COSC 5110 - Analysis of Algorithms Lecture Note 1

Libao Jin (ljin1@uwyo.edu)

December 6, 2018

1 Introduction

Definition 1.1 (Algorithm)**.** Algorithm is procedure for solving a problem that is precise, unambiguous, mechanical, and corret.

We are most interested in *efficient algorithms*.

1.1 What is efficient?

The two most basic computational resources are time and space:

- *time*: number of computation steps.
- *space*: amount of memory used during the computation.

Usually measured in terms of the *input size*. Formally, the size of an object is the number of bits needed to represent it as a binary string. Often we take a simpler approach and use some parameter of the input as the size.

- A graph on n vertices has size. Sometimes, we identify two parameters: a graph with n vertices and m edges.
- An array of size n.
- An $n \times n$ matrix has size nm.

Using bits

- An integer is represented in binary the number n has size $\approx \log n$.

2 Overview of Some Topics

- Preliminaries (Chapter 0)
- Divide-and-conquer
  - Sorting (# of comparisons)
    - Merge Sort: $O(n \log n)$ time.
    - Quick Sort: $O(n^2)$ worst-case time, $O(n \log n)$ average-case time.
    - Randomized Quick Sort: $O(n \log n)$ expected time.
  - Finding the median (# of comparisons)
    - $O(n)$ randomized algorithm - Quick Select
    - $O(n)$ deterministic algorithm - Median-of-Medians
  - Matrix Multiplication (# of multiplications & additions)
    - Multiply two $n \times n$ matrices
    - Standard algorithm is $O(n^3)$ time (three nested `for` loops)
    - Strassen's algorithm: $O(n^{\log_2 7}) \approx O(n^{2.81})$ time

    - $O(n^{2.37})$ time is achievable (some researchers conjecture that $O(n^2)$ or $O(n^{2+\varepsilon})$ is achievable)
  - Integer multiplication (multiply two n-bit numbers)
    - Standard (grade school) algorithm is $O(n^2)$
    - Karasuba's algorithms: $O(n^{\log_2 3}) \approx O(n^{1.6})$
    - Strassen & Soloway: $O(n \log n \log \log n)$
  - Fast Fourier Transform (many applications)
    - Convolution of vectors
    - Product of Polynomials
    - Application multiplying integers: $O(n \log n)$ time
    - Standard algorithm is $O(n^2)$
- Dynamical Programming (bottom-up vs. top-down: powerful variation of divide-and-conquer)
  - Longest common sequence (applications in computation biology)
  - Edit distance (applications in computation biology)
  - Knapsack
  - All-pairs shortest paths
  - Maximum flow problems
- Greedy Algorithms
  - Locally optimal choices lead to a globally optimal solution
  - Minimum spanning trees
    - Kruskal's algorithm (Union-find data structure)
    - Prim's algorithm
  - Huffman encoding
- Linear Programming (many applications)
  - Simplex algorithm
  - LP-duality
  - Applications to approximation algorithms
- Computational Intractability
  - Shortest paths (easy, polynomial time algorithm) vs. Longest paths (hard, NP-complete)
  - Eulerian cycle vs. Hamiltonian cycle
  - 2-SAT vs. 3-SAT
- NP-completeness
  - Traveling salesman problem
  - Knapsack
  - Clique vertex cover
  - Subset sum
- Other Hard Problems (conjectured to be hard but not NP-complete)
  - Factoring
  - Discrete logarithm
  - Graph isomorphism
- Coping with Intractability
  - Search techniques and heuristics (no provable guarantee but can work well in practice)
    - Backtracking
    - branch-and-bound
    - local search
    - simulated annealing

- Approximation algorithms with provable guarantees
  * (3/2) 2-approximation algorithm for TSP with triangle inequality (minimal spanning tree algorithm)
  * PTAS (Polynomial Time Approximation Solution) for Euclidean TSP (divide-and-conquer)
  * 2-approximation algorithm for vertex cover (greedy algorithm)
  * FPTAS for knapsack (dynamic programming)
- Cryptography
  * Using computational intractability to our advantage
  * Hardness of factoring -> RSA cryptosystem (public-key cryptography)
  * Discrete logarithm based cryptosystem
  * Quantum algorithms for factoring and discrete logarithm

2.1 Asymptotic Notation

Let $f : \mathbb{Z}^+ \to \mathbb{R}^+$ and $g : \mathbb{Z}^+ \to \mathbb{R}^+$.

1. We say $f(n) = O(g(n))$ if $(\exists c)(\exists n_0)(\forall n \geq n_0) f(n) \leq c \cdot g(n)$. Read "$f(n)$ is big-oh of $g(n)$". $f(n) = O(g(n))$ means "$f(n)$ grows no faster than $g(n)$".
2. We say $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$. Read "$f(n)$ is omega of $g(n)$". $f(n) = \Omega(g(n))$ means "$f(n)$ grows at least as fast as $g(n)$".
3. We say $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Read "$f(n)$ is theta of $g(n)$". $f(n) = \Theta(g(n))$ means "$f(n)$ and $g(n)$ have the same growth rate".
4. We say $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Read "$f(n)$ is little-oh of $g(n)$", "$f(n)$ is asymptotically smaller than $g(n)$".

Example 2.1.

1) $3n = O(n)$: $f(n) = 3n$, $g(n) = n$, $c = 3$, $n_0 = 1$.
2) $5n + 8 = O(n)$: $c = 6$, $n_0 = 8$.
3) $3n^2 + 4n + 2 = O(n^2)$.
4) $100n^2 + 1000n + 50000 = O(n^2)$.
5) $n = O(n^2)$.
6) $n^3 \neq O(n^2)$.
7) $n^2 = \Omega(n)$.
8) $\frac{1}{2}n^3 - n^2 + 6 = \Omega(n^3)$.
9) $3n^2 = \Theta(n^2)$.
10) $5n^3 + 8n^2 - n = \Theta(n^3)$.
11) $n = o(n^2)$, since $\lim_{n\to\infty} \frac{n}{n^2} = \lim_{n\to\infty} frac1n = 0$.
12) $4n^2 + 5n + 3 = o(n^3)$, since $\lim_{n\to\infty} \frac{4n^2+5n+3}{n^3} = \lim_{n\to\infty} \frac{4}{n} + \frac{5}{n^2} + \frac{3}{n^3} = 0$.
13) $2n = o(n \log n)$, since $\frac{2n}{n \log n} = \frac{2}{\log n} \to 0$ as $n \to \infty$.

Note: $f(n) = o(g(n))$ implies $f(n) = O(g(n))$. Analogies:
Asymptotic notation captures how well algorithms scale.

- $O(n)$ time algorithm: double the input size $\implies$ roughly twice as much computation time.
- $O(n^2)$ time algorithm: double the input size $\implies$ roughly four times as much computation time.
- $O(n^3)$ time algorithm: double the input size $\implies$ roughly eight times as much computation time.
- $O(2^n)$ time algorithm: double the input size $\implies$ exponential increase in computation time, i.e., $f(n) = 2^n$, $f(2n) = 2^{2n}$, $f(2n)/f(n) = 2^{2n-n} = 2^n$.

Notation	Analogy
O	$\leq$
Ω	$\geq$
Θ	$=$
o	$<$
ω	$>$

2.2 Multiplying Two Numbers

Example 2.2.

(1) $35 = 100011_2$

(2) $26 = 11010_2$

2.2.1 "Grade School" Algorithm

$100011 \times 11010 = 1110001110$, $O(n^2)$ time for two n-bit numbers, where there would be n^2 bit operations.
Note: Does not matter if we use another base. As for number N, we have

$$N \to\approx \log_2 N \text{ bits, binary}$$
$$N \to\approx \log_{10} N \text{ digits, decimal}$$

Hence we have

$$\begin{aligned}\log_2 x &= (\log_2 10) \log_{10} x,\\ \log_{10} x &= (\log_{10} 2) \log_2 x,\\ \log_2 x &= \Theta(\log_{10} x).\end{aligned}$$

2.2.2 Recursive Approach

Can we do better than $O(n^2)$?
Idea: use recursion. Recursively multiply two n-bit numbers x and y. Split each number into two numbers with $n/2$ bits.

$$\begin{aligned}x &= x_L x_R = 2^{n/2} x_L + x_R,\\ y &= y_L y_R = 2^{n/2} y_L + y_R,\end{aligned}$$

where x, y are n-bit binary numbers, and x_L, x_R, y_L, y_R are $n/2$ bit numbers.

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

Algorithm 1: Standard recursive algorithm for multiplying two n-bits numbers

```
Function multiply(x, y):
    Input: x and y are two n-bit numbers (assume n is a power of 2)
    Output: The product of x and y
    if n == 1 then
        return x · y;
    end
    x_L = leftmost n/2 bits of x;
    x_R = rightmost n/2 bits of x;
    y_L = leftmost n/2 bits of y;
    y_R = rightmost n/2 bits of y;
    p_1 = multiply(x_L, y_L);
    p_2 = multiply(x_L, y_R);
    p_3 = multiply(x_R, y_L);
    p_4 = multiply(x_R, y_R);
    p = 2^n p_1 + 2^(n/2) (p_2 + p_3) + p_4;
    return p;
end
```

Let $T(n) =$ overall runtime on inputs of size n, then

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + T(n/2) + T(n/2) + O(n) = 4T(n/2) + O(n) \\ &\vdots \\ T(1) &= O(1). \end{aligned}$$

where the runtimes for p_1, p_2, p_3, p_4 are $T(n/2)$, $O(1)$ is constant time.

Proposition 2.1.

$$T(n) = O(n^2).$$

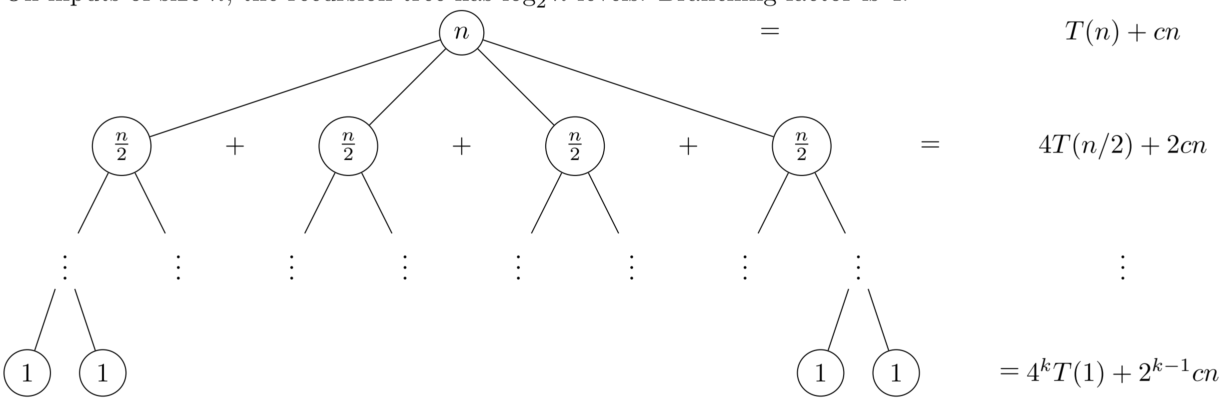
2.2.3 Backward Substitution

$$\begin{aligned} T(n) &= 4T(n/2) + cn \\ &= 4[4T(n/4) + cn/2] + cn \\ &= 16T(n/4) + 2cn + cn \\ &= 16[4T(n/8) + cn/4] + 2cn + cn \\ &= 64T(n/8) + 4cn + 2cn + cn \\ &= 256T(n/8) + 8cn + 4cn + 2cn + cn \\ &\vdots \\ &= 4^k T(n/2^k) + cn \sum_{i=0}^{k-1} 2^i \\ &= 4^k T(n/2^k) + cn(2^k - 1). \end{aligned}$$

Choose k such that $n/2^k = 1$, $2^k = n$, $k = \log_2 n$. Suppose $n = 2^k$, $k = \log_2 n$ then

$$\begin{aligned} T(n) &= 4^k T(n/2^k) + (2^k - 1)cn \\ &= 4^{\log_2 n} T(1) + (n-1)cn \\ &= n^2 O(1) + O(n^2) \\ &= O(n^2). \end{aligned}$$

On inputs of size n, the recursion tree has $\log_2 n$ levels. Branching factor is 4.



2.2.4 Better Approach: Karatsuba's Algorithm

Recall that $x \cdot y = 2^n(x_L y_L) + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$.
Idea: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$. Compute

$$\begin{aligned} p_1 &= x_L y_L, \\ p_2 &= x_R y_R, \\ p_3 &= (x_L + x_R)(y_L + y_R). \end{aligned}$$

Then $(x_L y_R + x_R y_L) = p_3 - p_1 - p_2$, therefore, $x \cdot y = 2^n p_1 + 2^{n/2}(p_3 - p_1 - p_2) + p_2$.

Algorithm 2: Karatsuba's algorithm for multiplying two n-bits numbers

Function multiply(x, y)**:**
- **Input:** x and y are two n-bit numbers (assume n is a power of 2)
- **Output:** The product of x and y
- **if** $n == 1$ **then**
  - **return** $x \cdot y$;
- **end**
- x_L = leftmost $n/2$ bits of x;
- x_R = rightmost $n/2$ bits of x;
- y_L = leftmost $n/2$ bits of y;
- y_R = rightmost $n/2$ bits of y;
- p_1 = multiply(x_L, y_L);
- p_2 = multiply(x_R, y_R);
- p_3 = multiply$(x_L + x_R, y_L + y_R)$;
- $p = 2^n p_1 + 2^{n/2}(p_3 - p_1 - p_2) + p_2$;
- **return** p;

end

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

$$\begin{aligned} T(n) &= 3T(n/2) + cn \\ &= 3[3T(n/4) + cn/2] + cn \\ &= 9T(n/4) + 3/2cn + cn \\ &= 9[3T(n/8) + cn/4] + 3/2cn + cn \\ &= 27T(n/8) + 9/4cn + 3/2cn + cn \\ &= 81T(n/16) + 27/8cn + 9/4cn + 3/2cn + cn \\ &\vdots \end{aligned}$$

$$
\begin{aligned}
&= 3^k T(n/2^k) + cn\sum_{i=0}^{k-1}(3/2)^i \\
&= 3^k T(n/2^k) + cn[2(3/2)^k - 2]. \\
&= 3^{\log_2 n} T(1) + O((3/2)^{\log_2 n} \cdot n) \\
&= n^{\log_2 3} O(1) + O(n^{\log_2 3}) \\
&= O(n^{\log_2 3}).
\end{aligned}
$$

2.2.5 Summary

- Standard algorithm (first recursive algorithm): $O(n^2)$.
- Karatsuba's algorithm (1961): $O(n^{\log_2 3}) \approx O(n^{1.58})$.
- Schöhage-Strassen using Fast Fourier Transform (1971): $O(n \cdot \log n \cdot \log\log n)$.
- Fürer (2007): $O(n \cdot \log n \cdot 2^{\log^* n})$, where $\log^*$ is the number of recursively taking logarithm to get to 1.
- Open problem: Is there on $O(n \log n)$ time algorithm?

