ECE358 Cheatsheet

Hanhee Lee

September 1, 2024

Contents

List of Figures

List of Tables

Read the textbook for this course.

1 Asymptotics (Ch. 3.1-2 pg. 50-63, L2)

Asymptotic efficiency focuses on understanding how the running time of an algorithm grows with the input size, particularly for large inputs.

1.1 Big-O

Definition: $O(g(n)) = \{f(n)$: $\exists$ positive constants c and n_0 such that $0 \leq f(n) \leq cg(n) \; \forall \; n \geq n_0\}$

- **Asymptotic upper bound:** Grows **no faster** than a certain rate, based on the highest-order term.

Warning: Every function $f(n)$ in the set $O(g(n))$ must be **asymptotically nonnegative** (i.e. $f(n)$ must be positive whenever n is sufficiently large).

1.2 Big-Omega

Definition: $\Omega(g(n)) = \{f(n)$: $\exists$ positive constants c and n_0 such that $0 \leq cg(n) \leq f(n) \; \forall \; n \geq n_0\}$

- **Asymptotic lower bound:** Grows **at least as fast** as a certain rate, based on the highest-order term.

1.3 Big-Theta

Definition: $\Theta(g(n)) = \{f(n)$: $\exists$ positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \; \forall \; n \geq n_0\}$

- **Asymptotically tight bounds:** Grows **precisely** at a certain rate, based on the highest-order term.
- **Constant factor:** Characterizes the rate of growth of the function to within a constant factor from above and below. These two constant factors need not be equal.

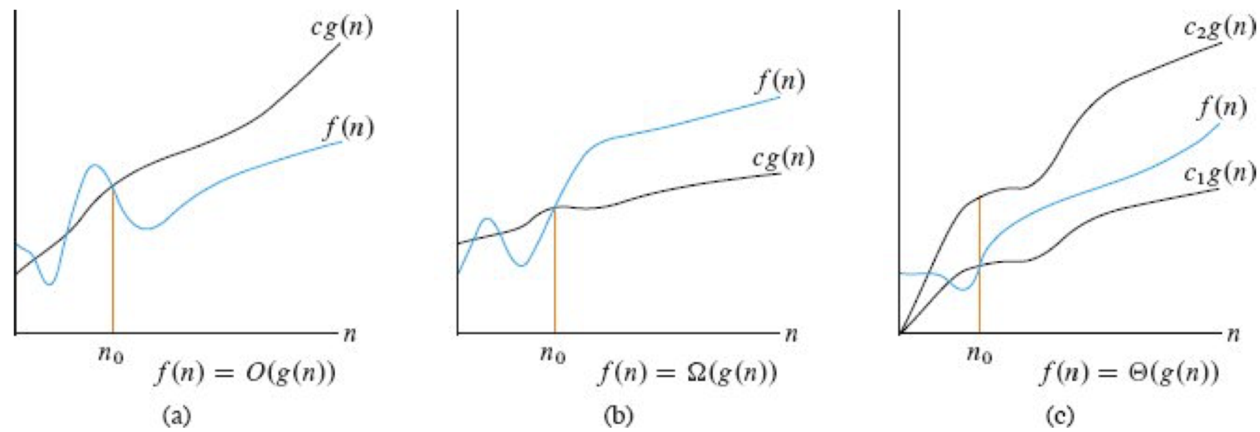


Figure 1: Graphical examples of the Big-O, Big-Omega, and Big-Theta.

Example: Find the Big-O, Big-Omega, and Big-Theta of the following function: $7n^3 + 100n^2 - 20n + 6$.

1. **Highest-order term:** $7n^3$
2. **Remove constants:** n^3
3. **Big-O notation:** $\mathrm{O}(n^3)$
   - In general, $\mathrm{O}(n^c)$ for any constant $c \geq 3$ because the function grows no faster than this.
4. **Big-Omega notation:** $\Omega(n^3)$
   - In general, $\Omega(n^c)$ for any constant $c \leq 3$ because the function grows at least as fast than this.
5. **Big-Theta:** Since O and Ω are the same, therefore, $\Theta(n^3)$ (by theorem)

Intuition/Tips: In all asymptotic notations, you are trying to describe a function after n_0 (i.e. ignore all flunctuation before).

1.4 Theorem 3.1

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

1.5 Asymptotic notation and running times

Warning: Make sure that the asymptotic notation you use is as precise as possible without overstating which running time (i.e. worst-case, best-case, or any other-case) it applies to.

Example: What asymptotic notation should you use for insertion sort's worst-case, best-case, and general running time?

- **Worst-case:** $O(n^2)$, $\Omega(n^2)$, and $\Theta(n^2)$ can be used, but $\Theta(n^2)$ is the most precise.
- **Best-case:** $O(n)$, $\Omega(n)$, and $\Theta(n)$ can be used, but $\Theta(n)$ is the most precise.
- **General:** $O(n^2)$ because in all cases, its running time grows no faster than n^2. Or $\Omega(n)$ because in all cases, its running time is at least as fast as n.

1.6 Abuses of asymptotic notation

Intuition/Tips:

- **Equality:**
  - When asymptotic notation stands alone on the RS of an equation (or inequality), then $=$ means $\in$.

  - When asymptotic notation is in a formula, it is an anonymous function (AF) that we do not care to name.
  - When asymptotic notation appears on the LS of an equation: No matter how the AF is chosen on the LS, there is a way to choose the AF on the RS to make the equation valid.
- **Variable tending toward ∞ must be inferred from context:**
  - e.g. $O(g(n))$, then we are interested in the growth of $g(n)$ as n grows.
  - e.g. $f(n) = O(1)$, then $f(n)$ is bounded from above by a constant as n goes to ∞.
  - e.g. $T(n) = O(1)$ for $n < 3$ is that there exists a positive constant c such that $T(n) \leq c$ for $n < 3$.

1.7 Comparing function properties

Definition:
Transitivity:
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

Reflexivity:
- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Symmetry:
- $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.

Transpose symmetry:
- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

Different functions:
- $n^a \in O(n^b)$, iff $a \leq b$.
- $\log_a(n) \in O(\log_b(n))$, for all a, b.
- $c^n \in O(d^n)$, iff $c \leq d$.
- If $f(n) \in O(f'(n))$ and $g(n) \in O(g'(n))$, then:
  - $f(n) \cdot g(n) \in O(f'(n) \cdot g'(n))$.
  - $f(n) + g(n) \in O(\max\{f'(n), g'(n)\})$.

Intuition/Tips:
- $f(n) = O(g(n))$ is like $a \leq b$
- $f(n) = \Omega(g(n))$ is like $a \geq b$
- $f(n) = \Theta(g(n))$ is like $a = b$

1.8 Polynomially-bounded

Definition: $f(n)$ is polynomially-bounded if $f(n) = O\left(n^k\right)$ for some real value of k.

Theorem: $f(n) = O\left(n^k\right)$ iff $lg(f(n)) = O(lg(n))$

1.9 Limit method

Definition: Find the asymptotic relationship between two functions for which you might not have any intuition about.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = \mathcal{O}(g(n)) \tag{1}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ 0 < c < \infty \implies f(n) = \Theta(g(n)) \tag{2}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \Omega(g(n)) \tag{3}$$

1.9.1 L'Hôpital's rule

Definition: If $\lim_{x\to c} f(x) = \lim_{x\to c} g(x) = 0$ or $\pm\infty$, then:

$$\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)} \tag{4}$$

1.9.2 Logs of limits and limits of logs

Definition:

$$\lg\left(\lim_{x\to c} g(x)\right) = \lim_{x\to c} \lg(g(x)) \tag{5}$$

2 Logarithms, Summations (L7)

2.1 Logarithms (Ch. 3.3 pg. 66-7)

2.1.1 Definition and notation

Definition:

$$a = b^c \iff \log_b a = c \tag{6}$$

Notation:

- $lg\ n = log_2 n$
- $ln\ n = log_e n$
- $lg^k\ n = (lg\ n)^k$
- $lg^{(2)} n = lg\ lg\ n = lg(lg\ n)$

2.1.2 Properties

Definition: $\forall$ real $a > 0,\ b > 0,\ c > 0,$ and $n,$ we have

1. $a = b^{\log_b a}$
2. $\log_c(ab) = \log_c a + \log_c b$
3. $\log_b a^n = n \log_b a$
4. $\log_b a = \dfrac{\log_c a}{\log_c b}$
5. $\log_b \left(\dfrac{1}{a}\right) = -\log_b a$
6. $\log_b a = \dfrac{1}{\log_a b}$
7. $a^{\log_b c} = c^{\log_b a}$
8. $\log_b \dfrac{a}{c} = \log_b a - \log_b c$

2.2 Functional iteration

Definition: $f(n)$ iteratively applied i times to an initial value of n.

