1$
  end
end

The time-complexity of the algorithm is $\Theta(n \log n)$, same as the Merge Sort.

3.3.2 Implementation of the Algorithm

Algorithm 8: Main Algorithm

`Sort-and-Count(`A, p, r`)`
if $p = r$ **then**
  return *0*
end
$q \leftarrow \lfloor (p + r)/2 \rfloor$
$c_1 \leftarrow$ `Sort-and-Count(`A, p, q`)`
$c_2 \leftarrow$ `Sort-and-Count(`$A, q + 1, r$`)`
$c_3 \leftarrow$ `Merge-and-Count(`A, p, q, r`)`
return $c_1 + c_2 + c_3$
First Call: `Sort-and-Count(`$A, 1, n$`)`

Algorithm 9: Merge-and-Count

Input: Two Arrays $L \leftarrow A[p...q]$ and $R \leftarrow A[q+1...r]$ of elements in sorted order
```
Merge-and-Count(A, p, q, r)
Append ∞ at the end of L and R
i ← 1, j ← 1, c ← 0
for k ← p to r do
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i + 1
    end
    else
        A[k] ← R[j]
        j ← j + 1
        c ← c + q − p − i + 2
    end
end
return c
```

3.4 Basic Summary of D&C: Problem Size & Number of Problems

Observations of D&C in Logarithmic Patterns:
- Break up problem of size n into p parts of size n/q.
- Solve parts recursively and combine solutions into overall solution.
- At level i, we break i times and we have p^i problems of size n/q^i.
- When we cannot break up any more, usually when the problem size becomes 1. Usually $i \approx \log_q n$.

The number of problems at (bottom) level $\log_q n$ is $p^i = p^{\log_q n} = n^{\log_q p}$.

Observations of D&C in Non-Logarithmic Patterns:
- Break up problem of size n into $p(\leq 2)$ parts of size $n - q$. (e.g. $q = 1$ for Hanoi Problem)
- Assume that $q = 1$
- At level i, we break i times and we have p^i problems of size $n - i$.
- If we stop when the problem size becomes 1, then $n - i = 1 \implies i = n - 1$.

The number of problems at (bottom) level $n - 1$ is: $p^i = p^{n-1}$.

3.5 Maximum Contiguous Subarray

Example: The Maximum Subarray Problem
Input: An array of numbers $A[1, ..., n]$, both positive and negative
Output: Find the maximum $V(i, j)$, where $V(i, j) = \sum_{k=i}^{j} A[k]$

Brute-Force Algorithm

Idea: Calculate the value of $V(i, j)$ for each pair $i \leq j$ and return the maximum value.
Requires three nested for-loop, time complexity: $\Theta(n^3)$.

A Data-Reuse Algorithm

Idea: $V(i, j) = V(i, j - 1) + A[j]$
Requires two nested for-loop, time complexity: $\Theta(n^2)$.

3.5.1 A D&C Algorithm

Idea: Cut the array into two halves, all subarrays can be classified into three cases: entirely in the first/second half, or crosses the cut.

Compare with the merge sort: Whole algorithm will run in $\Theta(n \log n)$ time if the cross-cut can be solved in $O(n)$ time.

Algorithm 10: Maximum Subarray

```
MaxSubArray(A, p, r)
if p = r then
 |  return A[p]
end
q ← ⌊(p + r)/2⌋
M_1 ← MaxSubArray(A, p, q)
M_2 ← MaxSubArray(A, q + 1, r)
L_m, R_m ← −∞
V ← 0
for i ← q to p do
 |  V ← V + A[i]
 |  if V > L_m then
 |   |  L_m ← V
 |  end
end
V ← 0
for i ← q + 1 to r do
 |  V ← V + A[i]
 |  if V > R_m then
 |   |  R_m ← V
 |  end
end
return max(M_1, M_2, L_m + R_m)
First Call: MaxSubArray(A, 1, n)
```

Recurrence: $T(n) = 2T(n/2) + n \implies T(n) = \Theta(n \log n)$

3.5.2 Kadane's Algorithm

Idea: Based on the principles of **Dynamic Programming**. Let $V[i]$ be the (local) maximum sub-array that ends at $A[i]$, then we let:

- $V[1] = A[1]$
- $V[i] = \max(A[i], A[i] + V[i-1])$

The maximum of $V[i]$, namely V_{max} is the maximum continuous subarray found so far.

Algorithm 11: Kadane's Algorithm

```
V_max ← −∞; V ← 0; start ← 1; end ← 1; temp ← 1 (Note: start & end specify the maximum sub-array)
for i ← 1 to n do
 |  V ← V + A[i]
 |  if V < A[i] then // Implies V[i − 1] is negative, restart from the current position
 |   |  V ← A[i]; temp ← i
 |  end
 |  if V > V + max then // Found a max sum, update start and end
 |   |  V_max ← V; start ← temp; end ← i
 |  end
end
```

Example: Maximizing Stock Profits

You are presented with an array $p[1 \dots n]$ where $p[i]$ is the price of the stock on day i.

Design an divide-and-conquer algorithm that finds a strategy to make as much money as possible, i.e., it finds a pair i, j with $1 \leq i \leq j \leq n$ such that $p[j] - p[i]$ is maximized over all possible such pairs. Note that you are only allowed to buy the stock once and then sell it later.

Idea 1: Divide and Conquer

- Cut the array into two halves.
- All i, j solutions can be classified into three cases: both i, j are entirely in the first(second) half, or i is in the left half while j is in the right half.
- Maximizing a Case 3 result $p[j] - p[i]$ means finding the smallest value in the first half and the largest in the second half.

Time Complexity: $T(n) = 2T(n/2) + n \implies T(n) = \Theta(n \log n)$

Idea 2: Kadane's Algorithm

- Create a **Profit** array with $Profit[i] = Price[i+1] - Price[i]$.
- Perform the Kadane's Algorithm.

Time Complexity: $O(n)$

3.6 Integer and Matrix Multiplication

3.6.1 A Simple D&C Algorithm for Integer Multiplication

Goal: Given two n-bit binary integers a and b, compute: $a \cdot b$.

Idea: Multiplication by 2^k can be done in one time unit by a left shift of k bits.

- Rewrite the two numbers as $a = 2^{n/2}a_1 + a_0$, $b = 2^{n/2}b_1 + b_0$.
- The product becomes: $a \cdot b = (2^{n/2}a_1 + a_0)(2^{n/2}b_1 + b_0) = 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + a_0 b_1) + a_0 b_0$
- The new computation requires 4 products of integers, each with $n/2$ bits.
- Apply D&C by splitting a problem of size n, to 4 problems of size $n/2$.

Algorithm 12: Binary Multiplication

```
Multiply(A, B)
n ← size of A
if n = 1 then
 | return A[1] · B[1]
end
mid ← ⌊n/2⌋
U ← Multiply (A[mid + 1..n], B[mid + 1..n]) // a1b1
V ← Multiply (A[mid + 1..n], B[1..mid]) // a1b0
W ← Multiply (A[1..mid], B[mid + 1..n]) // a0b1
Z ← Multiply (A[1..mid], B[1..mid]) // a0b0
M[1..2n] ← 0
M[1..n] ← Z // a0b0
M[mid + 1..] ← M[mid + 1..] ⊕ V ⊕ W // +[(a1b0 + a0b1) ≪ (left shift) n/2]
M[2mid + 1..] ← M[2mid + 1..] ⊕ U // +[a1b1 ≪ n]
return M
```

Time Complexity: $T(n) = 4T(n/2) + n \implies T(n) = \Theta(n^2)$

3.6.2 Karatsuba Multiplication

Goal: Given two n-bit binary integers a and b, compute: $a \cdot b$.
Idea:
- We've seen that $ab = a_1b_12^n + (a_1b_0 + a_0b_1)2^{n/2} + a_0b_0$, so we only need the result of $a_1b_0 + a_0b_1$.
- Note that $a_1b_0 + a_0b_1 = (a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0$, thus only requires performing 3 multiplications of size $n/2$.

Algorithm 13: Binary Multiplication (Karatsuba's Multiplication Algorithm)

```
Multiply(A, B)
n ← size of A
if n = 1 then
  | return A[1] · B[1]
end
mid ← ⌊n/2⌋
U ← Multiply (A[mid + 1..n], B[mid + 1..n]) // a_1 b_1
Z ← Multiply (A[1..mid], B[1..mid]) // a_0 b_0
A' ← A[mid + 1..n] ⊕ A[1..mid] // a_1 + a_0
B' ← B[mid + 1..n] ⊕ B[1..mid] // b_1 + b_0
Y ← Multiply (A', B') // (a_1 + a_0)(b_! + b_0)
M[1..2n] ← 0
M[1..n] ← Z // a_0 b_0
M[mid + 1..] ← M[mid + 1..] ⊕ Y ⊖ U ⊖ Z // +[(a_1 b_0 + a_0 b_1) ≪ (left shift) n/2]
M[2mid + 1..] ← M[2mid + 1..] ⊕ U // +[a_1 b_1 ≪ n]
return M
```

Time Complexity: $T(n) = 3T(n/2) + n \implies T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585\cdots})$
For recent research, see: Integer Multiplication in $O(n \log n)$ Time (David Harvey & Joris van der Hoeven, 2021)

3.7 Matrix Multiplication

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} \qquad c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Brute Force Method: $\Theta(n^3)$ time.

3.7.1 A D&C Solution to Matrix Multiplication

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad \begin{cases} C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{cases}$$

Recursion: $T(n) = 8T(n/2) + O(n^2) \implies T(n) = O(n^3)$

3.7.2 Strassen's Matrix Multiplication Algorithm

Idea: Muliply 2-by-2 block matrices with only 7 multiplications

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$\begin{cases} P_1 = A_{11} \times (B_{12} - B_{22}) \\ P_2 = (A_{11} + A_{12}) \times B_{22} \\ P_3 = (A_{21} + A_{22}) \times B_{11} \\ P_4 = A_{22} \times (B_{21} - B_{11}) \\ P_5 = (A_{11} + A_{12}) \times (B_{11} + B_{22}) \\ P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{cases} \qquad \begin{cases} C_{11} = P_5 + P_4 - P_2 + P_6 \\ C_{12} = P_1 + P_2 \\ C_{21} = P_3 + P_4 \\ C_{22} = P_5 + P_1 - P_3 - P_7 \end{cases}$$

Recursion: $T(n) = 7T(n/2) + n^2 \implies T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807\cdots})$
For recent research, see: Powers of Tensors and Fast Matrix Multiplication (Le Gall, 2014)

3.8 Master Theorem

Homework 1

Name: LIN, Xuanyu, SID: 20838295, Email Address: xlinbf@connect.ust.hk

Problem 1

For each pair of expressions (A, B) below, indicate whether A is $O, \Omega,$ or Θ of B. List all applicable relations. No explanation is needed.
(a) $A = n^3 - 100n$, $B = n^2 + 50n$
(b) $A = \log_2(n^2)$, $B = \log_{2.7}(n^4)$
(c) $A = 10^{10000}$, $B = \frac{n}{10^{10000}}$
(d) $A = 2^{n \log n}$, $B = n^{10} + 8n^2$
(e) $A = 2^n$, $B = 2^{n+\log n}$
(f) $A = 3^{3n}$, $B = 3^{2n}$
(g) $A = (\sqrt{2})^{\log n}$, $B = \sqrt{\log n}$

Solution.
(a) $A = \Omega(B)$
(b) $A = O(B), A = \Omega(B), A = \Theta(B)$
(c) $A = O(B)$
(d) $A = \Omega(B)$
(e) $A = O(B)$
(f) $A = \Omega(B)$
(g) $A = \Omega(B)$

Problem 2

Derive asymptotic upper bounds for T(n) in the following recurrences. Make your bounds as tight as possible. You may assume that n is a power of 2 for (a), n is a power of 4 for (b), and $\sqrt{n}$ is always an integer for (c).
(a) $T(1) = 1$; $T(n) = 4T(n/2) + n^2$ for $n > 1$.
(b) $T(1) = 1$; $T(n) = 16T(n/4) + n$ for $n > 1$.
(c) $T(2) = 1$; $T(n) = T(\sqrt{n}) + 1$ for $n > 1$.

Solution.
(a)

$$\begin{aligned} T(n) &= 4T(n/2) + n^2 = 4[4T(n/4) + (n/2)^2] + n^2 = 4\{4[4T(n/8) + (n/4)^2] + (n/2)^2\} + n^2 \\ &= 4\{4\{4\{\cdots[4T(1) + 2^2] + 4^2\} + \cdots + (n/4)^2\} + (n/2)^2\} + n^2 \\ &= 4^{\log_2 n} + 4^{\log_2 n-1} \times 2^2 + 4^{\log_2 n-2} \times 4^2 + \cdots + 4^1 \times (n/2)^2 + n^2 \\ &= n^2 + \frac{n^2}{4} \times 2^2 + \frac{n^2}{4^2} \times 4^2 + \cdots + 4 \times (n/2)^2 + n^2 \\ &= n^2(\log_2 n + 1) = O(n^2 \log n) \end{aligned}$$

(b)

$$\begin{aligned} T(n) &= 16T(n/4) + n = 16[16T(n/4^2) + n/4] + n = 16\{16[16T(n/4^3) + n/4^2] + n/4\} + n \\ &= 16\{16\{16\{\cdots[16T(1) + 4] + 4^2\} + \cdots + n/4^2\} + n/4\} + n \\ &= 16^{\log_4 n} + 16^{\log_4 n-1} \times 4 + 16^{\log_4 n-2} \times 4^2 + \cdots + 16^1 \times n/4 + n \\ &= n^2 + \frac{n^2}{4} + \frac{n^2}{4^2} + \cdots + 4n + n \\ &= \frac{n - 4n^2}{1-4} = \frac{4}{3}n^2 - \frac{1}{3}n = O(n^2) \end{aligned}$$

(c)

$$T(n) = T(\sqrt{n}) + 1 = T(n^{1/2^2}) + 2 = T(n^{1/2^3}) + 3 = \cdots = T(2) + \log_2 n - 1 = O(\log_2 n)$$

Problem 3

(a) Describe a recursive algorithm that returns a list of all possible $n \times n$ binary arrays where n is a positive input integer. An array is binary if each of its entry is either 0 or 1. You can either describe your algorithm in text or in a documented pseudocode. Make sure that your algorithm is recursive. Make sure that your description is understandable.
(b) Write down the recurrence for the running time of your recursive algorithm in (a) with the boundary condition(s). Explain your notations. Solve your recurrence from scratch to obtain the the running time of your algorithm.

Solution.
(a)

Algorithm 14: All $n \times n$ Binary Arrays

`AllBinaryArrays(`n`)`

First Call: `AllBinaryArrays(`n`)`

(b) **Recurrence:** $T(n) =$

Problem 4

Let $A[1..n]$ be an array of n elements. One can compare in $O(1)$ time two elements of A to see if they are equal or not; however, the order relations $<$ and $>$ do not make sense. That is, one can check whether $A[i] = A[j]$ in $O(1)$ time, but the relations $A[i] < A[j]$ and $A[i] > A[j]$ are undefined and cannot be determined.
In the tutorial you developed an $O(n \log n)$-time divide-and-conquer algorithm for finding a majority element of A if one exists. In this assignment you need to generalize this problem.
Let $k \in [1..n]$ be a fixed integer. An element of $A[1..n]$ is a k-major element if its number of occurrences in A is greater than n/k. For example, if $n = 30$, then a 10-major element should occur greater than 3 times (i.e., at least 4 times). Note that it is possible that no k-major exists for a particular k; it is also possible that there are multiple k-major elements for a particular k.
This problem concerns with designing a divide-and-conquer algorithm for finding **all** 10-major elements in $A[1..n]$ in $O(n \log n)$ time; if there is no 10-major element, report so. Answer the following questions.

(a) What is the maximum number of 10-major elements in $A[1..n]$? Explain.

(b) Design a divide-and-conquer algorithm that finds all 10-major elements in $A[1..n]$ in $O(n \log n)$ time; if there is no 10-major element, your algorithm should report so. Recall that one can check whether $A[i] = A[j]$ in $O(1)$ time, but the relations $A[i] < A[j]$ and $A[i] > A[j]$ are undefined and cannot be computed.
Write your algorithm in documented pseudocode. Also, explain in text what your pseudocode does. Explain the correctness of your algorithm.
Since your algorithm uses the divide-and-conquer principle, it should be recursive in nature. That is, it should work on $A[1..n]$ in the first call to return all 10-major elements of $A[1..n]$, and in subsequent recursive calls, it may recurse on many subarrays $A[p..q]$ for some $p, q \in [1..n]$ to return all 10-major elements of $A[p..q]$
Given a particular subarray $A[p..q]$, a 10-major element of $A[p..q]$ is not necessarily a 10-major element of $A[1..n]$. Conversely, a 10-major element of $A[1..n]$ is not necessarily a 10-major element of $A[p..q]$.

(c) Derive a recurrence relation that describes the running time $T(n)$ of your algorithm. Explain your reasoning. State the boundary condition(s).

(d) Solve your recurrence **from scratch** to show that $T(n) = O(n \log n)$.

Solution.

Homework 2

Name: LIN, Xuanyu, SID: 20838295, Email Address: xlinbf@connect.ust.hk

Problem 5

Solution.

Problem 6

Solution.

Problem 7

Solution.