CS3230 Analysis and Design of Algorithm Summary

Chen Yixun

May 9, 2026

Contents

1 Asymptotic Analysis

1.1 Definitions

- $\exists\ c > 0,\ n_0 > 0,\ s.t.\ \forall\ n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n) \Rightarrow f \in O(g)$
- $\exists\ c > 0,\ n_0 > 0,\ s.t.\ \forall\ n \geq n_0,\ 0 \leq c \cdot g(n) \leq f(n) \Rightarrow f \in \Omega(g)$
- $\exists\ c_1, c_2 > 0,\ n_0 > 0,\ s.t. \forall\ n \geq n_0,\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \Rightarrow f \in \Theta(g)$
- $\Theta(g) = O(g) \cap \Omega(g)$
- $\forall c > 0,\ \exists\ n_0 > 0\ s.t.\ \forall n \geq n_0 :\ 0 \leq f(n) < c \cdot g(n) \Rightarrow f \in o(g)$
- $\forall c > 0,\ \exists\ n_0 > 0\ s.t.\ \forall n \geq n_0 :\ 0 \leq c \cdot g(n) < f(n) \Rightarrow f \in \omega(g)$

1.2 Limits

- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} < \infty \Rightarrow f(n) \in \Theta(n)$
- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} > 0 \Rightarrow f(n) \in \Omega(n)$
- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \omega(n)$

1.3 Properties

- Reflexivity: $f \in O(f),\ \Theta(f),\ \Omega(f)$
- Transitivity: $f(n) \in O(g(n)),\ g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$, same for all five $O,\ \Theta,\ \Omega,\ o,\ \omega$
- Symmetry: $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$
- Complementarity: $f \in O(g) \Leftrightarrow g \in \Omega(f)$; $f \in o(g) \Leftrightarrow g \in \omega(f)$

1.4 Misc

$n! = \sqrt{2\pi n}(\dfrac{n}{e})^n(1 + O(1/n))$

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

2 Solve Recurrence

2.1 Telescoping

The sum of form $\sum_{n=0}^{N} a_n - a_{n+1} = a_0 - a_1 + a_1 - a_2 \cdots + a_{N-1} - a_N = a_0 - a_N$,

$$T(n) = f(n)T(\frac{n}{b}) + g(n),\ f(n) = \frac{h(n)}{h(\frac{n}{b})}$$

$$\frac{T(n)}{h(n)} = \frac{T(\frac{n}{b})}{h(\frac{n}{b})} + \frac{g(n)}{h(n)}$$

$$\frac{T(n)}{h(n)} = \frac{T(1)}{h(1)} + \sum_{i=1}^{\log_b n} F(i)$$

2.2 Substitution method

Guess a bound for function $T(n)$ and prove the guess is correct via **strong induction**. Need to be aware of the precise definition of $O(\cdot)$ and $\Omega(\cdot)$

How to construct guess (take an example of $O(\cdot)$)?

- Make a claim: $\forall\ n \geq n_0,\ T(n) \leq cf(n)$
- Decide the value of c and n_0 by arithmetic manipulation
- Reconstruct the formal proof using the chosen c and n_0

2.3 Recursion tree

It is a rooted tree where each vertex represents a recursive call and be labeled with the amount of local computation needed. The leaves represent base cases and will be labeled with fixed constant

2.4 Master theorem

2.4.1 Definition

Theorem (Master Theorem)**.** *Suppose that for some* $a > b,\ b > 1$, $T(n) = aT(\frac{n}{b}) + f(n)$. *let* $d = \log_b a$

- $\exists\ \epsilon > 0,\ f(n) \in O(n^{d-\epsilon}) \implies T(n) \in \Theta(n^d)$

- $\exists\ k \geq 0, f(n) \in \Theta(n^d \log^k n) \implies T(n) \in \Theta(n^d \log^{k+1} n)$
- $\exists\ \epsilon > 0,\ f(n) \in \Omega(n^{d+\epsilon})\ \wedge\ \exists\ 0 \leq c < 1\ \forall\ x > 0,\ (af(\frac{x}{b}) \leq cf(x)) \implies T(n) \in \Theta(f(n))$

Theorem (Extension)**.** *if* $f(n) \in \Theta(n^d \log^k n)$*, then*

- $k < -1, \implies T(n) \in \Theta(n^d)$
- $k = -1, \implies T(n) \in \Theta(n^d \log\log n)$
- $k > -1, \implies T(n) \in \Theta(n^d \log^{k+1} n)$

2.4.2 Proof of Master theorem

- Proof of case 3
  - Solving the base cases costs: $n^d T(1) \in o(f(n))$
  - Solving the splitting and combining costs $\sum_{k=0}^{\log_b n} a^k f(\frac{n}{b^k})$
  - $\because af(\frac{n}{b}) \leq cf(n), c < 1$ [*regularity condition*], $\sum_{k=0}^{\log_b n} a^k f(\frac{n}{b^k}) \leq \sum_{k=0}^{\log_b n} c^k f(n) = f(n)(1 + c + c^2 + \dots) = O(f(n))$
  - And obviously, that the splitting and combing term is $\Omega(f(n))$
  - Hence, the overall cost is $\Theta(f(n))$
- Proof of case 1
  - Solving the base case costs: $n^d T(1) = \Theta(n^d)$
  - Solving the splitting and combining costs $\sum_{k=0}^{\log_b n} a^k f(\frac{n}{b^k})$, it is enough to show that it is $O(n^d)$
  - $a^k f(\frac{n}{b^k}) \leq ca^k(\frac{n}{b^k})^{d-\epsilon} = cn^{d-\epsilon}(ab^{\epsilon-d})^k = cn^{d-\epsilon}b^{k\epsilon}$
  - $\therefore \sum_{k=0}^{\log_b n} a^k f(\frac{n}{b^k}) \leq \sum_k cn^{d-\epsilon}b^{k\epsilon} = cn^{d-\epsilon}\sum_k b^{k\epsilon}\ \therefore$ splitting costs is $O(n^d)$
  - Because the base case is $\Theta(n^d)$ and splitting is $O(n^d)$, the overall costs will be $\Theta(n^d)$
- Proof of case 2
  - Solving the base cases costs: $n^d T(1) \in o(n^d \log^{k+1} n)$
  - Need to show that splitting costs is $\Theta(n^d \log^{k+1} n)$
  - $\sum_{m=0}^{\log_b n} a^m f(\frac{n}{b^m}) \in \sum_m a^m \Theta((\frac{n}{b^m})^d \log^k \frac{n}{b^m}) = \sum_m \Theta(n^d \log^k \frac{n}{b^m}) = \Theta(n^d)(\sum_m \log^k \frac{n}{b^m}) = \Theta(n^d \log^{k+1} n)$

3 Proof of Correctness

3.1 iterative algorithms

3.1.1 Loop invariants

- Some desirable conditions that should be satisfied **at the start** of each iteration.
- **Initialization**: Loop invariant is true at the start of the first iteration.
- **Maintenance**: if the loop invariant is satisfied at the start of the current iteration, then the loop invariant must be satisfied at the start of **next iteration**.
- **Termination**: loop invariant at the end of the last iteration $\implies$ the algorithm outputs a correct answer

3.1.2 Examples

- Dijkstra Algorithms: $\forall u \in R, d(u) = dist(s, u)$, R is the set of all relaxed vertex. Assume that each time the vertex released is not correct, prove by contradiction
- Selection Sort: $A[1, j-1]$ is sorted and $\forall x \in A[1, j-1], y \in A[j, n],\ s.t.\ x \leq y$
- Insertion Sort: At the start of each iteration of the for loop of lines 1–8, the subarray A[1 : i – 1] consists of the elements originally in A[1 : i – 1], but in sorted order.

3.2 recursive algorithms

- Base case: show that algorithm is correct in the base case
- Inductive steps:
  - Assume algorithm is correct for any size of smaller than n
  - Show that the algorithm is correct for any input of size n

4 Divide and Conquer

Divide the problem into smaller subproblems. Solve the subproblems recursively. Combine subproblems to get the full problem.

4.1 Quick Exponential & Fibonacci

Quick Exponential:

- if $n \mod 2 \equiv 0$, $a^n = a^{\lfloor \frac{n}{2} \rfloor} a^{\lfloor \frac{n}{2} \rfloor}$
- if $n \mod 2 \equiv 1$, $a^n = a^{\lfloor \frac{n}{2} \rfloor} a^{\lfloor \frac{n}{2} \rfloor} a$

- Calculate $a^{\lfloor n/2 \rfloor}$ in $T(\lfloor n/2 \rfloor)$, then use above steps to get $\Theta(1)$ time.

Quick Fibonacci:

- $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
- $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$
- Apply quick exponential on this for $O(\log(n))$ time.

4.2 Matrix multiplication

For a multiplication of $n \times n$ matrix, we rewrite them into formulae of, each alphabet represent a sub-matrix of dimension $\frac{n}{2} \times \frac{n}{2}$:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

- $P_1 = a \cdot (f - h)$
- $P_2 = (a + b) \cdot h$
- $P_3 = (c + d) \cdot e$
- $P_5 = (a + d) \cdot (e + h)$
- $P_6 = (b - d) \cdot (g + h)$
- $P_7 = (a - c) \cdot (e + f)$
- $r = P_5 + P_4 - P_2 + P_6$
- $s = P_1 + P_2$
- $t = P_3 + P_4$
- $u = P_5 + P_1 - P_1 - P_7$

Perform 7 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices: $7T(\frac{n}{2})$ times. 18 additions of : $\Theta(n^2)$ time. $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$, $T(n) \in \Theta(n^{\log_2 7})$

4.3 Comparison-based sort

$O(n \log n)$ is the best possible bound. Elements can only be compared with each other using $<, \leq, =, >, \geq$.

Theorem. *The worst-case time complexity of any comparison based sorting algorithm is* $\mathbf{\Omega(n \log n)}$.

Definition. *Decision tree is a rooted tree*

- *start from root*
- *at every vertex, ask a question*
- *choose one child depends on the answer*
- *at leaf a decision is make*

- *a comparison ↔ a question asked at a node*
- *program state depends on the result of comparison ↔ chosen child depends on answer to the question*
- *output of the algorithm ↔ decision at a leaf*

Model comparison-based sort as a decision tree, which is a binary tree with at least $n!$ leaves as each permutation is a possible answer. The height of the binary tree is at least $\log(n!) \in \Theta(n \log n)$

4.4 Quicksort analysis

4.4.1 Description

Divide by partitioning (rearranging) the array A[p:r] into two (possibly empty) subarrays $A[p : q-1]$ (the low side) and $A[q+1 : r]$ (the high side) such that each element in the low side of the partition is less than or equal to the pivot $A[q]$. Compute the index q of the pivot as part of this partitioning procedure.

