

Efficient Algorithms and Intractable Problems

Daniel Deng

1 Complexity Analysis

1.1 Partial Sums

$$S_k = \sum_{n=1}^k a_n = \frac{k}{2}(a_1 + a_k) \quad (\text{Arithmetic Series})$$

$$S_k = \sum_{n=1}^k a_1(r)^n = a_1 \left(\frac{1 - r^k}{1 - r} \right) \quad (\text{Geometric Series})$$

1.2 Asymptotic Relations

$$f = O(g) \approx f(n) \leq c \cdot g(n)$$

$$f = o(g) \approx f(n) < c \cdot g(n)$$

$$f = \Omega(g) \approx f(n) \geq c \cdot g(n)$$

$$f = \omega(g) \approx f(n) > c \cdot g(n)$$

$$f = \Theta(g) \approx f(n) = c \cdot g(n)$$

Remark. $O(i^n \mid i > 1) > O(n^j) > O(\log^k n)$

Theorem 1.1 (Master Theorem). *If $T(n) = aT([n/b]) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then*

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

2 Polynomial Interpolation

Given a degree n polynomial $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$, the relationship between its values and coefficients can be represented by

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \quad (\text{evaluation})$$

where the matrix M is a *Vandermonde* matrix.

2.1 Fast Fourier Transform (FFT)

Definition 2.1 (Discrete Fourier Transform Matrix). For polynomials of degree $< n$ (n is even; polynomials can be 0-padded), the Discrete Fourier Transform can be represented by the matrix

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

where $\omega = e^{2\pi i/n}$ is the n th root of unity, and $M_n(\omega)$ is an unitary matrix whose columns forms the *Fourier Basis*.

Remark. $M_n^{-1}(\omega) = \frac{1}{n} \overline{M_n(\omega)} = \frac{1}{n} M_n(\omega^{-1})$

Algorithm 1: Fast Fourier transform

Input: A coefficient vector, $\vec{a} = \langle a_0, \dots, a_{n-1} \rangle$ and the n th root of unity, ω .

Output: $M_n(\omega)\vec{a}$

```

1 Function FFT( $\vec{a}, \omega$ ):
2   if  $\omega = 1$  then
3     return  $\vec{a}$ 
4   else
5     //  $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$ 
6      $\langle A_e(0), \dots, A_e(n/2 - 1) \rangle \leftarrow \text{FFT}(\langle a_0, a_2, \dots, a_{n-2} \rangle, \omega^2)$ 
7      $\langle A_o(0), \dots, A_o(n/2 - 1) \rangle \leftarrow \text{FFT}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \omega^2)$ 
8     for  $j := 0$  to  $n/2 - 1$  do
9        $A(j) \leftarrow A_e(j) + \omega^j A_o(j)$ 
10       $A(j + n/2) \leftarrow A_e(j) - \omega^j A_o(j)$ 
11   return  $\langle A(0), \dots, A(n-1) \rangle$ 

```

Remark. If A is evaluated at points $\pm\omega_0, \dots, \pm\omega_{n/2-1}$, then $A_e(x^2)$ and $A_o(x^2)$ will only need to evaluate half the amount of points due to squaring.

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

2.2 Applications of FFT

Algorithm 2: Fast Polynomial Multiplication

Input: Coefficient vectors, a and b , and the n th root of unity, ω .

Output: The coefficient vector of $A(x)B(x)$

```

1  $\hat{\vec{a}} \leftarrow M_n(\omega)\vec{a}$  (FFT)
2  $\hat{\vec{b}} \leftarrow M_n(\omega)\vec{b}$ 
3 for  $i = 0$  to  $n - 1$  do
4    $\hat{c}_i \leftarrow \hat{a}_i \hat{b}_i$ 
5 return  $\frac{1}{n} M_n(\omega^{-1})\hat{\vec{c}}$  (inverse matrix)

```

Definition 2.2 (Cross-Correlation). $\text{corr}(\vec{x}, \vec{y})[k] = \sum x_i y_{i-k}$, which measures similarity.

Algorithm 3: Cross-Correlation

Input: Two signal vectors, \vec{x} and \vec{y} .

