Study Guide

Paul Jones

Design and Analysis of Computer Algorithms

Professor Kostas Bekris

Rutgers University

May 13, 2014

Contents

1 Midterm 1 Study Guide **5**

1.1 Introduction to Concepts of Algorithmic Design, Computing Running Times 5

1.1.1 Practice questions 5

1.2 Number Theoretic Algorithms Complexity of adding and multiplying 2 n-bit numbers, modulo arithmetic 6

1.2.1 Chapter 31.2 from CLRS2: Greatest common divisor 6

1.2.2 Practice questions 6

1.3 RSA Cryptosystem, Fermat's Little Theorem and Modular Exponentiation 8

1.3.1 Practice questions 8

1.4 Greatest Common Divisor algorithms: Euclid's test, Modulo Multiplicative Inverse 10

1.4.1 Practice questions 10

1.5 Primality Testing and Universal Hashing 11

1.5.1 Reading material 11

1.5.2 Practice questions 12

1.6 Divide and Conquer Algorithms and Recurrence Functions, Mergesort 13

1.6.1 Practice questions 13

1.7 Quicksort, Lower bounds for comparison-based sorting 15

1.7.1 Practice questions 15

1.8 Computing Medians and Other Order Statistics, Linear-time Sorting Solutions 15

1.8.1 Practice questions 15

1.9 Greedy Algorithms: Huffman encoding 17

1.9.1 Reading material 17

1.9.2 Practice questions 18

2 Midterm 2 Study Guide **19**

2.1 Greedy Algorithms: Horn formulas, Set Cover 19

2.1.1 Chapter 5.2 from DPV: Huffman encoding 19

2.1.2 Chapter 5.3 from DPV: Horn formulas 20

2.1.3 Chapter 5.4 from DPV: Set cover 21

2.1.4 Chapter 16.2 from CLRS2: Elements of the greedy strategy 21

2.1.5 Chapter 16.3 from CLRS2: Huffman codes 22

2.1.6 Practice questions 22

2.2 Elements of Dynamic Programming, Matrices, Subsequences 23

2.2.1 Chapter 6.2 from DPV: Longest increasing subsequences 23

2.2.2 Chapter 6.3 from DPV: Edit distance 24

2.2.3 Chapter 15.2 from CLRS2: Matrix-chain multiplication . . . 24
2.2.4 Chapter 15.3 from CLRS2: Elements of Dynamic Programming . . . 25
2.2.5 Practice questions . . . 25
2.3 Elements of Dynamic Programming, Knapsack problem, Graph Representation . . . 27
2.3.1 Chapter 6.4 from DPV: Knapsack . . . 27
2.3.2 Chapter 3.1 from DPV: Why graphs? . . . 28
2.3.3 Chapter 22.1 from CLRS2: Representations of Graphs . . . 28
2.3.4 Practice questions . . . 28
2.4 Depth-First Search . . . 29
2.4.1 Chapter 3.2 from DPV . . . 29
2.4.2 Chapter 22.5 from CLRS2 . . . 31
2.4.3 Practice questions . . . 31
2.5 Breadth-First Search, Properties of Shortest Paths . . . 33
2.5.1 Chapter 4.1 from DPV: Distances . . . 33
2.5.2 Chapter 4.2 from DPV: Breadth-first search . . . 33
2.5.3 Chapter 22.2 from CLRS2: Breadth-first search . . . 33
2.5.4 Practice questions . . . 33
2.6 Dijkstra's algorithm . . . 35
2.6.1 Chapter 4.3 from DPV: Lengths on edges . . . 35
2.6.2 Chapter 4.4 from DPV: Dijkstra . . . 35
2.6.3 Chapter 4.5 from DPV: Priority queue implementations . . . 35
2.6.4 Chapter 24.3 from CLRS2: Dijkstra's algorithm . . . 35
2.6.5 Practice questions . . . 36
2.7 Bellman-Ford and Floyd-Warshall algorithms . . . 37
2.7.1 Chapter 4.6 from DPV: Shortest paths in the presence of negative edges . . . 37
2.7.2 Chapter 4.7 from DPV: Shortest paths in dags . . . 37
2.7.3 Chapter 6.1 from DPV: Shortest paths in dags, revisited . . . 38
2.7.4 Chapter 24.1 from CLRS2: The Bellman-Ford Algorithm . . . 38
2.7.5 Chapter 24.2 from CLRS2: Single-source shortest paths in directed acyclic graphs . . . 39
2.7.6 Practice questions . . . 39
2.8 Floyd-Warshall and Johnson's algorithms . . . 40
2.8.1 Chapter 25.2 from CLRS2: The Floyd-Warshall algorithm . . . 40
2.8.2 Chapter 25.3 from CLRS2: Johnson's algorithm for sparse graphs . . . 41
2.8.3 Chapter 6.6 from DPV2 . . . 41
2.8.4 Practice questions . . . 41

3 Final Study Guide **43**

3.1 Minimum Spanning Trees . . . 43

3.1.1 Chapter 5.1 from DPV: Minimum spanning trees . . . 43

3.1.2 Chapters 23.1 from CLRS2: Growing a minimum spanning tree . . . 44

3.1.3 Practice questions . . . 45

3.2 Kruskal's and Prim's algorithms . . . 46

3.2.1 Chapters 23.2 from CLRS2: The algorithms of Kruskal and Prim . . . 46

3.2.2 Chapter 21.2 from CLRS2: Linked-list representation of disjoint sets . . . 48

3.2.3 Chapter 21.3 from CLRS2: Disjoint-set forests . . . 48

3.2.4 Practice questions . . . 49

3.3 The class of NP problems - Examples of NP complete problems . . . 51

3.3.1 Chapter 8.1 from DPV: Search Problems . . . 51

3.3.2 Chapters 34.2 from DPV: Polynomial-time verification . . . 53

3.3.3 Practice questions . . . 53

3.4 Additional examples of NP complete problems - Reductions . . . 56

3.4.1 Chapter 8.2 from DPV: NP-complete problems . . . 56

3.4.2 Practice questions . . . 58

3.5 Examples of reductions between NP complete problems . . . 60

3.5.1 Practice questions . . . 60

3.6 Intelligent Exhaustive Search, Intro to Approximation Algorithms . . . 61

3.6.1 Practice questions . . . 61

3.7 Approximation Algorithms, Local Search Heuristics . . . 62

3.7.1 Chapters 9.2 from DPV: Approximation algorithms . . . 62

3.7.2 Chapter 9.3 from DPV: Local search heuristics . . . 64

3.7.3 Practice questions . . . 64

1 Midterm 1 Study Guide

1.1 Introduction to Concepts of Algorithmic Design, Computing Running Times

1.1.1 Practice questions

1. **You may be provided with an algorithm for computing the nth Fibonacci number and then be asked to compute its running time (think in terms of it complexity). Alternatively, you may be asked to provide an efficient algorithm for computing Fibonacci numbers.**

```
function fib1(n)
if n = 0: return 0
if n = 1: return 1
return fib1(n - 1) + fib1(n - 2)
```

```
(n) function fib2
if n=0 return 0
create an array f[0 . . . n] f[0] =0,f[1] =1
for i = 2 . . . n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

2. **Give a justification why improvement in hardware are typically not sufficient to make an algorithm with exponential running time reasonably tractable.**

   But technology is rapidly improving—computer speeds have been doubling roughly every 18 months, a phenomenon sometimes called Moore's law. With this extraordinary growth, perhaps fib1 will run a lot faster on next year's machines. Let's see—the running time of fib1(n) is proportional to $2^{0.694n} \approx (1.6)^n$, so it takes 1.6 times longer to compute $F_{n+1} than F_n$. And under Moore's law, computers get roughly 1.6 times faster each year. So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}. And the year after, F_{102}. And so on: just one more Fibonacci number every year! Such is the curse of exponential time.

3. **We have 3 algorithms for the same problem where the first runs in exponential time, the second in logarithmic, and the thir in linear time as a function of the same input. Which algorith do you prefer to use?**

Name	Running time
Exponential	$O(a^n)$
Logarithmic	$(\log(n))$
Linear	$O(n)$

4. **You may be provided running times of different algorithms and asked to show which running time dominates asymptotically.**

5. **Provide the definition of *O*-notation (or Theta or Omega).**

Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say f = O(g) (which means that "f grows no faster than g") if there is a constant $c > 0$ such that $f(n) \leq c \times g(n)$.

- Just as $O(\cdot)$ is an analog of $\leq$, you can define the other analyses as such:
  - $f = \Omega(g)$ means $g = O(f)$.
  - $f = \Theta(g)$ means $f = O(g)$ and $f = \Omega(g)$.

6. **You may be provided certain statements about asymptotic analysis of running times and asked to show if they are correct or not. For instance, "n to the a dominates n to the b if a is greater than b.**

1.2 Number Theoretic Algorithms Complexity of adding and multiplying 2 n-bit numbers, modulo arithmetic

1.2.1 Chapter 31.2 from CLRS2: Greatest common divisor

- In this section, Euclid's algorithm for computing the GCD of two integers efficiently.
  - Suprising connection with Fibonacci numbers.
    - This yields the worst case.
- Restricted to nonegative integers.

$$\gcd(a, b) = \gcd(|a|, |b|)$$

- One way of characterizing the problem:

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$
$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r}$$
$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)}$$

- Proof
  - We first show that the GCD of a and b divide each other.
    - If $d = \gcd(a, b)$, then $d|a$ and $d|b$.
  - Because a mod b is a linear combination of a and b, $d|(a \bmod b)$.
  - Because $d|b$ and $d|(a \bmod b)$, $d|\gcd(b, a \bmod b)$
  - Equivilently,

$$\gcd(a, b)|\gcd(b, a \bmod b)$$

  - You can show that the reversed is true to, and because if you can invert the divisible operator, the two things are plus or minus equal

1.2.2 Practice questions

1. **What is the "primality" problem and what is the "factoring" problem? Which one of the two is tractable? What are the advantages of the intractability of the other problem?**

Factoring problem Given a number N, express it as a product of prime factors.

Computationally intractable, fastest is exponential. This makes RSA work because keys and factoring large multiples.

Primality problem Given a number N, determine whether it is a prime.

Computationally tractable.

2. **How many digits do you need to represent a number N in base b?**

   - With k digits in base b we can express numbers up to $b^k - 1$.
     - For instance, in decimal, three digits get us all the way up to $999 = 10^3 - 1$.
   - By solving for k, we find that $\lceil \log_b(N+1) \rceil$ are need to write N in base b.

3. **How much does the size of the representation of a number changes when we change bases?**

   - Recall the rule for converting logarithms from base a to base b:

     $$\log_b N = (log_a N)/(log_a b)$$

     - So the size of integer N in base a is the same as its size in base b, times a constant factor $log_a b$.
     - In big-O, the base is irrelevant, and we write the size simply as

       $$O(\log N)$$

4. **What is the running time of the addition operation of a number for two n-bit numbers (think in terms of bit complexity)?**

   - Suppose x and y are n bits long.
   - The sume of x and y is *at most* $n+1$ bits long.
   - Each bit of the sume is computed in a fixed amount of time.
   - The total running time for addition will be

     $$c_0 + c_1 n$$

     In other words, *linear*.

     $$O(n)$$

5. **What is the running time of the tradition multiplication operation taught in grade school for two n-bit numbers (think in terms of n-bit complexit)?**

   - To compute each row, either "X" or "0", left-shifted
   - The rows are in the order of $2n$.
   - You have to sum them up, and you do this pairwise.
   - If you do n times an operation which costs n, your running time is going to be n^2.

6. **Provide a recursive formula for the multiplication of two numbers.**