Partition

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)
    end if
end procedure
```

```
procedure PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r − 1 do
        if A[j] ≤ x then
            i = i + 1
            exchange A[i] with A[j]
        end if
    end for
    exchange A[i + 1] with A[r]
end procedure
```

Loop invariants of partition:

- if $p \le k \le i$, $A[k] \le x$
- $i + 1 \le k \le j - 1$, $A[k] > x$
- $A[r] = x$

4.4.2 Performance (Non-randomized)

let pivot be the j th smallest element. We have $T(n) = T(n-j) + T(j-1) + cn$. Worst case is $T(n) = \max_{j\in[n]} (T(n-j) + T(j-1) + cn)$. Proved by substitution $T(n) \leq c_1 n^2$

- all numbers are distinct
- $a_1 < a_2 < a_3 < \cdots < a_n$
- the input array A can be described by a **permutation** π of $(a_1,\ a_2,\ a_3,\ \ldots,\ a_n)$
- Average running time/ expected running time:
  $A(n) = \sum_n \frac{1}{n!}$(running time of quick sort on π)
- the pivot is selected at random, the permutations of recursive calls are also uniformly random

Derivation of $A(n)$:

- $A(n) = \frac{1}{n}\sum_{j=1}^{n}[A(j-1) + A(n-j) + cn] = cn + \frac{2}{n}\sum_{j=0}^{n-1} A(j)$
- $nA(n) = cn^2 + 2 \cdot \sum_{j=0}^{n-1} A(j)$
- $(n-1)A(n-1) = c(n-1)^2 + 2 \cdot \sum_{j=0}^{n-2} A(j)$
- $nA(n) - (n-1)A(n-1) = c(2n-1) + 2A(n-1)$
- $nA(n) - (n+1)A(n-1) = c(2n-1)$
- $\frac{A(n)}{n+1} + \frac{A(n-1)}{n} = \frac{c(2n-1)}{n(n+1)} < \frac{2c}{n}$
  By telescoping methods
  $\frac{A(n)}{n+1} - \frac{A(1)}{2} < 2c \cdot (\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{2}) \implies A(n) \in O(n\log(n))$

Stable sorting: for elements of equal values, the original ordering is preserved. E.g. insertion sort, merge sort (if implemented correctly)

In-place sort: use little extra memory besides the input array

4.5 Non-comparison sort

4.5.1 Counting Sort

```
procedure COUNTING-SORT(A, n, k)
    let B[1 : n] and C[0 : k] be new arrays
    for i = 0 to k do
        C[i] = 0
```