Output: $\text{corr}(\vec{x}, \vec{y})$

- 1 $X(t) \leftarrow x_{m-1} + x_{m-2}t + \cdots + x_0t^{m-1}$
 - 2 $Y(t) \leftarrow y_0 + y_1t + \cdots + y_{n-1}t^{n-1}$
 - 3 $Q(t) \leftarrow X(t)Y(t)$ (Fast Polynomial Multiplication)
 - 4 **return** \vec{q}
-

3 Graphs

Definition 3.1 (Graph). A graph is a pair $G = (V, E)$, typically represented by an adjacency matrix or an adjacency list.

Table 1: Graph representations.

	Space	Connectivity	getNeighbors(u)	DFS Runtime
Adjacency Matrix	$\Theta(V ^2)$	$O(1)$	$\Theta(V)$	$\Theta(V ^2)$
Adjacency List	$\Theta(V + E)$	$\Theta(\text{degree}(u))$	$\Theta(\text{degree}(u))$	$\Theta(V + E)$

3.1 Depth-First Search

Algorithm 4: Depth-first search

Input: V, E of directed graph G .

1 **Function** DFS(V, E):

2 $n \leftarrow |V|$

3 $clk \leftarrow 1$

4 $visited \leftarrow \text{boolean}[n]$

5 $preorder, postorder = \text{int}[n]$

6 **for** $v \in V$ **do**

7 **if** $\neg visited[v]$ **then**

8 EXPLORE(v)

9 **Function** EXPLORE(v):

10 $visited[v] \leftarrow \text{True}$

11 $preorder[v] \leftarrow clk++$

12 **for** $(v, w) \in E$ **do**

13 **if** $\neg visited[w]$ **then**

14 EXPLORE(w)

15 $postorder[v] \leftarrow clk++$

/* Preorder-postorder intervals are either nested or disjoint. */

/* $postorder[u] \leq postorder[v]$ iff (u, v) is a back edge. */

/* G contains a cycle iff it contains a back edge. */

3.1.1 Applications of DFS

Algorithm 5: Topological sort.

Input: A directed cyclic graph G .

Output: An ordered list of V such that u_i comes before v_i for all $(u_i, v_i) \in E$ (i.e., ordered by decreasing dependency).

1 $post \leftarrow$ DFS-visited vertexes ordered by postorder visits

2 **return** $reverse(post)$

Definition 3.2 (Strongly Connected Component). A SCC is a maximal partition of a directed graph in which every vertex is reachable from every other vertex.

u is in sink SCC of graph $G \Leftrightarrow u$ is in source SCC of reverse graph G
 $\Leftrightarrow u$ is in source SCC if highest postorder number.

3.2 Single-Source Shortest Path

Algorithm 6: Single-Source Shortest Path

Input: A directed graph G and a start vertex S .

Output: Two arrays $prev[|V|]$ (shortest-path predecessor) and $dist[|V|]$ (shortest-path distance).

1 **Function** $BFS(G, S)$:

\lfloor /* Must have uniform edge weights. $O(|V| + |E|)$ runtime. */

2 **Function** $Dijkstra(G, S)$:

\lfloor /* Must have positive edge weights. $O(|V| \log |V| + |E|)$ runtime if
 implemented using Fibonacci heap. */

3 **Function** $Bellman-Ford(G, S)$:

\lfloor /* Can have arbitrary edge weights. */

3.3 Minimum Spanning Tree

Algorithm 7: Minimum spanning tree.

Input: A graph G and a starting vertex v .

Output: The minimum spanning tree T of G .

/ Use the cut property. */*

1 **Function** Prim(G, v):

/ Sequentially adds the closest neighbor of the running set.
 $O(|E| + \log |V|)$ runtime if implemented using Fibonacci heap. */*

2 **Function** Kruskal(G, v):

/ Sequentially adds the shortest edge that does not create a
 cycle. $O(|E| \log |V|)$ runtime if implemented using Union-Find
 data structure. */*

4 Greedy Algorithm

Definition 4.1 (Greedy Algorithm). A greedy algorithm is one that builds the solution iteratively using a sequence of local choices.

Algorithm 8: Example greedy algorithms.