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0, \\ f\left(f^{(i-1)}(n)\right) & \text{if } i > 0. \end{cases} \tag{7}$$

2.3 Iterated logarithm function

Definition: The minimum number of times i that the logarithm function must be applied to n for the result to be less than or equal to 1:

$$\lg^* n = \min\left\{i \geq 0 \ : \ \lg^{(i)} n \leq 1\right\} \tag{8}$$

Intuition/Tips:
- **Definition of** $\lg^{(i)} n$**:** The expression $\lg^{(i)} n$ denotes the logarithm function applied i times in succession.
  - If $i = 1$, then $\lg^{(1)} n = \lg n$. If $i = 2$, then $\lg^{(2)} n = \lg(\lg n)$, and so on.
  - This is different from $\lg^i n$, which would mean $(\lg n)^i$, i.e., raising $\lg n$ to the power i.
- **Conditions for Definition:** The iterated logarithm $\lg^{(i)} n$ is only defined if $\lg^{(i-1)} n > 0$. This constraint exists because the logarithm of a non-positive number is undefined in real numbers.
- **Useful formula:** $O(nlg^* n) \approx O(n)$

Example: The iterated logarithm is a *very* slowly growing function:
- $\lg^* 2 = 1$ because one application of the logarithm to 2 results in a value less than or equal to 1.
- $\lg^* 4 = 2$
- $\lg^* 16 = 3$ because three applications of the logarithm to reach a value less than or equal to 1.
- $\lg^* 65536 = 4$
- $\lg^*(2^{65536}) = 5$

2.4 Fibonacci Numbers (Ch. 3.3)

2.4.1 Definition

Definition:

$$F_i = \begin{cases} 0 & \text{if } i = 0, \\ 1 & \text{if } i = 1, \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2. \end{cases} \tag{9}$$

2.4.2 Golden ratio and its conjugate

Definition:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803\ldots \tag{10}$$

and its conjugate, by

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.61803\ldots \tag{11}$$

Specifically, we have

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} \tag{12}$$

2.5 Summations (Ap. A.1 pg. 1140-51)

2.5.1 Arithmetic series

Definition:

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2) \tag{13}$$

2.5.2 General arithmetic series

Definition: For $a \geq 0$ and $b > 0$,

$$\sum_{k=1}^{n} (a + bk) = \Theta(n^2) \tag{14}$$

2.5.3 Sums of squares and cubes

Definition:
Sums of squares:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{15}$$

Sums of cubes:

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} \tag{16}$$

2.5.4 Finite geometric series

Definition: For $x \neq 1$,

$$\sum_{k=1}^{n} x^k = 1 + x + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \tag{17}$$

2.5.5 Infinite decreasing geometric series

Definition: For $|x| < 1$,

$$\sum_{k=1}^{\infty} x^k = \frac{1}{1 - x} \tag{18}$$

2.5.6 Harmonic series

Definition: For positive integers n, the nth harmonic number is

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1) \tag{19}$$

2.5.7 Telescoping series

Definition: For any sequence $a_0, a_1, \ \ldots \ , a_n$,

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0 \quad \text{OR} \quad \sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n \tag{20}$$

- Each of the terms is added in exactly once and subtracted out exactly once.

2.5.8 Reindexing summations

Intuition/Tips:

$$\sum_{k=0}^{n} a_{n-k} = \sum_{j=0}^{n} a_j \tag{21}$$

- $j = n - k$
- If the summation index appears in the body of the sum with a minus sign, it's worth thinking about reindexing.

2.5.9 Products

Definition: The finite product $a_1 a_2 \cdots a_n$ can be expressed as:

$$\prod_{k=1}^{n} a_k \tag{22}$$

2.5.10 Product to summation

Definition:

$$lg\left(\prod_{k=1}^{n} a_k\right) = lg\left(a_1 \cdot a_2 \cdots a_n\right) = lg(a_1) + lg(a_2) + \ldots + lg(a_n) = \sum_{k=1}^{n} lg(a_k) \tag{23}$$

3 Induction, Contradiction (L3)

3.1 Induction (Ap. A.2)

Motivation: The most basic way to evaluate a series is to use induction.

Process: Given proposition $P(n)$

1. Basis: Prove the base case $P(1)$
2. Inductive hypothesis: Assume true for $P(n)$
3. Inductive step: Use the hypothesis to show its true for $P(n) \rightarrow P(n+1)$

Therefore, $\forall\ nP(n)$.

Intuition/Tips: You don't always need to guess the exact value of a summation in order to use induction. Instead, use induction to prove an upper or lower bound on a summation.

Example: Prove $\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$

1. **Basis:** $n = 1$, $\frac{1(1+1)}{2} = 1$
2. **Inductive hypothesis:** Assume true for n, $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$
3. **Inductive step:** Prove for $n+1$: $1 + 2 + \ldots + n + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$

Therefore, we proved by induction that the formula, $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ for $n+1$ is true for $n+1$.

Example: Prove the asymptotic upper bound $\sum_{k=0}^{n} 3^k = O(3^n)$ or $\sum_{k=0}^{n} 3^k \leq c3^n$ for some constant c.

1. **Basis:** $n = 0$: $\sum_{k=0}^{0} 3^k = 1 \leq c$ as long as $c \geq 1$
2. **Inductive hypothesis:** Assume that the bound holds for n.
3. **Inductive step:** Prove for $n+1$:

$$\begin{aligned}\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^{n} 3^k + 3^{n+1} \\ &\leq c3^n + 3^{n+1} \text{ by the inductive hypothesis} \\ &= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \text{ by factoring out } c3^{n+1} \\ &\leq c3^{n+1} \text{ since we are using the inequality it still holds true}\end{aligned}$$

Therefore, as long as $\left(\frac{1}{3} + \frac{1}{c}\right) \leq 1$ or $c \geq \frac{3}{2}$. Thus, $\sum_{k=0}^{n} 3^k = O(3^n)$.

Warning: Consider the following fallacious proof that $\sum_{k=1}^{n} k = O(n)$.

1. **Basis:** $\sum_{k=1}^{1} k = O(1)$
2. **Inductive hypothesis:** Assume that the bound holds for n.
3. **Inductive step:** Prove for $n+1$:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) \\ &= O(n) + (n+1) \\ &= O(n+1) \quad \text{(wrong!)}\end{aligned}$$

The bug in the argument is that the "constant" hidden by the "big-O" grows with n and thus is not constant. We have not shown that the same constant works for all n.

3.2 Contradiction

Process: Property $P(n)$ which you want to prove true, and it can be true or false.

1. If want to prove true, assume $\neg P(n)$.
2. Work towards a contradiction by working with the expression $\neg P(n)$ and prove this to be false.
3. If this resulted in a false statement then $P(n)$ is true.

Example: Prove that if $x^2 - 5x + 4 < 0,$ then $x > 0$

1. **ATaC:** Assume towards a contradiction (ATaC) that $x^2 - 5x + 4 < 0$ but $x \leq 0$.

2. Analyze the quadratic expression:

$$x^2 - 5x + 4 = (x-1)(x-4)$$

Thus, the inequality becomes:

$$(x-1)(x-4) < 0$$

3. This inequality implies that x must lie between the roots 1 and 4, i.e., $1 < x < 4$.
4. **Contradiction:** However, the assumption $x \leq 0$ contradicts this because there are no values of $x \leq 0$ that satisfy $1 < x < 4$.

Therefore, the contradiction shows that the assumption $x \leq 0$ cannot be true if $x^2 - 5x + 4 < 0$. Hence, if $x^2 - 5x + 4 < 0$, it must be that $x > 0$.

Example: Prove $\sqrt{2}$ is irrational.

1. **ATaC:** Suppose $\sqrt{2}$ is rational. Then we can write:

$$\sqrt{2} = \frac{a}{b}$$

where a and b are integers with no common divisors other than 1 (i.e., the fraction is in its simplest form).

2. **Square Both Sides:**

$$2 = \frac{a^2}{b^2} \Rightarrow a^2 = 2b^2$$

This implies that a^2 is even (since it is twice an integer). Therefore, a must also be even (by a lemma which states that if a^2 is even, then a is even).

3. **Express a as an Even Number:**

$$a = 2k \text{ for some integer } k$$

Substitute $a = 2k$ into the equation:

$$(2k)^2 = 2b^2 \Rightarrow 4k^2 = 2b^2 \Rightarrow 2k^2 = b^2$$

This implies that b^2 is even, and thus b must also be even.

4. **Contradiction:** Since both a and b are even, they have a common factor of 2. This contradicts our initial assumption that $\frac{a}{b}$ is in its simplest form.

The contradiction shows that our assumption that $\sqrt{2}$ is rational is false. Therefore, $\sqrt{2}$ is irrational.

4 Recurrences (Ch. 2.3, L4)

4.1 Recurrences introduction (Ch. 4.1)

Divide-and-conquer method is useful to solve recurrences, which has three steps:

1. **Divide** the problem into one or more subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the subproblem solutions to form a solution to the original problem.

Definition: A **recurrence** is an equation (or inequality) that describes a function in terms of its value on other, typically smaller, arguments.

- **Inequality:** You will use Ω (i.e. lower bound) or O (i.e. upper bound).

A recurrence $T(n)$ is **algorithmic** if, for every sufficiently large **threshold** constant $n_0 > 0$, the following two properties hold:

1. $\forall\ n < n_0$, $T(n) = \Theta(1)$ (i.e. $\exists\ c_1,\ c_2 \in \mathbb{R}$ s.t. $0 < c_1 \leq T(n) \leq c_2$ for $n < n_0$)

2. $\forall\ n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations (prevents infinite recursive loop or failure to compute a solution).