```
    end for
    for j = 1 to n do
        C[A[j]] = C[A[j]] + 1
    end for
    for i = 1 to k do
        C[i] = C[i] + C[i – 1]
    end for
    for j = n downto 1 do
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] – 1
    end forreturn B
end procedure
```

4.5.2 Radix Sort

procedure RADIX-SORT(A, n, d)
  for $i = 1$ to d **do**
    use a stable sort to sort $A[1 : n]$ on digit i
  end for
end procedure

The algorithm takes time $O(m(k + n))$, where $O(k + n)$ is the complexity of stable sorting based on the digits (from Counting Sort above). Here k is the number of possible values of a 'digit'/numeral.

Radix sort can be used to sort n numbers ranging between 1 and n^r , for any fixed positive integer r, in $O(n)$ time. To see this, note that number of bits needed to write n^r is $r \log n$. We can divide these bits into r groups of log n bits each. Consider each of these groups of $\log n$ bits as a digit ranging from 1 to n. Then, in the above algorithm, we have $m = r$ and each digit has $k = n$ possible values. This gives the time complexity as
$O(r(n + n)) = O(n)$, as r is a constant.

5 Randomized Algorithms

5.1 Techniques

Principle of deferred decision: $\forall x,\ Pr[\epsilon|X = x] \geq p \implies Pr[\epsilon] \geq p$,
$Pr[\epsilon] = \sum_x Pr[\epsilon|X = x] \cdot Pr[X = x] \geq p \cdot \sum_x Pr[X = x] = p.$

Union Bound: $\epsilon = \epsilon_1 \vee \epsilon_2 \vee \epsilon_3 \vee \epsilon_4 \vee \cdots \vee \epsilon_n,\ Pr[\epsilon] = Pr[\cup_i \epsilon_i] \leq \sum_i Pr[\epsilon_i]$

Markov Inequality: If X is a non-negative random variable and $a > 0$,
$Pr[X \geq aE[x]] \leq \frac{1}{a}$

Linearity of expectation: $X = A + B, E[X] = E[A] + E[B]$

Indicator random variable: let ϵ be an event, the indicator random variable $\mathbf{1}_\epsilon$ is

defined as

$$\mathbf{1}_\epsilon = \begin{cases} 1, \text{ if } \epsilon \text{ occurs} \\ 0, \text{ otherwise.} \end{cases}$$

$E[\mathbf{1}_\epsilon] = Pr[\epsilon]$

5.2 Freivalds' Algorithms

Steps:

- Purpose: check $AB = C$ efficiently
- Choose a $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$ to be uniformly random column from $\{0,1\}^n$
- Check $ABv = Cv$, done in $O(n^2)$ time by doing three matrix vector multiplication

Analysis:

- if $AB \neq C$, there is a chance that it will output an incorrect answer
- let $C* = \begin{pmatrix} \vec{c_1^*} & \vec{c_2^*} \cdots \vec{c_n^*} \end{pmatrix} = AB - C$
- $u = C * v = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}$
- $u_i = c^*_{i,1}v_1 + c^*_{i,2}v_2 + \cdots + c^*_{i,n}v_n$, Randomly pick a number $c^*_{i,j} \neq 0$ (this is guaranteed to find at least one because $AB \neq C$). Once we fixed the rest of the terms, there is at most one choice of v_j that makes $u_i = \{\cdots\} + c^*_{i,j}v_j = 0$, $Pr[u_i \neq 0] \geq \frac{1}{2}$
- $Pr[\text{Freivald's algorithm is successful}] = Pr[\exists k \in [n], u_k \neq 0]$

5.3 Coupon Collector's Problem/ Balls and bins/ Chain hashing

Problem statement:

- n different types of coupons
- Once obtain all n types of coupons receive aprice
- m boxes to buy to collect n types of coupons
- Throw m balls into n bins randomly and independently. What is the probability that every bin contain at least one ball?

Analysis:

- For a specific bin, the probability that it does not contain any ball is $(1-\frac{1}{n})^m = (1-\frac{1}{n})^{(-n)(-\frac{m}{n})} \leq e^{-\frac{m}{n}}$
- By union bound, the probability that at least one bin contains zero balls is at most $Pr[\cup\{\text{Bin i contains zero bin}\}] \leq n(1-\frac{1}{n})^m \leq ne^{-m/n}$
- The probability at least one Bin contain zero ball is at most $\frac{1}{n}$ if $m \geq 2n[\ln(n)]$
- $\therefore$ Buying $2n[\ln(n)]$ will guaranteed a success probability of $1-\frac{1}{n}$

Alternative:

- How many balls must be thrown to ensure every bin has a ball?
- let n_i be the number of tosses at the ith stage. Each stage count the number of toss from the $(i-1)$st hit to the i th hit.
- $E[n] = E[\sum_{i=0}^{b} n_i] = \sum_{i=1}^{b} \frac{b}{b-i+1} = b\sum_{i=1}^{b} \frac{1}{i} = b\ln b + \Theta(1)$

Extension (hashing with chain):

- Balls: elements to be stored; Bins: hash table

5.4 Randomized Quicksort

Analysis:

- $(a_1, a_2 \cdots a_n)$ = the array A in the sorted order
- $X_{i,j}$ = the number of comparisons between a_i and a_j
- $Z_{ij} = (a_i, \cdots, a_j)$, a_i is compared with a_j if one of them is first selected before all other elements in Z_{ij}
- The elements belongs to Zi, j stay all together for each recursive call until PARTITION chooses some pivot from the set. Each element is equally likely to be chosen at each round of the recursive call,
  $P(pivot = a_i \vee pivot = a_j | pivot \in Z_{ij}) = \frac{2}{j-i+1}$
- For all events where $E = [\text{pivots in } Z_{ij}]$, we have same probability. By deferred decisions, $E[X_{ij}] = Pr[a_i \text{ is compared with } a_j] = \dfrac{2}{j-i+1}$
- $E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log(n)) = O(n\log(n))$

5.5 Birthday Paradox (CLRS)

Problem Statement:

There are k people in the room. How many people must there be in a room for the expected number of pairs that have the same day is larger than 1

Analysis:

- X_{ij} person i and person j have the same birthday
- X be the number of people have same birthday,
  $E[X] = \sum_i \sum_j E[X_{ij}] = \binom{k}{2} \frac{1}{n} = \frac{k(k-1)}{2n}$
- $E[X] \geq 1, k(k-1) \geq 2n$

5.6 Streak (CLRS)

Problem Statement:

Flip a fair coin n times, what is the longest streak of consecutive heads that you expect to see?

Analysis:

1. Upper bound

- let $A_{i,k}$ denotes having **at least** k flips start with i
- $Pr[A_{i,k}] = (\frac{1}{2})^k$
- $k = 2\lceil \log(n) \rceil$, $Pr[A_{i,2\lceil \log n \rceil}] = \frac{1}{2^{2\lceil \log n \rceil}} \leq \frac{1}{n^2}$
- Probability of having a streak of length of at least k flips
  $Pr[\cup_{i=1}^{n-2\lceil \log n \rceil+1} A_{i,2\lceil \log n \rceil}] \leq \sum_{i=1}^{n-2\lceil \log n \rceil+1} Pr[A_{i,2\lceil \log n \rceil}] < \sum_{i=1}^{n} \frac{1}{n^2} = \frac{1}{n}$
- Let L_j be the event that the longest streak of heads has length of exactly j
- $E(L) = \sum_j j Pr[L_j] = \sum_{j=0}^{2\lceil \log n \rceil - 1} j Pr[L_j] + \sum_{2\lceil \log n \rceil}^{n} j Pr[L_j]$
- Apply the bounding summation,
  $E(L) < 2\lceil \log n \rceil \sum_{j=0}^{2\lceil \log n \rceil - 1} Pr[L_j] + n \sum_{2\lceil \log n \rceil}^{n} Pr[L_j] < 2\lceil \log n \rceil + n \cdot \frac{1}{n} = O(\log n)$

2. Lower bound

- $k = \lfloor \log(n)/2 \rfloor$, $Pr[A_{i,\lfloor \log(n)/2 \rfloor}] = \frac{1}{2^{\lfloor \log(n)/2 \rfloor}} \geq \frac{1}{\sqrt{n}}$
- The probability that the streak of heads of length at least $\lfloor \log(n)/2 \rfloor$ does not begin in i is at most $1 - \frac{1}{\sqrt{n}}$
- The probability that none of the streak has size $\lfloor \log(n)/2 \rfloor$ is $(1 - \frac{1}{\sqrt{n}})^{\lfloor n/\lfloor \log(n)/2 \rfloor \rfloor} \leq (1 - \frac{1}{\sqrt{n}})^{2n/\log n - 1} \leq (1 - \frac{1}{\sqrt{n}})^{-\sqrt{n}\frac{2n/\log n - 1}{-\sqrt{n}}} \leq e^{\frac{2n/\log n - 1}{-\sqrt{n}}} = O(e^{-\ln n}) = O(\frac{1}{n}) \because \frac{2n \log n - 1}{\sqrt{n}} \geq \ln n$ for sufficiently large n
- let P be the event that the longest streak is not smaller than $\lfloor \log n/2 \rfloor$, $Pr[P] = 1 - \Pr[\neg L] \geq 1 - Pr[P] = 1 - O(1/n)$
- $\therefore \sum_{j=\lfloor \log n/2 \rfloor}^{n} L_j \geq 1 - O(1/n)$, subst this back to the formulae we have $E[L] \geq 0 \cdot \sum_{j=0}^{\lfloor \log n/2 \rfloor - 1} Pr[L_j] + \lfloor \log n/2 \rfloor \sum_{j=\lfloor \log n/2 \rfloor}^{n} Pr[L_j] \geq 0 + \lfloor \log n/2 \rfloor (1 - O(1/n)) = \Omega(\log n)$

5.7 Online Hiring Problems (CLRS)

Problem Statement:

Interview with n candidates. After interview must determine whether to offer a position or not. Following algorithm is used: select a number k such that record the highest in the first k and reject all. Hire the first candidate that has a higher score than that number. And if none of them meet the requirement, choose the last one. What is a suitable k?

Analysis:

- let S_i be the event that best candidate occurs at the ith position and we successfully selected the best one
- let S be the event that the best is chosen. $Pr[S] = \sum_{i=k+1}^{n} S_i$
- B_i be the event that the best is in ith position, O_i be the case where all candidates from $k+1$ to $j-1$ must be smaller than best score in the first k i.e. the max value in the first $i-1$ position will appear in the first k positions. $Pr[O_i] = \frac{k(i-2)!}{(i-1)!} = \frac{k}{i-1}$
- $Pr[B_i \cap O_i] = Pr[B_i]Pr[O_i] = \frac{1}{n} \cdot \frac{k}{i-1}$, $\therefore Pr[S] = \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}$
- $\int_k^n \frac{1}{i} di \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{i} di \therefore Pr[S] \in [\frac{k}{n} \ln \frac{n}{k}, \frac{k}{n} \ln \frac{n-1}{k-1}]$

5.8 Number of inversions (Interview)

Problem Statement:

What is the expected number of inversions in an array?

Analysis:

- Let X_{ij} be the r.v. denotes that a_i and a_j are inversions
- $Pr[X_{ik} = 1] = \frac{1}{2}$
- $E(X) = \sum_i \sum_j X_{ij} = \frac{n(n-1)}{4}$

6 Dynamic Programming

6.1 Algorithm Paradigm

Optimal substructure: an optimal solution to a problem contains optimal solutions to subproblems

Overlapping subproblems: a recursive solution contains a small number of distinct subproblems repeated many times

- Express the solution recursively
- There are only polynomial number of subproblems, but there is a huge overlap among the subproblems. So recursive algorithm takes exponential time
- Bottom-up (table update of DAG) and top-down (memoization) approach

6.2 Longest Common Subsequence

Subsequence: C is a subsequence of A if there exist k integers $1 \leq i_1 \leq i_2 \cdots \leq i_k \leq n$ such that for all $1 \leq j \leq k$: $C[j] = A[i_j]$

Problem Statement: Two sequences $A[1 \dots n]$ and $B[1 \dots m]$, find the longest sequence C such that C is a subsequence of both A and B

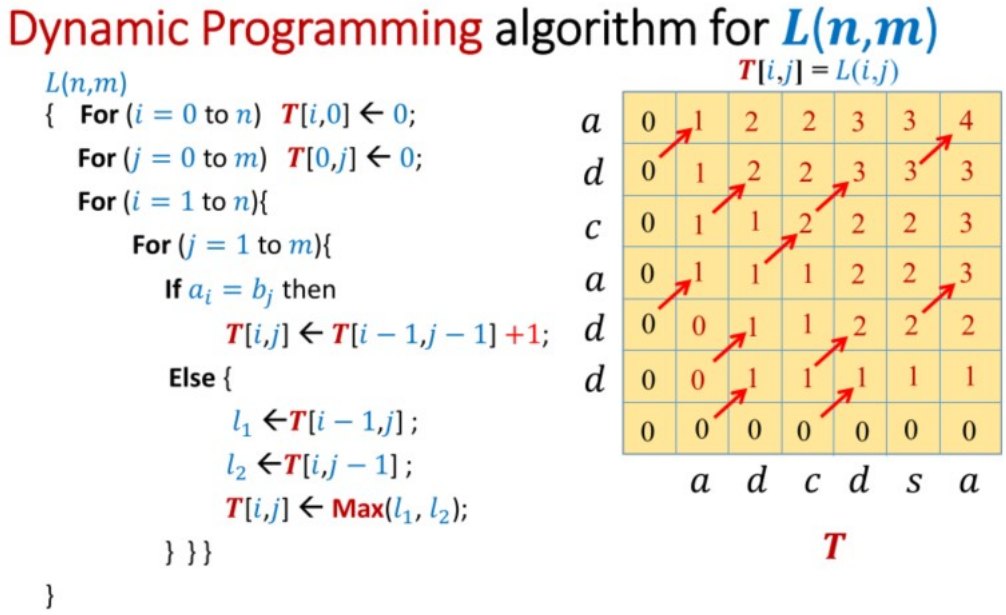
DP formulation

- Let $LCS(i, j)$ denotes the longest common subsequence of $A[1 \dots i]$ and $B[1 \dots j]$
- **Base case**: $LCS(i, 0) = LCS(j, 0) = \epsilon$
- if $A[i] = B[j]$, $LCS(i, j) = LCS(i-1, j-1) :: A[i]$. How to prove? cut-and-paste, assume not include $A[i]$ and show contradiction

- if $A[i] \neq B[j]$, $LCS(i-1, j)$ OR $LCS(i, j-1)$

This can be simplified to ask about the length of LCS.

Solve LCS in a bottom up fashion by updating dp table of size mn.



6.3 Knapsack Problem

Problem Statement:

Input: (w_1, v_1), $(w_2, v_2), \ldots, (w_n, v_n)$ and W

Output: a subset $S \subseteq \{1, 2, \ldots, n\}$ that **maximizes** $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$

Interesting Note:

This is an optimization problem. The corresponding decision problem of 0/1 Knapsack can be reduced from the Partition Problem: $S = \{s_1, \ldots, s_n\}$, $(w_i, v_i) = (s_i, s_i)$ and capacity W = $\sum s_i/2$ and the query is value $\geq \sum s_i/2$

Recurrence relation:

$dp[i][w]$ denotes the maximum value can be obtained using subset of items in $[1, \ldots, n]$ and total weight of not more than w.

Base case:

$$dp[i][j] = 0, i = 0, j = 0$$

Recurrence Relation:

$$dp[i][j] = max(dp[i-1][j-w_i] + v_i, dp[i-1][j]),\ w_i \leq j$$

$$dp[i][j] = dp[i-1][j],\ otherwise$$

6.4 Misc. Examples from LeetCode

6.4.1 Minimum Score triangle

You have a convex n-sided polygon where each vertex has an integer value. You are given an integer array values where $values[i]$ is the value of the ith vertex in clockwise order.

Polygon triangulation is a process where you divide a polygon into a set of triangles and the vertices of each triangle must also be vertices of the original polygon. Note that no other shapes other than triangles are allowed in the division. This process will result in $n-2$ triangles.

You will triangulate the polygon. For each triangle, the weight of that triangle is the product of the values at its vertices. The total score of the triangulation is the sum of these weights over all $n-2$ triangles.

Recurrence Relation:

let $dp[i][j]$ be the score from vertices i to j. from i to $i+1$ to $i+2$ to all the ways to j

all nodes between j to i like $j+1$, $j+2$, $j+3\ldots$ will be ignored

$$dp[i][j] = min(dp[i][k] + values[i] \times values[j] \times values[k] + dp[k][j])$$

Base case: for k in $[i+1, k-1]$

$$dp[i][i+2] = values[i] \times values[i+1] \times values[i+2]$$

6.4.2 Perfect Squares

Given an integer n, return the least number of perfect square numbers that sum to n. A perfect square is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

Recurrence Relation:

For each number i, can have at most i values by summing up i 1s

let $dp[k]$ denotes the minimum number of perfect squares needed to add up to k

$$dp[k] = min(dp[k - a^2] + 1 \mid \forall a, a^2 <= k)$$

Base case:

$$dp[0] = dp[1] = 1$$

6.4.3 Combination Sum

Given an array of distinct integers nums and a target integer target, return the number of possible combinations that add up to target.

Let $dp[t]$ denotes the number of ways to sum up to values of t

Base case:

$$dp[0] = 1, dp[i] = 0, i < 0$$

Recurrence Relation

$$dp[k] = \sum_i (dp[k - nums[i]])$$

6.4.4 Partition Equal Subset Sum

Given an integer array $nums$, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.
(pseudo-polynomial → NP-Complete)

let $dp[i][val]$ denotes the fact that we can reach val by choosing elements from $arr[: i]$
$1 \leq i \leq n$, $0 \leq val \leq target = \frac{\sum_i s_i}{2}$

Base case:

$$dp[i][0] = 1, dp[0][j] = 0 \ \forall \ j \neq 0$$

Recurrence relation:

$$dp[i][val] = dp[i-1][val], val < nums[i]$$

$$dp[i][val] = dp[i-1][val] \vee dp[i-1][val - nums[i]], val > nums[i]$$

6.4.5 Partition to K Equal Sum Subsets

Given an integer array nums and an integer k, return true if it is possible to divide this array into k non-empty subsets whose sums are all equal.

The length of the array is small. Use bitmask to represent state of being selected. let $dp[mask][rem] = T/F$ denote given which elements are used (mask), and current bucket sum (rem), can we still complete valid partitions. $FULL = (1 << n) - 1$, i.e. all elements are selected,

Base case:

$$dp[FULL][0] = T$$

Recurrence Relation:

if $rem + nums[i] < target$,

$$dp[mask][rem] \vee = dp[mask|(1 << i)][rem + nums[i]]$$

if $rem + nums[i] == target$,

$$dp[mask][rem] \vee = dp[mask|(1 << i)][0]$$

if $rem + nums[i] > target$

$$dp[mask][rem] \vee = False$$

6.4.6 Maximum Score from Performing Multiplication Ops

You are given two 0-indexed integer arrays nums and multipliers of size n and m respectively, where n >= m.

You begin with a score of 0. You want to perform exactly m operations. On the ith operation (0-indexed) you will:

- Choose one integer x from either the start or the end of the array nums.
- Add multipliers[i] * x to your score.
- Note that multipliers[0] corresponds to the first operation, multipliers[1] to the second operation, and so on.
- Remove x from nums.
- Return the maximum score after performing m operations.

$dp(i, j, k)$ denote max values use subarray nums[i..j] to perform the k th operation

Recurrence Relation:

$dp(i, j, k) = max(dp(i+1, j, k+1) + mult[k] * nums[i], dp(i, j-1, k+1) + mult[k] * nums[j])$

Base case:

$$i < j, return\ 0$$

$$k \geq m, return\ 0$$

Observation: $j = n - 1 - (k - i) \implies$ when reaches i, j, k, first i elements are picked + elements from j to the end is picked, total number of picks is n

Optimized dp:

$$dp(i,k) = \max\Big(multipliers[k] \cdot nums[i] + dp(i+1, k+1),$$
$$multipliers[k] \cdot nums[j] + dp(i, k+1)\Big)$$

where $j = n - 1 - (k - i)$

Base case:

$$dp[i, m] = T$$

7 Greedy Algorithm

7.1 General strategy

- Cast the optimization problem as one in which you make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe (**greedy choice property**).
- Demonstrate **optimal substructure** (if an optimal solution to the problem contains within it optimal solutions to subproblems) by showing that using , having made the greedy choice, what remains is a subproblem with the property that if you combine an optimal solution to the subproblem with the greedy choice you have made, you arrive at an optimal solution to the original problem.
- Proof of greedy choice property and optimal substructure should use cut-and-paste

7.2 Example

7.2.1 Fractional knapsack

Input:
$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and W

Output:
Integer weights $x_1, \ldots, x_n$ that maximize $\sum_i v_i \cdot \frac{x_i}{w_i}$ subject to:
$\sum_i x_i \leq W$ and $0 \leq x_j \leq w_j$ for all $j \in \{1,2,\ldots,n\}$.

Proof of optimal substructure: If removes w of one item j from the optimal knapsack solution, the remaining of the solution should also be the optimal solution to the subproblem with weight at most $W - w$ that one can take from $n - 1$ original items and $w_j - w$ of item j. This can be shown by **cut and paste**. Assume the optimum value is X, If there is a knapsack $> X - \frac{v_j}{w} w_j$, we can combine the $w$kg of j items with this new knapsack and has a value $> X$ and weight at most W. This is a contradiction.

Proof of greedy choice property: Let j^* be an item with the maximum value/kg, v_j/w_j. Then, there exists an optimal knapsack containing $min(w_{j^*}, W)$ kgs of item j^*. For any optimum solution does not contain $min(w_{j^*}, W)$ of item j^*, we can always decrease 1 kg of any other item and add 1 kg of j^*. Obviously, the total value will increase but the total weight is still within the limit.

7.2.2 MST - Prim's Algorithm

Optimal substructure: Remove $(u, v) \in MST(G)$, T is partition into two subtrees T_1 and T_2 and each of them are the MST of relevant subgraph.