1 Function SCHEDULING:

```

    /* Find the maximum set of jobs that can be completed within time
       by iteratively select the next job to have the smallest end
       time without conflicting existing schedule.  $O(n \log n)$  runtime
       if sorting the collection of jobs first. */

```

2 Function HUFFMAN:

```

    /* Find a prefix tree for prefix-free Huffman coding by
       iteratively combine the two least frequent elements of the
       alphabet and retrieve the order of the prefix tree accordingly.
        $O(n \log n)$  runtime if implemented with min-heap. */

```

Input: A set of partitions $S = \{S_1, \dots, S_m\}$ that covers the universe $\{1, \dots, n\}$.

Output: The indices of the smallest sub-collection of S that covers the universe.

3 Function SET-COVER:

```

    /* Greedy search yields sub-optimal but competitive solution to
       the set-cover problem. If the optimal solution uses  $k$  sets,
       then the greedy solution uses at most  $k \ln n$  sets. */
4    $A \leftarrow \{1, \dots, n\}$ 
5    $B \leftarrow \emptyset$ 
6   while  $|A| > 0$  do
7       let  $i \in [m] \setminus B$  be s.t.  $|A \cap S_i|$  is maximum
8        $A \leftarrow A \setminus S_i$ 
9        $B \leftarrow B \cup i$ 
10  return  $B$ 

```

5 Union-Find

Definition 5.1 (Amortized Analysis). Suppose a data structure supports k operations. Then the amortized cost of each operation is t_i if for any sequence of operations with N_i of O_i operations, the total time is at most $\sum_{i=1}^k t_i N_i$.

Algorithm 9: Union-find (disjoint forest implementation).

```

/*  $O((m+n) \log^* n)$  runtime. */
1  $parent[1, \dots, n]$ 
2  $rank[1, \dots, n]$  // rank is defined as the height if no path compression
   occur.
3 Function MAKE-SET( $x$ ):
4    $parent[x] \leftarrow x$ 
5    $rank[x] \leftarrow 0$ 
6 Function FIND( $x$ ):
7   if  $x = parent[x]$  then
8     return  $x$ 
9    $parent[x] \leftarrow FIND(parent[x])$  // path compression
10  return  $parent[x]$ 
11 Function UNION( $x, y$ ):
12    $x \leftarrow FIND(x)$ 
13    $y \leftarrow FIND(y)$ 
14   if  $x = y$  then
15     return // no work needed
16   if  $rank[x] > rank[y]$  then
17     swap  $x$  and  $y$ 
18    $parent[x] \leftarrow y$  if  $rank[x] = rank[y]$  then
19      $rank[y] \leftarrow rank[y] + 1$ 

```

Remark. Union-Find Invariants:

- a tree rooted at x has $\geq 2^{r[x]}$ items;
- $(\forall x)$, if x is not a root, $r[p[x]] > r[x]$;
- the number of items of exactly rank k is $\leq \frac{n}{2^k}$.

6 Dynamic Programming

Definition 6.1 (Top-Down DP/Memoization). Recursion + look-up table

Definition 6.2 (Bottom-Up DP). Fill up the look-up table iteratively instead of recursively

Remark. Bottom-up DP sometimes have better memory

6.1 Bellman-Ford

Definition 6.3 (Bellman-Ford). An algorithm for finding the SSSP on a directed graph with potentially negative weights. The algorithm defines a function $f(t, k) :=$ the length of the shortest path from s to t using $\leq k$ edges, and wishes to solve for $f(t, n - 1)$. The recurrence relation of the algorithm is

$$f(t, k) = \begin{cases} \infty & k = 0, t \neq s \\ 0 & k = 0, t = s \\ \min \begin{cases} f(t, k - 1) \\ \min_{(v,t) \in E} w(v, t) + f(v, k - 1) \end{cases} & \text{else} \end{cases}$$

Remark. Negative cycle detection: \exists negative cycle $\Leftrightarrow \exists v, f(v, n) < f(v, n - 1)$

Algorithm 10: Bellman-Ford

Input: G, s .

```

1 initialize  $T[1, \dots, n]$  to all  $\infty$ 
2  $T[s] \leftarrow 0$ 
3 for  $k = 1$  to  $n - 1$  do
4   foreach  $(u, v) \in E$  do
5      $T[v] \leftarrow \min \begin{cases} T[v] \\ w(u, v) + T[u] \end{cases}$ 
6 return  $T$ 
```