```
function multiplication(x, y) {
    if (y == 1) {
        return x;
    }

    else {
        return x + multiplication(x, y - 1);
    }
}
```

7. **What is the complexity of modular addition and modular multiplication for two n bit numbers?**

   - **Addition**: $O(n)$
   - **Multiplication**: $O(n^2)$

1.3 RSA Cryptosystem, Fermat's Little Theorem and Modular Exponentiation

1.3.1 Practice questions

1. **Give a private key protocol for a cryptography application. Given an example it can be compromised.**

   - There is the classic "codebook" private-key scheme, where Alice and Bob meet before hand and agree on a secret code.
   - The way that this is compromised is if Eve gets the codebook or knows a given message and backtraces the code from it.
   - Alternatively, the answer you're looking for might be that you can encode and decode with public keys like:

     $$x = d(e(x))$$

     Where x is a message, $d(\cdot)$ is a decoder, and $e(\cdot)$ is an encoder. Bob can
   - "Network sniffer"

2. **RSA is based on which basic property of modulo arithmetic?**

   - The basic feature of modulo arithmetic that RSA exploits is the dramatic contrast in the tractibility of factoring and primality testing.
     - Bob and Alice only need to make simple calculations.
     - Eve needs to make computations that would be struggle for the world's largest computers.
   - Pick any two prime p and q and let $N = pq$. For any e relatively prime to $(p-1)(q-1)$:

     $$(x^e)^d \equiv x \mod N$$

3. **Describe the steps of the RSA protocol. The security of the protocol is based on which assumption?**

   Given N, e, and $y = x^e \mod N$, it is computationally intractable to determine x.

4. **What are the basic operations that need to be performed accords to the RSA protocol and what is their running time?**

   - Pick two large primes
   - Multiply two large primes
   - Extended Euclid algorithm
   - Efficient modular exponentiation algorithm
   - $y^d \mod N$

5. **You may be provided an example message and asked to described the operations of the RSA protocol on it.**

6. **What does Fermat's Little Theorem specify? Prove it.**

   - **Fermat's Little Theorem**: If p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p.
   - **Proof**
     - Let us assume that a is positive and not divisible by p. The idea is that if we write down the sequence of numbers

       $$a2a, 3a, \cdots, (p-1)a$$

       and reduce each one modulo p, the result sequences turns out to be an arrangment of

       $$1, 2, 3, \cdots, p-1$$

       Therefore, if we multiple together the numbers in each sequences, the results must be indetifical modulo p:

       $$a \times 2a \times 3a \times \cdots \times (p-1)$$

       Which is equivilent to

       $$1 \times 2 \times 3 \times \cdots \times (p-1)$$

       Collecting the a terms yields

       $$a^{p-1}(p-1)! \equiv (p-1)!(\mod p)$$

       Finally, we may "cancel out" the numbers $1, 2, \cdots, p-1$ from both sides, obtains,

       $$a^{p-1} \equiv 1(\mod p)$$

7. **What is the importance of Fermat's little theorem in the RSA protocol?**

8. **How can we efficiently perform modular exponentiantion? What is the running time of the approach?**

   - Recursion, repeated squaring.
   - Let n be the size of bits x, y, and N (whichever is largest).
   - Like multiplication, there are *at most* n recursive calls.
   - During each call, it multiples n-bit numbers.
   - Doing computation modulo N saves us here.
   - Running time of $O(n^3)$

1.4 Greatest Common Divisor algorithms: Euclid's test, Modulo Multiplicative Inverse

1.4.1 Practice questions

1. **What does Euclid's rule specify? Prove it.**

   - **Euclid's rule**: If x and y are positive integers where $x \leq y$, then $\gcd(x, y) = \gcd(x \bmod y, y)$
   - **Proof**
     - Suppose $a, b \in \mathbb{Z}$ and $a \vee B \neq 0$
     - From the *Division Theorem,a = qb + r$ where $0 \leq r < |b|$.
     - From *GCD with Remainder*, the GCD of a and b is also the GCD of b and r.
     - Therefore, we may search instead for $\gcd(b, r)$.
     - Since $|r| < |b|$ and $b \in \mathbb{Z}$, we will reach $r = 0$ after finitely many steps.
     - At this point, $\gcd(r, 0) = r$ from *GCD with Zero*.
   - **Proof *from the book***:
     - It is enough to show the slightly simpler rule:

       $$\gcd(x, y) = \gcd(x - y, y)$$

       from which the one stated can be derived by repeatedly subtracting y from x.
     - Any integer that divides both x and y must also divide $x - y$, so

       $$\gcd(x, y) \leq \gcd(x - y, y)$$

     - Likewise, any integer that divides both $x - y$ and y must also divide both x and y, so

       $$\gcd(x, y) \geq gcd(x - y, y)$$

2. **What is Euclid's algorithm for finding the greatest common divisor? What is its running time and why?** ~ function euclid(a, b) { if (b == 0) { return a; }

```
else {
    return euclid(b, a % b);
}
```

   } ~

   - It is, "If d divides both a and b, and $d = ax + by$ for some integers x and y, then necessarily $d = \gcd(a, b)$."
   - **Run time**
     - Let $T(a, b)$ be the number of steps taken in the Euclidean algorithm.
     - Let $h = \log_{10} b$ be the number of digits in b.
     - Assume that the modulo function is $O(1)$.
     - The worst case is consecutive Fibonacci numbers.

       $$a = F_{n+1}, b = F_n$$

     - The algorith will calcuate the following until $n = 0$,

       $$\gcd(F_{n+1}, F_n) = \gcd(F_n, F_{n-1})$$

So,

$$T(F_{n+1}, F_n) = \Theta(n)$$
$$T(a, F_n) = O(n)$$

- Since $F_n = \Omega(\Phi^n)$, this implies that

$$T(a, b) = O(\log_\Phi b)$$

- Note that $h \approx log_{10} b$ and $log_b x = \frac{\log x}{\log b}$, this implies $\log_b x = O(\log x)$.
- So the worst case for Euclid's algorith is

$$O(\log_\Phi b) = O(h) = \log(b)$$

3. **Show that if d divides both a and b, and $d = a \times x + b \times y$ for some integers x and y, then $d = \gcd(a, b)$.**

   - By the first two condition, d is a common divisor of a and b so it cannot exceed the GCD, that is, $d \leq \gcd(a, b)$.
   - On the other hand, $\gcd(a, b)$ is a common divisor of a and b, it must also divide $ax + by = d$, which implies that $\gcd(a, b) \leq d$.
   - Therefore, $d = \gcd(a, b)$.

4. **What is the extended Euclid's algorithm for finding the greatest common divisor d of two numbers a and b, as well as numbers x and y, so that $d = a \times x + b \times y$? Prove its correctness (you will need to prove Euclid's algorithm first and then the extension).**

   EEA: If d divides both a and b, and $d = ax + by$ for some integers x and y, then $d = \gcd(a, b)$

   - *Proof*
     - By the first two condition, d is a common divisor of a and b so it cannot exceed the GCD, that is, $d \leq \gcd(a, b)$.
     - On the other hand, $\gcd(a, b)$ is a common divisor of a and b, it must also divide $ax + by = d$, which implies that $\gcd(a, b) \leq d$.
     - Therefore, $d = \gcd(a, b)$.

5. **When does the multiplicative inverse of a number x exists modulo N and why?**

6. **How can you compute the multiplicative inverse modulo N for two relative prime numbers? What is the running time of the corresponding algorithm?**

1.5 Primality Testing and Universal Hashing

1.5.1 Reading material

Chapter 1.3 from DPV: Primality testing

- Is there a test to tell us whether a number is prime? We place our home in a theorom from 1640.

Fermat's Little Theorem If p is prime, then $\forall 1 \leq a < p$,

$$a^{p-1} \equiv 1 (\mathrm{mod} p)$$

1.5.2 Practice questions

1. **Describe a test for evaluating whether a number is prime. What is the probability of this test returning the correct answer and why? Can you increase the probability of success for this test?**

   - Fermat's little theorem states that if p is prime and $1 \leq a < p$, then,

   $$a^{p-1} \equiv 1 (\mathrm{mod} p)$$

   - If we want to test if p is prime, then we can pick random a's in the interval and see if the equality holds.
     - If the equlaity *doesn't* hold for a value, then p is composite.
     - If the equality *does* hold for *many* values of a, the p is *probable prime*.
   - Using fast algorithms for modular exponentiation, the running time of this algorith is

   $$O(k \times \log^2 n \times \log\log n \times \log\log\log n)$$

   Where k is the number of time we test a random a, and n is the value we want to test for primality.
   - You increase the probability with a larger k.

- **What does Lagrange's Prime Number Theorem specify and why is it helpful for primality testing?**

  Lagrange's prime number theorem Let $\pi(x)$ be the number of prime $\leq x$. Then $\pi(x) \equiv x/(\ln x)$, or more precisely,

  $$\lim_{x \to \infty} \frac{\pi(x)}{(x/\ln x)} = 1$$

  - Such abundance makes it simple to generate a random n-bit prime:
    * Pcik a random n-bit number N.
    * Run a primality test on N.
    * If it passes the test, output N; else repeat the process.

1. **What properties should a good hash function provide?**

   - Wikipedia
     - Determinism, same input same output
     - Uniformity, map the expected inputs as evenly as possible.
     - Variable range, can be expanded or contracted in size.
     - Continuity, two similar inputs should be mapped to nearly equal hashes
   - CLRS2
     - A good hash function satisifes the assumption of simple uniform hasing: each key is equally likely to hash to any of the m slots, indepedantly of where any other key has hashed to.

2. **What is the property of universal hashing families of functions? Provide a universal hashing family of functions for an example problem and prove the corresponding property.**

- Universal hashing is designed to stop the effectiveness of a attacker who knows the hash function and wants to put every input in the same bin.
- To avoid this, at the beginning of the lifecycle of a hash table, a function is picked *at random.*

Property Consider any pair of distinct IP addresses $x = (x_1, \cdots, x_4)$ and $y = (y_1, \cdots, y_4)$. If the coefficients $a = (a_1, a_2, a_3, a_4)$ are chosen uniformly at random from $\{0, 1, \cdots, n-1$ then

$$Pr\{h_a(x_1, \cdots, x_2) = h_a(y_1, \cdots, y_4)\} = \frac{1}{n}$$

- *Proof*
  - Since $x = (x_1, \cdots x_4)$ and $y = (y_1, \cdots, y_4)$ are distinct, these quadruples must differ in some component, assume their not equal.
  - We wish to compute the probability $Pr[h_a(x_1, \cdots, x_4) = h_a(y_1 \cdots y_4)]$
  - That is, that
$$\sum_{i=1}^{4} a_i \times x_i \equiv \sum_{i=1}^{4} a_i \times y_i \bmod n$$
  - This last equation can be rewritten
$$\sum_{i=1}^{3} a_i \times (x_i - y_i) \equiv a_4 \times (y_4 - x_4) \bmod n$$
  - Suppose that we draw a random hash function h_a by picking $a = (a_1, a_2, a_3, a_4)$ at random.
  - We start by drawing a_1, a_2, a_3, and then pause to think: What is the probability that the last drawn number a_4 is such that the equation 2 bullet points back holds (call it (1))?
  - So far the left-hand side of equation (1) evaluates to some number, say c.
  - And since n is prime and $x_4 \neq y_4$, $(y_4 - x_4)$ has a unique inverse modulo n.
  - Thus for equation (1) to hold, the last number a_4 must be precisely $c \times (y_4 - x_4)^{-1} \bmod n$, out of its n possible values.
  - The probability of this happening is $\frac{1}{n}$, and the proof is complete.

1.6 Divide and Conquer Algorithms and Recurrence Functions, Mergesort

1.6.1 Practice questions

1. **What are the basic principles of divide-and-conquer algorithms?**

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

2. **Describe an approach for multiplication of two n-bit numbers that has a better running time than $O(n^2)$. Prove its running time.**

