CS170: Efficient Algorithms and Intractable Problems

UC Berkeley

Kelvin Lee

October 10, 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 1.1 (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

$$f(x) = \mathcal{O}(g(x)).$$

§1.1.2 Ω Notation

The Ω notation describes lower bounds for functions.

Definition 1.2 (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 1.3 (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 1.4 (Asymptotic Limit Rules)
If $f(n), g(n) \geq 0$:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = \mathcal{O}(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ 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 2.1 (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 $x = 2^{n/2}x_{\mathrm{L}} + x_{\mathrm{R}}$.

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

- 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

$$T(n) = 4T(n/2) + \mathcal{O}(n).$$

However, this can be improved by using 3 multiplications:

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

$$x_Ly_R + x_Ry_L = \left(x_L + x_R\right)\left(y_L + y_R\right) - x_Ly_L - x_Ry_R.$$

- The improved running time would then be

$$T(n) = 3T(n/2) + \mathcal{O}(n).$$

- 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 2.2 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 $2^nP_1 + 2^{n/2}\left(P_3 - P_1 - P_2\right) + P_3$

§2.2 Master Theorem

A divide-and-conquer algorithm might be described by

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

Theorem 2.3 (Master Theorem)
If $T(n) = aT(n/b) + \mathcal{O}\left(n^d\right)$ for $a > 0, b > 1$, and $d \geq 0$, then

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

Proof. Let $T(n) = aT\left(n/b\right) + \mathcal{O}\left(n^d\right)$. 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,

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

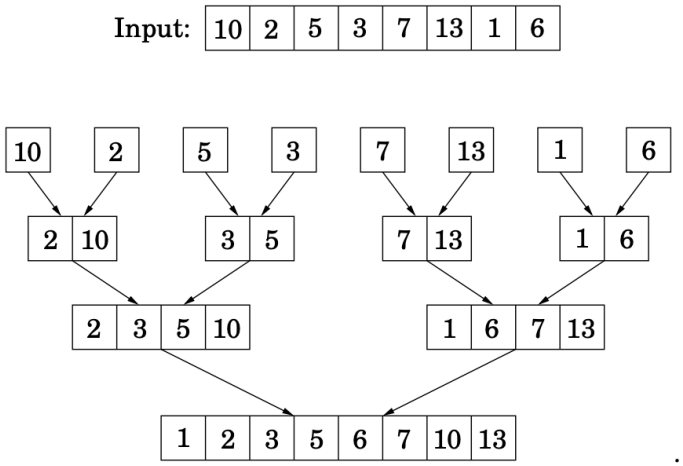
Intuitively, here the branching factor is more significant, so the total work done at each level increases, and the leaves of the tree "dominate".

□

§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 $\mathcal{O}(n)$ time, so the overall runtime is

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

§2.4 Inversions Counting

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

Definition 2.4 (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 2.5. The maximum number of inversions in a list of length n is $\binom{n}{2}$ (why?).

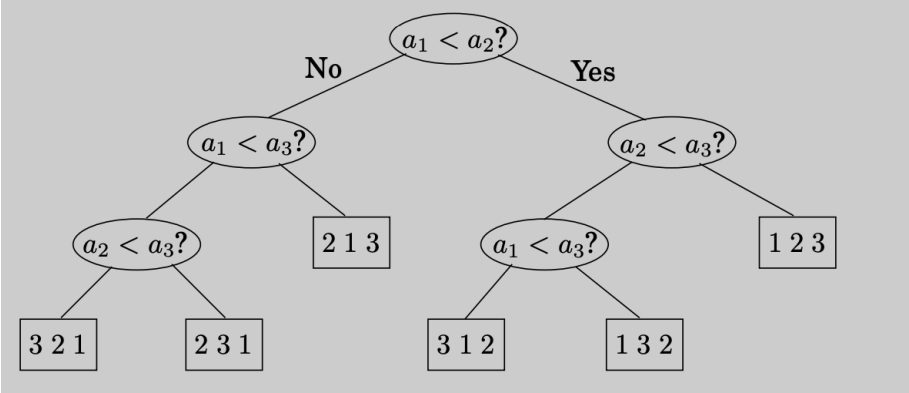
Now the question is how do we solve this by divide and conquer?

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 $\mathcal{O}(n)$ time. Hence, we still have the following runtime:

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

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

- 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 2.6. 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 $\mathcal{O}(n \log n)$. 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

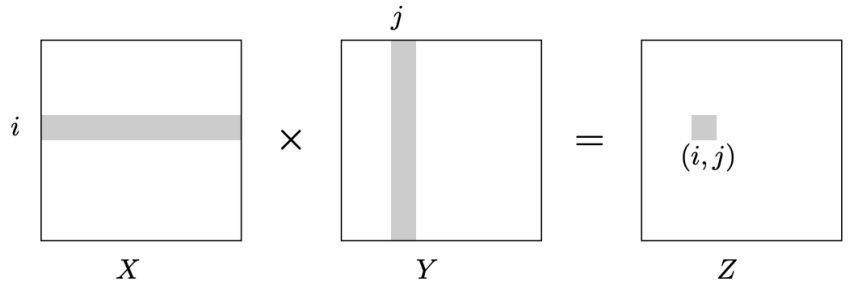
$$T(n) \leq T(3n/4) + \mathcal{O}(n) \implies T(n) = \mathcal{O}(n).$$

§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:

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

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 $\mathcal{O}(n^2)$. How can we do better?

§2.7.1 Representation of polynomials

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

Property 2.7. *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

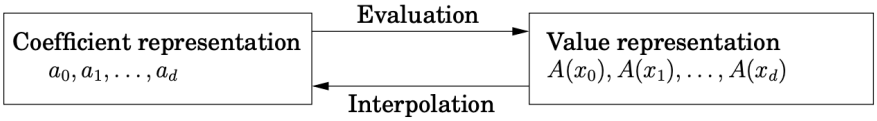
degree-d polynomial $A(x) = a_0 + a_1x + \ldots + a_dx^d$ with two representations:

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

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

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

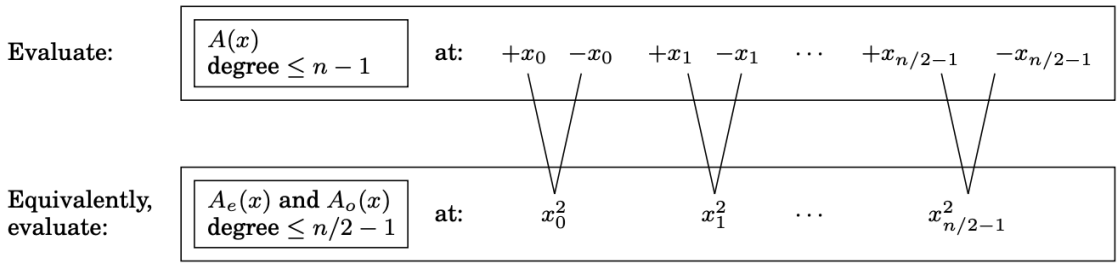
which can be written as

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

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 $A(-x_i)$:

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

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, $x_0^2, \ldots, x_{n/2-1}^2$.

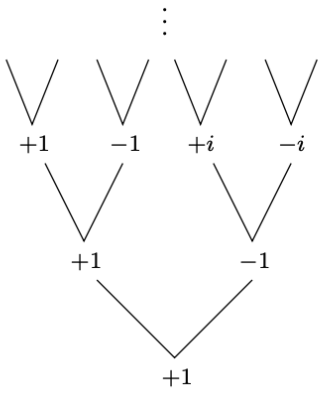


The original problem of size n is now divided into two subproblems of size $n/2$, followed by linear-time arithmetics. We get the following runtime:

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

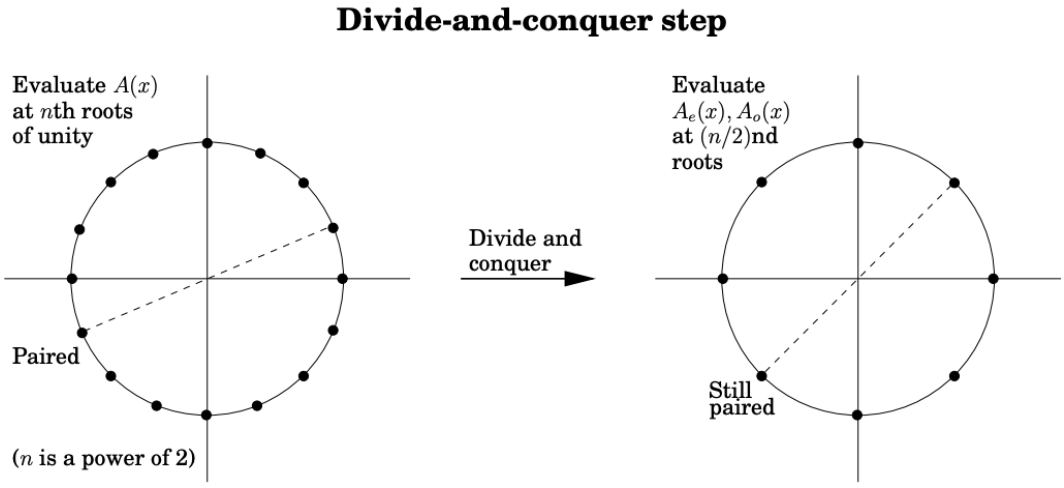
However, we now encounter a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we need the $n/2$ points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be plus-minus pairs. Hence, complex numbers come in handy now.

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) + x A_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 $\mathcal{O}(n \log n)$ steps with the nth roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

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

Property 2.9. *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***.

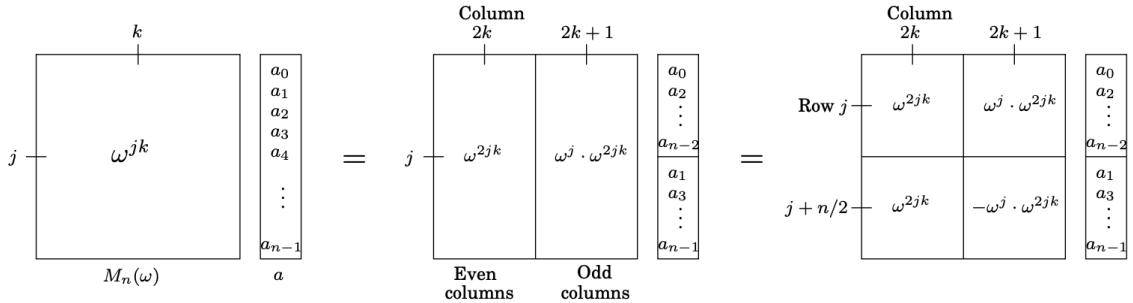
So multiplication by M represents a rotation of a vector from standard basis into the Fourier basis, which are defined by the columns of M. In other words, FFT is simply a change of basis!

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}.

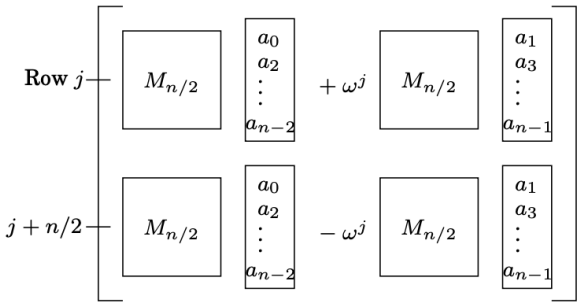
Theorem 2.10 (Inversion Formula)

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

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 $M_{n/2}(\omega^2)$ 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 3.1 (Graph)
A **graph** mathematical object which consists of vertices(nodes) V and edges E. We represent a graph G using the notation $G = (V, E)$.

Notation 3.2 (Directed vs Undirected)**.** 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.

Definition 3.3 (Directed and Undirected Graphs)
Graphs with directed edges are called **directed graphs** and graphs with undirected edges are called **undirected graphs**.

§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 $\mathcal{O}(|V|^2)$ 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 $\mathcal{O}(|V| + |E|)$ 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

Theorem 3.5
Explore$[u]$ returns only after every vertex connected to u is visited.

Algorithm 3.4 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′)
```

Proof. We know (visited[v]) $\implies$ (u is connected to v) and want to prove the converse. Suppose some connected v is not visited. There is a path from u to v. Let w be the furthest node along this path such that its successor w' is not reached. But it cannot be the case that Explore is called on w but not on w'. □

§3.3 Depth-first Search in Directed Graphs

Algorithm 3.6 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++
```

Based on the search order (incl. start points), there are 4 different types of edges.

tree edges are the closest paths to an adjacent point.

cross edges connect trees.

back edges go into a tree.

forward edges jump forward across tree edges.

§3.4 Dijkstra's Algorithm

§3.5 Depth-first Search

§4 Paths in Graphs

Definition 4.1 (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

§5 Greedy Algorithms

Definition 5.1 (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 5.2. Greedy algorithms can be disastrous for some computational tasks.

§5.1 Minimum Spanning Trees

Definition 5.3 (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 5.4 (Tree)

A **tree** is an undirected graph that is connected and acyclic.

Property 5.5. *Removing a cycle edge cannot disconnect a graph.*

Property 5.6. *A tree on n nodes has $n - 1$ edges.*

Property 5.7. *Any connected, undirected graph $G = (V, E)$ with $|E| = |V|1$ is a tree.*

Property 5.8. *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 5.9 (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

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

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 5.10. 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**. 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.

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 smalled 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

§5.2.1 Data Compression

Problem 5.11. *Suppose we are given an alphabet* $\{a_i\}_{i=1}^n$ *with frequencies* $\{f_i\}_{i=1}^n$. *How should we encode these characters?*

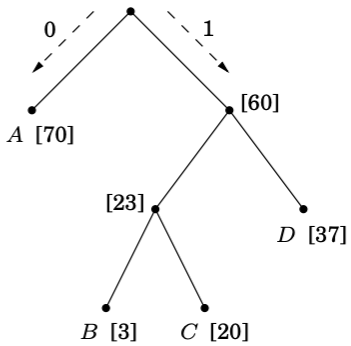
The naive way would be to represent each character using $\log N$ bits. However, can we do better than that?
The naive way of encoding is called **fixed-length code**, where each codeword has the same length. Therefore, our goal is to find a **variable-length code** where each codeword can have different lengths.

Remark 5.12. 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 can 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,

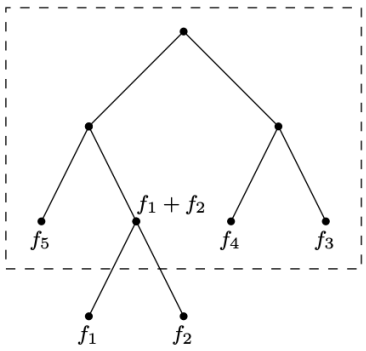
$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth of } i \text{ th symbol in tree}).$$

Remark 5.13. Number of bits required for a symbol is its depth in the tree.

There is another way to write this cost function that is very helpful. Although we are only given frequencies for the leaves, we can define the frequency of any internal node to be the sum of the frequencies of its descendant leaves; this is, after all, the number of times the internal node is visited during encoding or decoding. During the encoding process, each time we move down the tree, one bit gets output for every nonroot node through which we pass. So the total cost-the total number of bits which are output-can also be expressed thus: The cost of a tree is the sum of the frequencies of all leaves and internal nodes, except the root. The first formulation of the cost function tells us that the two symbols with the smallest frequencies must be at the bottom of the optimal tree,

as children of the lowest internal node (this internal node has two children since the tree is full). Otherwise, swapping these two symbols with whatever is lowest in the tree would improve the encoding.

This suggests that we start constructing the tree greedily: find the two symbols with the smallest frequencies, say i and j, and make them children of a new node, which then has frequency $f_i + f_j$. To keep the notation simple, let's just assume these are f_1 and f_2. By the second formulation of the cost function, any tree in which f_1 and f_2 are sibling-leaves has cost $f_1 + f_2$ plus the cost for a tree with $n-1$ leaves of frequencies $(f_1 + f_2), f_3, f_4, \ldots, f_n$:




```
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

Problem 5.14. *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*

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

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 running time for a good implementation of this algorithm is

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

Claim 5.15.

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 solutions to smaller subproblems.

§6.1 Shortest Path in DAGs, revisited

Recall: Given $G = (V, E)$ with $l : E \to \mathbb{Z}$.
Goal: Find the shortest path s to t.
Approach:

- Define a collection of subproblems: shortest path from s to v for any $v \in V$.
- Write a recurrence on the subproblems

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

- Write edge cases:
  1. $\text{dist}(s) = 0$
  2. $\text{dist}(v) = \infty$ if v is a source.
- Analyze runtime and memory.

There are n subproblems. Each subproblems takes $\mathcal{O}(\text{in\_deg}(v))$. Overall:

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

Remark 6.1. Can find longest path by changing the min to max.

§6.2 Longest Increasing Subsequences

Problem 6.2. *Given an array of n numbers, we are interested in finding the longest subsequence (non-consecutrive) that is strictly increasing.*

- Greedy is not optimal because we might end up with a shorter subsequence.
- Instead we use dynamic programming.
- We first consider the subproblems for this problem.

Subproblems:

- Try $i = 1, 2, \ldots, n$ and define $f(i)$ as the longest increasing subsequence that contains x_i.

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

  Return $\max(f(1), f(2), \ldots, f(n))$
- We have n subproblems where each can be solved from previous ones with additional time $\mathcal{O}(n)$.

- **Runtime:** $\mathcal{O}(n^2)$.
- **Memory:** $\mathcal{O}(n)$.

§6.3 Edit Distance

Problem 6.3. *Given two strings $x[1,\ldots,n]$ and $y[1,\ldots,m]$. We are interested in finding 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.*

Example 6.4
Suppose we have $s =$ "s n o w y" and $t =$ "s u n n y". Then the output should produce 3.

Subproblems:

- For $0 \le i \le n$ and $0 \le j \le m$ 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

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

- Edge cases: $f(i,0) = i$ and $f(0,j) = j$.
- **Runtime:** $(n+1)(m+1)$ subproblems.

§6.4 Knapsack

Problem 6.5. *We are robbing a bank! We find n items in the safe with weights $w_1,\ldots,w_n$ and values $v_1,\ldots,v_n$. We have a knapsack or a bag that can carry at most W pounds. We need to find the most valuable choice that fits the knapsack.*

- 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.
- Let's use dynamic programming again. Suppose $f(i,u)$ =max value when packing items $1,\ldots,i$ in a bag of capacity u.

References

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