2.2.6 Master Theorem for Recurrence Relations

Suppose that

$$
T(n) = \begin{cases} aT(n/b) + O(n^d), & n > 1 \\ O(1), & n = 1, \end{cases}
$$

where $a > 0$ (recursive calls), $b > 1$ (input size reduction factor), and $d > 0$ ($O(n^d)$ is local work) are constants. Then

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

Recursion tree has branching factor $a \implies$ the kth level of the recursion tree has a^k subproblems.

Subproblems are a factor b smaller at each level

1. $\implies$ subproblems at level k have size n/b^k.
2. $\implies$ depth of recursion tree is $\log_b n$ ($n/b^k = 1 \implies b^k = n \implies k = \log_b n$)

Total work:

$$
\sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k cn^d = cn^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k
$$

- Geometric series:

$$
S = \sum_{k=0}^{l} \alpha^k = \begin{cases} \frac{\alpha^{l+1}-1}{\alpha-1}, & \text{if } \alpha \neq 1 \\ l+1, & \text{if } \alpha = 1 \end{cases}
$$

```
* If $\alpha > 1$, this is $\Theta(\alpha^l)$.
* If $\alpha < 1$, this is $\Theta(1)$.
* If $\alpha = 1$, this is $\Theta(l)$.
```

Let $\alpha = \frac{a}{b^d}$. So

$$
S = \begin{cases} \Theta((\frac{a}{b^d})^{\log_b n}), & \text{if } a > b^d, \\ \Theta(1), & \text{if} a < b^d, \\ \Theta(\log_b n) & \text{if} a = b^d. \end{cases}
$$

Then it becomes

$$cn^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k = \begin{cases} O(n^d \left(\frac{a}{b^d}\right)^{\log_b n}), & \text{if } a > b^d, \\ \Theta(n^d), & \text{if} a < b^d, \\ \Theta(n^d \log_b n) & \text{if} a = b^d. \end{cases} = \begin{cases} O(n^d \left(n^{\log_b a}\right), & \text{if } d < \log_b a, (\text{work done at bottom dominates} : O(1) \cdot \\ \Theta(n^d), & \text{if} d > \log_b a, (\text{work done at top dominates} : O(n^d)) \\ \Theta(n^d \log_b n) & \text{if} d = \log_b a. (\text{about the same amount of work done at} \end{cases}$$

Example 2.3. 1. MergeSort: $T(n) = 2T(n/2) + O(n)$: $T(n) = O(n \log n)$ since $a = 2, b = 2, d = 1$.

2. $T(n) = 4T(n/2) + O(n)$: $T(n) = O(n^2)$ since $a = 4, b = 2, d = 1$.

3. Karatsuba's algorithm: $T(n) = 3T(n/2) + O(n)$: $T(n) = O(n^{\log_2 3})$ since $a = 3, b = 2, d = 1$.

4. $T(n) = 2T(n/4) + O(n)$: $T(n) = O(n)$ since $a = 2, b = 4, d = 1$.

2.2.7 Matrix Multiplication

Given two $n \times n$ matrices X and Y, compute the product $Z = X \cdot Y$.

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix}, Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix} Z = \begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix}.$$

- Standard Matrix Multiplication ($\Theta(n^3)$)

$$z_{ij} = \sum_{k=1}^{n} x_{ik} \cdot y_{kj}.$$

```
for i = 1 to n
    for j = 1 to n
        $z_{ij}$ = 0
        for k = 1 to n
            $z_{ij} = z_{ij} + x_{ik} \cdot y_{kj}, O(1)
```

2.2.8 Divide-and-Conquer Approach

Divide each matrix into 4 matrices $n/2$-by-$n/2$.

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

Reduce multiplying a pair of $n \times n$ matrices to multiplying 8 pairs of $n/2 \times n/2$ matrices, which leads to the recursive algorithm: $T(n) = 8T(n/2) + O(n^2) \implies T(n) = O(n^3)$, since $a = 8, b = 2, d = 2$. (No improvement over standard algorithm)

```
RecursiveMultiply(X, Y) // $n \times n$ matrices
if ($u = 1$) return $X \cdot Y$
Form A, B, C, D, E, F, G, H (O(n^2))
U_1 = RecusiveMultiply(A, E) + RecusiveMultiply(B, G) (T(n/2) + T(n/2) + O(n^2 / 4))
U_2 = RecusiveMultiply(A, F) + RecusiveMultiply(B, H) (T(n/2) + T(n/2) + O(n^2 / 4))
L_1 = RecusiveMultiply(C, E) + RecusiveMultiply(D, G) (T(n/2) + T(n/2) + O(n^2 / 4))
L_2 = RecusiveMultiply(C, F) + RecusiveMultiply(D, H) (T(n/2) + T(n/2) + O(n^2 / 4))
P = [U_1 & U_2 \\ L_1 & L_2] (O(n^2))
return P
```

$T(n) = 8T(n/2) + O(n^2) = O(n^3), a = 8, b = 2, c = 2.$

2.2.9 Strassen's Algorithm $T(n) = 7T(n/2) + O(n^2), a = 7, b = 2, c = 2.$.

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

as before.
Compute

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

Then

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

```
Strassen(X, Y) // $n \times n$ matrices
if ($u = 1$) return $X \cdot Y$
Form A, B, C, D, E, F, G, H (O(n^2))
P_1 = Strassen(A, F - H) (T(n/2) + T(n/2) + O(n^2 / 4))
P_2 = Strassen(A + B, F) (T(n/2) + T(n/2) + O(n^2 / 4))
P_3 = Strassen(C + D, E) (T(n/2) + T(n/2) + O(n^2 / 4))
P_4 = Strassen(D, G - E) (T(n/2) + T(n/2) + O(n^2 / 4))
P_5 = Strassen(A + D, E + H) (T(n/2) + T(n/2) + O(n^2 / 4))
P_6 = Strassen(B - D, G + H) (T(n/2) + T(n/2) + O(n^2 / 4))
P_7 = Strassen(A - C, E + F) (T(n/2) + T(n/2) + O(n^2 / 4))
P = [U_1 & U_2 \\ L_1 & L_2] (O(n^2))
return P
```

Algorithm 3: Karatsuba's algorithm for multiplying two n-bits numbers

Function *Strassen*(x, y)**:**
  Input: X and Y are two $n \times n$ matrices (assume n is a power of 2)
  Output: The product of X and Y
  if $n == 1$ **then**
    return $X \cdot Y$;
  end
  $P_1 = \text{Strassen}(A, F - H)(T(n/2) + T(n/2) + O(n^2/4))$;
  $P_2 = \text{Strassen}(A + B, F)(T(n/2) + T(n/2) + O(n^2/4))$;
  $P_3 = \text{Strassen}(C + D, E)(T(n/2) + T(n/2) + O(n^2/4))$;
  $P_4 = \text{Strassen}(D, G - E)(T(n/2) + T(n/2) + O(n^2/4))$;
  $P_5 = \text{Strassen}(A + D, E + H)(T(n/2) + T(n/2) + O(n^2/4))$;
  $P_6 = \text{Strassen}(B - D, G + H)(T(n/2) + T(n/2) + O(n^2/4))$;
  $P_7 = \text{Strassen}(A - C, E + F)(T(n/2) + T(n/2) + O(n^2/4))$;
  $P = \begin{bmatrix} U_1 & U_2 \\ L_1 & L_2 \end{bmatrix} (O(n^2))$;
  return P;
end

- Summary
  - Strassen (1961): $O(n^{2.81\cdots})$.
  - Cooper Smith and Winograd (1990): $O(n^{2.375477\cdots})$.
  - Current best (2014): $O(^{2.3728})$.
  - $O(n^{2+\varepsilon})$ for $\varepsilon > 0$ is conjectured by some researchers. Obvious $\Omega(n^2)$ is the lower bound. ω is the infimum of all w such that there is an $O(n^w)$ algorithm.
  - Conjecture: $\omega = 2$, known $2 \le \omega < 2.3728\ldots$.
- Why might we expect that $\omega = 2$? While it is unknown how to multiply matrices in $O(n^2)$ time, it is possible to check that the answer in $O(n^2)$ randomized time.

2.3 Verifying Matrix Multiplication

- Given: $n \times n$ matrices X, Y and Z.
- Question: $XY = Z$?
- Simpliest approach: multiply $X \cdot Y$ and check if it equals Z.
- $O(n^\omega) + O(n^2) = O(n^\omega)$.

2.3.1 Better Approach for Verifying Matrix Multiplication

Choose a vector $\vec{r} \in \{0, 1\}^n$ uniformly at random (n independent fair coin flips). * If $X \cdot Y = Z$, then for every $\vec{r}$,

$$XY\vec{r} = Z\vec{r}.$$

Theorem 2.1. If $X \cdot Y \neq Z$, then

$$\Pr_{\vec{r}\in\{0,1\}^n}[XY\vec{r} = Z\vec{r}] \le \frac{1}{2} \implies \Pr_{\vec{r}\in\{0,1\}^n}[XY\vec{r} \neq Z\vec{r}] \ge \frac{1}{2}.$$

How does this help? $Z\vec{r}$ is $O(n^2)$ time, $(XY)\vec{r} = X(Y\vec{r})$

2.3.2 Randomized Algorithm for Verifying Matrix Multiplication

Choose $\vec{r} \in \{0, 1\}^n$ uniformly at random. Compute

$$\vec{b} = Y \cdot \vec{r} \leftarrow O(n^2)\vec{a} = X \cdot \vec{b} \leftarrow O(n^2)\vec{c} = Z \cdot \vec{r} \leftarrow O(n^2)$$

If $\vec{a} == \vec{c}$, return true; If $\vec{a} \neq \vec{c}$, return false.

- Correctness of the algorithm.
  - If $XY = Z$, then Pr[algorithm outputs true] $= 1$.
  - If $XY \neq Z$, then Pr[algorithm outputs true] $\le 1/2$.
  - Equivalently, If $XY \neq Z$, then Pr[algorithm outputs false] $\ge 1/2$.
  - No false negatives. Whenever the algorithm outputs false, that is the correct answer.
  - There are false positives. If the algorithm outputs true, this is possibly the wrong answer (should be false). $XY \neq Z$, but $XY\vec{r} = Z\vec{r}$ for the chosen vector $\vec{r}$. (Happens at most half of the time.)

2.3.3 Improving the success probability

- Run the algorithm k times, where $k \ge 2$, each run is independent, different random vector each time.
- If true is output every time, then output true.
- If false is ever output, then output false.
  - If $XY = Z$, output true every time, so algorithm outputs true...
  - If $XY \neq Z$, Pr[algorithm outputs true] $\le 1/2^k$, so Pr[algorithm output true] $\le 2^{-k}$, Pr[algorithm output false] $\ge 1 - 2^{-k}$

- Take $k = 2$, then Pr[incorrect answer] $\leq \frac{1}{4}$.
- Take $k = 10$, then Pr[incorrect answer] $\leq \frac{1}{2^{10}}$.
- Take $k = 100$, then Pr[incorrect answer] $\leq \frac{1}{2^{100}}$.
- One-sided error algorithm, CORP algorithm (computation complexity theory).
- $P \subset RP \subset NP$ and $P \subset coRP \subset coNP$.

2.3.4 Proof

X, Y, and Z are $n \times n$ matrices. Choose $\vec{r} \in \{0,1\}^n$ uniformly at random. If $XY = Z$, then $XY\vec{r} = Z\vec{r}$ for all $\vec{r} \in \{0,1\}^n$.

Theorem 2.2. If $XY \neq Z$,

$$\Pr_{\vec{r}\in\{0,1\}^n}[XY\vec{r} = Z\vec{r}] \leq \frac{1}{2}.$$

Equivalently, $\Pr[XY\vec{r} \neq Z\vec{r}] \geq \frac{1}{2}$.

Proof. Assume $XY \neq Z$. Let $D = XY - Z$. Then $D \neq 0$ (the all zero's matrix). So D has at least one nonzero entry - WLOG, suppose it is d_{1n}. Suppose $XY\vec{r} = Z\vec{r}$ for a vector $\vec{r} \in \{0,1\}^n$. Then $D\vec{r} = (XY - Z)\vec{r} = XY\vec{r} - Z\vec{r} = \vec{0}$. In particular, the first component of the vector $D\vec{r}$ is 0:

$$\sum_{j=1}^{n} d_{1j} r_j = 0.$$

Equivalently,

$$r_n = \frac{-\sum_{j=1}^{n-1} d_{1j} r_j}{d_{1n}}. \tag{1}$$

Thought experiment: assume $r_1, \cdots, r_{n-1} \in \{0,1\}$ have been chosen at random. Choose $r_n \in \{0,1\}$. What is the probability that (1) is true. There are two choices for r_n: 0 and 1. At most one is correct. That implies probability is at most 1/2.

- If RHS $= 0$, then $\Pr[r_n = \text{RHS}] = \frac{1}{2}$.
- If RHS $= 1$, then $\Pr[r_n = \text{RHS}] = \frac{1}{2}$.
- If RHS $\notin \{0,1\}$, then $\Pr[r_n = \text{RHS}] = 0$.

