# cs3230 notes

| | |
|---|---|
| Tags | |
| Type | Notes |

📌 **algorithm:** sequence of unambiguous and executable instructions for solving a problem (obtain a valid output given a valid input)

## properties of good algorithms

1. correctness
2. generality: applicable to wide range of inputs
3. device independent (as far as possible)
4. efficient in terms of time, space, resources (worst/average/best case)
5. usable as "subroutine" for other problems
6. simple to code, understand, and debug
7. well documented

## dealing with really large outputs

- applying modulo to results ($m \approx 2^{wordsize}$)

## analysis of algorithms

- model of computation: RAM
- every instruction takes constant amount of time
- counting number of instructions needed
- complexity based on input size

## running time $T(n)$

- worst case: maximum time needed for any input of size (at most) $n$
- average case: expected time taken over all inputs of size $n$
    - assumes all inputs are equally probable (or follows some probability distribution)

## comparing efficiencies



- matters only for large sized inputs

## asymptotic analysis

- not measuring actual run time
- for large inputs, how does the run time behave?
- often ignore constant multiplicative factors
- nothing to do with best/worst/average case runtime
    - asymptotic analysis happens within each class of runtime

### steps to proof

1. find a $c, n_0$ that fit the definition for each of the terms of $f$
2. add up all your $c$, take the max of your $n_0$
3. add up all your inequalities to get the final inequality you want
4. explain what $c$ and $n_0$ are

### using limits to determine bounds

- assume $f(n), g(n) > 0$
- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) < \infty \Rightarrow f(n) = O(g(n))$
- $0 < \lim_{n\to\infty}(\frac{f(n)}{g(n)}) < \infty \Rightarrow f(n) = \Theta(g(n))$
- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) > 0 \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) = \infty \Rightarrow f(n) = \omega(g(n))$

### common time complexities

- in order of increasing time complexity

1. $O(1)$
2. $O(a^n), a < 1$
3. $O(\lg \lg n)$
4. $O(\lg n)$
5. $O(n)$
6. $O(n \lg n)$
7. $O(n^k), k > 1$
8. $O(a^n), a > 1$



9. $O(n!)$

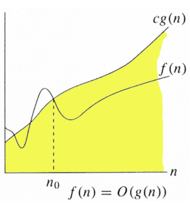
- $O(n^k) > O(n \lg n)$ but $n$ may have to be very large if $1 < n \leq 2$

### properties

- reflexivity: for $O, \Omega, \Theta$, $f(n) = O(f(n))$
- transitivity: for all, $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- symmetry: $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
- complementarity:
    - $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
    - $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

### upper bound: $f \in O(g)$

- if there exists constant $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$
- $g$ is an upper bound on $f$
- $O(g) = \{f : \exists c > 0, n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$
- $f$ grows no faster than $g$

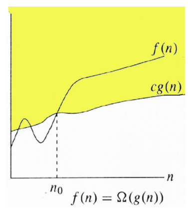



### lower bound: $f \in \Omega(g)$

- if there exists a constant $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq cg(n) \leq f(n)$
- $g$ is a lower bound on $f$
- $\Omega(g) = \{f : \exists c > 0, n_0 > 0 : \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$
- $f$ grows no slower than $g$
    - think of it as: "nothing gets worse than what $g$ is"
    - also can think of it as "it takes at least $\Omega(g)$ to run"

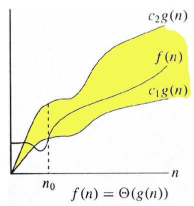


- while $f \in \Omega(1)$ is always a possibility, a better lower bound is one that for a much larger $n_0$ works as a "lower bound"




### tight bound: $f \in \Theta(g)$

- if there exists a constant $c_1, c_2 > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $g$ is a tight bound on $f$
- $\Theta(g) = \{f : \exists c_1, c_2 > 0, n_0 > 0 : \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$




- $\Theta(n) = O(g) \cap \Omega(g)$

### strict upper bound: $f \in o(g)$

- if for all constant $c > 0$ there exists a constant $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq f(n) < cg(n)$
- $g$ is a strict upper bound on $f$
- $o(g) = \{f : \forall c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$

### strict lower bound: $f \in \omega(g)$

- if for all constant $c > 0$ there exists a constant $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq cg(n) < f(n)$
- $g$ is a strict lower bound on $f$
- $\omega(g) = \{f : \forall c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$

## recurrences

- approximating $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ to be $\frac{n}{2}$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n), n > 1 \\ \Theta(1), n = 1 \end{cases}$$

- base case usually omitted
- often taken as constant for small (constant) size input

### telescoping method

- for any sequence $a_0, a_1, \ldots, a_n$, $\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n$
- given $T(n) = aT(\frac{n}{b}) + f(n)$, express it as $\frac{T(n)}{g(n)} = \frac{T(\frac{n}{b})}{g(\frac{n}{b})} + h(n)$ where $h(n) = \frac{f(n)}{g(n)}$
    - think of how to divide the expression
    - $\ell$ is the height of the recurrence

$$\begin{aligned} \frac{T(n)}{g(n)} &= \frac{T(n/b)}{g(n/b)} + h(n) \\ \frac{T(n/b)}{g(n/b)} &= \frac{T(n/2b)}{g(n/2b)} + h(n) \\ &\ldots \\ \frac{T(b)}{g(b)} &= \frac{T(1)}{g(1)} + h(n) \end{aligned}$$

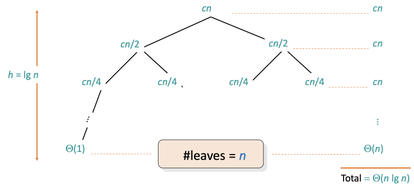
$$\begin{aligned} &\Rightarrow \frac{T(n)}{g(n)} = \frac{T(1)}{g(1)} + \ell \times h(n) \\ &\Rightarrow T(n) = g(n) \times T(1) + \ell \times h(n) \times g(n) \\ &\Rightarrow T(n) \in O(\ell \cdot h(n) \cdot g(n)) \end{aligned}$$

### recursion tree

- given recurrence $T(n) = g(n)T(k(n)) + f(n)$, draw a recursion tree



    - calculate the depth of the tree
    - calculate the work done per level
    - total work: depth * work done per level
        - alternative: sum work per level across depth
- common $g(n)$, $k(n)$ heights:
    - $g(n) = 1, k(n) = \frac{n}{b} \Rightarrow \log n$
    - $g(n) = \sqrt{n}, k(n) = \sqrt{n} \Rightarrow \log \log n$
    - $g(n) = 1, k(n) = n - 1 \Rightarrow n$




