Big O Notation

Runtime - the number of computer steps the algorithm takes as a function of the size of the input

Keep only largest term, omit constants

Polynomial → efficient
Exponential → intractable

O → upper bound
Ω → lower bound
Θ → same rate

Proving Asymptotic Relations

if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, $f(n) = O(g(n))$

if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, $f(n) = \Theta(g(n))$ ($c > 0$)

if $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, $f(n) = \Omega(g(n))$

Recurrence Relations

- function for amount of working needed to solve a problem of size n using smaller subproblems.

$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = O(n^2)$

$a + ar + ar^2 + \ldots + ar^{n-1} = \frac{a(1-r^n)}{1-r}$

$a + ar + ar^2 + \ldots = \frac{a}{1-r}$

for a series $O(1) + O(2) + \ldots + O(f(n))$, runtime is $O(f(n))$.

Divide and Conquer

1. Divide: break large problem into smaller subproblems
2. Conquer: recursively solve each subproblem
3. Combine: combine the answers of the subproblem

Master Theorem

if relation is of the form $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Graphs

- a set of vertices and directed/undirected edges

DFS: graph traversal that visits all nodes down one path before moving on to another path.

Preorder - order by time when node is first visited

Postorder - order by time when node is done exploring (including all child nodes)

Runtime: $O(|V| + |E|)$

Edge Types

tree edge: leads to a child, part of the DFS tree
forward edge: leads to a non child descendant
back edge: leads to an ancestor
cross edge: leads to a node that is neither a descendant nor an ancestor

DFS uses

- path between 2 vertices
- find connected components
- if graph has a cycle
- topological sort of graph

Topological Sort

- linear ordering of vertices such that for every edge (u, v), vertex u comes before vertex v.

Works only on directed, acyclic graphs

Breadth First Search $O(V+E)$

- Graph traversal that visits all nodes level by level.

UNWEIGHTED

Strongly Connected Components

- Subset of vertices in which there is a path from every vertex to every other vertex (cycle)

how to find? kosaraju's algorithm

1. reverse all edges in graph G to create G^R
2. run DFS and keep track of post order values
3. run DFS on G starting with next available vertex with highest post order value

Repeat until all vertices have been visited.

Dijkstra's Algorithm (weighted +)

- Finds shortest path from a starting vertex to all vertices in graph.

1. Insert all vertices into PQ, sorting vertices in order of distance from source.
   - Initially, source distance is 0, others are ∞
2. Repeat until PQ is empty
   - remove closest/smallest vertex from PQ and relax all edges pointing from v.
   - update distance to neighboring vertices u if $dist[v] + weight(u,v) < dist[u]$

Runtime: inserting $|V|$ times + deleting min $|V|$ times and updating $|E|$ times.

	Ins	Del	Upd	Tot
Array	$O(1)$	$O(V)$	$O(1)$	$O(V^2+E)$
Bin. Heap	$O(\log V)$	$O(\log V)$	$O(\log V)$	$O((V+E)\log V)$

Greedy Algorithms

- Always choose the locally optimal decision (what looks best at the time).
- Prove using exchange argument: Show that the greedy choice is at least as good as any other choice.

Exchange Arguments

1. Let G be greedy output and O be optimal output
2. Find first place where G and O disagree. G picks g, O picks o.
3. Modify O to include g by removing o if needed, swapping, etc.
4. Show all constraints still hold and value of modified O ≥ value of original O.
5. Therefore, the new O is optimal also. You can repeat for every differing job between O and G. Eventually, O will be modified into G and be optimal so G is also optimal.

Dynamic Programming

- Reuse information that has already been computed and store it so we do not need to compute it again

1. Define the subproblem
2. Identify the base case
3. Find the recurrence relation
4. Construct the order to solve the subproblems
   - Bottom up approach: Start from base case and build up to n
   - Top down approach: Start from n and recursively call down to base case
5. Find the final solution
6. Solve for runtime and space complexity
   - Time = O(# subproblems · time for 1 subproblem)
   - Space = O(# subproblems)
7. Prove correctness with induction.

Graph Example

Given graph $G=(V,E)$ and vertex s, let $p(v)$ = # of shortest paths from s to v, $p(s)=1$. Design algorithm to find $p(v)$ for every vertex, given G is undirected and unweighted.

Use BFS. When visiting edge $u \to v$, if v distance is ∞ set to dist[u]+1, set #paths[v] += #paths[u]. Append all unseen neighbors of u to queue.

Bellman Ford

- For possible negative edge weights

Init dist[s]=0 and all others to ∞.

Repeat V-1 times: Relax edge (u,v) by setting dist[v] = min(dist[v], dist[u] + w(u,v))
Have to do V-1 passes since initially, only source was correct. After 1 pass, first edges are correct, so continue pattern until V-1 (last) edge.

Runtime: O(VE)

DP Example 1

Knapsack with cap W and n types of items. Item i has weight w_i, value v_i.
Find max total value.