In any case, $\Pr[r_n = \text{RHS}] \leq \frac{1}{2}$. Then,

$$\Pr[XY\vec{r} = Z\vec{r}] \leq \Pr[r_n = \frac{-\sum_{j=1}^{n-1} d_{1j} r_j}{d_{1n}}] \leq \frac{1}{2}.$$

□

2.4 Evaluating Polynomials

Given a degree n polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i$, where a_i are the coefficients of x^i. Given x, evaluate $p(x)$

```
total = 0
for i = 0 to n
    val = a_i
    for j = 1 to i
        val = val x
    total = total + val
return total
```

Count # of multiplications:

$$\sum_{i=0}^{n}\sum_{j=1}^{i} 1 = \sum_{i=0}^{n} i = \sum_{i=1}^{n} i = \frac{n(n-1)}{2}.$$

Therefore, the time complexity $\Theta(n^2)$.

2.4.1 Optimized algorithm: $\Theta(n)$

```
total = a_0
xpow = 1 // = x^0
for i = 1 to n
    xpow = xpow * x // xpow
    total = total + xpow * a_i
return total
```

Count # of multiplications:

$$\sum_{i=1}^{n} 2 = 2n = \Theta(n).$$

2.5 Evaluating:

$A(x) = 3 + 4x + 6x^2 + 2x^3 + 4x^4 + 10x^5 + 8x^6 + 9x^7 = x(4 + 2x^2 + 10x^4 + 9x^6) + (3 + 6x^2 + 4x^4 + 8x^6) = x \cdot A_0(x^2) + A_e($

where $A_0(x) = 4 + 2x + 10x^2 + 9x^3$, $A_e(x) = 3 + 6x + 4x^2 + 8x^3$. Recursively evaluate A_0, A_e.

$$A_0(x) = x(2 + 9x^2) + (4 + 10x^2) = x \cdot A_{00}(x^2) + A_{0e}(x^2),$$

where $A_{00}(x) = 2 + 9x$, $A_{0e}(x) = 4 + 10x$.

2.6 Multiplying Polynomials

$p(x) = \sum_{i=0}^{n} a_i x^i$, $q(x) = \sum_{j=0}^{n} b_i x^i$. $r(x) = p(x) \cdot q(x)$ is a degree $2n$ polynomial.

$$\begin{aligned} r(x) &= \left(\sum_{i=0}^{n} a_i x^i\right)\left(\sum_{j=0}^{n} b_j x^j\right) \\ &= \sum_{k=0}^{2n} c_k x^k, \end{aligned}$$

where $c_k = \sum_{i=0}^{k} a_i b_{k-i}$. Takes $\Theta(k)$ time to compute c_k using this formula. Total time to compute all coefficients of $r(x)$ is $\Theta(n^2)$.

- Is there a faster way?

2.6.1 Fast Fourier Transform $O(n \log n)$ time.

Polynomial → Convolution → multiplication of convolutions → product polynomial.
HAHA: Fourier Transform = make it "four"ier by changing things to 4's.

- Basic idea: A degree d polynomial is determined by its values at any $d + 1$ distinct points. Can interpolate to recover the polynomial. Have p of degree n, q of degree n, want $r = p \cdot q$ of degree $2n$. We could evaluate p and q at $2n+1$ points $x_1, x_2, \cdots, x_{2n+1}$. Then we can compute r at $2n+1$ points: $r(x_i) = p(x_i) \cdot q(x_i) \to r(x_1), r(x_2), r(x_3), \ldots, r(x_{2n+1}) \to$ interpret to recover r.

2.6.2 Steps:

- Evaluate p, q at $2n+1$ points. ($\Theta(n^2)$)
- Compute r at these $2n+1$ points by multiplying values of p and q. ($\Theta(n)$)
- Interpolate to recover r. ($\Theta(n^2)$)

$\Theta(n^2)$ time - no improvement over standard approach. FFT evaluates the polynomials at $2n+1$ specially chosen points in $O(n \log n)$ time. (Complex numbers: roots of unity).

2.6.3 Roots

Complex numbers $a+bi$, where $i = \sqrt{-1}$ or $i^2 = -1$. There are two square roots of unity: $1, -1$. Can be written as $e^{i\theta} = cos\theta + i \sin\theta$ for $\theta = 0, \pi$. Hence the fourth roots of unity is $1, -1, i, -i$. In general, the nth roots of unity are given by

$$e^{\frac{2\pi i}{n}k}, \text{ for } k = 0, 1, 2, \ldots, n-1.$$

n evenly spaced points around the unit circle in the complex plane. $e^{\frac{2\pi i}{n}}$ is the principal nth root of unity. Denote $\omega = e^{\frac{2\pi i}{n}} = \cos\frac{2\pi}{n} + i \sin\frac{2\pi}{n}$.

2.6.4 Euler's Formula

$e^{\pi i} = -1$, and $e^{2\pi i} = 1$. For any θ,

$$e^{i\theta} = \cos\theta + i\sin\theta.$$

Why is this true? We take a look at the Taylor Series of e^x:

$$\exp(x) = e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots.$$

Also the Taylor Series for $\sin(x)$ and $\cos(x)$ are as follows:

$$\sin(x) = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k+1}}{(2k+1)!} = \sum_{k=0}^{\infty} \frac{x^{4k+1}}{(4k+1)!} - \frac{x^{4k+3}}{(4k+3)!}$$
$$\cos(x) = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k}}{(2k)!} = \sum_{k=0}^{\infty} \frac{x^{4k}}{(4k)!} - \frac{x^{4k+2}}{(4k+2)!}$$

Therefore, plugging in $x = i\theta$ to the Taylor Series of $\exp(x)$ gives

$$\begin{aligned}
\exp(i\theta) &= \sum_{k=0}^{\infty} \frac{(i\theta)^k}{k!} \\
&= \sum_{k=0}^{\infty} \frac{(i\theta)^{4k}}{(4k)!} + \frac{(i\theta)^{4k+1}}{(4k+1)!} + \frac{(i\theta)^{4k+2}}{(4k+2)!} + \frac{(i\theta)^{4k+3}}{(4k+3)!} \\
&= \sum_{k=0}^{\infty} \frac{\theta^{4k}}{(4k)!} + \frac{i\theta^{4k+1}}{(4k+1)!} + \frac{-\theta^{4k+2}}{(4k+2)!} + \frac{-i\theta^{4k+3}}{(4k+3)!} \\
&= \sum_{k=0}^{\infty} \frac{\theta^{4k}}{(4k)!} - \frac{\theta^{4k+2}}{(4k+2)!} + \frac{i\theta^{4k+1}}{(4k+1)!} - \frac{i\theta^{4k+3}}{(4k+3)!} \\
&= \left(\sum_{k=0}^{\infty} \frac{\theta^{4k}}{(4k)!} - \frac{\theta^{4k+2}}{(4k+2)!}\right) + i\left(\sum_{k=0}^{\infty} \frac{\theta^{4k+1}}{(4k+1)!} - \frac{\theta^{4k+3}}{(4k+3)!}\right) \\
&= \cos\theta + i\sin\theta.
\end{aligned}$$

2.6.5 FFT:

- Given polynomial A of degree $\leq n-1$ (assume n is a power of 2), ω is a principal nth root of unity.
- Output: $A(\omega^0), A(\omega^1), A(\omega^2), A(\omega^3), \cdots, A(\omega^{n-1})$ (values of A at nth roots of unity)
- Time: $O(n \log n)$ time.

Algorithm 4: FFT

```
Function FFT(A, ω, n):
    Input: A is a polynomial of degree ≤ n − 1, n is a power of 2, ω is a principal nth root of unity.
    Output: A(ω^0), ··· , A(ω^{n−1})
    if ω == 1 then                                                        /* base case */
        return A(1);
    end
    express A(x) as A_e(x^2) + xA_0(x^2)          /* A_e and A_0 have degree < n/2.          */
    FFT(A_e, ω^2, n/2)        /* result: A_e(ω^0), A_e(ω^2), A_e(ω^4), ··· , A_e(ω^{n−2})   */
    FFT(A_0, ω^2, n/2)        /* result: A_0(ω^0), A_0(ω^2), A_0(ω^4), ··· , A_0(ω^{n−2})   */
    for j ← 0 to n − 1 do
        A(ω^j) = A_e(ω^{2j}) + ω^j A_0(ω^{2j}) ;
    end
    return A(ω^0), A(ω^1), A(ω^2), ··· A(ω^{n−1});
end
```

Example 2.4. Evaluate $A(x) = 3x^3 + x^2 + 2x + 4$. Given that $n = 4, \omega = i$, evaluate A at $\omega^0 = 1, \omega^1 = i, \omega^2 = -1, \omega^3 = -i$. Let $A_e(x) = x + 4$, $A_0(x) = 3x + 2$, then $A(x) = A_e(x^2) + xA_0(x^2)$. Then evaluate A_e, A_0 at ω^0, ω^2. We have $A_e(\omega^0) = 5$, $A_e(\omega^2) = 3$, $A_0(\omega^0) = 5$, $A_0(\omega^2) = -1$. Therefore,

$$
\begin{aligned}
A(\omega^0) &= A_e(\omega^0) + \omega^0 A_0(\omega^0) = 10. \\
A(\omega^1) &= A_e(\omega^2) + \omega^1 A_0(\omega^2) = 3 - i. \\
A(\omega^2) &= A_e(\omega^4) + \omega^2 A_0(\omega^4) = 0. \\
A(\omega^3) &= A_e(\omega^6) + \omega^3 A_0(\omega^6) = 3 + i.
\end{aligned}
$$

2.6.6 Recursion tree

- $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0$, evaluate at $\omega^0, \omega^1, \omega^2, \cdots, \omega^7$, 8th roots of unity.
  - $a_6x^3 + a_4x^2 + a_2x^1 + a_0$, evaluate at $\omega^0, \omega^2, \omega^4, \omega^6$, 4th roots of unity
    - $a_4x + a_0$, evaluate at ω^0, ω^4, 2nd roots of unity
      - a_4, evaluate at ω^0
      - a_0, evaluate at ω^0
    - $a_6x + a_2$, evaluate at ω^0, ω^4, 2nd roots of unity
      - a_6, evaluate at ω^0
      - a_2, evaluate at ω^0
  - $a_7x^3 + a_5x^2 + a_3x^1 + a_1$, evaluate at $\omega^0, \omega^2, \omega^4, \omega^6$
    - $a_5x + a_1$, evaluate at ω^0, ω^4, 2nd roots of unity
      - a_5, evaluate at ω^0
      - a_1, evaluate at ω^0
    - $a_7x + a_3$, evaluate at ω^0, ω^4, 2nd roots of unity
      - a_7, evaluate at ω^0
      - a_3, evaluate at ω^0

Let $A(x) = a_{n-1}x^{n-1} + \cdots + a_1 x_1 + a_0$ be a polynomial of degree $n-1$, evaluate at $x_0, x_1, \ldots, x_{n-1}$.

$$\vec{A} = \begin{bmatrix} A(x_0) \\ A(x_1) \\ A(x_2) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = M * \vec{a},$$

where M is the Vandermonde matrix - invertible assuming $x_0, \ldots, x_{n-1}$ are all distinct, i.e., M^{-1} exists. Therefore,

$$\vec{a} = M^{-1}\vec{A} \implies M^{-1}\vec{A} = M^{-1}M\vec{a} = I\vec{a}.$$

Define

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

FFF "multiplies" $M_n(\omega)$ and the coefficient vector.

$$\begin{bmatrix} A(1) \\ A(\omega) \\ A(\omega^2) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix} = M_n(\omega) \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$M_n^{-1}(\omega)$ exists, that implies, can recover coefficients from values by multiplying by $M_n^{-1}(\omega)$ (can be done by FFT).

Proposition 2.2.

$$M_n^{-1}(\omega) = \frac{1}{n} M_n(\omega^{-1}), i.e., M_n^{-1}(\omega) = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Proof. Let

$$M_n(\omega) \cdot M_n(\omega^{-1}) = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix}$$

Now look at x_{ij}, we have

$$x_{ij} = \begin{bmatrix} 1 & \omega^i & \omega^{2i} & \cdots & \omega^{(n-1)i} \end{bmatrix} \begin{bmatrix} 1 \\ \omega^j \\ \omega^{2j} \\ \vdots \\ \omega^{(n-1)j} \end{bmatrix} = \sum_{k=0}^{n-1} \omega^{ki}\omega^{-kj} = \sum_{k=0}^{n-1} \omega^{k(i-j)} = \begin{cases} n & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

□

As for $\omega = e^{\frac{2\pi i}{n}}$, we have

$$\omega^{-1}\omega = 1 \implies \omega^{-1} = e^{-\frac{2\pi i}{n}}, e^{\frac{2\pi i}{n}(n-1)}$$

So

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \frac{1}{n} M_n(\omega^{-1}) \begin{bmatrix} A(1) \\ A(\omega) \\ A(\omega^2) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix}.$$

2.6.7 Multiplication Algorithm

Algorithm 5: Calculate the product of two polynomials

Function PolynomialMultiplication(A, B)**:**
- **Input:** $A(x) = a_0 + a_1x + a_2x + \cdots a_{m-1}x^{m-1}$ and $B(x) = b_0 + b_1x + b_2x + \cdots b_{l-1}x^{l-1}$ are two polynomials of degree $m-1$ and $l-1$, respectively.
- **Output:** The product C of $A \cdot B$.
- Choose $n > m + l$ so that n is a power of 2, where $n \leq 2 \cdot \max(m, l)$;
- $\omega \leftarrow e^{\frac{2\pi i}{n}}$, where $e^{\frac{2\pi i}{n}}$ is the principle nth root of unity;
- Call FFT(A, ω, n) and FFT(B, ω, n) to obtain values $A(\omega^0), A(\omega^1), \ldots, A(\omega^{n-1})$ and $B(\omega^0), B(\omega^1), \ldots, B(\omega^{n-1})$;
- Compute $C(\omega^i) = A(\omega^i) \cdot B(\omega^i)$ for $i = 0, 1, \ldots, n-1$;
- Call FFT(D, ω^{-1}, n), where $d_i = C(\omega^i)$;
- $c_i \leftarrow \frac{1}{n} D(\omega^{-1})$ for $i = 0, 1, \ldots, n-1$;
- **return** $c_0, c_1, \ldots, c_{n-1}$;

end

Running time: 3 FFT calls ($O(n \log n)$) and $O(n)$ additional work, in total, $O(n \log n)$ time. Recall: standard algorithm is $O(n^2)$.
FFT $\rightarrow$ Sch ohage - Strassen fast integer multiplication $O(n \log n \log \log n)$ time. $a = \overline{a_{n-1}a_{n-2}\cdots a_0} = \sum_{i=0}^{n-1} a_i 2^i = A(2)$

2.6.8 Quicksort

Algorithm 6: Quick sort

Function Quicksort($A[1 \ldots n]$)**:**
- **Input:** A is a array, of which all elements are distinct.
- **Output:** Sorted A
- **if** $n \leq 1$ **then**
  - **return** A;
- **end**
- Choose an element p of A as a pivot;
- Compare every other element of A to p and divide them into two subarrays: A_1 has the elements of A that are less than p;
- A_2 has the elements of A that are greater than p;
- Use Quicksort to sort A_1 and A_2;
- **return** the array A_1, p, A_2;

end

Suppse p has rank k in A (kth smallest element). Then $|A_1| = k - 1$, and $|A_2| = n - k$. Number of comparisons:

$$(n-1) + \text{ \# done by } \text{Quicksort}(A_1) + \text{ \# done by } \text{Quicksort}(A_2).$$

Then

$$C(n) = n - 1 + C(k-1) + C(n-k),$$

where $C(n) = $ # of comparisons on an array of size n.