### master theorem

- given recurrence of form $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$ and $f$ is asymptotically positive
- let $c_{crit} = \log_b(a)$
- compare $f(n)$ against $n^{c_{crit}}$, focusing on the power of $n^d$ if it exists in $f(n)$
- case 1 ($c_{crit} > d$): $f(n) = O(n^{c_{crit} - \varepsilon})$ for some constant $\varepsilon > 0$
    - work done at the leaves is more than that at the top
    - $f(n)$ grows asymptotically slower than $n^{c_{crit}}$ by a factor of $n^{\varepsilon}$
    - $T(n) \in \Theta(n^{\log_b(a)})$
- case 2 ($c_{crit} = d$): $f(n) = \Theta(n^{c_{crit}} \log^k(n))$ for some constant $k \geq 0$
    - work done at every level is the same
    - $f(n)$ and $n^{c_{crit}}$ grows at similar rates
    - $T(n) \in \Theta(n^{\log_b(a)} \log^{k+1}(n))$
- case 3 ($c_{crit} < d$): $f(n) = \Omega(n^{c_{crit}} + \varepsilon)$ for some constant $\varepsilon > 0$
    - work done at the root is more than that of the other levels
    - $f(n)$ grows polynomially faster than $n^{c_{crit}}$ by a factor of $n^{\varepsilon}$



    - $f(n)$ must also satisfy the regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$
        - guarantees that sum of subproblems is smaller than $f(n)$
    - $T(n) \in \Theta(f(n))$

### substitution method

- guess the form of the solution ($O(f(n))$) and verify by induction
- for induction
    - choose values for $c$ and $n_0$
    - prove base case where $n = n_0 = 1$ chosen such that $T(1)$ is satisfied
    - recursive case for $n > 1$
        - using strong induction, assume $T(k) \leq cf(n)$ for $n > k \geq 1$
        - use $T(n)$ and solve for the recurrence using induction
- choice of induction hypothesis is very important
    - key: split up the constants across $c_1, c_2, \ldots$ since they cannot be treated as the same

## correctness

- on all valid inputs, the algorithm gives correct outputs
- parts of a correctness proof:
    - invariants: define invariants (every recursive call or loop)
    - initialization: prove that invariants are true at the start
    - maintainance: show (usually by induction) that if they are true at the start, invariants hold true at the start of the next iteration
    - conclusion: conclude that at the end, the algorithm gives the right answer
- iterative algorithms:
    - inner loops rely on correctness of outer loops
    - the end of loop invariant is used to prove correctness of termination
    - show that
        - invariant true at initialization
        - correctly maintained
        - implies correctness with termination condition
- recursive algorithms:
    - usually induction based on parameters of algorithm (e.g. length of search for binary search)



    - base case: same base cases as recursion where we explicitly prove those work
    - induction step (using strong induction): if all other calls of algorithm work, then the recursive call to these sub-calls will also work given all the possible cases

## divide and conquer

1. divide the problem into smaller subproblems
2. solve the subproblems recursively (conquer)
3. combine/use subproblem solutions to get the solution to the full problem

$$T(n) = aT(n/b) + f(n)$$

* may not always work with master theorem

- uses induction for proof of correctness (given recursive algorithm)

## sorting

- input: sequence $(a_1, a_2, \ldots, a_n)$ of comparable objects
- output: permutation $(a_1', a_2', \ldots, a_n')$ of input such that $a_i' \leq a_{i+1}'$ for $0 \leq i < n$
- properties;
    - small runtime across worst case and average case
    - simple
    - in-place sorting
    - stability
    - comparison based

### in-place

- uses constant (or very little) extra memory besides the input list
- insertion sort ($O(1)$ extra space)
- randomized quick sort ($O(\log n)$ extra space)

### stable

- for "equal" elements, the original ordering is preserved
- can be maintained using an auxiliary array to indicate the position of elements
- insertion and merge sort

### comparison-based

- elements can only be compared with each other

- no other property of elements can be used
- insertion, merge, heap, and quick sort
- best worst-case runtime: $O(n \log n)$
- modelled using decision tree where each comparison is a node and leaf nodes is the sorted list based
  - each node is $a_i \leq a_j$
  - left subtree: yes, right subtree: no
  - worst case running time (number of comparisons performed) is longest path from root to leaf

> **theorem:** any comparison based algorithm takes at least $\Omega(n \log n)$ time

- model the algorithm as a tree and the tree contains at least $n!$ leaves for every possible permutation of the input
  - height of tree is at least $\log(n!) = n \log n - n \log e + O(\log n) \approx \Omega(n \log n)$ (using Stirling's approximation)

> **corollary:** merge sort is optimal for comparison based sorting

## quick sort

```
def quick_sort(A, p, r):
    if p >= r: return
    pivot = A[p]
    q = partition(A, p, r, pivot)
    quick_sort(A, p, q - 1)
    quick_sort(A, q + 1, r)

quick_sort(A, 0, n - 1)
```

- worst case: array sorted $T(n) = T(j-1) + T(n-j) + O(n) \in \Theta(n^2)$
- average case: distinct and sorted and choosing an ideal pivot that provides uniform mapping $O(n \log n)$
  - choose pivot at random to ensure uniform distribution of permutations

  > **theorem:** probability that the run time of randomized quick sort exceeds average by $x\% = n^{-\frac{x}{100} \ln \ln n}$

  - probability the run time of randomized quick sort is double the average given $n \geq 10^6$ is $10^{-15}$



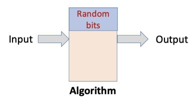
## counting sort $\Theta(n+k)$

- not comparison-based sorting
- input: $A[1..n]$, where $A[i] \in \{1, 2, \ldots, k\}$
- output: $B[1..n]$ (sorted version of $A$)
- $C[1..k]$ holds the number of elements smaller than or equal to $i$
  - final sorted array is created by moving $i$ to $B[C[i-1]+1]$ to $B[C[i]]$
- if $k = O(n)$, then $\Theta(n)$ time
  - if $k \geq n \log n$, $O(n+k) \geq O(n \log n)$

## radix sort $\Theta(\frac{bn}{\log n})/\Theta(dn)$

- sort least significant digit/bits first using counting sort since number of digits is small
- stable sorting
- $b$ bit word broken into $b/r$ groups of $r$ bit words
  - $r$ must be chosen well
  - $b/r$ passes
  - each pass is $\Theta(n + 2^r)$
  - total: $\Theta(\frac{b}{r}(n + 2^r))$
  - optimal $r = \log n$
    - $\Theta(\frac{bn}{\log n})$
  - if numbers in range $[1, n^d]$, then $b = d \log n$: $\Theta(dn)$

# randomized algorithms




- output and running time are functions of the input and random bits chosen
  - inputs cannot be controlled so randomize other things
- potential outcomes: TLE or WA



- if given a set of random bits and the answer is always right, then random bits are useless
- types:
  - las vegas: output is always correct, runtime has small probability of being too large and overall expected runtime is good
  - monte carlo: output is not always correct (but with small probability), overall runtime is consistently good
- analysis relies on linearity of expectation: $E(A+B) = E(A) + E(B)$
- increasing the probability of success involves increasing the number of times the experiment is performed
  - find an appropriate random variable $X$ to represent the outcome
- others: smallest enclosing circle, minimum cut, primality test
- perfect random number: using fresh start of computer and taking bits from the picoseconds or quantum physics
- union bound:

$$P[X_1 \cup X_2 \cup \cdots \cup X_n] \leq P[X_1] + P[X_2] + \cdots + P[X_n]$$

## Freivald's algorithm

- input: given matrices $A, B, C$
- output: true if $A \cdot B = C$, else false
- fastest deterministic algorithm: using Strassen's method $\Theta(n^{log_2 7})$
- algorithm $O(kn^2)$
  - pick uniformly random bit vector of size $n$, $\hat{x}$
  - check if $A \cdot (B \cdot \hat{x}) = C \cdot \hat{x}$
  - repeat $k$ times with independent $\hat{x}$
  - if all succeed, then true, else false
- error probability: at most $2^{-k}$ per pass