1. Subproblem
   DP(i, w) = max val with items 1 to i with cap W
2. Recurrence Relation

$$DP(i,w) = \max \begin{cases} DP(i-1, w) & \text{don't ever use } i \text{ again} \\ DP(i, w-w_i) + v_i & \text{use item } i \end{cases}$$

3. base case
   $DP(0,w)=0 \quad DP(i,0)=0$
4. Runtime
   # subproblems = O(nW)
   time per subproblem = O(1)
   Overall = O(nW)

DP Example 2

n spots for trees, x_i apples at each spot. If tree at i, cannot tree at i-1 or i+1.
Find max # of apples

1. Subproblem
   DP(i) = max # apples using spots up to i.
2. Recurrence Relation

$$DP(i) = \max \begin{cases} DP(i-2) + x_i & \text{plant at } i \\ DP(i-1) & \text{don't plant at } i \end{cases}$$

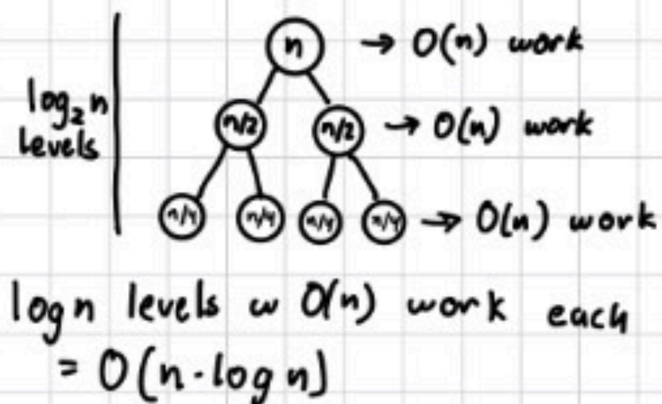
3. Base case
   $DP(0) = x_0 \quad DP(1) = \max(x_0, x_1)$
4. Runtime
   # Subproblems = O(n)
   Time per subproblem = O(1)
   Overall = O(n)

D&C Example

Array A[0, ..., n-1] where n>1 and all pos ints. Find max val of $A[i] + A[j]^2$ s.t. $0 \le i < j < n$

1. Divide: Split array in half
2. Conquer: Recurse to find max val in L and R (ℓ and r)
3. Combine: Find max crossing val
   $z = \max(L) + \max(R)^2$

   return max(ℓ, r, z)
4. Runtime: (check max of each half)
   $T(n) = 2T(\frac{n}{2}) + O(n)$ (2 recursive calls)



log n levels w O(n) work each
= O(n · log n)

Backpropagation

Suppose a function $L = L(x_1, x_2, \ldots, x_d)$ that depends on many variables and we want to get all partial derivatives.

Idea: Represent L as a directed, acyclic graph.

Source nodes (in-degree 0) are inputs $(x_1, x_2, \ldots, x_d)$ and constants.

Internal nodes store the result of the operations. For example, a node might have $+$ and a, b as inputs and an addition operation so it stores $c = a+b$ which it passes through its output edge.

Output node (out-degree 0) represents the output L.

If there is an edge $u \to v$, u is a child of v. The source nodes are leaves, the output node is root.

Forward Pass - Compute the value of each node from the inputs to the output, following topological order.

Cost: $O(|V|+|E|)$, each node is visited once, each edge is traversed once.

Backward Pass - Compute the gradient $\frac{\partial L}{\partial u_i}$ for each node u_i in reverse topological order.

Initialize $\frac{\partial L}{\partial u_n} = 1$, the output node's partial derivative

For $i = n-1, n-2, \ldots, 1$: $\frac{\partial L}{\partial u_i} = \sum_{u_j \in \text{parents}(u_i)} \frac{\partial L}{\partial u_j} \cdot \frac{\partial u_j}{\partial u_i}$

After the pass, each node stores the correct partial derivative.

Cost: $O(|V|+|E|)$, each node is visited once and each edge is traversed once.

Fast Fourier Transforms

Idea: Multiply two polynomials together quickly

A degree d polynomial can be determined by its values at $d+1$ points. If you know what the polynomial evaluates to at $d+1$ distinct points, you can find out the polynomial.

For multiplication: If you want $C(x) = A(x) \cdot B(x)$ and you already know $A(x_i)$ and $B(x_i)$ at a bunch of points x_i, you can calculate $C(x_i) = A(x_i) \cdot B(x_i)$

1. Pick points $x_0, \ldots, x_{2d}$ that will construct $C(x)$
2. Evaluate $A(x)$ and $B(x)$ at these points
3. Multiply the values to get $C(x_i) = A(x_i) \cdot B(x_i)$
4. Interpolate the $2d$ points to find the coefficients of $C(x)$

Nth Roots of Unity - the n complex numbers that, when raised to the nth power, equal 1. They are:

$$\omega^k = e^{2\pi i k/n} \quad \text{for } k = 0, 1, \ldots, n-1$$

The squares of the nth roots are just the $(n/2)$th roots: $(\omega^k)^2 = e^{2\pi i k \cdot 2/n} = e^{2\pi i k/(n/2)}$

So squaring the n points gives you $n/2$ distinct points.

When you split A into even and odd coefficients:

$$A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$$

$FFT(A(x), n)$ evaluates $A(x)$ at the nth roots of unity.