```
function multiply(x, y) {
    n = max(sizeof(x), sizeof(y));

    if (n == 1) {
        return xy;
```

```
    }

    x_l = leftmostBits(ceil(n / 2));
    x_r = rightmostBits(floor(n / 2));
    y_l = leftmost(ceil(n / 2));
    y_r = rightmostBits(floor(n / 2));

    p_1 = multiply(x_l, y_l);
    p_2 = multiplu(x_r, y_r);
    p_3 = multiply(x_l + x_r, y_l + y_r);

    return p_1 x 2^n + (p_3 - p_1 - p_2) x 2^(n / 2) + p_2;
}
```

- Run time
  - This running time can be derived by looking at the algorithm's pattern of recursive calls, which form a tree-structure.
  - At aech successive level of recursion the subproblems get halved in size. At the $log_2 n$th level, the subproblems get down to size 1, and so the recursion ends.
  - Therefore, the high of the tree is $\log_2 n$.
  - The branching factor is 4, each problem recursive produces three smaller ones - with the result that at depth k in the tree there are 3^k subproblems, each of size $n/2^k$.

3. **What does the Master theorem specify? Prove it. > Master theorem**: If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some > constants $a > 0, b > 1, d \geq 0$, then: >

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

- Proof
  1. Assume that n is a power of b, for convinience rounding.
  2. Notice that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $log_b n$ levels. This is the hight of the tree.
  3. Its branching factor is a, so the kth level of the tree is made up of a^k subproblems, each of size n/b^k. The total work is done:

$$a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k$$

  4. As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with the ration a/b^d. Finding the sum of such a series in big-O notation is easy and comes down to three cases:
     1. The ratio is less than 1.
     2. The ratio is greater than 1.
     3. The ratio is exactly 1.

4. **You may be provided a divide-and-conquer algorithm and asked to argue about its running time by using the Master theorem.**

5. **How does mergesort work, what is its running time and why? How does the iterative version of mergesort work?**

   Mergesort splits an unsorted array into two and sorts and concatenates the subarrays.

```
function interative-mergesort(a[1 .... n]) {
    Q = [] (empty queue);
    for i = 1 to n:
         inject(Q, a[i])l

    while(count(Q) > 1):
         inject(Q, merge(eject(Q), eject(Q));

    return eject(Q);
}
```

1.7 Quicksort, Lower bounds for comparison-based sorting

1.7.1 Practice questions

1. **What are comparison sorting algorithms? What is the lower limit for the running time of comparison sorting algorithms?**
2. **How does quick-sort work? When does the worst-case running time arise? When does the best-case running time arise?**

   1. Pick an element, called a pivot, from the list.
   2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
   3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

      The best case for the algorithm now occurs when all elements are equal. The worst case for the algorithm occurs when the elements are already sorted.

3. **Provide a rigorous proof for the worst-case performance of quick-sort.**
4. **Provide a rigorous proof for the expected running time of quick-sort.**

1.8 Computing Medians and Other Order Statistics, Linear-time Sorting Solutions

1.8.1 Practice questions

1. **How can we efficiently compute the median of a set of numbers? What is the running time of this solution?**
2. **What is the main idea behind counting sort? Provide a description of the algorithm and argue about its running time.**

the algorithm loops over the items, computing a histogram of the number of times each key occurs within the input collection. It then performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key, the starting position in the output array of the items having that key. Finally, it loops over the items again, moving each item into its sorted position in the output array.

```
# variables:
#    input -- the array of items to be sorted; key(x) returns the key for item x
#    n -- the length of the input
#    k -- a number such that all keys are in the range 0..k-1
#    count -- an array of numbers, with indexes 0..k-1, initially all zero
#    output -- an array of items, with indexes 0..n-1
#    x -- an individual input item, used within the algorithm
#    total, oldCount, i -- numbers used within the algorithm

# calculate the histogram of key frequencies:
for x in input:
    count[key(x)] += 1

# calculate the starting index for each key:
total = 0
for i in range(k):   # i = 0, 1, ... k-1
    oldCount = count[i]
    count[i] = total
    total += oldCount

# copy to output array, preserving order of inputs with equal keys:
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1

return output
```

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the Count array, and the second for loop which performs a prefix sum on the count array, each iterate at most k + 1 times and therefore take O(k) time. The other two for loops, and the initialization of the output array, each take O(n) time. Therefore the time for the whole algorithm is the sum of the times for these steps, O(n + k).

Because it uses arrays of length k + 1 and n, the total space usage of the algorithm is also O(n + k).[1] For problem instances in which the maximum key value is significantly smaller than the number of items, counting sort can be highly space-efficient, as the only storage it uses other than its input and output arrays is the Count array which uses space O(k).

3. **What is the main idea behind radix sort? Provide a description of the algorithm and argue about its running time.**

   Radix sort works by first sorting by the least significant bit, then the next most significant bit, all the way to the most significant bit.

4. **Prove the correctness of radix sort.**

For a value $r \leq b$, we view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range of 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. Each pass of counting sort takes time $\Omega(n + k) = \Omega(n + 2^r)$ and there are d passes. This makes a total running time of $\Omega(d(n + 2^r))$.

5. **What is the main idea behind bucket sort? Provide a description of the algorithm. Prove its running time.**

   - Idea

     1. Set up an array of initially empty "buckets".
     2. Scatter: Go over the original array, putting each object in its bucket.
     3. Sort each non-empty bucket.
     4. Gather: Visit the buckets in order and put all elements back into the original array.

     The idea behind bucket sort is that if we know the range of our elements to be sorted, we can set up buckets for each possible element, and just toss elements into their corresponding buckets. We then empty the buckets in order, and the result is a sorted list.

   - Psuedocode

```
function bucketSort(array, n) is
    buckets ← new array of n empty lists
    for i = 0 to (length(array)-1) do
        insert array[i] into buckets[msbits(array[i], k)]

    for i = 0 to n - 1 do
        nextSort(buckets[i]);

    return the concatenation of buckets[0], ...., buckets[n-1]
```

1.9 Greedy Algorithms: Huffman encoding

1.9.1 Reading material

Chapter 5.2 from DPV: Huffman encoding

- In general, how do we find the optimal coding tree, given the frequencies $f_1 \cdots f_n$ of n symbols? To make the problem precise, we want a tree whose leaves each correspond to a symbol and which minimizes the overall length of the encoding. **Cost of tree**, where d is the depth of the ith symbol in the tree:

$$\sum_{i=1}^{n} f_i \times d$$

- 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 visit during encoding or decoding.

  – The total cost can be expressed thusly:

    The cost of a tree is the sum of the frequences of all leaves and internal nodes, except the root.

