# COMP 3711 Course Notes

# Design and Analysis of Algorithms

**LIN, Xuanyu**

*ALGORITHMS*

COMP 3711 Design and Analysis of Algorithms

THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

November 28, 2023

# Contents

# 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** $i \leftarrow 1$ *to* $n-1$ **do**
  **for** $j \leftarrow i+1$ *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** $i \leftarrow 2$ *to* $n$ **do**
  $j \leftarrow i-1$ **while** $j \geq 1$ *and* $A[j] > A[j+1]$ **do**
    swap $A[j]$ and $A[j+1]$
  **end**
  $j \leftarrow j-1$
**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 $\pi$ 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**
$q \leftarrow [(p + r)/2]$
**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**
$q \leftarrow [(p + r)/2]$
`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 $L \leftarrow A[p...q]$ and $R \leftarrow A[q + 1...r]$ of elements in sorted order
`Merge`$(A, p, q, r)$
Append $\infty$ at the end of $L$ and $R$
$i \leftarrow 1,\ j \leftarrow 1$
**for** $k \leftarrow p$ *to* $r$ **do**
    **if** $L[i] \leq R[j]$ **then**
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    **end**
    **else**
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
    **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 $\infty$ at the end of $L$ and $R$
$i \leftarrow 1,\ j \leftarrow 1, c \leftarrow 0$
**for** $k \leftarrow p$ *to* $r$ **do**
  **if** $L[i] \leq R[j]$ **then**
    $A[k] \leftarrow L[i]$
    $i \leftarrow i + 1$
  **end**
  **else**
    $A[k] \leftarrow R[j]$
    $j \leftarrow j + 1$
    $c \leftarrow c + q - p - i + 2$
  **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
```

**Time Complexity:** $\Theta(n)$

**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 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]) // a_1 b_1
V ← Multiply (A[mid + 1..n], B[1..mid]) // a_1 b_0
W ← Multiply (A[1..mid], B[mid + 1..n]) // a_0 b_1
Z ← Multiply (A[1..mid], B[1..mid]) // a_0 b_0
M[1..2n] ← 0
M[1..n] ← Z // a_0 b_0
M[mid + 1..] ← M[mid + 1..] ⊕ V ⊕ W // +[(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:** $T(n) = 4T(n/2) + n \implies T(n) = \Theta(n^2)$

### 3.6.2 Karatsuba Multiplication

**Goal:** Given two $n$-bit binary integers $a$ and $b$, compute: $a \cdot b$.
**Idea:**
- We've seen that $ab = a_1b_1 2^n + (a_1b_0 + a_0b_1)2^{n/2} + a_0b_0$, 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`$(A, B)$
$n \leftarrow$ size of $A$
**if** $n = 1$ **then**
  **return** $A[1] \cdot B[1]$
**end**
$mid \leftarrow \lfloor n/2 \rfloor$
$U \leftarrow$ `Multiply` $(A[mid + 1..n], B[mid + 1..n])$ // $a_1b_1$
$Z \leftarrow$ `Multiply` $(A[1..mid], B[1..mid])$ // $a_0b_0$
$A' \leftarrow A[mid + 1..n] \oplus A[1..mid]$ // $a_1 + a_0$
$B' \leftarrow B[mid + 1..n] \oplus B[1..mid]$ // $b_1 + b_0$
$Y \leftarrow$ `Multiply` $(A', B')$ // $(a_1 + a_0)(b_! + b_0)$
$M[1..2n] \leftarrow 0$
$M[1..n] \leftarrow Z$ // $a_0b_0$
$M[mid + 1..] \leftarrow M[mid + 1..] \oplus Y \ominus U \ominus Z$ // $+[(a_1b_0 + a_0b_1) \ll$ (left shift) $n/2]$
$M[2mid + 1..] \leftarrow M[2mid + 1..] \oplus U$ // $+[a_1b_1 \ll n]$
**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

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

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

### 3.7.1 A D&C Solution to Matrix Multiplication

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

**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)
Conjecture: Close to $\Theta(n^2)$

## 3.8 Master Theorem

For recurrences of form

$$T(n) = aT(n/b) + f(n) \text{ or } T(n) \leq aT(n/b) + f(n), \text{ Let } c \equiv \log_b a$$

where
- $a \geq 1$ and $b > 1$ both being constants
- $f(n)$ is a (asymptotically) positive polynomial function
- $n/b$ could be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

### 3.8.1 Master Theorem for Equalities

(1) Work Increases: $f(n) = O(n^{c-\epsilon})$ for some $\epsilon \implies T(n) = \Theta(n^c)$
(2) Work Remains: $f(n) = \Theta(n^c \log^k n)$ for $k > -1 \implies T(n) = \Theta(n^c \log^{k+1} n)$
Note: For the case $k = -1$, $T(n) = \Theta(n^c \log\log n)$; For the case $k < -1$, $T(n) = \Theta(n^c)$
(3) Work Decreases: $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon \implies T(n) = \Theta(f(n))$
Note: Rigorously, the third case requires $af(n/b) \leq kf(n)$ for some $k < 1$ and sufficiently large $n$
(4) For a special case $T(n) = \sum_i T(\alpha_i n) + n$ where $\alpha_i > 0$ with $\sum_i \alpha_i < 1$, we have $T(n) = \Theta(n)$

### 3.8.2 Master Theorem for Inequalities

(1) Work Increases: $f(n) = O(n^{c-\epsilon})$ for some $\epsilon \implies T(n) = O(n^c)$
(2) Work Remains: $f(n) = O(n^c) \implies T(n) = O(n^c \log n)$
(3) Work Decreases: $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon \implies T(n) = O(f(n))$

# 4 Advanced Sorting Algorithms

## 4.1 Probability & Statistics, Random Permutation

$$E[X] = \sum i \cdot Pr[X = i]$$

$$E[X + Y] = E[X] + E[Y]$$

For independent random variables $X$ & $Y$,

$$E[XY] = E[X] \cdot E[Y]$$

**Algorithm 14:** Random Permutation

```
RandomPermute(A)
n ← A.length
for i ← 1 to n do
    swap A[i] with A[Random(1, i)]
end
```

## 4.2 Randomized Algorithm - Quicksort

**Idea:** Quicksort chooses item as pivot. It partitions array so that all items less than or equal to pivot are on the left and all items greater than pivot on the right. It then recursively Quicksorts left and right sides.

**Algorithm 15:** Quicksort

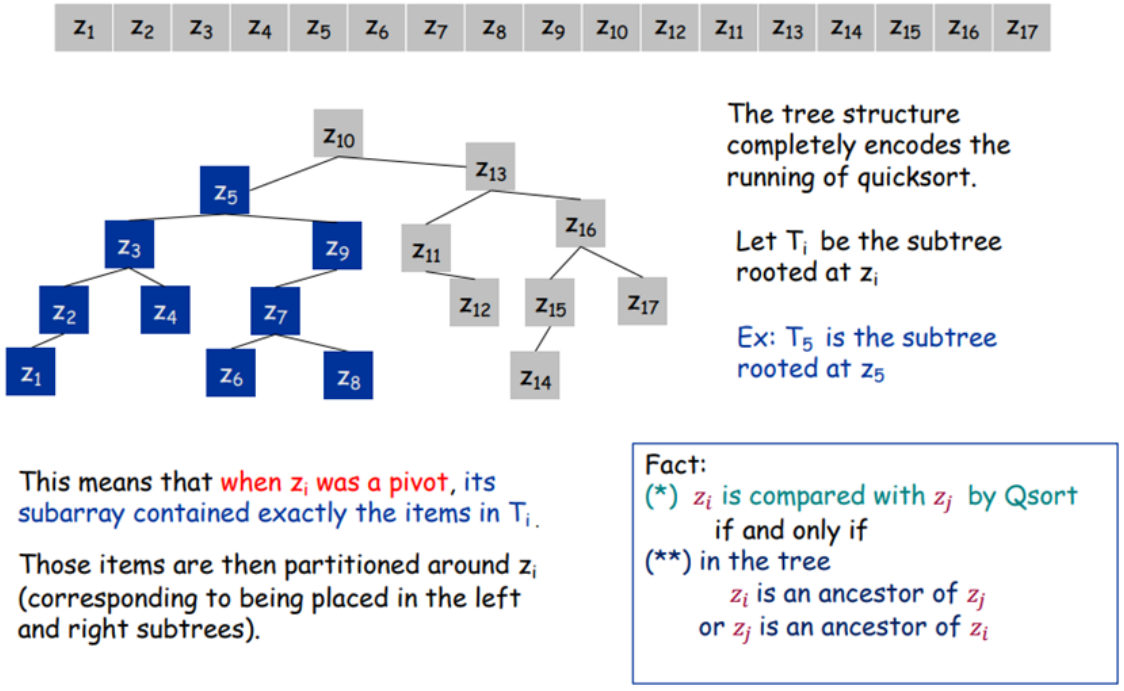
```
Quicksort(A, p, r) // Array from A[p] to A[r]
if p ≥ r then
    return
end
q =Partition (A, p, r) // Set a new pivot position
Quicksort (A, p, q − 1)
Quicksort (A, q + 1, r)
First Call: MaxSubArray(A, 1, n)