Greedy choice (cut-property): $G = (V, E)$, $A \subseteq V$, $\forall\ (u, v) \in E$, the least-weight edge connecting A to $V - A$ is in the MST. Proof by contradiction that if it is not, there will always be a way to find a tree with smaller wright which contradicts with definition of MST.

7.2.3 Huffman code

Alphabet set A: $\{a_1, a_2, \ldots, a_n\}$. Encode each alphabet to a unique binary string of some various length. More frequent alphabets should have coding with shorter bit string. Less

frequent alphabets should have coding with longer bit string. Average bit length:

$$ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$$

.

Prefix coding: $\gamma(A)$ is prefix coding iff there do not exist $x, y \in A$, $\gamma(x)$ is prefix of $\gamma(y)$. For each **prefix code** of a set A of n alphabets, there exists a **binary tree** T on n leaves such that: (1) there is a bijective mapping between the **alphabets** and the **leaves**; (2) the label of a path from root to leaf corresponding to the prefix code of the alphabet.

Problem statement: find the labeled binary tree for an optimal prefix codes. The binary tree corresponding must be a **full binary tree**. Let $\{a_1, a_2, \ldots, a_n\}$ be the alphabets in **non-decreasing** order of **frequencies**.

Solution: More frequent alphabets should be closer to the root and less frequent one should be farther from the root. There exists an optimal prefix coding in which a_1 and a_2 appear as siblings in the corresponding labeled binary tree.

Optimal substructure: Any coding for $\{a_3, \ldots, a_{1,2}\}$ can be converted to a coding $\{a_1, a_2, \ldots, a_n\}$ with ABL increases by $f(a_1) + f(a_2)$ (add one more layer).

Algorithm

- Huffman-Code($\{a_1, \ldots, a_n\}$):
  - If $n = 1$, output a single node.
  - Find two alphabets a_1, a_2 with least frequency.
  - Replace a_1, a_2 with $a_{1,2}$ with frequency $f(a_{1,2}) = f(a_1) + f(a_2)$.
  - Run Huffman-Code($\{a_3, \ldots, a_n, a_{1,2}\}$).
  - Replace the node $a_{1,2}$ with two children a_1 and a_2.

7.3 More Examples

7.3.1 Jump Game 2

You are given a 0-indexed array of integers $nums$ of length n. You are initially positioned at index 0. Each element $nums[i]$ represents the maximum length of a forward jump from index i. In other words, if you are at index i, you can jump to any index $(i + j)$ where: $0 \leq j \leq nums[i]$ and $i + j < n$. Return the minimum number of jumps to reach index $n - 1$. The test cases are generated such that you can reach index $n - 1$.

$dp(i)$ denotes the minimum number of jump from i to the end

$$dp(i) = min\{1 + dp(k), k \in [i+1, i+nums[i]]\}$$

greedy choice: each time i choose the k, with $max\{k + nums[k]\}$

Proof greedy choice:

if k is not chosen in the optimum, there exists another $j \neq k$ such that $1 + dp(j)$ is the optimum solution to the subproblem, $j \in [i+1, i+nums[i]]$

if $j > k$, the next step of starting from j is definitely reachable if choose to start from k because $k + nums[k]$ is the furthest can be reached

if $j < k$, if $j + nums[j] < k$, this is not the best step cause u can always reach $j + nums[j]$ in one step instead of visiting k. In other case, the next step of starting from j is definitely reachable if we choose to start from k because of the same above mentioned reasons

7.3.2 IPO

Suppose LeetCode will start its IPO soon. In order to sell a good price of its shares to Venture Capital, LeetCode would like to work on some projects to increase its capital before the IPO. Since it has limited resources, it can only finish at most k distinct projects before the IPO. Help LeetCode design the best way to maximize its total capital after finishing at most k distinct projects.

You are given n projects where the i^{th} project has a pure profit $profits[i]$ and a minimum capital of $capital[i]$ is needed to start it. Initially, you have w capital. When you finish a project, you will obtain its pure profit and the profit will be added to your total capital. Pick a list of at most k distinct projects from given projects to maximize your final capital, and return the final maximized capital. The answer is guaranteed to fit in a 32-bit signed integer.

let $dp(S, k, w)$ denotes the maximum capital from picking k projects from S with current capital of w

$$dp(S, k, w) = max\{dp(S - i, k - 1, w + profit[i]) \mid weight[i] < w\}$$

$$dp(S, k, w) = w, \ k = 0 \ \vee \ S \equiv \phi$$

Greedy options: choose the i such that $profit[i]$ is the max

Proof of greedy choice:

if $k = argmax_k(profit[k])$, $j = argmax_j\{dp(S-j, k-1, w+profit[j]) \mid weight[j] < w\}$

if $k == j$, our greedy option is true

if $k \neq j$, let's assume the set of events choosing for j to happen is M_j including j, we know the final capital will be $= sum\{profit[e], e \in M_j\}$

if $k \in M_j$, we choose we can choose k first then the $w + profit[k] > w + profit[j]$ and hence we should be able to select all elements in $M_j - \{k\}$

if $k \notin M_j$, replacing k for j will cause all the rest elements $M_j - \{j\}$ still possible to be chosen. This is because the w is always monotonically increasing and $w + profit[k] > w + profit[j]$ which is the new available capital for selection from the remaining. In addition, the replacement will cause total capital increase by $profit[k] - proft[j]$

7.3.3 Non-overlapping intervals

Given an array of intervals intervals where $intervals[i] = [starti, endi]$, return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping. Note that intervals which only touch at a point are non-overlapping. For example, $[1, 2]$ and $[2, 3]$ are non-overlapping.

7.3.4 Most Profit Assigning Work

You have n jobs and m workers. You are given three arrays: difficulty, profit, and worker where: $difficulty[i]$ and $profit[i]$ are the difficulty and the profit of the i^{th} job, and $worker[j]$ is the ability of j^{th} worker (i.e., the j^{th} worker can only complete a job with difficulty at most $worker[j]$). Every worker can be assigned at most one job, but one job can be completed multiple times.

For example, if three workers attempt the same job that pays 1, then the total profit will be 3. If a worker cannot complete any job, their profit is 0. Return the maximum profit we can achieve after assigning the workers to the jobs.

Solution

- sort workers in ascending $ability$
- sort work in ascending $(difficulty, profit)$
- for worker i, find the work that has the max profit while $difficulty \leq ability$
- start from the previously chosen work and for next worker, i.e. each work should only be traversed once

8 Amortized Analysis

Average the time required to perform a sequence of data-structure operations over all operations performed.

8.1 Aggregate method

a sequence of n operations takes $T(n)$ worst case time in total. The amortized cost will be $\frac{T(n)}{n}$

8.2 Accounting methods

Charge each operation more or less than they actually cost. The amount charge on an operation is amortized costs. If amortized costs exceeds actual cost, assign the difference to **credit** to offset future expensive operations. $Credit = \sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$

8.2.1 Example 1: Queue

Two operations: 1) INSERT 2) EMPTY

- An EMPTY is a sequence of DELETE's where each DELETE removes one element from the front of the queue
- Notice: #DELETE's $\leq$ #INSERT's
- If there are k INSERT's in the sequence, sum of cost of all the EMPTY's is $\leq k$.
- Total cost: $\leq k + k = 2k \leq 2n$. Amortized cost is $O(1)$.

Amortized Analysis of Queues

- For INSERT, set amortized cost to 2. (True cost is 1.)
- For EMPTY, set amortized cost to 0. (True cost is size of queue.)
- Whenever an element is inserted, we pay an extra 1. This extra 1 can be used as credit that can pay for deleting it later.
- Total cost is at most 2*#INSERT $\leq 2n$.

8.2.2 Example 2: Binary Counter

Observation: the most expensive flip happens when need to flip the last n bits from 0 to 1 and the $n+1$ bit from 0 to 1

Accounting Methods

- Charge 2 for each $0 \to 1$
- Every 1 bit in the counter will has 1 credits on the bank
- The 1 credit will be pay to reset 1 to 0
- invariant: bank balance will never drops below 0

8.3 Potential Method

The **potential method** represents the prepaid work as potential energy which can be related to future operations. The potential applies to the data structure as a whole. The **potential function** maps D_i to $\Phi(D_i)$. The amortized costs will be
$c'_i = c_i + \Phi(i) - \Phi(i-1)$. $\Phi(0) = 0, \Phi(i) \geq 0 \ \forall \ i$. By Telescoping methods
$\sum c'_i = \sum c_i + \Phi(n)$

The heuristic to find Φ is as follow: choose Φ which for an expensive i-th operation, $\Delta\phi$ is negative to such extent that it reduces the effect of actual expensive costs.

8.3.1 Example: Binary Counter (revisited)

There is only one operation: increment,
and sometimes it is costly (flips many bits).
(although usually it does not (flips only a few bits)).
Is there anything that is decreasing?
Hint: Observe the 1s...