```
function Huffman(f)
input: Any array f[1 ... n] of frequencies
output: An encoding tree with n leaves

let H be a priority queue of integers, ordered b 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)
```

Chapter 16.2 from CLRS2: Greedy algorithms

Greedy algorithm A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

- Greedy algorithms do not always yield optimal solution, but for many problems they do.
- The following steps apply to a greedy algorithm strategy:
  1. Determine the optimal substructure of the problem.
  2. Develop a recursive solution.
  3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choise. Thus, it always safe to make the greedy choice.
  4. Show that all but one of the subproblems induced by having made the greedy choice are empty.
  5. Develop a recursive algorithm that implements the greedy strategy.
  6. Convert the recursive algorithm to an iterative algorithm.

Wikipedia: Greedy algorithms

- A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.
  - For instance, to solve the travelling salesman problem with a greedy strategy, adopt the following strategy: "At each stage visit an unvisited city nearest to the current city."
- There are five components:
  1. A candidate set, from which a solution is created.
  2. A selection function, which chooses the best candate to be added to the solution.
  3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.
  4. An objective function, which assigns a value to a solution, or a partial solution, and
  5. A solution function, which will indicate when we have discovered a complete solution.

1.9.2 Practice questions

1. **What is the main principle behind greedy algorithms?**

- The main principle behind greedy algorithms is to always, in any given decision point, to select the locally optimal choice and hope that it leads to a globally optimal choice (which has no guarentee).

2. **Why does the greedy algorithm work for the coin changing problem given the US coin system?**

   - The US coin system placed in a set of integers has the property of being a **matroid**.

3. **What is the idea in Huffman encoding in order to achieve data compression? What is the prefix-free property?**

   - Huffman oding is an entropy encoding algorithm used for lossless data compression.
   - The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurance for each possible value of the source symbol.

4. **Provide the Huffman encoding algorithm and argue its running time.**

   1. Create a leaf node for each symbol and add it to the priority queue.
   2. While there is more than one node in the queue:
      1. Remove the two nodes of highest priority (lowest probability) from the queue.
      2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes probabilities.
      3. Add the new node to the queue.
   3. The remaining node is the root node and the tree is complete.

5. **Prove the greedy choice property for the Huffman encoding algorithm (first lemma).**

   Lemma 16.2 Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two character in C having the lowest frequencies. Then there exists an optimal prefix code C in which the codewords for x and y have the same length and differ only in the last bit.

6. **Prove the optimal substructure property for the Huffman encoding algorithm (second lemma).**

2 Midterm 2 Study Guide

2.1 Greedy Algorithms: Horn formulas, Set Cover

2.1.1 Chapter 5.2 from DPV: Huffman encoding

- Is there some sort of *variable-length encoding* in which just one bit is used for the most frequent symbol and the least amout of bits are used for the lesser freqeuent symbols?
- One danger of this is that resulting encodings might not be uniquely decipherable.
  - For instance, considering $\{0, 01, 11, 001\}$. Strings like 001 are ambigous.
  - We need the *prefix-free* property.

Prefix free No codeword can be a prefix of another codeword.

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

- The cost of a tree is the sums of the frequencies of all leaves and internal nodes, except the root.
- We can state constructing the tree *greedily*: find two symbols of smallest frequencies, i and j, and make them children of a new node, which has the frequency $f_i + d_j$.

```
procedure Huffman(f)
Input: an array of frequencies
Output: an encoding tree

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)
}
```

2.1.2 Chapter 5.3 from DPV: Horn formulas

- To display human intelligence, a computer must do some logical reasoning.
- Most primitive object in Horn formulas is the *Boolean variable*, which can be `true` or `false`. Ex,

$$x \equiv \text{the murder took place in the kitchen}$$

$$y \equiv \text{the butler is innocent}$$

$$z \equiv \text{the colonel was asleep at 8pm}$$

- A *literal* is either a variable x or its negation $\neg x$. In Horn formulas, knowledge about variables is represented in two kinds of *clauses*:

  1. *Implications*, whose left-hand is an "and" of any number of positive literals and whose right-hand side is a single positive literal. For instance,

     $$(z \wedge w) \rightarrow u$$

     "If the colonel was asleep at 8 pm and the murder took place at 8pm, then the colonel is innocent."
  2. Pure *negative cluases*, consisting of an "or" of any number of negative literals, as in,

     $$(\neg u \vee \neg v \neg y)$$

     "They can't all be innocent."

- Given a set of these two types, the goal is to determine whetheter there is a consistent explanation, an assignment of true/false values to the variables that satisfies all the cluases. *Satisfaction requirement.*

2.1.3 Chapter 5.4 from DPV: Set cover

- Given a set of towns and these two constraints: (1) each school should be in a town, and (2) no school should have to travel more than 30 miles to reach a school.
- This is a *set cover* problem.

  Set cover

  Input: A set of elements B; sets $S_1, \cdots, S_m \in B$

  Output: A selection of the S_i whose union is B.

  Cost: Number of sets picked

- There is a greedy solution:

```
While all elements B are uncovered:
    Pick the set $S_i$ with the largest number of uncovered elements
```

  Claim: Suppose B contains n elements and the algorithm consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.

- The ration between the greedy algorithm's solution and the optimal solution varies from iinput to input but is *way less* than $\ln n$

2.1.4 Chapter 16.2 from CLRS2: Elements of the greedy strategy

- A greedy algorithm obtains an optimal solution to a problem by making choices.
  - Every choice, the choice is the best *at the moment.*
  - This doesn't always work, but sometimes it does.
- The steps:
  1. Determine the optimal substructure of the problem.
  2. Develop a recursive solution.
  3. Prove that at any stage of the recusion, one of the optimal choices is the greedy choice. That is is always safe to make the greedy choice.
  4. Show that all but one of the subproblems induces by having made the greedy choice are empty.
  5. Develop a recursive algorithm that implements the greedy strategy.
  6. Convert the recrusive algorithm to an iterative algorithm.
- The steps, more generally:
  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is not always safe.
  3. Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that is we combine an optimal solution to the subprobelm with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Greedy-choice property

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

- How greedy algorithms differ from dynamic programming:
  - In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. It is "bottom", going from smaller subproblems.
  - In a greedy algorithm, we make a choice that is best at the moment and then solve the subproblem arising after choice is made.
    - Greedy algorithms are "top-down."

Optimal substructure

If an optimal solution to the porblem contains within it optimal solutions subproblems.

- This is key to greedy algorithms and dynamic programming.

2.1.5 Chapter 16.3 from CLRS2: Huffman codes

Prefix codes Codes in which no codeword is also a prefix of some other codeword.

- In the following psuedocode, assume C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$.
  - The algorithm building the tree T corresponding to the optimal code in bottom-up manner.

```
Huffman(C)
n <- |C|
Q <- C
for i <- to n - 1:
    do allocate a new node z
        left[z] <- x <- Extract-Min(Q)
        right[z] <- y <- Extract-Min(Q)
        f[z] <-  f[x] + f[y]
        Insert(Q, z)

return Extract-Min(Q)
```

2.1.6 Practice questions

1. **What is the main principle behind greedy algorithms?**

   The principle behind greedy algorithms is that at every point where a given algorithm can make a choice, instead of trying to figure out if there exists a "globally optimal" decision, a greedy algorithm just goes with the cheapest option in that moment and "hopes for the best."

2. **Why does the greedy algorithm work for the coin changing problem given the US coin system?**

   The reason that the US coin system allows the greedy choice to solvethe coin change problem is that a matroid can be formed from the set of US coins.

3. **What is the idea in Huffman encoding in order to achieve data compression?**

   The idea of Huffman encoding is that you take the characters which appear most often in your document and you make those the cheapest to encode. You subsequently take the least frequent characters in your document and make them only as expensive as the prefix-free property demands they be.

4. **What is the prefix-free property?**

   The prefix-free property is that for a given dictionary, no word in the dictionary has at its beginning any other word.

5. **Provide the Huffman encoding algorithm and argue its running time.**

   1. Create a leaf node for each symbol and add it to the priority queue.
   2. While there is more than one node in the queue:
      1. Remove the two nodes of lowest probability.
      2. Crate a new node with the sum of those two probabilities.
      3. Add that node to the queue.
   3. The remaining node is the root node and the tree is complete.

   Since efficient priority queue data structures require $O(logn)$ time per insertion, and a tree with n leaves has $2n-1$ nodes, this algoirhtm operates in $O(n \log n)$ time, where n is the number of symbols.

6. **Describe the greedy algorithm for logical reasoning with Horn formulas.**

```
Input: a horn formula
Output: a satisfying assignment, if one exists

set all variables to false
while there is an implication that is not satisfied:
    set the right-hand variable of the implication to true

if all pur negative clauses are satisfied: return the assignment
else: return ``formula is not satisfiable``
```

7. **What is the property of the greedy algorithm for the set cover example? Prove it.**

2.2 Elements of Dynamic Programming, Matrices, Subsequences

2.2.1 Chapter 6.2 from DPV: Longest increasing subsequences

- In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \cdots, a_n$.
  - A *subsequence* is any subset of these number taken in order, of the form $a_{i_1}, a_{i_2}, \cdots, a_{i_k}$ where $1 \geq i_1 < i_2 < ... < i_k \leq n$ and an *increaing* subsequence is one in which the numbers are getting strictly larger.
  - For instance, consider the following sequence:

5 2 8 6 3 6 9 7

    * The longest increasing subsequence is 2, 3, 6, 9.

- You can express this problem as a graph problem where the graph is a directored acyclic one, notice two properties:

  1. It's a DAG
  2. There is a one-to-one correspondance between increasing subsequences and paths in this dag.

- Just find the longest path in the dag!

```
for j = 1, 2, ..., n:
    L[j] = 1 + max{L(i) : (i, j) in E}
return max_j L(j)
```

- This is dynamic programming. We have defined a collection of subproblems with the following property that allows them to solved in a single pass:

  (*) There is an ordering on the subporblems, and a relation that shows how to solve a subproblem given the answer to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

- Each subproblem is solved using the relation:

$$L(j) = 1 + max\{L(i) : (i, j) \in E\}$$

2.2.2 Chapter 6.3 from DPV: Edit distance

- When a spell-checker encounters a possible misspelling, it looks in its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?
  - Perhaps how much they can be *aligned* or *matched-up*?

```
S - N O W Y   - S N O W - Y
S U N N - Y   S U N - - N Y
```

A dynamic programming solution

- Doing this, the most crucial question is: *What are the subproblems?*
  - As long as they have the propert (*), it is easy to write down the algorithm: iteratively solve one problem after the other in order of increasing size.
- Our goal is to find the distance between two strings. What is a good subproblem?
  - Find the prefix of the first and second strings.

2.2.3 Chapter 15.2 from CLRS2: Matrix-chain multiplication

- An example of *dynamic programming*.