Partition(A, p, r)
x ← A[r] // Set the last item as pivot, or randomly swap away the last item before choosing the pivot
i ← p − 1
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1
        swap A[i] and A[j] // Put all items ≤ A[r] on the left
    end
end
swap A[i + 1] and A[r]
return i + 1
```

### 4.2.1 Running Time

**Best Case:** Always select the median element as the pivot - $\Theta(n \log n)$ time.
**Worst Case:** Always select the smallest (or the largest) element - $\Theta(n^2)$ time.
To make running time independent of input, we can randomly choose an element as the pivot by swapping it with last item in array before running the partition.

### 4.2.2 Binary Tree Representation




### 4.2.3 Expected Running Time for Random-Based Quicksort

- Two elements $z_i$ and $z_j$ are compared at most once, iff $z_i$ or $z_j$ is the first to be chosen among $z_i, \cdots, z_j$
- The probability above (any indicated two elements $z_i$ and $z_j$ are compared) is $\frac{2}{j-1+1}$

$$\implies E_{\text{Num of comparisons made}} = \sum_{i<j} \frac{2}{j-1+1} = O(n \log n)$$

### 4.2.4 Find the i-th Smallest Element Using Quicksort

**Example: Find the i-th Smallest Element**
Given an unsorted array $A[1 \dots n]$ and an integer $i$, return the $i$-th smallest element of $A[1 \dots n]$.

**Idea:**
1. Choose a Pivot $x$ from $A[p \dots r]$
2. Partition $A$ around $x$. (linear time)
3. After partitioning, pivot $x$ will be at known location $q$
   If $i = q - p + 1$, then $x$ is the actual solution
   If $i < q - p + 1$, then the $i$-th element of $A[p \dots r]$ is the $i$-th element of $A[p \dots q-1]$, solve recursively
   If $i > q - p + 1$, then the $i$-th element of $A[p \dots r]$ is the $j = (i - q + p - 1)$-th element of $A[q+1 \dots r]$, solve recursively

**Algorithm 16:** i-th Smallest Element

`Select`$(A, p, r, i)$
**if** $p = r$ **then**
  **return** $A[p]$
**end**
Randomly choose an element in $A[p \dots r]$ as the pivot and swap it with $A[r]$
$q \leftarrow$Partition$(A, p, r)$
$k \leftarrow q - p + 1$
**if** $i = k$ **then**
  **return** $A[q]$
**end**
**else if** $i < k$ **then**
  **return** `Select`$(A, p, q - 1, i)$
**end**
**else**
  **return** `Select`$(A, q + 1, r, i - k)$
**end**
First Call: `Select`$(A, 1, n, i)$

### 4.2.5 Expected Running Time for Finding the i-th Smallest Element

• A pivot is "good" if it's between the 25%- and 75%-percentile of sorted $A$, eliminating at least 1/4 of the array. The probability for such "good" pivot is 1/2.

• Let $i$-th stage be the time between the $i$-th good pivot (not including) and the $(i+1)$-st good pivot (including), $i = 0, 1, 2, \cdots$, then the expected pivots selected within a stage is 2.

• Let $Y_i$ = the running time of $i$-th stage, $X_i$ = the num. of pivots (recursive calls) in $i$-th stage. Then $Y_i \leq X_i(3/4)^i n$.

$$\implies E[Y_i] \leq E[X_i(3/4)^i n] = 2(3/4)^i n \implies \text{Expected Total Running Time} \leq E[\sum_i Y_i] \leq \sum_i 2(3/4)^i n = O(n)$$

**Example: i-th Smallest Element in Two Sorted Arrays**

Given two sorted arrays $A1$ and $A2$ of sizes $m$ and $n$. Design an algorithm to find the $k$-th smallest element in the union of the elements in $A1$ and $A2$ ($k \leq m + n$).

**Algorithm 17:** i-th Smallest Element in Two Sorted Arrays

`Search`$($*array* $A1$, *array* $A2$, *start1* $1$, *end1* $k$, *start2* $1$, *end2* $k$, *Order* $k)$
**Main Idea:** Compare elements $A1[k/2]$ and $A2[k/2]$
**if** $A1[k/2] < A2[k/2]$ **then**
  Eliminate first half of $A1$
  **return** `Search`$(A1, A2, k/2 + 1,$ *end1*, *start2*, *end2*, $k/2)$
**end**
**else**
  Eliminate first half of $A2$
  **return** `Search`$(A1, A2,$ *start1*, *end1*, $k/2 + 1,$ *end2*, $k/2)$
**end**

**Time Complexity:** $\Theta(\log k)$

## 4.3 Heapsort

### 4.3.1 Priority Queues

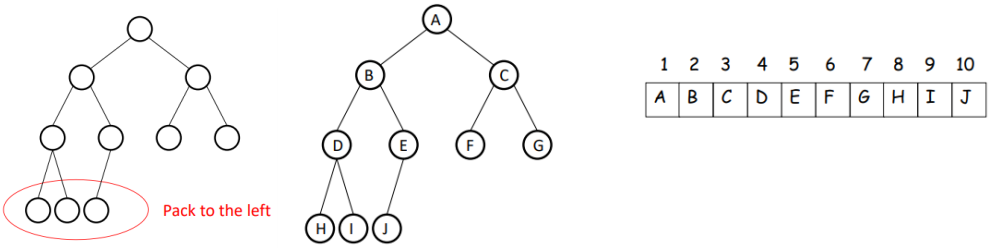
**Main Idea:** Processing the shortest job first - Extracting the smallest element from the queue.
A Priority Queue is an abstract data structure that supports two operations: Insert & Extract-Min.
**Implementations:**
1. Unsorted list + a pointer to the smallest element: $O(1)$ Insert & $O(n)$ Extract-Min
2. Sorted doubly linked list + a pointer to first element: $O(n)$ Insert & $O(1)$ Extract-Min

### 4.3.2 Binary Heap Implementation

- All levels are full except possibly the lowest level
- If the lowest level is not full, then nodes must be packed to the left
- The value of a node is at least the value of its parent —Min-heap
- Both Insert & Extract-Min can be done in $O(\log n)$ time

> **Notice**
> The binary tree here is DIFFERENT from the Binary Search Tree, which requires ALL left nodes $<$ parent, while ALL right nodes $>$ parent.

**Array Implementation of Heap**
- The root is in array position 1
- For any element in array position $i$, the left child is in position $2i$, the right child is in position $2i+1$, the parent is in position $\lfloor i/2 \rfloor$




### 4.3.3 Heapsort

**Insert**
- Add the new element to the next available position at the lowest level.
- Restore the min-heap property if violated.

**Algorithm 18:** Add item $x$ to heap $A[1 \dots i-1]$

```
Insert(x, i)
A[i] ← x
j = i
while j > 1 and A[j] < A[⌊j/2⌋] do // A[j] is less than its parent
  | Swap A[j] and A[⌊j/2⌋]
end
j = ⌊j/2⌋
```

**Time Complexity:** $O(\log n)$

**Extract-Min:** Should preserve both min-heap property & completeness

• Copy the last element to the root (overwrite).

• Restore the min-heap property by percolating (or bubbling down): if the element is larger than either of its children, then interchange it with the smaller of its children.

**Algorithm 19:** Remove the smallest item $A[1]$ in the heap $A[1 \dots i]$

```
Extract-Min(i)
Output(A[1])
Swap A[1] and A[i]
A[i] = ∞, j = 1, l = A[2j], r = A[2j + 1] // Left & Right Children
while A[j] > min(l, r) do  // if A[j] is larger than a child, swap with the smaller child
    if l < r then
        Swap A[j] with A[2j], j = 2j
    end
    else
        Swap A[j] with A[2j + 1], j = 2j + 1
    end
    l = A[2j], r = A[2j + 1]
end
```

**Time Complexity:** $O(\log n)$

**Total Time Complexity:** Build a binary heap of $n$ elements & Perform $n$ Extract-Min operations: $O(n \log n)$

**Example: Merging $k$ Sorted Arrays**

Suppose that you have $k$ sorted arrays, each with $n$ elements, and you want to combine them into a single sorted array of $kn$ elements.

**Solution 1:** Merge the first two arrays, then merge it with the third, and so on. Time Complexity $= \sum_{i=2}^{k} in = O(k^2 n)$

**Solution 2:** Divide recursively $k$ sorted arrays into two parts, conduct the merging for the subproblems. $T(k) = 2T(k/2) + kn \implies$ Time Complexity $= T(k) = O(kn \log k)$

**Solution 3 (Heapsort):** Insert the first element of each array into an empty min-heap. Extract-min every time and insert the next item of in the same array as the one being extracted. Time Complexity $= O(kn \log k)$

[**Operation Implementation**]

**Decrease-Key:** Decreases the value of one specified element (Used in Dijkstra's Algorithm)

**Modification of the heaps to support it in $O(\log n)$ time:** Change the heap tree to a binary search tree.

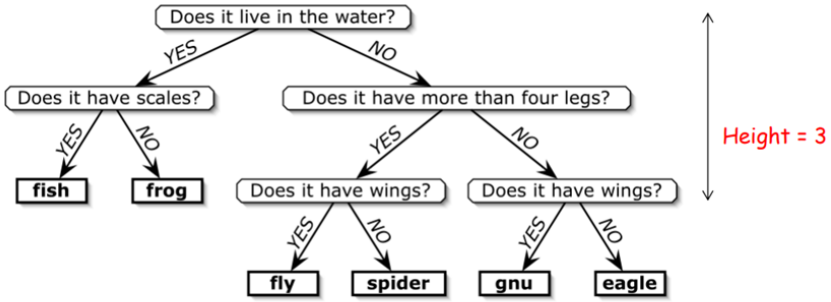
For more information, see: Binary Heap (Wikipedia)

Some websites markdown: Zhihu Web 1 Web 2

## 4.4 Linear-Time Sorting

### 4.4.1 Decision Trees and Lower Bounds

**Decision Tree Model**




A decision tree to choose one of six animals.

**Fact:** A binary tree with $n$ leaves must have height $\Omega(\log n)$.

**Theorem:** Any algorithm for finding location of given element in a sorted array of size $n$ must have running time $\Omega(\log n)$ in the decision-tree model.

**Theorem:** Any **comparison-based sorting algorithm** (only by using comparisons without using thetr accurate values) requires $\Omega(n \log n)$ time.

Given $n$ numbers, there are $n!$ possible permutations, resulting in the tree height being $\Omega(\log(n!))$. Thus, the time complexity is bounded as $\Omega(\log(n!)) = \Omega(n \log n)$.

### 4.4.2 Linear-time Sorting

**Counting-Sort**

- Assumes that the elements are integers from 1 to $k$

**Algorithm 20:** Counting-Sort Algorithm

**Input:** $A[1 \cdots n]$ where $A[j] \in 1, 2, \cdots, k$
**Output:** $B[1 \cdots n]$, sorted
`Counting-Sort`$(A, B, k)$
Let $C[1 \cdots k]$ be a new array
**for** $i \leftarrow 1$ *to* $k$ **do** // Initialize Counters
  $C[i] \leftarrow 0$
**end**
**for** $j \leftarrow 1$ *to* $n$ **do** // Count the number of each element
  $C[A[j]] \leftarrow C[A[j]] + 1$
**end**
**for** $i \leftarrow 2$ *to* $k$ **do** // Count the accumulative number of elements
  $C[i] \leftarrow C[i] + C[i-1]$
**end**
**for** $j \leftarrow n$ *to* $1$ **do** // Move the items into proper location
  $B[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$
**end**

**Time Complexity:** $\Theta(n + k)$
**Space Complexity:** $\Theta(n + k)$

**Radix-Sort**

**Algorithm 21:** Radix-Sort Algorithm

**Input:** An array of $n$ numbers, each has at most $d$ digits
**Output:** A sorted array
`Radix-Sort`$(A, d)$
**for** $i \leftarrow 1$ *to* $d$ **do**
  Use Counting-Sort to sort array $A$ on digit $i$
**end**

**Time Complexity:** $\Theta(d(n+k))$

## 4.5 Sorting Reprise & Comparison

| | Insertion Sort | Merge Sort | Quick Sort | Heap Sort | Radix Sort |
|---|---|---|---|---|---|
| Running Time | $\Theta(n^2)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(d(n+k))$ |
| Randomized | No | No | Yes | No | No |
| Working Space | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\log n)$ | $\Theta(1)$ | $\Theta(n+k)$ |
| Comparison-Based | Yes | Yes | Yes | Yes | No |
| Stable | Yes | Yes | No | No | Yes |
| Cache Performance | Good | Good | Good | Bad | Bad |
| Parallelization | No | Excellent | Good | No | No |

# 5 Greedy Algorithms

## 5.1 Basic Greedy Algorithms

A greedy algorithm always makes the choice that looks best at the moment and adds it to the current partial solution.

Greedy algorithms don't always yield optimal solutions, but when they do, they're usually the simplest and most efficient algorithms available.

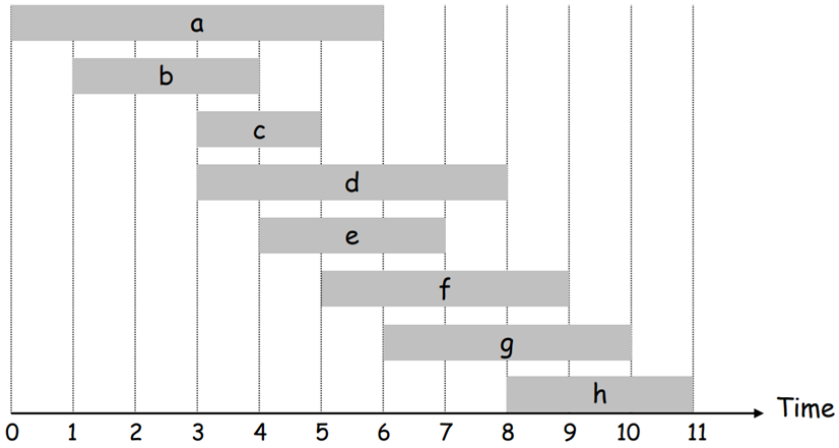
Usually, greedy algorithms involve a sorting step that dominates the total cost.

**Example: Interval Scheduling**

Job $j$ starts at $s_j$ and finishes at $f_j$.

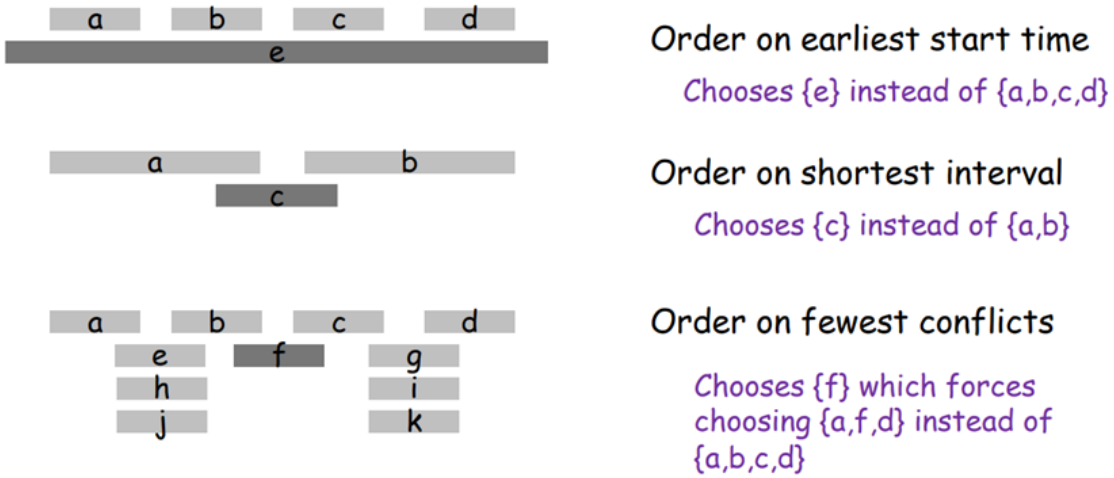
Two jobs are compatible if they don't overlap.

Goal: Find maximum size subset **(Note: Not the longest duration)** of mutually compatible jobs.



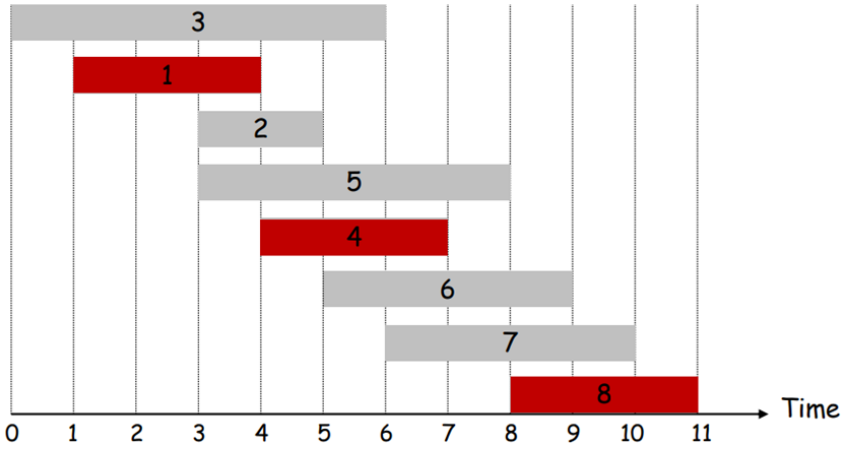

**Three Possible Rules:**

- [Earliest Start Time] Consider jobs in increasing order of start time $s_j$.
- [Shortest Interval] Consider jobs in increasing order of duration $f_j - s_j$.
- [Fewest Conflicts] Consider jobs in increasing order of number of conflicts $c_j$ with other jobs.




All three rules may not yield optimal solutions.

**Correct Greedy Algorithm:**
Consider jobs in increasing order of finish time $f_j$. Take each job if compatible with all previously taken jobs.




**Algorithm 22:** Greedy Algorithm

Sort jobs by finish time $f_j$, i.e. $f_1 \leq f_2 \leq \cdots \leq f_n$
$A \leftarrow \emptyset$, $last \leftarrow 0$
**for** $j \leftarrow 1$ *to* $n$ **do**
  **if** $s_j \geq last$ **then**
    $A \leftarrow A \cup \{j\}$
    $last \leftarrow f_j$
  **end**
**end**
**return** $A$

See Page 13 of 11_Greedy_all.pdf for the proof.

**Example: The Fractional Knapsack Problem**
Input: Set of $n$ items: item $i$ has weight $w_i$ and value $v_i$, and a knapsack with capacity $W$.
Goal: Find $0 \leq x_1, \ldots, x_n \leq 1$ to maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$. (Note: $x_i$ is the fraction of item $i$ to be taken)
- The $x_i$ must be 0 or 1: The 0/1 knapsack problem.
- The $x_i$ can be any value in [0,1]: The fractional knapsack problem.

**Algorithm 23:** Fractional-Knapsack Algorithm

`Fractional-Knapsack`$(w, v, W, x[])$
Sort $\frac{v_i}{w_1}$ so that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$
**for** $i \leftarrow 1$ *to* $n$ **do**
  **if** $w_i < W$ **then**
    $x_i \leftarrow 1$
    $W \leftarrow W - w_i$
  **end**
  **else**
    $x_i \leftarrow W/w_i$
    **return** $x$
  **end**
**end**
**return** $x$

**Example: Hiking Problem**

Suppose you are going on a hiking trip over multiple days. For safety reasons you can only hike during daytime. You can travel at most $d$ kilometers per day, and there are $n$ camping sites along the hiking trail where you can make stops at night. Assuming the starting point of the trail is at position $x_0 = 0$, the camping sites are at locations $x_1, \cdots, x_n$, and the end of the trail is at position $x_n$. Design an $O(n)$-time algorithm to find a plan that uses the minimum number of days to finish the trip. You can assume that $x_{i+1} - x_i \leq d$ for all $i$ (otherwise there is no solution).

**Idea:** For each day $i$, stop at the furthest camping site, i.e. stop at the largest $x_j$ such that $x_j$ minus the start location of day $i$ is at most $d$.

**Example: Interval Partitioning**

Lecture $j$ starts at $s_j$ and finishes at $f_j$. Two lectures are compatible if they don't overlap. Goal: Find minimum number of classrooms to schedule all lectures.

**Idea:** Sort the class by start time. Insert in order, opening new classroom when needed.

## 5.2 Huffman Coding

**Example: Huffman Coding**

**Input:** A set of characters and their frequencies.

**Goal:** Find a prefix-free binary code (i.e. no codeword is a prefix of another) with minimum expected codeword length.

**Rigorous Problem Definition:** Given an alphabet $A$ of $n$ characters $a_1, \cdots, a_n$ with weights $f(a_1), \cdots, f(a_n)$, find a binary tree $T$ with $n$ leaves labeled $a_1, \cdots, a_n$ such that

$$B(T) = \sum_{i=1}^{n} f(a_i) \cdot d_T(a_i)$$

is minimized, where $d_T(a_i)$ is the depth of leaf $a_i$ in $T$.

**Greedy Idea:**

- Pick two characters $x, y$ from $A$ with the smallest weights
- Create a subtree that has these two characters as leaves
- Label the root of this subtree as $z$ and set $f(z) \leftarrow f(x) + f(y)$
- Remove $x$ and $y$ from $A$ and add $z$ to $A$
- Repeat until only one character is left

See Page 13 of 12_Huffman_all.pdf for the proof.

## 5.3 Stable Matching

**Example: Stable Matching Problem**

**Problem:** Finding a stable matching between two equally sized of elements given an ordering of preferences for each element. A matching is a bijection from the elements of one set to the elements of the other set.

**Input:** Two sets $A$ and $B$ of equal size $n$, each with a strict ordering of preferences for the elements of the other set.

**Output:** A matching $M$ between $A$ and $B$ such that there is no pair $(a, b)$ and $(a', b')$ in $M$ such that $a$ prefers $b'$ to $b$ and $b'$ prefers $a$ to $a'$.

**Gale-Shapley Algorithm [Gale-Shapley 1962]**

Intuitive algorithm that guarantees to find a stable matching.
Also known as the deferred acceptance algorithm or propose-and-reject algorithm.

**Algorithm 24:** Gale-Shapley Algorithm