```
1011111
 v|||||
1100000
```

Answer: The number of 1s decrease.

So, we set $\phi(i)$ to be the number of 1s in the counter after the i-th increment.

The actual cost of the i-th increment $= l_i + 1$.
where l_i is the length of the longest suffix with all 1s ($1 \to 0$).
the $+1$ is because we set just one $0 \to 1$ in this case.

The $\Delta\phi(i) = -l_i + 1$,
because $\phi(i) = x - l_i + 1$ (l_i bits $1 \to 0$ and 1 bit $0 \to 1$),
and $\phi(i-1) = x$,
where x is the number of 1s after $(i-1)$ increments.

Therefore:

Actual cost	$\Delta\phi(i)$	Amortized cost
$l_i + 1$	$-l_i + 1$	$(l_i + 1) + (-l_i + 1) = 2 \in O(1)$

Thus, the amortized cost of each increment $= 2 \in O(1)$,
so we have shown that the actual cost of each increment $\in O(1)$.

8.4 More Complex Examples

8.4.1 Example 1: Dynamic table

Problem Statement: table that supports append operation. When the table is full, the next append will trigger table-doubling process and copying the existing content to the new table before append.

Aggregate Method

- It will take $O(1)$ time for normal append operation, heavy operation will only happens when $n-1$ is power of 2 $\because$ table size grows in $1,\ 2,\ 4,\ \ldots, n$
- $t(i)$ denotes time at the i^{th} iteration. Let $t_1(i)$ be the actual cost of appending and $t_2(i)$ is the costs of expanding
- $T(n) = \sum_{i=1}^{n} t_1(i) + t_2(i) = \sum_{j=0}^{log(n-1)} 2^j + \sum_{i=1}^{n} 1 \leq 3n$
- Amortized cost $= \frac{T(n)}{n} = O(1)$

Potential Method

Formulation of potential method

- Anything decreased during the case during expansion? $-size(T)$. But this is not the appropriate potential as $\Phi(i) \geq 0$
- given the expanding properties, we have $size(T)/2 \leq i \leq size(T)$
- let $\Phi(i) = 2i - size(T)$
- To find the $\Delta\Phi$ when the operation is expensive,
  $\Phi(i-1) = i-1, \Phi(i) = 2, \Delta\Phi = 3-i$, actual cost $= i$, amorized cost $= 3$
- Similarly, when table is not full, the cost is also 3

8.4.2 Example 2: Union Find

Quick-find

- Maintain an array id[1..n].
- Objects in the same set have the same id.

- Union(p,q): For all objects r such that id[r] = id[p], set id[r] = id[q].
- Find(p,q): Check whether id[p] = id[q].

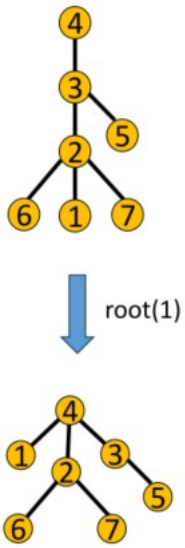
Object	1	2	3	4	5	6
Id[i]	1	2	3	4	5	6

Path Compression

- Problem: the depth of the objects could be very large.

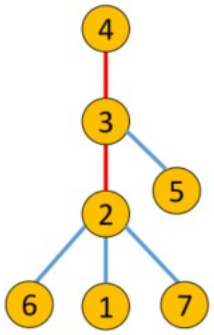
- Path Compression: after calling root(i), change id of every object on the path to the root.

- Idea: Although the depth of the tree might still be very large, but after compression, the tree will be flattened, making future operation cost small.
  - Small amortized cost.
  - We will show that the amortized cost is O(log n).



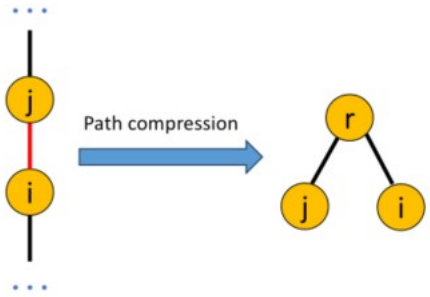
Analysis

- Let size(i) be the number of objects in the subtree of i.
- Suppose the parent of i is j,
  - If size(i) ≤ size(j)/2, then we say the edge (i,j) is blue.
  - If size(i) > size(j)/2, then we say the edge (i,j) is red.
- Observation: On the path from a node to the root, there are at most log n blue edges.



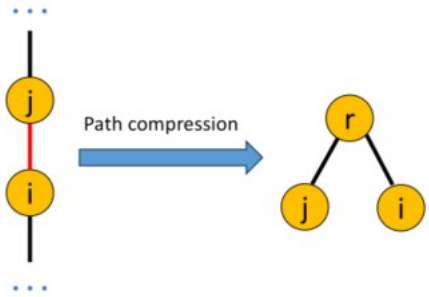
Analysis

- Observation: On the path from a node to the root, there are at most log n blue edges.
- Depth = O(log n) + #red edges.
- Observation: If j is not a root, after path compression, size(j) is decreased by size(i).
  - If (i,j) is a red edge, then size(j) is decreased by at least half.



Potential Method

- Let $\phi_i = \log(size(i))$, and let $\phi = \sum_i \phi_i$.
- Observation: If j is not a root, and (i,j) is a red edge, then size(j) is decreased by at least half.
  - ϕ_j is decreased by at least 1.
- Claim: If we compress a path that contains α red edges, then ϕ is decreased by at least $\alpha - 1$.



Analyze the Costs

- Calling root(i):
  - Cost = O(depth(i))
  - Suppose on the path from i to the root, there are α red edges.
  - depth(i) = O(log n) + α.
  - ϕ is decreased by at least $\alpha - 1$.
- Calling Union(p,q): set id[root(p)] = root(q).
  - 2 calls for root().
  - When we change id[root(p)] to root(q), ϕ is increased by at most log n.
- Calling Find(p,q): check whether root(p) = root(q).
  - 2 calls for root().
- During an operation (Union or Find), let ϕ^+ be the amount of ϕ increases due to Union, ϕ^- be the amount of ϕ decreases due to path compression,
  - $Cost = O(\log n + \phi^-)$
  - $\phi^+ = O(\log n)$

Amortized Cost

- Suppose there are T operations in total.
- For $0 \leq t \leq T$, let $\phi(t)$ be the potential at time t.
  - Let $\phi^+(t)$ be the amount of potential increases due to Union during operation t.
  - Let $\phi^-(t)$ be the amount of potential decreases due to path compression during operation t.
- We have:
  - $Cost(t) = O(\log n + \phi^-(t))$
  - $\phi^+(t) = O(\log n)$
  - $\phi(t) = \phi(t-1) + \phi^+(t) - \phi^-(t)$
  - $\phi(0) = 0, \phi(t) \geq 0$
  - $\sum_{t=1}^{T} \phi^+(t) \geq \sum_{t=1}^{T} \phi^-(t)$
- Putting these all together:
  - $\sum_{t=1}^{T} Cost(t) = O(T \log n) + O(\sum_{t=1}^{T} \phi^-(t)) \leq O(T \log n) + O(\sum_{t=1}^{T} \phi^+(t)) = O(T \log n)$

9 Reduction and intractability

9.1 Reduction

Given two problems A and B, A can be solved as follows

Input: **instance** α of A

- Convert α into an instance β of B
- Solve β and obtain a solution $B(\beta)$ for β
- Based on solution $B(\beta)$ of β, obtain solution $A(\alpha)$ of α

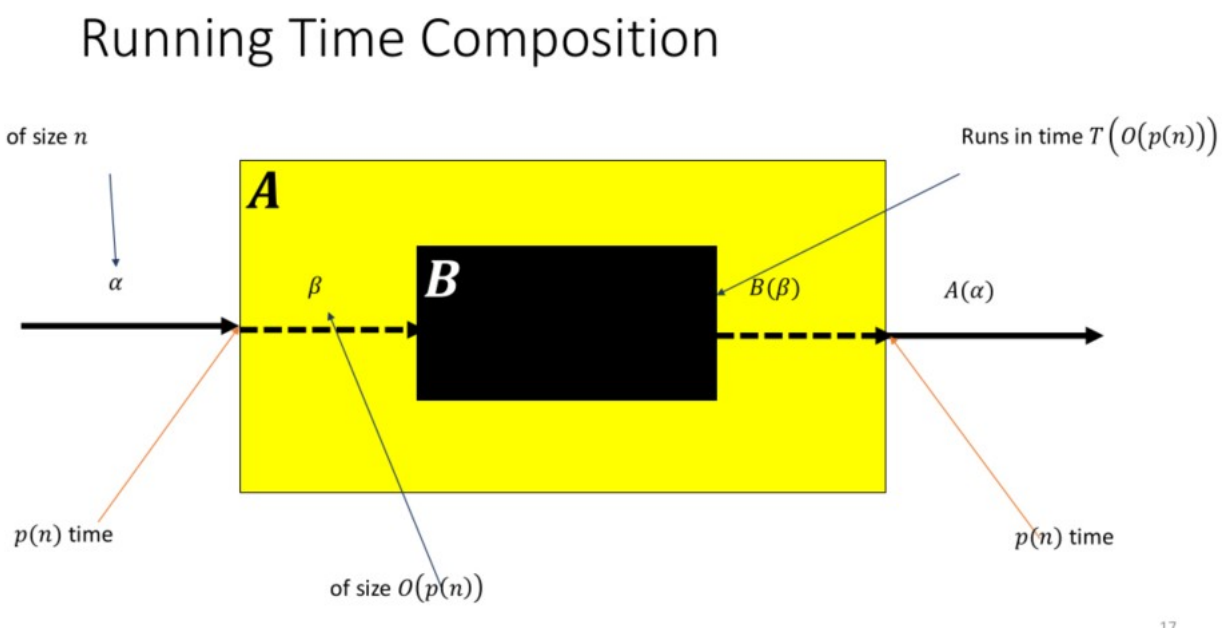
Output: A solution $A(\alpha)$ for α.

This is called: A **reduces to** B.

9.1.1 Polynomial-Time Reduction

If for any instance α of problem A of size n:

- An instance β for problem B can be constructed in $p(n)$ time
- A solution to problem A for instance α can be recovered from a solution to B for instance β in time $p(n) = O(n^c)$, for some $c \in Z$
- $A \leq_p B \Leftrightarrow p(n) = O(n^c),\ c \in Z$ reduction from A to B



Lemmas:

- If B has "easily solvable", then A also have
- If A is "hard", then so is B

9.1.2 Polynomial time and encoding

- **Encoding** of a set S of abstract objects is a mapping of e from S to the set of binary strings.
- For polynomial time, the runtime is **polynomial** in the **length of the encoding of problem instance**.
- Standard encoding: (1) Binary encoding of integers; (2) mathematical objects (graphs, matrices, lists).
- For some problems, there is **flexibility** in selecting the encoding
- An algorithm that runs in time polynomial in the numeric value of the input but is exponential in the length of the input is called a pseudo-polynomial time algorithm, E.g. Fibonacci algorithm runs at $O(n)$

9.2 Decision Problem

A **decision problem** is a function that maps an instance space I (the set of all binary strings that are valid coding) to the solution space YES, NO.

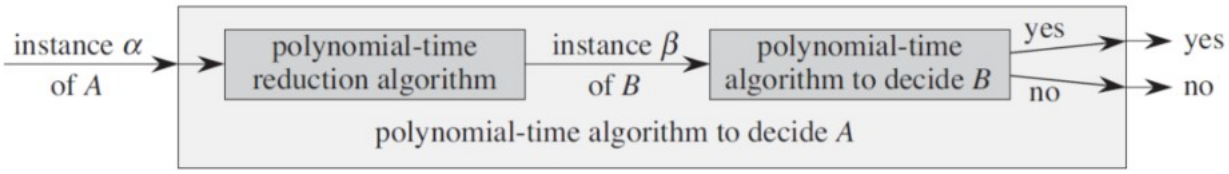
Decision problem and optimization problem:

- Optimization can be **converted** into a decision problem: given an instance of **optimization** problem and a number k, **decide** if exists a solution whose value $\leq k$. E.g. the minimum spanning tree with weight $\leq k$, longest common subsequence with length $> k$
- Decision **reduces** to optimization: Given the value of the optimal solution, just check that it is $\leq k$ ($> k$ depends on question context) $\implies$ Decision problem $\leq_p$ Optimization problem
- Optimization **reduces** to decision: binary search for the value of the optimal solution.
- decision can be solved in polynomial time **iff** the optimization problem can be solved in polynomial time

Reduction between decision problems (reduction):

Given two decision problems A and B, a **polynomial-time reduction** from A to B, denoted $A \leq_p B$, is a transformation from instances α of A to instances β of B such that:

- α is a YES-instance of A **iff** β is a YES-instance for B
- The transformation takes polynomial time in size of α



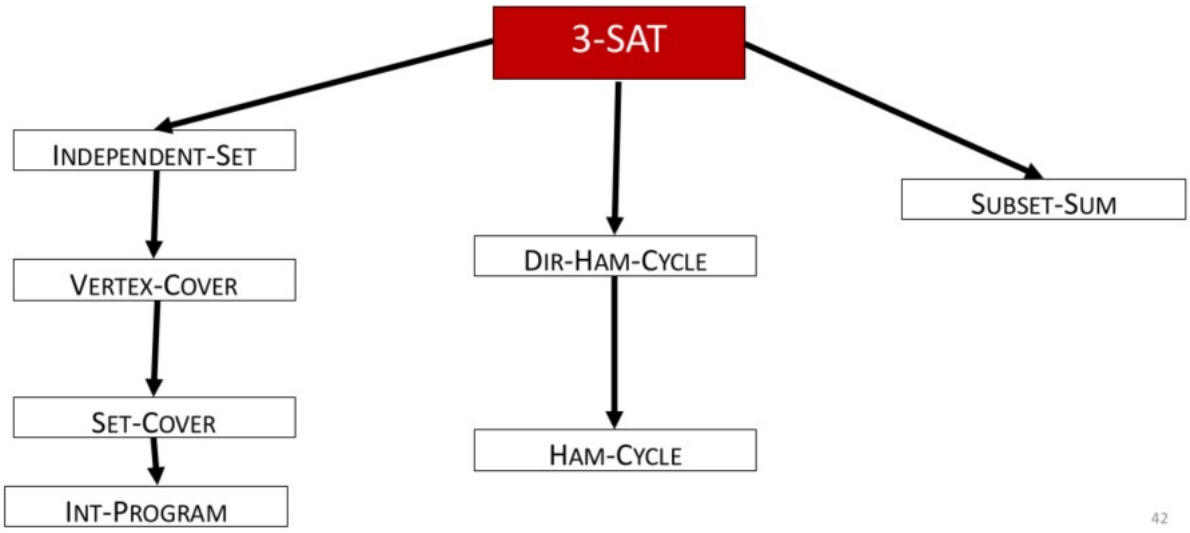
Suffices to show:

- Reduction runs in polynomial time
- If α is a YES-instance of A, β is a YES-instance of B
- If β is a YES-instance of B, α is a YES-instance of A

why we need this prove?

two-way relationship, prove equivalence between alpha and beta are equivalent

9.3 Examples



9.3.1 Independent-Set $\leq_p$ Vertex-Cover

Independent-Set: Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or more) vertices such that no two are adjacent

Vertex cover: Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset

Reduction: G has independent set with size $k \implies$ check G has a vertex cover of size $n - k$, n is the number of vertices G

Proof:

- $\Rightarrow$: (G, k) is a YES-instance of Independent-Set, subset S of V with $\geq k$ vertices that is independent set. $V - S$ is a vertex cover of size $\leq n - k$.
  $\because \forall (u, v) \in E,\ u \notin S \text{ or } v \notin S$, one of them must in $V - S$
- $\Leftarrow$: Suppose $(G, n - k)$ is a YES-instance of Vertex-Cover, $V - S$ is an independent set of size k. $\exists (u, v) \in E$, if both u and v in $V - S$, then S is not the vertex cover. Hence, no edge should be between any pairs of u and v in $V - S$ (Proof by contradiction)

9.3.2 Vertex-Cover $\leq_p$ Set-Cover

Set-Cover: Given integers k and n, and a collection S of subsets of $\{1, \ldots, n\}$, are there $\leq k$ of these subsets whose union equals $\{1, \ldots, n\}$?

Reduction: Given (G, k) for a instance of Vertex-Cover, generate a instance (n, k', S) for

Set-Cover. Set $n = |E(G)|$, $k' = k$, $S = \{S_v | S_v = \{i | e_i \text{ incident on } v\}, v \in V(G)\} \rightarrow$ Clearly reduction is polynomial time

Proof:

- $\Rightarrow$ let V' be the set cover of size $\leq k$ and $V' \subseteq V_G$, $\forall e_i = (u, v) \in E(G)$, either u or v will be in V_G, i.e. $i \in S_v \vee i \in S_u$. Therefore, the union of all member of the collection of sets that represents node in V' i.e. $S' = \{S_v | v \in V'\}$ will contains all edges in $E(G)$