Intuition/Tips: Whenever a recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic.

- **Implication:** This means we can pick any sufficiently large threshold constant n_0.

4.2 Mergesort

Definition: The Merge Sort algorithm is defined as:

$$\text{mergesort}(A, p, r) \to O(n \log n) \tag{24}$$

where A is the array to be sorted, p is the starting index, and r is the ending index.

```
def merge_sort(A, p, r):
    if p >= r:                    # zero or one element?
        return

    q = (p + r) // 2              # midpoint of A[p : r]
    merge_sort(A, p, q)           # recursively sort A[p : q] --> T(n/2)
    merge_sort(A, q + 1, r)       # recursively sort A[q + 1 : r] --> T(n/2)
    # Merge A[p : q] and A[q + 1 : r] into A[p : r]
    merge(A, p, q, r) # --> O(n)
```

Listing 1: Merge Sort Pseudocode

The time complexity of mergesort is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \tag{25}$$

- $2T(n/2)$ is the recursive time complexity of handling a subproblem half the size.
- $O(n)$ is the linear time required to merge the results.

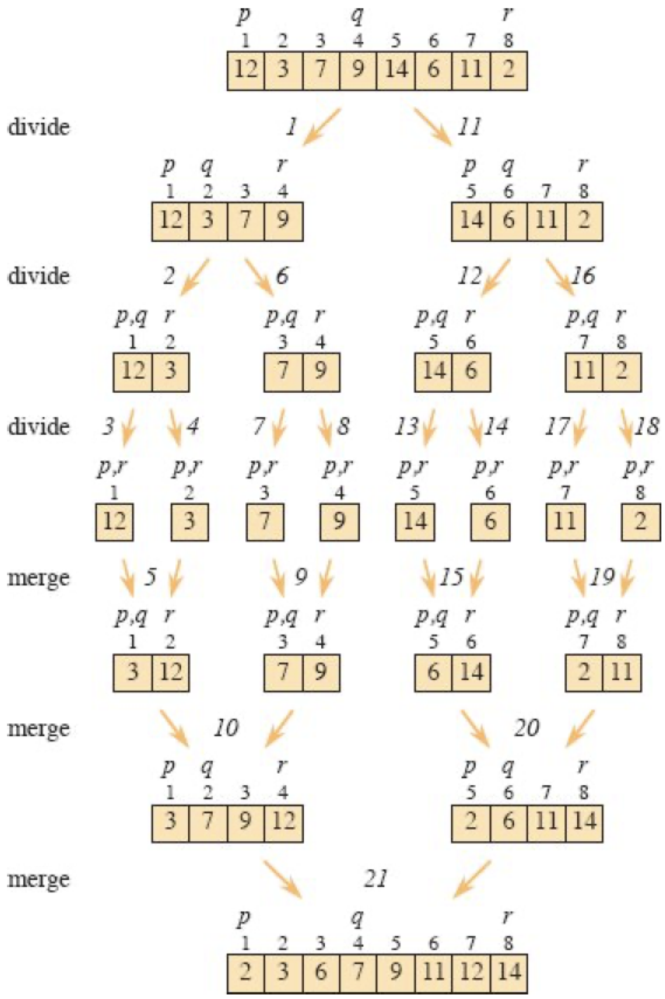


Figure 2: Merge sort visualization.

4.3 Master theorem (Ch. 4.5 pg. 101-6)

Theorem: Let $a \geq 1$, $b > 1$, and $f(n)$ be a function, so that the recurrence is

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{26}$$

Then the asymptotic behavior of $T(n)$ is

1. If $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ for $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b(a)}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\log n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$ for $\epsilon > 0$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for $0 < c < 1$, then $T(n) = \Theta(f(n))$.

Process:

1. Identify the recurrence relationship.
2. State a, b, and $f(n)$. Make sure the conditions are met.
3. Calculate $n^{\log_b a}$.
4. Compare $f(n)$ with $n^{\log_b a}$ to see which case the function applies too.
   (a) If ϵ case is used, then apply an abitrary value to see (usually natural numbers work well).
5. Write down the answer by applying the Master Theorem.

Example: Find the time complexity of Merge Sort using Master Theorem.

1. **Identify the Recurrence Relation:**
   - The recurrence relation for the merge sort algorithm is given by:
   $$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$
   - This represents dividing the problem into two subproblems of half the size and then merging the results in linear time.
2. **State Parameters:**
   - Compare the recurrence relation with the general form:
   $$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
   - For the given problem:
     - $a = 2$: The number of subproblems.
     - $b = 2$: The factor by which the problem size is divided.
     - $f(n) = O(n)$: The cost of dividing and merging the results.
3. **Calculate $n^{\log_b a}$:**
   - Compute $\log_b a$:
   $$\log_b a = \log_2 2 = 1$$
   - Thus, $n^{\log_b a} = n^1 = n$.
4. **Compare $f(n)$ with $n^{\log_b a}$:**
   - $f(n) = O(n)$ and $n^{\log_b a} = n$.
   - Since $f(n) = O(n)$ and $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$, this fits Case 2 of the Master Theorem.
5. **Apply the Master Theorem - Case 2:**
   - Case 2 states: If $f(n) = \Theta(n^{\log_b a})$, then:
   $$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$
   - Therefore, the time complexity of merge sort is:
   $$T(n) = \Theta(n \log n)$$

Example: Find the time complexity of this recurrence using Master Theorem.

1. **Identify the Recurrence Relation:**
   - The recurrence relation is:
   $$T(n) = 9T\left(\frac{n}{3}\right) + n$$
2. **State Parameters:**
   - Comparing with the general form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we have:
     - $a = 9$: Number of subproblems.
     - $b = 3$: Factor by which the problem size is divided.
     - $f(n) = n$: The cost of the work done outside the recursive calls.
3. **Calculate $n^{\log_b a}$:**
   - Compute $\log_b a$:
   $$\log_3 9 = 2$$
   - Thus, $n^{\log_b a} = n^2$.
4. **Compare $f(n)$ with $n^{\log_b a}$:**
   - Given $f(n) = n$, we have:
   $$f(n) = n = O\left(n^{\log_b a - \epsilon}\right) = O\left(n^{2-1}\right) = O(n)$$
   - Since $f(n) = O\left(n^{\log_b a - \epsilon}\right)$, this fits Case 1 of the Master Theorem.
5. **Apply the Master Theorem - Case 1:**
   - Case 1 states: If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$, then:
   $$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^2)$$

- Hence, the time complexity is:
$$T(n) = \Theta(n^2)$$

Example: Find the time complexity of this recurrence using Master Theorem.

1. **Identify the Recurrence Relation:**
   - The recurrence relation is:
   $$T(n) = 3T\left(\frac{n}{4}\right) + n\log(n)$$
2. **State Parameters:**
   - Comparing with the general form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we have:
     - $a = 3$: Number of subproblems.
     - $b = 4$: Factor by which the problem size is divided.
     - $f(n) = n\log(n)$: The cost of the work done outside the recursive calls.
3. **Calculate $n^{\log_b a}$:**
   - Compute $\log_b a$:
   $$\log_4 3 \approx 0.793$$
   - Thus, $n^{\log_b a} = n^{0.793}$.
4. **Compare $f(n)$ with $n^{\log_b a}$:**
   - $f(n) = n\log(n)$ is compared with $\Omega\left(n^{0.793+0.2}\right)$, which implies:
   $$n\log(n) = \Omega\left(n^{0.993}\right)$$
   - This indicates that $f(n)$ dominates $n^{\log_b a}$ with a polynomial difference, which suggests considering Case 3 of the Master Theorem.
5. **Verify Condition for Case 3 of the Master Theorem:**
   - Check if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$:
   $$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq \frac{3}{4}n\log(n)$$
   - This inequality is true for the chosen constants, satisfying Case 3.
6. **Apply the Master Theorem - Case 3:**
   - Since $f(n) = \Omega\left(n^{\log_b a+\epsilon}\right)$ and the regularity condition is satisfied, we conclude:
   $$T(n) = \Theta(n\log n)$$

4.4 Substitution (Ch. 4.3 pg. 90-4)

Process:

1. Guess the form of the solution for $T(n) =$?
2. Use induction to show that the solution works.
   (a) Basis: Find the base case using values of n that correspond (i.e. make sense) with the guessed solution.
   (b) Inductive hypothesis:
   (c) Inductive step:
3. Find the constants.

Intuition/Tips:

- **Bounds:** Rather than trying to prove Θ-bound directly, first prove an O-bound, and then prove an Ω-bound, then use Theorem 3.1.
- **Making a good guess:**
  - See if the recurrence is similar to one you've seen before, then guessing a similar solution.
  - Determine loose upper and lower bounds on the recurrence and then reduce your range of uncertainty.
- **Trick:** Subtract a lower-order term when the math fails to work out in the induction proof.
- **Avoid:**
  - Don't use asymptotic notation in the inductive hypothesis for the sub-method.

- You must be careful that the constants hidden by any asymptotic notation are the same constants throughout the proof.

Example: Find the time complexity of the recurrence using sub-method.