## finding approximate median

- deterministic: $O(n)$ (quick select)
- approximate median: element $y$ which has rank between $n/4$ and $3n/4$
- randomly pick an element, the probability of being an approximate median is at least $1/2$
  - repeat this $k$ times to increase probability of success
  - probability of error: less than $\frac{1}{n^2}$ if $k = 1 + 10 \log n$

## analysis: balls in bins



> use an indicator variable of the event and find the sum of that instead
>
> if event involves multiple variables, compute the summation across all variables

- given $m$ balls to be placed in $n$ bins with uniformly random probability
- probability $k$ bins are empty: $(1 - \frac{k}{n})^m$
- probability at least 1 bin is empty: $OBE(n, m) = \binom{n}{1}(1 - \frac{1}{n})^m - \binom{n}{2}(1 - \frac{2}{n})^m + \ldots (-1)^{k+1}\binom{n}{k}(1 - \frac{k}{n})^m + \ldots$
- expected number of empty bins: let $X_i$ be random variable such that
  - $E(X_i) = 1 \times P(i^{th} \text{ bin is empty}) + 0 \times P(i^{th} \text{ bin is not empty}) = (1 - \frac{1}{n})^m$
  - $X_i$ follows a bernoulli random variable distribution hence $E(X_i) = p$

$$X_i = \begin{cases} 1, & \text{if } i^{th} \text{ bin is empty} \\ 0 \end{cases}$$

- depending on the distribution that $X_i$ follows, $E(X_i)$ may not always be $p$
  - for e.g. $X_i \sim Geom(p) \Rightarrow E(X_i) = 1/p$

# dynamic programming

- overlapping subproblem: recursive solution contains a "small" number of distinct subproblems repeated many times
  - does not need to be entirely overlaps, but the more the better (faster)
- optimal substructure: optimal solution of a state can be constructed from the optimal solution of subproblems
- cut-and-paste argument (extension of proof by contradiction)
  - suppose an "optimal" solution is found with suboptimal substructures
  - cut the suboptimal substructures out and paste the optimal substructure to reveal an even more optimal solution
  - therefore, there is a contradiction
- top-down vs bottom-up
  - both work
  - top-down sometimes saves some computation of unnecessary subproblems but can introduce overhead of recursive call
  - both provide same asymptotic time complexity
  - top-down may suffer from space overhead of recursive call and space optimization is harder

- longest common subsequence, $lcs(i,j)$ is the longest common subsequence of $A[:i]$ and $B[:j]$A

$$lcs(i,j) = \begin{cases} lcs(i-1,j-1)+1, A[i] = B[j] \\ \max\{lcs(i-1,j), lcs(i,j-1)\}, A[i] \neq B[j] \\ 0, i = 0 \lor j = 0 \end{cases}$$

$$T(n) \in \Theta(nm)$$

- longest palindromic subsequence, $dp(i,j)$ is the longest palindromic subsequence between $A[i:j]$
  - extension of LCS: run LCS on original and reversed string and find the overlap
  - optimal solution:

$$dp(i,j) = \begin{cases} dp(i+1,j-1)+1, A[i] = B[j] \\ \max\{dp(i+1,j), dp(i,j-1)\}, A[i] \neq B[j] \\ 0, i = 0 \lor j = 0 \end{cases}$$

$$T(n) \in \Theta(nm)$$

### knapsack problem

- input: $(w_1,v_1),(w_2,v_2),\dots,(w_n,v_n)$ and $W$
- output: subset of $\{1,2,\dots,n\}$ such that $\sum_{i\in S} v_i : \sum_{i\in S} w_i \leq W$

$$dp(i,j) = \begin{cases} 0, i = 0 \lor j = 0 \\ \max\{dp(i-1,j), dp(i-1,j-w_i)+v_i\}, w_i \leq j \\ dp(i-1,j) \end{cases}$$

$$T(n) \in \Theta(nW)$$

- $dp(i,j)$ is the maximum value achievable given $items[:i]$ items and $j$ maximum $W$
- if infinite supply of weights, modify the algorithm so once a weight is taken, we can continue to take from it by going to $dp(i, j-w_i)$

## greedy

- recast the problem so only one subproblem needs to be solved each step
  - beats dynamic programming and divide and conquer if it works
- greedy choice: let `g` be the greedy choice. there exists an optimal solution `s` that contains `g` (i.e. a locally optimal choice is globally optimal)
  - state the greedy choice `g` and `s` be the optimal solution



- if `s` contains `g` then done
- if `s` does not contain `g`, then swap the existing solution with `g` to get the same or even better solution
- optimal substructure: if we have `A \ {j}` where `A` is the input array, the remaining solution `s'` must be the optimal solution as well
  - proof by contradiction in assuming that no optimal solution that contains the greedy choice contains the optimal substructure
    - let `s` be the optimal solution with the greedy choice
    - `s' = s - {g}` with `s'` not being optimal
    - let `s''` be some other optimal solution
      - `s''` is better than `s'`
      - `s''` cannot be better than `s`
      - therefore, contradiction
    - **alternative name:** cut and paste argument
  - constructive proof: showing that $OPT(A) = \hat{A}$ and $OPT(\hat{A}) = A$ using Huffman code
    - "these are the rules and if we played by the rules, we would have gotten the optimal solution no matter what so there's no other way to get a sub-optimal solution"

### fractional knapsack

- input: $(w_1,v_1),(w_2,v_2),\dots,(w_n,v_n)$ and $W$
- output: weights $x_1,\dots,x_n$ that minimizes $\sum_i v_i \cdot \frac{x_i}{w_i}$ such that $\forall j \in [n], \sum_i x_i \leq W \land 0 \leq x_j \leq w_j$
- optimal substructure: removing $w$ of item $j$ results in the reamining load of the optimal knapsack weighing at most $W - w$ such that $n-1$ original items can be taken and $w_j - w$ of item $j$
  - proof: let $S$ be the optimal knapsack given $n$ items and the remaining load after removing $w$ of item $j$ be $S' = S - w_j$. Suppose that $S'$ is not optimal, thus there exists a knapsack where $S'' > S'$ for the remaining $n-1$ items. $S'' > S' \Rightarrow S'' > S - w_j \Rightarrow S'' + w_j > S$ which contradicts the initial statement that $S$ is the optimal knapsack. Therefore, there exists an optimal structure
- greedy choice property: let $j^*$ be the item with the maximum $\frac{v_j}{w_j}$. There exists an optimal knapsack containing $\min\{w_{j^*}, W\}$ kgs of $j^*$
  - proof: suppose an optimal knapsack contains $x_1 + x_2 + \dots + x_n = S$. If $S$ contains the greedy choice, then the greedy choice property is proven. Otherwise, without loss of generality, replace $x_1$ with $g$. Notice that $S' = S$ so the greedy choice is proven
- time complexity: $O(n \log n)$