- $\Leftarrow (n, k, S)$ being a YES-instance, the set of vertex corresponding to sets $S_{v_1}, \ldots, S_{v_k}$, $v_1, \ldots v_t$ form a vertex cover in G

9.3.3 3-SAT $\leq_p$ Independent-Set

3-SAT:

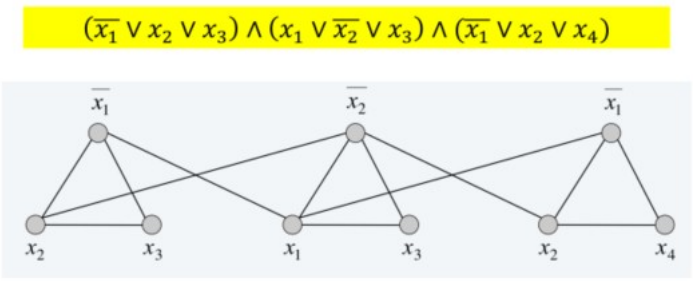
SAT definition:

- **Literal**: boolean variable and negation
- **Clause**: disjunction of literals
- **Conjunctive Normal Form (CNF)**: a conjunction of clauses
- SAT: given a CNF formulae Φ, does it have a satisfying assignment?

$3 - SAT$ definition: SAT where each clause contains exactly 3 literals. YES-instance if f it admits at least one assignment

Reduction:

General idea: construct an INSTANCE (G, k) for Independent-Set and G has an Independent Set of size k based on CNF Φ



- 3 vertices of each clause, one for each literal in the clause
- Connect 3 vertices for each clause in a triangle
- Connect literal to all of its negation

Proof:

- ($\Leftarrow$) (G, k) is a YES-instance. Let S be size of independent set of size k. Each k triangle must contain exactly one vertices. Set literal to True and clause will satisfied
- ($\Leftarrow$) Φ is an YES-instance, select a true literal from each of the clause, the k vertices will eventually form an independent set of size k

9.3.4 3-SAT $\leq_p$ Vertex Cover (Sisper)

Reduction:

General idea: construct an instance (G, k) for Vertex Cover, and G with $2m + 3l$ vertices will have a vertex cover of $m + 2l$ where Φ has m variables and l clauses.

- **Variable gadget**: for each variable $x \in \Phi$, produce two nodes represent itself and its negation, join them with vertices
- **Clause gadget**: Each clause is a triple of nodes labeled with the literals
- the nodes in clause gadget are to connected to the corresponding node in the variable gadget that has the same literal

Proof:

- ($\Rightarrow$) pick the literal that is True from each clause, picked all other node that is connected to it, there will be $2l$ nodes from each clause and m nodes from each gadget. All edges are incident on one *node*
- ($\Leftarrow$) If there is such a vertex cover, each variable gadget will have at least one node being selected. Just treat that one to be True literal. and in each clause in Phi it contains at least one of them to be true.

9.3.5 3-SAT $\leq_p$ CLIQUE (CLRS)

Reduction:

- $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge \cdots \wedge C_k$
- Each clause, created three nodes for each of the literal.
- Connect v_i and v_j if they are not negation to each other and they are not in the same clause.
- Set k to 3

Proof:

- ($\Rightarrow$) each clause will have at least one True literal. Pick them, and they should be connected with each other and forming a clique of size 3. There will be an edge between any two based on construction
- ($\Leftarrow$) if there is a clique, each of them must be from different clauses based on construction. assigning them to True will make sure each clause to be True without

conflicting assignments

9.3.6 CLIQUE $\leq_p$ VERTEX-COVER

Take the complement of the instance for CLIQUE (G, k). Take its complement $\overline{G}$, If G has a clique of $V' \subseteq V(G)$, it will have a vertex cover in $\overline{G}$ in $V - V'$.
$\because \forall (u, v) \in E(\overline{G}), u \notin V' \wedge v \notin V'$ by definition of clique.

10 NP Completeness

10.1 Complexity class

Let $t : N \to R^+$ be a function. Define the **time complexity class** $TIME(t(n))$ to be collection of all languages (set of all instances of a problem) that are decided by an $O(t(n))$ time Turing machine (solved in $O(t(n))$ time algorithm)

10.2 class P

Definition: problems that are solvable in a deterministic polynomial time

Alternative definition: P is a class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine, i.e.

$$P = \cup_k Time(n^k)$$

10.2.1 Examples of class P

- Minimum spanning tree (Prim and Kruskal algorithms)
- 2-CNF (converted into implication graph => Find strongly connected component using Tarjan component in polynomial time => check x and $\neg x$ in the SCC)
- Single Source Shortest Path (Dijkstra OR Bellman Ford)

10.3 class NP (non-deterministic polynomial)

Polynomial Verifiable: A **verifier** for a language A is an algorithm V where $A = \{w | V$ **accepts** (w, c) for some string $c\}$. V is polynomial time in the length of w, it is a polynomial time verifier, and corresponding A is polynomially verifiable. c is a **certificate** and it should have **polynomial length** in the length of w.

Class NP Definition: the class of problem which polynomial-time verifiable certificates of membership of a YES-instance exist, where the certificate are of polynomial size (If x is a YES instance, there is a polynomial-size certificate that I can verify in polynomial time).

Alternative Definition: class of languages that have polynomial time verifier.

Examples

CLIQUE (determine whether a graph contains a clique of specific size k) is class-NP problem, On input $<< G, k >, c >$, c be the subgraph of $k - clique$

How to verify:

- Test c has k nodes
- Test G contains all edges connecting nodes in c
- if both passes, *accept*, else *reject*

SUBSET-SUM (determine in set S, wether exists $S' \subseteq S$, s.t. $\sum_i s_i = t, s_i \in S'$)

Verifier V = On input $<< S, t >, S' >$

- Test c is sum to t
- Test whether S contains all numbers in c
- *accept* and *reject* correspondingly

Some lemmas:

- Any problem in P is also in NP
- $P = NP$ is still unsolved. We believe that $P \neq NP$ and $NP - complete$ problems we assume to be nonpolynomiality.

10.4 NP-Completeness

NP-Hard: A problem A is **NP-hard** if for every problem B in NP: $B \leq_p A$

NP-Complete: A problem A that is **NP-Hard** and in **NP**

If B is NP-complete and $B \in P$, then $P = NP$

Cook-Levin theorem: SAT (test whether a Boolean formula is satisfiable) is NP-Complete (Proof of all NP problems to reduced to it by constructing a polynomial time reduction)

Can easily shows that all SAT instance can be rewritten (reduced) into an instance of 3-SAT $\implies$ 3-SAT is also NP-Complete.

10.5 More examples on the proof of NPCs

10.5.1 SUBSET-SUM $\leq_p$ PARTITION

Simply add two elements to subset sum instance $e_1 = S$ and $e_2 = 2T$, then total sum of new set will be $2S + 2T$. By algebraic manipulation, we can show both $\Leftarrow$ and $\Rightarrow$ easily.

10.5.2 INDEPENDENT-SET $\leq_p$ CLIQUE

Definition of independent set: given a graph $G = (V, E)$, and an integer k. Is there a subset of k vertics such that **no two are adjacent**?

Definition of clique: Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or more) vertices such that there is an edge between each pair of vertices?

Intuition: complementary graph of a k-CLIQUE contains k nodes that are not connected to each other at all, i.e. there exists a k size independent set in complementary of k-CLIQUE

Reduction: To check whether G has a independent set of size k, check whether $\overline{G}$ has a clique of size k.

Proof:

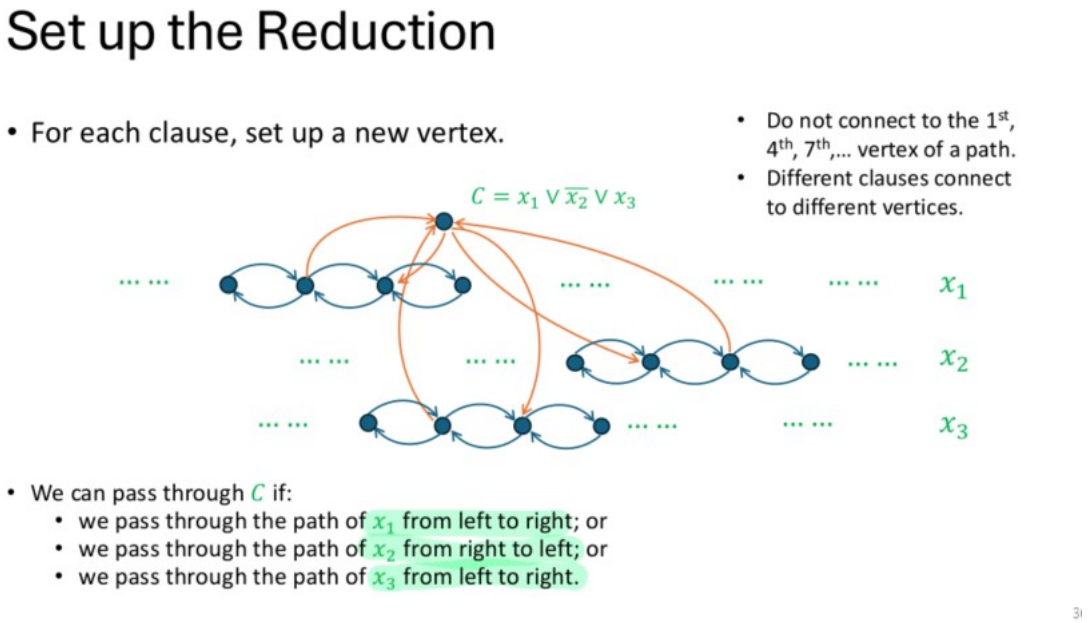
- ($\Rightarrow$) G has independent set of size k, there is no edge between any pair of these vertices, therefore there will be an edge between each pair of this in $\overline{G}$
- ($\Leftarrow$) $\overline{G}$ has a clique of size k, there is an edge between each pair of vertices in $\overline{G}$, there is no edge between any pair of these in original G

10.5.3 3-SAT $\leq_p$ DIRECTED-HAM-CYCLE

Set up the Reduction

- For each literal, we set up a path.

- Hamiltonian cycle either goes from left to right or from right to left
  - If left to right, x_i = True.
  - If right to left, x_i = False.
- The length is $3m + 1$, where m is the number of clauses.



Proof

- Given an instance Φ of 3-SAT, construct the directed graph G.
- ($\Rightarrow$) If Φ is satisfiable, then given the satisfying assignment, we can construct the Hamiltonian Cycle of G
- ($\Leftarrow$) IF G has a Hamiltonian Cycle, after visit the clause vertex it must travels back to the path it comes from.

10.5.4 DIRECTED-HAM-CYCLE $\leq_p$ UNDIRECTED-HAM-CYCLE

Intuition: split nodes in G_d in to $v_i^{in}, v_i^{mid}, v_i^{out}$ and connected from u_i^{out} to v_i^{in} if there exists a directed edge of (u, v)

10.5.5 PARTITION-SPECIAL $\leq_p$ FANTASTIC-HALF

Definition of FANTASTIC-HALF: Given non negative integers $a_1, \ldots, a_n$, and $b_1, \ldots, b_n$, decide whether there is a set $S \subseteq \{1 \ldots, n\}$ of size $n/2$ such that $\sum_{i \in S} a_i \geq A/2$ and $\sum_{i \in S} b_i \geq B/2$

Definition of PARTITION-SPECIAL: Given non-negative integers $x_1, \ldots, x_n$ decide whether they can be partitioned into two parts with equal size and equal sum (NP-Complete as it can be reduced from PARTITION with padding 0s).

Intuition: defined the A and B such that it shows the sum of some number is equal to HALF of the total sum. $\sum -b_i \geq -B/2 \implies \sum b_i \leq -B/2$

Reduction: let $a_i = x_i$, $b_i = X - x_i$

- ($\Leftarrow$) $\sum_i a_i > A/2 = X/2$, $\sum_i b_i > B/2 = (nX - X)/2$, Look at the second formulae

$nX/2 - \sum_i x_i \geq nX/2 - X/2$, this implies $\sum_i x_i \leq X/2$ which further implies $\sum_i x_i = A/2$

- ($\Rightarrow$) it is possible to find $i \in S, |S| = n/2$ and $\sum_{i \in S} x_i = X/2$. Hence, the S for FANTASTIC HALF exists for both $\sum_{i \in S} a_i$ and $\sum_{i \in S} b_i$

10.5.6 3-SAT $\leq_p$ SUBSET-SUM

Intuition: The structure of SUBSET-SUM contains numbers of large magnitude presented in decimal notation. The goal is to find the represent variables by pairs of numbers. Clauses should be encoded into some position into the number. And the target need to be chose carefully such that within each clauses, at least one variable that can be assigned to True

	x_1	x_2	x_3	C_1	C_2	C_3	C_4	Value in S	Meaning
$v_1 =$	1	0	0	1	0	0	1	1001001_{10}	
$v_1' =$	1	0	0	0	1	1	0	1000110_{10}	x_1 = FALSE satisfy C_2 and C_3
$v_2 =$	0	1	0	0	0	0	1	100001_{10}	
$v_2' =$	0	1	0	1	1	1	0	101110_{10}	x_2 = FALSE satisfy C_1, C_2, and C_3
$v_3 =$	0	0	1	0	0	1	1	10011_{10}	x_3 = TRUE satisfy C_3 and C_4