1. **Guess the Form of the Solution:**
   - Given recurrence relation:
   $$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$
   - Guess: $T(n) = cn \log n$.
2. **Basis:**
   - Check the base cases:
   $$T(2) = 4, \quad T(3) = 5$$
   - Both satisfy $T(n) = cn \log n$, verifying the base cases.
3. **Inductive Hypothesis:**
   - Assume $T(k) \leq ck \log k$ for all $k < n$.
   - Specifically, assume:
   $$T\left(\frac{n}{2}\right) \leq c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right)$$
4. **Inductive Step:**
   - Show it holds for $T(n)$:
   $$T(n) = 2T\left(\frac{n}{2}\right) + n$$
   - Substitute the inductive hypothesis:
   $$T(n) \leq 2\left(c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right)\right) + n$$
   - Simplify:
   $$= cn \log\left(\frac{n}{2}\right) + n$$
   - Use the logarithm property $\log(ab) = \log a + \log b$:
   $$= cn(\log n - \log 2) + n$$
   $$= cn \log n - cn \log 2 + n$$
   - Factor n:
   $$= n(c \log n - c \log 2 + 1)$$
5. **Find the Constant c:**
   - To keep $T(n) \leq cn \log n$, ensure:
   $$c \log n - c \log 2 + 1 \leq c \log n$$
   - Simplifying, we need:
   $$-c \log 2 + 1 \leq 0$$
   - This implies:
   $$c \geq \frac{1}{\log 2}$$
   - Choose $c \geq 2$ (since $\log 2 \approx 0.693$), which satisfies the inequality $2 \geq 1.44$.

4.5 Recursion tree method (Ch. 4.4 pg. 95-101)

Definition: In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

Process:

1. Sum the costs within each level of the tree to obtain the per-level costs.

2. Sum all the per-level costs to determine the total cost of all levels of the recursion.
3. (1) Generate a good-guess, then verify using sub-method. (2) Use as a direct solution.

Intuition/Tips: How to make the recursion tree based on the recurrence.

Example:

1. **Given Recurrence Relation:**
   - The recurrence relation is:
   $$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{2n}{3}\right) + n$$
2. **Building the Recursion Tree:**
   - The tree starts with $T(n)$ at the root.
   - Each node $T(n)$ branches into two child nodes:
   $$T\left(\frac{n}{4}\right) \quad \text{and} \quad T\left(\frac{2n}{3}\right)$$
   - This branching continues recursively until the problem size becomes small (base case).

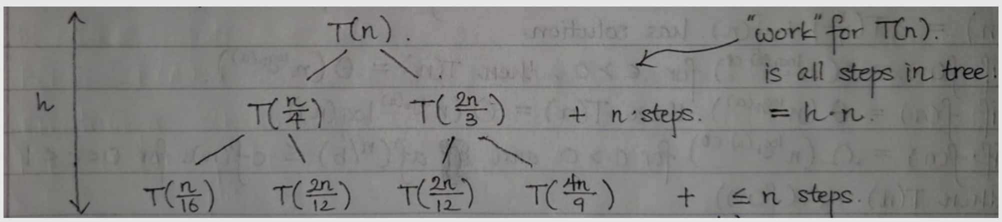


Figure 3: Recursion tree that is made by subbing in the $T(\#)$ into the recurrence relation to get the nodes below.

3. **Calculating Work Done at Each Level:**
   - At the root, the work done is n.
   - At the next level, the work is divided between:
   $$T\left(\frac{n}{4}\right) \quad \text{and} \quad T\left(\frac{2n}{3}\right)$$
   - This pattern continues, and the work at each level is the sum of the work done by each subproblem.
   - The total work at each level is n, as shown by the distribution of the work across the nodes.
4. **Height of the Tree:**
   - The longest path (height h) of the tree is determined by the rightmost path since $\frac{2}{3}$ is larger than $\frac{1}{4}$.
   - The height h can be calculated using the formula:
   $$\left(\frac{2}{3}\right)^h \cdot n = 1$$
   - Solving for h:
   $$h = \log_{3/2}(n)$$
5. **Total Work Done in the Tree:**
   - The total work is the sum of the work done at each level times the height of the tree:
   $$h \cdot n$$
   - Substituting the value of h:
   $$h \cdot n = \log_{3/2}(n) \cdot n$$
   - This expression simplifies to:
   $$O(n \log n)$$
   - Therefore, the total work done by the recursion tree is $O(n \log n)$.

5 Graphs, Trees (Ap. B.4-5, L6)

5.1 Graphs

5.1.1 Directed and undirected graphs

Definition:

- **Directed graph (digraph):** G is a pair (V, E), which are vertices V and edges E.
  - **Self-loop:** Edges from a vertex to itself.
- **Undirected graph:** $G = (V, E)$, where E consists of *unordered* pairs of vertices (i.e. direction doesn't matter)
  - **Self-loop:** Forbidden.

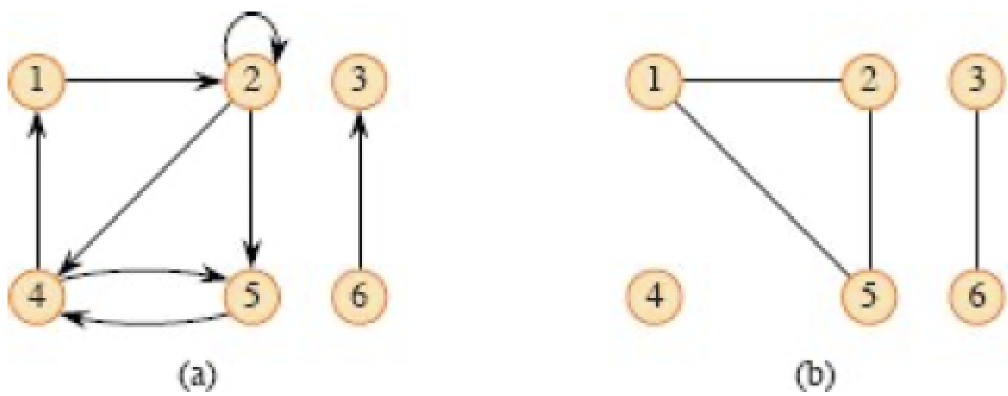


Figure 4: (a) Directed graph, (b) Undirected graph.

5.1.2 Terminology

Terminology:

- **Weighted G:** e.g. distance, cost, etc.
- **Path:** A sequence of vertices in which each vertex is adjacent to the next one.
- **Simple Path:** A path with no repetition of vertices.
- **Simple Cycle:** Simple path with same start/end vertex.
- **Acyclic Graph:** A graph with no cycles is an acyclic graph.
- **Directed Acyclic Graph:** A DAG is a directed acyclic graph.
- **Connected:** Two vertices are connected if there is a path between them.
- **Connected graph:** $\exists$ path between $\forall$ 2 vertices.
- **Degree of V (Undirected G):** Number of edges incident on it.
- **In/Out Degree of V (Directed G):** Out-degree is the # of edges leaving it, while in-degree is the # of edges entering it.
- **Degree of V (Directed G):** In-degree plus out-degree.
- **Degrees of all V**: $2E$
- **Bipartite Gs:** V can be partitioned into 2 sets V_1 and V_2 s.t. $V_1 \cap V_2 =$ and $V_1 \cup V_2 = V$ and adjacencies only between elements of V_1 and V_2.

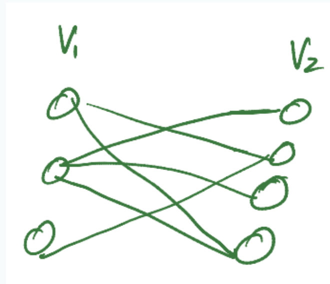


Figure 5: Bipartite graph.

- **Induced subgraph:** Subset of G and the associated edges.
- **Complete G (clique):** $\exists$ edge between $\forall$ 2 vertices.

5.1.3 Graph representation

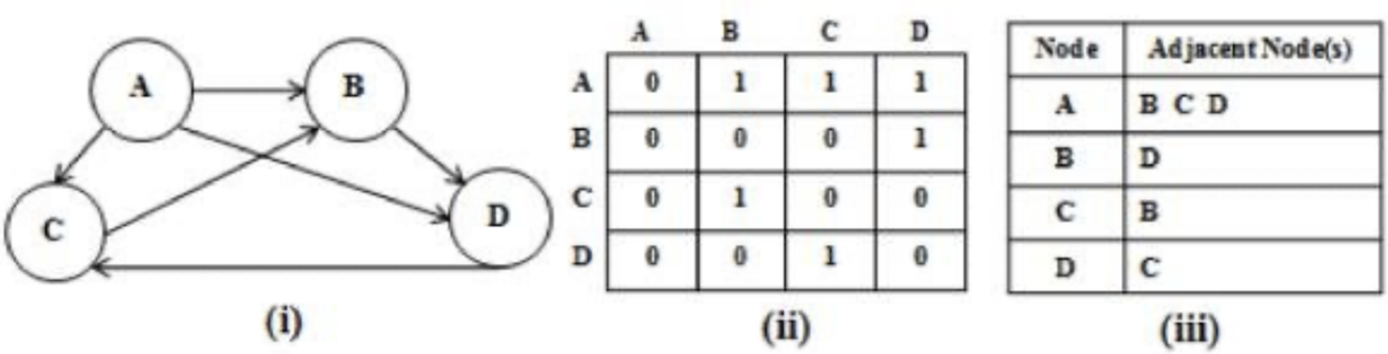
Definition:
Adjacency matrix (AM): An $n \times n$ matrix where $M[i][j] = 1$ if there is an edge between v_i and v_j, and 0 otherwise.