- Memory: $O(n)$ with bottom-up
- Runtime: $O(n^2 + mn)$

6.2 Floyd-Warshall

Definition 6.4 (Floyd-Warshall). An algorithm for finding the all pairs shortest path (APSP) on a directed graph. Assume the graph is complete (pretend $w(e) = \infty$ for all

$e \notin E$), the algorithm defines a function $f(i, j, k) :=$ the length of the shortest path from i to j when all intermediate vertices in the path must be in $\{1, \dots, k\}$. The recurrence relation is

$$f(i, j, k) = \begin{cases} w(i, j) & k = 0, i \neq j \\ 0 & k = 0, i = j \\ \min \begin{cases} f(i, j, k-1) \\ f(i, k, k-1) + f(k, j, k-1) \end{cases} & \text{else} \end{cases}$$

Algorithm 11: Floyd-Warshall

Input: G

```

1 initialize  $T[1 \dots n][1 \dots n]$  such that  $T[i][j] \leftarrow w(i, j)$  for all  $i, j$ 
2 for  $k = 1$  to  $n$  do
3   for  $i = 1$  to  $n$  do
4     for  $j = 1$  to  $n$  do
5        $T[i][j] \leftarrow \begin{cases} T[i][j] \\ T[i][k] + T[k][j] \end{cases}$ 
6 return  $T$ 
```

- Memory: $O(n^2)$
- Runtime: $O(n^3)$

6.3 Longest Increasing Subsection

Definition 6.5 (LIS). Define $f(last, i) :=$ length of the LIS of $A[i \dots n]$ such that all values used are $> A[last]$. The recurrence relation is

$$f(last, i) = \begin{cases} 0 & i = n + 1 \\ f(last, i + 1) & A[i] \leq A[last] \\ \max \begin{cases} f(last, i + 1) \\ f(i, i + 1) + 1 \end{cases} & \text{else} \end{cases}$$

- Memory: $O(n)$
- Runtime: $O(n^2)$

6.4 Knapsack

Definition 6.6 (Knapsack). Given an array $A[1 \dots n]$ of items, each being a (weight, value) pair. Given a knapsack that can hold $\leq W$ weight, find the maximum value containable in the knapsack. The algorithm for solving this problem defines a function $f(i, C) :=$ maximum value we can pack among $A[i \dots n]$ with capacity C . The recurrence relation is

$$f(i, C) = \begin{cases} 0 & i = n + 1 \\ f(i + 1, C) & w[i] > C \\ \max \begin{cases} f(i + 1, C) \\ f(i + 1, C - w[i]) + v[i] \end{cases} & \text{else} \end{cases}$$

- Memory: $O(W)$
- Runtime: $O(nW)$ *pseudopolynomial

Remark. For knapsack problems with replacement, the recurrence relation is instead

$$f(C) = \max_i (f(C - w[i]) + v[i])$$

6.5 Traveling Salesman Problem

Definition 6.7 (Traveling Salesman Problem). Given n locations with distances $D[i][j]$. The traveling salesman wishes to visit all locations, starting at 1, while minimizing total travel distance. The DP algorithm defines a function $f(i, S) :=$ minimum traveling distance to visit all locations in S when starting at 1. The recurrence relation is

$$f(i, S) = \begin{cases} 0 & S \neq \emptyset \\ \min_{x \in S} (D[i][x] + f(x, S \setminus \{x\})) & \text{else} \end{cases}$$

- Memory: $O(\sqrt{n} \cdot 2^n)$
- Runtime: $O(n^2 \cdot 2^n)$

6.6 Matrix Chain Multiplication

Definition 6.8 (Matrix Chain Multiplication). Given s_1, \dots, s_{n+1} such that A_i is a $s_i \times s_{i+1}$ matrix, and we want to find the minimum number of flops to compute $A_1 \times \dots \times A_n$. The DP algorithm defines a function $f(i, j) :=$ minimum number of flops to compute $A_i \times \dots \times A_j$. The recurrence relation is

$$f(i, j) = \begin{cases} 0 & i = j \\ \min_{i \leq k \leq j} (f(i, k) + f(k + 1, j) + s_i s_{j+i} s_{k+1}) & \text{else} \end{cases}$$

- Memory: $O(n^2)$
- Runtime: $O(n^3)$

7 Linear Programming

Definition 7.1 (Linear Programming). Linear programming (LP) describes a broad class of optimization tasks in which both the constraints and the optimization criterion are linear functions. The optimum of a linear program is achieved at a vertex of the convex feasible region.