```
MatrixMultiply(A, B) {
    if columns(A) != columns(B)
        then error "incompatible dimensions"
        else for i <- 1 to rows[A]
            do for j <- 1 to columns[B]
                do C[i, j] <- 0
                    for k <- 1 to clumns[A]
                        do C[i, j] <- C[i, j] + A[i, k] * B[k, j]
        return C
}
```

2.2.4 Chapter 15.3 from CLRS2: Elements of Dynamic Programming

- The elements of dynamic programming: optimal substructure and overlapping subproblems.

Optimal substructure If an optimal solution to the poroblem contains within it optimal solutions to subproblems.

- How to find it, some characteristic:
  1. You should that a solution to the problem consists of making choices, where making a choice leaves a subproblem to be solved.
  2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this, you just assume it's been given to you.
  3. Given this choice, you determine which subproblems ensue and how to best characterize the result space of subproblems.
  4. You show that the solutions to the subprobelsm used within the optimal solution to the problem must themselves be optimal. You do so by supposing otherwise and deriving a contradiction.
- Optimal substructure differes in two ways:
  1. How many subproblems are used in an optimal solution to the original problem
  2. How many choices we have in determineing which subporblems to use in an optimal solution.

Overlapping subproblems When a recursive algorithm revists the same problem over and over again, we say that the optimization problem has *overlapping subproblems*.

2.2.5 Practice questions

1. **How does the dynamic programming approach decompose the chain-matrix multiplication into subproblems and how are their solutions combined in order to solve a larger problem?**

   Suppose that an optimal parenthesization of $A_iA_{i+1}\cdots A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the "prefix" subchain $A_iA_{i+1}\cdots A_k$ whithin this optimal parenthesization of $A_iA_{i+1}\cdots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\cdots A_k$. Why? If there were a less costly way to parenthesize $A_iA_{i+1}\cdots A_k$ substituting that parenthesization in the optimal parenthesization of $A_iA_{i+1}\cdots A_j$ would produce another parenthesization of $A_iA_{i+1}\cdots A_j$

whose cost was lower than optimum: a contraduction. A similar observation holds for the parenthesization of the subchain $A_{k+1}A_{k+2}\cdots A_j$ in the optimal parenthesization of $A_iA_{i+1}\cdots A_j$: it must be an optimal parenthesization of $A_{k+1}A_{k+2}\cdots A_j$.

2. **What is the dynamic programming algorithm for chain-matrix multiplication? What is its running time?**

    1. Take the sequence of matrices and separate it into two subsequences.
    2. Find the minimum cost of multiplying out each subsequence.
    3. Add these costs together, and add in the cost of multiplying the two result matrices.
    4. Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

3. **You may be asked to compute a product of matrices using the dynamic programming approach and report the number of operations required given the dimensionality of the input matrices.**

4. **What type of subproblems does the dynamic programming approach consider for solving the longest increasing subsequence and how does it combine them in order to solve larger ones?**

    The subproblems for the LIS problem as the suffixes for any given input. Find the longest increasing subsequence $A[j..n]$ that includes $A[j]$. Do this for each $j > i$.

5. **What is the running time of the dynamic programming solution for the longest increasing subsequence problem?**

    There are n subproblems. For every subproblem i, the worst case is that you need go through all the following items to get the maximum, which is $O(n-i)$

$$\sum_{i=1}^{n} O(i) = O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

6. **You may be given an example sequence and asked to compute its longest increasing subsequence following a dynamic programming approach. Your solution should provide the intermediate values computed by the algorithm (e.g., length of longest subsequence, previous pointer for each digit).**

7. **What are the subproblems defined by the dynamic programming approach for the longest common subsequence problem and how are they combined to solve larger problems?**

    Let $X = \{x_1, x_2, \cdots, x_m\}$ and $Y = \{y_1, y_2, \cdots, y_n\}$, and let $Z = \{z_1, z_2, \cdots, z_k$ be any LCS of X and Y.

    Three cases:

    1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
    2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
    3. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X and Y_{n-1}.

8. **What is the running time of the dynamic programming solution for the longest common subsequence problem?**

    To build th LCS, you must construct a table with m columns and n rows. Each entry into the table is constant time.

$$O(m \times n)$$

9. **You may be given two strings and asked to compute their longest common subsequence following a dynamic programming approach. Your solution should provide the intermediate values computed by the algorithm (e.g., the matrix of values corresponding to the cost of the matching and the parent pointers).**

2.3 Elements of Dynamic Programming, Knapsack problem, Graph Representation

2.3.1 Chapter 6.4 from DPV: Knapsack

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His knapsack will hold a total weight of *at most* W pounds. There are N items to pick from, of which $w_1, \cdots, w_n$ and dollar value $v_1, \cdots, v_n$. What's the most valuable combination he can fit in his bag?

Knapsack with repetition

- What are the subproblems? We can look at it two ways:
  1. We can look at smaller knapsack capacities $w \leq W$
  2. We can look at fewer items (for instance, items $1, 2, \cdots, j$, for $J \leq n$)
- The first restriction, roughly,

  $K(w)$ is the maximum achievable value with a knapsack of capacity w.

  - Can we express this in terms of smaller subproblems? Well, if you remove an item i from $K(w)$ In other words, $K(w) = K(w - w_i) + v_i$
- If the optimal solution to $K(w)$ includes the item i, then removing this item from the knapsack leaves an optimal solution to

$$K(w) = \max_{i:w_i \leq w}\{K(ww_i) + v_i\}$$

```
K(0) = 0
for w = 1 to W:
    k(w) = max \{ K(w - w_i) + v_i \}
return K(W)
```

Knapsack without repetition

$K(w, j)$ is the maximum value achievable using a knapsack of capacity w and items $1, \cdots, j$.

- Either item j is needed to achieve optimal value, or it isn't needed:

$$K(w, j) = max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

```
Initmizalize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n:
    for w= 1 to W:
        if w_j > w: K(w, j) = K(w, j - 1)
        else: K(w, j) = max(K(w, j - 1), K(w - w_j, j - 1) + v_j)

return K(W, n)
```

2.3.2 Chapter 3.1 from DPV: Why graphs?

- We can represent a graph with an *adjacency matrix*, if there are $n = |V|$ vertice $v_1, ..., v_n$, this is an $n \times n$ array whose (i, j)th entry is:

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

  - The biggest convinience of this is that an edge can be checked in constant time.
  - However, it takes $O(n^2)$ space, wasteful if not many edges.

- An alternative representation is an *adjacency list*.

  - It consists of $|V|$ linked lists, one per vertex.
  - Each edge appear in exactly one of the linked lists if the graph is directed or two of the lists if the graph is undirected.
  - The total size of the data structure is $O(|E|)$.
  - Checking an edge is no longer linear, sift through u's adjacency list.

2.3.3 Chapter 22.1 from CLRS2: Representations of Graphs

- A graph is sparse if $|E|$ is much less than $|V|^2$.
- A graph is dense if $|E|$ is close to $|V|^2$.
- Adjency list representations of graphs consist of any array of $|V|$ lists, one for each vertex in V.

  - The adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$.

2.3.4 Practice questions

1. **What are the subproblems defined by the dynamic programming approach for the knapsack problem with repetition how are they combined to solve larger problems? What is the runtime?**

   Define $K(w)$ as the maximum value achievable with a knapsack of capacity w. The subproblems, then, where v is value of item w at i:

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

   The algorithm falls nicely out:

```
K(0) = 0;
for w = 1 to W:
    K(w) = max(K(w - w_i) + v_i : w_i <= w)
return K(W)
```

Each entry can tke up to $O(n)$ time to compute, so the overall running time is

$$O(nW)$$

2. **What are the subproblems defined by the dynamic programming approach for the knapsack problem without repetition how are they combined to solve larger problems? What is the runtime?**

   Define $K(w, j)$ as the "maximum value chieable using a knapsack of capacity w and items $1, \cdots, j$." The answer we seek is $K(W, n)$. The subproblems, therefore, are where j is either needed or not:

   $$K(w, j) = \max\{L(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

   The algorithm consists of filling out a two dimesional table, with $W + 1$ rows and $n + 1$ columns. Each table entry is constant time, even though the table is much larger than in the previous case, the running time is still:

   $$O(nW)$$

3. **You may be given the parameters of a knapsack problem and asked to compute the answer following a dynamic programming approach (for either case: with or without repetition). Your solution should provide the intermediate values computed by the algorithm (e.g., the vector or matrix of intermediate payoffs and objects selected by the intermediate solutions).**

4. **What are the advantages and disadvantages of an adjacency matrix representation for graphs?**

   - Advantages:
     - Constant time checking of edges.
   - Disadvantages:
     - Requires $O(n^2)$ space no matter how dense or sparse.

5. **What are the advantages and disadvantages of an adjacency list representation for graphs?**

   - Advantages:
     - Less space than matrix, very small for sparse graphs.
   - Disadvantages:
     - For dense graphs, look up for edges will be costly, linear.

2.4 Depth-First Search

2.4.1 Chapter 3.2 from DPV

- Depth-first search is a verstile linear-time procedire that reveals a wealth of information about a graph.

What parts of the graph are reachable from a given vertex?

```
procedure explore(G, v) {
    input:  G=(V, E) is a graph; v in V
    output: visited(u) is set to true for all nodes u reachable from v

    visited(v) = true;
    previsit(v);

    for each edge(v, u) in E:
        if not visited(v): explore(u);

    postvisit(v);
}

procedure dfs(G) {
    for all v in V
        visited(v) = false;

    for all v in V
        if not visisted(v): explore(v);
}
```

- Analysis:
  - To mark all as unvisisted:

$$O(|V|)$$

  - Each edge is exacmied exactly twice,

$$O(|E|)$$

  - Therefore,

$$O(|V| + |E|)$$

Connectivity in undirected graphs

- An undirected graph is connected if there is a path between any pair of vertices.
- In unconnected graphs, there are at least two connected components.
- All it takes to make DFS assign connected component numbers to each on,

```
procedure previsit(v) {
    ccnum[v] = cc
}
```

2.4.2 Chapter 22.5 from CLRS2

- An application for DFS: decomposisiting a graph into strongly connected components.
  - To be a strongly connected component of a directed graph is to have a maxmimal set $C \subset V$ such that for every pair of vertices u and v in C, we have both a path from u to v and vice versa.

2.4.3 Practice questions

1. **Provide an algorithm that discovers in linear time the set of nodes in a graph reachable from a specific node. Argue about its correctness.**