### huffman code**

> **key problem**
>
> given an alphabet set
> $A = \{a_1, a_2, \dots, a_n\}$ and a text file $T \subseteq A^m$, find the number of bits to encode the text file, where average bit length is $ABL(\gamma)$ and encoding function is $\gamma(a_i)$

- fixed length coding: encode each letter in $A$ in binary encoding
  - let $k$ be the longest letter, so total number of bits to encode $A$ is $O(\log_2(k))$ to follow the longest letter
  - total encoding of $T$ is $O(m\log(k))$ bits
- variable length coding: exploits the variation in frequencies of alphabets
  - more frequent → coding with shorter bit string and inverse
  - naive variable encoding → key problem: $\gamma(b)$ is a prefix of $\gamma(d)$ so it is easy to mix up the encoding
- prefix coding: $\gamma(A)$ is a prefix coding if $\nexists x, y \in A : \gamma(x) \text{ is prefix } \gamma(y)$, prefix coding $\subseteq$ variable length coding

> **reformulated problem**
>
> given
> $A$ and the respective frequencies $f(a_i)$, compute $\gamma$ such that $\gamma$ is prefix coding and $ABL(\gamma)$ is minimum

- start by treating encoding as a labelled binary tree with edges labelled as 0 and 1
  - cost of an alphabet is the cost of label of path from root
  - leaf nodes are the alphabets that the path corresponds to

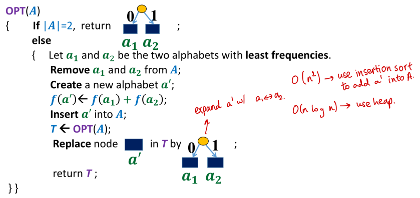


> **theorem**
>
> for each prefix code of
> $A$, there exists a binary tree $T$ with $n$ leaves such that:
> - there is a bijective mapping from alphabets to leaves
> - the label of a path from root to leaf node is a prefix code of the corresponding alphabet
>
> $$ABL(\gamma) = \sum_{x\in A} f(x) \cdot |\gamma(x)| = \sum_{x \in A} f(x) \cdot |depth_T(x)|$$

> **lemma**
>
> the binary tree corresponding to optimal prefix coding must be a full binary tree where every internal node has degree exactly 2

> **lemma**
>
> there exists an optimal prefix coding in which $a_1^{'}$ and $a_2^{'}$ are siblings (given that it is a complete tree)
>
> this implies merging
> $a_1^{'}$ and $a_2^{'}$ to $a''$ with the combined frequencies

- **solving:** sort $A = A'$ by non-decreasing frequency where $f(a_1^{'}) \leq f(a_2^{'}) \leq \dots \leq f(a_n^{'})$
  - more frequent alphabets exist closer to the root, less frequent alphabets exist away from the root
    - swapping lower frequency nodes with higher frequency nodes will not result in a decrease in $ABL(\gamma)$ (i.e. bringing lower frequency higher up in the tree)




- **constructive proof** — optimality with relation to $\hat{A}$: show that $OPT_{ABL}(A) = OPT_{ABL}(\hat{A}) + f(a_1^{'}) + f(a_2^{'})$
  - let $\hat{A}$ be the greedy subproblem
  - showing $OPT_{ABL}(A) \leq OPT_{ABL}(\hat{A}) + f(a_1^{'}) + f(a_2^{'})$ and $OPT_{ABL}(A) \geq OPT_{ABL}(\hat{A}) + f(a_1^{'}) + f(a_2^{'})$
  - prefix coding of $A$ derived from $OPT(\hat{A})$ and prefix coding of $\hat{A}$ from $OPT(A)$

# amortized analysis

- strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive
- guarantees average performance of each operation in the *worst case*

## types of analysis:

1. aggregate method
2. account method
3. potential method

### aggregate method

- very crude, not to be used for more precise computations
- sum of cost for all $n$ operations and divide by $n$
  - for operations that occur on every single element, consider applying the operation elementwise and upper bounding that number

### accounting method

- assuming 1 unit of work takes $1 from the bank
- charge an operation an amortized cost $c(i)$ with any excess being stored into the bank
- inexpensive operations should provide additional charge so there is excess in the bank for expensive operations



- bank balance cannot be negative
  - **must prove this fact**
- $\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i), \forall n$ where $t(i)$ is the true cost
  - this allows amortized cost to act as an estimate for the true cost overall

### potential method

- select potential function $\phi(i)$ that estimates the potential at the end of the $i^{th}$ operation where
  - $\phi(0) = 0$
  - $\phi(i) \geq 0 \forall i$
- amortized cost of $i^{th}$ operation: actual cost ($\gamma(i)$) + $\phi_i - \phi_{i-1} (= \Delta\phi_i)$ = $\alpha(i)$
- amortized cost of $n$ operations: $\sum_{i=0}^{n} \alpha(i) \geq \sum_{i=0}^{n} \gamma(i)$
- select $\phi$ such that expensive operations cause $\Delta\phi_i < 0$ such that it nullifies/reduces the effect of the actual cost
  - implicitly: choose $\phi$ that result in $\alpha(i)$ as the same across all cases of an operation or does not depend entirely on $i$
  - try finding quantities that decrease during operations
    - look at the types of properties of the data structure like size, relative size to another, remaining space, occupancy ratio, suffixes, etc.
    - also consider the negation of various properties like size
      - but must provide some offset to ensure $\phi(i) \geq 0$
    - for numbers, consider things like the sum of digits
- use table to show potential method (sufficient)
  - amortized cost = actual cost + $\Delta\phi_i$
  - try to achieve amortized cost that is not dependent on $i$

| Operation | Actual Cost | $\phi(i)$ | $\phi(i-1)$ | $\Delta\phi_i$ | Amortized cost |
|---|---|---|---|---|---|
| | | | | | |

### k-bit binary counter

📌 count total bit flips during $n$ increments = $T(n)$

- naive analysis: let $t(i)$ be the number of flips in increment $i$, max $t(i) = k$ where suffix contains all `1`



$$T(n) = \sum_{i=1}^{n} t(i) = O(nk)$$

- counting bit flips of bit $i$ instead: $f(i)$ where $f(0) = n, f(1) = n/2, f(i) = n/2^i$

$$T(n) = n \sum_{i=0}^{k-1} 2^{-i} < 2n = O(n)$$

  - amortized cost per increment: $T(n)/n < 2 = O(1)$
- k-bit binary counter: charge $2 per 0 → 1
  - excess $1 stored for when 1 → 0 (resets)
  - after $i$ increments, the banks has enough left for 1 → 0 flips
  - amortized cost per operation: $2 = O(1)$

## modified queue

- `INSERT(x)`: inserting 1 element `x`
- `EMPTY()`: emptying the entire queue
- amortized: $O(1)$ for both operations
  - `EMPTY()` with `k` deleted → `INSERT(x)` called `k` times before

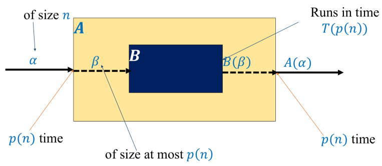
## dynamic table

- data structure (table) that grows when it overflows and shrinks when it underflows
  - existing data moved to new table (expensive operation)
- notation:
  - `n`: number of elements in table
  - `createTable(k)`: creates table of size `k`
  - `size(T)`: size of table `T` (irrespective of number of elements)
  - `copy(T, T')`: copying contents of `T` to `T'`
  - `free(T)`: free space occupied by `T`
- naive: resizing for every element
  - cost of $n$ insertions: $O(n^2)$
- aggregate method
  - cost of $n$ insertions: $\sum_{i=1}^{n} t(i) \leq n + \sum_{j=0}^{\log(n-1)} 2^j \leq 3n$
  - average cost: $O(n)/n = O(1)$
- potential method
  - let $\phi(i) = 2i - size(T)$



- account method
  - charging $2 per insertion to pay for `copy(T, T')`

# intractability




## reductions

- given instance $\alpha$ of $A$ (input of $A$), $A \leq_P B$ ($A$ reduces to $B$) if
  1. $\alpha \rightarrow \beta$ where $\beta \in B$ (instanceof $B$) (converting input of $A$ to an input for $B$)
  2. $solve(\beta) = S_\beta$
  3. $S_\beta \rightarrow S_\alpha$

📌 **layman terms**

A reduces to B iff converting $\alpha \in A$ to $\beta \in B$ and solving $\beta$ can derive a solution for $\alpha$ so the problem of A can be reduced to (represented in terms of B) and solved through B

alternative: A is a special case of B so if we can convert A to the form of B, we can solve B → solving A

### MAT-SQR $\leq_P$ MAT-MUL

- $\alpha \in$ MAT-SQR → matrix $C$

1. $\alpha \rightarrow \beta$: $C$ as both $A$ and $B$ of MAT-MUL
2. $solve(C) = A \cdot B = C \cdot C = C^2$
3. $S_\beta \rightarrow S_\alpha$: $C^2$ is the solution for MAT-SQR as well

### T-SUM $\leq_P$ 0-SUM

- $\alpha \in$ T-SUM → array $B$ of length $n$ and number $T$

1. $\alpha \to \beta$: $A[i] = B[i] - T/2$
2. $solve(A) = A[i] + A[j] = 0$
3. $S_\beta \to S_\alpha$: $(B[i] - T/2) + (B[j] - T/2) = 0 \Rightarrow B[i] + B[j] = T$

## poly-time

- $p(n) = O(n^c)$ where $c \leq 3$ (typically)
- refers to runtime in relation to the length of encoding of the problem instance
  - looking at binary representation (i.e. how many bits to store the input)
  - integers (`n`) → number of bits (usually $\log(n)$)
  - lists (`m`) → number of bits for max entry * number of entries

### pseudo-polynomial algorithms

- runs in polynomial time for input value
- does not run in polynomial time for input length (e.g. exponential)
- knapsack vs fractional knapsack
  - input length: $(n \log M + \log W)$ where $M$ is the maximum value of the knapsack
  - knapsack runtime: $O(n2^{\log(W)})$ which is not polynomial to input length
  - fractional knapsack: $O(n \log n + n)$ which is polynomial in input length
    - $W$ does not matter since we're just using all $n$

## polynomial $p(n)$ time reduction

- let $n$ be the size of $\alpha$
- $\alpha \to \beta$ can be constructed in $p(n)$ time
- $S_\beta \to S_\alpha$ can be recovered in $p(n)$ time
- there is a $p(n)$-time reduction from $A$ to $B$ ($A \leq_P B$)
  - $p(n) = O(n^c)$ where $c$ is a constant
- ***why it is important?***
  - if there is a $p(n)$-time reduction from $A$ to $B$ and there exists a $T(n)$-time algorithm to solve $B$ on instances of size $n$, then there is a $T(p(n)) + O(p(n))$ time algorithm to solve $A$ on instances of size $n$
  - $T(p(n))$ → solving $\beta \in B$
  - $O(p(n))$ → converting $\alpha \to \beta$ and $S_\beta \to S_\alpha$
  - implications:
    - if B has a poly-time algorithm, then so does A



    - if B is easily solvable, then so is A
    - if A is hard to solve, then so is B
- **proof:** $A \leq_P B$
  1. show how to convert an instance of A to an instance of B (how to convert the problem)
  2. show that $\alpha \in A \Leftrightarrow \beta \in B$ (positive instance of A maps to positive instance of B and negative instance of A maps to negative instance of B)
  3. show that both operations can be performed in poly-time

## decision problems

- function that maps an instance space $I$ to the solution set $\{YES, NO\}$
- decision vs optimization
  - decision: does a solution with $k$ constraint exist?
  - optimization: what is the solution?
- decision → optimization
  - given an instance of the optimization problem and a number $k$, does a solution with value $\leq / \geq k$ exist?
    - minimization problem: $\leq k$
    - maximization problem: $\geq k$
  - decision problems are a special case of optimization problems
  - no harder than optimization, so if decision cannot be solved quickly, optimization cannot be solved quickly

### karp reduction

- given 2 decision problems $A$ and $B$, a poly-time reduction from $A \leq_P B$ is a transformation from instance $\alpha \in A \to \beta \in B$ such that
  - $\alpha$ is a YES-instance of A iff $\beta$ is a YES-instance of B
  - transformation takes poly-time in the size of $\alpha$

$$w \in A \iff f(w) \in B$$

- ***show:***
  - reduction takes poly-time
  - if $\alpha$ is a YES-instance of A, then $\beta$ is a YES-instance of B
  - if $\beta$ is a YES-instance of B, then $\alpha$ is a YES-instance of A



> 📌 given an instance $\alpha$ of A, goal is to construct an instance $\beta$ of B such that $\beta$ is YES-instance of B iff $\alpha$ is a YES-instance of A

## ▼ vertex cover $\leq_P$ independent set

### vertex cover (VC)

- given an undirected graph $G = (V, E)$, $X \subseteq V$ is a vertex cover if $\forall u \in E, (\forall v \in E, ((u \in X \vee v \in X) \wedge \sim (u \in X \wedge v \in X)))$
- optimization: compute VC of smallest size
- decision: does there exist a VC of size $\leq k$

### independent set (IS)

- given an undirected graph $G = (V, E)$, $X \subseteq V$ is an independent set if $\forall u \in X, (\forall v \in X, ((u, v) \not\in E))$
- optimization: compute the IS of largest size
- decision: does there exist an IS of size $\geq k$

> 📌 show that $X \subseteq V$ is a vertex cover of $G$ iff $V \setminus X$ is an independent set of $G$

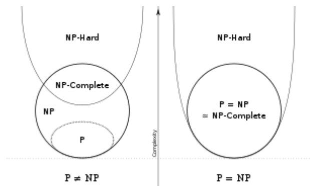
- (→) $X \in VC \Rightarrow (V \setminus X) \in IS$
  - suppose $X \in VC, \forall u \in E, (\forall v \in E, ((u \in X \vee v \in X))$
  - let $Y = V \setminus X$
  - $\forall u \in Y, (\forall v \in Y, ((u, v) \not\in E))$ by definition of IS
  - proof by contradiction: suppose $(u, v) \in E$, then $X \not\in VC$ by definition of VC
  - therefore, $Y \in IS$
- (←) $(V \setminus X) \in IS \Rightarrow X \in VC$
  - let $Y = V \setminus X$
  - suppose $Y \in IS, \forall u \in Y, (\forall v \in Y, ((u, v) \not\in E))$ by definition of IS
  - at most 1 of $\{u, v\} \in X$, therefore, at least 1 of $\{u, v\} \in Y$ which means edge $(u, v)$ is covered by $Y$
  - therefore, $X \in VC$

> 📌 show that the reduction $f$ takes poly-time

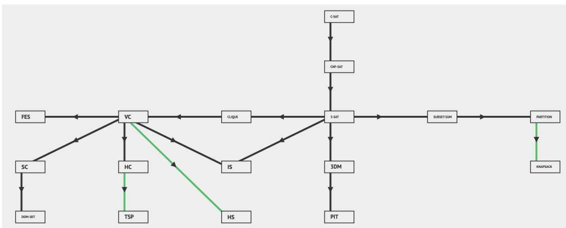
- given input $(G, k)$, $f$ returns $(G, n - k)$ which means it takes poly-time



# NP completeness




- NP (non-deterministic polynomial time) class: set of decision problems where the time to come up with a solution but is easy to verify the certificate (input); i.e. have an efficient certifier
  - efficient certifier: poly-time algorithm with output $\{YES, NO\}$ with respect to certificate $s$
  - e.g. hamiltonian-cycle: easy to verify the certificate (cycle) if it visits each vertex at most once




- P class: set of all decision problems that have an efficient poly-time algorithm
  - $P \subseteq NP$
  - the efficient certifier of $P$ is just the solution of $P$ solved in poly-time
- NP-complete: $X \in NP$ is NP-complete if $\forall A \in NP, A \leq_P X$ (i.e. all other problems in NP reduces to X, is a special case of X)
  - makes $X$ the hardest problem in NP since everything reduces to it
  - ***show:***
    - $X \in NP$

- given some NP-complete problem $Y$, show that $Y \leq_P X$

## dealing with NP-complete problems

- solve smaller instances optimally using exponential time
- check if problem instance has special features that make it more efficiently solvable
- design approximation algorithm

## circuit satisfiability

- given a DAG with nodes corresopnding to AND, NOT, OR gates and $n$ binary inputs, does there exist any binary inputs which gives output `1`
- NP-complete
- satisfiability (CNF-SAT)
  - given a conjunctive normal form (formula $\Phi$ that is a conjunction of clauses $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ where $C_i = x_1 \vee x_2 \vee \cdots \vee x_m$), does it have a satisfying truth assignment
- 3-SAT where each clause contains exactly 3 literals

Circuit satisfiability $\leq_P$ CNF-SAT $\leq_P$ 3-SAT

- fastest algorithm: $1.308^n$ for finding the assignment
  - exponential time hypothesis: no $2^{o(n)}$-time algorithm exists

## ▼ 3-SAT $\leq_P$ IS

- given an instance $\Phi$ of 3-SAT, construct an instance of $(G, k)$ of IS such that $G$ has an independent set iff $\Phi$ is satisfiable

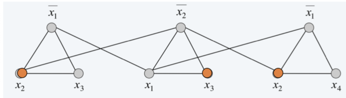
**biggest property of n-SAT**

if an edge is drawn between a literal and its negation, then only one vertex of that edge can be chosen at a time

- reduction: let $G$ have 3 vertices per clause, with 1 per literal
  - connect 3 literals in a clause in a triangle
  - connect each literal to each of its negations
  - $k$ = number of clauses
- (←) suppose $\Phi$ is a YES-instance



  - take any satisfying assignment for $\Phi$ and select a true literal from each clause, these $k$ vertices form an IS of $G$



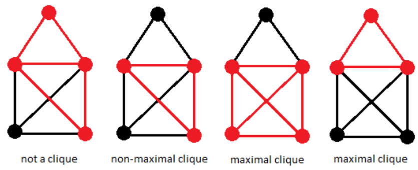
$(\overline{x_1} \vee x_2 \vee x_3)$
$\wedge (x_1 \vee \overline{x_2} \vee x_3)$
$\wedge (\overline{x_1} \vee x_2 \vee x_4)$

- (→) suppose $(G, k)$ is a YES-instance
  - let $S$ be the IS of size $k$
  - each of the $k$ triangles must contain exactly one vertex in $S$ and use these literals as true

3-SAT $\leq_P$ IS $\leq_P$ VC
IS
$\leq_P$ Max-Clique

## ▼ IS $\leq_P$ Max-Clique

- given an undirected graph $G$ and an integer $k$, is there a clique of size at least $k$ or not in $G$?
  - a **clique**, $C$, in an undirected graph $G = (V, E)$ is a subset of the vertices, $C \subseteq V$, such that every two distinct vertices are adjacent
    - everything is one-edge away from each other



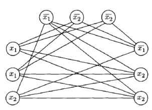

- in layman terms, an IS is one where every vertex in the set does not have an edge between them; a clique is one where every vertex in the set has an edge between them
- therefore, an IS is a clique in the complement of the graph $G$
- given $(G, k)$, create $(G^C, k)$
  - $G$ has an independent set of size $k$ iff $G^C$ has a clique of size $k$



## ▼ 3-SAT $\leq_P$ MAX-CLIQUE

- given $\Phi$ with $m$ clauses over $n$ variables
- nodes of $G$ organized into $k$ groups of 3, each corresponding to a clause $C_i$ and each node corresponds to a literal in $C_i$
- edges of $G$ connect all nodes but two types of pairs (*exceptions*):
  1. nodes in the same triple
  2. nodes that are complementary

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2).$$



- proof: boolean formula is satisfiable iff $G$ has a k-clique
  - (→) boolean formula is satisfiable implies $G$ has a k-clique
    - suppose the boolean formula is satisfiable
    - at least 1 literal is true in every clause → in each $C_i$ in the graph, select 1 node corresponding to a true literal in the assignment
      - if more than 1 literal can be chosen, choose arbitrarily
    - nodes selected form a k-clique
    - each pair of nodes is joined by an edge as no pair fits one of the *exceptions*, they cannot be from the same clause, and they cannot have contradictory labels (i.e. $x_i$ and $\bar{x}_i$)
    - therefore, $G$ has a k-clique
  - (←) $G$ has a k-clique implies boolean formula is satisfiable
    - suppose $G$ has a k-clique
    - no two of the clique's nodes occur in the same clause since the nodes of the same clause aren't connected by edges
    - each of the clauses contains exactly one of the k-clique nodes
    - assign truth values to variables such that each literal in a clique node is true
    - assignment satisfies $\Phi$ since each clause has a clique node and each clause contains a literal that is assigned true
    - therefore, $\Phi$ is satisfiable



