# 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

October 6, 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 ← 1 to n − 1 do
    for j ← i + 1 to n do
        if A[i] > A[j] then
            swap A[i] and A[j]
        end
    end
end
```

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

## 2.2 Insertion Sort

**Algorithm 2:** Insertion Sort

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

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

## 2.3 Wild-Guess Sort

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

## 2.4 Worst-Case Analysis

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

**Algorithm 3:** Wild-Guess Sort

**Input:** An array $A[1...n]$ of elements
**Output:** Array $A[1...n]$ of elements in sorted order (asending)
$\pi \leftarrow [4, 7, 1, 3, 8, 11, 5, ...]$ Create random permutation Check if $A[\pi[i]] \leq A[\pi[i+1]]$ for all $i = 1, 2, ..., n-1$ If yes, output A according to $\pi$ 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 ← [(p + r)/2]
if x = A[q] then
  | return q
end
if x < A[q] then
  | BinarySearch(A[], p, q − 1, x)
end
else
  | BinarySearch(A[], q + 1, r, x)
end
```

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

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

**Example: Binary Search in Rotated Array**

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

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

**Algorithm:**

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

$Find - x(A, x)$

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

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

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

**Example: Finding the last 0**

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

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

**Algorithm:**

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

## 3.2 Merge Sort

**Principle of the Merge Sort:**

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

**Algorithm 5:** Merge Sort

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

**Algorithm 6:** Merge

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

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

**Merge Sort Recurrence:**

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

**Simplification:**

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

**Result:**

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

## 3.3 Inversion Counting

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

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

### 3.3.1 Counting Inversions: Divide-and-Conquer

**Principle of the Algorithm:**

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

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

**Algorithm 7:** Inversion Count during Combination

```
Input: Two Arrays L ← A[p...q] and R ← A[q+1...r] of elements in sorted order
Count(A, p, q, r)
i ← 1, j ← 1, c ← 0
while (i ≤ q − p + 1)&&(j ≤ r − q) do
    if L[i] ≤ R[j] then
        i ← i + 1
    end
    else
        I[j] = q − p − i + 2
        c ← c + I[j]
        j ← 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 ← ⌊(p + r)/2⌋
c1 ← Sort-and-Count(A, p, q)
c2 ← Sort-and-Count(A, q + 1, r)
c3 ← Merge-and-Count(A, p, q, r)
return c1 + c2 + c3
First Call: Sort-and-Count(A, 1, n)
```

**Algorithm 9:** Merge-and-Count

**Input:** Two Arrays $L \leftarrow A[p...q]$ and $R \leftarrow A[q+1...r]$ of elements in sorted order

```
Merge-and-Count(A, p, q, r)
Append ∞ at the end of L and R
i ← 1, j ← 1, c ← 0
for k ← p to r do
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i + 1
    end
    else
        A[k] ← R[j]
        j ← j + 1
        c ← c + q − p − i + 2
    end
end
return c
```