Remark. A linear program does not have an optimum *iff* its feasible region is infeasible and/or unbounded.

Definition 7.2 (Simplex Method). A standard greedy algorithm for solving LP by hill-climbing on vertices of the feasible region.

Remark. Solves real-life LP in polynomial time.

7.1 LP Conversion

1. (Maximization \leftrightarrow minimization) multiply the coefficients of the objective function by -1
2. (Inequality \rightarrow equality) $ax \leq b \rightarrow ax + s = b \mid s \geq 0$
3. (Equality \rightarrow Inequality) $ax = b \rightarrow ax \leq b \wedge ax \geq b$
4. (Signed \leftrightarrow unsigned) $x \leftrightarrow x^+ - x^- \mid x^+, x^- \geq 0$

7.2 Duality

Theorem 7.1 (Duality theorem). *If a linear program has a bounded optimum, then so does its dual, and the two optimum values coincide (strong duality; weak duality states that primal opt. \leq dual opt.).*

Primal LP:

$$\begin{aligned} \max c^T x \\ Ax \leq b \\ b \geq 0 \end{aligned}$$

Dual LP:

$$\begin{aligned} \min y^T b \\ y^T A \geq c^T \\ y \geq 0 \end{aligned}$$

Remark. Dual/Primal unbounded \implies Primal/Dual unfeasible.

7.3 Network Flow

Definition 7.3 (Flow). Given a directed graph $G = (V, E)$ with capacities $c_e > 0$ on all edges. The flow f from source s to sink t satisfies the constraints:

1. For all $e \in E$,

$$0 \leq f_e \leq c_e$$

2. For all $e \in E \setminus \{s, t\}$,

$$\sum_{(w,u) \in E} f_{wu} = \sum_{(u,z) \in E} f_{uz} \quad (\text{conservation of flow})$$

Remark. By the conservation principle,

$$\text{size}(f) = \sum_{(s,u) \in E} f_{su}$$

Algorithm 12: Ford-Fulkerson

```
// Simplex algorithm for solving max-flow problem. Pseudopolynomial
// time complexity;  $O(|E|F)$ 
Input:  $G = (V, E), s, t$ 
Output:  $f$ 
//  $G_f :=$  residual graph (also contains back-edges)
1 while  $\exists$  an augmenting path in  $G_f$  do
2   Find an arbitrary augmenting path  $P$  from  $s$  to  $t$ 
3   Augment flow  $f$  along  $P$ 
4   Update  $G_f$ 
```

Theorem 7.2 (Max-flow Min-cut Theorem). *The size of the maximum flow in a network equals the capacity of the smallest (s, t) -cut (L, R) (total capacity of the edges crossing the cut)*

Remark. L contains all reachable vertices from s in the final residual G^f and R contains all remaining vertices.

7.4 Zero-Sum Game

Theorem 7.3 (Min-Max Theorem). *For zero-sum games, there exists an equilibrium such that*

$$\max_x \min_y \sum_{i,j} G_{i,j} x_i y_j = \min_y \max_x \sum_{i,j} G_{i,j} x_i y_j$$

Remark. To convert game strategies to LP:

$$\bullet \max_x \min_y \sum_{i,j} G_{i,j} x_i y_j \implies$$

$$\begin{aligned} & \max z \\ \forall y, & \sum_{i,j} G_{i,j} x_i y_j \geq z \\ & \sum_i x_i = 1 \\ & \forall i, x_i \geq 0 \end{aligned}$$

$$\bullet \min_y \max_x \sum_{i,j} G_{i,j} x_i y_j \implies$$

$$\begin{aligned} & \min w \\ \forall x, & \sum_{i,j} G_{i,j} x_i y_j \leq w \\ & \sum_j y_j = 1 \\ & \forall j, y_j \geq 0 \end{aligned}$$

It is apparent that the two LPs are dual; therefore, the equilibrium can be found in polynomial time via LP.

