CS 170
Efficient Algorithms and Intractable Problems

Lecture Notes

Druv Pai

Contents

1 Introduction to Runtime Analysis

Example 1
Does a list have a cycle? Specifically, provide an algorithm which determines whether a list has a cycle. You can take only $O(1)$ space.

This is a bit of a loaded question. We need to know what the definition of list is; in this case we treat it as a linked list. We can provide an algorithm. Clearly this does not work, because we do not mark any node

Algorithm 1 Fake cycle detector

Input: A linked list $n_1 \to n_2 \to \dots$
Output: Whether the list has a cycle or not

```
p ← n_1
Mark p
while p exists and is not marked do
    p ← next(p)
return whether p refers to a node
```

that's not the starting node. An obvious modification is:

Algorithm 2 Real cycle detector

Input: A linked list $n_1 \to n_2 \to \dots$
Output: Whether the list has a cycle or not

```
p ← n_1
while p exists and is not marked do
    Mark p
    p ← next(p)
return whether p refers to a node
```

A more efficient way is given by

Claim 2. The algorithm is correct.

Proof. We prove correctness. If there is no cycle, the slow pointer never catches the fast one. If there is a cycle, both pointers will enter the cycle at some time. Denote d as the distance from the fast pointer to the slow pointer. Then if there is a cycle, d decreases every step, and since d is an integer, d will eventually reach 0. ■

Claim 3. The runtime is $O(n)$ to complete.

Algorithm 3 Efficient cycle detector

Input: A linked list $n_1 \to n_2 \to \dots$
Output: Whether the list has a cycle or not

$p_1 \leftarrow n_1$
$p_2 \leftarrow n_1$
while p_2 not at end of list and $p_2 \neq p_1$ **do**
  $p_1 \leftarrow \text{next}(p_1)$
  $p_2 \leftarrow \text{next}(\text{next}(p_2))$
return whether $p_1 = p_2$

Proof. There are at most n steps to get to the cycle and at most n steps to catch up within the cycle since within the circle, d decreases by 1 on each iteration. Hence there are $O(n)$ steps in total. ■

In some sense, the solutions to these puzzle questions are algorithms.
Requirements for these algorithms are correctness and efficiency.

Example 4
The Fibonacci numbers are given by $F_0 = 0$ and $F_1 = 1$ and $F_n = F_{n-1} + F_{n-1}$ for $n > 1$. A rudimentary computer program to find these numbers is

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

Claim 5. This is a terrible, but working, algorithm.

Proof. This algorithm obviously works, since it directly implements the definition of F_n.
Note that $T(n) = T(n-1)+T(n-2)+2$, so $T(n) > F_n$. Also, $F_n = F_{n-1}+F_{n-2} \geq 2F_{n-2}$ so $T(n) \geq 2^{n-2}$ by induction. This is a really bad algorithm, since $T(n) = O(2^n)$. ■

Claim 6. We can implement this in $O(n^2)$ operations.

Proof. We provide a pseudocode:

```
def fib(n):
    f_n = 1
    f_n_minus_one = 0
    for i in range(2, n + 1):
        f_n, f_n_minus_one = f_n + f_n_minus_one, f_n
    return f_n
```

Empirically, doubling n makes this program's running time increase by a factor of four, so this is $O(n^2)$. ■

In a **polynomial time algorithm**, the runtime is $O(n^k)$ for a constant k. Scaling the input by c grows the runtime bound by c^k.
In an **exponential time algorithm**, scaling the input by c raises the runtime bound to the cth power.
In asymptotic notation, we:

- Ignore constant factors. That is, $O(2n^2) = O(4n^2) = O(n^2)$.
- Ignore smaller order terms. That is, $O(n^2 + n) = O(n^2)$.
- Use any upper bound. That is, $O(n^2) \subset O(n^3)$.

Formally, $g(n)$ is $O(f(n))$ if there exists N and c such that for each $n > N$, we have $f(n) \geq c \cdot g(n)$.

The same idea holds for the lower bound $\Omega(n)$. Formally, $g(n)$ is $\Omega(f(n))$ if there exists N and c such that for each $n > N$, we have $f(n) \leq c \cdot g(n)$.

2 Divide and Conquer

Today, we cover integer multiplication using Gauss' trick, and generalize to solve recurrences with the Master Theorem. Then, we apply the Master Theorem to the problem of matrix multiplication, making a recursive algorithm that reduces the runtime from the naive $O(n^3)$ algorithm.

If we have two n bit numbers $x = x_{n-1}, \ldots, x_0$ and $y = y_{n-1}, \ldots, y_0$, the kth place of xy is the coefficient of 2^k in the binary expansion of xy (in a way that's analogous to place value in the decimal system). We obtain

$$(xy)_k = \sum_{i \leq k} x_i y_{k-i}$$

and so

$$xy = \sum_{k=0}^{2n} 2^k \sum_{i \leq k} x_i y_{k-i}$$

The runtime of this is $\sum_{k=0}^{2n} \min(k, 2n-k) = \Theta(n^2)$. The $\min(k, 2n-k)$ is just because we take k products per term until we hit the middle where $k = n$, then we start decreasing the number of products at the same rate as more terms die out/aren't considered because we've already considered the same sum.

Now we develop a recursive algorithm. Think of $x = x_L x_R = 2^{n/2} x_L + x_R$ and $y = y_L y_R = 2^{n/2} y_L + y_R$. Then

$$xy = \left(2^{n/2} x_L + x_R\right)\left(2^{n/2} y_L + y_R\right) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

We have four $n/2$-bit multiplications: $x_L y_L$, $x_L y_R$, $x_R y_L$, and $x_R y_R$. The recurrence given by this algorithm is

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

The recursion tree is a degree 4 tree of depth $\lg(n)$, with $\Theta(n^2)$ leaves or base cases.

The elementary school multiplication leads to an $\Theta(n^2)$ algorithm, but the multiplication implemented in Python empirically gives $\Theta\left(n^{\log_2(3)}\right)$ time. How is this realistically possible?

Gauss' trick starts with complex multiplication. Note that $(a + b\text{i})(c + d\text{i}) = (ac - bd) + (ad + bc)]\text{i}$. This leads to four multiplications. If we drop the i, we get $P_1 = (a+b)(c+d) = ac + bc + ad + bd$, which essentially gives us four multiplications from one. But if we do two more multiplications $P_2 = ac$ and $P_3 = bd$. Then $(a + b\text{i})(c + d\text{i}) = (P_2 - P_3) + (P_1 - P_2 - P_3)\text{i}$, which gives us three multiplications; we discount the additional additions, which are low cost.

We develop another multiplication algorithm based on this idea. Let $x = 2^{n/2} x_L + x_R$ and $y = 2^{n/2} y_L + y_R$; then $xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$. Essentially, we need the terms $x_L y_L$, $x_R y_R$, and $x_L y_R + x_R y_L$, and do $O(n)$ work to sum them up, since multiplication by a power of 2 is $O(1)$. We compute $P_1 = (x_L + x_R)(y_L + y_R) = x_L y_L + x_L y_R + x_R y_L + x_R y_R$, and $P_2 = x_L y_L$ and $P_3 = x_R y_R$. Then $x_L y_R + x_R y_L = P_1 - P_2 - P_3$, and we have all the terms we need with three multiplications. In particular,

$$xy = 2^n P_2 + 2^{n/2}(P_1 - P_2 - P_3) + P_3$$

The recurrence is

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

which gives $T(n) = \Theta\left(n^{\log_2(3)}\right)$, the same as Python.

In general, such recurrences take the form

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

There are a^i subproblems in the ith level, each subproblem is of size n/b^i, the time per problem is $c(n/b^i)^d$, and the time per level is $(a/b^d)^i cn^d$. The depth is $\log_b(n)$, and the total work is $n^d \sum_{i=0}^{\log_b(n)} (a/b^d)^i$. This leads to the conclusion:

Theorem 7 (Master Theorem)
For the recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d (\log_b(n))^k\right)$$

where a, b, d, and k are constants, and k can be 0, then if

- $d > \log_b(a)$, then $T(n) = O\left(n^d (\log_b(n))^k\right)$.
- $d < \log_b(a)$, then $T(n) = O\left(n^{\log_b(a)}\right)$.
- $d = \log_b(a)$, then $T(n) = O\left(n^d \log_b(n)^{k+1}\right)$.

Take the case of matrix multiplication. If X and Y are $n \times n$ matrices and $Z = XY$, then Z_{ij} is the dot product of the ith row of X with the jth column of Y. In particular,

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

There are n^2 entries in Z, and each entry is computed in $O(n)$ time, so the naive matrix multiplication is $O(n^2)$.

Now let's use a divide and conquer approach. Let

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}, \quad Z = XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

where A through H are $n/2 \times n/2$ quadrants of the original matrices. This gives eight matrix multiplications, giving

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

which by the Master Theorem gives $O(n^3)$.

Now, let $P_1 = A(F - H)$, $P_2 = (A + B)H$, $P_3 = (C + D)E$, $P_4 = D(G - E)$, $P_5 = (A + D)(E + H)$, $P_6 = (B - D)(G + H)$, and $P_7 = (A - C)(E + F)$. Then

$$Z = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 - P_3 + P_1 - P_7 \end{bmatrix}$$

The runtime is

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

which by the Master Theorem is $O(n^{\log_2(7)})$.

Merge sort is a sorting algorithm for an n element array. We give an algorithm:

Algorithm 4 MergeSort, Merge Sort

Input: Array A
Output: Sorted A
  if $n \leq 1$ **then**
    return A
  return Merge(MergeSort($A[0, \lfloor n/2 \rfloor]$), MergeSort($A[\lfloor n/2 \rfloor + 1, n]$))

The subroutine Merge is provided.

Algorithm 5 Merge

Input: Sorted arrays $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_n]$
Output: A sorted array C which has all elements of A and B
  $C \leftarrow []$
  while A or B is not empty **do**
    if A.Front() $<$ B.Front() **then**
      C.AddBack(A.Front())
      A.PopFront()
    else
      C.AddBack(B.Front())
      B.PopFront()
  if $|A| > 0$ **then**
    C.ExtendBy(A)
  else if $|B| > 0$ **then**
    C.ExtendBy(B)
  return C

How do we code Merge? Take the lowest element out of both lists, then pop that one out, then look at the second lowest element, and so on. Basically treat them as stacks.

The time to split is $O(n)$ for obvious reasons, and the same for merge. We have two recursive problems of size $n/2$. By Master Theorem, we have $T(n) = 2T(n/2) + \Theta(n)$ and so $T(n) = \Theta(n \log(n))$.

Algorithm 6 Iterative Merge Sort

Input: Array A
Output: Sorted A
  $q \leftarrow$ new queue
  $[a_1], \ldots, [a_n] \leftarrow A$
  for $i \in \{1, \ldots, n\}$ **do**
    q.AddBack($[a_i]$)
  while $|q| > 1$ **do**
    $l_1 \leftarrow q$.PopFront()
    $l_2 \leftarrow q$.PopFront()
    q.AddBack(Merge(l_1, l_2))
  return q.PopFront()

For iterative mergesort, we use queues. First, we break every element into its own list, then merge every pair of lists and push it to the back of the queue, then keep merging pairs of lists at the front of the queue

and push the result to the back until we get one single list. Each pass through the queue touches each element once, for $O(n)$ time. Each pass also halves the number of lists, which has $O(\log(n))$ passes; thus the iterative merge sort is $O(n\log(n))$.

In fact, all comparison sorts are at best $O(n\log(n))$. Is there a way to get better by not using a comparison sort? Radix sorts are better, by comparing the ith bit of each element of the list for each i.

Can we do better than $O(n\log(n))$ for sorting?

Theorem 8
Comparison sorts require $\Omega(n\log(n))$ comparisons.

Proof. Let π be a permutation of $1,\dots,n$. Then there are $n!$ such π, and the algorithm must be able to output each of these permutations. The algorithm can be viewed as a tree of comparisons, and after some sequence of comparisons we either reach termination or 1 permutation. Let S be a set of π at some point on the algorithm. After we do a comparison between a_i and a_j for $i \neq j$, then we break S into two subsets $S_1 \cup S_2 = S$. Our permutation is in at least one of S_1 or S_2; thus $\max(|S_1|,|S_2|) \geq |S|/2$. Each comparison divides possible outputs by at most 2. So we need at least $\log_2(n!)$ comparisons to get to 1 permutation and thus termination. By Stirling we have $\log_2(n!) = \Theta(n\log(n))$, so the algorithm is $\Omega(n\log(n))$. □

But, we could use Radix Sort, which is 'better' in some sense because it uses bitwise comparisons.

Lemma 9 (Rahul)
If we have a list of nondistinct elements $a_1,\dots,a_n$ with n_d distinct elements, then comparison sorts require $\Omega(n + n_d\log(n_d))$ comparisons.

Now we introduce an algorithm to select the kth smallest element S from a set of n elements.

Algorithm 7 SELECT, Quick Selelect Algorithm

Input: Set S and integer k
Output: The kth smallest element of S

```
if k = 1 and |S| = 1 then
    return the single element of S
Choose random pivot element v ∈ A
S_L ← all elements of S less than v
S_v ← all elements of S equal to v
S_R ← all elements of S greater than v
if k ≤ |S_L| then
    return SELECT(k, S_L)
else if k ≤ |S_L| + |S_v| then
    return v
else
    return SELECT(k − |S_L| − |S_v|, S_R)
```

Claim 10. The worst case runtime is $\Theta(n^2)$.

Proof. Let $k = n$, and the partition element is the smallest element every time. The size of the list decreases by 1 in each recursive call. The time for partition is $O(i)$ when there are i elements. Thus the runtime is $\Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$. ■

Claim 11. The average runtime is $\Theta(n)$.

Proof. The probability that a random pivot element has an index in the interval $[n/4, 3n/4]$ is $\geq 1/2$ assuming a uniform distribution of pivots, with the remaining probability not increasing the problem size. If we pick in the middle half then the subproblem size is $3n/4$, and we expect that after 2 iterations such a reduction happens. The expected time recurrence is thus

$$\text{Ex}(T(n)) = \text{Ex}\left(T\left(\frac{3}{4}n\right)\right) + O(n)$$

and by Master Theorem the runtime is $\text{Ex}(T(n)) = \Theta(n)$. ■

We can also do a deterministic pivot selection. We can find an element that "has" to be in the middle, and this helps the worst case runtime. We can implement the algorithm:

Algorithm 8 SelectPivot, Pivot Selection Algorithm

Input: Array A
Output: A heuristically chosen pivot of A

```
Split A into equally-sized groups A_1, ..., A_{⌈n/9⌉}
S ← []
for i ∈ {1, ..., ⌈n/9⌉} do
    S.AddBack(Select(A_i, ⌊|A_i|/2⌋))
return Select(S, ⌊|S|/2⌋)
```

Lemma 12
The pivot obtained by the above algorithm is at least as large as, and at least as small as, $5n/18$ of the elements.

Proof. Slides and Homework 02. ■

Consider a univariate polynomial $p(x) = \sum_{i=0}^{d} p_i x^i$ where $\deg(p) = d$. The simplest representation of this polynomial we can think of is a vector of coefficients $\begin{bmatrix} p_0 \\ \vdots \\ p_d \end{bmatrix}$.

The simplest operation we can do with this polynomial is evaluate it at a point α with the evaluation $p(\alpha) = \sum_{i=0}^{d} p_i \alpha^i = \sum_{i=0}^{d} p_i \prod_{j=1}^{i} \alpha$. This naive evaluation is $\Theta(n^2)$, where the cost is mostly from computing powers of α. How do we optimize this? Use Horner's method. Write $p(\alpha) = p_0 + \alpha(p_1 + \alpha(\cdots + \alpha p_d))$. This has d additions and d multiplications, for a runtime $\Theta(d)$. Other solutions are to use a for-loop to exponentiate α and add to the evaluated total on every iteration of the loop ($\Theta(d)$ time), or to use fast exponentiation ($\Theta(d \log(d))$ time).

How do we add two polynomials? Add their coefficients; since there are d coefficients the runtime is $\Theta(d)$. Hereafter we assume scalar operations are $\Theta(1)$.

How do we multiply two polynomials? The naive algorithm is multiply all pairs of coefficients, add the results, and simplify. This is $\Theta(d^2)$ since there are $d(d-1)/2$ products being computed in the first step. But, we can get this in $\Theta(d \log(d))$ time with the Fast Fourier Transform.

Polynomial multiplication has a direct correlation to convolution. Both operations are really useful. But to use the Fast Fourier Transform, we need a different representation of a polynomial. In particular, to specify a polynomial p of degree $d = n - 1$, we specify the points $p(\alpha_1), \ldots, p(\alpha_n)$.

Fact 13
On a polynomial, $d+1$ distinct evaluations uniquely determine a $(d+1)$-degree polynomial.

Given $p = (p(\alpha_1), \ldots, p(\alpha_n))$ and $q = (q(\alpha_1), \ldots, q(\alpha_n))$, where $n \geq d+1$, their sum is just

$$p + q = (p(\alpha_1) + q(\alpha_1), \ldots, p(\alpha_n) + q(\alpha_n))$$

This operation takes $\Theta(d)$ time.

Given $p = (p(\alpha_1), \ldots, p(\alpha_n))$ and $q = (q(\alpha_1), \ldots, q(\alpha_n))$, where $n \geq 2d+2$, their product is just

$$p \cdot q = (p(\alpha_1)q(\alpha_1), \ldots, p(\alpha_n)q(\alpha_n))$$

This operation takes $\Theta(n)$ time.

	Coefficient Representation	Evaluations Representation
Evaluation	$\Theta(d)$	Preprocessing
Addition	$\Theta(d)$	$\Theta(n)$
Multiplication	$\Theta(d^2)$	$\Theta(n)$

The idea is, if we can convert fast between representations, we can convert from coefficient representation to evaluation multiplication, do the multiplication, and then convert back. It turns out that conversion between coefficient representation and evaluation representation is the hard part. This is something we solve with the Fast Fourier Transform.

Algorithm 9 CoordinateToEvaluation(C, P), a first attempt at a polynomial coordinate-based representation into its evaluation representation

Input: C, the set of coefficients of the polynomial; P, the set of points to evaluate the polynomial at.
Output: $\{p(\alpha_i)\}_{\alpha_i \in P}$

```
if |C| = 2 then
    p_0, p_1 ← C
    α_1, ..., α_n ← P
    return {p_0 + p_1 α_i}_{i=1}^n
p_0, ..., p_d ← C
α_1, ..., α_n ← P
p_odd(x) ← Σ_{i=0}^{⌊d/2⌋} p_{2i+1} x^i
p_even(x) ← Σ_{i=0}^{⌊d/2⌋} p_{2i} x^i
u_1, ..., u_n ← CoordinateToEvaluation({p_{2i}}_{i=0}^{⌊d/2⌋}, {α_i^2}_{i=1}^n)
v_1, ..., v_n ← CoordinateToEvaluation({p_{2i+1}}_{i=0}^{⌊d/2⌋}, {α_i^2}_{i=1}^n)
for i ∈ {1, ..., n} do
    p(α_i) ← u_i + α_i · v_i          ▷ p(x) ← p_even(x^2) + x · p_odd(x^2)
return {p(α_i)}_{i=1}^n
```

How do we compute the runtime? We have the recurrence $T(n, d) = 2T(n, d/2) + \Theta(n)$ with $\Theta(n, 1) = \Theta(n)$. This evaluates to $T(n, d) = \Theta(nd)$. This is still pretty bad, so let's make some edits.

Note that the degree of the polynomials created at each subproblem is is $\lfloor n/2 \rfloor$. Also note that any appropriately-sized set of distinct points for evaluation works, so we use the points that are easiest to compute.

In particular, note that we are using α_i^2 in the subproblems. So if we pick $\alpha_{2i} = -\alpha_{2i+1}$ for $i \in \{1, \ldots, \lfloor n \rfloor 2\}$, then we only have to do $\lfloor n \rfloor 2$ evaluations. We want the set of points which we are using for

evaluation to be rich with those pairs. But as we recurse, we run out of distinct numbers which square to the same number. Or do we? Imagine that we can use complex numbers for our evaluation; then we just take square roots as we please, using complex numbers as our α_i. Every number a has two square roots, $\{\sqrt{a}, -\sqrt{a}\}$. Squaring the complex numbers in layer i (of the tree where a and b is connected if $b = a^2$ or $a = b^2$) gives only numbers in layer $i-1$. Then if we consider that the top is 1, we get that the ith layer (starting from 0) is the set of 2^ith root of unity.

Suppose $\alpha_1, \dots, \alpha_n$ are the nth roots of unity, and that n is a power of 2. Then ${\alpha^2}_1, \dots, {\alpha^2}_{n/2}$ are the $n/2$th roots of unity, and so on. The degree is decreasing by half on each recursive call, and the number of points is also being halved, if we run Algorithm 9 with the given α_i.

Let $d+1 = n$. Then $T(n) = 2T(n/2) + \Theta(n)$, which means that by the master theorem we have $T(n) = \Theta(n \log(n))$.

Definition 14
The nth roots of unity are all complex solutions to the equation $\omega^n = 1$. We list them as $\{\omega_0, \dots, \omega_{n-1}\}$.

The matrix formulation can be given by:

$$\begin{bmatrix} 1 & \omega_0 & \cdots & \omega_0^{n-1} \\ 1 & \omega_1 & \cdots & \omega_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \cdots & \omega_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} p(\omega_0) \\ p(\omega_1) \\ \vdots \\ p(\omega_{n-1}) \end{bmatrix}$$

The inverse of this problem is given by

$$\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & \omega_0 & \cdots & \omega_0^{n-1} \\ 1 & \omega_1 & \cdots & \omega_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \cdots & \omega_{n-1}^{n-1} \end{bmatrix}^{-1} \begin{bmatrix} p(\omega_0) \\ p(\omega_1) \\ \vdots \\ p(\omega_{n-1}) \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & \omega_0 & \cdots & \omega_0^{n-1} \\ 1 & \omega_1 & \cdots & \omega_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \cdots & \omega_{n-1}^{n-1} \end{bmatrix}^{\mathrm{H}} \begin{bmatrix} p(\omega_0) \\ p(\omega_1) \\ \vdots \\ p(\omega_{n-1}) \end{bmatrix}$$

We finally provide our Fast Fourier transform.

Algorithm 10 FastFourierTransform(P, n), an $\mathcal{O}(n \log(n))$ polynomial coordinate-based representation into its evaluation representation

Input: P, the set of coefficients of the polynomial; n, a power of 2
Output: $\{p(\omega_n^i)\}_{i=0}^{n-1}$

```
if |P| = 2 then
    p_0, p_1 ← P
    return {p_0 + p_1 ω_n^i}_{i=0}^{n-1}
p_0, ..., p_d ← P
p_odd(x) ← Σ_{i=0}^{⌊d/2⌋} p_{2i+1} x^i
p_even(x) ← Σ_{i=0}^{⌊d/2⌋} p_{2i} x^i
u_1, ..., u_n ← CoordinateToEvaluation({p_{2i}}_{i=0}^{⌊d/2⌋}, n/2)
v_1, ..., v_n ← CoordinateToEvaluation({p_{2i+1}}_{i=0}^{⌊d/2⌋}, n/2)
for i ∈ {0, ..., n − 1} do
    p(ω_n^i) ← u_i + ω_n^i · v_i          ▷ p(x) ← p_even(x^2) + x · p_odd(x^2)
return {p(ω_n^i)}_{i=0}^{n-1}
```

The inverse problem is done in the same way. In particular, by taking the even and odd rows and columns

of the Vandermonde matrix of ω, we obtain two subproblems, similar to the work done in Homework 2 with the Hadamard matrices.

3 Graphs

We return to the problem of graph coloring. Say that there's a graph which has the undergraduate classes as vertices, and an edge between every vertex if a certain number of people take the classes together. Then the number of exam slots booked is the same as the number of colors needed to color the graph.

Definition 15
A **graph** $G(V, E)$ has vertices V and edges $E \subseteq V \times V$.

We discuss two representations of a graph. An **adjacency list** representation is a subset of the Cartesian product, $\ell(G) \subseteq V \times V$, where a pair (p_1, p_2) of vertices is in the set if and only if there's an edge from (between, in the undirected case) p_1 and p_2.

	Adjacency Matrix	Adjacency List
Edge between u and v?	$\mathcal{O}(1)$	$\mathcal{O}(\lvert V\rvert)$
Query neighbors of u	$\mathcal{O}(\lvert V\rvert)$	$\mathcal{O}(d(u))$
Space	$\mathcal{O}\left(\lvert V\rvert^2\right)$	$\mathcal{O}(\lvert E\rvert)$

Note that in the adjacency list there's two pairs for every edge.
One central graph task is to find out which nodes are reachable from a vertex v.

Algorithm 11 ExploreGraph(G, v) – a method to traverse the graph. Finds the minatour of the graph.

Input: A graph G.
Output: Nothing.

```
Let m be a map between vertex and whether the vertex has been visited.
function Explore(G, v)
    m(v) = TRUE
    for (v, w) ∈ Neighbors(G, v) do
        if w ∉ m then
            Explore(G, w)
Explore(G, v)
```

Claim 16. Every node reachable from v is reached by Explore(G, v); further, the only nodes reached are reachable from v.

Proof. When u is visited, the stack contains nodes in a direct path from v to u. We can show this by induction on the distance from v.

If u is reachable, there's a path to it. If u is not found, then there's some earlier vertex w that's not found, otherwise it's either connected to v, which leads to a contradiction due to the recursion calls, or disconnected, in which case it's not reachable. Make this w have an edge with u. But by induction on the distance from v, we know that Explore(G, w) must have been called, and so w was found. But then Explore(G, w) calls Explore(G, u) and so we obtain a contradiction. ∎

Claim 17. The runtime is $\mathcal{O}(|V| + |E|)$.

Proof. We explore each vertex exactly once. From each vertex, we process each incident edge; each incident edge is processed twice (once from each vertex on the list). So calling EXPLORE on each vertex takes $\mathcal{O}(|V|)$ time, and exploring each edge takes $\mathcal{O}(|E|)$ time, so we have total $\mathcal{O}(|V| + |E|)$ time. ■

Algorithm 12 DEPTHFIRSTSEARCH(G) – a method to traverse the graph.

Input: A graph G.
Output: Nothing.

```
Let m be a map between vertex and whether the vertex has been visited.
Let t ← 0 be the clock counter.
function EXPLORE(G, v)
    m(v) = TRUE
    PREVISIT(G, v)
    for (v, w) ∈ NEIGHBORS(G, v) do
        if w ∉ m then
            EXPLORE(G, w)
    POSTVISIT(G, v)
function PREVISIT(G, v)
    p(v) ← t
    t ← t + 1
function POSTVISIT(G, v)
    q(v) ← t
    t ← t + 1
for v ∈ V do
    m(v) = FALSE
for v ∈ V do
    if m(v) = FALSE then
        EXPLORE(G, v)
```

If we want to output the connected components, we alter it like so:

Algorithm 13 DepthFirstSearchCC(G) – a method to traverse the graph and return the connected components.

Input: An undirected graph G.
Output: The connected components in G.

```
Let m be a map between vertex and whether the vertex has been visited.
Let c be a map between vertex and the identity of its connected component.
Let t ← 0 be the clock counter.
n ← 0
function Explore(G, v)
    m(v) = TRUE
    PreVisit(G, v)
    for (v, w) ∈ Neighbors(G, v) do
        if w ∉ m then
            Explore(G, w)
    PostVisit(G, v)
function PreVisit(G, v)
    c(v) = n
    p(v) ← t
    t ← t + 1
function PostVisit(G, v)
    q(v) ← t
    t ← t + 1
for v ∈ V do
    m(v) = FALSE
for v ∈ V do
    if m(v) = FALSE then
        Explore(G, v)
    n ← n + 1
```

Let $p(v)$ be the time that v is put on the stack and $q(v)$ be the time that v is taken off the stack; further, let $I(v) = [p(v), q(v)]$. Then for any nodes u and v, $I(u)$ and $I(v)$ are either disjoint (if they're not on the stack at the same time), or one is contained in the other (if there's some time where they're on the stack at the same time). The proof is just by the stack structure and how it's last-in-first-out.

If we explore an edge (u, v) first from u, then it's a tree edge if and only if $[p(v), q(v)] \subseteq (p(u), q(u))$; that is, u is on the stack before v. It's a 'back edge' if and only if $[p(u), q(u)] \subseteq [p(v), q(v)]$. This means that v is on the stack again, and so there's a path from v to u, and a cycle.

If $[p(u), q(u)]$ and $[p(v), q(v)]$ are disjoint, then there's no edge between u and v.

Definition 18
A **directed graph** $G(V, E)$ has vertices V and edges $E \subseteq V \times V$. An element $e \in E$ can be written as a pair (u, v) which leads from vertex u and points to vertex v.

There's some more terminology. A **root** is the starting point for our graph search. A vertex v is the **ancestor** of a vertex u if and only if there's a path from v to u. In this case, we call u a **descendant** of v; a tree edge is an edge from an ancestor to descendant, and a back edge is an edge from a descendant to an ancestor. If v is on the stack the whole time u is on the stack, then u is a descendant of v, and v is an

ancestor of u. A **cross-edge** (u, v) occurs when v enters and leaves the stack before u enters or leaves the stack.

Definition 19
A **cycle** is some sequence of vertices $v_0 \to v_1 \to \cdots \to v_k \to v_0$ where each pair of successive edges has an edge from the first vertex in the pair to the second element.

A **directed acyclic graph** is a directed graph without a cycle.

Theorem 20
A graph has a cycle if and only if there is a back edge.

Proof. If there's a back edge from u to v, then v is the ancestor to u, therefore there's a path from v to u, thus we obtain a cycle.

If there is a cycle $v_0 \to v_1 \to \cdots \to v_k \to v_0$ and v_0 is the first node explored (by relabeling), then all nodes on the cycle are explored when $\textsc{Explore}(G, v_0)$ is returned. For every $v_j \in \{v_0, \ldots, v_k\}$, we know that v_0 is on the stack whenever v_j is on the stack, so (v_k, v_0) is a back edge. Thus there's a back edge for any cycle. □

We can thus do cycle finding in linear ($\mathcal{O}(|V| + |E|)$) time.

Definition 21
A **topological sort** of a directed acyclic graph is an ordering of the vertices such that $v_i < v_j$ in the ordering if v_i has an edge to v_j.

Theorem 22
To find the topological sort, essentially, we find $q(v_i)$ for every v, and then order the vertices from maximum to minimum q.

Proof. There are no back edges since there are no cycles. If there is a tree edge (u, v), then $I(u) \supseteq I(v)$, so $q(u) > q(v)$. If there is a cross edge (u, v), then $I(u) > I(v)$ so $q(u) > q(v)$. So for every edge, (u, v) we have $q(u) > q(v)$. Thus in the reverse post ordering we have $u < v$, so u comes before v, which is correct according to the definition of the topological sort. □

A **source** is a node with no incoming edges. A **sink** is a node with no outgoing edges. The highest post order node is a source; the lowest post order node is a sink; every DAG has at least one source and sink. A naive algorithm for topological sort is to find the source, output, and repeat, which gives $\mathcal{O}(|V||E|)$.

Definition 23
In an undirected graph $G(V, E)$, we know that u and v are **connected** if there is a path from u to v (and thus a path from v to u).

Definition 24
In a directed graph $G(V, E)$, we know that u and v are **strongly connected** if there is a path from u to v and a path from v to u.

Nodes are strongly connected to themselves. If u is strongly connected to v and v is strongly connected to w, then u is strongly connected to w (due to transitivity). In particular, let C be a binary relation denoting strong connection. Then C is transitive, and in particular is an equivalence relation. Then the equivalence classes are strongly connected components.

We can quotient out by strongly connected components (treat a strongly connected component as one node) to form a DAG. This is because any cycle collapses nodes into a single strongly connected component.

How to find all the strongly connected components? Since $\textsc{Explore}(G, u)$ reaches all nodes reachable from u, we can run Explore on a node in the sink component, get all nodes in the sink component, and output all the visited nodes. Then we repeat with another strongly connected component.

Algorithm 14 Find Strongly Connected Components.

Input: A directed graph $G(V, E)$.
Output: A set of strongly connected components.

```
Let m be a map between vertex and whether the vertex has been visited.
Let c be a map between vertex and the identity of its connected component.
Let t ← 0 be the clock counter.
function Explore(G, v)
    m(v) = TRUE
    PreVisit(G, v)
    for (v, w) ∈ Neighbors(G, v) do
        if w ∉ m then
            Explore(G, w)
    PostVisit(G, v)
function PreVisit(G, v)
    p(v) ← t
    t ← t + 1
function PostVisit(G, v)
    q(v) ← t
    t ← t + 1
Reverse edges of G to make G^R.
Run DepthFirstSearch(G^R) to obtain p(·), q(·)
Let s ← ∅ be the set of connected components.
while |V| > 0 do
    s_v ← ∅
    v ← argmax_{v'∈V}(q(v'))
    Let m be generated by ExploreGraph(G, v).
    for v ∈ V do
        if m[v] = TRUE then
            s_v ← s_v ∪ v
            V ← V \ v
    s ← s ∪ {s_v}
```

This is $\mathcal{O}(|V| + |E|)$ since the initial depth-first search takes $\mathcal{O}(|V| + |E|)$ time and the exploration is also basically a depth-first search so it takes $\mathcal{O}(|V| + |E|)$ time. The node with the highest q value is in a source component.

Lemma 25
If C and C' are strongly connected components with an edge from C to C', then $\max_{v \in C} q(v) \geq \max_{v' \in C'} q(v')$.

Proof. If a node $v' \in C'$ is explored first, then EXPLORE(G, v) gets to all of C' and none of C (which would cause a cross-edge). So every node in C is explored after every node in C' (so every value of $q(v)$ is greater than every value of $q(v')$).

If a node $v \in C$ is explored first, then all of C and C' will be explored before returning from EXPLORE(G, v), so $g(v) = \max_{v' \in C \cup C'} q(v)$, so the lemma is proved. ■

The highest post numbered node is in the source strongly connected component.

We now introduce the idea of breadth first search, and the related data structures.

Say that we have a graph $G(V, E)$. Define the distance $d(u, v)$ between nodes u and v as the length of the shortest path between u and v. Then $d(s, s) = 0$ and for each $n \in N(s)$, we have that $d(s, n) = 1$. In general, the untouched neighbors of the distance x nodes are distance $x + 1$ from the source. What data structure should we use to organize this?

We use a queue.

Algorithm 15 BREADTHFIRSTSEARCHSP(G, s) - Finds shortest paths from s to $v \in V$ in an unweighted graph.

Let v be a map between nodes and whether they've been visited.
Let Q be a queue.
$d(s) \leftarrow 0$
$v(s) \leftarrow$ `TRUE`
Q.ENQUEUE(S)
while $|Q| > 0$ **do**
  $v \leftarrow Q$.DEQUEUEFRONT$()$
  for $(v, u) \in E$ **do**
    if $v(u) =$ `FALSE` **then**
      $v(u) \leftarrow$ `TRUE`
      $d(u) = d(v) + 1$
      Q.ENQUEUEBACK(u)

Claim 26. We prove by induction that the queue only has distance d and distance $d + 1$ nodes. When all distance d nodes are explored, the queue has all (and only) distance $d + 1$ nodes.

Proof. When s is first explored, we enqueue all its neighbors, which are all distance 1 nodes, so it's true for $d = 0$. Any distance $d + 1$ node has a distance d neighbor, by the definition of distance. So when this neighbor is visited, this node is added to the queue. When the last distance d node is explored, then every node at distance $d + 1$ is enumerated as the neighbors of the degree d nodes and so they're enqueued. Thus the queue has all distance $d + 1$ nodes; there are no vertices with higher degree because only $\leq d$ level nodes explored, and no vertices with lower degree since they're removed by induction. ■

This is a really bad algorithm when we use edge weights. If our edge weights l_e are all positive integers, we can change each edge into a length l_e path and use breadth first search. The time for this modification is $\mathcal{O}(\sum_e l_e)$, which is not good.

We can just use a priority queue, instead, ordered by edge weights.

Algorithm 16 DijkstraSP(G, s) - Finds shortest paths from s to $v \in V$ in a weighted graph.

Let Q be a priority queue of nodes, ordering by $d(v)$.
for $v \in V$ **do**
    $d(v) \leftarrow \infty$
Q.Enqueue$(s, 0)$
while $|Q| > 0$ **do**
    $v \leftarrow Q$.DequeueMin()
    for $(v, u) \in E$ **do**
        if $d(u) > d(v) + \ell(v, u)$ **then**
            $d(u) \leftarrow d(v) + \ell(u, v)$
            Q.InsertOrDecreaseKey$(u, d(u))$

There are $|V|$ dequeues, $|E|$ edits to the priority queue, and $\mathcal{O}(|V|)$ insertions or changes. A binary heap can do each of these in $\log(V)$ time. Thus we have time $\mathcal{O}((|V| + |E|)\log(V))$.

How do we construct a binary heap? Define the **min-heap property** as the property of a tree where for each node, the children have greater labels than their parent. To insert, we just put a heap in the first available slot, and keep swapping it with its parent until the heap property holds. To delete the minimum, we delete the root, and then grab the last element and put it at the top, then keep swapping it with its minimum child until the heap property holds.

For a d-ary heap (where each node has at most d children), if the degree is d and the depth is at most $\log_d(n)$. The edit (InsertOrDecreaseKey) is also $\log_d(n)$. The DequeueFront operation is also $d\log_d(n)$, since we need to check the minimum of each child at each level. Then Dijkstra's algorithm involves $\mathcal{O}(|V|)$ DequeueFront operations where it takes $\mathcal{O}(d\log_d(n))$ time. There are $\mathcal{O}(|E|)$ InsertOrDecreaseKey operations, which each take $\log_d(n)$ time. So we obtain our bound.

Dijkstra essentially adds the closest vertex v to the source s outside of our tree R; note that this is true as long as the edge lengths are positive, or otherwise there could be other nodes which aren't accounted for while the others get in the way. Let u be the vertex by which we access v; $d(u)$ is correct by induction, so $d(v) \leq d(u) + \ell(u, v)$ is also correct by induction. Indeed, $d(v)$ is set by some u' such that $d(v) = d(u') + \ell(u', v)$. Thus, we go through u' before v, and so when v is added, $d(v)$ is correct and corresponds to the length of the shortest path.

So Dijkstra works if the edges are all positive length. The Bellman-Ford algorithm gives the solution when there are negative edge lengths:

Algorithm 17 BellmanFordSP(G, s) - Finds shortest paths from s to $v \in V$ in a weighted graph.

$d(s) \leftarrow 0$
for $v \in V \setminus s$ **do**
    $d(v) \leftarrow \infty$
for $i \in \{1, \ldots, |V| - 1\}$ **do**
    for all $(u, v) \in E$ **do**
        $d(v) = \min(d(v), d(u) + \ell(u, v))$

The properties of this update rule are that $d(v)$ is correct if u is second to last node on shortest path, and $d(u)$ is correct, and it never makes $d(v)$ too small.

After the ith loop, $d(v)$ is correct for v with respect to i-edge shortest path, which is how we prove correctness. Since we are doing the update rule $\mathcal{O}(|E|)$ times for each of $\mathcal{O}(|V|)$ iterations, our total complexity is $\mathcal{O}(|V||E|)$.

Indeed, $d(v)$ is bounded above by the length of the i edge shortest path; this assumes the length of the shortest path is $n-1$. This is because there are no cycles (we can get arbitrarily negative shortest path distances with negative cycles). After n iterations, if some distance changes, then there must be a negative cycle.

But this problem doesn't happen in directed acyclic graphs, since there are no cycles at all (positive or negative). So for a directed acyclic graph, we can linearize (using topological sort), process each node, and update its neighbors using the update rule of Bellman-Ford.

Definition 27 (Tree)
A tree is a connected graph with no cycles.

Claim 28. A tree has $|E| = |V| - 1$.

Proof. Start with an empty graph with n components. Adding any (undirected) edge between components reduces the number of components by one; adding an undirected edge within a component creates a cycle since there are two ways to get between the vertices (the initially defined path that made the two vertices part of the same component, and the new added edge). After adding $n-1$ such edges, there is one component, and we cannot add any intra-component edges. ■

Claim 29. A connected graph with $n-1$ edges is a tree, where $n = |V|$.

Proof. If not, there is an $n-1$ edge connected graph with a cycle. If we remove an edge on a cycle, the graph is still connected, with $n-2$ edges. But we know that we need $n-1$ edges to be connected, which is a contradiction. Thus, a connected graph with $n-1$ edges has no cycle and thus is a tree. ■

Claim 30. A graph is a tree if and only if it has a unique path between every pair of nodes.

Proof. If there are two paths between two vertices u and v in the graph, then they differ at some node w and converge at another node x. Then there is a cycle $w \to x \to w$. Thus, the graph is not a tree.

If there's a unique path between every pair of nodes, then the graph is connected, and since a cycle creates a path between every pair of nodes, there are no cycles in the graph. Thus, the graph is a tree. ■

Definition 31 (Minimum Spanning Tree)
Given a weighted graph $G = (V, E, d)$, the cheapest possible subgraph that is a tree is the **minimum spanning tree** $T = (V, E_T, d)$.

If all edge weights are positive, the cheapest possible subgraph will be a tree naturally; if some are negative, then we restrict the minimum spanning tree to be a tree, although it might not necessarily be the lowest weight subgraph.

To obtain the minimum spanning tree, we can use a greedy algorithm. We sort the edges, and add the minimum edge as long as it doesn't introduce a cycle.

Fact 32 (Vertex Cut Property)
A **vertex cut** of an undirected graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that S and $V \setminus S$ are different connected components. The smallest edge across any vertex cut is in some minimum spanning tree.

Proof. Let S be a cut. Assume for the sake of contradiction that some non-minimal e is an edge in the minimum spanning tree across the cut. Then if e' is the minimal edge, we replace e by e'. This new graph is a tree, since it still has $n-1$ edges and is connected (since every path through e can now go through e', since the rest of the tree is connected); also, the weight is minimal, since $d(e) \leq d(e')$. The claim follows. ■

Algorithm 18 Kruskal's algorithm for generating a minimum spanning tree

Input: Weighted graph $G = (V, E, d)$
Output: Minimum spanning tree T of G.

```
E_S = E.SORT(d)
E_T = ∅
for e ∈ E_S do
    if no cycle in T after adding e then
        E_T ← E_T ∪ e
return T = (V, E_T, d)
```

How do we check whether a graph has a cycle? We can use the union-find data structure, to maintain the connected components. The data structure keeps a list of connected components by representing vertices as nodes of separate trees. The routine FIND(x) finds the root of vertex x's tree. The routine UNION(x, y) finds the roots of x's and y's tree, then makes the smaller tree's root a child of the larger tree's root, incrementing the larger tree's **rank** if they were equal before this. The data structure starts with every vertex being its own tree (connected component), and rank 1. The rank is basically the depth of the tree.

Claim 33. Use $\pi(x)$ to denote x's parent in its tree. If x is a root, then $\pi(x) = x$. The rank function r gives $r(x) \leq r(\pi(x))$.

Proof. Clearly, the subtree rooted at $\pi(x)$ has the subtree rooted at x as a subtree, so the first subtree has at least as much depth as the second. ■

Claim 34. A tree rooted at v with $r(v) = k$ has at least 2^k vertices.

Proof. Induction. Base case is easy; when we add 2 nodes of rank k together, they both have 2^k nodes at least, so their sum has 2^{k+1} nodes at least, which is the only way to get a tree of rank $k+1$. ■

Note that adding a smaller subtree to a larger subtree does not increase the larger subtree's rank, so there aren't necessarily exactly 2^k vertices in a subtree of rank k.

A rank k subtree has 2^k vertices at least. There are at most n nodes in total. Every rank is at most $\log(n)$, since otherwise we get more than $2^{\log(n)} = n$ vertices in total. There are only k steps in find, so $k \leq \log(n)$ is the complexity of FIND.

Definition 35 (Edge Cut)
An **edge cut** of an undirected graph $G = (V, E)$ is a set of edges E_S whose removal disconnects G. An

alternate definition of a cut is that for some partition of V into $(S, V \setminus S)$, the edge cut is the set of edges across it: $E_C = E \cup (S \times (V \setminus S))$.

The cut property, restated, says that any edge of minimal weight in an edge cut is in some minimum spanning tree (if it's of unique minimal weight, it's in all spanning trees).

Actually, this is the basis for another minimum spanning tree algorithm. The idea is to essentially run Dijkstra's algorithm, with the vertex cut being the set of processed edges.

Algorithm 19 Prim's algorithm to find the minimum spanning tree.

Input: $G = (V, E, d)$, an undirected, weighted graph, and a source node $s \in V$.
Output: A minimum spanning tree T_G.

```
c maps vertex to how close they are to the vertex cut.
p maps vertex to its parent in the tree.
Q ← priority queue using c(·) as priority
for all v ∈ V do
    c(v) ← ∞
    p(v) ← ∅
    Q.Add(v)
c(s) ← 0
p(s) ← s
while |Q| > 0 do
    v ← Q.PopMin()
    for all w ∈ Q such that (v, w) ∈ E do
        if d(v, w) < c(w) then
            c(w) ← d(v, w)
            p(w) ← v
            Q.DecreaseKey(w, d(v, w))
```

The difference between this and Dijkstra's algorithm is essentially that we compare the distance from source instead of distance from cut. The runtime is $\mathcal{O}((|V| + |E|) \log(|V|))$, the same as Dijkstra's algorithm. With a Fibonacci heap, this decreases to $\mathcal{O}(|V| \log(|V|) + |E|)$.

The algorithm is correct due to the cut property.

Now we discuss prefix codes.

Definition 36
A **prefix-free** code is an encoding scheme ϕ on an alphabet Σ where for each pair σ_1 and $\sigma_2 \in \Sigma$, we know that $\phi(\sigma_1)$ is not a prefix of $\phi(\sigma_2)$. For notation, say that the encoding alphabet is Φ.

The upshot of this is that messages sent as a prefix code are unique. This leads to compression algorithms.

Our first claim is that any prefix-free code corresponds to a full $|\Phi|$-ary tree; each internal node has $|\Phi|$ children. The proof is to assume not; then there's some node has $< |\Phi|$ children, in which case there's a common prefix to two nodes by Pidgeonhole. By completeness, there's another node with this value, so the code is not prefix-free.

We can create such a tree algorithmically.

As a proof of correctness, it's easy to see that every encoded symbol corresponds to a leaf. If S_p is a subset of nodes of T corresponding to nodes at path p in the tree, then S_p corresponds to strings where p is

Algorithm 20 Prefix-free code tree construction.

Input: Alphabet Σ, encoding scheme ϕ.
Output: Prefix-free code tree.

```
S ← Φ^n
function TreeConstruction(S)
    if |S| = 0 then
        return ∅
    V_T ← S
    for i ∈ 1, ..., |Φ| do
        S' ← {s | φ(σ_i)s ∈ S}
        T_S' ← TreeConstruction(S')
        V_T ← V_T ∪ V_T_S'
        E_T ← E_T ∪ E_T_S' ∪ (S, Root(T_S'))
    return (V_T, E_T)
T ← TreeConstruction(S)
```

a prefix. If there's an internal node S_p with $p \in S$, then p is a prefix of another codeword, a contradiction. So this encoding works.

We consider the case where we have symbol frequencies f_i. Then the general strategy, which is called **Huffman Coding**, is just that of doing this construction, merging the two lowest frequency symbols into a 'super symbol', and repeat, as long as these symbols are not siblings. This way, the shortest codes are the highest used.

We prove that this algorithm is correct, that is, that there is an optimal tree where the two lowest frequency symbols are siblings. Consider the optimal tree. Consider the lowest frequency pair of symbols (assuming no ties). If they are siblings, then the proof is claimed; otherwise, switching each with the deepest pair of siblings improves the tree, since the greater-used symbols have a shorter encoding length.

4 Dynamic Programming

Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem B we need the answer to subproblem A, then there is a (conceptual) edge from A to B. In this case, A is thought of as a smaller subproblem than B – and it will always be smaller, in an obvious sense.

Directed acyclic graphs are at the heart of dynamic programming. The special distinguishing feature of a directed acyclic graph is that its nodes can be linearized using topological sort; we can place the nodes in a line. Dynamic programming processes the nodes in linear order, solving subproblems (values at earlier nodes) and processing their solutions to find the value at the given node. In dynamic programming we are not given a directed acyclic graph; the directed acyclic graph is implicit.

Example 37 (Longest Increasing Subsequences)
In the **longest increasing subsequence** problem, the input is a sequence of numbers $a_1, \ldots, a_n$. A **subsequence** is any subset of these numbers taken in order, of the form $a_{i_1}, \ldots, a_{i_k}$, where $1 \leq i_1 < \cdots < i_k \leq n$, and an increasing subsequence is one in which the numbers are getting strictly larger.

Solution. The algorithm is to make a graph where the i^{th} node is a_i and an edge (i, j) occurs whenever it's possible for a_i and a_j to be consecutive elements in a increasing subsequence, that is, when $i < j$ and $a_i < a_j$. Clearly this graph is a directed acyclic graph, since every edge is (i, j) where $i < j$. The goal to find the longest path in this directed acyclic graph. We set $L[1] = 1$ and set $L[j] = 1 + \max(L[i] \colon (i, j) \in E)$.

This takes $\mathcal{O}(n^2)$ time for obvious reasons, and it takes a slight edit to the algorithm to recover the actual longest increasing subsequence. □

Note that generic divide and conquer algorithms deal with subproblems that are much smaller than the original problem; this is how they avoid having exponentially many nodes. Dynamic programming has exponentially many nodes, indeed, but most of them are repeats, which are memoized.

When solving a problem by dynamic programming, the most crucial question is, What are the subproblems? As long as they are chosen so that they can be solved in a linear order, it is an easy matter to write down the algorithm: iteratively solve one subproblem after the other, in order of increasing size.

Example 38

The string edit distance problem asks for the minimum number of insertions, deletions, and substitutions to turn s_1 into s_2.

Solution. What are our subproblems? What about looking at the edit distance between some prefix of the first string $s_1[1, \ldots, i]$, and some prefix of the second string $s_2[1, \ldots, j]$? Call this subproblem $E(i, j)$.

The best alignment between $s_1[1, \ldots, i]$ and $s_2[1, \ldots, j]$ can be either only $s_1[i]$, $s_2[j]$, or both. The first case incurs a cost of 1 for this column, and it remains to align $s_1[1, \ldots, i-1]$ with $s_2[1, \ldots, j]$, which is the subproblem $E(i-1, j)$. In a similar way, the second case results in the subproblem $E(i, j-1)$, with a cost of 1. In the final case, the cost is 0 if $s_1[i] = s_2[j]$, or 1 otherwise, and the subproblem is $E(i-1, j-1)$. So we can provide the definition

$$E(i, j) = \min(1 + E(i-1, j), 1 + E(i, j-1), \mathbb{1}(s_1[i] \neq s_2[j]) + E(i-1, j-1))$$

The order that we fill in these subproblems don't matter, as long as $E(i-1, j)$ and $E(i, j-1)$, and $E(i-1, j-1)$ are handled before $E(i, j)$. The very smallest cases are $E(0, j) = j$ and $E(i, 0) = i$. □

Common subproblem sequences are:

- The input is $x_1, \ldots, x_n$ and a subproblem is $x_1, \ldots, x_i$; the number of subproblems is $\mathcal{O}(n)$.
- The input is $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$; a subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$; the number of subproblems is $\mathcal{O}(mn)$.
- The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, \ldots, x_j$; the number of subproblems is $\mathcal{O}(n^2)$.
- The input is a rooted tree. A subproblem is a rooted subtree. If the tree has n nodes, there are $\mathcal{O}(n)$ subproblems.

Example 39 (Knapsack)

Storing things of weight $w_1, \ldots, w_n$ and value $v_1, \ldots, v_n$ and finding the most profitable set of things to store that take at most a total weight W is a dynamic programming problem.

This problem is solved in $\mathcal{O}(nW)$ time, where there are either limited or unlimited quantities of each item available.

Let's start with the problem that has unlimited quantities of each item. We can either look at smaller total weight allowed, or fewer items, as a subproblem. We go with the first option. Define $K(w)$ as the maximum value achievable with a knapsack of capacity w. Recursively, $K(w) = \max_{i; w_i < w} (K(w - w_i) + v_i)$. We're

done, thankfully, this works and the algorithm is easy. This fills in a one-dimensional table of length $W+1$, and each entry takes $\mathcal{O}(n)$ time to compute, so the running time is $\mathcal{O}(nW)$. The associated directed acyclic graph task is to find the longest path.

What if only one quantity is allowed? This is a totally different problem. We add another parameter j, and define $K(w,j)$ as the maximum value achievable using at most w weight and the first j items. The answer we seek is $K(w,n)$. Either j is needed to achieve the optimal value, or it isn't needed, so $K(w,j) = \max(K(w-w_j, j-1)+v_j, K(w,j-1))$. The first case is invoked only if $w_j \le w$. We can express $K(w,j)$ in terms of $K(\cdot, j-1)$. The base cases are $K(0,j) = K(w,0) = 0$. We fill out the two-dimensional table, with $W+1$ rows and $n+1$ columns, where each entry takes constant time, in $\mathcal{O}(nW)$ time.

An alternate method to dynamic programming to solve these problems is to memoize the result.

Example 40

Matrix multiplication is associative and the order we perform multiplications in can reduce computational cost. If $A_1, \ldots, A_n$ are matrices with A_i having dimensions $i \times i+1$, what is the optimal order of multiplications?

The first thing to notice is that a particular parenthesization can be represented naturally by a binary tree in which matrices correspond to leaves, the root is the final product, and interior nodes are the intermediate products. Define $C(i,j)$ as the minimum cost of multiplying $A_i \times A_{i+1} \times \cdots \times A_j$. The size of this subproblem is the number of matrix multiplications $|j-i|$. The smallest subproblem is $C(i,i) = 0$. For $j > i$, we pick a k such that $i \le k < j$, and split the product in two pieces, of the form $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots A_j$. The cost of the subtree is then the cost of these two partial products, plus the cost of combining them: $C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$. The question is to pick the optimal k:

$$C(i,j) = \min_{i \le k < j} \left(C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\right)$$

The answer to our question is to return $C(1,n)$.

Another example of dynamic programming is to find all-pairs shortest paths in a directed graph, where the subproblem is $D(i,j,k)$ and the base cases are $D(i,j,0) = \ell(i,j)$, and the recurrence is $D(i,j,k) = \min(D(i,k,k-1) + D(k,j,k-1), D(i,j,k-1))$. The idea behind this is to let k be the maximum breadth of permissible intermediate nodes. We keep expanding it until it incorporates every node.

The traveling salesman problem is the problem of computing an Eulerian tour with minimum total length. For a subset of vertices S, and $j \in S$, let $C(S,j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j. When $|S| > 1$, define $C(S,1) = \infty$. What's the recurrence for $C(S,j)$? We start at 1 and end at j, so we pick the second-to-last city as some $i \in S$, so the overall path length is the distance from 1 to i, namely $C(S \setminus \{j\}, i)$, plus the length of the final edge, so $C(S,j) = \min_{i \in S; i \ne j} C(S \setminus \{j\}, i) + \ell(i,j)$. The subproblems are ordered by $|S|$, and $C(\{1\},1) = 0$. There are at most $n \cdot 2^n$ subproblems, and each one takes linear time to solve. The total running time is $\mathcal{O}(n^2 2^n)$.

A subset of nodes S is an independent set of G if there are no edges between them. If the graph is not a tree, the problem is intractable; if it is a tree, you can solve it by dynamic programming. If $I(u)$ is the size of the largest independent set of the subtree rooted at u, then our final goal is $I(r)$; we obtain the recurrence $I(u) = \max(1 + \sum_{w \text{ grandchild of } u} I(w), \sum_{w \text{ child of } u} I(w))$. The number of subproblems is the number of vertices, and this is essentially the same as running a depth-first search, so the runtime is $\mathcal{O}(|V| + |E|)$.

5 Linear Programming

Linear programming is a broad class of optimization tasks in which both the constraints and the optimization criterion are linear functions. Many problems can be expressed in this way. We are given a set of variables – a vector variable $\mathbf{x}$ – and we want to assign real values to them to satisfy a set of linear equations and/or linear inequalities – the system $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$ – and maximize or minimize a given linear objective function – $\mathbf{c}^{\mathrm{T}}\mathbf{x}$ – given some constraint on $\mathbf{x}$ – such as $\mathbf{x} \geq \mathbf{0}$. In linear programming, such a system would be portrayed as

$$\begin{aligned} &\text{Maximize } \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ &\text{subject to } \boldsymbol{A}\mathbf{x} \leq \mathbf{b} \\ &\mathbf{x} \geq \mathbf{0} \end{aligned}$$

The set of **feasible solutions** of the linear program are all $\mathbf{x}$ which optimize the objective and obey the constraints. In general, since all constraints are linear, the feasible region is a polytope in n dimensions. The optimal $\mathbf{x}$ is achieved at a vertex of the feasible region. The only exceptions are cases in which there is no optimum; this can happen in two ways:

- The linear program is **infeasible**; that is, it's impossible to satisfy all the constraints simultaneously.
- The constraints are so loose that the feasible region is **unbounded**, and it is possible to achieve arbitrarily high or low objective values.

We can solve linear programs through the simplex method. This algorithm starts at a vertex of the polytope and repeatedly looks for an adjacent vertex with a better objective value. The local optimality implies global optimality, which can be shown through convex geometry.

A general linear program has many degrees of freedom:

- It can be either a maximization or a minimization problem.
- Its constraints can be equations and/or inequalities.
- The variables are often restricted to be nonnegative, but they can also be unrestricted in sign.

Given constraints and objective function where everything is an integer, any linear program has integer vertices.

These types of linear programs can all be reduced to each other.

- To turn a maximization problem into a minimization problem, or vice versa, just do $\mathbf{c} \mapsto -\mathbf{c}$.
- To turn an inequality $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$ into an equality, introduce some slack variables $\mathbf{s}$ such that $\boldsymbol{A}\mathbf{x} + \mathbf{s} = \mathbf{b}$ and $\mathbf{s} \geq \mathbf{0}$. To turn an inequality $\boldsymbol{A}\mathbf{x} \geq \mathbf{b}$ into an equality, introduce $\mathbf{s}$ such that $\boldsymbol{A}\mathbf{x} + \mathbf{s} = \mathbf{b}$ and $\mathbf{s} \leq \mathbf{0}$.
- To change an equality constraint into an inequality constraint is easy; $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ becomes $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$ and $\boldsymbol{A}\mathbf{x} \geq \mathbf{b}$.
- To change a variable $\mathbf{x}$ that's unrestricted in sign to a system which is restricted in sign, write $\mathbf{x} = \mathbf{x}^+ - \mathbf{x}^-$, where $\mathbf{x}^+$ and $\mathbf{x}^-$ are both $\geq \mathbf{0}$.

In this way we can turn any linear program into a variety of different forms, each of which may give us insight about the solution.

The standard form for a linear program is a minimization problem with positive variables and "greater than" inequalities.

The dual linear programming problem gives a set of multipliers for the equations in the primal linear program that form a solution. Any feasible value of the dual linear program gives an upper bound on the original primal linear program. If we find a pair of primal and dual feasible values that are equal, then they are both optimal. Here's how to construct the dual linear program:

Primal	Dual
Maximize $\mathbf{c}^T\mathbf{x}$ subject to $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$	Minimize $\mathbf{b}^T\mathbf{y}$ subject to $\boldsymbol{A}^T\mathbf{y} \geq \mathbf{c}$ and $\mathbf{y} \geq \mathbf{0}$
Maximize $\mathbf{c}^T\mathbf{x}$ subject to $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$	Minimize $\mathbf{b}^T\mathbf{y}$ subject to $\boldsymbol{A}^T\mathbf{y} = \mathbf{c}$ and $\mathbf{y} \geq \mathbf{0}$
Maximize $\mathbf{c}^T\mathbf{x}$ subject to $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$	Minimize $\mathbf{b}^T\mathbf{y}$ subject to $\boldsymbol{A}^T\mathbf{y} \geq \mathbf{c}$

In the most general cse of linear programming, we have a set I of inequalities and a set E of equalities (a total of $m = |I| + |E|$ constraints) over n variables, of which a subset N are constrained to be nonnegative. The dual has $m = |I| + |E|$ variables, of which only those corresponding to I have nonnegativity constraints. In this fashion, if the primal linear program is

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{n} c_i x_i \\ \text{subject to } & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i \in I \\ & \sum_{j=1}^{n} a_{ij} x_j = b_i \text{ for } i \in E \\ & x_j \geq 0 \text{ for } j \in N \end{aligned}$$

then the dual linear program is

$$\begin{aligned} \text{minimize } & \sum_{i=1}^{m} b_i y_i \\ \text{subject to } & \sum_{i=1}^{m} a_{ij} y_i \geq c_j \text{ for } j \in N \\ & \sum_{i} a_{ij} y_i = c_j \text{ for } j \neq N \\ & y_i \geq 0 \text{ for } i \in I \end{aligned}$$

Theorem 41 (Weak Duality)
For a primal program $\mathcal{P}$ with objective p and dual $\mathcal{D}$ with objective d, for any feasible solution $\mathbf{x}$ of $\mathcal{P}$ and any feasible solution $\mathbf{y}$ of $\mathcal{D}$, we have that $p(\mathbf{x}) \leq d(\mathbf{y})$.

Proof. If $\mathbf{x}$ is a feasible solution for $\mathcal{P}$ and $\mathbf{y}$ is a feasible solution for $\mathcal{D}$, then

$$p(\mathbf{x}) = \mathbf{c}^T\mathbf{x} \leq \mathbf{y}^T\boldsymbol{A}\mathbf{x} \leq \mathbf{y}^T\mathbf{b} = \mathbf{b}^T\mathbf{y} = d(\mathbf{y})$$

as desired. □

Theorem 42 (Strong Duality)
If a linear program $\mathcal{P}$ has a bounded value, and $\mathcal{P}$ and its dual $\mathcal{D}$ have feasible solutions, then $\mathcal{D}$ is bounded and has the same value.

Proof. We prove that if $\mathcal{P}$ is bounded and $\mathcal{P}$ and $\mathcal{D}$ are feasible, then $\mathcal{D}$ is bounded.

Weak duality says that if both $\mathcal{P}$ and $\mathcal{D}$ are both feasible, then they are both bounded, because every feasible solution to $\mathcal{D}$ gives a finite upper bound to the optimum of $\mathcal{P}$ (which then cannot be $+\infty$) and

every feasible solution to $\mathcal{P}$ gives a finite lower bound to the optimum of $\mathcal{D}$ (which then cannot be $-\infty$). Thus if $\mathcal{P}$ is bounded and $\mathcal{P}$ and $\mathcal{D}$ are feasible then $\mathcal{D}$ is bounded.

The other part of the theorem is harder. □

Theorem 43 (Complementary Slackness)
For a linear program $\mathcal{P}$ where we maximize $\mathbf{c}^{\mathrm{T}}\mathbf{x}$ subject to $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$, and its dual $\mathcal{D}$, if $\mathbf{x}$ and $\mathbf{y}$ are feasible solutions then they're both optimal if and only if $x_i(c_i - (\mathbf{y}^{\mathrm{T}}\boldsymbol{A})_i) = 0$ and $y_j(b_j - (\boldsymbol{A}\mathbf{x})_j) = 0$ for each i and j.

Proof. For each i we have $x_i\left(c_i - \left(\mathbf{y}^{\mathrm{T}}\boldsymbol{A}\right)_i\right) = 0$ if and only if $\sum_i \left(c_i - \left(\mathbf{y}^{\mathrm{T}}\boldsymbol{A}\right)_i\right)x_i = 0$. The "only if" direction is obvious; summing 0s yields 0. The "if" direction is true because $\mathbf{y}^{\mathrm{T}}\boldsymbol{A} \geq \mathbf{c}$, so for each i we have $c_i - \left(\mathbf{y}^{\mathrm{T}}\boldsymbol{A}\right)_i \leq 0$, implying that $x_i\left(c_i - \left(\mathbf{y}^{\mathrm{T}}\boldsymbol{A}\right)_i\right) \leq 0$, and summing over i yields 0 if and only if each term is 0.

Then through reversible arithmetic operations $\sum_i \left(c_i - \left(\mathbf{y}^{\mathrm{T}}\boldsymbol{A}\right)_i\right)x_i = 0$ if and only if $\mathbf{c}^{\mathrm{T}}\mathbf{x} - \mathbf{y}^{\mathrm{T}}\boldsymbol{A}\mathbf{x} = 0$ if and only if $\mathbf{c}^{\mathrm{T}}\mathbf{x} = \mathbf{y}^{\mathrm{T}}\boldsymbol{A}\mathbf{x}$.

Conversely, for each j we have $y_j\left(b_j - (\boldsymbol{A}\mathbf{x})_j\right) = 0$ if and only if $\sum_j (b_j - (\boldsymbol{A}\mathbf{x})_j)y_j = 0 = \mathbf{b}^{\mathrm{T}}\mathbf{y} - \mathbf{y}^{\mathrm{T}}\boldsymbol{A}\mathbf{x}$. The "only if" direction is obvious; summing 0 yields 0. The "if" direction is true because $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$, so for each j we have $b_j - (\boldsymbol{A}\mathbf{x})_j \geq 0$, implying that $y_j\left(b_j - (\boldsymbol{A}\mathbf{x})_j\right) \geq 0$, and summing over j yields 0 if and only if each term is 0.

Then through reversible arithmetic operations $\sum_j y_j(b_j - (\boldsymbol{A}\mathbf{x})_j) = 0$ if and only if $\mathbf{b}^{\mathrm{T}}\mathbf{x} - \mathbf{y}^{\mathrm{T}}\boldsymbol{A}\mathbf{x} = 0$ if and only if $\mathbf{b}^{\mathrm{T}}\mathbf{x} = \mathbf{y}^{\mathrm{T}}\boldsymbol{A}\mathbf{x}$.

Thus $\mathbf{c}^{\mathrm{T}}\mathbf{x} = \mathbf{b}^{\mathrm{T}}\mathbf{y}$. □

We revisit the simplex algorithm. Essentially, the simplex algorithm starts at a vertex of the polytope generated by the linear program, then moves to a better (by objective value) neighboring vertex, until there are no better neighboring vertices. In n dimensions – that is, when $\mathbf{x} \in \mathbb{R}^n$, then n constraints define a vertex. A constraint defines an affine hyperplane, and a vertex is the intersection of n hyperplanes.

To prove that the simplex algorithm is correct, we need to check whether the resulting polytope vertex gives the optimal objective value, and provide a mechanism to find which vertex to go to next.

Consider the canonical linear program:

$$\begin{aligned} &\text{maximize } \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ &\text{subject to } \boldsymbol{A}\mathbf{x} \leq \mathbf{b} \\ &\mathbf{x} \geq \mathbf{0} \end{aligned}$$

Start at the origin, supposing it is feasible. The origin is a vertex, since it's the intersection of the n constraints $\mathbf{x} \geq \mathbf{0}$ in the form $\mathbf{x} = \mathbf{0}$. If $\boldsymbol{c} \leq \mathbf{0}$, then increasing any x_i decreases the objective value, so the origin is the optimal vertex. If there's a $c_i > 0$ then increasing x_i increases the objective value, so the origin is not the optimal vertex. Then increase x_i until we hit another constraint. Then $x_i \geq 0$ is no longer tight, but the new constraint is, so we have the intersection of n constraints – that is, $n-1$ constraints of the form $x_j \geq 0$, and 1 more constraint caused by x_i being nonzero. Thus, we've reached another vertex – let this vertex be $\mathbf{v}$. Then define y_i as the distance of $\mathbf{x}$ from the affine hyperplane representing constraint i, for i in the set of n constraints whose intersection defines the current vertex. Indeed, for each such constraint i, we have $y_i = b_i - \left(\left(\boldsymbol{A}^{\mathrm{T}}\right)_i\right)^{\mathrm{T}}\mathbf{x}$ (where $\left(\left(\boldsymbol{A}^{\mathrm{T}}\right)_i\right)^{\mathrm{T}}$ is the corresponding row of $\boldsymbol{A}$) and $\mathbf{y} = (y_1, \ldots, y_n)$. Then $\mathbf{y}(\mathbf{v}) = \mathbf{0}$, and we're at the origin in this new coordinate system. We rewrite the linear program in this new coordinate system and repeat until we have the optimal solution.

If the origin is not feasible, then we find a feasible vertex by making a new linear program, where for each inequality whose index is i we introduce a positive variable z_i, defining the inequality constraints by $\left(\left(\boldsymbol{A}^{\mathrm{T}}\right)_i\right)^{\mathrm{T}}\mathbf{x} - z_i \leq b_i$, and the objective becomes $\max \sum_i -z_i$. Then there's an easy starting vertex; for some i, set $z_i = b_i$, and for $j \neq i$ set $z_j = 0$. Then we can solve this linear program by the simplex method.

If the optimum value of this linear program is 0 and achieved at vertex $(\mathbf{v}, \mathbf{z})$, then $\mathbf{z} = \mathbf{0}$, and hence $\mathbf{v}$ is a starting feasible vertex of the original linear program. If the optimum objective is positive, then the original linear program needs some nonzero z_i's to become feasible, so the original linear program is infeasible.

If more than n constraints intersect at a point, then we obtain degenerate vertices and the algorithm fails to proceed, since it's possible that we enter an infinite cycle. We can either apply Bland's anticycling rule, or perturb the problem slightly to avoid this.

The running time of the simplex algorithm is multi-step. To check optimality, it requires $\mathcal{O}(n)$ time. To find tight constraints, it requires $\mathcal{O}(m)$ time. To find the new coordinate system and rewrite the linear program in terms of this coordinate system, it takes $\mathcal{O}(nm)$ time, since we only change one constraint at a time from vertex to vertex.

Where's the dual? It's present in the new coordinate system. If $\boldsymbol{A}'$ is the matrix of tight constraints for a given vertex, and $\mathbf{b}'$ is the set of coordinates of $\mathbf{b}$ corresponding to each tight constraint, then the coordinate change is written as $\mathbf{y} = \mathbf{b}' - \boldsymbol{A}'\mathbf{x}$. Thus $\mathbf{x} = (\boldsymbol{A}')^{-1}(\mathbf{b}' - \mathbf{y})$. Then

$$\max \mathbf{c}^{\mathrm{T}}\mathbf{x} = \max\left(\mathbf{c}^{\mathrm{T}}(\boldsymbol{A}')^{-1}(\mathbf{b}' - \mathbf{y}\right) = \max \mathbf{c}^{\mathrm{T}}\left((\boldsymbol{A}')^{-1}\mathbf{b}'\right) - \mathbf{c}^{\mathrm{T}}(\boldsymbol{A}')^{-1}\mathbf{y}\Big)$$

Writing $\mathbf{z}' = \left((\boldsymbol{A}')^{-1}\right)^{\mathrm{T}}\mathbf{c}$ gives the coefficients of the new objective function. Note that they're all positive at the optimal value, so $\mathbf{z}' \geq 0$. Then for the subset of tight equations, $\boldsymbol{A}'^{\mathrm{T}}\mathbf{z}' = \boldsymbol{A}^{\mathrm{T}}\left((\boldsymbol{A}')^{-1}\right)^{\mathrm{T}}\mathbf{c} = \mathbf{c}$, so $\boldsymbol{A}'^{\mathrm{T}}\mathbf{z}' \geq \mathbf{c}$. If we set the other dual variables in $\mathbf{z}$ to 0, then we obtain $\boldsymbol{A}^{\mathrm{T}}\mathbf{z}' \geq \mathbf{c}$.

Now we deal with some problems solvable using linear programs.

Maximum Flow

Let $G = (V, E)$ be an directed, unweighted graph. Let $s \in V$ be a source vertex, and let $t \in V \setminus \{s\}$ be a sink vertex. For each edge $e \in E$, define the capacity $c_e \in \mathbb{R}_{>0}$ (and define a corresponding capacity vector $\mathbf{c} \in \mathbb{R}^{|E|}$). Then the problem is to find a flow vector $\mathbf{f} \in \mathbb{R}^{|E|}$ such that:

- for each edge e we have $0 \leq f_e \leq c_e$.
- if $u \neq s$ and $u \neq t$ then

$$\sum_{(w,u)\in E} f_{(w,u)} = \sum_{(u,w)\in E} f_{(u,w)}$$

and the quantity $\text{size}(\mathbf{f}) = \sum_{(s,u)\in E} f_{(s,u)}$ is maximized.

This is a linear program in the decision variables $\mathbf{f}$; the constraints are linear, and the optimization function is linear. This is not necessarily an integer linear program, since values of f_e can be non-integers.

There's an algorithm to solve this problem that is basically the instantiation of the simplex method. Specifically, it's the Ford-Fulkerson algorithm:

Algorithm 21 The Ford-Fulkerson maximum flow algorithm.

Input: A directed graph $G = (V, E)$, source vertex s, sink vertex t, and capacity $\mathbf{c}$.
Output: The flow vector $\mathbf{f}$ where $\text{size}(\mathbf{f}) = \sum_{(s,u)\in E} f_{(s,u)}$ is maximized.

$G^r \leftarrow G$
$\mathbf{c}^r \leftarrow \mathbf{c}$
$f_{\max} \leftarrow 0$
while $p(s,t) \leftarrow \text{DFS}(G^r, s)$ and $|p(s,t)| > 0$ **do**
  $c_{\min} \leftarrow \infty$
  for $(u,v) \in p(s,t)$ **do**
    $c_{\min} \leftarrow \min(c_{\min}, c^r_{(u,v)})$
    if $(v,u) \notin E(G^r)$ **then**
      $E(G^r) \leftarrow E(G^r) \cup \{(v,u)\}$
  for $(u,v) \in p(s,t)$ **do**
    $f_{(u,v)} \leftarrow f_{(u,v)} + c_{\min}$
    $f_{(v,u)} \leftarrow f_{(v,u)} - c_{\min}$
    $c^r_{(u,v)} \leftarrow c^r_{(u,v)} - c_{\min}$
    $c^r_{(v,u)} \leftarrow c^r_{(v,u)} + c_{\min}$
    if $c^r_{(u,v)} = 0$ **then**
      $E(G^r) \leftarrow E(G^r) \setminus \{(u,v)\}$
    if $c^r_{(v,u)} = 0$ **then**
      $E(G^r) \leftarrow E(G^r) \setminus \{(v,u)\}$
    $f_{\max} \leftarrow f_{\max} + c_{\min}$
return $(\mathbf{f}, f_{\max})$

This algorithm sometimes does not terminate; the best we can do is to show that it delivers the maximum flow when it terminates.

Indeed, the capacity constraints are fulfilled; we only increase flow to c_e for a given edge, since we modify c^r_e accordingly. Thus, at the end, for a given edge, $0 \le f_e \le c_e$. For conservation constraints, we only work with paths from s to t, and send units of flow into a vertex at the same time we send units of flow out of a vertex.

Definition 44
An s-t cut is a partition of V into two disjoint subsets (S, T) where $V = S \cup T$, and $s \in S$ and $t \in T$. Its capacity $c_{S,T} = \sum_{e\in E\cap(S\times T)} c_{(a,b)}$.

Lemma 45
The capacity of any s-t cut is an upper bound on the maximum flow through a graph.

Proof. For a valid flow, the flow out of S is the flow out of s, and the flow into T is the flow into t. Then

$$c_{S,T} = \sum_{e\in E\cap(S\times T)} c_e \ge \left(\sum_{e\in E\cap(S\times T)} f_e\right) - \left(\sum_{e\in E\cap(T\times S)} f_e\right)$$

Thus, the value of any valid flow is at most $c_{S,T}$. □

Theorem 46
In any flow network, the maximum s-t flow is equal to the minimum cut.

Proof. At the termination of the augmenting path algorithm, there's no path with residual capacity from s to t. Then if S is the set of reachable nodes form S, there's no edge in $E(G^r)$ with positive residual capacity leaving S, so all edges in E leaving S have the maximum possible flow, and no edges into S have any flow. The total flow leaving S is thus $c_{S,T}$. The flow is valid by construction. The value of the flow equals the value of $c_{S,T}$, while the optimal flow is at most $c_{S,T}$ by the previous lemma. Also, this cut is minimum, since every flow is at most the capacity of any cut, and this flow is equal to the capacity of the associated cut. So this is the maximum flow and minimum s-t cut. □

When the capacities are integers or rational numbers, the algorithm will terminate, since on every step the flow keeps increasing until it achieves the minimum cut, by at least 1 unit per iteration. Then the algorithm takes $\mathcal{O}(|E|f_{\max})$ time.

The Edmonds-Karp algorithm is identical to Ford-Fulkerson, except it uses BFS instead of DFS. The time complexity is $\mathcal{O}\left(|V||E|^2\right)$ time.

Bipartite Matching

The problem is as follows: given an undirected graph $G = (A \cup B, E)$, find the largest set of edges (a, b) between $a \in A$ and $b \in B$ without common vertices. We can solve it using a maximum flow algorithm. Indeed, for $e \in E$, assign $c_e = 1$. Make a new source vertex s and for each $v \in A$ make an edge (s, v) with capacity $c_{(s,v)} = \infty$. Make a new sink vertex t and for each $v \in B$ make an edge (v, t) with capacity $c_{(v,t)} = \infty$. Then the maximum matching is just the maximum flow through this network.

Zero Sum Games

Consider a game with N players. Each player has a strategy set $\{S_1, \ldots, S_N\}$. Let $\mathbf{u}(s_1, \ldots, s_N)$ be the function assigning payoffs to each combination of strategies.

A Nash Equilibrium on such a game is is a situation in which the payoff function ensures that no player has an incentive to change their strategy depending on the other players' strategy choices.

We deal now with the case of 2-person zero-sum games. In this case the payoff function is an $m \times n$ matrix $\boldsymbol{A}$, where $\boldsymbol{A}_{i,j}$ is the payoff if the row player plays i and the column player plays j. It's the row player's job to maximize the payoff, and the column player's job to minimize the payoff. Let $\mathbf{x}$ be a probability distribution over the m choices that the row player can make, and let $\mathbf{y}$ be a probability distribution over the n choices that the column player can make; these are mixed strategies. The payoff for the strategy pair $(\mathbf{x}, \mathbf{y})$ is

$$p(\mathbf{x}, \mathbf{y}) = \sum_i \sum_j a_{ij} x_i y_j = \mathbf{x}^{\mathrm{T}} \boldsymbol{A} \mathbf{y}$$

The row player seeks to find $\mathbf{x}$ such that $\min_{\mathbf{y}} p(\mathbf{x}, \mathbf{y})$ is maximized. Thus the row player executes the linear program

$$\begin{aligned} &\text{maximize } \min_j (\mathbf{x}^{\mathrm{T}} \boldsymbol{A})_j \\ &\text{subject to } \mathbf{x}^{\mathrm{T}} \mathbf{1} = 1 \\ &\mathbf{x} \geq \mathbf{0} \end{aligned}$$

But since the objective function isn't linear, we introduce v such that $(\mathbf{x}^{\mathrm{T}}\boldsymbol{A})_j \geq v$ for all j. The new linear program becomes

$$\begin{aligned} &\text{maximize } v \\ &\text{subject to } \mathbf{x}^{\mathrm{T}}\boldsymbol{A} \geq v \cdot \mathbf{1}^{\mathrm{T}} \\ &\mathbf{x}^{\mathrm{T}}\mathbf{1} = 1 \\ &\mathbf{x} \geq \mathbf{0} \end{aligned}$$

Similarly, the column player seeks to find $\mathbf{y}$ such that $\max_{\mathbf{x}} p(\mathbf{x}, \mathbf{y})$ is minimized. Thus the row player executes the linear program

$$\begin{aligned} &\text{minimize } \max_i (\boldsymbol{A}\mathbf{y})_i \\ &\text{subject to } \mathbf{y}^{\mathrm{T}}\mathbf{1} = 1 \\ &\mathbf{y} \geq \mathbf{0} \end{aligned}$$

But since the objective function isn't linear, we introduce w such that $(\boldsymbol{A}\mathbf{y})_i \leq w$ for all i. The new linear program becomes

$$\begin{aligned} &\text{minimize } w \\ &\text{subject to } \boldsymbol{A}\mathbf{y} \leq w \cdot \mathbf{1} \\ &\mathbf{y}^{\mathrm{T}}\mathbf{1} = 1 \\ &\mathbf{y} \geq \mathbf{0} \end{aligned}$$

The pair of strategies $(\mathbf{x}^*, \mathbf{y}^*)$ is an equilibrium pair if and only if

$$p(\mathbf{x}^*, \mathbf{y}^*) = (\mathbf{x}^*)^{\mathrm{T}}\boldsymbol{A}\mathbf{y}^* = \min_y (\mathbf{x}^*)^{\mathrm{T}}\boldsymbol{A}\mathbf{y} = \max_{\mathbf{x}} \mathbf{x}^{\mathrm{T}}\boldsymbol{A}\mathbf{y}^*$$

One can show (perhaps by contradiction) that all equilibriums have the same payoff value.

Define $R = \min_{\mathbf{y}} \max_{\mathbf{x}} \mathbf{x}^{\mathrm{T}}\boldsymbol{A}\mathbf{y} = \inf w$ and $C = \max_{\mathbf{x}} \min_{\mathbf{y}} \mathbf{x}^{\mathrm{T}}\boldsymbol{A}\mathbf{y} = \sup v$.

Weak duality implies that $R \geq C$, which makes sense since it's possible to choose an adversarial strategy for the column player. Strong duality in fact implies that $R = C$, and the attaining vertices are the equilibrium $\mathbf{x}^*$, $\mathbf{y}^*$, which follows from the duality gap.

By definition of R and C, we obtain $\max_i \boldsymbol{A}^{(i)}\mathbf{y}^* = (\mathbf{x}^*)^{\mathrm{T}}\boldsymbol{A}\mathbf{y}^*$. This is achievable because we can set $\mathbf{x}^* = \mathbf{e}_{\operatorname{argmax}(\boldsymbol{A}\mathbf{y})}$, and optimal by definition. Also, we obtain $\min_j \mathbf{x}^*\left(\boldsymbol{A}^{\mathrm{T}}\right)^{(j)} = (\mathbf{x}^*)^{\mathrm{T}}\boldsymbol{A}\mathbf{y}^*$. This is achievable because we can set $\mathbf{y}^* = \mathbf{e}_{\operatorname{argmax}((\mathbf{x}\boldsymbol{A}^{\mathrm{T}})^{\mathrm{T}})} = \mathbf{e}_{\operatorname{argmax}(\boldsymbol{A}\mathbf{x}^{\mathrm{T}})}$, and optimal by definition.

Because $\mathbf{x}$ can be $\mathbf{e}_i$ and $\mathbf{y}$ can be $\mathbf{e}_j$, an equilibrium point always exists. This is another way to show strong duality for zero-sum games.

Multiplicative Weights

Assume that there's a decision space $\mathcal{X}$ for a given period of time and n experts advising you on the correct decision. Assume that each period of time has only one associated correct decision. Define the regret of an algorithm as the difference between the loss or gain of the process using the algorithm compared to the loss or gain obtained by always following the most accurate expert.

If we know that this system has a perfect expert, that is, an expert whose output is always the time period's correct decision, then the most naive algorithm makes a total of $n-1$ mistakes before never getting it wrong again; every mistake finds an incorrect expert.

A much better algorithm ensures that we make $\lceil \log(n) \rceil$ mistakes. In particular, we start with every expert being a possible perfect expert; then, on every mistake, no expert in the incorrect majority can be

the perfect expert. Thus, on every mistake, the number of possible perfect experts drops by a factor of 2. Thus, the total number of mistakes is $\lceil \log(n) \rceil$, before we find the perfect expert and always follow their decision making, resulting in no further mistakes.

If we are not guaranteed that there exists a perfect expert, the goal shifts to do as well as the best expert b, who we assume makes at most m mistakes in total. We can again go with the majority, and penalize the inaccurate experts. We do this by assigning weights $w_i(0)$ as a function of time to each expert; initially $w_i(0) = 1$. Fix some $\epsilon > 0$, and every time t where w_i is incorrect, do $w_i(t+1) \leftarrow w_i(t) \cdot (1-\epsilon)$, or else $w_i(t+1) \leftarrow w_i(t)$. Let S_x be the set of experts who recommend decision x for a given time period; then the decision d we make at time t is given by

$$d(t) = \operatorname*{argmax}_x \frac{\sum_{i \in S_x} w_i(t)}{\sum_{i=1}^n w_i(t)}$$

Then $w_b(t) \geq (1-\epsilon)^m$ for any t, since w_b is multiplied by $1-\epsilon$ at most m times. Define the potential function $\phi(t) = \sum_i w_i(t)$, and $\phi(0) = n$. Note that if the majority is incorrect, at least half of the total weight is put under incorrect decisions; this weight is multiplied by $1-\epsilon$, so on a given time step at least $1 - \frac{\epsilon}{2}$ of the total weight remains after the weight adjustment. Thus if M is the total number of algorithm mistakes we have

$$(1-\epsilon)^m \leq \lim_{t \to \infty} \phi(t) \leq \left(1 - \frac{\epsilon}{2}\right)^M n$$

Cutting out the middle man, we obtain

$$(1-\epsilon)^m \leq \left(1 - \frac{\epsilon}{2}\right)^M n$$

Algebra and the approximation $-\epsilon - \epsilon^2 \leq \log(1-\epsilon) \leq -\epsilon$ for $\epsilon \leq \frac{1}{2}$ yields

$$M \leq 2(1+\epsilon)m + \frac{2\log(n)}{\epsilon}$$

and as $m \to \infty$ then we approach a factor of 2 of the best expert performance.

This is the best case. Consider two experts A and B, where A is correct on even days and B is correct on odd days. The best expert performance is $T/2$ mistakes, and the algorithm makes $T-1$ mistakes, so it's guaranteed to be a factor of almost 2 times as bad as the best expert.

To make this better, we can use randomization. At time t, expert i loses $\ell_i(t) \in [0,1]$.

Algorithm 22 Multiplicative Weights

Input: Experts E, time-dependent losses $\ell \in [0,1]$, and correct decision sequence c.
Output: A sequence of decisions which has an upper bound on the regret.

```
for i ∈ E do
    w_i(0) ← 1
for t ∈ ℕ \ {0} do
    for i ∈ E do
        w_i(t) ← w_i(t − 1)
    W(t) ← Σ_i w_i(t)
    Choose expert i ∈ E with probability w_i(t)/W(t)
    d(t) ← d_i(t)
    if d(t) ≠ c(t) then
        w_i(t) ← w_i(t) · (1 − ε)^{ℓ_i(t)}
return d
```

Like before, $W(t) = \sum_i w_i(t)$, and $W(0) = n$. If b is the best expert who loses L^* total, then $\lim_{t\to\infty} W(t) \geq \lim_{t\to\infty} w_b(t) \geq (1-\epsilon)^{L^*}$. Define $L(t) = \sum_i \frac{w_i(t)\ell_i(t)}{W(t)}$ as the expected loss of the algorithm at time t; this can easily be derived by iterated expectation.

Claim 47. For $\epsilon \leq \frac{1}{2}$, we have $W(t+1) \leq W(t)(1-\epsilon L(t))$.

Proof. By bash:

$$\begin{aligned} W(t+1) &= \sum_i (1-\epsilon)^{\ell_i(t)} w_i(t) \\ &\leq \sum_i (1-\epsilon\ell_i(t)) w_i(t) \\ &= \left(\sum_i w_i(t)\right) - \left(\epsilon \sum_i w_i(t)\ell_i(t)\right) \\ &= \sum_i w_i(t)\left(1 - \epsilon\frac{\sum_i w_i(t)\ell_i(t)}{\sum_i w_i(t)}\right) \\ &= W(t)(1-\epsilon L(t)) \end{aligned}$$

as claimed. ∎

Claim 48. The total loss is asymptotically within $(1+\epsilon)$ of the best expert's loss.

Proof. We obtain as trivial bounds for any T

$$(1-\epsilon)^{L^*} \leq W(T) \leq n \cdot \prod_{t=1}^{T} (1-\epsilon L(t))$$

Taking logarithms and abusing Taylor series obtains

$$\sum_{t=1}^{T} L(t) \leq (1+\epsilon)L^* + \frac{\log(n)}{\epsilon}$$

and in the limit the claim is proved. ∎

We now cover approximate equilibrium for zero-sum games. Let $C(\mathbf{x}) = \min_{\mathbf{y}} \mathbf{x}^{\mathrm{T}} \boldsymbol{A}\mathbf{y}$ and $R(\mathbf{y}) = \max_{\mathbf{x}} \mathbf{x}^{\mathrm{T}} \boldsymbol{A}\mathbf{y}$. Then if $(\mathbf{x}^*, \mathbf{y}^*)$ is an equilibrium pair, $R(\mathbf{y}) = C(\mathbf{x})$, so $R(\mathbf{y}) - C(\mathbf{x}) = 0$. For $\epsilon > 0$, define a strategy pair $(\mathbf{x}, \mathbf{y})$ to be in ϵ-approximate equilibrium if $R(\mathbf{y}) - C(\mathbf{x}) \leq \epsilon$. Recall that by weak duality $R(\mathbf{y}) \geq C(\mathbf{x})$. This means that the response $\mathbf{y}$ to $\mathbf{x}$ is within ϵ of the best possible response, and the response $\mathbf{x}$ to $\mathbf{y}$ is within ϵ of the best possible response.

Claim 49. For any $\epsilon > 0$, there exists an ϵ-approximate equilibrium.

Proof. We use multiplicative weights, and the randomized algorithm, to construct an ϵ-approximate equilibrium pair $(\mathbf{x}^*, \mathbf{y}^*)$. As setup, fix an $\epsilon > 0$, and define $R(\mathbf{x})$ and $C(\mathbf{x})$ as above. Let $T = \frac{\log(n)}{\epsilon^2}$. Let our experts be the m pure column strategies, that is, let our experts' choices be $\mathbf{e}_1, \ldots, \mathbf{e}_m$. We use multiplicative weights to produce a column distribution.

Let $\mathbf{y}(t)$ be the distribution on day t. Each day, the adversary plays the best row response to $\mathbf{y}(t)$; let $\mathbf{x}(t)$ be the indicator vector for this row. Let $\mathbf{x}^* = \frac{1}{T}\sum_{t=1}^{T} \mathbf{x}(t)$ and $\mathbf{y}^* = \operatorname{argmin}_{\mathbf{y}(t)} \mathbf{x}(t)^{\mathrm{T}} \boldsymbol{A}\mathbf{y}(t)$.

We now claim that $(\mathbf{x}^*, \mathbf{y}^*)$ are 2ϵ-optimal for the matrix A and $T = \frac{\log(n)}{\epsilon^2}$. By the choice of $\mathbf{y}^*$, the loss $L(t)$ on day t, which is $L(t) = \mathbf{x}(t)^{\mathrm{T}} \boldsymbol{A}\mathbf{y}(t)$, obeys $L(t) \geq R(\mathbf{y}^*)$. Thus the total algorithm loss $L = \sum_{t=1}^{T} L(t) \geq T \cdot R(\mathbf{y}^*)$.

Say that the best expert $\mathbf{e}_b$ incurs a loss of L^*. Since it is the best expert, it's the best column against the row play $\sum_{t=1}^{T} \mathbf{x}(t)^{\mathrm{T}} \boldsymbol{A}$, and by construction $T\mathbf{x}^* = \sum_{t=1}^{T} \mathbf{x}(t)$, so it's the best column against the row

play $T(\mathbf{x}^*)^{\mathrm{T}}\boldsymbol{A}$. Hence, $L^* \leq TC(\mathbf{x}^*) = T\min_{\mathbf{y}} \mathbf{x}^*\boldsymbol{A}\mathbf{y}$. Applying the bound $L \leq (1+\epsilon)L^* + \frac{\log(n)}{\epsilon}$ and lower-bounding L and upper bounding L^* proves the claim. □