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

**Observations of D&C in Logarithmic Patterns:**

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

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

**Observations of D&C in Non-Logarithmic Patterns:**

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

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

## 3.5 Maximum Contiguous Subarray

**Example: The Maximum Subarray Problem**

**Input:** An array of numbers $A[1, ..., n]$, both positive and negative

**Output:** Find the maximum $V(i, j)$, where $V(i, j) = \sum_{k=i}^{j} A[k]$

**Brute-Force Algorithm**

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

**A Data-Reuse Algorithm**

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

### 3.5.1 A D&C Algorithm

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

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

---

**Algorithm 10:** Maximum Subarray

---

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

---

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

### 3.5.2 Kadane's Algorithm

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

•$V[1] = A[1]$

•$V[i] = \max(A[i], A[i] + V[i-1])$

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

---

**Algorithm 11:** Kadane's Algorithm

---

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

---

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

**Example: Maximizing Stock Profits**

You are presented with an array $p[1 \ldots n]$ 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]) // a1b1
V ← Multiply (A[mid + 1..n], B[1..mid]) // a1b0
W ← Multiply (A[1..mid], B[mid + 1..n]) // a0b1
Z ← Multiply (A[1..mid], B[1..mid]) // a0b0
M[1..2n] ← 0
M[1..n] ← Z // a0b0
M[mid + 1..] ← M[mid + 1..] ⊕ V ⊕ W // +[(a1b0 + a0b1) ≪ (left shift) n/2]
M[2mid + 1..] ← M[2mid + 1..] ⊕ U // +[a1b1 ≪ n]
return M
```

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

### 3.6.2 Karatsuba Multiplication

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

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

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

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

## 3.7 Matrix Multiplication

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

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

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

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

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

### 3.7.2 Strassen's Matrix Multiplication Algorithm

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

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

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

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

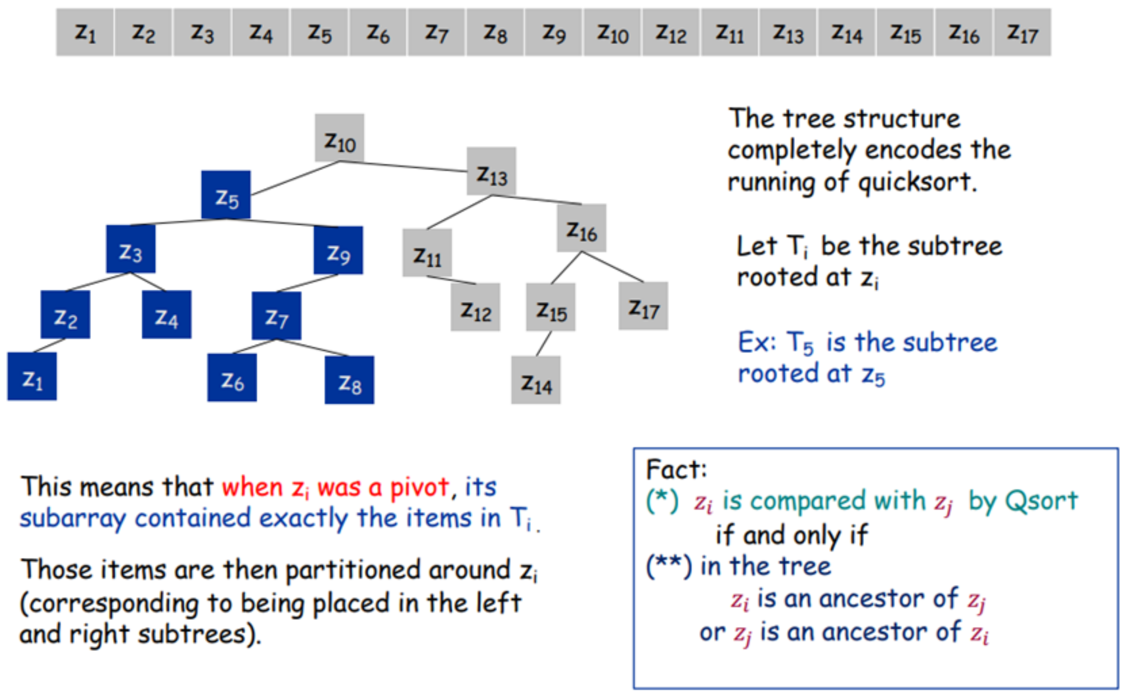
```
Quicksort(A, p, r) // Array from A[p] to A[r]
if p ≥ r then
 | return
end
q =Partition (A, p, r) // Set a new pivot position
Quicksort (A, p, q − 1)
Quicksort (A, q + 1, r)
First Call: MaxSubArray(A, 1, n)

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 $\frac{2}{j-1+1}$

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

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

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

**Idea:**
1. Choose a Pivot $x$ from $A[p \ldots r]$
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 $A[p \ldots r]$ is the $i$-th element of $A[p \ldots q-1]$, solve recursively
   If $i > q - p + 1$, then the $i$-th element of $A[p \ldots r]$ is the $j = (i - q + p - 1)$-th element of $A[q+1 \ldots r]$, 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 ... r] as the pivot and swap it with A[r]
q ←Partition(A, p, r)
k ←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

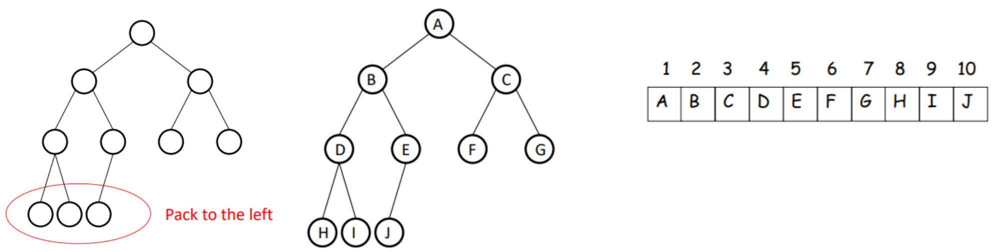
**Main Idea:** Processing the shortest job first - Extracting the smallest element from the queue.
A Priority Queue is an abstract data structure that supports two operations: Insert & Extract-Min.
**Implementations:**
1. Unsorted list + a pointer to the smallest element: $O(1)$ Insert & $O(n)$ Extract-Min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

---

`Insert`$(x, i)$
$A[i] \leftarrow x$
$j = i$
**while** $j > 1$ *and* $A[j] < A[\lfloor j/2 \rfloor]$ **do** // $A[j]$ is less than its parent
  | Swap $A[j]$ and $A[\lfloor j/2 \rfloor]$
**end**
$j = \lfloor j/2 \rfloor$

---

**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 $A[1 \ldots i]$

```
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.

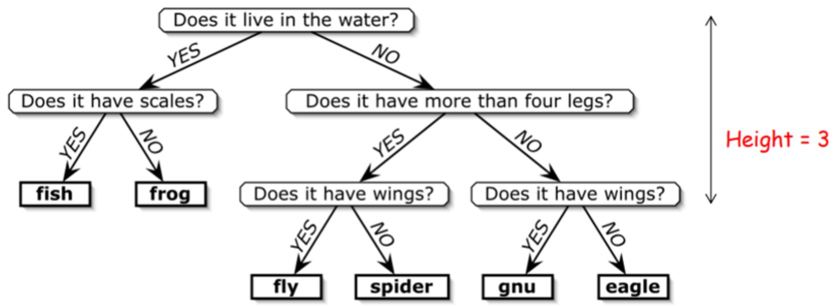
For more information, see: Binary Heap (Wikipedia)

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 |