Adjacency list (AL): For $n = |V|$ vertices, n linked lists. The ith linked list, $L[i]$ is a list of all the vertices that are adjacent to vertex i.

Is there an edge between v_i and v_j?
- **AM:** $O(1)$
- **AL:** $O(d)$ where d is the maximum degree in the graph.

Find all vertices adjacent to v_i:
- **AM:** $O(|V|)$ where $|V|$ is the number of vertices in the graph.
- **AL:** $O(d)$

Space requirements:
- **AM:** $O(|V|^2)$
- **AL:** $O(|V| + |E|)$
  - AL is good for sparse G (i.e. $E << V^2$).



	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	1	0	0
D	0	0	1	0

Node	Adjacent Node(s)
A	B C D
B	D
C	B
D	C

Figure 6: (i) Directed graph, (ii) Adjacency matrix, (iii) Adjacency list

5.1.4 Clique

Definition: Every two vertices have an edge.

$$\#edges = \frac{V(V-1)}{2}$$

5.2 Free trees

Definition: A free tree is a connected, acyclic, undirected graph.

5.2.1 Properties

Definition: Let $G = (V, E)$ be an undirected graph. The following statements are equivalent:

1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected, and $|E| = |V| - 1$.
5. G is acyclic, and $|E| = |V| - 1$.
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

- **Note:** There's a proof to show each of these statements are equivalent.

5.3 Forest

Definition: An undirected graph is acyclic but possibly disconnected.

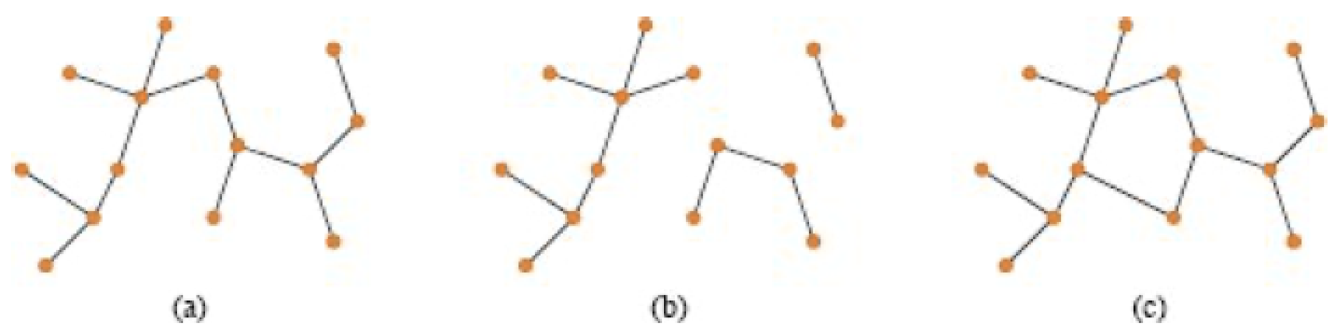


Figure 7: (a) A free tree, (b) A forest, (c) a graph that contains a cycle and is therefore neither a tree nor a forest.

5.4 Rooted and ordered trees

5.4.1 Rooted trees

Definition: A **rooted tree** is a free tree in which one of the vertices is distinguished from the others.

- **Root:** Distinguished vertex of the tree.
- **Node:** Vertex of a rooted tree.

5.4.2 Ordered trees

Definition: An **ordered tree** is a rooted tree in which the children of each node are ordered.

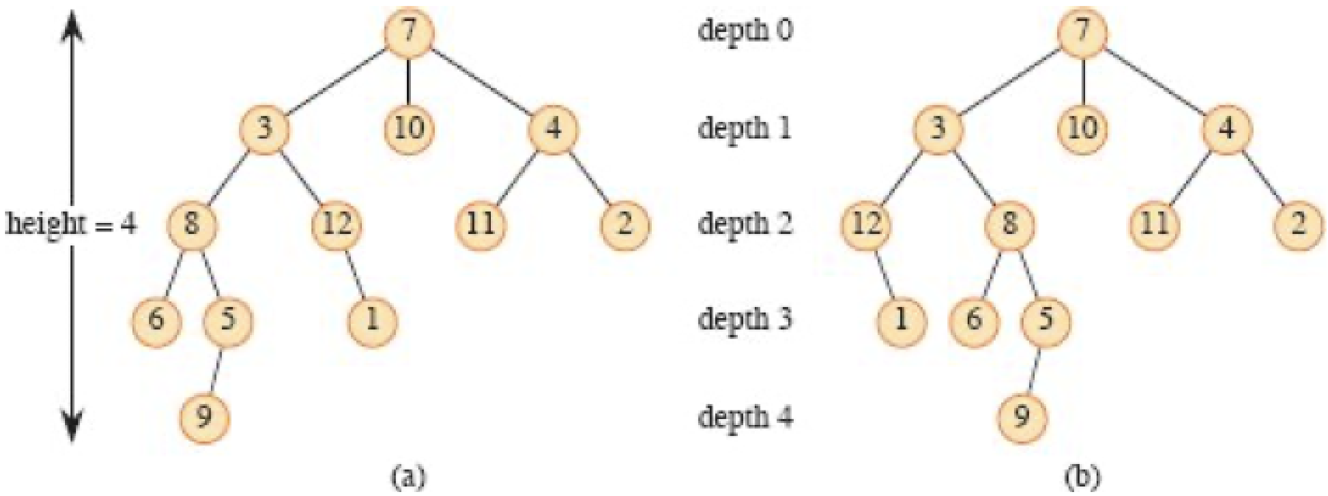


Figure 8: Rooted and ordered trees. If the tree is ordered, the relative left-to-right order of the children of a node matters, but if rooted, then they are the same tree.

5.4.3 Terminology

Terminology:
- **Parent:** A node y is the parent of node x if y is directly connected to x on the path from the root.
- **Child:** A node x is a child of node y if y is the parent of x.
- **Siblings:** Nodes are siblings if they share the same parent.
- **Leaf (or External Node):** A leaf is a node with no children.
- **Internal Node:** An internal node is a nonleaf node, which means it has at least one child.
- **Degree:** The degree of a node x is the number of children it has.
- **Depth:** The depth of a node x is the length of the path from the root to x.
- **Level:** A level of a tree consists of all nodes at the same depth.
- **Height:** The height of a node is the number of edges in the longest path from that node to a leaf.
  - **Height of tree:** From root to any leaf.

5.5 Binary and positional trees

5.5.1 Binary trees

Definition: A **binary tree** T is a structure defined on a finite set of nodes that either
- contains no nodes, or
- is composed of three disjoint sets of nodes: a **root** node, a **left subtree**, and a **right subtree**.

5.5.2 Terminology

Terminology:
- **Empty Tree:** A binary tree with no nodes.
- **Left and Right Child:** The roots of the non-empty left and right subtrees of the root.
- **Full Binary Tree:** Every node is either a leaf or has exactly two children.
- **Position Matters:** The distinction between left and right children is crucial in a binary tree, unlike in general ordered trees.

5.5.3 Positional trees

Definition: The children of a node are labeled with distinct positive integers.

5.5.4 K-ary trees

Definition: A positional tree in which each node $\leq k$ children. A binary tree has $k = 2$.

A **complete k-ary tree** is a k-ary tree in which all leaves have the same depth, and every internal node has exactly k children.

6 Permutations, Combinations (Ap. C.1, L5)

6.1 Rule of sum and product

Definition: If there are m-ways for event A to happen and n-ways for event B to happen then...

Rule of product: $\exists\, m \times n$ ways for A *and* B to happen.

Rule of sum: $\exists\, m + n$ ways for A *or* B to happen.

6.2 Permutations

Definition: Number of ways to pick r distinct objects out of n where *order matters* and *repetition isn't allowed.*

$$P(n, r) = n(n-1)(n-2)\cdots(n-r+1) = \frac{n!}{(n-r)!} \tag{27}$$

- n: total number of elements in the set.
- r: number of elements taken from the set.

6.3 Permutations with identical items

Definition: If there are m kinds of items and q_k, $k = 1, \dots, m$ of each kind, then total number of permutations where *order matters* is

$$\binom{n}{q_1, \dots, q_m} = \frac{n!}{q_1!\, q_2! \cdots q_m!} \tag{28}$$

- $\sum_{k=1}^{m} q_k = n$

6.4 Permutations with repetitions

Definition: Number of ways to arrange r-objects out of n objects with unlimited repetition is given by: n^r.

6.5 Combinations

Definition: Number of ways to choose r objects from n where *order doesn't matter.*

$$C(n, r) = \binom{n}{r} = \frac{P(n, r)}{r!} = \frac{n!}{r!(n-r)!} \tag{29}$$

6.6 Binomial theorem

Definition:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} \tag{30}$$

- $n \in \mathbb{N}$ and $x, y \in \mathbb{R}$

7 Probability (Ap. C.2, L8)

7.1 Sample space

Definition: The set of *all possible outcomes* of a statistical experiment, denoted by S.

7.2 Event

Definition: A subset of a sample space S. An event is any outcome or combination of outcomes.

7.3 Probability axioms

Definition: For A, $B \subseteq S$

1. $0 \leq P(A) \leq 1$
2. $P(S) = 1$
3. $P(A \cup B) = P(A) + P(B)$ for two mutually exclusive events A and B.

7.4 Additive rule

Definition:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{31}$$

7.5 Uniform distribution

Definition: If $\forall s \in S$ has probability $P(s) = \dfrac{1}{|S|}$, then it is a uniform distribution.

7.6 Independence

Definition: $P(A \cap B) = P(A)P(B)$ if A, B independent.

7.7 Bayes theorem

Definition: For events with $P(A) > 0$ and $P(B) > 0$, the probability A happens given B happens is:

$$P(B|A) = \frac{P(B \cap A)}{P(A)} = \frac{P(A|B)P(B)}{P(A)} \tag{32}$$

7.8 Bayes' rule with total probability

Definition: Suppose $C_1, \ldots, C_k$ is a partition. Then

$$P(B|A) = \frac{P(B)P(A|B)}{\sum_{i=1}^{k} P(C_i) P(A|C_i)} \tag{33}$$

Often B is an element of $C_1, \ldots, C_k$, say $B = C_n$. Then

$$P(C_n|A) = \frac{P(C_n) P(A|C_n)}{\sum_{i=1}^{k} P(C_i) P(A|C_i)} \tag{34}$$

Process:
1. Write down all the probabilities.
2. Try solving the problem directly using definitions.

Intuition/Tips: If given P(A|B) and want P(B|A), then automatically use Bayes' Rule.

7.9 Discrete random variable

Definition: An RV is a function that associates a real number with each element of the sample space. Denote RVs with capital letters.

7.10 Probability mass function

Definition: The set of ordered pairs $(x, f(x))$ of the discrete RV X if, for each possible outcome x,
1. $f(x) \geq 0$ for each outcome $X = x$
2. $\sum_x f(x) = 1$ (i.e. total probability sums to 1)
3. $f(x) = P(X = x)$ (i.e. probability of each outcome)

7.11 Expectation

Definition: Let X be an RV with distribution $f(x)$, then

$$E[X] = \sum_{x \in X} x f(x) \tag{35}$$

where the sum is taken over all possible values of X.

7.12 Properties of expectation

Definition:
1. $E[X + Y] = E[X] + E[Y]$ (linearity)
2. $E[\alpha X] = \alpha E[X]$ (linearity)
3. $E[XY] = E[X]E[Y]$ if independent

8 Heaps, Heapsort (Ch. 6, L9)

8.1 Intro to heapsort

8.1.1 In-place Sorting

Definition: Given an array A to sort the numbers, sorts within the array and uses a constant number of variables to do bookkeeping.

- **Time Complexity**: $O(n \log n)$.
- **Explanation:** Describe in terms of a tree.
- **Pseudo-code:** Uses the array representation.

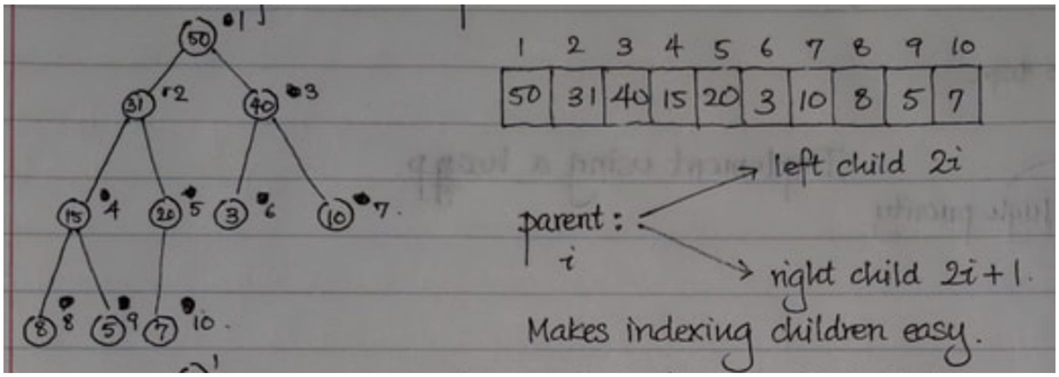


Figure 9: (Left) Tree format heap. (Right) Array format heap with formulas for parent and children.

8.1.2 Indexing

Definition:

1. **Parent:** $\left\lfloor \frac{i}{2} \right\rfloor$
2. **Left child:** $2i$
3. **Right child:** $2i + 1$

8.1.3 Heap-tree: (2 properties)

Definition:

- **Heap Shape:** A complete binary tree where the last level is not filled, but leaves are pushed to the left.
- **Heap Order (maxheap):** A[Parent(i)] $\geq$ A[i]
- **Heap Order (minheap):** A[Parent(i)] $\leq$ A[i]

8.1.4 Height

Definition: A heap of n elements is based on a complete binary tree, its height is $\Theta(lgn)$.

8.2 Heap operations

8.2.1 Insert

Definition:

```
Insert:
    A[length + 1] = new_key
    length = length + 1
    bubble_up(A, length)
```

```

```

- **Time Complexity:** $O(lgn)$

8.2.2 Bubble up

Definition:

```
bubble_up (A, i):
    repeat
        swap (A[i] <=> A[floor(i/2)]) # comparing yourself with parent
        if A[i] is larger

```

- **Time Complexity:** $O(lgn)$

Intuition/Tips: Insert at the end and then bring up to its proper position.

8.2.3 Extract max

Definition:

```
Extract_Max (A):
    max = A[1]
    A[1] = A[length] # put last element at the top
    length = length - 1
    bubble_down(A[1]) # bubble down to the proper location

```

- **Time Complexity:** $O(lgn)$

8.2.4 Bubble down

Definition:

```
bubble_down (i):
    repeat
        compare A[i] with A[2i] and A[2i + 1]
        exit if A[i] is larger or A[i] is a leaf
        swap A[i] <=> swap(A[2i], A[2i + 1])

```

- **Time Complexity:** $O(lgn)$

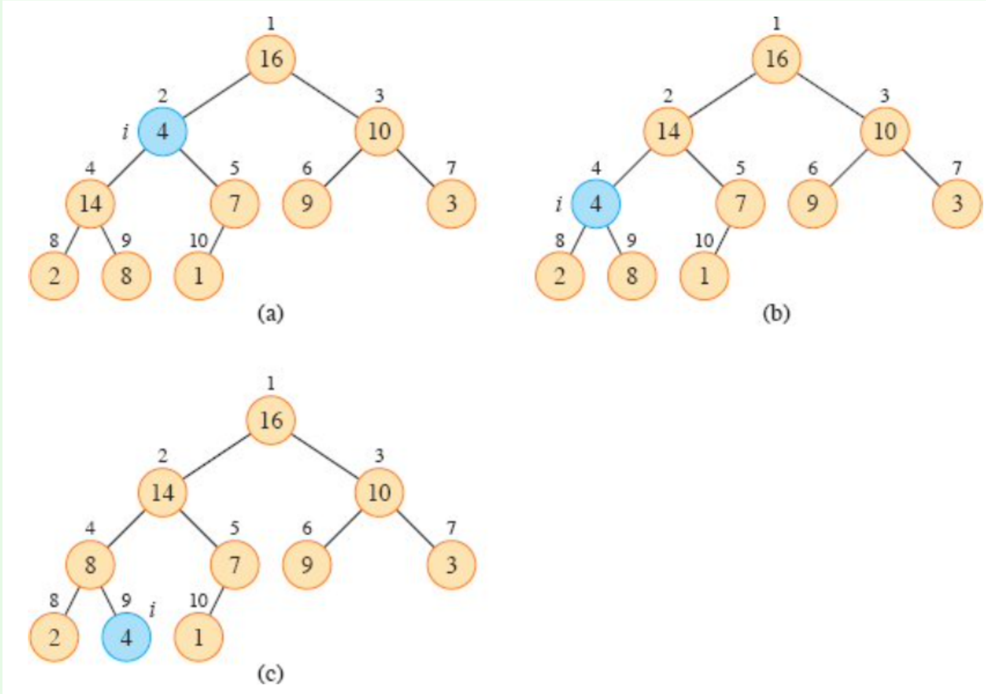


Figure 10: Bubble down.

8.3 Heapsort

Definition:

```
    Heapsort (A):
        Build-Heap(A)        # O(n)
        for i = 1 to n - 1   # O(n) iterations
            Extract_Max(A)   # O(log n)
```

- **Time Complexity:** $O(nlgn)$

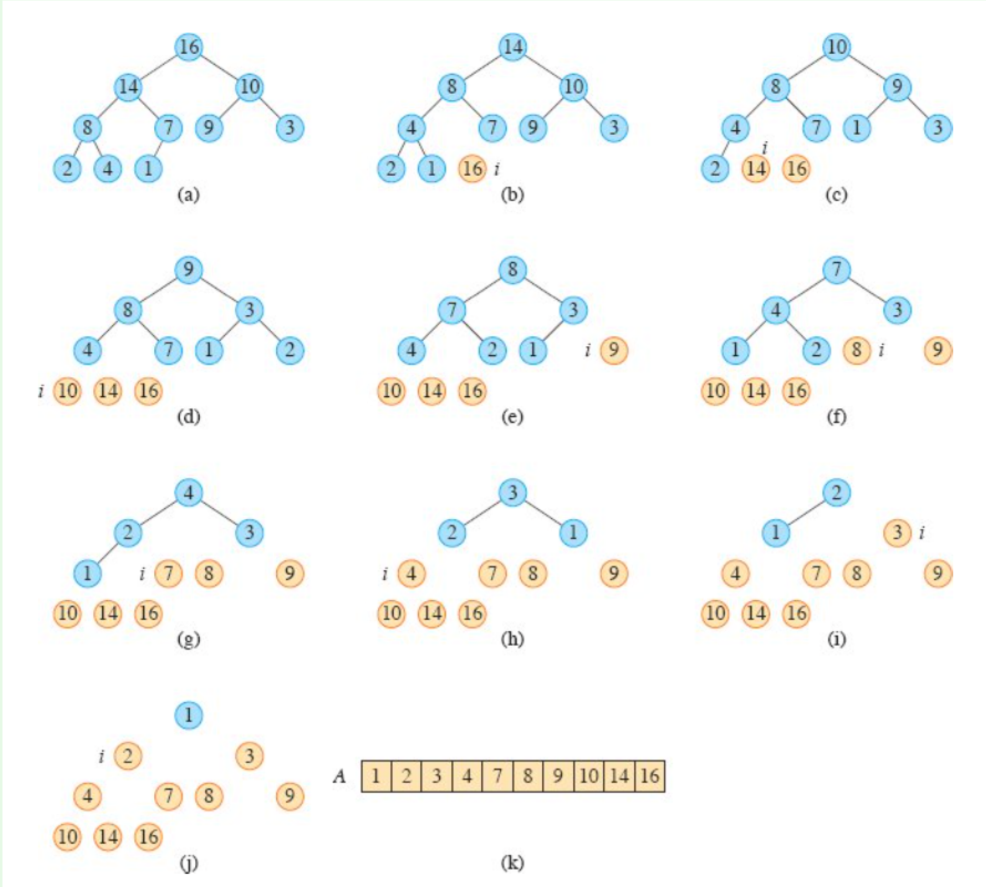


Figure 11: (a) Max-heap data structure after Build-Heap. (b)-(j) Extract max. (k) Sorted array.

8.3.1 Build heap

Definition:

```
Build_heap (A):
    for i = floor(length / 2) down to 1
        bubble_down(A, i)
```

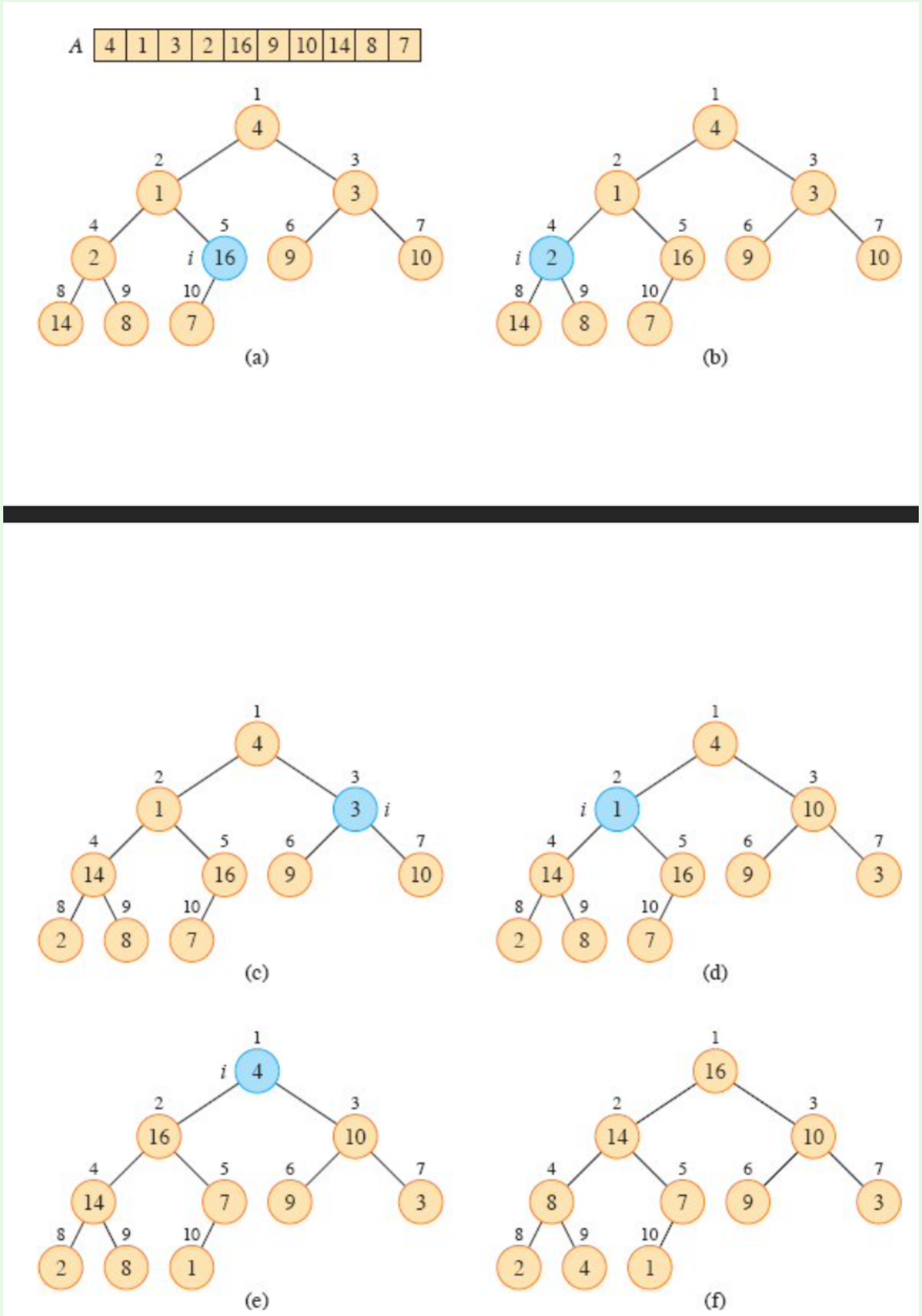


Figure 12: Build heap.

8.4 Heap runtime and priority queue

8.4.1 Tight bound for build heap

Definition: There are *at most* $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h in a heap.

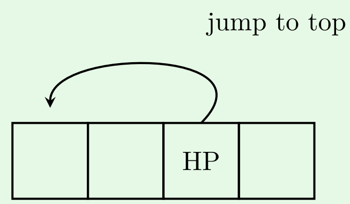
The tight bound is

$$\sum_{h=0}^{\log(n)} \# \text{ nodes at height h} \cdot \text{height of the node} = \sum_{h=0}^{\log(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h$$
$$\leq O\left(\frac{n}{2} \cdot \sum_{h=0}^{\infty} h \cdot \frac{1}{2^h}\right)$$
$$= O(n) \cdot \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}$$
$$= O(n) \cdot 2 = O(n)$$

8.4.2 Priority queue

Definition: A queue where the first element dequeued is the one with the highest priority.

- Implement a PQ using a heap for efficient priority management.



9 Quicksort (Ch. 7, L10)

9.1 Intro

9.1.1 QS algorithm

Definition:

```
Quicksort (list in, int left, int right)
    pivot = Partition(in, left, right)
    if (pivot > left)
        Quicksort(in, left, pivot)
    if (pivot < right)
        Quicksort(in, pivot + 1, right)

```

Listing 2: Quicksort Algorithm Pseudocode

9.1.2 Partition

Definition:

```
int Partition (in, left, right)
    ls = left
    pivot = in(left)
    for i = left + 1 to right
        if (in(i) <= pivot)
            ls = ls + 1
            swap(in(i), in(ls))
    swap(in(left), in(ls))
    return ls

```

Listing 3: Partition Function Pseudocode

Intuition/Tips:

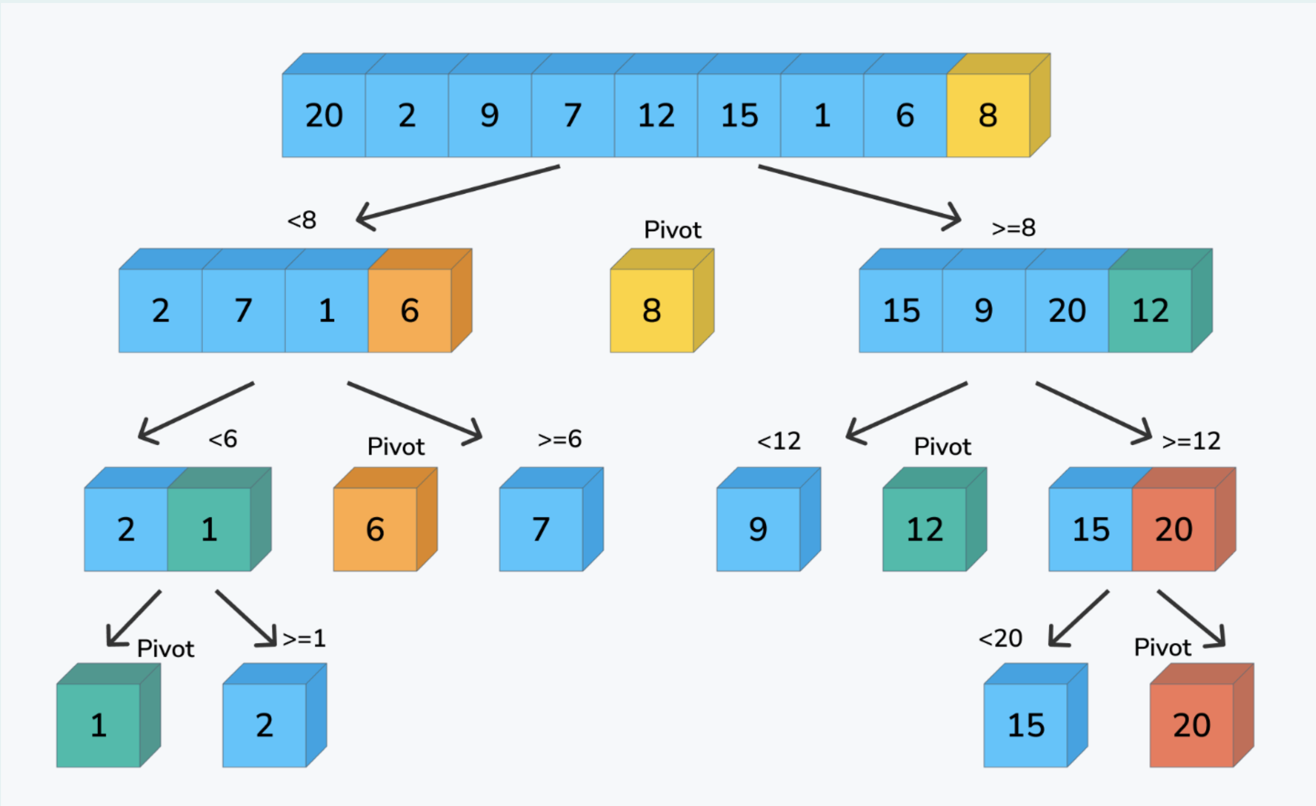


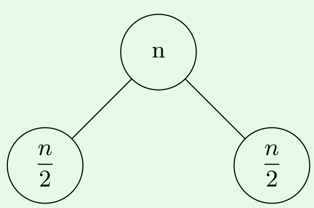
Figure 13: Quicksort example.

9.2 QS basic analysis

9.2.1 QS best case

Definition: The array is always split exactly in half, leading to a balanced partition. The recurrence relation for quicksort in this scenario is:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$



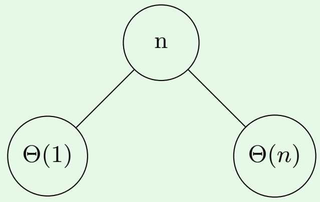
Now using the Master Theorem:

$$T(n) = \Theta(n \log n)$$

9.2.2 QS worst case

Definition: The array is already sorted (or reverse sorted) and we choose the first or last element as the pivot, the recurrence relation for quicksort is:

$$T(n) = T(n-1) + \Theta(n)$$



This recurrence relation expands as follows:

$$
\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= (T(n-2) + \Theta(n-1)) + \Theta(n) \\
&= (T(n-3) + \Theta(n-2)) + \Theta(n-1) + \Theta(n) \\
&= \cdots \\
&= \Theta\left(\sum_{i=1}^{n} i\right) \\
&= \Theta\left(\frac{n(n+1)}{2}\right) \\
&= \Theta(n^2)
\end{aligned}
$$

9.2.3 QS average case

Definition: In the average case, the recurrence relation for quicksort can be expressed as:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$$

We can visualize this with a recursion tree:

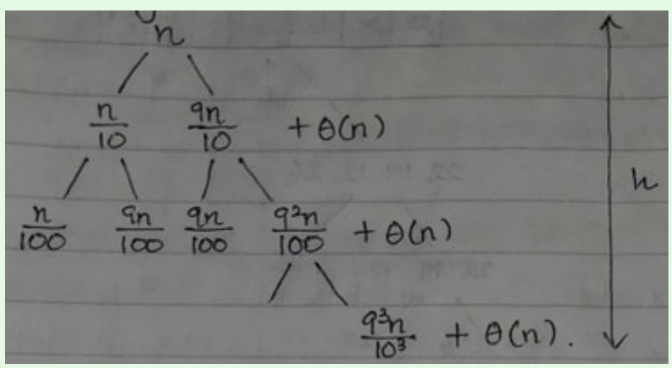


Figure 14: Quicksort average case in which each level is derived by subbing in the $T(\#)$ back into the equation above.

Based on the recursive tree structure and the average-case recurrence relation, we can derive the time complexity as follows:

$$T(n) = h \cdot \Theta(n)$$

Now, let's calculate the height h of the tree:

$$\left(\frac{9}{10}\right)^h n = 1$$

$$h = \log_{10/9}(n)$$

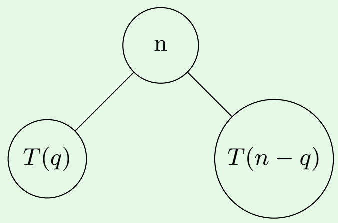
Substituting this back into the overall complexity:

$$
\begin{aligned}
T(n) &= h \cdot \Theta(n) \\
&= \log_{10/9}(n) \cdot \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$

9.3 Worst-case (formal)

Definition: The worst-case recurrence relation for quicksort can be expressed as:

$$T(n) = \text{time to QS n-elements} = \max_{1 \leq q \leq n-1} \{T(q) + T(n-q)\} + \Theta(n)$$



We use substitution to show that $T(n) \leq cn^2$ for some constant c.

1. Guess $T(n) \leq cn^2$.
2.

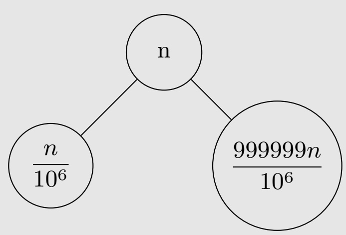
$$
\begin{aligned}
T(n) &\leq \max_{1 \leq q \leq n-1} \left\{cq^2 + c(n-q)^2\right\} + \Theta(n) \quad \text{(Achieves max at } q = 1 \text{ or } q = n-1\text{)} \\
&= c \max_{1 \leq q \leq n-1} \left\{q^2 + (n-q)^2\right\} + \Theta(n) \quad \text{(As second derivative is positive, plug } q = 1\text{)} \\
&\leq cn^2 - 2c(n-1) + \Theta(n) \quad \text{(We can pick a large } c \text{ to dominate the constant } \Theta(n)\text{)} \\
&\leq cn^2
\end{aligned}
$$

Therefore, using the substitution method, we can show that $T(n) \leq cn^2$, confirming that the worst-case time complexity of the quicksort algorithm is $O(n^2)$.

9.4 Randomized QS

9.4.1 Motivation for randomized QS

Example:



In this example, the array of size n is split into highly unbalanced sub-arrays:

- One sub-array is $\frac{n}{10^6}$, very small compared to n.
- The other sub-array is $\frac{999999n}{10^6}$, almost the entire size of n.

This unbalanced split may lead to increased recursion depth and higher running times, potentially reaching the worst-case $O(n^2)$ complexity.

Motivation for Randomized QS:

- Avoiding worst-case scenarios
- Ensuring balanced splits

- Works against sorted and reverse sorted arrays.

9.4.2 Random partition

Definition:

```
Rand-Partition (list in, left, right)
    i = random(left, right)
    swap(in(left), in(i))
    return Partition(in, left, right)

```

Listing 4: Rand-Partition Function Pseudocode

10 Counting sort, Radix sort (Ch. 8)

10.1 Lower bound on sorting and counting sort

10.2 Radix sort

11 Selection sort, Binary search trees (Ch. 12)

11.1 Selection sort

11.2 Binary search trees

12 Red black trees (Ch. 13)

12.1 Properties

12.2 Balance proof

12.3 Operations

13 Hash tables, Hashing (Ch. 11)

13.1 Motivation

13.2 Resolution by chaining

13.3 Resolution by open addressing

14 Dynamic programming (Ch. 14)

14.1 DP matrix multiplication

14.2 DP longest common subsequence

15 Greedy algorithms (Ch. 15)

16 Amortized analysis (Ch. 16)

17 Splay trees

18 Graph algorithms (Ch. 20)

18.1 Intro

18.2 Breadth-first search

18.3 Depth-first search

19 Minimum spanning trees (Ch. 21)

20 Shortest paths (Ch. 22)

21 Maximum flow (Ch. 24)

22 P, NP, and NPC introduction (Ch. 34)

23 NPC (Ch. 34)