```
function get_connected_nodes(node){
    nodes = set();
        foreach(child in node.children){
                nodes.add(child);
                        nodes = nodes.union(get_connected_nodes(child));
                            }
                                return nodes;
                                }
        }
}
```

2. **Provide an algorithmic description for depth-first search. What is its running time?**

   Depth first search operates by first marking all vertexes in a graph as unvisisted. Then, it travereses all the vertexes in the graph. If a vertex has not been visisted, that vertex is explored. When a vertex is explored, it is set to have been viststed, and subseqeuntly every vertex that it shares and edge with is explored as well (that is, recursively). Optionally, the explore function can have a previsit and postvisit operation.

   There are two important elements to an analysis of depth first search. First, there is the marking of every node as unvisisted, which will take as much time as there are nodes. Then, note that every edge is visisted twice in the explore function. First, when it is found in the first node, and second, when its found in a node that looks at it, that is, in a undirected graph.

$$O(|E| + |V|)$$

3. **What is the pre-visit and post-visit order of nodes in depth-first search?**

   For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained within the other. Because $[pre(u), post(u)]$ is essentially the time during which vertex u was on the stack. The last in, first out behavior of the stack.

4. **How can depth-first search be used in order to detect the connected components of a graph?**

   If you modify the previsit function of DFS's explore function to number the visist node with a number that begins at 0 and is incrememented every time that explore function is called in the DFS function, you'll get marked nodes with a mark of their component.

5. **You may be provided an undirected or directed graph, a start node and asked to provide the search tree arising from a depth-first search. You may also be asked to identify tree edges, forward edges, back edges and cross edges (directed cases).**

6. **Show that a directed graph has a cycle if and only if its depth-first search reveals a back edge.**

   If (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

   If the graph *has* as cycle $v_0 \to v_1 \to \cdots \to v_k \to v_0$, look at the first node on this cycle to be discovered. Suppose it is v_i. All other v_j on the cycle are reachable from it and will therefore be its descendants in the search tree. In particular, the edge $v_{i-1} \to v_i$ (or $v_k \to v_0$ if $i = 0$) leads from a node to its ancestor and is thus by definition a back edge.

7. **What is a directed acyclic graph (DAG)?**

   A cycle in a directed graph is a circular path. A graph without one of these is acyclic. If that acyclic graph's edges have direction, then it is a DAG.

8. **What is the topological ordering of nodes in a DAG and why it is useful?**

   To linearize a DAG, perform DFS and perform tasks in decreasing order of their post number.

   Alternatively, find a source, output it, and delete it from the graph.

   It's useful because you can model causalityies, hierarchies, and temporal dependancies.

9. **How is it possible to identify a sink node or a source node in a DAG using depth-first search?**

   To indentify a sink node in a DAG, perform DFS and when a node doesn't have any edges, it will be a sink.

   To identify a source node in a DAG using DFS,

10. **What is the definition of strongly connected components?**

    If two components in a graph have an edge between u and v going in both directions, then those components are strongly connected.

11. **How is it possible to compute the decomposition of a directed graph into its strongly connected components? Provide an algorithm and argue about its running time.**

    The algorithm:

    - Let G be a directed graph and S be an empty stack.
    - While S does not contain all vertices:
      - Choose an arbitrary vertex v not in S. Perform DFS starting at v. Each time that DFS finished expanding a vertex u, push u onto S.
    - Obtain the transpose of G.
    - While S is not empty:
      - Pop the top vertex v from S.
      - Perform DFS starting at v in the transpose graph. The set of visisted vertices will give the strongly connected component containing v.
      - Record this and remove all these vertices from the graph G and the stack S.

    The run time: This algorithm performs two traversals of the graph.

$$O(|V| + |E|)$$

2.5 Breadth-First Search, Properties of Shortest Paths

2.5.1 Chapter 4.1 from DPV: Distances

- The *distance* between two nodes is the length of the shortest path between them.

2.5.2 Chapter 4.2 from DPV: Breadth-first search

- Each vertex is queued exactly once, when it is first encountered, so there are $2|V|$ queue operations.
  - The rest of the work is done in the algorithm's innermost loop.
  - Over the course of execution, this loop looks at each edge once in directed graphs or twice in undirected graphs, and therefore takes $O(|E|) time.
- Depth first search makes deep uncursion into a graph, retreating only it runs out of new nodes to visit.
  - BFS makes sure to visist vertices in increasing order of their distance from the starting point. This is broader, shallower search, rather like the progpagation of a wave upon water.
  - Same as DFS, but with a queue instead of a stack.
  - Nodes not reachable from s are simply ignored.

2.5.3 Chapter 22.2 from CLRS2: Breadth-first search

- BFS is one of the simplest algorithms for searching a graph.
- Given a graph $G = (V, E)$ and a dinstinguished source vertex s, BFS systematically explores the edges of G to "discover" every vertex that is reachable from s.
- It's named this because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - Discoveres distances at k before $k + 1$.

2.5.4 Practice questions

1. **Provide the algorithm that performs breadth-first search on a graph.**

```
procedure bfs(G, s)
input: Graph G=(V, E) directed or undirected; vertex s in V
output: For all vertices u reachable form s, dist(u) is set to distance
        from s to u.

for all u in V:
    dist(u) = inf

dist(s) = 0
Q = [s] (queue just containing s)
while Q is not empty:
    u = enject(Q)
    for all edges (u, v) in E:
        if dist(v) = inf
```

```
inject(Q, v)
dist(v) = dist(u) + 1
```

2. **What is the inductive argument for the correctness of the approach?**

   What we expect:

   For each $d = 0, 1, 2 \cdots$, there is a moment at which:

   1. All nodes at distance $\leq d$ from s have their distances correctly set;
   2. All other nodes have distances set to ∞; and
   3. The queue contains exactly the nodes at distance d.

   Proof: Assume, for the purpose of contraduction, that some vertex receives a d value not equal to its shortest path distance. Let v be th vertex with minimum $\Delta(s, v)$ and

3. **What is the running time of the algorithm?**

   $$O(|V| + |E|)$$

   Each vertex is put on the queue exactly once, when it is first encountered, so there are $2|V|$ queue operations. The loop then looks at each edge once (or twice in undirected), and therefore takes $O(|E|)$ time.

4. **What is the definition of a single-source shortest path problem?**

   in which we have to find shortest paths from a source vertex v to all other vertices in the graph.

5. **What is the definition of a single-destination shortest path problem?**

   in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v.

   Protip: Reverse the direction of SSSP.

6. **What is the definition of a single-pair shortest path problem?**

   in which we have to find the shortest path from source s to destination d.

7. **What is the definition of all-pairs shortest path problem?**

   in which we have to find shortest paths between every pair of vertices v, v' in the graph.

8. **Which of these problems are equivalent?**

   None of them?

9. **Do we know faster algorithms for the single-pair shortest path problem than the single-source shortest path problem?**

   Yes, single pair will be faster than single-source, single source has to compute more.

   Djikstra's: $O(|E| + |V| \log |V|)$

   Bellman-Ford: $O(|V| \times |E|)$

10. **What is the "optimal substructure" property of shortest paths? Prove its validity.**

    If you split a shortest path in half, the shortest path from the source to halfway will be the first half of the shortest path to the destination.

11. **What happens with shortest paths on graphs that contain negative cycles? Can a shortest path contain positive cycles?**

    1. There are going to be infinitely many shortest paths, you can always go around that vertex more than once.
    2. Yes.

2.6 Dijkstra's algorithm

2.6.1 Chapter 4.3 from DPV: Lengths on edges

- BFS treats all edges as having the same length. This is rarely true in applications where shortest paths are to be found.
  - Annotating every edge $e \in E$ with length l_e.
  - If $e = (u, v)$, we will sometimes also write $l(u, v)$ or l_{uv}

2.6.2 Chapter 4.4 from DPV: Dijkstra

2.6.3 Chapter 4.5 from DPV: Priority queue implementations

Array

- The simplimest implementation.
  - Key values for all potential elements.
- `insert` and `decreasekey` is fast, because it just involves adjusting the key value.
  - `deletemin` requires a scan of the list.

Binary heap

- `insert`: place a new element at the bottom of the tree and let it "bubble up", where the number of switches is at most $\log n$, height of the tree.
  - `decrease key` is similar.
- `deletemin` requires deleting the root.

2.6.4 Chapter 24.3 from CLRS2: Dijkstra's algorithm

- Solves the single single-source shortest path problem on a weighted directed graph $G = (V, E)$ for the case where the edge-weights are non-negative.

2.6.5 Practice questions

1. **Provide Dijkstra's algorithm for computing single-source shortest paths.**

```
procedure dijkstra(G, l, s)
input:   Graph G = (V, E), directed or undirected;
         positive edge length {l_e : e in E}; vertex s in V
output:  For all vertices u reachable from s, dist(u) is set
         to the distance from s to u

for all u in V:
    dist(u) = inf
    prev(u) = nil
dist(s) = 0;

H = makeque(V) (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) in E:
        if (dist(v) > dist(u) + l(u, v))
        dist(v) = dist(u) + l(u, v)
        prev(v) = u
        decreaseKey(H, v)

djkstra(G, w, s) {
    initialize-single-source(G, s)
    s <- empty set
    q = V[G]
    while Q is not empty
        do u = extract-min(Q)
        S = S union {u}
        for each vertex v in Adj[u]
            do relax(u, v, w)
    }
```

2. **What is the requirement in order to be able to apply Dijkstra's algorithm?**

   Edge weight must all be positive.

3. **You may be provided an example graph and asked to return the search tree that arises from Dijkstra, as well as the state of the priority queue during its iteration of the algorithm.**

   Can do.

4. **Prove the correctness of Dijkstra's algorithm on non-negative weight graphs.**

   Cannot do.

5. **What is the best running time of Dijkstra's algorithm and for what implementation of the priority queue data structure? What is the running time of Dijkstra's algorithm if the priority queue is implemented as a simple array? What is the running time of Dijkstra's algorithm if the priority queue is implemented as a binary heap?**

Implementation	`deletemin`	`increase/decreasekey`	Total
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary hear	$O(\frac{d \log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O((\|V\| \times d + \|E\|) \frac{\log \|V\|}{\log d})$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$	$O(\|V\| \log \|V\| + \|E\|)$

6. **When is each of these implementations preferred over the other?**

   It depends whether the the graph is *dense* or *sparse* edges. To be dense is to have many edges, to be sparse is to have fewer edges. For all graphs, $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, then cearly the array implementation is faster. On the other hand, the binary heap becomes preferable as soon as $|E|$ dips below $||V|^2 \log |V|$.

2.7 Bellman-Ford and Floyd-Warshall algorithms

2.7.1 Chapter 4.6 from DPV: Shortest paths in the presence of negative edges

- How can you account for the fact that when you have a negative edge, the shortest path may go through more nodes than any other path?
- What has to change to accomodate this sort of change?
  - Distance estimates under dijkstra are always overestimates or exactly correct. Consider:

```
procedure update((u, v) in E) {
    dist(v) = min(dist(v), dist(u) + l(u, v));
}
```

  - This *update* operation is simply an expression of the fact that the distance to v cannot possibly be more than the distance to u. It has the following properties:
    1. It gives the corect distance to v in the case where it is the second last node in the shortest path to v.
    2. It will never make `dist(v)` too small, making it *safe*.

Negative cycles

- In situations with negative edge weights, it doesn't make sense to even ask about shortest paths.
- The shortest-path problem is ill-posed in graphs with negative cycles.
  - Shortest path slipped in when we used the notion of "existence" (evidently).
- There is a negative cycle if and only if some `dist` value is reduced during the final round.

2.7.2 Chapter 4.7 from DPV: Shortest paths in dags

- There are two subclasses of graph that exclude the possibility of negative cycles:
  1. Graphs without negative edges

2. Graphs without cycles

- The key source of efficiency:

  In any path of a dag, the vertices appear in increasing linearized order.

- You can find the longest path in a dag by negating all edge lengths.

2.7.3 Chapter 6.1 from DPV: Shortest paths in dags, revisited

- The special feature of dags is that the nodes can be *linearized*, that is, put on a straight line from left to right.
- You can compute all distances with a single pass of the following algorithm:

```
initialize all dist(.) values to inf;
dist(s) = 0;
for each v in V\{s}, in linearized order:
    dist(v) = min_{(u, v) in E} (dist(u) + L(u, v));
```

- Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by indentifying a collection of subproblem and tacking them one by one, smallest first, using the answers to small problems to help figure out larger onces, until the whole lot of them are solved.
  - In dynamic programming, you aren't always given a dag, the dag is implicit.
  - It's nodes are the subproblems we define, and it edges are the dependancies between subproblems:
    - If we solve subproblem B we need to answer subproblem A, making a conceptual edge from A to B.

2.7.4 Chapter 24.1 from CLRS2: The Bellman-Ford Algorithm

- The Bell-Ford Algorithm solves the single-source shortest-path in the general case in which edge weight may be negative.
  - Given a weight, directed graph $G = (V, E)$ with source s and weight function $w : E \to R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source.
    - If there is such a cycle, the algorithm indicates that there is no such solution.
  - If there is no cycle, the algorithm produces the shortest paths and their weights.
- The algorithm uses relaxation, progessively decreasing an estimate $d[v]$ on the weight of a shortest path.
  - The algorithm returns true if and only if there are no negative weighted cycles that are reachable form the source.

2.7.5 Chapter 24.2 from CLRS2: Single-source shortest paths in directed acyclic graphs

- By relaxing the edges of a weighted dag according to a toplogical sort of vertices, we can compute shortest paths from a single source in $\Omega(V + E)$ time.
  - Shortest paths are always well-edfined in a dag, since even if there are non-negative edge weights, there are no negative cycles (because there are no cycles).
- The algorithm starts by topologically sorting the dag to impose a linear ordering on the vertices. If there is a path from u to v, it is topologically latter.
  - Only need a single pass.
    dag-shortest-paths(G, w, s) { topologically sort the vertices of G initialize-single-source(G, s); for each vertex U, take in topologically sorted order do for each vertex v in Adj[u] do relax(u, v, w); }
- The running tim eof this is easy.
  - The topological sort is
    $$\Omega(V + E)$$
  - The initialize-single-source takes
    $$\Omega(V)$$
  - Then to two inner loops examine every V and E.
    $$\Omega(V + E)$$

2.7.6 Practice questions

1. **Provide the Bellman-Ford algorithm for computing single-source shortest paths.**

```
procedure shortest-paths(G, l, s) {
input:  Directed graph G = (V, E);
        edge lengths {l_e : e in E} with no negative cycles;
        vertex s in V
output: For all vertices u reachable form s, dist(u) is set
        to the distance from s to u.

for all u in V:
    dist(u) = inf;
    prev(u) = nil;

dist(s) = 0;
repeat|V| - 1 times:
    for all e in E:
        update(e);
}

procedure BellmanFord(list vertices, list edges, vertex source) {
    // step 1: initialization
    for each vertex v in vertices:
        if v is source then distace[v] := 0;
```

```
        else distace[v] := infinity
        precessir[v] := null

    // step 2: relax edges repeatedly
    for i from 1 to size(vertices) - 1:
        for each edge (u, v) with weight w in edges:
            if distace[u] + v < distance[v]:
            distance[v] := distance [u] + w
            predecessor[v] := u

    // step 3: check for negative weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph containts negative weight cycle."
}
```

2. **What is the running time of the approach and why?**

   The Bellman-Ford algorithm run in time $O(VE)$ since the initialization takes $O(V)$ time, and each of the $|V| - 1$ passes over the edges takes $\Theta(E)$ time, and the for look takes $O(E)$ time.

3. **Why is it correct?**

4. **You may be provided a graph and asked to trace the dynamic programming matrix that arises from the operation of Bellman-Ford.**

5. **How can you detect the existence of negative cycles using the Bellman-Ford algorithm?**

   There is a look at the end like so,

```
for each edge (u, v) with weight w in edges:
   if distance[u] + w < distance[v]:
       error "Graph containts negative weight cycle."
```

6. **What is an efficient approach for computing single-source shortest paths on directed acyclic graphs that may contain negative weight edges and what is the running time of this solution?**

7. **Why is the Floyd-Warshall algorithm preferred over calling |V| times the Bellman-Ford algorithm on a graph to solve all-pair shortest path problems?**

8. **What is the dynamic programming formulation of Floyd-Warshall, i.e., what are the subproblems defined by the algorithm and how are they combined in order to address more complex problems?**

2.8 Floyd-Warshall and Johnson's algorithms

2.8.1 Chapter 25.2 from CLRS2: The Floyd-Warshall algorithm

- A dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$.
  - Floyd-warshall runs in $\Omega(V^3)$ time.
  - Negative weight edges may be present, but we assume that there are no negative-weight cycles.
  - Will follow the dynamic-programming process to develop the algorith,

The structure of a shortest path

- The FW algorithm uses a different charaterization of the shortest path than the matrix-multiplcation-based all-pairs algorithms.
  - The algorithm considers the "intermediate" vertices of a shortest path, where intermediate means a node on the shortest path.
- Assume that the vertices of G are $V = \{1, 2, \cdots, n\}$ and consider the ubset $\{1, 2, \cdots, k\}$ of vertices for some k.
  - The Floyd-Warshall algorithm explain a relationship between path p and shortest paths from i to j with all intermediate nodes in $1, 2, \cdots, k-1\}$.

2.8.2 Chapter 25.3 from CLRS2: Johnson's algorithm for sparse graphs

- Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \log V + VE)$ time.
  - For sparese graphs, it is asymptotically better than either repeated squaring matrices of the Floyd-Warshall algorithm.
  - The algorithm either returns a matrix of short-path weights for all pairs of matrices or reports that the graph contains a negative weight cycle.
- Johnson's algorithm uses the technique of *reweighting*, which is:
  - If all edge weights w in graph $G = (V, E)$ are nonnegative, we can find shortest paths between all pairs of certices by running Dijkstra's algorithm once from each vertex.
  - With the Fibonacci-heap min-priority queue, the running time of this all-pairs will be $O(V^2 \log V + VE)$.
  - If G has negative-weight edges but no negative-weight cycles, we simply compute a new set of nonnegative edge weights that allow us to use the same method.
    - The new set of edge weights must satisfy two important properties:
      1. For all pairs of vertices u, $v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $w*$
      2. For all edges (u, v), the new weight $w * (u, v)$ is nonnegative.

2.8.3 Chapter 6.6 from DPV2

2.8.4 Practice questions

1. **Provide the Floyd-Warshall algorithm for solving all-pair shortest path problems. What is the running time of the approach?**

```
floud-warshall() {
    let dist be a |V| x |V| array of minimum distances initialized to inf
    for each vertex v:
        dist[v][v] = 0;

    for each edge (u, v) {
        dist[u][v] := w(u, v); // the weight of the edge
    }
```

```
    for k from 1 to |V|
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j]
                    dist[i][j] = dist[i][k] + dist[k][j]
}
```

2. **You may be provided a graph and asked to compute the dynamic programming matrix that arises from a few iterations of the Floyd-Warshall algorithm.**

3. **What is the transitive closure of a graph and how can it be computed following a dynamic programming approach? What is the relation to the Floyd-Warshall algorithm?**

   The transitive closure of G is defined as the graph $G* = (V, E*)$, where $E* = \{(i, j) :$ there is a path from vertex$i$to vertuex$j$in$G\}$.

4. **If a graph has non-negative edges, is it preferable to run the Floyd-Warshall algorithm to solve an all-pair shortest path problem or is it preferable to call |V| times Dijkstra's algorithm?**

   The complexity for running Dijkstra on all nodes will be $O(EV + V^2 \log V)$. This complexity is lower than $O(V^3)$ iff $E < V^2$.

5. **What is the main idea in Johnson's algorithm and why can it be preferable over the Floyd-Warshall algorithm when solving all-pair shortest path problems?**

   The main idea behind Johnson's algorithm is that it uses Bellman-Ford to remove all negative edge weights from a graph using an artificial vertex, and then performs Djikstra's algorithm to solve the all-pair shortest path problem.

   It will be preferable over Floyd-Warshall because it has a better run time.

6. **Given the above transformation of weights for a graph G(V,E), how should the values h be computed for the vertices of the graph so that all weights end up having non-negative values?**

7. **Describe Johnson's algorithmic steps. What is its running time?**

```
Johnson(G) {
    compute G', where V[G'] = V[G] union {s},
        E[G'] = E[G] union {(s, v) : v in V[G]} and
        w(s, v) = 0 for all v in V[G]

    if Bellman-Ford(G', w, s) = false
        then print "the input graph contain a negative-weight cycle"
        else for each vertex v in V[G']
            do set h(v) to the value of delta(s, v)
                computed by Bellman-Ford

        for each edge (u, v) in E[G']
            do w*(u, v) := w(u, v) + h(u) - h(v);
        for each vertex u in V[G]
            do run Dikstra(G, w*, u) to compute delta *(u, v) for all v in V[G]
```

```
                for each vertex v in V[G]
                    do d_{u, v} = delta *(u, v) + h(v) - h(u)

        return D
}
```

- The steps to the algorithm:
  1. A new node q is added to the graph, connected by zero-weight edges to each of the other nodes.
  2. The Bellman-Ford algorithm is used, starting from the new vertex q to find for each vertex v the minimum height $h(v)$ of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.
  3. The edges of the original graph are reweighted using the value computed by the Bellman-Ford algorithm: an edge from u to v, having length $w(u, v)$, is given the new length $w(u, v)+h(u)-h(v)$.
  4. q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.
- Time complexity:
  - Using Fibonacci heaps in the implementatino of Dijkstra's algorithm is $O(V^2 \log V + VE)$.
    - The algorithm uses $O(VE)$ time for the Bellman-Ford stage of the algorith,
  - $O(V \log V + E)$ for each of the V instantiations of Dijkstra's algorithm.
  - Thus, when the graph is sparse, the total time will be faster than the Floyd-Warhshall algorithm, which solves the same problem in $O(V^3)$.

3 Final Study Guide

3.1 Minimum Spanning Trees

3.1.1 Chapter 5.1 from DPV: Minimum spanning trees

- This is in search of the optimal set of edges.
- This cannot contain an edge – edges are always extraneous.

A greedy approach

- Kruskal's minimum spanning tree algorithm starts with an empty graph and then selects edges from E according to the following rule:

  Repetedly add the next lightest edge that doesn't produce a cycle.

- It constructs the tree edge by edge, and simply picks whichever edge is cheapest at the moment.

Cut property

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

- A *cut* is any partition of the vertices into two groups, S and $V - S$. What this property says is that it is always safe to add the lightest edge across any cut provided X has no edges across the cut.

$$T' = T \cup \{e\} - \{e'\}$$

Kruskal's algorithms At any given moment, the edge it has already chose form a partial solution, a collection of connected components each of which has a tree structure. The next edge e to be added connects two of these components; call them T_1 and T_2. Since e is the lightest edge that doesn't produce a cycle, it is certain to be the lightest edge between T_1 and $V - T_1$ and therefore satisfyies the cut property.

Now we fill in some implementation details. At each state, the algorithm chooses an edge to add to its current partial solution. To do so, it need to test each candidate edge $u - v$ to see whether the endpoints u and v lie in different components; otherwise the edge produces a cycle. And once an edge is chose, the corresponding components need to be merged. What kind of data structure supports this operation?

We will model the algorithm's state as a collection of disjoint sets, each of which contains the nodes of a particular component.

`makeset(x)`: create a singleton set containing just x. Initially each node is in a component by itself.

`find(x)`: to which set does x belong? We repeatedly test pairs of nodes to see if they belong to the same set.

`union(x, y)`: merge the sets containing x and y. And wherever we add an edge, we are merging two components.