Polynomial Multiplication: d is degree of $A/B(x)$

1. Pick the nth roots of unity to construct $C(x)$ where d is the degree of the product. Round up to nearest power of two.
2. Evaluate $FFT(A(x), n)$ and $FFT(B(x), n)$ to get values at nth roots of unity.
3. Multiply corresponding pairs of values to get value representation of $C(x)$
4. Use inverse $FFT = \frac{1}{n} FFT(C(x) \text{ values}, \omega^{-1})$ to get coefficients. (↑ normalize; ↖ same FFT; ↖ $\omega = e^{2\pi i/n}$)

Runtime: $O(n \log n) + O(n) + O(n \log n) = O(n \log n)$ (Step 2, Step 3, Step 4)

Algorithm	Work (T_1)	Span (T_∞)	Idea
Parallel Prefix (Scan)	$O(n)$	$O(\log n)$	Associativity
Carry-Lookahead Add	$O(n)$	$O(\log n)$	Linear recurrence → prefix
Integer Multiplication	$O(n^2)$	$O(\log n)$	3-for-2 + scan

Parallel Algorithms

Model parallel algorithms as directed acyclic graphs.

Nodes - represent a unit of computation

Edges - represent a dependency, which is a constraint that forces one computation to come before another.

Let T_p = time to finish the computation on p processors.

Work $= T_1$ = # of vertices in the DAG.
- 1 processor, no parallelism
- $T_p \geq Work/p = T_1/p$, best case scenario.

Span $= T_\infty$ = length of the longest directed path in the DAG (critical path)
- count edges for length.
- cannot do better than T_∞

Speedup $(S_p) = \frac{T_1}{T_p}$. Max speedup is p

Efficiency $(E_p) = \frac{T_1}{pT_p}$. Measures how well each processor is utilized. Perfect efficiency $= 1$, $E_p \leq 1$

Parallelism $= \frac{T_1}{T_\infty}$. High parallelism, processors can stay busy. Low parallelism means the algorithm is mostly sequential.

Brent's Rule - Runtime: $T_p \leq \frac{T_1}{p} + T_\infty$

3-for-2 Trick - Carry Save

Three n bit numbers X, Y, Z. We want to reduce them to two numbers S and C s.t. $S + C = X + Y + Z$, in constant time on n processors.

Take bit position k: add x_k, y_k, z_k. sum is between 0 and 3 in format $c_k s_k$.

$s_k = x_k \oplus y_k \oplus z_k$ (XOR - odd number of 1s = 1, even = 0)

$c_k = \text{majority}(x_k, y_k, z_k)$, 1 iff 2 out of 3 are 1s.

s_k is placed at position k of S and c_k is placed at $k+1$ of C.

Integer Multiplication

1. Compute partial products (get to the step of needing to sum everything).
2. Group the n rows into triples and apply the 3-for-2 trick to each triple simultaneously. Repeat. Each round the # of rows shrinks by a factor of 2/3. After $O(\log n)$ rounds, you have exactly two rows.
3. Add the 2 rows using carry lookahead, which takes $O(\log n)$ time.

Total Span $= O(\log n)$ Total Work $= O(n^2)$

Parallel Prefix (Scan) - Given a, b, c, d: outputs $a, a{+}b, a{+}b{+}c, \ldots$

Takes a sequence and an associative binary operator (addition, multiplication, logical AND/OR, minimum, maximum, GCD, matrix multiplication, etc.)

$$y_k = x_1 \circ x_2 \circ \ldots \circ x_k, \quad k = 1, 2, \ldots, n$$

1. Reduce - Pair up adjacent elements and compute $z_k = x_{2k-1} \circ x_{2k}$ for $k = 1, 2, \ldots, n/2$ where $\circ$ is the operator. This takes $O(1)$ span since all pairs are independent, so the computations can happen on different processors.
2. Recurse - Compute the prefix scan of z recursively. Let y' denote the result: $y'_k = z_1 \circ z_2 \circ \ldots \circ z_k = x_1 \circ x_2 \ldots x_{2k}$

   So $y'_1 = z_1 = x_1 + x_2 = y_2$

   $y'_2 = z_1 + z_2 = x_1 + x_2 + x_3 + x_4 = y_4$

   $y'_3 = z_1 + z_2 + z_3 = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = y_6$

   $\vdots$

   So y'_k is the prefix of the original sequence up to position $2k$. So we can get all even indexed outputs for free.
3. Expand - Recover all outputs

   Even indices: $y_{2k} = y'_k$

   Odd indices: $y_{2k+1} = y'_k \circ x_{2k+1}$

   All odd-indexed outputs can be computed at the same time, so $O(1)$ span.

Work $= O(n)$ Span $= O(\log n)$

$T_1(n) = T_1(n/2) + O(n)$ $T_\infty(n) = T_\infty(n/2) + O(1)$

This algorithm does no more work than the sequential algorithm.

Carry Lookahead Addition

1. Compute g and p for each bit position in parallel ($O(1)$ span)

   $g_i = a_i \text{ AND } b_i$ — 1 if both a_i and b_i are 1

   $p_i = a_i \text{ XOR } b_i$ — 1 if $a_i \neq b_i$
2. Each bit position gets a 2×2 matrix:

   $$M_i = \begin{pmatrix} p_i & g_i \\ 0 & 1 \end{pmatrix}$$

   Use parallel prefix to do matrix multiplication and compute $M_0 = C_1$

   $M_1 \cdot M_0 = C_2$

   $\vdots$

   So you will have all carry bits in $O(\log n)$ span, $O(n)$ work.
3. $s_i = a_i \text{ XOR } b_i \text{ XOR } c_i$

   1 if odd # of 1's, 0 if even.

Linear Programming

Constructing a program that maximizes a linear objective function given a set of linear constraints

1) Define your variables
2) Find constraints and objective function

Form: $\max\ c_1x_1 + c_2x_2 + \ldots + c_nx_n$ — Objective Function

s.t. $Ax \leq b$ — Constraints

$x \geq 0$ — Always needed

Objective Function must maximize. If you want to minimize, negate the function.

Constraints must have $x \leq$ since you are maximizing. If $\geq$, multiply both sides by -1.

Feasible Region - set of points that satisfy all constraints
- feasible: there exists at least one solution that satisfies all constraints
- unfeasible: there is no solution that satisfies all constraints
- unbounded: feasible region is open and extends to infinity in some direction

Dual - combines the constraints to get the upper bound of the objective function.
- Weak duality (always holds): the optimal value of the dual will be an upper bound to the optimal value of the primal.
  - primal objective ≤ dual objective
- Strong duality (if primal is feasible and not unbounded): The dual optimal value equals the primal optimal value.

If primal is maximization, dual is minimization

Dual Form: $\min\ y^Tb$

s.t. $y^TA \geq c^T$ or $A^Ty \geq c$

$y \geq 0$

Simplex Algorithm - solves linear program

1. Start at a vertex
2. For all neighboring vertices:
   - If there exists a neighbor value > current value, repeat step 2 with neighbor.
   - Else, current vertex is optimal value.

Runtime: $O(2^n)$

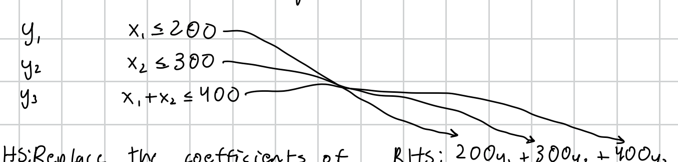
Linear Programming Example

Primal LP: $\max\ x_1 + 6x_2$

s.t. $x_1 \leq 200$, $x_2 \leq 300$, $x_1 + x_2 \leq 400$, $x_1, x_2 \geq 0$

Optimal Solution: $(100, 300) \to 1900$. Solve by Simplex algorithm.

For each constraint, assign a value y_i:



LHS: Replace the coefficients of x_1 and x_2. x_1 is involved in y_1 and y_3. x_2 is involved in y_2 and y_3. Therefore:

RHS: $200y_1 + 300y_2 + 400y_3$

$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3$

$= x_1 + 6x_2$

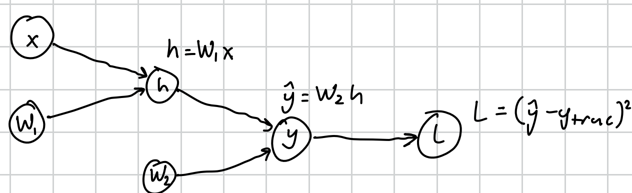
↑ Dual objective function to minimize.

Dual constraints: $y_1 + y_3 \geq 1$, $y_2 + y_3 \geq 6$, $y_1, y_2, y_3 \geq 0$

Backpropagation Example

$x = \begin{pmatrix}1\\1\end{pmatrix}$ $W_1 = \begin{pmatrix}2 & 1\\ 0 & 3\end{pmatrix}$ $W_2 = (2 \; 1)$ $h = W_1 x$ $\hat{y} = W_2 h$ $y_{true} = 15$

Perform forwards then backpropagation to compute $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}$ of $L = (\hat{y} - y_{true})^2$



Forward Pass:

$h = W_1 x = \begin{pmatrix}2 & 1\\ 0 & 3\end{pmatrix}\begin{pmatrix}1\\1\end{pmatrix} = \begin{pmatrix}3\\3\end{pmatrix}$

$\hat{y} = W_2 h = (2 \; 1)\begin{pmatrix}3\\3\end{pmatrix} = 9$

$L = (9-15)^2 = 36$

Back Pass:

$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y_{true}) = 2(9-15) = -12$

$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W_2} = -12 \cdot (h_1 \; h_2) = -12(3 \; 3) = (-36 \; -36)$

$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h} = -12 \cdot W_2 = -12(2 \; 1) = (-24 \; -12)$

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial x} + \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial x} = -24\begin{pmatrix}2\\1\end{pmatrix} - 12\begin{pmatrix}0\\3\end{pmatrix} = \begin{pmatrix}-48\\-60\end{pmatrix}$

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial W_1} + \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial W_1} = -24\begin{pmatrix}1 & 1\\ 0 & 0\end{pmatrix} - 12\begin{pmatrix}0 & 0\\ 1 & 1\end{pmatrix} = \begin{pmatrix}-24 & -24\\ -12 & -12\end{pmatrix}$

Parallel Algorithms Example

Given a string of $2n$ parentheses, determine if it's valid (every open paren is matched to a later closed paren) in $O(n)$ work and $O(\log n)$ time.