8 Multiplicative Weight Updates

Definition 8.1 (Online Decision Making). A problem where one chooses to follow expert $i^{(t)}$ out of n experts on day $t \in \{1, \dots, T\}$, who incurs a loss of $l_i^{(t)}$ on day t ($\forall i, t, l_i^{(t)}$ is bounded by $[0, 1]$; range can be normalized), with the goal of minimizing the total loss

$$L := \sum_{t=1}^T l_i^{(t)}$$

Realistically, the problem aims to minimize the regret

$$R := L - L^* \quad \left(L^* := \min_{i \in [n]} \sum_{t=1}^T l_i^{(t)} \right)$$

Remark. It would be trivial to define the offline optimum L^* as $\sum_{t=1}^T \min_{i \in [n]} l_i^{(t)}$ in minimizing regret.

8.1 Hedge/MWU

Algorithm 13: Hedge/MWU

```

/* Defined expected loss on day  $t$  to be  $L_t := \langle x^{(t)}, l^{(t)} \rangle$  and the total
   loss to be  $L := \sum_{t=1}^T L_t$ , where  $x^{(t)}$  is the probability distribution
   of choosing any expert on day  $t$ . */
Input:  $\epsilon \in [0, \frac{1}{2}]$ .
1  $\forall i \in [n], w_i^{(1)} \leftarrow 1$ 
2  $x_i^{(t)} \leftarrow \frac{w_i^{(t)}}{W^{(t)}} \quad // \quad (W^{(t)} := \sum_{j=1}^n w_j^{(t)})$ 
3  $w_i^{(t+1)} \leftarrow w_i^{(t)} (1 - \epsilon)^{l_i^{(t)}}$ 

```

Lemma 8.1. $W^{(T+1)} \geq (1 - \epsilon)^{L^*}$

Proof.

$$\begin{aligned}
 W^{(T+1)} &\geq w_{i^*}^{(T+1)} \\
 &= \prod_{t=1}^T (1 - \epsilon)^{l_{i^*}^{(t)}} \\
 &= (1 - \epsilon)^{L^*}
 \end{aligned}$$

□

Lemma 8.2. $W^{(T+1)} \leq n \cdot \prod_{t=1}^T (1 - \epsilon \cdot L_t)$

Proof. TODO □

Theorem 8.3. *Hedge*(ϵ) achieves $\mathbb{E}[R] \leq \epsilon \cdot T + \frac{\ln n}{\epsilon}$ (or $\mathbb{E}[R] \leq 2\sqrt{T \ln n}$ if $\epsilon = \sqrt{\frac{\ln n}{T}}$).

Proof.

$$\begin{aligned}
 (1 - \epsilon)^{L^*} &\leq n \cdot \prod_{t=1}^T (1 - \epsilon \cdot L_t) \\
 \implies L^* \ln(1 - \epsilon) &\leq \ln n + \sum_{t=1}^T \ln(1 - \epsilon \cdot L_t) \\
 \implies L^* (-\epsilon - \epsilon^2) &\leq \ln n - \epsilon \sum_{t=1}^T L_t \\
 \implies \sum_{t=1}^T L_t - L^* &\leq \frac{\ln n}{\epsilon} + \epsilon \cdot L^* \\
 \implies \mathbb{E}[R] &\leq \epsilon \cdot T + \frac{\ln n}{\epsilon}
 \end{aligned}$$

□

Remark. $\forall z \in [0, \frac{1}{2}], -z - z^2 \leq \ln(1 - z) \leq -z$

9 Reductions

Definition 9.1 (Reduction). Given two problems A and B . If A reduces to B , \exists efficient algorithm for $B \implies \exists$ efficient algorithm for A .

10 Search Problems

Definition 10.1 (Binary relations).

$$(x, w) \in R \subseteq \{0, 1\}^* \times \{0, 1\}^*$$

11 P/NP

Definition 11.1 (P (polynomial-time)). P is the class of all R such that there exists an algorithm for solving $Decide(R)$ in time $poly(|x|)$.

Definition 11.2 (NP (Nondeterministic Polynomial-Time)). NP is the set of all R such that there exists TODO

- NP-complete: if all other search problems reduce to it
- NP-hard: