CS170: Efficient Algorithms and Intractable Problems

UC Berkeley

Kelvin Lee

November 16, 2020

These are course notes for UC Berkeley's CS170 Efficient Algorithms and Intractable Problems, instructed by Professor Avishay Tal and Umesh Vazirani.

Contents

1 Introduction

1.1 Asymptotic Notation

Let $f, g : \mathbb{R} \to \mathbb{R}$ for the following definitions.

1.1.1 O Notation

The O notation describes upper bounds for function.

Definition 1 (O Notation)**.** If there exists a constant $c > 0$ and N such that for $x > N, |f(x)| \leq c|g(x)|$, we say that

$$f(x) = O(g(x)).$$

1.1.2 Ω Notation

The Ω notation describes lower bounds for functions.

Definition 2 (Ω Notation)**.** If there exists a constant $c > 0$ and N such that for $x > N, |f(x)| \geq c|g(x)|$. This indicates that

$$f(x) = \Omega(g(x))$$

1.1.3 Θ Notation

The Θ notation describes both upper and lower bounds for functions.

Definition 3 (Θ Notation)**.** If there exist constants $c_1, c_2 > 0$, and N such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for $x > N$, we have $f(x) = O(g(x)) = \Omega(g(x))$, which implies

$$f(x) = \Theta(g(x)).$$

Theorem 4 (Asymptotic Limit Rules)**.** If $f(n), g(n) \geq 0$:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = O(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some $c > 0 \implies f(n) = \Theta(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \Omega(g(n))$.

2 Divide and Conquer

Definition 5 (Divide and Conquer)**.** The **divide-and-conquer** strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

2.1 Karatsuba's Algorithm

Suppose we are multiplying two n-bit integers, x and y. Split up x into half, x_L and x_R, so that $x = 2^{n/2}x_{\mathrm{L}} + x_{\mathrm{R}}$.

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

- The additions take linear time, as do the multiplications by powers of 2 (left-shifts).
- It requires 4 multiplications on $n/2$ bit numbers, we get the recurrence relation

$$T(n) = 4T(n/2) + O(n).$$

However, this can be improved by using 3 multiplications:

- only need $x_L y_L, x_R y_R$, and $\left(x_L + x_R\right)\left(y_L + y_R\right)$ because

$$x_L y_R + x_R y_L = \left(x_L + x_R\right)\left(y_L + y_R\right) - x_L y_L - x_R y_R.$$

- The improved running time would then be

$$T(n) = 3T(n/2) + O(n).$$

- The constant factor improvement occurs at every level of the recursion, which dramatically lowers time bound of $O\left(n^{\log_2 3}\right)$.

Algorithm 1 Karatsuba's Algorithm

function MULT(x, y)
  $P_1 \leftarrow \text{Mult}(x_L, y_R)$
  $P_2 \leftarrow \text{Mult}(x_R, y_L)$
  $P_3 \leftarrow \text{Mult}(x_L + x_R, y_L + y_R)$
  return $2^n P_1 + 2^{n/2}\left(P_3 - P_1 - P_2\right) + P_3$

2.2 Master Theorem

A divide-and-conquer algorithm might be described by

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

Theorem 6 (Master Theorem). If $T(n) = aT(n/b) + O\left(n^d\right)$ for $a > 0, b > 1$, and $d \geq 0$, then

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

Proof. Let $T(n) = aT\left(n/b\right) + O\left(n^d\right)$. For simplicity, assume that $T(1) = 1$ and that n is a power of b. From the definition of big-O, we know that there exists constant $c > 0$ such that for sufficiently large $n, T(n) \leq aT\left(n/b\right) + cn^d$.

Suppose we have a recursive tree with $\log_b n + 1$ level. Consider level j. At level j, there are a^j subproblems. Each of size $\frac{n}{b^j}$, and will take time at most $c\left(\frac{n}{b^j}\right)^d$ to solve (this only considers the work done at level j and does not include the time it takes to solve the subsubproblems). Then the total work done at level j is at most $a^j \cdot c\left(n/b^j\right)^d = cn^d\left(\frac{a}{b^d}\right)^j$, where a is the branching factor and b^d is the shrinkage in the work needed (per subproblem). Summing over all levels, the total running time is at most $cn^d \sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j$. Consider each of the three cases:

1. $(a < b^d)$: $\frac{a}{b^d} < 1$, then

$$\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty}\left(\frac{a}{b^d}\right)^j = \frac{1}{1-\frac{a}{b^d}} = \frac{b^d}{b^d - a}.$$

   Hence, $T(n) = cn^d \cdot \frac{b^d}{b^d - a} = O\left(n^d\right)$. Intuitively, in this case the shrinkage in the work needed per subproblem is more significant, so the work done in the highest level "dominates" the other factors in the running time.

2. $(a = b^d)$: The amount of work done at each level is the same: cn^d. since there are $\log_b n$ levels, $T(n) = \left(\log_b n + 1\right) cn^d = O\left(n^d \log n\right)$.

3. $(a > b^d)$: We have

$$\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j = \frac{\left(\frac{a}{b^d}\right)^{\log_b n + 1} - 1}{\frac{a}{b^d} - 1}.$$

   Since a, b, c, d are constants,

$$T(n) = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \cdot \frac{a^{\log_b n}}{b^d \log_b n}\right) = O\left(n^d \cdot \frac{n^{\log_b a}}{n^d}\right) = O\left(n^{\log_b a}\right)$$

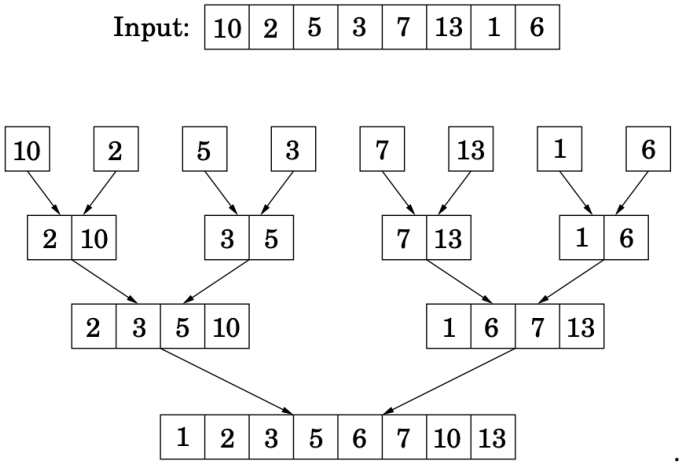
   Intuitively, here the branching factor is more significant, so the total work done at each level increases, and the leaves of the tree "dominate".

□

2.3 Mergesort

One great example of a divide-and-conquer problem is mergesort: splitting the list into two halves, recursively sort each half, and then merge the two sorted sublists.

Figure 2.4 The sequence of merge operations in `mergesort`.




```
function merge(x[1...k], y[1...l])
if k = 0:  return y[1...l]
if l = 0:  return x[1...k]
if x[1] ≤ y[1]:
  return x[1] ∘ merge(x[2...k], y[1...l])
else:
  return y[1] ∘ merge(x[1...k], y[2...l])
```

The merging part takes $O(n)$ time, so the overall runtime is

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

2.4 Inversions Counting

Now let's consider the problem of counting the number of inversions in a list.

Definition 7 (Inversion)**.** Two elements $a[i]$ and $a[j]$ form an **inversion** if $a[i] > a[j]$ and $i < j$.

We are interested in the number of inversions for several reasons:

- it tells us how sorted the list is,
- we can measure how similar two lists are by their number of inversions.

Remark. The maximum number of inversions in a list of length n is $\binom{n}{2}$ (why?).

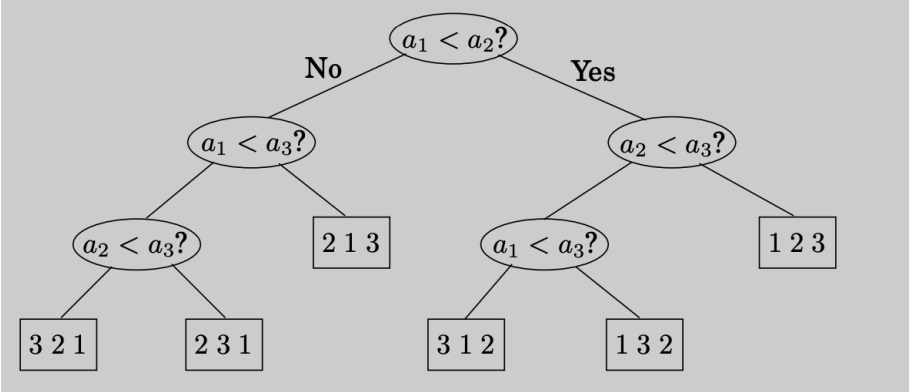
Now the question is how do we solve this by divide and conquer?

As usual, we first split the list into two halves. Then we perform two operations recursively on each half: sorting and counting. Thus, each recursive call should return the sorted half and the number of inversion in that sorted sublist. When merging the two halves, we keep track of the number of skipped elements of the right half when inserting the elements from the left half. We then sum them up to get the total number of inversions. Similar to mergesort, this merging process also takes $O(n)$ time. Hence, we still have the following runtime:

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

Now consider another question: can we do better that $O(n \log n)$?

- Consider the array $a[1, \ldots, n]$ as an abstraction.
- Imagine a decision-tree-like structure where we compare $a[i], a[j]$ for each $i, j \in 1, \ldots, n$ and $i < j$.



- In order for the algorithm to work, we need each leaf to contain exactly one permutation. Hence, the number of leaves must be at least $n!$.
- The maximum number of comparisons (height of tree) we execute is $\log n!$, which is also equivalent to the worst-case time complexity. Thus, the worst-case runtime of the algorithm would be $\Omega(\log n!)$.
- By Stirling's Approximation, we get

$$\Omega(\log n!) = \Omega\left(\log\left(\frac{n}{e}\right)^n\right) = \Omega(n \log n).$$

Remark. Recall that the Stirling's formula is

$$n! \approx \sqrt{\pi\left(2n + \frac{1}{3}\right)} \cdot n^n \cdot e^{-n}.$$

2.5 Order Statistics (Median Finding)

Median finding is important in many situations, including database tasks. A precise statement of the problem is as follows:

Given an array $a[1, ..., n]$ of n numbers and an index k $(1 \leq k \leq n)$, find the kth smallest element of a.

One obvious way to do so is to sort the list using mergesort and then select the ith element. This would take $O(n \log n)$. Here comes the question again: can we do even better?

Another approach to this problem would be using divide-and-conquer again. Here's how:

- We randomly select an element x from between $i = \frac{n}{4}$ and $i = \frac{3n}{4}$.
- Then we split the list into three categories: elements smaller than x , those equal to x (including duplicates), and those greater than x.
- The search can instantly be narrowed down to one of these sublists and we can check quickly which of these contain the median by checking k against the sizes of these sublists.
- However, the choice of x would determine the efficiency of the algorithm.

We determine x to be good if it lies within the 25th to 75th percentile of the list because then we ensure that the sublists have size at most $\frac{3n}{4}$ so that the array shrinks substantially. So x has $1/2$ chance of being good, which implies after two split operations on average, the array will shrink to at most $3/4$ of its size. Let $T(n)$ be the expected runtime. Then we have

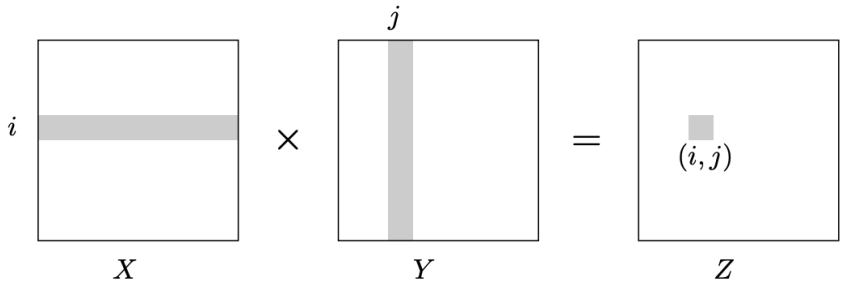
$$T(n) \leq T(3n/4) + O(n) \implies T(n) = O(n).$$

2.6 Matrix Multiplication

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j) th entry

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

To make it more visual, Z_{ij} is the dot product of the i th row of X with the j th column of Y :



The naive way indicates an $O\left(n^3\right)$ algorithm: there are n^2 entries to be computed, and each takes $O(n)$ time. What if we multiply using divide and conquer?

We can split the matrix blockwise. For example we can carve X into four $n/2 \times n/2$ blocks, and also Y :

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Then their product can be expressed in terms of these blocks.

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

One divide and conquer approach is to compute the size-n product XY by recursively computing eight size-$n/22$ products $AE, BG, AF, BH, CE, DG, CF, DH$, and then do a few $O\left(n^2\right)$ time additions. The total runtime is given by the following:

$$T(n) = 8T(n/2) + O\left(n^2\right) \implies T(n) = O(n^3).$$

It is still $O\left(n^3\right)$, the same as for the default algorithm. However, it turns out XY can be computed from just seven $n/2 \times n/2$ subproblems, here's how:

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

where

$$\begin{aligned} P_1 &= A(F-H) & P_5 &= (A+D)(E+H) \\ P_2 &= (A+B)H & P_6 &= (B-D)(G+H) \\ P_3 &= (C+D)E & P_7 &= (A-C)(E+F) \\ P_4 &= D(G-E) \end{aligned}$$

So our new runtime is now

$$T(n) = 7T(n/2) + O\left(n^2\right) \implies T(n) = O\left(n^{\log_2 7}\right) \approx O\left(n^{2.81}\right).$$

2.7 Fast Fourier Transform

We see how divide-and-conquer produces fast algorithms for integers and matrices multiplication, but what about polynomials? Let's say we have the following:

$$\begin{aligned} C(x) &= A(x) \cdot B(x) \\ &= (a_0 + a_1 x + \ldots + a_{n-1}x^{n-1})(b_0 + b_1 x + \ldots + b_{n-1}x^{n-1}) \\ &= \sum_{i,j} a_i b_j x^{i+j}. \end{aligned}$$

This multiplication process is equivalent to a convolution of two vectors and requires $O(n^2)$. How can we do better?

2.7.1 Representation of polynomials

Let's first recall a quite important property about polynomials.

Property 8. A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.

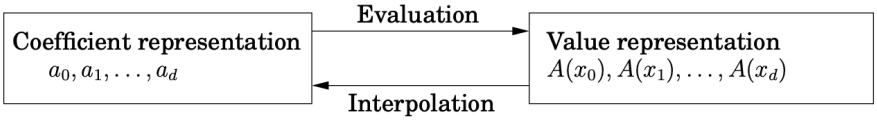
One intuitive example of this is that any two points determine a line. Hence, any $d+1$ points $x_0, x_1, \ldots, x_d$ would determine a unique degree d polynomial. We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ with two representations:

1. Its coefficients $a_0, a_1, \ldots, a_d$.
2. The values $A\left(x_0\right), A\left(x_1\right), \ldots, A\left(x_d\right)$.

The second choice would be useful for polynomial multiplication because:

- the product $C(x)$ has degree $2d$, which can be determined by its value at any $2d+1$ points;
- the value at any given point z can be given by $C(z) = A(z) \cdot B(z)$. Thus polynomial multiplication takes linear time in the value representation.

However, we expect the input polynomials and the product to be specified by coefficients. So we need a translation from coefficients to values. In particular, we evaluate the polynomial at the chosen points, then multiply in the value representation, and finally translate back to coefficients, this process is known as **interpolation**.



However, this approach actually takes quadratic time. In fact, we can actually do better by cleverly choosing the n points.

2.7.2 Evaluation by divide-and-conquer

We choose them to be positive-negative pairs, that is,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

then there will be overlaps in computations required for each $A(x_i)$ and $A(-x_i)$ because of the even powers terms. Let's explore deeper, we split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = \left(3 + 6x^2 + x^4\right) + x\left(4 + 2x^2 + 10x^4\right)$$

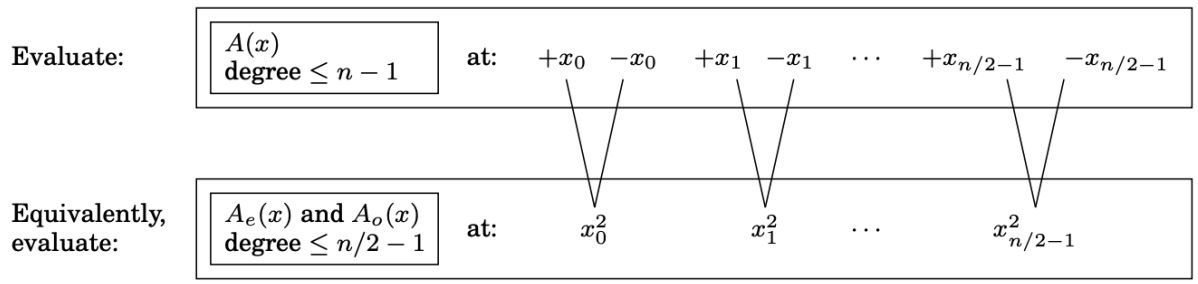
which can be written as

$$A(x) = A_e\left(x^2\right) + xA_o\left(x^2\right)$$

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$ (assume for convenience that n is even). Given points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$A(x_i) = A_e\left(x_i^2\right) + x_i A_o\left(x_i^2\right)$$
$$A(-x_i) = A_e\left(x_i^2\right) - x_i A_o\left(x_i^2\right)$$

In other words, evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ boils down to evaluating $A_e(x)$ and $A_o(x)$ (each have half the degree of $A(x)$) at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.

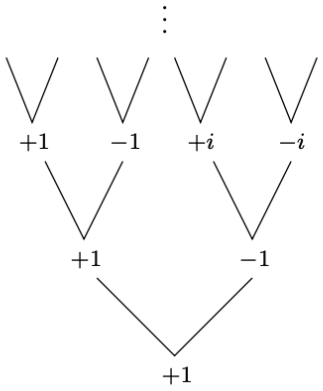


The original problem of size n is now divided into two subproblems of size $n/2$, followed by linear-time arithmetics. We get the following runtime:

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

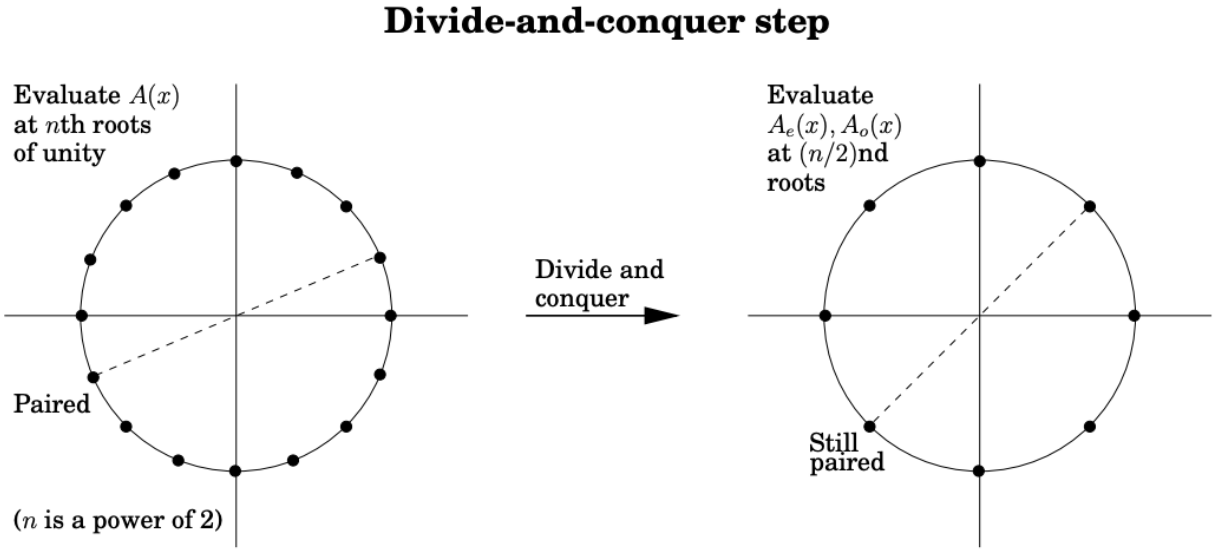
However, we now encounter a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we need the $n/2$ points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be plus-minus pairs. Hence, complex numbers come in handy now.

To figure out the choices of complex numbers, we can "reverse engineer" the process. Let's say we have a 1 at the very bottom of the recursion. Then the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$, and we continue in this fashion.



We will finally reach the desired n points. It turns out that these n points are in fact the nth roots of unity, i.e., solutions to $z^n = 1$, which are $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}$ where $\omega_n = e^{2\pi i/n}$. If n is even,

1. The nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$.
2. $\omega_n^2 = \omega_{n/2}$.



Thus, if we start with these numbers for some n that is a power of 2, then at successive levels of recursion we will have the $\left(n/2^k\right)$th roots of unity, for $k = 0, 1, 2, 3, \ldots$. All these sets of

numbers are plus-minus paired and so the divide-and-conquer approach would work perfectly, which results in the **fast Fourier transform**.

Figure 2.7 The fast Fourier transform (polynomial formulation)

```
function FFT(A,ω)
Input:  Coefficient representation of a polynomial A(x)
        of degree ≤ n−1, where n is a power of 2
        ω, an nth root of unity
Output:  Value representation A(ω^0),...,A(ω^{n−1})

if ω = 1:  return A(1)
express A(x) in the form A_e(x^2) + xA_o(x^2)
call FFT(A_e,ω^2) to evaluate A_e at even powers of ω
call FFT(A_o,ω^2) to evaluate A_o at even powers of ω
for j = 0 to n−1:
   compute A(ω^j) = A_e(ω^{2j}) + ω^j A_o(ω^{2j})

return A(ω^0),...,A(ω^{n−1})
```

2.7.3 Interpolation

Although multiplications can be easily done in the value representation, we actually need to convert it back to coefficient representation in order to recover the final polynomial. The process of recovering a polynomial from value representation is called **interpolation**.

So far we have discovered the way converting from coefficients to values in $O(n \log n)$ steps with the nth roots of unity $\left(1, \omega, \omega^2, \ldots, \omega^{n-1}\right)$.

Let's get a clearer view on the relationship between the two representations via a matrix M called the ***Vandermonde*** matrix. Suppose $A(x)$ is a polynomial of degree $\leq n-1$:

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

Here are some few properties about the ***Vandermonde*** matrix:

Property 9. If $x_0, \ldots, x_{n-1}$ are distinct numbers, then M is invertible.

Property 10. Evaluation is multiplication by M, while interpolation is multiplication by M^{-1}.

Now what values are we gonna pick for $x_0, \ldots, x_{n-1}$? As you may have guessed, the nth roots of unity! Now we have

$$M_n(\omega) = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

where n is a power of 2 and the (j,k)th entry is ω^{jk}(index starts at 0). Multiplying this matrix

maps the kth coordinate axis (vector with all zeros except for a 1 at k) onto the kth column of M. Another important fact about this matrix is that the columns are orthogonal to each other and thus can be treated as the axes of an alternative coordinate system called the ***Fourier basis***.

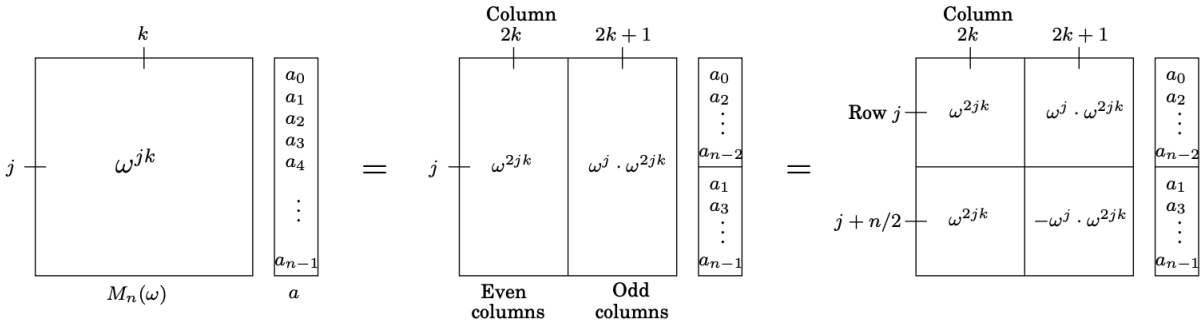
So multiplication by M represents a rotation of a vector from standard basis into the Fourier basis, which are defined by the columns of M. In other words, FFT is simply a change of basis!

If there's a way to rotate, there must also be a way to undo the rotation. In fact, this can be done by multiplying M^{-1}.

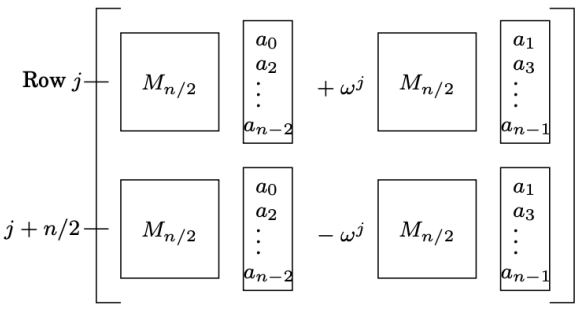
Theorem 11 (Inversion Formula)**.**

$$M_n(\omega)^{-1} = \frac{1}{n} M_n\left(\omega^{-1}\right).$$

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n th roots of unity. It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$. Using divide-and-conquer, M's columns are segregated into evens and odds:



So we actually have



In short, the product of $M_n(\omega)$ with vector $(a_0, \ldots, a_{n-1})$, a size-n problem, can be expressed in terms of two size-$n/2$ problems: the product of $M_{n/2}\left(\omega^2\right)$ with $(a_0, a_2, \ldots, a_{n-2})$ and with $(a_1, a_3, \ldots, a_{n-1})$. The runtime for this is

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

Figure 2.9 The fast Fourier transform

```
function FFT(a, ω)
Input:  An array a = (a_0, a_1, ..., a_{n-1}), for n a power of 2
        A primitive nth root of unity, ω
Output:  M_n(ω) a

if ω = 1:  return a
(s_0, s_1, ..., s_{n/2-1}) = FFT((a_0, a_2, ..., a_{n-2}), ω^2)
(s'_0, s'_1, ..., s'_{n/2-1}) = FFT((a_1, a_3, ..., a_{n-1}), ω^2)
for j = 0 to n/2 - 1:
    r_j = s_j + ω^j s'_j
    r_{j+n/2} = s_j - ω^j s'_j
return (r_0, r_1, ..., r_{n-1})
```

2.7.4 Summary on FFT

To sum up, our goal is to peform polynomial multiplication, which easier to deal with in the Fourier basis than in the standard basis. To do so,

- we first rotate vectors into the Fourier basis **(evaluation)**,
- then perform the task **(multiplication)**,
- and finally undo the rotation **(interpolation)**.
- The initial vectors are coefficient representations, while their rotated counterparts are value representations.

Finally, to efficiently switch between these, back and forth, is the province of the FFT.

3 Decompositions of Graphs

3.1 Graphs

Definition 12 (Graph)**.** A **graph** mathematical object which consists of vertices(nodes) V and edges E. We represent a graph G using the notation $G = (V, E)$.

Definition 13 (Directed and Undirected Graphs)**.** Graphs with directed edges are called **directed graphs** and graphs with undirected edges are called **undirected graphs**. Let $e = \{x, y\}$ denote an **undirected** edge, and let $e = (x, y)$ to denote a **directed** edge from x to y and vice versa.

3.1.1 Representation of Graphs

A graph can be represented using either an *adjacency matrix* or an *adjacency list*.

- **Adjacency matrix:** A $|V| \times |V|$ matrix whose (i, j)th entry is

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

  The matrix for an undirected graph is symmetric because an edge u, v can be taken in either direction.
  Pros: the presence of a particular edge can be checked in constant time, with just one memory access.
  Cons: takes $O(|V|^2)$ space. Not good for sparse graphs.

- **Adjacency list:** It consists of $|V|$ linked lists, one per vertex. The linked list for each vertex u contains v for which $(u, v) \in E$.
  Each edge appears in exactly one of the linked lists if the graph is directed or two of the lists if undirected.
  Pros: Takes $O(|V| + |E|)$ space and it is easy to iterate through all neighbors of a vertex (running down the corresponding linked list).
  Cons: Checking for a particular edge (u, v) is no longer constant time, because it requires sifting through u's adjacency list.

3.2 Depth-first Search in Undirected Graphs

Algorithm 2 Depth-first search.

```
visited[u] ← false   ∀u ∈ V
function EXPLORE(u ∈ V)
    visited[u] ← true
    for u′ ∈ neighbors(u) do
        if not visited[u′] then
            Explore(u′)
```

3.3 Depth-first Search in Directed Graphs

Algorithm 3 Depth-first search on a directed graph with enter and exit times.

```
visited[u] ← false   ∀u ∈ V
pre[u], post[u] maps to integer
clock ← 0
function TimedExplore(u ∈ V)
    visited[u] ← true
    for u → u′ in E do
        pre[u] = clock++
        if not visited[u′] then
            Explore(u′)
        post[u] = clock++
```

3.3.1 Types of Edges

Tree Edge: part of the DFS forest.

Forward Edge: leads to a non-child descendent in the tree.

Back Edge: leads to an ancestor in the tree.

Cross Edge: leads to a node that's neither descendant nor ancestor in the tree.

Theorem 14 (Parenthesis Theorem)**.** An edge $(u, v) \in E$ is a:

- **Tree/Forward Edge** if

$$\underset{u}{[}\ \underset{v}{[}\ \underset{v}{]}\ \underset{u}{]}\ .$$

$$\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$$

- **Back Edge** if

$$\underset{v}{[}\ \underset{u}{[}\ \underset{u}{]}\ \underset{v}{]}\ .$$

$$\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$$

- **Cross Edge** if

$$\underset{v}{[}\ \underset{v}{[}\ \underset{u}{]}\ \underset{u}{]}\ .$$

$$\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$$

3.4 Strongly Connected Components

Definition 15 (Strongly Connected)**.** A graph is said to be **strongly connected** if every vertex is reachable from every other vertex.

- To find an SCC, run DFS on G^R, a graph copy of G with reverse edges, to get the post-order of the graph.
- Run DFS on G starting at the max post-order of G^R (sink of G).
- No more vertices to explore implies one SCC.
- **Runtime:** $O(|V| + |E|)$.

4 Paths in Graphs

Definition 16 (Distance)**.** The **distance** between two nodes is the length of the shortest path between them.

4.1 Breadth-first Search

Although Depth-first search is able to identify all reachable vertices from a selected starting vertex s, the paths it finds may not be the most economical ones possible. Hence, we introduce **Breadth-frst Search**(BFS), an algorithm commonly used in finding shortest paths. Unlike DFS tree, BFS tree has the property that all its paths from s are the shortest possible. Therefore, it is a shortest-path tree.

Given $G = (V, E)$ and source node s, compute shortest paths from s to every node in G.

4.2 Dijkstra's Algorithm

- **Runtime:** $O\left((|V| + |E|) \log |V|\right)$.
- Calculates shortest path from s to every other vertex
- Assumes non-negative edges.

4.3 Bellman-Ford's Algorithm

- **Runtime:** $O(|V| \cdot |E|)$.
- Procedure updates all edges $|V| - 1$ times.
- Detect negative cycles with one extra round of Bellman-Ford, if distance value changes.

4.4 DAG Shortest Path

- **Runtime:** $O(|V| + |E|)$.
- Works for negative edges.
- Visits vertices in topological order.

5 Greedy Algorithms

Definition 17 (Greedy Algorithm)**.** A **greedy algorithm** is an algorithm which attempts to build up a solution for an optimization problem piece by piece by always choosing the next "locally optimal" piece.

Remark. Greedy algorithms can be disastrous for some computational tasks. It works for problems where making locally optimal choices yields a global optimum.

5.1 Minimum Spanning Trees

Definition 18 (Minimum Spanning Tree)**.** Let $G = (V, E, w)$ be a weighted undirected graph. A **minimum spanning tree** (MST) of G is a subset $T \subseteq E$ such that $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimal.

There are many problems that can be translated into a MST problem. For example, suppose V is the set of buildings in a city and E is the complete graph on V, and if $w(u, v)$ is the distance between u and v, then finding a MST would be very useful for building minimum-length infrastructure for the city. We can formulate a MST problem as follows:

Input: A connected, undirected weighted graph $G = (V, E, w)$.

Output: A spanning tree T such that the total weight

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

is minimal. Recall the definition and properties of a tree.

Definition 19 (Tree)**.** A **tree** is an undirected graph that is connected and acyclic.

Property 20. Removing a cycle edge cannot disconnect a graph.

Property 21. A tree on n nodes has $n - 1$ edges.

Property 22. Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

Property 23. An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

5.1.1 The Cut Property

Theorem 24 (Cut Property)**.** Suppose edges X are part of a minimum spanning tree of $G = (V, E, w)$. Pick any subset of nodes S for which X does not cross between S and $V\backslash S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

Proof. A **cut** is any partition of the vertices into two groups, S and $V\backslash S$.
Suppose edges X are part of some MST T of G and $e = (u, v) \notin T$(otherwise there's nothing to prove). We construct a different MST T' that contains $X \cup \{e\}$ by changing one edge. If we add the edge e to T, the resulting graph has a cycle. This cycle must cross the cut $(S, V\backslash S)$ in some other edge, so in addition to e, there must be some other edge $e' = (u', v')$ in the cycle, such that e' crosses the cut. If we remove e', then we have $T' = T \cup \{e\} - \{e'\}$. Using the properties above, we can show that T' is a tree. Also, since e is a light edge, we have $w(e) \leq w(e')$. Thus

$$w\left(T'\right) = w(T) + w(e) - w\left(e'\right) \leq w(T)$$

since T is a minimum spanning tree and T' is a spanning tree, we also have $w(T) \leq w(T')$, which implies that $w(T) = w(T')$ and so T' is a minimum spanning tree containing e. □

Remark. This property implies that it is always **safe** to add the lightest edge across any cut (that is, between a vertex in S and one in V S), provided X has no edges across the cut.

5.1.2 Kruskal's Algorithm

The famous **Kruskal's algorithm**, which finds the MST of a given graph G, can be justified with the **cut property**.

- **Main Idea:** Repeatedly add the next lightest edge that doesn't produce a cycle.
- **Runtime:** $O(|E| \log |V|)$.

Below is the algorithm:

Figure 5.4 Kruskal's minimum spanning tree algorithm.

```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

The proof of correctness is left as an exercise.

Let's think about the data structures involved for this algorithm. When building up the subgraph X, we need to somehow keep track of the connected components of X. It suffices to know which vertices are in each connected component, so the relevant information is a partition of V. Each time a new edge is added to X, two of the connected components merge. Hence, a **disjoint-set** data structure is what we need. The following are the supported operations:

- `makeset`(x) : creates a singleton set containing just x.
- `find`(x) :to which set does x belong?
- `union`(x, y) : merge the sets containing x and y.

The algorithm uses $|V|$ `makeset`, $2|E|$ `find`, and $|V| - 1$ `union` operations. The total running time is $\mathcal{O}(|E| \log |E|)$.

5.1.3 Prim's Algorithm

We now study a second MST algorithm: **Prim's algorithm**, which works similarly to the Dijkstra's algorithm.

- **Main Idea:** On each iteration, pick the lightest edge between a vertex in the current subtree S and a vertex outside S.
- **Runtime:** $O(|E| \log |V|)$.

In Prim's algorithm, The intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices. On each iteration X grows by one edge, namely, the lightest edge between a vertex in S and a vertex outside S. Equivalently, S is also growing to include the smallest cost vertex $v \notin S$:

$$\texttt{cost}(v) = \min_{u \in S} w(u, v).$$

In Prim's algorithm, the value of a node is the weight of the lightest incoming edge from set S, whereas in Dijkstra's it is the length of an entire path to that node from the starting point.

Figure 5.9 *Top:* **Prim's minimum spanning tree algorithm.** *Below:* **An illustration of Prim's algorithm, starting at node A. Also shown are a table of `cost`/`prev` values, and the final MST.**

```
procedure prim(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the array prev

for all u ∈ V:
   cost(u) = ∞
   prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0

H = makequeue(V)     (priority queue, using cost-values as keys)
while H is not empty:
   v = deletemin(H)
   for each {v, z} ∈ E:
      if cost(z) > w(v, z):
         cost(z) = w(v, z)
         prev(z) = v
         decreasekey(H, z)
```

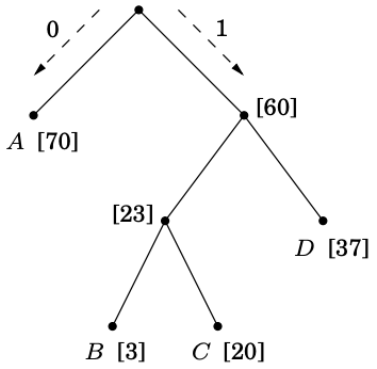
5.2 Huffman Encoding

- **Goal:** Given characters/frequencies $\{(i, f_i)\}$, encode each character in binary so the final encoding is maximally efficient.
- **Main Idea:** Find the two symbols with the smallest frequencies, say i and j, and make them children of a new node (meta-node), which then has frequency $f_i + f_j$, and repeat until one node remaining.
- The naive way of encoding is called **fixed-length code**, where each codeword has the same length. There is also a **variable-length code** where each codeword can have different lengths.
- **Cost**(average bits/character): $\sum_{i=1}^{n} f_i \ell_i$ where ℓ_i is the number of bits to encode the ith character.

Remark. One issue with the **variable-length code** is that the resulting encoding may not be uniquely decipherable. For instance, if the codewords are $0, 01, 11, 001$, the decoding of strings like 001 is ambiguous. To avoid this problem, we have the codewords to be **prefix-free**: no codeword can be a prefix of another codeword.

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

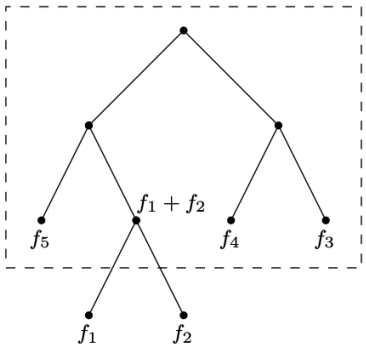
Symbol	Codeword
A	0
B	100
C	101
D	11



To find the optimal coding tree, we want a tree whose leaves each correspond to a symbol and which minimizes the overall length of the encoding,

$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth of } i \text{ th symbol in tree}).$$

Remark. Number of bits required for a symbol is its depth in the tree.




```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output:  An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n:  insert(H, i)
for k = n + 1 to 2n − 1:
  i = deletemin(H),  j = deletemin(H)
  create a node numbered k with children i, j
  f[k] = f[i] + f[j]
  insert(H, k)
```

5.3 Set Cover

- Given a set U and subsets $S_1, \ldots, S_m \subseteq U$ such that $\bigcup_{i=1}^{m} S_i = U$, find indices $I \subseteq \{1, \ldots, m\}$, with $|I|$ minimal, such that

$$\bigcup_{i \in I} S_i = U.$$

- **Goal:** Find the minimum number of subsets that cover a set $U = \{1, 2, \ldots, n\}$.
- **Main Idea:** Pick the set S_i with the largest number of uncovered elements. Repeat until all vertices are covered.

Approximation Algorithm:

1 **while** U is not empty **do**
2 Pick the largest subset S_i
3 Remove all elements of S_i from U and from the other subsets
4 **return** a list of the sets we chose

The runtime for a good implementation of this algorithm is

$$O\left(\sum_{i=1}^{m} |S_i|\right)$$

Claim. The algorithm gives a $(\ln |U| + 1)$-approximation.

Proof. Assume the optimal cover has size k. Let U_i denote the value of U (our universe set) after i iterations of the loop. Clearly, for all i, the set U_i can be covered by k sets. So one of these k sets must contain at least $\frac{|U_i|}{k}$ elements, and therefore the set chosen on line 2 has size at least $\frac{|U_i|}{k}$. Thus, we have

$$|U_{i+1}| \leq \left(1 - \frac{1}{k}\right) |U_i|$$

for all i. This implies that

$$|U_i| \leq \left(1 - \frac{1}{k}\right)^i |U_0|$$

for all i. since $1 - \frac{1}{k} \leq e^{-1/k}$, it follows that

$$|U_i| \leq e^{-i/k} |U_0|$$

In particular, letting $n = |U_0|$, we have

$$\left|U_{k(\ln n+1)}\right| < 1 \Rightarrow \left|U_{k(\ln n+1)}\right| = 0$$

Thus, the loop exits after at most $k(\ln n + 1)$ iterations. □

6 Dynamic Programming

Like the greedy and divide-and-conquer paradigms, **dynamic programming** is an algorithmic paradigm in which one solves a problem by combining the computed solutions to smaller subproblems.

High Level Approach:

- Define subproblems such that the solution to a big problem can be easily derived from the solutions to its subproblems.
- Solve all subproblems from small to large, using results from previous subproblems to solve the current subproblem.
- Both recursion with memoization (top-down) and iteration (bottom-up) approaches exist.

6.1 Shortest Path in DAGs, revisited

- **Goal:** Given a DAG, find the length of the shortest path from s to t.
- **Main Idea:** For every $v \in V$, we define $\text{dist}(v)$ as the length of the shortest path from s to v. Order of subproblems: topological order.
- **Base cases:** $\text{dist}(s) = 0$, and $\text{dist}(v) = \infty$ if $v \neq s$ is a source.
- To calculate the shortest path to v, we look at edges (u, v) going into v, we use all the pre-computed $\text{dist}(u) + \ell(u, v)$ and find the minimum:

$$\text{dist}(v) = \min_{(u,v)\in E} \{dist(u) + \ell(u, v)\}.$$

- There are $|V|$ subproblems. Each subproblems takes $O(in\_deg(v))$. Overall:

$$\sum_{v\in V} c(1 + in\_deg(v)) = c(|V| + |E|) = O(|V| + |E|).$$

Remark. We can tweak this algorithm to find the longest path by changing the min to max.

6.2 Longest Increasing Subsequences

- **Goal:** Given an array of n numbers, find the longest subsequence (non-consecutrive) that is strictly increasing.
- **Main Idea:** Try $i = 1, 2, \ldots, n$ and define $f(i)$ as the longest increasing subsequence that contains x_i.

$$f(i) = \max\left\{1, \max_{j<i; x_j<x_i} (1 + f(j))\right\}$$

- Return $\max(f(1), f(2), \ldots, f(n))$
- We have n subproblems where each can be solved from previous ones with additional time $O(n)$.
- **Runtime:** $O(n^2)$.
- **Memory:** $O(n)$.

Remark. Greedy is not optimal because we might end up with a shorter subsequence.

6.3 Edit Distance

- **Goal:** Given two strings $x[1, \ldots, n]$ and $y[1, \ldots, m]$, find the fewest number of edits to turn x into y. Edits includes:
  1. Insert character to x.
  2. Delete character from x
  3. Substitute a character for another.
- **Main Idea:** For $0 \leq i \leq n$ and $0 \leq j \leq m$ and define $f(i, j) = \text{EditDistance}(x[1, \ldots, i], y[1, \ldots, j])$. Let's look at the last character in the optimal alignment. There are three cases:
  1. $x[i]$ is aligned to _ . Return $1 + f(i-1, j)$.
  2. _ is aligned to $y[j]$. Return $1 + f(i, j-1)$.
  3. $x[i]$ is aligned to $y[j]$. Return $f(i-1, j-1) + \delta_{i,j}$.
- Finally, we return

$$f(i, j) = \min\{1 + f(i-1, j), 1 + f(i, j-1), f(i-1, j-1) + \delta_{i,j}\}.$$

- **Base cases:** $f(i, 0) = i$ and $f(0, j) = j$.
- **Runtime:** $(n+1)(m+1)$ subproblems, thus takes $O(nm)$ time.

6.4 Knapsack

- **Goal:** Given knapsack with max capacity W and n items of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$, find the most valuable combination of unique items to fit in this knapsack.
- **Main Idea:** Define $f(i, u)$ as the max value achievable with a knapsack of a capacity u and items $1, \ldots, i$. The optimal solution to $f(i, u)$ has two cases: to include item i, or to not include it in the knapsack. If we include it, we need to subtract its capacity. Thus, the recurrence is:

$$f(i, u) = \max\{f(i-1, u), f(i-1, u - w_i) + v_i\}.$$

- **Runtime:** Storing our $f(i, u)$ values in a $2D$ array of $n+1$ rows and $W+1$ columns, the algorithm fills this array from left to right, with each entry taking $O(1)$ steps, giving total runtime of $O(nW)$.

Remark. Greedy would also not work for this problem because we soon exceeds the weight limit after picking the items with the greatest value with potentially greatest weight.

7 Linear Programming

Linear Programming is used to solve optimization problems where all the constraints, as well as the *objective function*, are linear equalities or inequalities.

- In general, the **standard form** of LP consists of
  1. variables: $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^\top$
  2. objective function: $\boldsymbol{c}^T\boldsymbol{x}$.
  3. constraints: $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$
- The goal of LP is to maximize/minimize the objective function.
- A linear constraint designates a **half-space**, the region on one side of the line.
- The set of all valid solutions of a linear program is called the **feasible region**, which is convex.
- We want to find the point in this region where the objective function is maximized/minimized, which is achieved at a vertex of the region. However, there are cases where we may not find a solution:
  - **infeasible**: it is impossible to satisfy all constraints. For instance, $x \leq 1, x \geq 2$.
  - **unbounded**: constraints are too lose. For example, $\max x_0 + x_1$ subject to $x_0, x_1 \geq 0$.

7.1 Simplex Method

- **Main Idea:** The optimal point is at an intersection of the geometrical representation of the constraints. If the chosen vertex is not optimal, then the objective function can be improved by following an outgoing edge from this vertex.

7.2 Variants of Linear Programming

- LP can be a maximization or minimization problem.

7.3 Network Flow

One important application of linear programming is to solve the **maximum flow problem**: given a graph with each edge e being assigned some capacity c_e, what is the maximum flow possible to be sent from the source vertex s to t?

Definition 25 (Flow Network)**.** A **flow network** is a directed graph $G = (V, E)$ with distinguished vertices s (source) and t (sink), in which each edge $e = (u, v) \in E$ has a nonnegative capacity c_e. There are no loops in such network. If $u, v \in V$ with $(u, v) \notin E$, then $c_e = 0$.

Definition 26 (Flow)**.** Given a flow network $G = (V, E)$, a **flow** in G is a function $f : V \times V \to \mathbb{R}$ satisfying

1. **Capacity constraint**: $0 \leq f_e \leq c_e$ for all $e \in E$.
2. **Flow conservation:** for each $u \in V \backslash \{s, t\}$,

$$\underbrace{\sum_{v \in V} f_{(v,u)}}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f_{(u,v)}}_{\text{flow out of } u}.$$

The size of a flow (**flow value**) is the total quantity sent from s to t and, by the conservation principle, is equal to the quantity leaving s :

$$|f| = \sum_{(s,u)\in E} f_{(s,u)}.$$

In general, the maximum-flow problem can be viewed as the following LP:

$$\begin{array}{rl} \textbf{objective function:} & \max \sum_{e\in(s,\cdot)f_e} \\ \text{subject to} & f_e \le c_e \quad \forall e \\ & f_e \ge 0 \\ & \sum_{e=(\cdot,v)} f_e - \sum_{e=(v,\cdot)f_e} \ge 0 \quad \forall v \in V\backslash\{s,t\} \\ & -\left(\sum_{e=(\cdot,v)} f_e - \sum_{e=(v,\cdot)f_e}\right) \ge 0. \end{array}$$

Proposition 27. If the capacities are integers or rational numbers, then Ford-Fulkerson eventually terminates.

7.3.1 The Ford-Fulkerson Algorithm

The Ford–Fulkerson algorithm is an elegant solution to the maximum-flow problem. Fundamentally, it works as follows:

- Initialize flow $f = 0$ for each edge $e \in E$.
- At each iteration, increase the flow value in G by finding an **augmenting path** in the **residual network** G^f.
- Repeatedly augment the flow until the residual network has no more augmenting paths (no path from s to t is found).

More formally, suppose that we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. We define the **residual capacity** $c^f_{(u,v)}$ by

$$c^f_{(u,v)} = \begin{cases} c_{(u,v)} - f_{(u,v)} & \text{if } (u,v) \in E \\ f_{(v,u)} & \text{if } (v,u) \in E \\ 0 & \text{otherwise.} \end{cases}$$

In general, given a flow network $G = (V, E)$ and a flow f, the **residual network** of G induced by f is $G^f = \left(V, E^f\right)$, where E^f is the set of **residual edges** defined as

$$E^f = \left\{(u,v) \in V \times V : c^f_{(u,v)} > 0\right\}.$$

Remark. The back edges in residual graph can undo suboptimal flows in some edges for future iterations.

Runtime:

- Let f^* denote the maximum flow. We execute the while loop at most $|f^*|$ times since the flow value increases by at least one unit in each iteration.
- Each iteration of searching a path p takes $O(E)$ time using DFS or BFS.
- Thus, the algorithm takes $O(E \cdot |f^*|)$ time.
- BFS implementation is the **Edmonds-Karp Algorithm**.

7.3.2 The Edmonds-Karp Algorithm

The **Edmonds–Karp algorithm** is a BFS implementation of the Ford–Fulkerson algorithm in which the the augmenting path p is chosen to have minimal length among all possible augmenting paths (each edge is assigned length 1, regardless of capacity).
Runtime:

- The total number of augmentations is $O(|V| \cdot |E|)$. Thus it takes $O(|V| \cdot |E|^2)$.
- To be added.

7.3.3 Max-Flow Min-Cut Theorem

- An (s,t)-cut partitions V into two subsets S, T, where $s \in S, t \in T$.
- **Capacity of the cut C:**
$$\text{val}(C) = \sum_{e \in \{(u,v) | u \in S, v \in T\}} c_e$$
.

Theorem 28 (Max-Flow Min-Cut Theorem)**.** Maximum flow is equal to minimum cut.

$$\text{val}\left(f^*\right) = \text{val}\left(C^*\right).$$

In the residual graph, the set of vertices, X, reachable from s, yields the cut $(X, V \backslash X)$ whose capacity equals the max flow.

7.3.4 Bipartite matching

We have a bipartite graph of boys and girls, where there is an edge between b_i and g_i if boy b_i likes girl g_i. Is such a pairing possible?

Connect a source to all the boys and a sink to all the girls. Point all edges towards the sink. Set each edge's capacity to 1. The maximum flow of this graph is how many source-boy edges and girl-sink edges are saturated, which is the number of feasible couples. This is true because the max flow algorithm chooses integral flows on all edges.

Theorem 5. If all the edges have integral capacities, then the flow on each edge is integral. For LP, instead of maximizing with x_i, solve for a least upper bound by multipliers on the constraints. (duality)

7.4 Duality

- **Formulation of dual:**
  1. multiply by non-negative numbers.
  2. sum up inequalities.
- **Primal LP:**
$$\begin{gathered} \max \boldsymbol{c}^T \boldsymbol{x} \\ \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ \boldsymbol{x} \geq 0. \end{gathered}$$
- **Dual LP:**
$$\begin{gathered} \min \boldsymbol{y}^T \boldsymbol{b} \\ \boldsymbol{y}^T \boldsymbol{A} \geq \boldsymbol{c}^T \\ \boldsymbol{y} \geq 0. \end{gathered}$$

Theorem 29 (Weak Duality Theorem)**.** Let $x_1, x_2, \dots x_n$ be primal feasible and let $y_1, y_2, \dots y_n$ be dual feasible then

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{i=1}^{m} b_i y_i.$$

Theorem 30 (Strong Duality Theorem)**.** If either primal or dual has a finite optimal value, then so does the other, the optimal values coincide, and optimal solutions to both exist.

7.5 Zero-sum games and Multiplicative weight update Algorithm

7.5.1 Zero-Sum games

- **Setup:**
  - Given a $2D$ matrix where the rows are player A's moves, the columns are player B's moves, and the values are A's reward for each move.
  - Whichever player goes first has to announce their strategy first.
  - **Strategy** is the vector probabilities of playing specific moves.
- **Main Idea:** If player A goes first, player B will pick the move that minimizes A's reward. So A's best strategy will be the maximum of the minimum of B's moves. If player B goes first, player A will pick the move that maximizes A's reward. So B's best move will be the minimum of A's moves. These two equations come out to be the dual of each other.

	B_1	B_2
A_1	a	c
A_2	b	d

- A's strategy is $[x_1, x_2]$ and B's strategy is $[y_1, y_2]$.
- A picks $[x_1, x_2]$ to maximize the value $\min\{ax_1 + bx_2, cx_1 + dx_2\}$, whereas B picks $[y_1, y_2]$ to minimize the value $\max\{ay_1 + cy_2, by_1 + dy_2\}$.

Theorem 31 (Min-Max Theorem)**.**

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

Example 7.1 (Even-odd game). Given values $\{0, 1\}$, Alice and Bob are choosing one number and they both don't know what number the other person chooses. The game goes as follows: if their sum is even then Alice's payoff is the sum, otherwise the payoff will be the negative sum. Let x_0, x_1 be the probability of Alice choosing 0 and 1 respectively. Then our task is to pick x_0, x_1 to maximize $z = \min\{-x_1, -x_0 + 2x_1\}$.
The LP is as follows:

$$\begin{aligned} &\max z \\ \text{subject to} \quad &z \leq -x_1 \\ &z \leq -x_0 + 2x_1 \\ &x_0 + x_1 = 1 \\ &x_0, x_1 \geq 0. \end{aligned}$$

On the other hand, if Bob anounces his strategy y_0, y_1. Then Alice's payoff would be $\max\{-y_1, -y_0 + 2y_1\}$. Then Bob has to pick y_0, y_1 such that $\max\{-y_1, -y_0 + 2y_1\}$ is minimized. The LP is as follows:

$$\begin{aligned} &\min z \\ \text{subject to} \quad &z \geq -y_1 \\ &z \geq -y_0 + 2y_1 \\ &y_0 + y_1 = 1 \\ &y_0, y_1 \geq 0. \end{aligned}$$

It turns out that they are **duals** of each other, which implies that $\max z = \min z$. This duality only holds when the two players are playing optimally.

7.6 Multiplicative Weight Update (MWU)

The **multiplicative weights update** method is commonly used for decision making and prediction, and also widely deployed in game theory and algorithm design.

- The simplest use case is the problem of prediction from expert advice, in which a decision maker needs to iteratively decide on an expert whose advice to follow.
- We assign initial identical weights to the experts and updates these weights multiplicatively and iteratively according to performance of the expert: reducing it in case of poor performance, and increasing it otherwise.
- **Setup:** We have n experts $E_1, \ldots, E_n$. ith expert on day t: loss $\ell_i^{(t)} \in [0, 1]$. T = number of days. L^* = total loss of best expert over T days.
  1. Initialize $w_i = 1$ to $\{E_i\}_{i=1}^n$. $W = \sum_{i=1}^n w_i$.
  2. On each day we choose expert i with probability $\frac{w_i}{W}$.
  3. At the end of the day, expert j gets assigned $w_j = w_j(1 - \epsilon)^{\ell_j^{(t)}}$.

L = expected loss usffered by MWU over T days.

Theorem 32.

$$L \leq \frac{\ln n}{\epsilon} + (1+\epsilon)L^*.$$

Proof. $W(t) = \sum_{i=1}^{n} w_i^{(t)}$ and $W(0) = n$. Best expert's weight on day $T = (1-\epsilon)^{L^*}$. Thus,

$$(1-\epsilon)^{L^*} \leq W(T).$$

$$L_t = \text{expected loss suffered by MWU on day } t.$$

$$W(t+1) \leq W(t)(1-\epsilon L_t)$$

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

$$L^* \ln(1-\epsilon) \leq \ln n + \sum_{t=1}^{t} \ln(1-\epsilon L_t)$$

$$-L^*(\epsilon+\epsilon^2) \leq \ln n - \epsilon\sum_{t} L_t$$

$$L^*(1+\epsilon) \leq \frac{\ln n}{\epsilon} - L$$

$$L \leq \frac{\ln n}{\epsilon} + (1+\epsilon)L.$$

□

7.7 Reductions

Claim. Suppose M is a matching in G. Then there exists an integarl flow on $\widetilde{G}$ with $\text{val}(f) = |M|$.

Proof. Push 1 unit of flow on edges in M.

$$L(M) = \text{vertices in } L \text{ touching } M.$$

$$R(M) = \text{vertices in } R \text{ touching } M.$$

Push 1 unit of flow from s to v, $\forall v \in L(M)$. Push 1 unit of flow from s to u, $\forall u \in R(M)$. Then

$$\text{val}(f) = |L(M)| = |M|.$$

□

Claim. f is an integral flow on $\widetilde{G}$. Then there exists a matching M on G such that $|M| = \text{val}(f)$.

Proof. Since capacities in $\widetilde{G}$ are all 1, the flow on each edge could be either 0 or 1.

$$M = \{(u,v) | u \in L, v \in R, f_{(u,v)} = 1\}.$$

$$|M| = \text{val}(f).$$

M is a matching because f is a flow and every vertex in L can get at most 1 unit of flow in and every vertex in R can push at most 1 flow unit to t. □

7.7.1 The Notion of a Reduction

A problem A **reduces** to a problem B if any subroutine to solve B can be used to solve A.

7.7.2 Reduction Between the Path Problem and the Cycle problem

- There is a simple relation between the problems of finding a Hamiltonian path and a Hamiltonian cycle:
- In one direction, the Hamiltonian path problem for graph G is equivalent to the Hamiltonian cycle problem in a graph H obtained from G by adding a new vertex x and connecting x to all vertices of G.
- Thus, finding a Hamiltonian path cannot be significantly slower (in the worst case, as a function of the number of vertices) than finding a Hamiltonian cycle.
- In the other direction, the Hamiltonian cycle problem for a graph G is equivalent to the Hamiltonian path problem in the graph H obtained by copying one vertex v of G, v', that is, letting v' have the same neighbourhood as v, and by adding two dummy vertices of degree one, and connecting them with v and v', respectively.

8 NP-complete Problems

8.1 Complexity

Definition 33 (Binary Relation)**.** A **binary relation** $\mathcal{R}$ is a subset $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$. It contains elements (x, y), where x is the instance and y is the witness.
We say that a binary relation is *efficiently verifiable* if given (x, y) there exists an efficient algorithm that decides whether $(x, y) \in \mathcal{R}$.

Definition 34 (Language)**.** A **language** $\mathcal{L}$ is a subset of $\{0,1\}^*$.

Definition 35 (Language of Binary Relation)**.** The language associated with $\mathcal{R}$ is defined as

$$\mathcal{L}(\mathcal{R}) = \{x \mid (x, y) \in \mathcal{R}\}.$$

Definition 36 (Decision Problem)**.** A **decision problem** is a computation problem to which the answer is either *yes* or *no*.

$$\text{Decide}(\mathcal{R}) := \text{Given } x \text{ decide whether } \exists y \text{ such that } (x, y) \in \mathcal{R}.$$

Definition 37 (Search Problem)**.** A **search problem** asks not just whether a solution exists, but also what the solution is.

$$\text{Search}(\mathcal{R}) := \text{Given } x \text{ find } y \text{ such that } (x, y) \in \mathcal{R} \text{ if one exists.}$$

Definition 38 (Complexity Class)**.** A **complexity class** is simply a set of decision problems.

Claim. If $\mathcal{R}$ is *efficiently verifiable*, then Decide($\mathcal{R}$) can be solved in $2^{poly(|x|)}$ time.

Proof. Consider the following brute force search algorithm:
$\mathcal{A}_{\mathcal{R}}(x) :=$

1. For every possible $y \in \{0,1\}^{poly(|x|)}$, check whether $(x, y) \in \mathcal{R}$.
2. Output NO.

Thus the runtime is $2^{poly(|x|)} \cdot poly(|x|) = 2^{poly(|x|)}$. □

8.2 P and NP

Problem. Can decide($\mathcal{R}$) be solved **efficiently**, i.e. in **polynomial-time**?

Definition 39 (Polynomial-time)**.** An algorithm is **polynomial-time** if there exists a constant c such that the running time on an input of size n is $O(n^c)$.

Definition 40 (P)**.** **P** is the set of decision problems which have polynomial-time solutions. Mathematically,

$$\mathbf{P} = \{\mathcal{R} \text{ such that Decide}(\mathcal{R}) \text{ can be solved efficiently.}\}$$

Definition 41 (Nondeterministic Polynomial-time NP)**.** **NP** is the class of problems that are efficiently verifiable. Mathematically,

$$\mathbf{NP} = \{\mathcal{R} \text{ such that } \mathcal{R} \text{ is } \textit{efficiently verifiable.}\}$$

Fact 42. $\mathbf{P} \subseteq \mathbf{NP}$.

Remark. The problem of $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is a famous open problem.

8.3 NP-completeness

Definition 43 (NP-hard)**.** A problem A is **NP-hard** if $\forall B \in \mathbf{NP}, B \to A$ (B reduces to A).

Definition 44 (NP-complete)**.** A problem is **NP-complete** if it is in **NP** and is **NP**-hard.

Question. Do **NP**-complete problems exist?

Answer. Yes!

A detour to boolean circuits.

Definition 45 (Boolean Circuit)**.** A **boolean circuit** is a DAG of gates of the following types:

- INPUT gates: $X_i = 0$ or $X_i = 1$.
- AND gates
- OR gates.
- NOT gates.

Problem (Circuit Value Problem)**.** Given a boolean circuit, is its output 1?

Claim. Every efficiently solvable problem reduces to the circuit value problem.

Proof. Out of scope. Take CS172 for more details. □

Now back to our original question, here's an example of a **NP**-complete problem.

Definition 46 (CSAT)**.** CSAT is the binary relation $\mathcal{R}_{CSAT}$ such that

$$(\mathcal{C}, y) \in \mathcal{R}_{\text{CSAT}} \iff \mathcal{C}(y) = 1,$$

where $\mathcal{C}$ is a boolean circuit.

Claim. CSAT is **NP**-complete.

Proof. Using the definition, we need to show two statements:

1. ($CSAT \in \mathbf{NP}$): the process is efficiently verifiable with topological sort.
2. (CSAT is **NP**-hard): suppose we have a problem $B \in \mathbf{NP}$, then that means we have an efficient *verifier* $V_B(x, y) = \mathcal{R}_B(x, y)$.
   - Preprocess: output circuit $\mathcal{C}$ for $V_B(x, \cdot)$ from the previous claim shown.
   - Pass in $\mathcal{C}$ to CSAT, which would output y, which is exactly the witness for B.

□

Definition 47 (Formulas vs Circuits)**. Circuits** are DAGs of gates whereas **formulas** are trees of gates. The difference is that the latter does not reuse outputs from previous gates.

Definition 48 (SAT)**.** Also known as SATISFIABILITY. Given a boolean formula $\phi : \{0,1\}^n \to \{0,1\}$, decide if $\exists y \in \{0,1\}^n$ such that $\phi(y) = 1$.

An instance of SAT looks as follows:

$$(x \vee y \vee z) \wedge (x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (z \vee \bar{x}) \wedge (\bar{x} \vee \bar{y} \vee \bar{z}).$$

This is a Boolean formula in **conjunctive normal form (CNF)**: a collection of clauses with each consisting of the disjunction ($\vee$) of several literals, where a literal is either a Boolean variable or the negation of one.

Theorem 49 (Cook-Levin)**.** SAT is **NP**-complete.

Proof. We first show that $SAT \in \mathbf{NP}$, i.e., given ϕ, y, is there an algorithm to decide if $\phi(y) = 1$? Yes! In fact, we can just directly evaluate the formula. Thus, we know that SAT must be in **NP**. Now we show that SAT is **NP**-hard, i.e., for each $B \in \mathbf{NP}$, we can reduce B to SAT. It suffices to show that CSAT reduces to SAT since we know that CSAT is **NP**-complete from above. □

Definition 50 (3SAT)**.** Given a CNF formula ϕ on inputs $x_1, \ldots, x_n$ where each clause involves at most 3 literals. Decide whether $\exists x \in \{0,1\}^n$ that satisfies ϕ.

Claim. 3SAT is **NP**-complete.

Proof. □

9 NP-completeness

References

[1] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. *Algorithms.*

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.*