- Worst case: $k = 1$ every time.

$$\begin{aligned} C(n) &= n - 1 + C(0) + C(n-1) \\ &= n - 1 + C(n-1) \\ &= (n-1) + (n-2) + C(0) + C(n-2) \\ &\ \vdots \\ &= (n-1) + (n-2) + \cdots + 1 \\ &= \sum_{i=1}^{n-1} i \\ &= \binom{n}{2} \\ &= \frac{n(n-1)}{2} \\ &= \Theta(n^2). \end{aligned}$$

Note: $k = $ largest every time is also worst case – $\binom{n}{2}$ comparisons.

- Best case: $k = n/2$

$$\begin{aligned} C(n) &= (n-1) + C(\frac{n}{2}) + C(\frac{n}{2}) \\ &= 2C(\frac{n}{2}) + (n-1) \\ &= \Theta(n \log n) \\ &\approx 2n \log n. \end{aligned}$$

This is optimal: lower bound – any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

- Average case analysis. Suppose

$$\frac{n}{4} \le k \le \frac{3n}{4},$$

that is, pivot in middle half. And

$$C(n) \le n - 1 + C(\frac{n}{4}) + C(\frac{3n}{4}) = O(n \log n).$$

Intuitively: get pivot in the middle half about half of the time, so performance should be about the same. Let $(y_1, y_2, \ldots, y_N)$ be the sorted order of A, where y_i is the element of rank i. Define a random variable

$$X_{ij} = \begin{cases} 1 & \text{if } y_i \text{ and } y_j \text{ are compared,} \\ 0 & \text{otherwise.} \end{cases}$$

for each i and j. Let X be the number of comparisons performed,

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

and $Y_{ij} = (y_i, y_{i+1}, \ldots, y_j)$, y_i and y_j are compared $\iff$ the first pivot selected from Y_{ij} is y_i or y_j. So

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}.$$

since $Y_{ij} = (y_i, y_{i+1}, \ldots, y_j)$. That implies the expectation

$$E[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j - i + 1}.$$

Note: If $Z \in \{0, 1\}$ is $0 - 1$ valued, then Z is called an indicator random variable. Then $\Pr[Z = 1] = p$ and $\Pr[Z = 0] = 1 - p$, therefore $E[Z] = 1 \cdot \Pr[Z = 1] + 0 \cdot \Pr[Z = 0] = \Pr[Z = 1]$. Then

$$\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\
&= 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&= 2 \sum_{k=2}^{n} \sum_{i=1}^{n-k+1} \frac{1}{k} \\
&= 2 \sum_{k=2}^{n} \frac{n - k + 1}{k} \\
&= 2 \sum_{k=2}^{n} \left(\frac{n+1}{k} - 1\right) \\
&= 2 \left[(n+1) \sum_{k=2}^{n} \frac{1}{k} - (n-1)\right] \\
&= 2 \left[(n+1) \sum_{k=1}^{n} \frac{1}{k} - (n-1) - (n+1)\right] \\
&= 2 \left[(n+1) H_n - 2n\right] \\
&= 2(n+1) H_n - 4n \\
&= 2(n+1)\Theta(\log n) - 4n \\
&= \Theta(n \log n).
\end{aligned}$$

- A different proof: Probablistic recurrence relation:

$$C(n) = \frac{1}{n} \sum_{k=1}^{n} [(n-1) + C(k-1) + C(n-k)]. = (n-1) + \frac{2}{n} \sum_{k=1}^{n-1} C(k). \tag{2}$$

$$nC(n) = n(n-1) + 2 \sum_{k=1}^{n-1} C(k) \tag{3}$$

$$(n-1)C(n-1) = (n-1)(n-2) + 2 \sum_{k=1}^{n-2} C(k) \tag{4}$$

Then (3) - (4) gives

$$nC(n) - (n-1)C(n-1) = n(n-1) - (n-2)(n-1) - 2 \sum_{k=1}^{n-2} C(k) + 2 \sum_{k=1}^{n-1} C(k)$$

$$
\begin{aligned}
&= 2(n-1) + 2C(n-1) \\
\implies nC(n) &= 2(n-1) + (n+1)C(n-1) \\
\implies C(n) &= \frac{2(n-1)}{n} + \frac{(n+1)C(n-1)}{n} \\
\implies \frac{C(n)}{n+1} &= \frac{2(n-1)}{n(n+1)} + \frac{C(n-1)}{n} \\
&= \frac{2(n-1)}{n(n+1)} + \frac{2(n-2)}{n(n-1)} + \frac{C(n-2)}{n-1} \\
&= 2\sum_{k=2}^{n} \frac{k-1}{k(k+1)} \\
&= 2\sum_{k=2}^{n} \left(\frac{1}{k+1} - \frac{1}{k(k+1)}\right) \\
&= 2\left(\sum_{k=2}^{n} \frac{1}{k+1} - \sum_{k=2}^{n} \frac{1}{k(k+1)}\right) \\
&= 2\left[\sum_{k=2}^{n} \frac{1}{k+1} - \sum_{k=2}^{n} \left(\frac{1}{k} - \frac{1}{k+1}\right)\right] \\
&= 2\left[\sum_{k=3}^{n+1} \frac{1}{k} - \left(\frac{1}{2} - \frac{1}{n+1}\right)\right] \\
&= 2\left[\sum_{k=1}^{n} \frac{1}{k} + \frac{1}{n+1} - 1 - \frac{1}{2} - \left(\frac{1}{2} - \frac{1}{n+1}\right)\right] \\
&= 2H_n - \frac{4n}{n+1} \\
\implies C(n) &= 2(n+1)H_n - 4n.
\end{aligned}
$$

2.6.9 Finding Medians and Order Statistics

Let S be an (unsorted) array of n elements with no duplicates.

- If $|S|$ is odd, then the median of S is the middle element of S when sorted.
- If $|S|$ is even, then there are two medians, $|S| = n = 2k \implies$ elements of ranks k and $k+1$ are medians.

In any case, an element of rank $\lfloor \frac{n}{2} \rfloor + 1$ is a median. More generally, the ith-order statistic is the element of rank i.

- Sort S, select middle element (or desired rank).
  - $\Theta(n \log n)$ time: MergeSort.
  - Expected $\Theta(n \log n)$ time: QuickSort.
- QuickSelect (randomized algorithm)
  - $O(n)$ exptected time
  - $O(n^2)$ worst case time
- Deterministic divide-and-conquer algorithm:
  - $O(n)$ time with large constants.

Example 2.5. $n = 8$ and $k = 5$, 5th order statistic.

$$S = [4, 12, 3, 8, 2, 6, 15, 5]$$

$$
\begin{aligned}
S &= [3, 2, 4, 12, 8, 6, 15, 5] \\
S &= [12, 8, 6, 15, 5] \\
S &= [8, 6, 5, 12, 15] \\
S &= [8, 6, 5] \\
S &= [6, 5, 8] \\
S &= [6, 5] \\
S &= [5, 6]
\end{aligned}
$$

Algorithm 7: Quick Select

```
Function QuickSelect(S[1...n], k):
    Input: Select the kth order statistic from S
    Output: Return the k the order statistic
    if n = 1 then
        return S[1];
    end
    Choose a pivot p from S;
    /* Partition into two subarrays:                                  */
    S_1 = elements of S that are < p;
    S_2 = elements of S that are > p;
    r ← |S_1| + 1;
    /* the rank of p                                                  */
    if r = k then
        return p;
    else if k < r then
        return QuickSelect(S_1, k);
    else
        return QuickSelect(S_2, k − r);
    end
end
```

- Worst case: $\binom{n}{2} = \Theta(n^2)$ comparisons. Elements of ranks $n, n-1, n-2, \ldots, k+1$ are chosen, as pivots, then ranks $1, 2, 3, \ldots, k-1$. Problem size decreases by 1 each time:

$$\#\text{ of comparisons} = \sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} i = \binom{n}{2}.$$

- For $r \in \{1, \ldots, n\}$, let

$$X_r = \begin{cases} 1 & \text{if pivot of rank } r \text{ is chosen,} \\ 0 & \text{otherwise.} \end{cases}$$

Then $\Pr[X_r = 1] = \frac{1}{n} = E[X_r]$. Subproblem size:

$$Y = \sum_{r=1}^{k-1} X_r(n-r) + \sum_{r=k+1}^{n} X_r(r-1).$$

The expected subproblem size would be

$$
\begin{aligned}
E[Y] &= E\left[\sum_{r=1}^{k-1} X_r(n-r) + \sum_{r=k+1}^{n} X_r(r-1)\right] \\
&= E\left[\sum_{r=1}^{k-1} X_r(n-r)\right] + E\left[\sum_{r=k+1}^{n} X_r(r-1)\right]
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{r=1}^{k-1} E[X_r(n-r)] + \sum_{r=k+1}^{n} E[X_r(r-1)] \\
&= \sum_{r=1}^{k-1} E[X_r](n-r) + \sum_{r=k+1}^{n} E[X_r](r-1) \\
&= \frac{1}{n}\left[\sum_{r=1}^{k-1}(n-r) + \sum_{r=k+1}^{n}(r-1)\right] \\
&= \frac{1}{n}\left[\sum_{i=n-k+1}^{n-1} i + \sum_{r=k}^{n-1} r\right] \\
&= \frac{1}{n}\left[\left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{n-k} i\right) + \left(\sum_{r=k}^{n-1} r - \sum_{r=1}^{k-1} r\right)\right] \\
&= \frac{1}{n}\left[\binom{n}{2} - \binom{n-k+1}{2} + \binom{n}{2} - \binom{k}{2}\right] \\
&= (n-1) - \frac{1}{n}\left[\binom{n-k+1}{2} + \binom{k}{2}\right]
\end{aligned}
$$

Now let's take a closer look at $\binom{n-k+1}{2} + \binom{k}{2}$, we have

$$
\begin{aligned}
\binom{n-k+1}{2} + \binom{k}{2} &= \frac{k(k-1)}{2} + \frac{(n-k+1)(n-k)}{2} \\
&= \frac{1}{2}\left[k^2 - k + (n-k)^2 + (n-k)\right] \\
&= \frac{1}{2}\left[2k^2 - 2k(n+1) + n^2 + n\right] \\
&= k^2 - k(n+1) + \frac{1}{2}n^2 + \frac{1}{2}n.
\end{aligned}
$$

Differentiating with respect to (w.r.t.) k yields

$$
2k - (n+1) = 0 \text{ when } k = \frac{n+1}{2}.
$$

Thus, we have

$$
\begin{aligned}
\binom{n-k+1}{2} + \binom{k}{2} &= \binom{\frac{n}{2}+\frac{1}{2}}{2} + \binom{\frac{n}{2}+\frac{1}{2}}{2} \\
&= 2\binom{\frac{n}{2}+\frac{1}{2}}{2} \\
&= \frac{n+1}{2}\frac{n-1}{2} \\
&= \frac{n^2-1}{4}.
\end{aligned}
$$

Then

$$
E[Y] = (n-1) - \frac{1}{n}\frac{n^2-1}{4} = \frac{3n}{4} + \frac{1}{4n} - 1.
$$

Let Y_i be the problem size in ith call to QuickSelect, $Y_1 = n$, $E[Y_2] \leq \frac{3}{4}n = \frac{3}{4}Y_1$. More generally,

$$
E[Y_{i+1}|Y_i] \leq \frac{3}{4}Y_i.
$$

By induction,

$$
E[Y_i] \leq \left(\frac{3}{4}\right)^{i-1} n \text{ for all } i \geq 1.
$$

Let X_i be the number of comparisons done in the ith call.

$$X_i = \begin{cases} Y_i - 1 & \text{if } Y_i > 0, \\ 0 & \text{if } Y_i = 0. \end{cases}$$

Then, $E[X_i] = E[Y_i] - 1 \le E[Y_i] \le \left(\frac{3}{4}\right)^{i-1} n$. Let X be the total number of comparisons,

$$X = \sum_{i=1}^{\infty} X_i.$$

Hence,

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{\infty} X_i\right] \\ &\le \sum_{i=1}^{\infty} E[X_i] \\ &\le \sum_{i=1}^{\infty} \left(\frac{3}{4}\right)^i n \\ &\le n \sum_{i=1}^{\infty} \left(\frac{3}{4}\right)^i \\ &\le n \frac{0 - 3/4}{3/4 - 1} \\ &\le 3n \end{aligned}$$

2.6.10 Deterministic Selection in $O(n)$ Time (Median-of-median algorithm)

Suppose we have an array A of size n, then we break A into $n/5$ of 5, find median of each group by sorting $O(1)$ time per group, which has less than $\binom{5}{2}$ comparisons. Therefore, it takes $O(n)$ time to find all group medians. Next, form an array $M_{n/5}$ of the group medians. Recursive call to find meand x of $M_{n/5}$. Use x as the pivot to partition A: make recursive call as in QuickSelect on left or right half (See Algorithm 8).