① Build an array where '(' is +1 and ')' is -1. You can do this in parallel.

② Compute all prefix sums in parallel using parallel prefix algorithm. This will compute the running sums up to the current position. If any of the prefix sums < 0, there is an unmatched close paren so it is invalid. The final sum must be 0 so all parenthesis are paired.

Work = $O(n)$ since parallel prefix does $O(n)$ work.

Time = Span = $O(\log n)$ since parallel prefix takes $O(\log n)$ time.

Parallel Algorithms Example

Given an array of n numbers, remove all negative numbers and output the remaining elements in their original order.

In parallel, you can identify negative elements easily but it is difficult to place each surviving element.

① Build a binary array P where $P[i] = 1$ if $A[i] \geq 0$, else 0.

② Compute prefix sums of P using parallel prefix

③ The prefix sum at i gives the target index $t(i)$ for each element $A[i]$ in the output array.

④ Copy each non-negative $A[i]$ to output$[t(i)]$ in parallel.

Work = $O(n)$ Time = Span = $O(\log n)$

Linear Programming

Minimize food cost while meeting daily nutritional minimums.

		Protein	Carbs	Fat
Meat	\$5	500	0	500
Bread	\$2	50	300	25
Shakes	\$4	300	100	200

LP: min $5m + 2b + 4s$ ← min cost: cost per item times amt of item.

s.t. $500m + 50b + 300s \geq 500$
$300b + 100s \geq 100$
$500m + 25b + 200s \geq 400$
$m, b, s \geq 0$

Dual:

p $500m + 50b + 300s \geq 500$
c $300b + 100s \geq 100$
f $500m + 25b + 200s \geq 400$

RHS: $500p + 100c + 400f$

LHS: $m(500p + 500f) + b(50p + 300c + 25f) + s(300p + 100c + 200f)$
$= 5m + 2b + 4s$

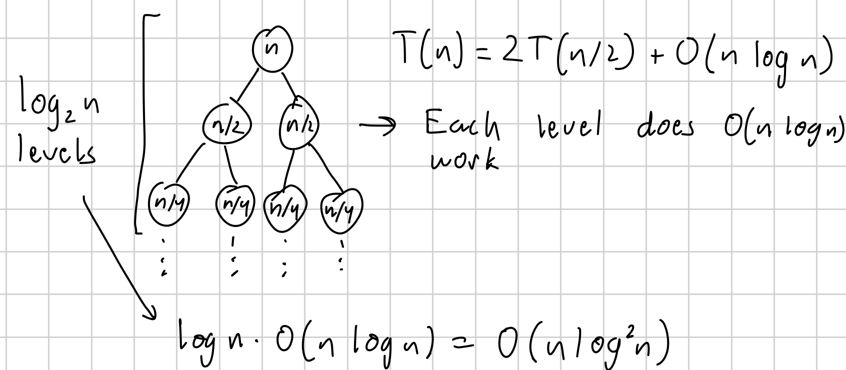
Dual: Max $500p + 100c + 400f$

s.t. $500p + 500f \leq 5$
$50p + 300c + 25f \leq 2$
$300p + 100c + 200f \leq 4$
$p, c, f \geq 0$

FFT Example 1

Let $p(x) = \prod_{i=1}^{n}(x - r_i)$ with distinct roots $r_1, \dots, r_n$. $p(x) = a_n x^n + a_{n-1}x^{n-1} + \dots + a_0$. Give an algorithm to compute coefficients $a_0, \dots, a_n$ and show it runs in $O(n \log^2 n)$.

Given the n roots, split them in two halves. Recurse on each half until you reach the base case where each half has only one factor. Then, start combining the halves using FFT multiplication.



$T(n) = 2T(n/2) + O(n \log n)$ → Each level does $O(n \log n)$ work

$\log n \cdot O(n \log n) = O(n \log^2 n)$

FFT Example 2

Given an array A of n integers where each value is between 0 and n, determine if any three elements (with repetition allowed) sum to exactly n.

$A[i] + A[j] + A[k] = n$ is equivalent to $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^n$

Let $f(x) = x^{A[0]} + x^{A[1]} + \dots + x^{A[n-1]}$. When you multiply $(f(x))^3$, the coefficient of x^n is the number of ways to pick three monomials from $f(x)$ whose exponents add to n. So you can apply FFT at the mth roots of unity where $m \geq 3n$ and is a power of 2. Then, cube each value pointwise to get values of $f(x)^3$. Then, apply inverse FFT, so $\frac{1}{m}$ FFT($f(x)^3$ values, $e^{-2\pi i/m}$) to recover the coefficients. Check if the coefficient of x^n is 0. If it is, there are no 3 elements that sum to n, otherwise there is.

Runtime: $O(n \log n)$

Zero Sum Games

A conflict between two players where one player's gain is the other's loss. The total payoff always sums to 0.

Payoff Matrix G: Represents the game

- G_{ij} = payoff to P1 (row player) if P1 plays i and P_2 plays j. P_2 pays the payoff so P2 wants it low and P1 wants it high.

For example, for rock-paper-scissors:

$$\begin{array}{c|ccc} & r & p & s \\ \hline r & 0 & -1 & 1 \\ p & 1 & 0 & -1 \\ s & -1 & 1 & 0 \end{array}$$

Strategies:

① Pure strategy: always play one fixed move ($p=1$ on 1 option)

② Mixed strategy: play each move with some probability.

- P1's strategy is a vector $x = [x_1\ x_2\ \dots]$ where x_i = probability that P1 plays row i. Same for P2 with y.
- $\sum x_i = 1$

Expected Payoff: On any given round, there's an $x_i \cdot y_j$ chance P1 plays row i and P2 plays column j. P1 (row player) wants to maximize expected payoff, while P2 (column player) wants to minimize expected payoff

Expected Payoff $= \sum_{i,j} G_{ij} \cdot x_i \cdot y_j$

Optimal Strategy and LP Formulation

If P2 knows P1's strategy and the strategy will never change, P2 responds with the pure strategy (single column) that minimizes payoff.

So P1's payoff $= \min_j (\sum_i G_{ij} \cdot x_i)$. P1 wants to maximize this worst case. This can be represented as an LP.

P1 LP (Maximizer):

Let z = P1's guaranteed minimum payoff

max z

s.t. $z \le \sum_i G_{ij} \cdot x_i$ for every column j

$\sum_i x_i = 1$

$x_i \ge 0$

P2 LP (Minimizer):

Let w = payoff

min w

s.t. $w \ge \sum_j G_{ij} \cdot y_j$ for every row i

$\sum_j y_j = 1$

$y_j \ge 0$

Minimax Theorem

P1's LP and P2's LP are duals of each other. By LP duality, they have the same optimal value v = the value of the game.

$$\max_x \left(\min_y \left(\sum_{i,j} G_{ij} \cdot x_i \cdot y_j\right)\right) = \min_y \left(\max_x \left(\sum_{i,j} G_{ij} \cdot x_i \cdot y_j\right)\right)$$

This means that it doesn't matter who announces their strategy first. If both play optimally, the expected payoff is v.

To find v, solve either LP, this gives v and the mixed strategy

For a 2x2 payoff matrix, you can solve by setting the 2 constraints equal (these ones → $z \le \sum_i G_{ij} \cdot x_i$)

Multiplicative Weights Update (MWU)

Simple, iterative algorithm that constructs optimal strategies.

P1's m rows are "experts". Each expert gets a weight w_i.

① Initialize weights $w_i = 1$ for all rows i.

② For each round t:

① P1 plays mixed strategy $x_i^{(t)} = \frac{w_i^{(t)}}{\sum_k w_k^{(t)}}$

- Finds the ratio of row i's weight to the total weight. This is used as the probability for row i for this round.

② P2 best responds with $j^{(t)} = \text{argmin}_j (\sum_i x_i^{(t)} \cdot G_{ij})$

- P2 sees P1's full mixed strategy. For each column j, P2 computes the expected payoff $\sum_i x_i^{(t)} G_{ij}$. P2 picks the column j with the minimum payoff.

③ Update weights: $w_i^{(t+1)} = w_i^{(t)} \cdot (1-\varepsilon)^{(-G_{i,j(t)})}$ ← entry picked this round; ε ← should be given

Output $\bar{x} = \frac{1}{T} \sum_t x^{(t)}$ ← the time averaged strategy.

Zero Sum Games Example

Payoff matrix:

	Bob 1	Bob 2
Alice 1	4	1
Alice 2	2	5

Let p = payoff

Alice LP: max p

s.t. $p \le 4x_1 + 2x_2$ (col 1)

$p \le x_1 + 5x_2$ (col 2)

$x_1 + x_2 = 1$

$x_1, x_2 \ge 0$

Bob LP: max p

$p \ge 4y_1 + y_2$ (row 1)

$p \ge 2y_1 + 5y_2$ (row 2)

$y_1 + y_2 = 1$

$y_1, y_2 \ge 0$

Solving: $4x_1 + 2x_2 = x_1 + 5x_2$

$3x_1 = 3x_2$

$x_1 = x_2$

$x_1 + x_2 = 1$

$x_2 + x_2 = 1$

$2x_2 = 1$

$x_2 = \frac{1}{2}, x_1 = \frac{1}{2}$

$4(\frac{1}{2}) + 2(\frac{1}{2}) = 2 + 1 = 3$

Optimal solution: $x_1 = \frac{1}{2}$, $x_2 = \frac{1}{2}$, $p = 3$

Optimal value: 3

NP Complete Problems

3SAT:
Given a boolean formula in CNF where every clause has ≤3 literals, find a satisfying truth assignment or report none exists.

Independent Set:
Given a graph $G=(V,E)$ and integer k, find a set of k vertices s.t. no two are connected by an edge or report none exists.

Vertex Cover:
Given a graph $G=(V,E)$ and integer k, find a set of k vertices s.t. every edge has at least one endpoint in the set, or report none exists.

Clique:
Given a graph $G=(V,E)$ and integer k, find a set of k vertices that are fully connected, or report none exists.

Rudrata Cycle:
Given a graph $G=(V,E)$, find a cycle that visits every vertex exactly once, or report none exists.

TSP:
Given a complete graph G with edge weights and budget b, find a cycle that visits every vertex exactly once with total cost ≤b, or report none exists.

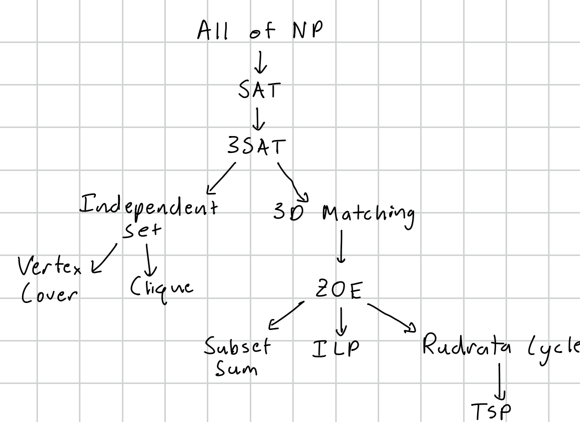
Subset Sum:
Given a set of integers and target t, find a subset of S that sums to exactly t, or report none exists.

ZOE (Zero One Equations):
Given a matrix A and vector b, find a 0/1 vector x s.t. $Ax = b$, or report none exists. (↑ every entry is 1 or 0)

3D Matching:
Given three disjoint sets X, Y, Z each of size n and a set of triples $T \subseteq X \times Y \times Z$, find n triples from T s.t. every element of X, Y, Z appears in exactly one triple, or report none exists.

Reduction Tree

Arrow means "reduces to"



Reduction Proof

"Prove X is NP-Complete. You may use the fact that Y is NP-Complete".

Steps:

① Prove X is in NP: Given a proposed solution, argue it runs in polynomial time.

② Prove X is NP-Hard by reducing Y to X:

- Describe the transformation: Given an instance I of Y, describe how to construct an instance I' of X in polynomial time.
- Prove correctness forward: Assume I has a solution, show how it gives you a solution to I'.
- Prove correctness backward: Assume I' has a solution, show how to recover a solution to I.
- Argue polynomial time: State that the transformation takes polynomial time.

NP Completeness

Types of Problems:

① Optimization: Given an instance I, find the best solution (e.g. min/max)

② Search: Given an instance I, find a solution S or state that no solution exists

③ Decision: Given an instance I, output "yes" if a solution exists or "no" if not.

Conjunctive Normal Form (CNF): A literal is a variable or its negation like x or $\neg x$. A clause is an OR of literals $(x_1 \vee x_2 \vee \neg x_3)$. CNF is an AND of clauses. For example, $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$

Examples of search problems

SAT: Given a boolean formula in CNF, find a satisfying truth assignment or report that none exists.

- 2-SAT: clauses with ≤2 literals
- 3-SAT: clauses with ≤3 literals

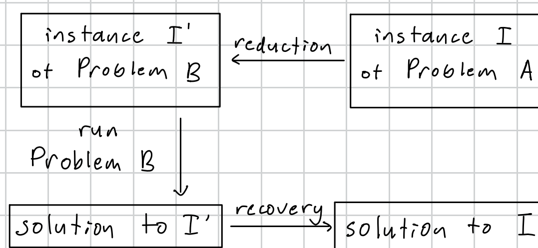
Search TSP: Given a graph G, find a cycle that contains all nodes exactly once with total cost ≤ budget b or report that none exists.

Euler/Rudrata Path: Given a graph G, find a path that contains every edge/visits every vertex exactly once

Euler/Rudrata Cycle: Given a graph G, find a cycle that contains every edge/visits every vertex exactly once.

Reduction

Polynomial time algorithm that transforms an instance I from problem A into an instance I' for problem B.



Problem B is at least as hard as Problem A

Reduction Proof

① If there is a solution for instance I', there must exist a solution for instance I.

② If there is a solution for instance I, there must exist a solution for instance I'.

Complexity Classes

P: problem that can be solved in polynomial time.

NP: Problem in which a candidate solution can be checked in polynomial time.

NP-Complete: An NP Complete problem is one that every problem in NP can reduce to.

NP-Hard: Problem that is at least as hard as an NP-Complete problem.

NP-Complete Proof

① Prove that problem is NP
- Show a solution can be verified in polynomial time.

② Prove that problem is NP-hard
- Reduce an NP-Complete problem to the problem.

Hard problems (NP-complete)	Easy problems (in P)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

When does P=NP?

① If an NP-Complete problem can be solved in polynomial time

② If you can reduce an NP-Complete problem to a P problem

Reduction Example 1

Reduce rudrata path to rudrata cycle

Rudrata path: Given $G=(V,E)$, find a path that visits every vertex exactly once

Rudrata cycle: Given $G=(V,E)$, find a cycle that visits every vertex exactly once

Given instance I: graph $G=(V,E)$

Construct instance I': add a vertex that connects to all vertices

Use rudrata cycle problem: finds cycle with all vertices

Finding solution: Remove added vertex and connected edges to end up with a path that uses all vertices.

Reduction Example 2

Prove that Independent Set is NP-Complete. You may use the fact that 3-SAT is NP-Complete.

Step 1: Given a proposed set S of k vertices, verify no two share an edge. This is $O(k^2)$, which is polynomial.

Step 2:

Given: A 3SAT formula with k clauses

For each clause, create a triangle (3 nodes, one per literal, all connected to each other). Then add edges between a literal and its negation across triangles.

Claim: The formula is satisfiable $\Leftrightarrow$ the graph has an independent set of size k.

Forward: If the formula is satisfiable, pick one true literal from each clause. Those k nodes form an independent set since no two are in the same triangle and no two are connected across triangles.

Backward: If there's an independent set of size k, at most one node per triangle, so one literal per clause is picked. Assign those literals as true. There is no contradiction because a literal and its negation can't both be in the set because of the cross triangle edges.

Hashing

A hash function maps n elements from a large universe to m buckets. If two elements land in the same bucket, it's a collision, and the bucket becomes a linked list. The cost of lookup/insert/delete is proportional to the length of the linked list at that bucket.

Hash functions are random so that they can't be exploited w/ specific inputs.

Hash Families

Universal Family: A hash family H is universal if for every pair of distinct keys $x \neq y$:

$$P(h(x) = h(y)) \leq \frac{1}{m}$$

Collision probability is $\frac{1}{m}$ at most.

Pairwise-independent family: H is pairwise independent if for every $x \neq y$ and every $a, b \in$ set of buckets:

$$P(h(x) = a \text{ and } h(y) = b) = \frac{1}{m^2}$$

h(x) and h(y) are uniform, they are equally likely to end up in any of the m buckets. Also h(x) and h(y) are independent.

Pairwise independence implies universality but not vice versa.

For any query key x and n keys, the expected number of collisions is n/m. Lookup $= O(\frac{n}{m})$

Uniform: all functions $h: [U] \rightarrow [m]$ where $P(h(k) = h(k')) = \frac{1}{m}$ $\forall k \neq k'$

$$E[\text{query time}] = O\Big(\underbrace{1}_{\text{compute hash}} + \underbrace{\sum_{k \neq k'} P(h(k) = h(k'))}_{\# \text{ of collisions}}\Big) = O(1 + \frac{n}{m}) = O(1)$$

($m \geq n$)

Hash Function Properties

1. Deterministic: same input always gives the same output
2. Fast: expected query time (insertion/deletion) should be $O(1)$
3. Well distributed: different inputs spread roughly uniformly across [m] so collisions are rare
   - $m \geq n$

Streaming

You see a sequence of elements $x_1, \ldots, x_n$ one at a time. You can only store a tiny amount of data.

Solution: Process each element and discard it, and output an answer at the end.

Space: $O(\log n)$ bits   Runtime: $O(n \cdot$ runtime per element$)$

Count-Distinct Problem

Given a stream, count the number of distinct elements d.

Use Min-Hash Algorithm:

For a pairwise independent hash family H:

1. Pick h from H uniformly at random
2. Initialize $V = \infty$
3. For each stream element x, set $V = \min(V, h(x))$
4. Output $d = \frac{M}{V} - 1$, M = # of buckets

Why it works: The d points are scattered uniformly on [0, M]. This creates d+1 gaps, and the minimum one sits at $\frac{M}{d+1}$. So $E[V] = \frac{M}{d+1} \Rightarrow E[\frac{M}{V}] \approx d+1 \Rightarrow d = \frac{M}{V} - 1$

Hashing/Streaming Example

Let $h: U \rightarrow \{0, 1, \ldots M-1\}$ be a pairwise independent hash function.

a) Show that for any two distinct elements $x \neq y$, $P(h(x) = h(y)) \leq \frac{1}{m}$

Since h is pairwise independent, for any $a, b \in [m]$

$P(h(x) = a \text{ and } h(y) = a) = \frac{1}{m^2}$

There is m possible values of a so $P(h(x) = h(y)) = \frac{m}{m^2} = \frac{1}{m}$

b) Using this hash function, design a streaming algorithm that estimates the number of distinct elements in a stream $x_1, \ldots, x_n$ using $O(\log N)$ bits of memory.

Algorithm:

1. Pick h from a pairwise-independent family $H: U \rightarrow \{0, \ldots, M-1\}$ with $M = N^3$
2. Initialize $V = M$
3. For each stream element x, set $V = \min(V, h(x))$
4. Output $d = \frac{M}{V} - 1$

Memory: Storing h requires $O(\log n)$ bits and storing V requires $O(\log n)$ bits. Total $= O(\log n)$