## ▼ VC $\leq_P$ Hitting Set

- hitting set: for a set of sets $\{S_1, S_2, \ldots, S_n\}$, set $H$ is a hitting set if $\forall S_i \in S, H \cap S_i \neq \emptyset$ and the hitting set problem states that given the set of sets and $k$, decide if there exists a hitting set of size at most $k$
- hitting set is in NP: hitting set of size at most $k$ is the certificate, size of certificate is polynomial, verification is also polynomial
- transformation: given an instance $(G, k) \in VC, \forall e \in E, S_e = \{u, v\}$ (create a set for every edge)
  - instance of hitting set: $\{S_e | e \in E\}, k$
- proof: $X \subseteq V$ is a VC of $G$ of size at most $k$ iff $\{S_e | e \in E\}, k$ is a hitting set of at most $k$
  - (→) $X \subseteq V \in VC \Rightarrow \{S_e | e \in E\}, k$ is a hitting set of at most $k$
    - suppose $X$ is a VC of $G$
    - $\forall u \in V, (\forall v \in V, ((u \in X \vee v \in X) \wedge \sim (u \in X \wedge v \in X)))$ by definition of VC
    - $\forall (u, v) \in E$, at most 1 vertex is found in $X$ by definition of VC
    - $\therefore$ every $x \in X$ is a vertex from each set of edges in the hitting set transformation
    - every vertex is an element that would appear in the hitting set
    - given that $X$ is at most $k$ size, then there are at most $k$ elements in the hitting set, assuming that there are no duplicates
      - if there are duplicates, then the hitting set has less than $k$ elements
  - (←) $\{S_e | e \in E\}, k$ is a hitting set of at most $k$ size $\Rightarrow X \subseteq V \in VC$
    - suppose $\{S_e | e \in E\}, k$ is a hitting set of at most $k$ size
    - notice that the elements in the hitting set correspond to the nodes in the VC

## order statistics

- given an unsorted list, find the `i`-th smallest element in the list, where $i \in [1, n]$
  - naive solution: sort and take the `i`-th index: $\Theta(n \log n)$
- lower bound: $\Omega(n)$
  - in $< n$ steps, it looks at at most $n - 1$ elements which results in some element $x_i$ not being seen which means it cannot be determined that `i`-th element is the smallest
- assumption: all elements are distinct otherwise use relative position to distinguish: `(A[i], i)`
- worst-case linear time order statistic:

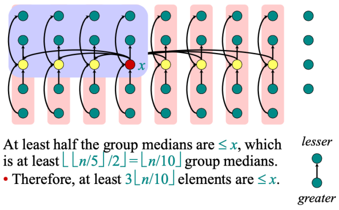
Select ($i, n, A$)

1. Divide the array A into $\lceil \frac{n}{5} \rceil$ groups of 5 elements each
2. Let $B$ be the set of $\lceil \frac{n}{5} \rceil$ elements of the medians of each of the above groups
3. Recursively find the median $x$ of $B$ (that is, Select(|B|/2, |B|, B)).
4. Partition $A$ around pivot $x$. Let $k$ be the rank of $x$, $A'$ and $A''$ be the list of elements $< x$ and $> x$ respectively.
5. If $i = k$, then return $x$
6. Else, if $i < k$, then return $\text{Select}(i, k-1, A')$
7. Else, if $i > k$, then return $\text{Select}(i-k, n-k, A'')$

End

- time complexity: $T(n) \leq T(n/5) + T(7n/10) + c_1 n \in \Theta(n)$
  1. $O(n)$
  2. $O(n)$
  3. $O(n/5)$
  4. $O(n)$
  5. $O(n)$
  6. (and 7) $O(7n/10)$
     a. at most $n/10$ group medians are $\leq x$ just by definition of how we divided and partitioned
        i. at least half the groups are $\leq x$ so it's $(n/5)/2$
        ii. within each half group, it has 3 elements at most so we have to multiply by 3
        iii. these form the elements that don't need to be recursed on
        iv. then, we want to know how many we actually need to recurse on which is just $n - k$ where $k$ is the number of elements that are on the other side of the clause
     b. so recursive calls are done on at least $n - 3 \cdot (n/10)$ that are $\leq x$

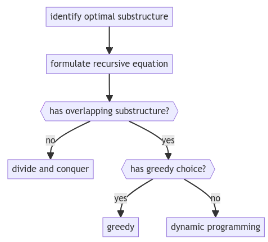





blue section are the nodes that are at most x and

# general revision

## solving optimization problems




## shortest path in graph

- input: given a directed graph $G = (V, E)$ with the values of $E$ being $\omega : E \to R$ (i.e. both positive and negative numbers) and source vertex $s \in V$ with $n = |V|, m = |E|$
- goal: $\forall v \in V \setminus \{s\}, \delta(s, v)$ where $\delta(u, v)$ is the distance from $u$ to $v$ OR $\forall v \in V \setminus \{s\}, P(u, v)$ where $P(u, v)$ is the shortest path from $u$ to $v$

### dijkstra's algorithm

- if $\omega : E \to R^+$ (i.e. no negative edge weights), then Dijkstra solves the problem in $O(m + n \log n)$ using Fibonacci heap
  - non-negative weights are necessary, otherwise the invariant (once a node is visited, it has the shortest path to reach it) would be violated as a visited node could have a decreased cost even after being visited



- both facts are violated if negative weights are allowed
  - fact 1: nearest neighbor is also the vertex nearest to $s$ (so it always has the shortest distance)
    - if negative weights are allowed, then there's no way to know that $\delta(s, v)$ is the shortest path
  - fact 2: optimal subpath property
    - every subpath of a shortest path is also a shortest path, otherwise, replacing the subpath by the shortest path will reduce the larger path cost
- **optimal substructure property:** given a directed $G = (V, E)$ with $\omega : E \to R$, all shortest paths in $G$ possess optimal substructure property if there is no negative cycle
  - $L(v, i)$ is the weight of the shortest path from $s \to v$ having at most $i$ edges

$$L(v, i) = \begin{cases} 0 \text{ if } v = s \wedge i = 1 \\ \infty \text{ if } i = 1 \wedge (s, v) \notin E \\ \omega(s, v) \text{ if } i = 1 \wedge (s, v) \in E \\ L(v, i-1) \text{ if } < i \text{ edges} \\ \min_{(x,v) \in E}\{L(x, i-1) + \omega(x, v)\} \end{cases}$$

### bellman-ford algorithm

- if $\omega : E \to R$, then Bellman Ford is a better option, takes $O(nm)$
  - iterate over the number of edges and relax all vertices once

```
Bellman-Ford-algo(s,G)
{
  For each v ∈ V\{s} do
      If (s,v) ∈ E  then L[v,1]← ω(s,v)
                    else L[v,1]← ∞;
  L[s,1]← 0;
  For i = 2 to n − 1 do
  {   For each v ∈ V do
      {
        L[v,i]←L[v,i − 1];
        For each (x,v) ∈ E do
            L[v,i]← min( L[v,i] , L[x,i − 1] + ω(x,v) )
      }
  }
}
```

Initializing $L[*, 1]$

Computing $L[v, i]$

- uses $L(v, i)$ computed earlier
  - uses $O(n^2)$ space but can be reduced to $O(n)$ since we're only looking back at $L(x, i-1)$ so we can do a 1-state caching with only storing the past state to compute the present state
  - the shortest path extracted by storing the "parent" of each node in the DP state
  - negative weight cycles can be detected by performing another round of relaxation (after `n - 1` of them) and then seeing if the costs go down anymore, if they do, then a negative weight cycle is found



## minimum spanning tree

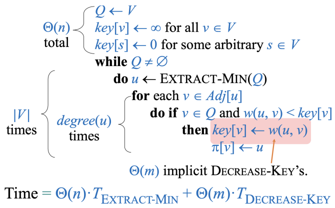
- input: connected, undirected $G = (V, E)$ with weight function $\omega : E \to R$ (assumed to be distinct)
- output: spanning tree $T$ that connects all vertices of minimum weight

$$\omega(T) = \sum_{(u,v) \in T} \omega(u, v)$$

- **optimal substructure:** remove any edge $(u, v) \in T$ and $T$ is partitioned into 2 subtrees $T_1$ and $T_2$
  - $T1$ and $T_2$ are both the MST of the sub-graphs
  - proof:
    - let $\omega(T) = w(u, v) + \omega(T_1) + \omega(T_2)$
    - if $\omega(T_1') < \omega(T_1)$, then $T' = \{(u, v)\} \cup T_1' \cup T_2'$ would be a lower weight spanning tree than $T$
- **greedy choice:** let $T$ be a MST of $G = (V, E)$ and $A \subseteq V$. suppose $(u, v) \in E$ is the least-weight edge connecting $A$ to $V \setminus A$, then $(u, v) \in T$
  - proof:
    - suppose $(u, v) \notin T$
    - let $P(u, v)$ be the unique path from $u$ to $v$ in $T$
    - swap $(u, v)$ with the first edge in $P(u, v)$ that connects $A$ to $V \setminus A$
    - the resulting $\omega(P(u, v)') < \omega(P(u, v))$

### prim's algorithm

- maintain $V \setminus A$ as a priority queue $Q$ and key each vertex in $Q$ with the weight of the least-weight edge connecting it to a vertex in $A$
  - think of this algorithm as building from a single vertex outwards, considering all edges connecting to the currently built tree
  - $\{(v, \pi[v])\}$ is the MST
  - $m$ is the number of edges




📌 note that although the operations are nested, they effectively run at most $\Theta(n)$ and $\Theta(m)$ times for each operation so we sum those instead. don't assume nested → multiply all probability

| $Q$ | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total |
|---|---|---|---|
| Array | $O(n)$ | $O(1)$ | $O(n^2)$ |
| Binary heap | $O(\lg n)$ | $O(\lg n)$ | $O(m \lg n)$ |
| Fibonacci heap | $O(\lg n)$ amortized | $O(1)$ amortized | $O(m + n \lg n)$ worst case |

# math revision

## exponentials

- $a^{-1} = \frac{1}{a}$
- $(a^m)^n = a^{mn}$
- $a^m \times a^n = a^{m+n}$
- $e^x \geq 1 + x$
- exponentials of different bases differ by an exponential factor (cannot be ignored)

📌 any exponential function with base $a > 1$ grows faster than any polynomial

**lemma:** for any constants $k > 0$ and $a > 1$, $n^k = o(a^n)$

## logarithms



- binary log: $\lg n = \log_2 n$
- natural log: $\ln n = \log_e n$
- exponentiation: $\lg^k n = (\lg n)^k$
- composition: $\lg \lg n = \lg(\lg n)$
- $a = b^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c b$
- $\log_b a^n = n \log_b a$
- $\log_b a = \dfrac{\log_c a}{\log_c b}$
- $\log_b(1/a) = -\log_b a$
- $\log_b a = \dfrac{1}{\log_a b}$
- $a^{\log_b c} = c^{\log_b a}$
- base of logarithm does not matter in asymptotic analysis

## Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

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

## summations

- arithmetic series
  - $a_n = a_1 + (n-1) \times d$
  - $S_n = \frac{n}{2}(2a_1 + (n-1) \times d) = \frac{n \times (a_1 + a_n)}{2} \in \Theta(n^2)$
- geometric series
  - $g_n = g_1 \times r^{n-1}$
  - $S_n = \dfrac{a(1-r^n)}{1-r}$
  - $S_\infty = \dfrac{a}{1-r}$ when $|x| < 1$
- harmonic series
  - $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n = \sum_{k=1}^{n} 1/k = \ln n + O(1)$

# common algorithms



## fibonacci

$$F_n = F_{n-1} + F_{n-2}$$
$$F_{2n-1} = F_n^2 + F_{n-1}^2$$
$$F_{2n} = F_n(F_{n-1} + F_{n+1})$$

### recursive

```
def fib(n):
    if n == 0: return 0
    if n == 1: return 1
    return fib(n - 1) + fib(n - 2)
```

$T(n) = T(n-1) + T(n-2) + O(1)$

$T(n) \in O(2^n)$

more precisely: $O(\phi^n)$

### iterative

```
def fib(n):
    if n == 0: return 0
    if n == 1: return 1
    p2, p1 = 0, 1
    for i in range(2, n + 1):
        p2, p1 = p1, p1 + p2
    return p1
```

$T(n) \approx 5n$

### using matrix multiplication

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

## merge sort

```
def merge_sort(arr):
    if len(arr) == 1: return arr
    left = merge_sort(arr[:len(arr)//2])
    right = merge_sort(arr[len(arr)//2:])
    return merge(left, right)
```

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n), n > 1 \\ \Theta(1), n = 1 \end{cases}$$

## powering a number

- find $F(a, n) = a^n$ (may use $a^n \% m$ to avoid large numbers)



$$F(a, n) = \begin{cases} F(a, \lfloor \frac{n}{2} \rfloor)^2, n\&1 = 0 \\ F(a, \lfloor \frac{n}{2} \rfloor)^2 * F(a, 1), n\&1 = 1 \end{cases}$$

- $T(n) = T(n/2) + \Theta(1) \in \Theta(\log n)$
- used to calculate fibonacci numbers in $\Theta(\log n)$ time

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \psi^n)$$

## matrix multiplication

### naive $\Theta(n^3)$

```
# Outer left dimension
for i in range(n):
    # Outer right dimension
    for j in range(n):
        # Inner dimension
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

### Strassen's method $\Theta(n^{log_2 7})$

- leverages the fact that matrix addition is faster than matrix multiplication
- divide matrix into quadrants of size $n/2 \times n/2$

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

- original equations: $T(n) = 8T(n/2) + \Theta(n^2)$
  - $r = ae + bg$
  - $s = af + bh$
  - $t = ce + dg$
  - $u = cf + dh$
- Strassen's equations: $T(n) = 7T(n/2) + \Theta(n^2)$ with 1 less matrix multiplication
  - $P_1 = a \times (f - h)$
  - $P_2 = (a + b) \times h$
  - $P_3 = (c + d) \times e$
  - $P_4 = d \times (g - e)$
  - $P_5 = (a + d) \times (e + h)$

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

## longest increasing subsequence

- dynamic programming approach: $O(n^2)$

$$dp(i) = \max_{j \in [0, i-1]} \left\{ \begin{cases} 1, \text{ if } nums[j] \geq nums[i] \\ dp(j) + 1, \text{ if } nums[j] < nums[i] \end{cases} \right\}$$

  - alternative is find the LCS between sorted numbers and numbers
- greedy approach — patience solitaire: $O(n \log n)$
  - for each number $nums[i]$, (a) place it on the leftmost pile where the top card is greater than $nums[i]$ or (b) create a new pile with $nums[i]$ as the top
  - the topmost card is always in ascending order
  - to find the leftmost pile, use binary search $O(\log n)$
  - linear search through the entire pile: $O(n)$
  - LIS is the number of piles formed