Proposition 2.3. x is a good pivot, where x has rank between $\frac{3}{10}n$ and $\frac{7}{10}n$. Thus, the subproblem size decreases by at least 30%.

The running time would be

$$T(n) \le T(n/5) + T(7n/10) + O(n).$$

Intuitively, x should be *close* to the median of A.

1. x is greater than or equal to $m/2$ elements of M.
2. Each element of M is greater than or equal to 3 elements in its group.

That implies, x is greater than or equal to $m/2 \cdot 3 = 3n/10$ elements of A. Similarly, x is less than or equal to $m/2 \cdot 3 = 3n/10$ elements of A. Therefore the subarray of A (A_1 or A_2) that select is recursively called on has at most $7n/10$ elements.
Let $T(n)$ be the maximum time for select on an array of size n. Then

$$T(n) \le T(n/5) + T(7n/10) + O(n).$$

- Each of the $n/5$ groups is sorted using less than $\binom{5}{2} = 10$ comparisons, then we have less than $10 \cdot n/5 = 2n$ comparisons to sort all groups. In other words, it takes $O(n)$ to sort all groups.
- Form M: $O(n)$ time.
- Recursively find median of M: $T(n/5)$.

Algorithm 8: Deterministic Selection in $O(n)$ Time (Median-of-Medians algorithm)

```
Function Select(A[1...n], k):
    Input: A[1...n] is an array of n elements, where n is a power of 10.
    Output: Find the median of A[1...n].
    /* base case                                                        */
    if n = 1 then
        return A[1];
    end
    Let m = n/5. Partition A into m groups of 5 elements. Insertion sort each of the m groups;
    Let M be an array of size n containing the medians from each of the 5 gorups;
    Use Select to find the median x of M: /* x is the median-of-medians  */
    x ← Select(M[1...m], m/2);
    Partition A into two subarrays: A_1 = elements of A that are less than x;
    A_2 = elements of A that are greater than x;
    Let r = |A_1| + 1 be the rank of x in A;
    if r = k then
        return x;
    else if k < r then
        return Select(A_1[1...r − 1], k);
    else
        return Select(A_2[1...n − r], k − r);
    end
end
```

- Partition A around the median of medians: $n-1$ comparisons – $O(n)$ time.
- Recursively call select on a subarray of A of size less than $7n/10 \le T(7n/10)$

Proposition 2.4. $T(n) = O(n)$.

Proof. This is true if there is a constant c such that $T(n) \le c \cdot n$ for all sufficiently large n. Let

$$T(n) \le T(n/5) + T(7n/10) + an.$$

Suppose that $T(n) \le cn$ for some c. Then

$$T(n) \le c \cdot \frac{n}{5} + c \cdot \frac{7n}{10} + an = \left(\frac{9c}{10} + a\right) n \le cn, \text{ if } c \ge 10a.$$

□

In practice, $a = 3$ for comparisons, then $T(n) \le 30n$ comparison overall, while QuickSelect has less than $4n$ comparisons on average.

2.7 Dynamic Programming

- Divide-and-conquer: top-down
- Dynamic Programming: bottom-up

2.7.1 Longest Increasing Subsequence Problem

- Given a sequence of numbers $a_1, \ldots, a_n$.
- Goal: find a longest increasing subsequence, that is, find $i_1, \ldots, i_k$ such that $1 \le i_1 < i_2 < \cdots < i_k \le n$ and $a_{i1} < a_{i_2} < \cdots < a_{ik}$, where k is maximized.

Example 2.6. Let $a = [a_1, a_2, a_3, a_4, a_5, a_6, a_7 a_8] = [5, 2, 8, 6, 3, 6, 9, 7]$. The longest increasing subsequence could be $[2, 3, 6, 9]$ or $[2, 3, 6, 7]$.

Brute force: try all 2^n possible subsequence. Dynamic programming can reduce the time substatially. For each j, $1 \leq j \leq n$, write $L(j)$ for the longest increasing subsequence of $a_1, \ldots, a_j$, we will compute

$$L(1), L(2), L(3), \ldots, L(n),$$

in order.

Proposition 2.5. $L(j) = 1 + \max\{L(i) | a_i < a_j \text{ and } i < j\}$.

Go back to the example, we would have

$$\begin{aligned} L(0) &= 0, \\ L(1) &= 1, \\ L(2) &= 1, \\ L(3) &= 2, \\ L(4) &= 2, \\ L(5) &= 2, \\ L(6) &= 3, \\ L(7) &= 4, \\ L(8) &= 4. \end{aligned}$$

Algorithm 9: Finding longest increaseing subsequence based on dynamic programming

```
Function longestIncreasingSubsequence([a_1, a_2, ..., a_n]):
    Input: Unsorted sequence [a_1, a_2, ..., a_n].
    Output: Find the longest inreaseing subsequence.
    for j = 1 to n do
        /* Predcessor                                        */
        pred(j) = 0;
        max = 0;
        for i = 1 to j - 1 do
            if a_i < a_j and L(i) > max then
                max = L(i);
                pred(j) = i;
            end
        end
        L(j) = max + 1;
    end
    Then find j such that L(j) is maximized, where L(j) is the length of longest increasing
      subsequence;
    Follow pred links back to extract the sequence;
end
```

Could we use recursion? Take a look at the formula: $L(j) = 1 + \max\{L(i) | a_i < a_j \text{ and } i < j\}$.

- $L(n)$
  - $L(n-1)$
    - $L(n-2)$
    - $L(n-3)$

    - $\vdots$
    - $L(1)$
  - $L(n-2)$
    - $L(n-3)$
    - $\vdots$
    - $L(1)$
  - $\vdots$
  - $L(1)$

This would be exponential time. However, the same subproblems are solved over and over. This can be made efficient - "memoization"

2.7.2 Longest Common Subsequence (LCS)

Algorithm 10: Finding longest common subsequence based on dynamic programming

```
Function longestCommonSubsequence(x[1...n], y[1...m]):
    Input: Two strings x[1...n] and y[1...m]
    Output: Compute a longest common subsequence of x and y, that is, a string z[1...k] such
            that z is a subsequence both x and y and k is maximized
    for i = 0 to n do
        L(i, 0) = 0;
    end
    for j = 1 to m do
        L(0, j) = 0;
    end
    for i = 1 to n do
        for j = 1 to m do
            if x[i] = y[i] then
                L(i, j) = L(i − 1, j − 1) + 1;
            else
                L(i, j) = max{L(i − 1, j), L(i, j − 1)};
            end
        end
    end
end
```

Example 2.7. Given two strings $x = ABCBDAB, y = BDCABA$ then BCA is a common subsequence, $BCAB$ and $BCBA$ are the longest common subsequence.

$$\begin{bmatrix} & & B & D & C & A & B & A \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ B & 0 & 1 & 1 & 1 & 1 & 2 & 2 \\ C & 0 & 1 & 1 & 2 & 2 & 2 & 2 \\ B & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ D & 0 & 1 & 2 & 2 & 2 & 3 & 3 \\ A & 0 & 1 & 2 & 2 & 3 & 3 & 4 \\ B & 0 & 1 & 2 & 2 & 3 & 4 & 4 \end{bmatrix}$$

Then the LSC's are $BCBA$, $BDAB$, $BCAB$.

Write $L(i, j)$ for the length of a LCS of $x[1 \dots i]$ and $y[1 \dots j]$, where $0 \le i \le n$ and $0 \le j \le m$, let $z[1 \dots k]$ be a longest common subsequence of $x[1 \dots i]$ and $y[1 \dots j]$.

- If $x[i] = y[j]$, then $z[k] = x[i] = y[j]$.
- If $x[i] \ne y[j]$, then
  - If $z[k] \ne x[i]$, then $z[1 \dots k]$ is a LCS of $x[1 \dots i-1]$ and $y[1 \dots j]$.
  - If $z[k] \ne y[j]$, then $z[1 \dots k]$ is a LCS of $x[1 \dots i]$ and $y[1 \dots j-1]$.
  - If $z[k] \ne y[j]$ and $z[k] \ne x[i]$, then $z[1 \dots k]$ is a LCS of $x[1 \dots i-1]$ and $y[1 \dots j-1]$.

Corollary 2.1.

1. If $x[i] = y[j]$, then $L(i, j) = L(i-1, j-1) + 1$.
2. If $x[i] \ne y[i]$, then $L(i, j) = \max\{L(i-1, j), L(i, j-1)\}$.

2.7.3 Edit Distance

Example 2.8. 'SNOWY' and 'SUNNY':

$$\begin{bmatrix} & & S & U & N & N & Y \\ & 0 & 1 & 2 & 3 & 4 & 5 \\ S & 1 & 0 & 1 & 2 & 3 & 4 \\ N & 2 & 1 & 1 & 1 & 2 & 3 \\ O & 3 & 2 & 2 & 2 & 2 & 3 \\ W & 4 & 3 & 3 & 3 & 3 & 3 \\ Y & 5 & 4 & 4 & 4 & 4 & 3 \end{bmatrix}$$

Operation	Cost
insertion	1
deletion	1
mismatch	1
mutation	1

Let $E(i, j)$ be the cost of optimal alignment of $x[1 \dots i]$ and $y[1 \dots j]$. Then we have three possibilities for optimal of $x[1 \dots i]$ and $y[1 \dots j]$

- Cost $E(i-1, j-1)$: optimal alignment of $x[1 \dots i-1]$ and $y[1 \dots j-1]$ (either or mismatch)

  ```
  x[i]
  y[j]
  ```

  then

  $$E(i, j) = \begin{cases} E(i-1, j-1) & \text{if } x[i] = y[j] \\ E(i-1, j-1) + 1 & \text{if } x[i] \ne y[j] \end{cases}$$

- Cost $E(i-1, j)$: optimal alignment of $x[1 \dots i-1]$ and $y[1 \dots j]$ (deletion), then $E(i, j) = E(i-1, j) + 1$.

  ```
  x[i]
  ----
  ```

- Cost $E(i, j-1)$: optimal alignment of $x[1 \dots i]$ and $y[1 \dots j-1]$ (insertion), then $E(i, j) = E(i, j-1) + 1$.

  ```
  ----
  y[j]
  ```

The longest common subsequence is a special case of the edit cost by setting $match = -1$, $insertion/deletion/mutation = 0$.

Algorithm 11: Finding an optimal alignment (minimal cost) based on dynamic programming

```
Function editDistance(x[1...n], y[1...m]):
    Input: Two strings x[1...n] and y[1...m]
    Output:
    for i = 0 to n do
        E(i, 0) = i;
    end
    for j = 1 to m do
        E(0, j) = j;
    end
    for i = 1 to n do
        for j = 1 to m do
            E(i, j) = min{E(i − 1, j), L(i, j − 1)};
        end
    end
end
```

2.7.4 Knapsack Problem

- Knapsack capacity W, n items to choose from with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$.
- Goal: choose the most valuable collection of tiems that fit in the bag.

Two versions:

- With repitition: unlimited supply of each item.
- Without repitition (standard knapsack problem): only one of each item.
- **With repitition**
  - Subproblems: Knapsacks of capacity w, $1 \leq w \leq W$. (Another possibility is to consider fewer items - solve for items $1, 2, \ldots, i$ for $i \leq n$, or combine both approaches – vary both number of items and knapsack size).
  - Let $K(w)$ be the maximum value achievable in a knapsack of capacity w.
  - Suppose that the last item added to achieve $K(w)$ (optimal solution) is item i with weight w_i and value v_i. Take item i out of the knapsack: frees up w_i weight and decreases value by v_i. We're left with a set of items that fits in a knapsack of capacity $w - w_i$ and has value $K(w) - v_i$. This must be an optimal solution for capacity $w - w_i$. (If it isn't, pick a better solution, add item i to it, and we have a better solution for capacity $K(w)$, a contradiction.) Then we have

    $$K(w - w_i) = K(w) - v_i \implies K(w) = K(w - w_i) + v_i,$$

    which assumes ith item is added last. Then

    $$\begin{cases} K(0) = 0, & \text{(base case)} \\ K(w) = \max_{1 \leq i \leq n, w_i \leq w}\{K(w - w_i) + v_i\}. \end{cases}$$

- **Without repitition**
  - Subproblems: Knapsacks of weight w, $0 \leq w \leq W$ using items $1, 2, \ldots, j$, $0 \leq j \leq n$. Let $K(w, j)$ be the maximum value achievable using knapsack of capcity w and items $1, \ldots, j$. Consider an optimal solution s for $K(w, j)$:
    * Case 1: Item j is not included, then s is also an optimal solution for $K(w, j) = K(w, j - 1)$.

Algorithm 12: Dynamic Programming for Knapsack Problem with repitiion $O(nW)$ time

```
Function knapsack(w[1...n], v[1...n]):
    Input:
    Output:
    K(0) = 0;
    for w = 1 to W do
        K(w) = max{K(w - w_i) + v_i | 1 ≤ i ≤ n, w_i ≤ w};
    end
    return K(W);
end
```

∗ Case 2: Item j is included, then remove item j: $s - \{j\}$ has weight $w - w_j$ and value $K(w, j) - v_j$. $s - \{j\}$ is an optimal solution for $K(w - w_j, j - 1) = K(w, j) - v_j \implies K(w, j) = K(w - w_j, j - 1) + v_j$. Then

$$K(w_j) = \begin{cases} \max\{K(w, j-1), K(w - w_j, j-1) + v_j\}, & \text{if } w_j \leq w, \\ K(w, j-1), & \text{otherwise.} \end{cases}$$

Algorithm 13: Dynamic Programming for Knapsack Problem with repitiion $O(nW)$ time

```
Function knapsack(w[1...n], v[1...n]):
    Input:
    Output:
    Initialize K(0, j) = 0 for 0 ≤ j ≤ n and K(w, 0) = 0 for 1 ≤ w ≤ W;
    for j = 1 to n do
        for w = 1 to W do
            if w_j > w then
                K(w, j) = K(w, j - 1);
            else
                K(w, j) = max{K(w, j - 1), K(w - w_j, j - 1) + v_j};
            end
        end
    end
    return K(W, n);
end
```

Example 2.9. Let $W = 10$, and Then we have

item	weight	value
1	6	30
2	3	14
3	4	16
4	2	9

- Another approach:

Let the total value $V = \sum_{i=1}^{n} v_i$. For all $0 \leq v \leq V$ and $0 \leq j \leq n$, let $K(v, j)$ be the minimum weight to attain value exactly v with items $1, 2, \ldots, j$, and $K(w, j) = \infty$ if not possible to get value v with those items.

$$K(v, j) = \begin{cases} \min\{K(v, j-1), K(v - v_j, j-1) + v_j\} & \text{if } v_j \leq v, \\ K(v, j-1) & \text{otherwise.} \end{cases}$$

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

The base cases are $K(v, 0) = \infty$ for all $1 \leq v \leq V$, and $K(0, j) = 0$ for all $0 \leq j \leq n$. Leads to an $O(nV)$ time algorithm. After all values computed, look for the v that maximizes $K(v, n)$ and $K(v, n) \leq W$.