```
Gale-Shapley()
Initialize each participant (employers and applicants) to be free.
while some employer is free and has an applicant to send an offer to do
    Choose such an employer e
    a = 1st applicant on e's list to whom e has not yet sent an offer
    if a is free then
        assign e and a to be matched
    end
    else if a perfers e to the current employer e' then
        assign e and a to be matched, and e' to be free
    end
    else
        a rejects e
    end
end
```

**Time Complexity:** Algorithm terminates after at most $n^2$ iterations of while loop, as each employer can only send at most 1 offer to each of the $n$ applicants.

**Efficient Implementation Details:**

To evaluate whether a perfers e to the current employer e', we can create inverse of preference list of employers for each applicant. Such structure yields a constant time access for comparison and requires $O(n^2)$ (Same as the current time complexity) for all applicants.

**Note:**

• For a given problem instance, several stable matchings might exist. All executions of Gale-Shapley (for different orders in which employers send offer) yield the same stable matching result, which is optimal for the employers. (i.e. each employer gets the best possible partner in any possible stable matching)

• To get an applicant optimal algorithm, let applicants apply and employers accept/reject (apply the algorithm in reverse order)

• Suppose the graph is not bi-partite. i.e. for each of the $2n$ person, we have the list of preferences, over the remaining $2n-1$ person, then there may not be a stable matching.

## Three Different Proof Techniques

1. Modifying an Optimal Solution into Greedy (cost common)
   - Did this for **Interval Scheduling** & **Fractional Knapsack** and several exercises
   - Start (conceptually) with different **G**(greedy) and **O**(optimal) solutions
     - Show how **O** can be modified to **O'** that is still optimal but closer to **G**
     - Repeat, until have created optimal solution that is exactly **G**

2. Lower Bound Technique
   - Did this for **Interval Partitioning**
   - For problems trying to find **minimum** solution (can be modified for max)
     - Define value **L** that is a lower bound on ANY feasible solution
     - Show that Greedy solution has value **L**

3. Algorithm-specific Techniques
   - Inductive Proof for **Huffman Coding**
   - Proof by Contradiction for **Stable Matching**

# 6 Dynamic Programming

## 6.1 1D Dynamic Programming

## 6.2 2D Dynamic Programming

## 6.3 Dynamic Programming over Intervals

## 6.4 Summary of Dynamic Programming

# 7 Graph Algorithms

## 7.1 Graph Introduction

### 7.1.1 Finding Euler Path

**Graph** $G = (V, E)$

**Assumption:** In most cases we assume simple graphs, i.e. no self-loops and no parallel edges. (at most one or two edges between any two vertices in undirected or directed graph)

- **Undirected Graph:** $\sum_{v \in V} \deg(v) = 2E$
- **Directed Graph:** $\sum_{v \in V} \deg^{out}(v) = \deg^{in}(v) = E$

**Theorem:** A (multi)graph has such a path (known as an Euler path) iff it contains exactly 0 or 2 vertices with an odd degree.

**Hierholzer's Algorithm:** Find an Euler path in $O(E)$ time.

**Algorithm 25:** Hierholzer's Algorithm

```
u ← any odd-degree vertex
if u does not exist then
    u ← any vertex
end
FindPath(u)
while there are still edges not yet taken do
    u ← any previously seen vertex that is endpoint of an unused edge
    p ←FindPath(u)
    insert p into the existing path at u
end

FindPath(u):
while u has an edge not yet taken do
    take that edge
    u ← v
end
```

### 7.1.2 Graph Representation

**Adjacency List:** For each vertex $v$, store a list of all vertices $u$ such that $(v, u) \in E$.

- **Space Complexity:** $O(V + E)$
- More commonly used, as most graphs are sparse

**Adjacency Matrix:** $A[i][j] = 1$ if $(i, j) \in E$, $A[i][j] = 0$ otherwise.

- **Space Complexity:** $O(V^2)$

### 7.1.3 Path and Connectivity

**Path:** A sequence of vertices $v_1, v_2, \cdots, v_k$ such that $(v_i, v_{i+1}) \in E$ for all $i = 1, \cdots, k-1$. The length of such path is $k-1$.

**Simple Path:** A path with no repeated vertices.

**Undirected Connected Graph:** A graph is connected if there is a path between every pair of vertices.

**Directed Strongly Connected Graph:** A directed graph is strongly connected if there is a path from $u$ to $v$ and a path from $v$ to $u$ for every pair of vertices $u$ and $v$.

**Directed Weakly Connected Graph:** A directed graph is weakly connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.

**Cycle:** A cycle is a path $v_1, v_2, \cdots, v_k$ such that $(v_1, v_k) \in E$.

**Distance:** The distance between two vertices $u$ and $v$ is the length of the shortest path between them.

### 7.1.4 Trees

**Tree:** A tree is an undirected graph that is connected and acyclic (not containing a cycle).

**Rooted Tree:** A rooted tree is a tree in which one vertex has been designated the root. Every edge is directed away from the root.

**Leaf:** A leaf is a vertex of degree 1.

**Parent, Child, Ancestor, Descendant:** If $(u, v) \in E$, then $u$ is the parent of $v$ and $v$ is the child of $u$. $u$ is an ancestor of $v$ and $v$ is a descendant of $u$.

**Forest:** A forest is an undirected graph that is acyclic.

**Theorem.** *For any undirected graph $G = (V, E)$, the following statements are equivalent:*

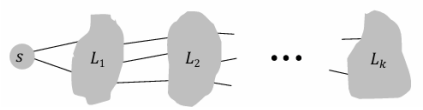
- *$G$ is a tree*
- *$G$ is connected and $|E| = |V| - 1$*
- *$G$ is acyclic and $|E| = |V| - 1$*
- *Any two vertices of $G$ are connected by a unique simple path*
- *$G$ is connected and removing any edge disconnects $G$*
- *$G$ is acyclic and adding any edge creates a cycle*
- *$G$ is connected and has no cycles*
- *$G$ is acyclic and has $|E| = |V| - 1$*
- *$G$ is connected, and for any vertices $u$ and $v$, there is a unique simple path from $u$ to $v$*

## 7.2 Breadth First Search

**Idea:** Explore outward from $s$ in all possible directions, adding nodes one "layer" at a time.

**Property:** $L_{i+1}$ contains all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$.

**Theorem:** For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer, where the distance from $u$ to $v$ is the number of edges on the shortest path from $u$ to $v$.




**Algorithm 26:** BFS Algorithm

```
BFS(G, s)
foreach u ∈ V − {s} do // Initialize the discoverys state, distance and parent except the starting point
    u.color ← WHITE
    u.d ← ∞
    u.p ← NIL
end
s.color ← GRAY
s.d ← 0
Initialize an empty queue Q // Temporary storage of the vertices to be processed
Enqueue(Q, s)
while Q ≠ ∅ do
    u ← Dequeue(Q) // Dequeue the first element in the queue (FIFO), reduce the required space for the queue
    foreach v ∈ G.Adj[u] do
        if v.color = WHITE then
            v.color ← GRAY // Discovered but unprocessed
            v.d ← u.d + 1
            v.p ← u
            Enqueue(Q, v)
        end
    end
    u.color ← BLACK // Processed
end
```

**Time Complexity:** $\sum_u (1 + \deg u) == \Theta(E)$ if the graph is connected.

To transfer through non-connected graphs, simply check if any vertex is still white after the first BFS, if so, run BFS again starting from that vertex.

### 7.2.1 Strong Connectivity in Directed Graphs

**Def.** Node $u$ and $v$ are mutually reachable if there is a path from $u$ to $v$ and a path from $v$ to $u$.

**Def.** A graph is strongly connected if every pair of nodes is mutually reachable.

**Observation 1:** A graph $G$ is strongly connected if there exists such a vertex $v$ that every other vertex forms a mutually-reachable pair with $v$. In this case, every other vertex has the similar property.

**Observation 2:** A graph $G$ is strongly connected if and only if every vertex $v$ is mutually reachable with every other vertex.

**Idea to check strong connectivity in directed graphs:** Run BFS from $v$, then **REVERSE ALL THE EDGES** and run BFS again from $v$. If all vertices are visited in both BFS, then the graph is strongly connected.

**Note:** Given the adjacency list, one can convert $G$ to its reverse $G^T$ within $O(V + E)$ time.

**Time Complexity:** $O(E)$

### 7.2.2 Strongly Connected Components in $O(VE)$ Time

**Algorithm 27:** Strongly Connected Components

`StronglyConnectedComponents(`$G$, $s$`)`
create $G^T$ // Reverse all the edges
**while** $G$ *has unvisited vertices* **do**
  Run BFS on $G$ starting from an unvisited vertex $v$
  Run BFS on $G^T$ starting from an unvisited vertex $v$
  $C \leftarrow$ all visited vertices in both BFSs
  output $C$ as a strongly connected component
  remove $C$ and all its edges from $G$ and $G^T$
**end**

**Time Complexity:** $O(VE)$

### 7.2.3 Strongly Connected Components in $O(V + E)$ Time

Let $C$ and $C'$ be distinct strongly connected components in directed graph $G = (V, E)$.

**Observation 1:** Let $u, v \in C$ and $u', v' \in C'$. If there is an edge from $u$ to $u'$, then there is no edge from $v'$ to $v$.

**Observation 2:** Let $d(U) = \min u.d : u \in U$ being the discovery time and $f(U) = \max u.f : u \in U$ being the finish time of a set of vertices $U$. Suppose $u \in C$ and $v \in C'$ with $(u, v) in E$. Then $f(C) > f(C')$. (i.e. $C$ is finished later than $C'$)

**Observation 3:** Suppoose $f(C) > f(C')$, then $E^T$ contains no edge from $C$ to $C'$.

**Key Point:** In general, when the depth-first search of $G^T$ in line 3 visits any strongly connected component, any edges out of that component must be to components that the search has already visited. Each depth-first tree, therefore, corresponds to exactly one strongly connected component.

**Algorithm 28:** Strongly Connected Components Modified

`StronglyConnectedComponents(`$G$, $s$`)`
call $DFS(G)$ to compute finish times $u.f$ for each vertex $u$
create $G^T$ // Reverse all the edges
call $DFS(G^T)$, but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

**Note:** See next subsection for the DFS algorithm.

**Time Complexity:** $\Theta(V + E)$

**Example: Check Bipartite Graph**

**Problem:** Given an undirected graph $G = (V, E)$, check if $G$ is bipartite.

**Idea:** Run BFS from any vertex. Use $d[v]$ to store the distance from the root to any other vertex $v$. Set $S$ to be the set of all vertices with even $d[v]$, and $V - S$ to be the set of all vertices with odd $d[v]$.

- $G$ is partite if and only if all edges $(u, v)$ in the graph satisfy that the parity of $d[u]$ and $d[v]$ are different.
- Alternatively, whether $G$ is bipartite is equivalent to whether $G$ does not contain any odd cycle. To check so, simply do BFS to find if there is any edge, both of whose endpoints are in the same layer.

## 7.3 Depth First Search

**Algorithm 29:** DFS Algorithm

```
DFS(G)
foreach u ∈ V do
    u.color ← WHITE
    u.p ← NIL
end
time ← 0
foreach u ∈ V do
    if u.color = WHITE then
        DFSVisit(u)
    end
end

DFSVisit(u)
time ← time + 1
u.d ← time
u.color ← GRAY
foreach v ∈ G.Adj[u] do
    if v.color = WHITE then
        v.p ← u
        DFSVisit(v)
    end
end
u.color ← BLACK
time ← time + 1
u.f ← time
```

### 7.3.1 Cycle Detection in Undirected Graphs

**Example: Cycle Detection**

**Problem:** Given an undirected graph $G = (V, E)$, check if $G$ contains a cycle.

**Idea:** A tree (connected and acyclic) has $|E| = |V| - 1$. If $G$ contains a cycle, then $|E| > |V| - 1$. Therefore, we can check if $|E| > |V| - 1$ after running DFS.

**Time Complexity:** $\Theta(V + E)$

**Example: Cycle Detection**

**Problem:** Given an undirected graph $G = (V, E)$, check if $G$ contains a cycle. If so, find a cycle.

**Def.** All edges, after running BFS or DFS, can be classified into three categories: tree edges, back edges and forward/cross edges.

**Tree edges:** Traversed by BFS/DFS.

**Back edges:** Connecting a node with one of its ancestors in the BFS/DFS tree which is not a tree edge.

**Forward/Cross edges:** Connecting two nodes with no ancestor/descendant relationship in the BFS/DFS tree.

**Theorem.** *In a BFS on an undirected graph, there are no back edges.*

**Theorem.** *In a DFS on an undirected graph, there are no cross edges.*

**Idea:** Run DFS. If there is a back edge, then there is a cycle. To find a cycle, simply find the path from the current vertex to the ancestor vertex. If no, then there is no cycle.

**Time Complexity:** $\Theta(V)$

**Algorithm 30:** DFS Algorithm for Cycle Detection

```
CycleDetection(G)
foreach u ∈ V do
    u.color ← WHITE
    u.p ← NIL
end
time ← 0
foreach u ∈ V do
    if u.color = WHITE then
        DFSVisit(u)
    end
end
return "No Cycle"

DFSVisit(u)
u.color ← GRAY
foreach v ∈ G.Adj[u] do
    if v.color = WHITE then
        v.p ← u
        DFSVisit(v)
    end
    else if ) then // back edge (u, v
        v ≠ u.p
    end
    output "Cycle detected"
    while u ≠ v do
        output u
        u ← u.p
    end
    output v
    return
    u.color ← BLACK
end
```

## 7.4 Topological Sort

**Def.** Directed acyclic graph (DAG) is a directed graph with no cycles.

**Def.** A topological sort of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering.

**Idea:** A DAG must contain at least one vertex with no incoming edges. Output a vertex $u$ with in-degree zero in current graph. Remove this vertex and all its outgoing edges. Repeat until all vertices are removed.

**Algorithm 31:** Topological Sort

```
TopologicalSort(G)
Initialize an empty queue Q
foreach u ∈ V do
    if in-degree(u) = 0 then
        Enqueue(Q, u) // Enqueue all vertices with in-degree 0
    end
end
while Q ≠ ∅ do
    u ← Dequeue(Q)
    output u
    foreach v ∈ G.Adj[u] do
        in-degree(v) ← in-degree(v) - 1 // remove u's outgoing edges
        if in-degree(v) = 0 then
            Enqueue(Q, v)
        end
    end
end
```

For each vertex, we need to check all its outgoing edges $\sum_{v \in V} out - degree(v) = E$.

**Time complexity:** $O(V + E)$

**Alternative Implementation Using DFS:** Apply DFS from a node with in-degree 0. Output the vertices in reverse order of their finish times.

## 7.5 Minimum Spanning Tree

### 7.5.1 Definition

**Spanning Tree:** Given a connected undirected graph $G = (V, E)$, a spanning tree of $G$ is a subgraph that is a tree and connects all the vertices together.

**Minimum Spanning Tree:** A minimum spanning tree (MST) or minimum weight spanning tree $S$ is a subset of the edges $T \subseteq E$ of a connected, edge-weighted (un)directed graph that connects all the vertices together, with the minimum possible total edge weight $\sum w(e)$.

### 7.5.2 Prim's Algorithm

**Idea:**
- Initialize the set $S$ to have any single vertex $s$. Initialize $T$ to be empty.
- Add minimum cost edge $e = (u, v)$ with $u \in S$ and $v \notin S$ to $T$.
- Add $v$ to $S$.
- Repeat until $S = V$.

**Algorithm 32:** Prim's Algorithm

```
Prim(G, root)
foreach u ∈ V do
    u.key ← ∞
    u.p ← NIL
    u.color ← WHITE
end
root.key ← 0
Initialize an empty min-priority queue Q // e.g. Use min-heap to implement so that we can conduct extract-min
 method
while Q ≠ ∅ do
    u ← Extract-Min(Q)
    u.color ← BLACK
    foreach v ∈ G.Adj[u] do
        if v.color = WHITE and w(u, v) < v.key then
            v.p ← u
            v.key ← w(u, v) // Let v.key specify the minimum weight of any edge connecting v to a vertex already
             processed.
            Decrease-Key(Q, v, w(u, v)) // Changes the value in the min-priority queue (or min-heap), same as the
             above line. See page 17 for Decrease-Key method.
        end
    end
end
```

**Time Complexity:** $O(E \log V)$

### 7.5.3 The Cut Lemma

**Assumption:** All edge weights are distinct.

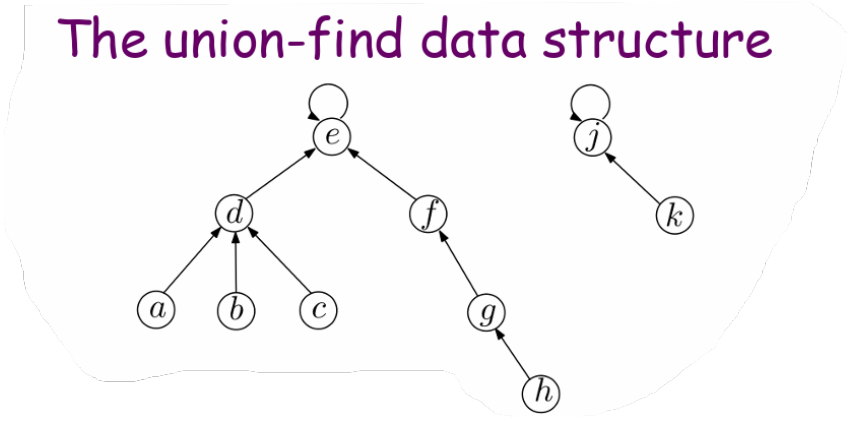
**Lemma.** *Let $S$ be any subset of nodes, and let $e = (u, v)$ be the minimum weight edge connecting $S$ to $V - S$. Then, every minimum spanning tree must contain the edge $e$.*

### 7.5.4 Kruskal's Algorithm

**Idea:** Start with an empty tree $T$. Add edges in increasing order of weight (according to the cut lemma), skipping those whose addition would create a cycle.

• To check whether the new edge forms a cycle, using DFS would cost a total time of $O(E \cdot V)$. Alternatively, we can use "union-find" data structure to check it.

**Union-Find Data Structure:**
- Maintain a collection of disjoint sets that support the following operations:
- **Find-Set($u$):** Given a node $u$, find a set which contains $u$.
- **Union($u, v$):** Given two nodes $u$ and $v$, merge the two sets containing $u$ and $v$.




**Methods of Union-Find Data Structure:**
- Initalization: Make-Set($x$): Set the parent to itself, and height to 0
- Find-Set($x$): Return the root of the tree containing $x$ by calling its parent.
- Union($x, y$): Let $a, b$ be the roots of the trees containing $x$ and $y$ with $a.height \leq b.height$. Make the shorter tree $a$ subtree of the taller tree by setting $a.p = b$.

**Theorem.** *The running time fo Find-Set and Union are both $O(\log n)$. Specifically, for a tree with height $h$, it contains at least $2^h$ nodes.*

**Algorithm 33:** Kruskal's Algorithm

```
Kruskal(G)
foreach u ∈ V do
    Make-Set(u)
end
sort the edges of G into increasing order by its weight
foreach edge (u, v) ∈ E in the above order do
    if Find-Set(u) ≠ Find-Set(v) then
        output edge u, v
        Union(u, v)
    end
end
```

**Time Complexity:** $O(E \log E + E \log V) = O(E \log V)$
**Note:** If the edges are already sorted, then the time complexity can be reduced to be closed to $O(E)$.
**Remark:**
- When the distinct-weight-assumption is removed, the only thing that needs to be changed is that, instead of choosing **the** smallest cost edge, we choose **a** smallest cost edge that does not create a cycle.
- If the graph $G$ is an undirected connected graph with distinct edge weights. Then, there is a unique MST for $G$.

**Example: Bottleneck Weight of MST**

**Theorem:** Let $G = (V, E)$ be a connected undirected graph with weights on the edges. The bottleneck weight of any spanning tree $T$ of $G$ is the maximum weight of an edge in $T$. Prove that the minimum spanning tree (MST) minimizes the bottleneck weight over all spanning trees.

**Idea:** Suppose the heaviest weight $e$ with weight $w(e)$ connects two components $A$ and $B$, with $A \cup B = V$. Assume another MST $T'$ has a smaller bottleneck weight. Then, there must be an edge $e'$ in $T'$ with weight $w(e')$ that connects $A$ and $B$. Remove $e$ from $T$ and add $e'$ to $T$ forms a new spanning tree $T''$ with smaller total weight. This contradicts the assumption that $T$ is an MST.

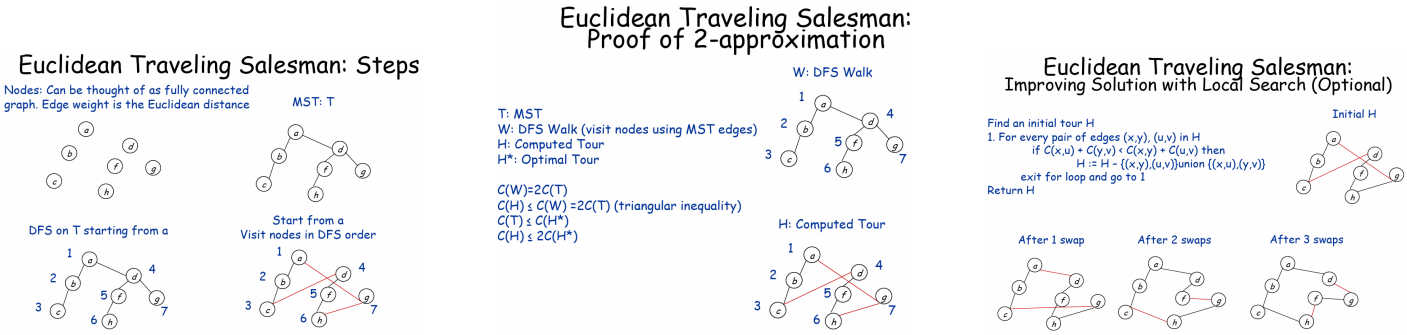
**Example: Euclidean Traveling Salesman Problem**

**Input:** Undirected graph $G = (V, E)$ with edge cost $C(e)$ for each $e \in E$.

**Output:** A minimum-cost cycle that visits each vertex exactly once.

**Notes: Euclidean** Traveling Salesman implies that the cost is the Euclidean distance between the two vertices, which satisfies the triangle inequality: $C(u, v) \leq C(u, w) + C(w, v)$.

**Remark:** This problem is NP-hard. There is no known polynomial-time algorithm that solves it. There are two approximate solutions using MST.




Euclidean Traveling Salesman Problem

## 7.6 Shortest Paths

## 7.7 Maximum Flow and Bipartite Matchings

# 8 AVL Trees

# 9 Basic String Matching

# 10 Hashing

# 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 $\Theta$ 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)

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

**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) Main Idea: Recursively solve the problem by reducing the $k \times k$ array to $(k-1) \times (k-1)$ array. Then generate all the binary array of the remaining $1 \times (k-1)$, $(k-1) \times 1$ and $1 \times 1$ array. For each kind of array for the $(k-1) \times (k-1)$ array, we insert the remaining $(2k-1)$ elements into it, forming the $(k \times k)$ array.

**Algorithm 34:** All $n \times n$ Binary Arrays

`AllBinaryArrays(`$n$`)`
EmptyArray ← array[$n$][$n$]
ArrayList<Type=array[$n$][$n$]> ← [ ]
**if** $n$ = *1* **then**
  ArrayList.append(array[$n$][$n$] = 0, array[$n$][$n$] = 1) // Set the last element to 0 & 1, others remaining 0
  **return** *ArrayList*
**end**
ArrayList_1, ArrayList_2, ArrayList_3 ← `GenerateRemaining(`$n$`)`
PreviousArrayList[ ][2 : $n$][2 : $n$] ← `AllBinaryArrays(`$n-1$`)`
// Insert the remaining $(2n-1)$ elements into the $n \times n$ array, forming all kinds of $(n \times n)$ array
**foreach** *Combination of ArrayList_1, ArrayList_2, ArrayList_3* **do**
  ArrayList.append(PreviousArrayList[ ][2 : $n$][2 : $n$].set(PreviousArrayList[ ][1][2 : $n$] ← ArrayList_1,
  PreviousArrayList[ ][2 : $n$][1] ← ArrayList_2, PreviousArrayList[ ][1][1] ← ArrayList_3))
**end**
**return** *ArrayList*

/* Recursively generate all the kinds of $1 \times (n-1)$, $(n-1) \times 1$ and $1 \times 1$ array respectively */
`GenerateRemaining(`$n$`)`
ArrayList_1<Type=array[1][$n-1$]> ← [ ]
ArrayList_2<Type=array[$n-1$][1]> ← [ ]
ArrayList_3<Type=array[1][1]> ← [ ]
**if** $n = 2$ **then**
  **return** [[0], [1]], [[0], [1]], [[0], [1]] // To be assigned to $1 \times (n-1)$, $(n-1) \times 1$ and $1 \times 1$ array
**end**
ArrayList_1[ ][1][2 : $n-1$], ArrayList_2[ ][2 : $n-1$][1], ArrayList_3[ ][1][1] ← `GenerateRemaining(`$n-1$`)` //
Assign all possible cases of the previous arrays into all the array in the corresponing list
// Set the new element inserted to be 0 and 1
ArrayList_1 = ArrayList_1[ ][1][2 : $n-1$].set[ ][1][1] ← 0 + ArrayList_1[ ][1][2 : $n-1$].set[ ][1][1] ← 1
ArrayList_2 = ArrayList_2[ ][2 : $n-1$][1].set[ ][1][1] ← 0 + ArrayList_1[ ][2 : $n-1$][1].set[ ][1][1] ← 1
**return** *ArrayList_1, ArrayList_2, ArrayList_3*
First Call: `AllBinaryArrays(`$n$`)`

(b) **Recurrence:** Suppose $T(n) = T(n-1) + O(f(n))$

$f(n)$ contains generating the remaining $1 \times (n-1)$, $(n-1) \times 1$ and $1 \times 1$ arrays as well as inserting such arrays into the original $n \times n$ array. The first part requires $O(n)$ time as a simple recursion while the second part requires $O(n \times 2^n)$ time, considering the insertion time.

Thus,

$$T(n) = T(n-1) + O(n \times 2^n) = \sum_{k=1}^{n} k \times 2^k = O(n2^n)$$

**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.**

(a) The maximum number of 10-major elements is 9. If there were 10 10-major elements in an array of $n$ numbers, each 10-major elements should appear at least $\lfloor n/10 \rfloor + 1$ times, resulting in a total of at least $10\lfloor n/10 \rfloor + 10 > 10(n/10 - 1) + 10 = n$ numbers to be presented. For 9 10-major elements, supposing there are 100 numbers in total, each such elements appear 11 times would satisfy the requirement.

(b) Since the 10-major elements require the elements to appear at least $\lfloor n/10 \rfloor + 1$ times, we can infer that, if we cut the array into 10 subarrays, such elements must also be a 10-major elements in at least one of the subarray. Reporting such elements in the subarray to the previous recurrsion and checking if they are still 10-major elements in the previous recurrsion can help us get all the 10-major elements.

**Algorithm 35:** Get All 10-Major Elements

```
GetMajorElements(A[n])
if len(A) = 0 or 1 then
    return A
end
// Return A itself as it stores nothing or the only 10-major element
PossibleMajorElements ← [ ]
TrueMajorElements ← [ ]
for i ← 0 to 9 do
    start ← ⌈ni/10⌉
    end ← ⌈n(i + 1)/10⌉ − 1
    PossibleMajorElements.append(GetMajorElements(A[start : end]))
    // Check whether the elements in the PossibleMajorElements array are still 10-major elements in A
    foreach elements in PossibleMajorElements do
        Count its appearances in A, append it into TrueMajorElements if its appearances ≥ ⌊n/10⌋ + 1
    end
end
return TrueMajorElements
First Call: GetMajorElements(A[n])
```

Correctness: As any 10-major element must be a 10-major element in at least one of the subarray, we must get all the possible answers.

(c) **Recurrence:** Suppose $T(n) = 10T(n/10) + O(f(n))$ with $T(1) = 1$

$f(n)$ contains counting the appearance number of each element in PossibleMajorElements within the array $A$. We've derived that there are at most 9 10-major elements in any subarray, indicating that there are no more than $9 \times 10 = 90$ candidates stored in PossibleMajorElements. Comparing such 90 elements to the original array $A$ costs $O(n)$ time complexity. Thus, $f(n) = n$

$$T(n) = 10T(n/10) + O(n) \text{ with } T(1) = 1$$

(d)

$$T(n) = 10T(n/10) + O(n) = 100T(n/100) + 11n = 1000T(n/1000) + 111n = \cdots = 10^{\log_{10} n} T(1) + 100n \log_{10} n = O(n \log n)$$

# Homework 2

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

1. (20 points) Let $A[1..n]$ be an array that stores $n$ possibly non-distinct numbers.

   (a) (10 points) A *range query* specifies two integers $x$ and $y$ that form an interval $(x, y]$. The answer to a range query is the number of elements in $A$ that are greater than $x$ and less than or equal to $y$. There is NO requirement that $x$ and $y$ are elements of $A$.

   As an example if $A = [4, 5, 2, 6, 8, 3, 4, 4]$, the range query for the interval $(3, 6]$ would return 5 because there are five elements in $A$ that lie in $(3, 6]$, namely $A[1]$, $A[2]$, $A[4]$, $A[7]$, and $A[8]$. Similarly, the range query for the interval $(7, 10]$ would return 1.

   Describe an algorithm that generates a data structure $C$ such that $C$ uses $O(n)$ space and you can use $C$ to answer any range query in $O(\log n)$ time. Explain the running time of your algorithm for constructing $C$. Describe in detail how you use $C$ to answer a range query and explain why it takes $O(\log n)$ time.

   (b) (10 points) Suppose that the numbers in $A[1..n]$ come from the range $[1..k]$ for some positive integer $k$ given to you. A *range-sum query* specifies two integers $x$ and $y$ in the range $[1, k]$ that form an interval $(x, y]$. The answer to a range-sum query is the sum of elements in $A$ that are greater than $x$ and less than or equal to $y$.

   As an example if $k = 9$ and $A = [4, 5, 2, 6, 8, 3, 4, 4]$, the range-sum query for the interval $(3, 6]$ would return 23 . The interval $(7, 10]$ does not define a range-sum query because 10 is outside the range $[1, 9]$.

   Describe in detail how you would organize the data structure $C$ and construct $C$ so that each range-sum query can be answered in $O(1)$ time. Explain the running time of your algorithm for constructing $C$. Explain why a range-sum query can be answered in $O(1)$ time.

**Solution.**

(a) **Main Idea:** Construct a segment tree data structure such that each node stores the count of numbers within a specific range. The root represent the whole range of the sequence, with its left and right children represent two halves of its range. If a range doesn't contain any number, then the node ends.

In such tree structure, we can get the range query by starting at the root node of the segment tree and going through its nodes according to the rule:

- If the current node's range falls completely within the query range, return its count.
- If the current node's range does not overlap with the query range, return 0.
- Otherwise, recursively query the left and right child nodes and return the sum their results.

**Space Occupied:** $O(1 + 2 + 4 + \cdots + n) = O(2n - 1) = O(n)$

**Time Complexity:** $O(\log n)$ as it's a binary tree structure.

(b) **Main Idea:** Construct a array with length as large as the maximum number of the given array to store the sum of the numbers that are smaller than the index of the current position.

The detail are as follows:

- Suppose the largest element of the array is $k$, construct an array of $k$ numbers, calling it $A[k]$.
- $A[0] = 0$. For $i = 0$ to $k$, $A[i] = A[i-1] + i \times$ #Number of elements equal to $i$ in the array
- To get the range-sum in $(i, j]$, simply substract $A[j]$ by $A[i]$ to get the answer in $O(1)$ time.

**Space Occupied:** $O(k)$ where $k$ is the largest element in the array.

**Time Complexity:** $O(1)$

2. (20 points) Consider a switch with two inputs A and B and two outputs C and D. Each input is a single bit. Each output is also a single bit. The switch has two settings. In one setting, A is connected to C, and B is connected to D. That is, the outputs C and D are just the inputs A and B, respectively. In the other setting, A is connected to D, and B is connected to C. That is, the outputs C and D are the inputs B and A, respectively. Let $n$ be a positive power of 2. Describe how to assemble such switches into one device such that the device has $n$ inputs and $n$ outputs, and by setting the switches appropriately in the device, the ouptuts can be any of the $n!$ permutations of the inputs. Your device should not use more than $O(n \log n)$ switches. Explain the correctness of your device.

**Solution.**

**First Attempt:** Make use of the Batcher's odd even merge sort algorithm, construct a sorting network. We can construct a sorting network which consists of $\log_2 n$ stages, with wach stage having $n/2$ switches. (No further idea about such method)

**Reference:** Wiki: Batcher's Odd Even Merge Sort

**Second Attempt:** Make use of the binary tree construction. First, construct a binary tree with $n$ inputs being placed at $n$ nodes in a tree with height $\log n$. At each node, a switch is placed to switch the position of, and only the position of its two children. The tree is constructed via pre-order tree transversal. After all the switches are set to a specific state, the output is extracted accordingly also via pre-order tree transversal.

Total switches used:

$$\sum_{i=1}^{\log n} \frac{n}{2^i} = O(n \log n)$$

**Correctness:** We are proving that different types of combination of switches yeilds distinct solutions, so that using $O(n \log n)$ switches, we are able to perform $2^{n \log n} \sim n!$ distinct types of output.

By modifying a switch at a node, we're only modifying the two children which is directly connected to the current node, i.e, whenever the states of the switches are not identical, the two trees are not identical, resulting in the different extracted result via pre-order tree transversal. Thus, $O(n \log n)$ switches are enough to generate all the $n!$ distinct permutations of the array.

**Reference:** Tree Traversal Techniques –Data Structure and Algorithm Tutorials

3. (20 points) You are given $n$ numbers, where $n$ is a positive power of 2. Describe an algorithm that finds the largest and second largest numbers in $n + \log_2 n - 2$ comparisons. Explain the correctness of your algorithm. Show that your number of comparisons is indeed $n + \log_2 n - 2$.

**Solution.**

**Main Idea:** Use divide-and-conquer to find the largest element, which requires $n - 1$ comparisons. During such comparison, the second largest element must be eliminated by the largest element. As a result, we are able to examine through the $\log_2 n$ to get the second larget element.

• To get the largest element, we've conducted the comparison like a "single elimination tournament" with guaranteed $n - 1$ comparisons.

• Note that the second largest element must be eliminated by the largest element. Thus, we construct a list for every of the $n$ element to store all the elements eliminated by this number. Every time when we eliminate an element, we append the number eliminated by it (the "loser" of the comparison) to the list of the "winner".

• After we've got the largest element, we traverse through the list stored by the largest element to obtain the second largest element. The list contains $\log n$ elements, which requires $\log n - 1$ comparisons.

• Thus, we need $n + \log n - 2$ comparisons in total.

4. (20 points; from text book) Let RANDOM$(1, k)$ be a procedure that draws an integer uniformly at random from $[1, k]$ and returns it. We assume that a call of RANDOM takes $O(1)$ worst-case time. The following recursive algorithm RANDOM-SAMPLE generates a random subset of $[1, n]$ with $m \leq n$ distinct elements. Prove that RANDOM-SAMPLE returns a subset of $[1, n]$ of size $m$ drawn uniformly at random.

```
RANDOM-SAMPLE(m, n)
  if m = 0 then
    return ∅
  else
    S ← RANDOM-SAMPLE(m − 1, n − 1)
    i ← RANDOM(1, n)
    if i ∈ S then
      return S = S ∪ {n}
    else
      return S = S ∪ {i}
    end if
    return S
  end if
```

**Proof.**

The algorithm is identical as the procedure described as below:

• First, randomly choose a number in range $[1, n-m+1]$ to append to $S$.

• Next, randomly choose a number in range $[1, n-m+2]$ to append to $S$. If it's already in $S$, then append $n-m+2$ to $S$.

• Then, randomly choose a number in range $[1, n-m+3]$ to append to $S$. If it's already in $S$, then append $n-m+3$ to $S$.

• ...

• Finally, randomly choose a number in range $[1, n]$ to append to $S$. If it's already in $S$, then append $n$ to $S$. (A total of $m$ random pick is conducted)

We can check the consistency of the probability by calculating the probability of each number to be chosen.

Consider an arbitrary number $k$, evaluating the probability of $k$ NOT to be chosen:

• If $k \leq n-m+1$, then

$$\text{The number } k \text{ is NOT chosen} = \frac{n-m}{n-m+1} \cdot \frac{n-m+1}{n-m+2} \cdots \frac{n-1}{n} = \frac{n-m}{n}$$

• Suppose $n-m+1 < k \leq n$, then we've already chosen $k-n+m-1$ numbers $\implies$ At the random draw in range $[1, k]$, the probability of $k$ NOT to be chosen is $[k-(k-n+m-1+1)]/k = (n-m)/k$. (Condition: At the draw in range $[1, k]$, neither the elements that have been drawn nor $k$ itself are pulled out) In the remaining random pick, the calculation is the same as the above circumstance.

$$\text{The number } k \text{ is NOT chosen} = \frac{n-m}{k} \cdot \frac{k}{k+1} \cdots \frac{n-1}{n} = \frac{n-m}{n}$$

From the above discussion, we see that the probabilities of each elements to be drawn are identical, all equals to $1 - \frac{n-m}{n} = \frac{m}{n}$. This yields directly that the probability of each combination of RANDOM-SAMPLE to be drawn is also identical.

5. (20 points, from textbook) We explore a different analysis of the application of randomized quicksort to an array of size $n$.

   (a) (2 points) For $i \in [1, n]$, let $X_i$ be the indicator random variable for the event that the $i$th smallest number in the array is chosen as the pivot. That is, $X_i = 1$ if this event happens, and $X_i = 0$ otherwise. Derive $\mathrm{E}[X_i]$.

   (b) (2 points) Let $T(n)$ be a random variable that denotes the running time of randomized quicksort on an array of size $n$. Prove that

   $$\mathrm{E}\big[T(n)\big] = \mathrm{E}\left[\sum_{i=1}^{n} X_i \cdot (T(i-1) + T(n-i) + \Theta(n))\right].$$

   (c) (2 points) Prove that $E\big[T(n)\big] = \frac{2}{n} \cdot \sum_{i=2}^{n-1} \mathrm{E}\big[T(i)\big] + \Theta(n)$.

   (d) (7 points) Prove that $\sum_{k=1}^{n-1} k \log k \le \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$. (Hint: Consider $k = 2, 3, \ldots, (n/2) - 1$ and $k = n/2, \ldots, n-1$ separately.)

   (e) (7 points) Use (d) to show that the recurrence in (c) yields $\mathrm{E}\big[T(n)\big] = \Theta(n \log n)$. (Hint: Use substitution to show that $\mathrm{E}\big[T(n)\big] \le cn \log n$ for some positive constant $c$ when $n$ is sufficiently large.)

Note: The summation range should be from 2 to $n-1$ instead of 1 to $n-1$ for (d).

**Solution.**

(a) In the quicksort, the probabilities of each element to be chosen are identical, all equals to $1/n$, where $n$ indicates the number of elements.

$$\implies E[X_i] = \frac{1}{n}$$

(b) Note that according to the quicksort, the array to be sorted is splitted into two subarrays according to the element that is randomly chosen. The two subarrays are of length $i-1$ and $n-i$ if the $i$-th smallest element is chosen randomly. According to the recursive algorithm, the "Merge" step requires $\Theta(n)$ time. Thus,

$$E[T(n)] = E\left[\sum_{i=1}^{n} X_i \cdot (T(i-1) + T(n-i) + \Theta(n))\right]$$

(c) The deriviation is as follows:

$$E[T(n)] = E\left[\sum_{i=1}^{n} X_i \cdot (T(i-1) + T(n-i) + \Theta(n))\right] = E\left[\sum_{i=1}^{n} \frac{1}{n} \cdot (T(i-1) + T(n-i) + \Theta(n))\right]$$

$$= \Theta(n) + \frac{1}{n} \cdot E\left[\sum_{i=1}^{n} (T(i-1) + T(n-i))\right] = \Theta(n) + \frac{2}{n} \sum_{i=1}^{n} E[T(i-1)] = \Theta(n) + \frac{2}{n} \sum_{i=0}^{n-1} E[T(i)] = \Theta(n) + \frac{2}{n} \sum_{i=2}^{n-1} E[T(i)]$$

(d) According to the integral inequality, the left hand rule,

$$\sum_{k=2}^{n-1} k \log k \le \int_2^n k \log k = \left.\frac{\frac{k^2}{2} \log k - \frac{k^2}{4}}{\log 2}\right|_2^n = \frac{n^2}{2} \log n - \frac{n^2}{4 \log 2} - 1$$

$$\implies \sum_{k=2}^{n-1} k \ln k \le \frac{n^2}{2} \log n - \frac{n^2}{4 \log 2} - 1 \le \frac{n^2}{2} \log n - \frac{n^2}{8}$$

(e) Recurrence:

$$E[T(n)] = \Theta(n) + \frac{2}{n} \sum_{i=2}^{n-1} E[T(i)] \le \Theta(n) + \frac{2}{n} \sum_{i=2}^{n-1} [i \log i + \Theta(n)] \le \Theta(n) + \frac{2}{n} \sum_{i=2}^{n-1} i \log i$$

$$\le \frac{2}{n}\left[\frac{n^2 \log n}{2} - \frac{n^2}{8}\right] + \Theta(n) = n \log n - \frac{1}{4} n + \Theta(n) = n \log n + \Theta(n)$$

# Homework 3

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

1. (20 points) Consider the following (unrealistic) scenario. You are running a political campaign and want to collect petition signatures in the park. Your research team tells you that everyone who goes to the park visits for one of $n$ known time intervals during the day (if they come for a particular interval, they stay for the entire time interval; the different time intervals might overlap).

   If you go to the park at any time during one of the intervals you will get the signatures of everyone there for that interval. You want to figure out a minimum sized set of times that you have to go to the park to get everyone's signature.

   In computer science such a situation is called a *covering problem* and is modelled formally as described below.

   - A real *interval* is $I = [s, f]$ where $s \leq f$
   - A real *point* $x$ *covers* interval $I = [s, f]$ if $x \in I$, i.e., $s \leq x \leq f$.
   - Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of $n$ intervals and $A = \{a_1, a_2, \ldots a_k\}$ a set of points.
     Then "$A$ covers $\mathcal{I}$" if every interval in $\mathcal{I}$ is covered by at least one point in $A$. More formally, if, for every $I_j \in \mathcal{I}$ there exists $a_i \in A$ such that $a_i \in I_j$.
   - $|A|$, the size of $A$, is just the number of points in $A$.
     $A$ is a *minimal cover* of $\mathcal{I}$ if $A$ is a smallest cover of $\mathcal{I}$.
     That is, for every cover $A'$ of $\mathcal{I}$, $|A| \leq |A'|$.

   The input to this problem is the $2n$ numbers $s_1, f_1, s_2, f_2, \ldots, s_n, f_n$ with $s_j \leq f_j$, that define $\mathcal{I}$; $I_j = [s_j, f_j]$.

   The problem is to construct a minimal cover for $\mathcal{I}$ in $O(n \log n)$ worst-case time. Your output should be a set $A$ which is a minimal cover.

   (a) Prove that there exists a minimal cover $A$ such that every point in $A$ is one of the $f_j$. That is, $A \subseteq \{f_1, \ldots, f_n\}$.

   (b) Give documented pseudocode for your algorithm.
   In addition, below your algorithm explain in words/symbols what your algorithm is doing

   (c) Prove correctness of your algorithm. Your proof must be mathematically formal and understandable.
   *Note: Break your proof up into clear logical pieces and skip space between the pieces. Explicitly state what each part is assuming and proving.*
   *For examples of such proofs please see the lecture notes and tutorials.*

   *As seen in class, greedy algorithms tend to be simple. Their proofs of correctness are more complicated.*

   (d) Explain why your algorithm runs in $O(n \log n)$ worst case time.

**Solution.**
(a)

*Proof.* The proof goes as follows:
1. There must exists a minimal cover $A$ of $\mathscr{L}$.
2. If there exists a minimal cover $A$ of $\mathscr{L}$ such that at least one point in $A$ is not any of the $f_j$, then we are able to construct a new cover $A'$ of the same size such that every point in $A'$ is one of the $f_j$.

1. Easy to see that the set $A_1$ with size in $[1, n]$ consisting of all $s_i$s is a cover of $\mathscr{L}$. This set the upper bound of the size of the minimal set to be $n$. The size of any other possible set $A_2$ has a lower bound, which is 1. Thus, there must exists a minimal cover $A$ of $\mathscr{L}$.

2. If there exists a minimal cover $A$ of $\mathscr{L}$ such that at least one point in $A$ is not any of the $f_j$, we store such points in a set $B$. For every points $i$ in $B$, we trace the timeline to find the smallest $f_i$ such that $f_i \geq i$.[1] We then replace $i$ with $f_i$ in $A$, forming the new set $A'$ which is still a cover of $\mathscr{L}$, and the size of $A'$ is the same as $A$. Thus, we've constructed a new cover $A'$ of the same size such that every point in $A'$ is one of the $f_j$.

Note: [1] If there doesn't exist a $f_i \geq i$, that means $i$ is not covered by any interval, which contradicts the assumption that $A$ is a cover of $\mathscr{L}$. □

(b) The algorithm is as follows:

**Algorithm 36:** Find a Minimal Cover $A$

**Input:** $n$ sets of intervals $\mathscr{L} = \{I_1, ..., I_n\}$
**Output:** A minimal cover $A$ of $\mathscr{L}$
`FindMinimalCover(`$\mathscr{L}$`)`
Sort all the intervals in $\mathscr{L}$ in ascending order based on their ending time $f_j$
**while** $\mathscr{L} != null$ **do**
  $A$.append($\mathscr{L}[0].f$) // Append the interval with the smallest ending time to $A$
  $cur_f = \mathscr{L}[0].f$
  **foreach** $I \in \mathscr{L}$ **do**
    **if** $I.s \leq cur_f$ **then**
      $\mathscr{L}$.remove($I$) // Remove all the intervals that are already covered by the current point
    **end**
  **end**
**end**
**return** $A$

(c)

*Proof.* Suppose the minimal cover genrated by the former algorithm is $A = \{a_1, ..., a_m\}$, and the actual minimal cover is $A' = \{a'_1, ..., a'_{m'}\}$ with $m' < m$.

We first rank the interval set $\mathscr{L}$ in ascending order based on their ending time $f_j$, resulting in $\mathscr{L} = \{I_{k_1}, I_{k_2}, ..., I_{k_n}\}$.

1. If there's any element in $A'$ not in $A$, i.e., $\exists \alpha \in A'$ such that $\alpha \notin A$, find the minimal subscript $x$ such that $a_1 = a'_1, ..., a_{x-1} = a'_{x-1}$ but $a_x \neq a'_x$.

1.1 If $a'_x < a_x$, then modifying $a'_x$ to $a_x$ in $A'$ also forms a cover of $\mathscr{L}$ with the same total size. This is because we've already covered all the intervals that ends before $a_x$. If we apply the algorithm to a modified set that elimnates the covered intervals, then $a_x$ is the end of the first interval while $a'_x$ is not. Problem (a) has already proved that we are able to modify any non-$f_j$ point to one of the $f_j$.

1.2 If $a'_x > a_x$, then at least one interval is not covered by $A'$. We first eliminate all the intervals that ends before $a_x$. According to the algorithm, $a_x$ is the end of the first non-eliminated interval (i.e., not being covered by previous $a$). If $a'_x > a_x$, then the non-eliminated interval $I_{k_i}$ has the property that $s_{k_i} < f_{k_i} < a'_x$, resulting in $A'$ not covering $I_{k_i}$. Contradiction!

2. If $A' subsetneqq A$, the previous part already suggested that $A = A' \cup \{a_{m'+1}, a_{m'+2}, ..., a_m\}$. However, the previous algorithm already suggested that each of the $a_i$ is an end of some interval $I_{k_i}$ such that $I_{k_i}$ is not covered by the previous items, i.e., $a_{m'+1}$ is the end of ont of $I_{k_i}$, such that $I_{k_i}$ is not covered by $A'$. Thus, $A'$ is not a minimal cover of $\mathscr{L}$. Contradiction! □

(d) Sorting the intervals in ascending order based on their ending time $f_j$ requires $O(n \log n)$ time. The while loop requires $O(n)$ time. Thus, the total time complexity is $O(n \log n)$.

2. (20 points) The table below lists a 10 letters ($a$ to $j$) and their frequencies in a document. Apply Huffman coding algorithm taught in class to construct an optimal codebook. Your solution should contain three parts:

   (a) A full Huffman tree.

   (b) The final codebook that contains the binary codewords representing $a$ to $j$ (sorted in the alphabetical order on $a$ to $j$).

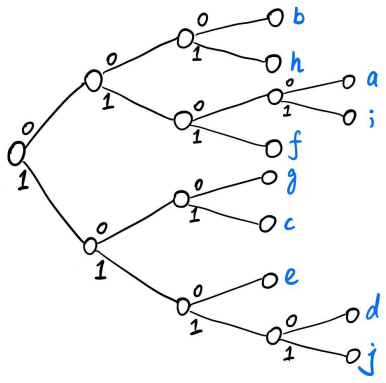
   (c) The codebook from part (b) but now sorted by the increasing lengths of the codewords.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $a_i$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ | $h$ | $i$ | $j$ |
| $f(a_i)$ | 21 | 35 | 28 | 16 | 28 | 50 | 27 | 44 | 26 | 17 |

**Rules:**

- In part (a) you should explicitly label each edge in the tree with a '0' or '1'. You should also label each leaf with its corresponding character $a_i$ and $f(a_i)$ value.
- Part (b) should be a table with 2 columns. The first column should be the letters $a$ to $j$. The second column should be the codeword associated with each character.
- In part (c) you should break ties using the alphabetical order of the characters. For example, if $a$, $d$ and $e$ all have codewords of length 5, then write the codeword for $a$ on top of the codeword for $d$ on top of the codeword for $e$.

**Solution.**
(a)




(b)

| Letter | a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| Code | 0100 | 000 | 101 | 1110 | 110 | 011 | 100 | 001 | 0101 | 1111 |

(c)

| Letter | b | c | e | f | g | h | a | d | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| Code | 000 | 101 | 110 | 011 | 100 | 011 | 0100 | 1110 | 0101 | 1111 |

3. (20 points) Run the stable marriage algorithm as described in class on the following set of people and jobs. Let the people make proposals. Each person has a priority list of jobs, and there is also a priority list of the people for each job. As in the lecture notes, the priority decreases from left to right in the two tables below.

| People | Jobs | | | |
|---|---|---|---|---|
| $p_1$ | $j_4$ | $j_1$ | $j_2$ | $j_3$ |
| $p_2$ | $j_2$ | $j_3$ | $j_1$ | $j_4$ |
| $p_3$ | $j_4$ | $j_2$ | $j_3$ | $j_1$ |
| $p_4$ | $j_2$ | $j_3$ | $j_4$ | $j_1$ |

| Jobs | People | | | |
|---|---|---|---|---|
| $j_1$ | $p_4$ | $p_1$ | $p_2$ | $p_3$ |
| $j_2$ | $p_4$ | $p_1$ | $p_3$ | $p_2$ |
| $j_3$ | $p_3$ | $p_1$ | $p_2$ | $p_4$ |
| $j_4$ | $p_1$ | $p_3$ | $p_4$ | $p_2$ |

Show the intermediate results of your run as in the lecture notes.

**Solution.**
1. $p_1$ sends offer to $j_4$: $p_1 - j_4$
2. $p_2$ sends offer to $j_2$: $p_1 - j_4$, $p_2 - j_2$
3. $p_3$ sends offer to $j_4$, $j_4$ rejects: $p_1 - j_4$, $p_2 - j_2$
4. $p_4$ sends offer to $j_2$, $j_2$ unpairs with $p_2$: $p_1 - j_4$, $p_4 - j_2$
5. $p_2$ sends offer to $j_3$: $p_1 - j_4$, $p_4 - j_2$, $p_2 - j_3$
6. $p_3$ sends offer to $j_2$, $j_2$ rejects: $p_1 - j_4$, $p_4 - j_2$, $p_2 - j_3$
7. $p_3$ sends offer to $j_3$, $j_3$ unpairs with $p_2$: $p_1 - j_4$, $p_4 - j_2$, $p_3 - j_3$
8. $p_2$ sends offer to $j_1$: $p_1 - j_4$, $p_4 - j_2$, $p_3 - j_3$, $p_2 - j_1$ Stable!

Final Matching: $p_1 - j_4$, $p_4 - j_2$, $p_3 - j_3$, $p_2 - j_1$

4. (20 points) Let $T$ be a set of $n$ tasks. Each task $t \in T$ is associated with a value $v(t)$, a length $\ell(t)$, a release time $r(t)$, and a deadline $d(t)$. The values, lengths, release times, and deadlines are positive integers for all tasks in $T$. We assume that $\ell(t) \leq d(t) - r(t)$ for every task $t \in T$, and that the release times and deadlines of the $n$ tasks in $T$ are distinct.

   A schedule of all $n$ tasks in $T$ is *legal* if no task $t$ starts before its release time $r(t)$ and the processor runs at most one task at any time. If a task $t$ finishes by its deadline $d(t)$, you collect its value $v(t)$. However, there is no guarantee that all $n$ tasks finish before their deadlines in a legal schedule; if a task $t$ does not finish by its deadline $d(t)$, you do not collect its value $v(t)$.

   Design a dynamic programming algorithm to determine the maximum value that you can collect by running a legal schedule of the $n$ tasks in $T$ on a single processor. Define and explain your notations. Define and explain your recurrence and boundary conditions. Write your algorithm in pseudocode. Derive the running time of your algorithm.

**Solution.**

**Main Idea:** First sort the tasks by its deadlines $d(t)$. Create an array $p(t)$ such that for each task $t$, $p(t)$ returns the largest index of the compatible tasks $t'$ with $t' < t$. Suppose the maximum possible total value for the first $i$ sorted tasks is $V(i)$, then

$$\begin{cases} V[i] = \max\{V[i-1], v_i + V[p(i)]\} \\ V[0] = 0 \end{cases}$$

**Algorithm 37:** Task Scheduling

```
TaskScheduling(T)
Sort the tasks by its deadlines d(t)
for i = 1 to n do
    p[i] ← largest index of the (sorted) compatible tasks i' with i' < i. // Do binary search for previous
      deadlines d(t_{i'}) and compare with the current releasing time r(t_i)
end
V[0] ← 0
for i = 1 to n do
    V[i] ← max{V[i − 1], v_i + V[p(i)]}
end
return V[n]
```

**Time complexity:**

- Sorting the tasks -> $O(n \log n)$
- Comstruct $p[i]$ -> each value requires $O(\log n)$ time using binary search -> $n \times O(\log n) = O(n \log n)$
- Calculating $V[i]$ -> $O(n)$

Altogether, the algorithm requires a time complexity of $O(n \log n)$

5. (20 points) Consider a two-dimensional array $A[1..n, 1..n]$ of distinct integers. We want to find the *longest increasing path* in $A$. A sequence of entries $A[i_1, j_1], A[i_2, j_2], \ldots, A[i_k, j_k], A[i_{k+1}, j_{k+1}], \ldots$ is a path in $A$ if and only if every two consecutive entries share a common index and the other indices differ by 1, that is, for all $k$,

   - either $i_k = i_{k+1}$ and $j_{k+1} \in \{j_k - 1, j_k + 1\}$, or
   - $j_k = j_{k+1}$ and $i_{k+1} \in \{i_k - 1, i_k + 1\}$.

   A path in $A$ is increasing $A[i_1, j_1], A[i_2, j_2], \ldots, A[i_k, j_k], A[i_{k+1}, j_{k+1}], \ldots$ if and only if $A[i_k, j_k] < A[i_{i+1}, j_{k+1}]$. The length of a path is the number of entries in it.

   Design a dynamic programming algorithm to find the longest increasing path in $A$. Your algorithm needs to output the maximum length as well as the indices of the array entries in the path. Note that there is no restriction on where the longest increasing path may start or end. Define and explain your notations. Define and explain your recurrence and boundary conditions. Write your algorithm in pseudo-code. Derive the running time of your algorithm.

**Solution.**
First Attempt: Simply do the DFS for each grid in four directions

**Algorithm 38:** DFS for Grid Paths

**Input:** $A[1 \cdots n][1 \cdots n]$
**Output:** length, indices
`DFS`(*i, j,prev,path[]*)
**if** $i < 0$ *or* $j < 0$ *or* $i >= n$ *or* $j >= n$ *or* $A[i][j] <$*prev* **then**
  **return** *0,[]*
**end**
left, leftPath[] = `DFS` $(i, j-1, A[i][j]$,leftPath)
leftPath.append($A[i][j]$)
right, rightPath[] = `DFS` $(i, j+1, A[i][j]$,rightPath)
rightPath.append($A[i][j]$)
top, topPath[] = `DFS` $(i-1, j, A[i][j]$,topPath)
topPath.append($A[i][j]$)
bottom, bottomPath[] = `DFS` $(i+1, j, A[i][j]$,bottomPath)
bottomPath.append($A[i][j]$)
**return** max *left,right,top,bottom* $+ 1$,*Corresponding_Path[]*
First Call:
Max = 0, MaxPath[] = null
**foreach** *Grid* **do**
  curMax, curMaxPath[] = `DFS`(*Grid.x,Grid.y,-1,[]*) **if** *curMax* $>$ *Max* **then**
    Max = curMax
    MaxPath = curMaxPath
  **end**
**end**
**return** *curMax, curMaxPath*

However, this algorithm has a time complexity of $O(n^2 \cdot 4^{n^2})$, too large!

Second Attempt: The previous attempt may recall the DFS algorithm on the same grid for multiple times. All we need to do is creating an array to store the calculated results to avoid recalculation.

**Algorithm 39:** DFS for Grid Paths

**Input:** $A[1 \cdots n][1 \cdots n]$
**Output:** length, indices
Memory = [][][] // For each grid, reserve an array to store the longest increasing result starting at such grid
`DFS(`$i, j$*,prev,path[]*`)`
**if** *Memory[i][j] != null* **then**
  **return** *len(Memory[i][j]),Memory[i][j]*
**end**
**if** $i < 0$ *or* $j < 0$ *or* $i >= n$ *or* $j >= n$ *or* $A[i][j] <$*prev* **then**
  **return** *0,[]*
**end**
left, leftPath[] = `DFS` $(i, j-1, A[i][j]$,leftPath)
leftPath.append($A[i][j]$)
right, rightPath[] = `DFS` $(i, j+1, A[i][j]$,rightPath)
rightPath.append($A[i][j]$)
top, topPath[] = `DFS` $(i-1, j, A[i][j]$,topPath)
topPath.append($A[i][j]$)
bottom, bottomPath[] = `DFS` $(i+1, j, A[i][j]$,bottomPath)
bottomPath.append($A[i][j]$)
Get the longest path $P$[] among the four candidates
$Memory[i][j] \leftarrow P$
**return** max *left,right,top,bottom* $+ 1$*,Corresponding_Path[]*
First Call:
Max = 0, MaxPath[] = null
**foreach** *Grid* **do**
  curMax, curMaxPath[] = `DFS(`*Grid.x,Grid.y,-1,[]*`)` **if** *curMax* $>$ *Max* **then**
    Max = curMax
    MaxPath = curMaxPath
  **end**
**end**
**return** *curMax, curMaxPath*

**Time Complexity:** DFS for each grid is called at most once, as later calls would directly return the value stored in $Memory[][][]$. All the other functions calls requires a constant time. Thus, the time complexity is $O(n^2)$.

# Homework 4

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

1. (20 points)

   (a) (2 points) Prove that if $G$ is an undirected graph with $n$ vertices and $n$ edges with no vertices of degree 0 or 1, then the degree of every vertex is 2.

   (b) (4 points) Let $G$ be an undirected graph with at least two vertices. Prove that it is impossible for every vertex of $G$ to have a different degree.

   (c) (14 points) In a group of 10 people, each one has 7 friends among the other nine people. Prove that there exist 4 people who are friends of each other.

**Solution.**

(a) Suppose at least one vertex $v$ has degree $\geq 3$, while other vertices have degree $\geq 2$. Then, the number of edges $l$ satisfies:

$$l = \frac{1}{2}\sum_{v\in V} d(v) \geq \frac{1}{2} \times 3 + \frac{1}{2} \times (n-1) \times 2 = n + \frac{1}{2} > n$$

Contradicts with the assumption that the graph has $n$ edges. Thus, every vertex has a degree of 2.

(b) Suppose the undirected graph $G$ with $n$ vertices has its every vertex of different degree. Notice that each vertex should have a degree between 0 and $n-1$ (a total of $n$ different values), there must be a vertex with degree 0 and a vertex with degree $n-1$. However, the vertex with degree $n-1$ must be connected to all the other vertices, which contradicts with the assumption that there exists a vertex with degree 0. Thus, there must be at least two vertices with the same degree.

(c) We begin the proof with a lemma:

**Lemma.** *For any graph $G$ with $n$ vertices, if every vertex has a degree of two, then $G$ must either be a cycle, or a union of several cycles, each with at least three vertices.*

*Proof.* We start from an arbitrary vertex $v_1$ in the graph, tracing its two edges to the other two vertices $v_{11} \neq v_{12}$ called "ends", forming a tree. We continue to trace the two branch $v_{11}$ & $v_{12}$. Each new branch should either introduce a new vertex, forming a new "end" or connect to each other, forming a complete cycle. (That is, unable to connect to any vertex that already exists in the tree.) Since the number of vertices is finite, the process must terminate, forming a cycle. Thus, every vertex should be in a cycle of length $\geq 3$. In this case, the graph must either be a cycle, or a union of several cycles, each with at least three vertices. □

We then come to the main proof:

*Proof.* Restate the problem in an equivalent way as follows (Invert the connection and disconnection):

In a group of 10 vertices, each one of them has a degree of 2. Prove that there exists 4 vertices such that non of them connects to the other three.

According to **Lemma**, the graph must either be a cycle, or a union of several cycles, each with at least three vertices. In the case of 10 vertices, there are at most 3 cycles. Note to the fact that if two vertices are not in the same cycle, they are unconnected.

• If the graph is a single cycle. We start by naming any one of the vertex to be $v_1$, transversing the cycle to name the vertices as $v_2, v_3, ..., v_{10}$. Then, we are able to find 4 vertices such that non of them connects to the other three, i.e., $v_1, v_3, v_5, v_7$.

• If the graph is a union of two cycles, then there must be either at least one cycle of length $\geq 6$ or both cycles of length 5. In the former case, we take three vertices from the cycle of length $\geq 6$ (i.e., $v_1, v_3, v_5$) and one vertex from the other cycle. In the latter case, we take two unconnected vertices from each of the two cycles. Both cases yields the result.

• If the graph is a union of three cycles, then there must be two cycles of length 3 and one of length 4. In this case, we take the two unconnected vertices from the cycle of length 4, and one vertex from each of the cycle of length 3, which yields the result.

□

2. (20 points) Write the pseudocode of a variant of DFS that checks whether an undirected graph is bipartite or not. Your pseudocode must run in $O(n + m)$ time, where $n$ is the number of vertices and $m$ is the number of edges. Note that the input graph may not be connected.

**Solution.**
**Main Idea:**

**Algorithm 40:** DFS Check Bipartite Graph

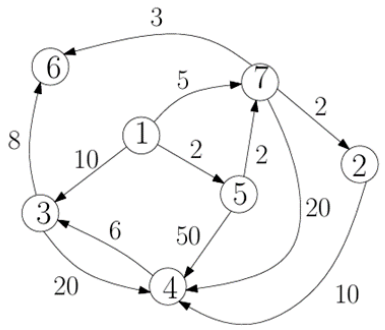
`CheckBipartiteGraph`$(G, n)$

3. (20 points) Let $G = (V, E)$ be an undirected connected graph with $n$ vertices and $m$ edges. Each edge in $G$ is also given an non-negative integer weight. Given a path $P$ in $G$ from a vertex $u$ to a vertex $v$, the *bottleneck weight* of $P$, denoted by $wt(P)$, is the minimum edge weight in $P$. A *maximum bottleneck path* between $u$ and $v$ is the path $Q$ between $u$ and $v$ such that $wt(Q) \geq wt(P)$ for all paths between $u$ and $v$. Our problem is to report the maximum bottleneck paths between all pairs of vertices in $G$. Show that this problem can be solved by finding the minimum spanning tree of some graph. Explain the running time of your algorithm.

**Solution.**

4. (20 points)

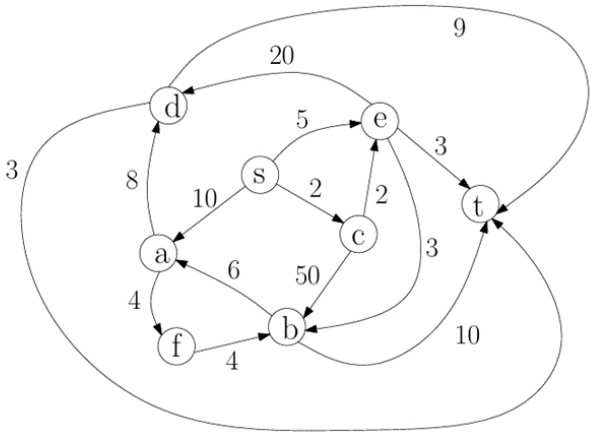
   (a) (10 points) Run Dijkstra's single-source shortest path algorithm on the following directed graph $G$. Use vertex 1 as the source. Show the graph $G$ and the values of $u.d$ and $u.p$ for every vertex $u$ after removing and processing each vertex from the min-heap $Q$ as in the lecture notes. Use the same convention and notation as in the lecture notes to show your intermediate steps.




   (b) (10 points) Run Floyd-Warshall algorithm on the above graph. For each $k = 0, 1, 2, 4, 5, 6, 7$ in this order, show $d_{ij}^{(k)}$ as a $7 \times 7$ matrix.

**Solution.**

5. (20 points) Run Ford-Fulkerson's maximum flow algorithm on the following directed graph $G$. Use $s$ as the source and $t$ as the sink. Use the same convention and notation as in the lecture notes to show:
   - The residual graph $G_f$ and the augmenting path selected in $G_f$.
   - The flow values on the edges of $G$ after using the augmenting path selected to update the flow.




**Solution.**