$v_3' =$	0	0	1	1	1	0	0	11100_{10}	
$s_1 =$	0	0	0	1	0	0	0	1000_{10}	Take both +1 slack
$s_1' =$	0	0	0	2	0	0	0	2000_{10}	and +2 slacks for C_1
$s_2 =$	0	0	0	0	1	0	0	100_{10}	
$s_2' =$	0	0	0	0	2	0	0	200_{10}	Take only +2 slacks for C_2
$s_3 =$	0	0	0	0	0	1	0	10_{10}	Take only +1 slack for C_3
$s_3' =$	0	0	0	0	0	2	0	20_{10}	
$s_4 =$	0	0	0	0	0	0	1	1_{10}	Take both +1 slack
$s_4' =$	0	0	0	0	0	0	2	2_{10}	and +2 slacks for C_4
$W =$	1	1	1	4	4	4	4	1114444_{10}	$C_1/C_2/C_3/C_4$ has target 4/4/4/4

11 Techniques for proving NPC

Restriction

For a given $\Pi \in NP$, it consists a known NP-Complete problem Π' as a special case. Specify restrictions on the instance Π such that it restricted problems will be equivalent to Π

Component Design

Pick some aspects of NP-complete instance to make up a collection of basic units, obtain instance of target by replace each basic units.

Local Replacement

Use constituents of target problem instance to design components that can be combined to realize instance of known NP-complete problems.

11.1 More examples

11.1.1 Minimum Equivalent Digraph

A directed graph $G = (V, E)$, and a positive integers K. Is there a graph $G' = (V, E')$ where $E' \subseteq E$ s.t. G' contains a path between u and v iff G also has

Reduced from DIRECTED HAMILTONIAN CIRCUIT by setting $K = |V|$

11.1.2 Multiprocessor scheduling

A finite set A of tasks, a length $l(a) \in Z^+$ for every $a \in A$. A number of m of processors and a deadline D. Is it possible to partition A into m disjoint set such that $max\{\sum_{a \in A_i} l(a), i \in [1, m]\} \leq D$

Reduce from PARITION, set $m = 2$, $D = \dfrac{\sum l(a)}{2}$

11.1.3 Lpath (Cisper)

$\{(G, a, b, k)|$G contains a simple path of length at least k from a to b$\}$

Reduced from UNDIRECTED HAMILTONIAN CYCLE, split an arbituary node v into v_1 and v_2, joins v_i to all u that $(u, v) \in E(G)$, $k = n$

11.1.4 Half-Clique (Cisper)

G is an undirected graph, it has a complete subgraph with at least $m/2$ nodes, where m is the number of nodes in G

Reduced from a CLIQUE, depends on the number of k and total number of nodes m, add additional nodes and edges when necessary to ensure that there are enough nodes for Half-Clique inputs.

11.1.5 NAE-3SAT

3SAT but assignment in a way such that each clause must contain two literal with unequal truth values.

This can be reduced from 3SAT. We now show a reduction from 3-SAT. We define a single "reference variable" z for the entire NAE-SAT formula. Split each clause in 3-SAT (x_1, x_2, x_3) to $(x_1, x_2, w_i), (\neg w_i, x_3, z)$

it is obvious that an valid instance 3-SAT will implies a valid assignment for NAE-3SAT by setting the w and b. If both x_1 and x_2 are false, set w_i to True. Otherwise, set it to False. Assign False to z.

The negation of an instance of NAE-3SAT is also a valid assignment. Set z to False by flipping the whole assignment when necessary. Therefore, at least one of the x_1, x_2, x_3 will be True. If all False, at least one clause will not be True

11.1.6 Set-Splitting (Cisper)

S is a finite sets and C is a subset of S, each element in S can be colored to be red or blue but no C_i can be colored with same color

Reduced from NAE-3SAT, True for read and False for Blue.

11.1.7 Minimum Sum of Squares

Finite set A, $s(a) \in Z^+$, positive integers k and j. Query: Can elements of A be partitioned into K disjoint sets such that $\sum_k (\sum_{i \in S_k} s(a))^2 \leq j$.

Reduce from PARITION, choose $K = 2$, and observe the formulae will become: let S be total sum and x be the first sum of first set, $x^2 + (S - x)^2 = 2(x - \frac{S}{2})^2 + \frac{S^2}{2}$, which has minimum of $S^2/2$, if we set $j = S^2/2$, the YES instance will imply equality

A Mathematics Useful properties (CLRS)

- $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- $-\lceil x \rceil = \lfloor x \rfloor;\ -\lfloor x \rfloor = \lceil -x \rceil$
- $\lceil \frac{\lceil x/a \rceil}{b} \rceil = \lceil \frac{x}{ab} \rceil$
- $\lfloor \frac{\lfloor x/a \rfloor}{b} \rfloor = \lfloor \frac{x}{ab} \rfloor$
- $\lceil \frac{a}{b} \rceil \leq \frac{a+b-1}{b}$
- $\lfloor \frac{a}{b} \rfloor \geq \frac{a-b-1}{b}$
- $a \mod b = a - n\lfloor a/n \rfloor$
- $e^x = 1 + x + \frac{x^2}{2!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$
- $1 + x < e^x;\ |x| \leq 1 \implies 1 + x \leq e^x \leq 1 + x + x^2$
- $\lim_{n\to\infty} (1 + \frac{x}{n})^n = e^x$
- $1 + x \leq e^x$
- $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$
- $n! = o(n^n);\ n! = \omega(n^n);\ \log(n!) = \Theta(n \log n)$
- $\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}$
- $\sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$

Bounding summations and Mathematical induction:

for a series $\sum_{k=1}^{n} a_k$, $a_{max} = max\{a_k : 1 \leq k \leq n\}$, then $n \cdot a_{min} \leq \sum_{k=1}^{n} a_k \leq n \cdot a_{max}$

splitting summation: $\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k_0-1}^{n} a_k = \Theta(1) + \sum_{k=k_0}^{n} a_k$

Approximation in integrals:

$f(x)$ is monotonically increasing

$$\int_{m-1}^{n} f(x)dx \ \leq \ \sum_{k=m}^{n} f(x) \ \leq \ \int_{m}^{n+1} f(x)dx$$

$f(x)$ is monotonically decreasing

$$\int_{m}^{n+1} f(x)dx \ \leq \ \sum_{k=m}^{n} f(x) \ \leq \ \int_{m-1}^{n} f(x)dx$$

Binomial coefficients:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

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

B Theory of Computation Useful Notes

B.1 Turing Machine (Sisper TOC)

B.1.1 Strings and Language

- **alphabet**: an nonempty finite set
- **symbol**: member of alphabet
- **string over alphabets**: finite sequence of symbols from that alphabet
- The input to a Turing Machine is always a **string**. We must represent **object** as a **string** $\implies$ String can easily represent polynomials, graphs, grammars, automata and any combination of those objects (**encoding**)

B.1.2 Definition of Turing Machine

A Turing Machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets.

- Q is the set of all states.
- Σ is the input alphabet not containing **blank symbol** .
- Γ is the tape alphabet, $\in \Gamma$ and $\Sigma \subseteq \Gamma$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.
- $q_0 \in Q$ is the start state.
- $q_{accept} \in Q$ is the **accept** state the TM should halts when reaches this state.
- $q_{reject} \in Q$ is the **reject** state the TM should halts when reaches this state.

The machine may **accept**, **reject** and **loop** (i.e. do not halt).

B.1.3 Decidability and Recognizability

- **Recognizable**: A language L is Turing-recognizable if there exists a Turing machine M such that:

  $$\omega \in L \Rightarrow M \text{ accepts } \omega$$

  (No restriction on what happens if $\omega \notin L$.)

- **Decidable**: A language L is Turing-decidable if there exists a Turing machine M

such that:

$$\omega \in L \Rightarrow M \text{ accepts } \omega$$

$$\omega \notin L \Rightarrow M \text{ rejects } \omega$$

and M halts on all inputs.

Note: Decidable languages correspond to problems that are algorithmically solvable in finite time. In complexity theory, this notion underlies classes such as **P** (polynomial-time decidable languages) and **NP** (nondeterministically polynomial-time verifiable languages). In particular, **P** $\subseteq$ **NP** $\subseteq$ Decidable.

Language Theory	**Algorithm / Intuition**
$x \in L$	x is a YES-instance
$x \notin L$	x is a NO-instance
$M(x)$ accepts	Algorithm outputs YES
$M(x)$ rejects	Algorithm outputs NO
Turing Machine M decides L	Algorithm solves the decision problem

$$L \subseteq \{0,1\}^* \quad \Longleftrightarrow \quad \text{the set of all YES-instances}$$

$$\text{Decision problem} \equiv \text{Language} \equiv \text{Set of YES-instances}$$

B.2 Lambda Calculus (CS4212)

The syntax of pure lambda calculus can be defined as:

$$\langle expr \rangle ::= \langle var \rangle \mid \lambda \langle var \rangle . \langle expr \rangle \mid (\langle expr \rangle \ \langle expr \rangle)$$

$$\langle var \rangle ::= \texttt{identifier}$$

Interpretation

- `<var>` represents variables (e.g., x, y, z)
- λ `x.M` represents function abstraction
- `(M N)` represents function application

Computation Rule

The only computation rule is β-reduction:

$$(\lambda x.M)\ N \to M[x := N]$$

which captures function application via substitution.

B.3 Hilbert's Problems (Sisper TOC)

- Is there a general algorithm that decides whether a polynomial has an integral root? (Hilbert 10^{th} problem)
- Is there a mechanical procedure that can decide whether any mathematical statement is true or false? (Entscheidungsproblem)

B.4 Church vs Turing Computability (Sisper TOC)

B.4.1 Turing Computability

A function is **Turing-computable** if there exists a Turing machine that:

- encodes the input as a finite string over a finite alphabet,
- performs step-by-step symbol manipulation on an infinite tape,
- produces the correct output (and halts for total computable functions).

Thus, computation is viewed as a sequence of state transitions over encoded strings.

B.4.2 Church (Lambda Calculus) Computability

A function is **Church-computable** if it can be expressed in the lambda calculus such that:

- inputs and outputs are represented as lambda terms,
- computation proceeds purely by function application and β-reduction,
- evaluation eventually reduces to a normal form (form that cannot be further reduced by β reduction) representing the output.

Thus, computation is viewed as symbolic rewriting of expressions.

B.4.3 Church–Turing Thesis

Both models define the same class of effectively computable functions:

$$\text{Lambda calculus computable} \equiv \text{Turing computable} \equiv \text{Intuitive notions of algorithm}$$

This equivalence is known as the Church–Turing thesis.

Key insight:

- Turing machines: computation = state + tape manipulation
- Lambda calculus: computation = function rewriting

Despite different formalisms, both capture the same notion of algorithm.