2.7.5 Matrix Chain Multiplication

Example 2.10. Given matrices $A_{50\times20}, B_{20\times1}, C_{1\times10}, D_{10\times100}$, want to compute product $ABCD$, matrix multiplication is as associative: $A(BC) = (AB)C$. We have

- $(AB)(CD)$: 7000;
- $(A(BC))D$: 60200;
- $A((BC)D)$: 120200;
- $((AB)C)D$: 51500;
- $A(B(CD))$: 13000;

For $A_{j\times k}$ and $B_{k\times l}$, computing AB with standard algorithm takes jkl operations (multiplications).
How many parenthsesization are there? Let P_n be the number of ways that n factors can be parenthesized.

- $n = 1$: (A), $P_1 = 1$.
- $n = 2$: (AB), $P_2 = 1$.
- $n = 3$: $A(BC)$ and $(AB)C$, $P_3 = 2$.
- $n = 4$: $(AB)(CD)$: $P_2 \cdot P_2$, $(A(BC))D$ and $((AB)C)D$: $P_3 \cdot P_1$, $A((BC)D)$ and $A(B(CD))$: $P_1 \cdot P_3$, then $P_4 = P_3 \cdot P_1 + P_2 \cdot P_2 + P_1 \cdot P_3 = 5$.
- $n = 5$: Possible splits: $(1, 4)$, $(2, 3)$, $(3, 2)$, $(4, 1)$. Then $P_5 = P_1 \cdot P_4 + P_2 \cdot P_3 + ¶_3 \cdot P_2 + P_4 \cdot P_1 = 5 + 2 + 2 + 5 = 14$.
- More generally, $P_n = \sum_{i=1}^{n-1} P_i \cdot P_{n-i}$, which is closedly related to Catalan numbers: $C_1, C_2, \ldots$, then $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$. Therefore, $P_n = C_{n-1}$. Closed formula is $C_n = \frac{1}{n+1}\binom{2n}{n} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$ which is exponential in n, so is P_n. Worth mentioning, P_n is also the number of full binary with n leaves (see Figure 1).

- Given: n matrices $A_1, A_2, \ldots, A_n$ of demensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, where A_i has dimension $m_{i-1} \times m_i$.
- Goal: compute the optimal parenthesization (minimize the number of operations to compute product).
- Look for substructure: What multiplication is done last? There are $n - 1$ possibilities:
  Let $C(i, j)$ be the optimal cost for multiplying $A_i A_{i+1} \cdots A_j$. Then

$$C(1, n) = \min_{1 \leq i < n} \{C(1, i) + C(i + 1, n) + m_0 m_i m_n\},$$

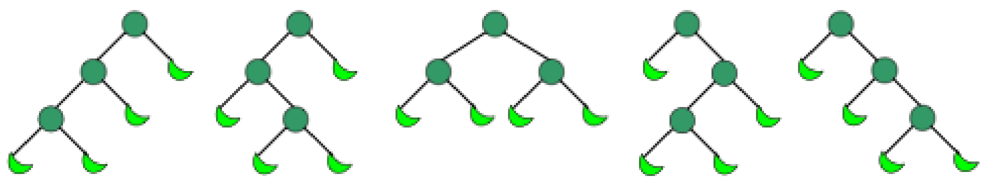


Figure 1: Catalan number binary tree example ($n = 4$)

Operations	Cost
$(A_1)(A_2 \cdots A_n)$	$C(1,1) + C(2,n) + m_0 m_1 m_n$
$(A_1 A_2)(A_3 \cdots A_n)$	$C(1,2) + C(3,n) + m_0 m_2 m_n$
$(A_1 A_2 A_3)(A_4 \cdots A_n)$	$C(1,3) + C(4,n) + m_0 m_2 m_n$
$\vdots$	$\vdots$
$(A_1 A_2 \cdots A_i)(A_{i+1} \cdots A_n)$	$C(1,i) + C(i+1,n) + m_0 m_2 m_n$
$\vdots$	$\vdots$
$(A_1 A_2 \cdots A_{n-1}) A_n$	$C(1,n-1) + C(n,n) + m_0 m_2 m_n$

$$
\begin{aligned}
C(i,j) &= \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} m_k m_j\}, \\
C(i,i) &= 0, \forall 1 \le i \le n.
\end{aligned}
$$

Algorithm 14: Dynamic Programming for Multiplying Matrices Chain

```
Function MMC(A_1 A_2 ... A_n):
    Input:
    Output:
    for i = 1 to m do
        C(i,i) = 0;
    end
    for s = 1 to n − 1 do
        for i = 1 to n − s do
            j = i + s;
            C(i,j) = min_{i≤k<j}{C(i,k) + C(k+1,j) + m_{i−1} m_k m_j};
        end
    end
    return C(1,n);
end
```

Computes two-dimensional table. Backtrack: to get optimal parenthesization, splitting on the index that attained the minimum.
The running time would be

$$
\begin{aligned}
\sum_{s=1}^{n-1} \sum_{i=1}^{n-s} \sum_{k=i}^{i+s} 1 &= \sum_{s=1}^{n-1} \sum_{i=1}^{n-s} (s+1) \\
&= \sum_{s=1}^{n-1} (n-s)(s+1)
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{s=1}^{n-1}(ns - s^2 + n - s) \\
&= n\binom{n}{2} - \frac{(n-1)n(2n-1)}{6} + n(n-1) - \binom{n-1}{2} \\
&= \frac{n^3 - n^2)}{2} - \frac{2n^3 - 3n^2 - n + 1}{6} + n^2 - n - \frac{n^2 - n}{2} \\
&= O(n^3).
\end{aligned}
$$

2.7.6 Approximation Algorithm for Kanpsack

FPTAS - fully polynomial-time approximation scheme, n is the number of items. Let OPT be the value of the optimal solution. The approximation algorithm will produce a solution with value $\geq (1-\varepsilon)OPT$, for any $\varepsilon > 0$, in time polynomial in n and 1/2. The running time for this algorithm is $O(n^2 \cdot 1/\varepsilon)$.
Note: PTAS, e.g., $O(n^{1/\varepsilon})$ polynomial for each fixed ε.
Let $V = \max_i v_i$. Define for all $v \leq nV$ and $i \leq n$, $A(v, i)$ be the mininmal weight of a subset of $1, \ldots, i$ with total value exactly equal to v; ∞ if does not exist.

$$
\begin{cases}
A(v,i) = \min\{A(v - v_i, i-1), A(v, i-1)\}, & \text{if } v \leq v_i, \\
A(v, i-1), & \text{otherwise.}
\end{cases}
$$

That leads to the dynamic programming algorithm: $O(n^2V)$, where $nV \cdot n$ entries of A to compute. Note that V may be exponentially large in n (values/weights encoded in binary). If the values are small (bounded by polynomial in n, e.g., $v_i \leq n^3$, for all i, $\implies O(n^5)$ time), this algorithm runs polynomial time. We will scale (round) the values to be small (divide by some "large" number) for our approximation algorithm.

Algorithm 15: FPTAS for knapsack

Function ():
  Input: Knapsack instance, approximation parameter $\varepsilon > 0$. items $1, \ldots, n$, values $v_1, \ldots, v_n$, weights $w_1, \ldots, w_n$, capacity W
  Output:
  $V = \max_i v_i$;
  $D = \frac{\varepsilon V}{n}$;
  For each object i, define $v_i^{'} = \lfloor \frac{v_i}{D} \rfloor$;
  Run the dynamic programming algorithm using the $v_i^{'}$ values to obtain a solution $S' \subset \{1, \ldots, n\}$. Output S'.
end

The running timeis $V' = max_i v_i'$ and

$$
V' = \lfloor V/D \rfloor = \left\lfloor V \cdot \frac{n}{\varepsilon V} \right\rfloor = \left\lfloor \frac{n}{\varepsilon} \right\rfloor = O(\frac{n}{\varepsilon}).
$$

Let OPT be the value of optimal solution for original instance. We have the following lemma.

Lemma 2.1.

$$
\sum_{i \in S'} v_i \geq (1-\varepsilon) \cdot OPT.
$$

We interpret the left-hand side as solution to original instance (original values).

Proof. Let $\mathcal{O} \subset \{1, \ldots, n\}$ be an optimal solution.

$$
\sum_{i \in \mathcal{O}} v_i = OPT.
$$

For each object i, $v_i \geq D \cdot v'_i \geq v_i - D$. Therefore

$$\begin{aligned}
\sum_{i \in \mathcal{O}} v_i &\geq D \cdot \sum_{i \in \mathcal{O}} v'_i \\
&\geq \sum_{i \in \mathcal{O}} (v_i - D) \\
&= \sum_{i \in \mathcal{O}} v_i - D|\mathcal{O}| \quad (|\mathcal{O} \leq n) \\
&\geq OPT - Dn \quad (Dn = \frac{\varepsilon V}{n} \cdot n = \epsilon V) \\
&= OPT - \varepsilon V \quad (V \leq OPT, \text{assuming all items fit in knapsack}) \\
&\geq OPT - \varepsilon OPT. \\
&= OPT(1 - \varepsilon).
\end{aligned}$$

The solution S' from the dynamic programming algorithm satisfies

$$\sum_{i \in S'} v'_i \geq \sum_{i \in \mathcal{O}} v'_i,$$

because S' is optimal for the rounded values. Then we have

$$\sum_{i \in S'} v_i \geq D \sum_{i \in S'} v'_i \geq D \sum_{i \in \mathcal{O}} v'_i \geq OPT(1 - \varepsilon).$$

Therefore S' has value $\geq (1 - \varepsilon)OPT$. Completed in time $O(n^3 \frac{1}{\varepsilon})$, thus the algorithm is an FPTAS. □

Knapsack is NP-complete - all known poly-time algorithms for exact solutions have worst-case exponential run time.

2.7.7 All Pairs Shortest Paths

- Undirected graph with vertices $\{1, 2, \ldots, n\}$, $l(i, j)$ is the length (or cost) from i to j [$l(i, j) = \infty$ if no edge].
- Goal: compute shortest paths for all pairs of vertices i and j.
- Define $\text{dist}(i, j, k)$ to be the length of shortest path from i to j using only vertices from $1, 2, \ldots, k$ as intermediate nodes, where $1 \leq i, j, k \leq n$.
- The idea is to compute $\text{dist}(i, j, 0), \ldots, \text{dist}(i, j, n)$. Initially, $\text{dist}(i, j, 0) = l(i, j)$. Relate $\text{dist}(i, j, k)$ to smaller problems.
- $i - -k : \text{dist}(i, k, k - 1)$.
- $i - -j : \text{dist}(i, j, k - 1)$.
- $k - -j : \text{dist}(k, j, k - 1)$.
- Two possibilities for $\text{dist}(i, j, k)$.
  - $\text{dist}(i, j, k - 1)$: don't use vertex k.
  - $\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1)$: use vertex k as an intermediate node.
- Compute $\text{dist}(i, j, k) = \min\{\text{dist}(i, j, k - 1), \text{dist}(i, k, k - 1), \text{dist}(k, j, k - 1)\}$.

2.7.8 Traveling Salesman Problem (TSP)

- Instance: n cities numbered $1, \ldots, n$, for each pair i, j of cities, d_{ij} is the distance (cost of traveling) from i to j. (not necessarily symmetric $d_{ij} \neq d_{ji}$ is possible) [Complte weighted directed graph]

Algorithm 16: Floyd-Warshall Algorithm

```
Function ():
    Input:
    Output:
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, 0) = l(i, j);
        end
    end
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
                dist(i, j, k) = min{dist(i, j, k − 1), dist(i, k, k − 1), dist(k, j, k − 1)};
            end
        end
    end
end
```

- Goal: find an optimal tour of the n cities: start at 1, visit each city exactly once, and return to 1 with minimum total distance. Let $[n] = \{1, \dots, n\}$, find a permutation $\pi : [n] \to [n]$ such that

$$c(\pi) = \left[\sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)}\right] + d_{\pi(n),\pi(1)}$$

is minimized.
- There are $(n-1)!$ permutations to consider: $1, \pi(2), \pi(3), \dots, \pi(n)$. Brute force search (consider all permutations) is $O(n!) = O(2^{n \log n})$ time.
- Subproblems: Let $S \subset [n]$ with $1 \in S$ and $j \in S$, find the path from 1 to j that visits all cities in S with minimum total cost. For $S \subset [n]$ with $1, j \in S$, define $C(S, j)$ to be the length of shortest path from 1 to j that visits each city in S exactly once.

$$\min_j C([n], j) + d_{j1} = \text{ cost of optimal tour.}$$

- Base case: $C(\{1\}, 1) = 0$. $C(S, 1) = \infty$ if $|S| > 1$.
- How to compute $C(S, j)$, suppose the second to last city on the optimal path through S from 1 to j is i. Then

$$C(S, j) = \min_{i \in S : i \neq j} C(S - \{j\}, i) + d_{ij}.$$

Run time: $\leq 2^n$ subsets of $[n]$, $\leq n$ subproblems $C(S, j)$ for each subset S. $O(n)$ time for each subproblem $\implies O(2^n n^2) = O(2^{n+2\log n})$. To be more exact,

$$\sum_{s=2}^{n} \binom{n-1}{s-1}(s-1)^2 = \sum_{s=1}^{n-1} \binom{n-1}{s} s^2 \leq n^2 \sum_{s=1}^{n-1} \binom{n-1}{s} = n^2(2^{n-1} - 1) = O(2^n n^2).$$

All known exact algorithms for TSP require exponential time. Can get fast approximate solutions in some special cases. TSP with triangle inequality

$$d_{ij} \leq d_{ik} + d_{kj}, \forall i, j, k.$$

Polynomial-time 2-approximation algorithm (at most twice optimal cost), based on minimum spanning tree algorithms. With little more work, which uses minimum cost perfect matching, can be improved to 3/2-approximation.

- Euclidean TSP: distances are Euclidean distances cities are points in the plane (or $\mathbb{R}^n$). There is a PTAS. For each fixed ε, get a $(1 + \varepsilon)$-approximation solutions in time polynomial in n.

Algorithm 17: Dynamic Programming for TSP

```
Function TSP():
    Input:
    Output:
    C({1}, 1) ← 0;
    for s = 2 to n do
        for all S ⊂ [n] with |S| = s and 1 ∈ S do
            C(S, 1) = ∞;
            for all j ∈ S, j ≠ 1 do
                C(S, j) = min_{i∈S,i≠j}{C(S − {j}) + d_ij};
            end
        end
    end
    return min_{j≠1}{C([n], j) + d_j1};
end
```

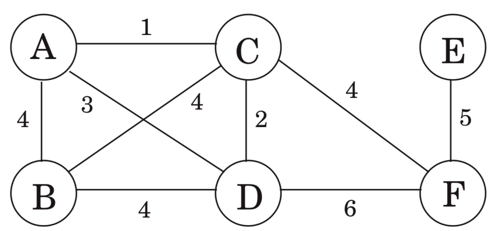
2.8 Greedy Algorithms

Make locally optimal decisions, (e.g., continually extending a partial solution one step at a time with the decision that looks best at the moment, the greedy choice). Prove this leads to a globally optimal solution. Of course, only works for some problems and some greedy strategies.

2.8.1 Minimum Spanning Tree (MST)

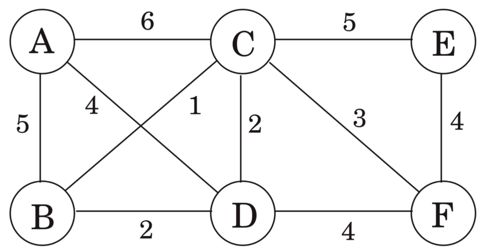
Given a weighted, connected, undirected graph, compute a spanning tree (a tree that includes all the vertices) of minimum total weight.
More formally:

- Instance: undirected graph $G = (V, E)$, where $E \subset V \times V$, edge weights w_e for each $e \in E$.
- Goal: compute a tree $T = (V, E')$ with $E' \subset E$ that minimizes weight$(T) = \sum_{e \in E'} w_e$.



Example 2.11.

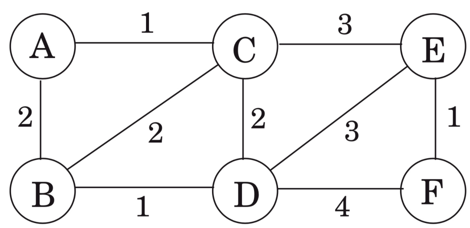
- Properties of Trees
  - A tree is a connected, acyclic graph.
  - A tree on n vertices has $n - 1$ edges.
  - A connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ edges is a tree.
  - An undirected graph is a tree if and only if there is a unique path between each pair of vertices.
- Greedy Strategy (Krsukal's Algorithm)
  - Start with empty graph.
  - Repeatedly add the next lightest edge that does not induce a cycle.
  - We need to show correctness (this always yields a MST)
  - How to implement efficiently



Example 2.12.

- Cut Property: Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick any subset $S \subset V$ for which X does not cross from S to $V - S$. Let e be the lightest edge across the S, $V - S$ partition. Then $X \cup \{e\}$ is part of MST.

Proof. X is part of some MST T. If is part of T, there is nothing to prove. Suppose e is not part of T. We will construct a new MST T' that contains $X \cup \{e\}$ by modifying T. Add e to T, which creates a cycle, so there must be some other edge e' corssing the cut. Let $T' = T - \{e'\} \cup \{e\}$. T' is also a tree - connected, acyclic, same numbers of edges, vertecies. weight(T') = weight$(T) - w_{e'} + w_e$ and $w_e \leq w_{e'}$ (because e is a lightest edge across the cut). That implies weight$(T') \leq$ weight(T). Since T is a MST, weight$(T) \leq$ weight(T'), so weight(T) = weight(T') and T' is also a MST. T' contains $X \cup \{e\}$. □



Example 2.13.

Algorithm 18: Kruskal's algorithm

Function MST(E)**:**
  Input:
  Output:
  Repeatedly add the next lightest edge that does not induce a cycle;
end

- e is lightest edge that does not make a cycle
- each edge across cut does not make cycle
- $\implies$ e is lightest edge across cut.
- $\implies$ e is safe to add by the cut property.

2.8.2 Disjoint Sets Data Structure (also called Union-Find Data Structure)

- For efficient implementation of Kruskal's algorithm.
  - Kruskal's algorithm maintain a forest that is a subgraph of a MST.
  - Initially, each vertex is in its own tree.

  - In each step, two trees in the forest are merged.
  - We will store the trees as sets in this data structure.
- Operations: for elements x and y of the universe, e.g. vertices, under consideration
  - `makeset(x)`: create a singleton set containing `x`.
  - `find(x)`: to which set does `x` belong?
  - `union(x, y)`: merge the sets containing `x` and `y`.
- Implementation - use trees

Algorithm 19: makeset with run time $O(1)$

```
Function makeset(x):
    Input: π is the parent points, unless root node then points to itself, rank is the height of
           subtree rooted at x
    Output:
    π(x) ← x;
    rank(x) ← 0;
end
```

Algorithm 20: find with run time $O(\log n)$ which is proportional to depth of x in its tree depth which is less than or equal to $\log n$

```
Function find(x):
    Input:
    Output: returns root of x's tree
    while x ≠ π(x) do
        x ← π(x);
    end
    return x;
end
```

Example 2.14. Given elements A, B, C, D, E, F, G.

1. $\text{makeset}(A), \text{makeset}(B), \ldots, \text{makeset}(G)$:
$$A^0 \quad B^0 \quad C^0 \quad D^0 \quad E^0 \quad F^0 \quad G^0.$$
2. $\text{union}(A, D), \text{union}(B, E), \text{union}(C, F)$
$$A^0 \to D^1 \quad B^0 \to E^1 \quad C^0 \to F^0 \quad G^0.$$
3. $\text{union}(C, G), \text{union}(E, A)$
$$C^0 \to F^1 \leftarrow G^0 \quad B^0 \to E^1 \to D^2 \leftarrow A^1.$$
4. $\text{union}(B, G)$.

Proposition 2.6.

Property 1. For any x, $\text{rank}(x) < \pi(\text{x})$. (ranks along a path to a root are strictly increasing)

Property 2. A root node of rank k has at least 2^k nodes in its tree.

Proof. A root node of rank k is formed by joining two trees of rank $k-1$. Statement follows by induction:

Algorithm 21: union with run time $O(\log n)$

```
Function union(x):
    Input:
    Output:
    r_x ← find(x);
    r_y ← find(y);
    if r_x = r_y then // x and y are already in the same set
        return
    end
    if rank(r_x) > rank(r_y) then
        π(r_y) = r_x;
    else
        π(r_x) = r_y;
        if rank(r_x) = rank(r_y) then
            rank(r_y) ← rank(r_y) + 1;
        end
    end
    return;
end
```

- $k = 0 \implies$ 1 node (makeset)
- if statement is true for $k-1$, then it is also true for k two trees of rank $k-1$ have greater than 2^{k-1} nodes each. That implies resulting tree of rank k has greater than $2^{k-1} + 2^{k-1}i = 2^k$ nodes.

□

Property 3. If there are n elements overall, there are at most $n/2^k$ elements of rank k.

Proof. Let R be the number of elementsof rank k. Then there are more than $R \cdot 2^k$ nodes in these R trees. Because there are n nodes overall,

$$R2^k \leq n \implies R \leq \frac{n}{2^k}.$$

□

Corollary 2.2. The maximum rank is less than $\log n$.

Kruskal's algorithm maintains a collection of connected components (trees). Initially, each vertex is in its own components. Repeatedly joins components by adding the next lightest edge.

- Run time:
  - $|V|$ makeset operations (intialization)
  - $2|E|$ find operations: $O(\log |V|)$ (look up endpoints of each edge)
  - $|V| - 1$ union operations: $O(\log |V|)$ (merging trees)
  - sort E: $O(|E| \log |E|) = O(|E| \log |V|)$.

2.8.3 Some optimization for the data structure

The path compressed find(x) has "typical" run time $O(1)$. Formally, the amortized run time is $O(\log^* n)$, where $\log^* n = \min\{i | \log^{(i)} n \leq 1\}$ = number of times the logarithm needs to be taken to get down to 1. That means any sequence of n operations takes at most $O(n \log^* n)$ time.

Algorithm 22: Kruskal's algorithm

```
Function ():
    Input: Connected, weighted graph G = (V, E) with edge weights w_e
    Output: MST defined by edges X
    for v ∈ V do
        makeset(v);
    end
    X ← ∅;
    Sort the edges E by weight;
    for {u, v} ∈ E in increasing order of weight do
        if find(u) ≠ find(v) then
            add edge (u, v) to X;
            union(u, v);
        end
    end
    return X;
end
```

Algorithm 23: find (based on path compression)

```
Function find(x):
    Input:
    Output:
    if x ≠ π(x) then
        π(x) = find(π(x));
    end
    return π(x);
end
```

Example 2.15. 1. $\log^* 2 = 1$.

2. $\log^* 4 = 2$.

3. $\log^* 16 = 3$.

4. $\log^* 2^{16} = 4$.

5. $\log^* 2^{2^{16}} = 5$.

2.8.4 Amortized Analysis Example

- Binary counter on n bits.
  - increment operation
    - bits cost
    - 00000: 0
    - 00001: 1
    - 00010: 2
    - 00011: 1
    - 00100: 3
    - 00101: 1
    - 00110: 2
    - 00111: 1
    - 01000: 4
    - $\vdots$
    - 11111: 1
    - 00000: 5
  - worst case: flip n bits
  - most of the time doing less
    - 1/2 of time (2^{n-1}): 1 flip
    - 1/4 of time (2^{n-2}): 2 flips
    - 1/8 of time (2^{n-3}): 3 flips
    - $\vdots$
    - $1/2^k$ of time (2^{n-k}): k flips
    - $\vdots$
    - $1/2^n$ of time (1): n flips
  - The total cost over 2^n increments:

$$2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + \cdots + 2^{n-k} \cdot k + \cdots + 1 \cdot n = \sum_{k=1}^{n} 2^{n-k} k \leq 2 \cdot 2^n.$$

2.8.5 Amortized analysis (accounting method)

- n elements
  - All ranks are between 0 and $\log n$ (we prove max rank is less than $\log n$)
- Divide positive ranks into intervals:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, \ldots, 16\}, \{17, \ldots, 2^{16} = 65536\}, \ldots, \text{ up to } \log n.$$

  - intevarls are of form $\{k+1, \ldots, 2^k\}$ up to $2^k = \log n$ (assume n is a power of 2 for simplicity)
  - number of intervals is $\log^* n$.
- Start with $n \log^* n$ dollars.
  - Each operation must be paid for with dollars.

- Each node is given an allowance, when it ceases to be a root node.
- If the rank is in the interval $\{k+1, \dots, 2^k\}$, the node receives 2^k dollars.
- The number of nodes with rank greater than k is less than

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \cdots = n\left(\frac{1}{2^{k+1}} + \frac{1}{2^{k+2}} + \frac{1}{2^{k+3}} + \cdots\right) = n \cdot \frac{1}{2^k} = \frac{n}{2^k}.$$

  That implies for the interval $\{k+1, \dots, 2^k\}$, we pay out at most $\frac{n}{2^k}2^k = n$ dollars. There are $\log^* n$ intervals, so we pay out at most $n \log^* n$ dollars.

- Look at a find operation find(x):
  - For each node y along the path either:
    1. rank(y) and rank$(\pi(y))$ are in the same interval.
    2. rank(y) and rank$(\pi(y))$ are in different intervals. (rank$(\pi(y))$ is in a higher interval).
    * There are at most $\log^* n$ nodes of type 2.
    * Each node of type 1 we'll pay a dollar for the computation step. Need that each node has enough money to make these payments. Each time a node makes a payment, it gets a new parent with higher rank than the old parent. If y's rank is in the interval $\{k+1, \dots, 2^k\}$, it has to pay at most 2^k dollars before its parent has rank in a higher interval.
    * For each find call, step of type 2 happends at most $\log^* n$ times. It is less than $n \log^* n$.
    * Across all find calls, type 1 happens at most $n \log^* n$ times because we allocated $n \log^* n$ dollars and the nodes are able to pay a dollar for each type 1 step. It is less than $n \log^* n$.

2.8.6 Prim's Algorithm

Recall that Kruskal's algorithm a forest that is a subset of a MST, while Prim's algorithm grows a tree that is a subset of a MST. Repeatedly add lightest edge going out of the tree.

- Implemented using a priority queue.

Algorithm 24: Prim's Algorithm

```
Function primMST(V, E):
    Input: connected, undirected graph G = (V, E) with edge weights w_e
    Output: a MST defined by array prev
    for all u ∈ V do
        cost(u) = ∞;
        prev(u) = nil;
    end
    Choose an initial node u_0;
    cost(u) = 0;
    H = makequeue(V) ; // using cost values as keys, O(|v|)
    while H is not empty do // O(|v| log |v|)
        v = delete(H) ; // extracts the vertex in H with lowest cost, O(log |v|)
        for each {v, z} ∈ E do
            if cost(z) > w(v, z) + cost(v) then
                cost(z) = w(v, z) + cost(v) ; // H is updated with the new cost for z
                prev(z) = v;
            end
        end
    end
    return prev;
end
```

2.8.7 Boruvka's Algorithm for MST

- Brauvka phase
  - For each vertex v, mark the lightest edge touching v.
  - Determine the connected components formed by the marked edges.
  - Contract each component to a single vertex, keeping only lightest edges between components.

Let G' be the graph obtained after the Boruvka phase.

Proposition 2.7. If G has n vertices, then G' has at most $n/2$ vertices.

Proposition 2.8. The marked edges are part of an MST (follows from cut property).

Can be make faster by adding randomization.

Definition 2.1. Let F be a forest in graph G and let u, v be two vertices.

1. If u, v are in the same tree of F, there is a unique path $P(u, v)$ from u to v in F. Let $w_F(u, v)$ be the max weight of an edge on $P(u, v)$. (If u, v are on the same tree, $w_F(u, v) = \infty$.)
2. We say that (u, v) is F-heavy if $w(u, v) > w_F(u, v)$.
3. We say that (u, v) is F-light if $w(u, v) \leq w_F(u, v)$.

Lemma 2.2. Let F be any forest in G. If (u, v) is F-heavy, then (u, v) is not in any MST for G.

Theorem 2.3. Given a graph G and a forest F, all F-heavy edges can be identified in $O(n + m)$ time. (MST verification algorithm)

Lemma 2.3. Let G be a graph and $p \in (0, 1)$ be probability. Obtain a subgraph G' of G by keeping each edge with probability p. Let F be a minimum spanning forest in G' Then the expected number of F-light edges in G is at most n/p.

2.8.8 Randomized-MST(G)

Algorithm 26: Randomized-MST

```
Function Randomized-MST(G):
    Input: G = (V, E) and |V| = n, |E| = m
    Output:
    Use three Boruvka phases to compute a graph G_1, with at most n/8 edges. Let C be the set of
      edges marked during the three phases. If G_1 has only one vertex, return C;
    Randomly select a subgraph G_2 of G_1 by including each edge with probility 1/2;
    Call Randomized-MST(G_2) to obtain a minimum spanning forest F_2 for G_2;
    Identify the F_2-heavy edges in G_1, and delete them to obtain a new graph G_3;
    Call Randomized-MST(G_3) to obtain a minimum spanning forest F_3 for G_3;
    return the forest F = C ∪ F_3;
end
```

Proposition 2.9. Expect run time is $O(n + m)$.

- G: n vertices, m edges,
- G_1: $\leq n/8$ vertices, $\leq m$ edges, $O(n + m)$
- G_2: $\leq n/8$ vertices, $\approx m/2$ edges, $T(n/8, m/2)$
- Identifying F_2-heavy edges in G_1, $O(n + m)$
- G_3: $\leq n/8$ vertices, $\approx m/4$ edges, $T(n/8, n/4)$.

Proposition 2.10.

$$T(n, m) \leq T(m/8, n/2) + T(n/8, n/4) + c(n + m) \leq 2c(n + m).$$

2.9 Computational Complexity

- P vs. NP problem is the biggest open problem in theoretical computer science.
- NP-complete: presumably intractable problems (suspected to require exponential time).
- P (determistic polynomial-time): problems for which solutions can be found by a polynomial-time algorithm.
- NP (nondetermistic polynomial-time): problems for which solutoins can be verified (to be correct or not) by a polynomial-time algorithm.
- Polynomial-time algorithm: $O(n^c)$ time for some constant c.
- NP-complete: "hardest" problems in NP. If some NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time.

Example 2.16. Composites problem:

- Instance: a number n.
- Question: is n a composite number? (if so, produce two factors)
- Solution: a pair of numbers $k, l > 1$ such that $n = k \cdot l$.
- Verification algorithm: input: number n, candidate solution (k, l): multiply $k \cdot l$, and check if the answer equals n:
  - if it does, output yes.
  - if it doesn't, output no.
- Suppose n has 1000 bits, finding two 500-bit factors would take $O(2^{500})$ time. But given two 500-bit numbers, can multiply together efficiently and check. Finding the factors is the hard part.

P	NP
Shortest path	longest simple path
2-SAT	3-SAT
Eulerian cycle	Hamiltonian cycle

Example 2.17. • Eulerian cycle: find a cycle that use each edge once.

- Hamiltonian cycle: find a cycle that visits each vertex once.
- 3-SAT - Canonical NP-complete problem.
  - instance: 3CNF formula (CNF: conjunctive normal form)
  - Example: $\phi = \underbrace{(x_1 \vee \overline{x}_2 \vee x_4)}_{\text{clause consists of 3 literals}} \wedge (x_5 \vee \overline{x}_1 \vee x_2) \wedge (x_4 \vee x_5 \vee x_3)$, where $x_1, \ldots, x_n$ are Boolean variables, literal: variable x_i or negation $\overline{x}_i$
  - Question: is ϕ satisfiable? That is to say, is there a way to assign T/F values to $x_1, \ldots, x_n$ so the formula evaluates to true?
- 2-SAT - P:
  - instance: 2 CNF formula ϕ.
  - Example: clauses of size 2 instead of size 3: $\phi = (x_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_5) \wedge (x_4 \vee x_3)$.
  - Solvable in polynomial-time: reduce (convert) into a graph problem then do some graph reachability test.

NP-problems:

- 3-SAT
- CLIQUE: Given a graph G and $k \geq 1$, is there a fully connected subset of vertices of size k?
- TSP
- VERTEX-COVER: Given a graph G and $k \geq 1$, is there a set of vertices of size k that touch every edge?
- HAM-CYCLE
- SUBSET-SUM: Given a list of numbers $L = (a_1, a_2, \ldots, a_n)$ and a target number t, is there a sublist of L that sums to t?
- KNAPSACK

Definition 2.2. A problem is NP-complete if every problem in NP is reducible to it in polynomial time.

Definition 2.3. A is polynomial-time reducible to B if there is a polynomial-time computable function f such that for all instances I of A,

$$\begin{aligned} I \in A &\implies f(I) \in B, \\ I \notin A &\implies f(I) \notin B, \end{aligned}$$

where $I \in A$ means I is a positive instance of A (yes answer), and $I \notin A$ means I is a negative instance of A (no answer).

Proposition 2.11. If A is polynomai-time reducible to B, and $B \in P$, then $A \in P$.

Proof. Given instance I of A, compute $f(I)$ and use the algorithm for B to solve $f(I)$. Output this answer as the answer for I. □

Proposition 2.11 implies easiness translates downward over reductions.

Proposition 2.12. If A is polyonmial-time reducible to B, and $A \notin P$, then $B \notin P$.

Proof. Counterpositive of Proposition 2.11. □

Proposition 2.12 implies hardness translates upward over reductions.

Proposition 2.13. If A reduces to C and C reduces to B, then A reduces to B.

Proof. Compose the two reduction. □

Theorem 2.4. 3-SAT polynomial-time reduces to CLIQUE.

Proof. Let ϕ be a 3CNF formula with m classes $c_1, \ldots, c_n$ and n variables $x_1, \ldots, x_n$. We will construct an m-partite graph with m triples of 3 vertices. For each class clause, there is a triple of vertices labeled by the classes's literals. Connect two vertices if and only if:

- they are in different triples.
- they have compatible labels (don't connect x_i to $\overline{x}_i$).

□

Example 2.18. Let $\phi = \underbrace{(x_1 \vee \overline{x}_2 \vee x_3)}_{c_1} \wedge \underbrace{(\overline{x}_1 \vee x_2 \vee x_3)}_{c_2} \wedge \underbrace{(\overline{x}_1 \vee x_2 \vee \overline{x}_3)}_{c_3}$.

Proposition 2.14. ϕ is satisfiable if and only if G has a clique of size m.

2.9.1 Subset-Sum Problem

Given a collection of numbers $x_1, \ldots, x_k$ and a target number t, is there a subcollection that sumes to t? More precisely, is there a subset $I \subset \{1, \ldots, k\}$ such that $\sum_{i \in I} x_i = t$?

Theorem 2.5. Subset-Sum is NP-complete.

Proof. We will show 3-SAT polynomial-time reduces to Subset-Sum. Given a formula ϕ with variables $x_1, \ldots, x_l$, and clauses $c_1, \ldots, c_k$, we define numbers $y_1, \ldots, y_l, z_1, \ldots, z_l, g_1, \ldots, g_k, h_1, \ldots, h_k$ ($2l + 2k$ numbers) as follows. Each number has $k + l$ digits. □

Proposition 2.15. ϕ is satisfiable if and only if and list has a sublist that sublist that sums to t.

Example 2.19. $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3), l = 3 = k$, we have 12 numbers with 6 digits.

2.10 Set Cover

Given a universe U of n elements, a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of subsets of U, and a cost function $c : \mathcal{S} \to \mathbb{Q}_+$, find a minimum cost subcollection of $\mathcal{S}$ that covers U. In other words, find $I \subset \{1, \ldots, k\}$ such that $U \subset \bigcup_{i \in I} S_i$ and $\sum_{i \in I} c(S_i)$ is minimized. Note: Set Cover is NP-complete.

2.10.1 Greedy Approximation Algorithm

- Idea: iteratives pick the most cost-effective set and remove the covered elsements, until all elements are covered. Let C be the set of elements already covered at the beginning of an iteration. The **cost-effectiveness** of a set S is the average cost at which it covers new elments:

$$\frac{c(S)}{|S - C|}.$$

Algorithm 27:

Function ():
  Input:
  Output:
  $C = \emptyset$;
  while $C \neq U$ **do**
    Find a set S whose cost-effectiveness is smallest (S minimizes $\frac{c(S)}{|S-C|}$);
    Let $\alpha = \frac{c(S)}{|S-C|}$;
    Add S to the collection and for each $e \in S - C$, set price$(e) = \alpha$;
    $C = C \cup S$;
  end
  Output the collection of selected sets;
end

Number the elements of U as $e_1, e_2, \ldots, e_n$ in the order they are covered by the algorithm, resolving ties arbitrarily.

Lemma 2.4. For each $k \in \{1, \ldots, n\}$,

$$\text{price}(e_k) \leq \frac{OPT}{n - k + 1},$$

where OPT is the cost of an optimal solution.

Proof. In any iteration, the remaining elements can be covered at a cost of at most OPT (use the sets in the optimal solution that we haven't selected). Therefore, there must be a set with cost-effectiveness at most $OPT/|\overline{C}|$ (averaging argument). In the iteration where e_k is covered, $|\overline{C}| \geq n-k+1$ elements. Since e_k is covered by the most cost-effective set in the iteration,

$$\text{price}(e_k) \leq \frac{OPT}{|\overline{C}|} \leq \frac{OPT}{n-k+1}.$$

□

Theorem 2.6. This is an H_n-approximation algorithm, where $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$.

Proof. The total cost of the set cover is

$$\sum_{k=1}^{n} \text{price}(e_k) \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1} = OPT \sum_{k=1}^{n} \frac{1}{n-k+1} = OPT \cdot H_n.$$

□

Corollary 2.3. This is an $O(\log n)$ - approximation algorithm. H_n is tight for this algorithm, n elements $x_1, x_2, \ldots x_n$, set $S_1, \ldots, S_n, S$, $S_i = \{x_i\}$, $\text{cost}(x_i) = \frac{1}{i}$, $S = \{x_1, \ldots, x_n\}$, $\text{cost}(S) = 1 + \varepsilon$.

For Greedy Approximation Algorithm: it picks $S_n, S_{n-1}, \ldots, S_2, S_1$, cost H_n, and the performance ratio is $H_n/(1+\varepsilon)$.

2.11 Approximating TSP with Triangle Inequality

Let G be a complete graph on n vertices. For each pair u, v of vertices there is a $\text{cost}(u, v)$. Triangle inquality: for all u, v, w, we have

$$\text{cost}(u, w) \leq \text{cost}(u, v) + \text{cost}(v, w).$$

Goal: find a minimum cost tour.

Algorithm 28: Approximating TSP with Triangle Inequality

Function ():
  Input:
  Output:
  Find an minimum spanning tree T of G;
  Double every edge of T to obtain an Eulerian graph ; // every vertex has even degree
  Find an Eulerian cycle $\mathcal{T}$ of this graph;
  Let C be the tour that follows $\mathcal{T}$ visiting vertices in order of first appearance in $\mathcal{T}$; // taking shortcuts to not repeat vertices
  Output C;
end

Theorem 2.7. This is a 2-approximation algorithm.

Proof.

- $\text{cost}(T) \leq OPT$.
- $\text{cost}(\mathcal{T}) = 2\,\text{cost}(T)$.
- $\text{cost}(C) \leq \text{cost}(\mathcal{T})$.

Therefore, $\text{cost}(C) \leq \text{cost}(\mathcal{T}) = 2\,\text{cost}(T) \leq 2 \cdot OPT$. □

Algorithm 29:

Function ():
  Input:
  Output:
  Find an minimum spanning tree T of G;
  Compute a minimum-cost perfect matching M on the set of odd-degree vertices of T;
  Add M to T, obtaining an Eulerian graph;
  Compute an Eulerian cycle $\mathcal{T}$;
  Let C be the shortcut tour of $\mathcal{T}$;
  Output C;
end

Theorem 2.8. This is a 2-approximation algorithm.

Proof.

- $\text{cost}(T) \leq OPT$.
- $\text{cost}(M) \leq \frac{OPT}{2}$.
- $\text{cost}(\mathcal{T}) = \text{cost}(T) + \text{cost}(M)$.
- $\text{cost}(C) \leq \text{cost}(\mathcal{T})$.

Therefore, $\text{cost}(C) \leq \text{cost}(\mathcal{T}) = \text{cost}(T) + \text{cost}(M) \leq \frac{3}{2} \cdot OPT$. □

2.12 Computational Complexity

- $p \implies q$ is logically equivalent to $\neg p \vee q$.
- $p \implies q$ is logically equivalent to $\neg q \implies \neg p$.

Proposition 2.16. ϕ is unsatifiable if and only if there is a variable x such that there is path from x to $\neg x$.

2.12.1 Savitch's Algorithm

The graph recheability problem is

$$GR = \{\langle G, u, v \rangle : G \text{ is a directed graph and there is a path from } u \text{ to } v \text{ in } G\}.$$

- Instance: Graph G, vertices u, v.
- Question: Is there a path from u to v.
- BFS, DFS: linear time, linear space.
- Savithc's algorithm: sublinear space $O(\log^2 n)$.