```
procedure kruskal(G, w)
input:  a connect undirect graph G = (V, E) wieth edge weights w_e.
output: a minimum spanning tree defined by the edges X

for all u in V
    makeset(u)

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

3.1.2 Chapters 23.1 from CLRS2: Growing a minimum spanning tree

```
generic-MST(G, w)
    a = {}
    while A does not form a spanning tree
        do find an edge (u, v) that is safe for A
            A = A + {(u, v)}
    return A
```

- A *cut* $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V.

- We say that an edge $(u, v) \in E$ *crosses* the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$
- We say that a cut *respects* a set A of edge if no edge in A crosses the cut.
- An edge is a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut.
- More generally, we say an edge is a *light edge* satisfying a given property if its weight is the minimum of any edge satisfying the property.

3.1.3 Practice questions

1. **What kind of graph-based optimization problems require as a solution the computation of a Minimum Spanning Tree?**

   In the design of electronic circuitry, it is often necessary to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, we can use an arrangement of $n - 1$ wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

2. **Why the solution to such problems does not contain cycles?**

   The solution to any MST problem does not contain cycles because of the property that any cycle can be removed and disconnect a graph – they are always extraneous.

3. **Provide a generic description of an algorithm for the computation of minimum spanning trees.**

4. **What is a cut of a graph?**

   A cut is a division of vertices in an MST into S and $S - V$. The cut property says that it is always safe to add the lightest edge between the two groups and get an MST when the extraneous edge is removed.

5. **When does an edge cross a cut?**

   When this is true, where T' and T are MSTs:

   $$T' = T \cup e - e'$$

6. **When does a cut respect a set of edges?**

   We say that a cut *respects* a set A of edge if no edge in A crosses the cut.

7. **What is the definition of a light edge?**

   An edge is a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut. More generally, we say an edge is a *light edge* satisfying a given property if its weight is the minimum of any edge satisfying the property.

8. **Assume a connected, undirected graph $G(V, E)$ with weights w and a set of edges $A \in E$ that is a subset of minimum spanning tree T. Consider then any cut $(S, V - S)$ that respects the set A. Prove that if (u, v) is a light-edge crossing $(S, V - S)$, then (u, v) can be safely be added to A as an edge that is part of a minimum spanning tree of G.**

   - Let T be a minimum spanning tree that includes A, and assume that T does not contain the light edge (u, v), since if it does, we are done. We shall construct another minimum spanning tree T' that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that (u, v) is a safe edge for A.

- The edge (u, v) forms a cycle with the edges on the path p from u to v in T. Since u and v are on opposite sides of the cut $(S, V - S)$, there is at least one edge in T on the path p that also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique path from u to v in T, removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree $T = T - \{(x, y)\} \cup \{(u, v)\}$.
- We next show that T is a minimum spanning tree. Since (u, v) is a light edge crossing $(S, V - S)$ and (x, y) also crosses this cut, $w(u, v) \leq w(x, y)$. Therefore,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) \end{aligned}$$

- But T is a minimum spanning tree, so that $w(T) \leq w(T')$; thus, T' must be a minimum spanning tree also.

9. **Given the above statement, prove the following: Let $C = (V_C, E_C)$ be a connected component (tree) in the forest $GA(V, A)$. If (u, v) is a light-edge connecting C to another component in G_A, then (u, v) is safe for A.**

    - The cut $(V_C, V - V_C)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A.

3.2 Kruskal's and Prim's algorithms

3.2.1 Chapters 23.2 from CLRS2: The algorithms of Kruskal and Prim

- Each algorithm given here is a filling in of the generic MST.
  - In Kruskals, the set A is a forest. The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.
  - In Prim's, the set A forms a single tree. The sage edge added to A is always a least-weight edge connecting the tree to a vertex in the tree.

Kruskal's

Description

- It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight.
- Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.
- It uses a disjoint-set data structure to maintain several disjoint sets of elements.
- Each set contains the vertices in a tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u.
- Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v).
- The combining of trees is accomplished by the UNION procedure.

Steps

- Lines 1–3 initialize the set A to the empty set and create |V | trees, one containing each vertex.
- The edges in E are sorted into nondecreasing order by weight in line 4.
- The for loop in lines 5–8 checks, for each edge (u,v), whether the endpoints u and v belong to the same tree.
  - If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded.
  - Otherwise, the two vertices belong to different trees.

Running time

- with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known.
- Initializing the set A in line 1 takes $O(1)$ time,
- and the time to sort the edges in line 4 is $O(E \lg E)$.
- The for loop of lines 5–8 performs $O(E)$ `FIND-SET` and `UNION` operations on the disjoint-set forest. Along with the $|V|$ `MAKE-SET` operations, these take a total of $O((V + E)\alpha(V))$ time,
- $\alpha(|V|) = O(\lg V) = O(lgE)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

Prim's

Description

- At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree.
- In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Steps

- Lines 1–5 set the key of each vertex to infinity (except for the root r, whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to `NIL`, and initialize the min- priority queue Q to contain all the vertices. The algorithm maintains the following three-part loop invariant:
- Prior to each iteration of the while loop of lines 6–11,
  1. $A = \{(v, \pi[v])v \in V - \{r\} - Q\}$.
  2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
  3. For all vertices $v \in Q$, if $\pi[v] \neq NIL$, then $key[v] < \infty$ and % is the weight of a light edge $(v, \pi[v])$ connecting v to some vertex already placed into the minimum spanning tree.
- Line 7 identifies a vertex $u \in Q$ incident on a light edge crossing the cut $(V - Q, Q)$ (with the exception of the first iteration, in which u = r due to line 4). Removing u from the set Q adds it to the set $V - Q$ of vertices in the tree, thus adding $(u, \pi[u])$ to A. The for loop of lines 8–11 update the key and π fields of every vertex v adjacent to u but not in the tree. The updating maintains the third part of the loop invariant.

Running time

3.2.2 Chapter 21.2 from CLRS2: Linked-list representation of disjoint sets

- A simple way to implement a disjoint-set data structure is to represent each set by a linked list. The first object in each link list serves as its set's representative. Each object in the linked list contains a set member, a point to the object containing the next set member, and a pointer back to the representative. Each list mains pointers *head* to the representative and *tail* to the last object in the list. Within each linked list, the objects may appear in any order.
- With linked-list representation, both `Make-Set` and `Find-Set` are easy, requiring $O(1)$ time. To carry out `Make-Set(x)`, we create a new linked list whose only object is x. For `Find-Set`, we just return the pointer from x back to the representative.

3.2.3 Chapter 21.3 from CLRS2: Disjoint-set forests

- In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set.
  - In a disjoint-set forest, illustrated in Figure 21.4(a), each member points only to its parent.
  - The root of each tree contains the representative and is its own parent.
  - As we shall see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, by introducing two heuristics - union by rank and path compression - we can achieve the asymptotically fastest disjoint-set data structure known.
- We perform the three disjoint-set operations as follows.
  - A MAKE-SET operation simply creates a tree with just one node.
  - We perform a FIND-SET operation by following parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the find path.
  - A UNION operation, shown in Figure 21.4(b), causes the root of one tree to point to the root of the other.
- The first heuristic, **union by rank**, is similar to the weighted-union heuristic we used with the linked-list representation.
  - The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes.
  - Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis.
  - For each node, we maintain a rank that is an upper bound on the height of the node.
  - In union by rank, the root with smaller rank is made to point to the root with larger rank during a UNION operation.
- The second heuristic, **path compression**, is also quite simple and very effective.
  - As shown in Figure 21.5, we use it during FIND-SET operations to make each node on the find path point directly to the root.
  - Path compression does not change any ranks.

3.2.4 Practice questions

1. **Given the generic algorithm for the computation of a minimum spanning tree, how is the safe edge computed in Kruskal's algorithm?**

   It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight.

2. **How is it computed in Prim's algorithm?**

   At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$

3. **You may be provided a graph and asked to trace the steps of Kruskal's and Prim's algorithms for the computation of the minimum spanning tree of the graph.**

   Okay!

4. **Provide Kruskal's algorithm.**

```
MST-Kruskal(G, w) {
    A = {}
    for each vertex v in V[G]
        do Make-Set(v)
    sort the edges of E into nondecreasing order by weight w
    for each edge (u, v) in E, taken in nondecreasing order by weight
        do if Find-Set(u) != Find-Set(v)
            then A = A + {(u, v)}
                Union(u,v)
    return A
```

5. **Consider an implementation of the disjoint sets data structure with the union-by-rank heuristic as part of the implementation of Kruskal's algorithm. What is the running time of Kruskal's algorithm in this case and why?**

6. **Consider an implementation of the disjoint sets data structure with the union-by-rank and the path compression heuristic and that the edge weights are upper bounded by the value $|E|$. What is the running time of Kruskal's algorithm in this case and why?**

   - Instation takes $O(1)$, and the time to sort the edges is $O(E \lg E)$
   - The $|V|$ Make Set calls take $O(V + E\alpha(V))$

$$\alpha(V) = O(\lg V) = O(\lg E)$$

$$O(E \lg V)$$

7. **Describe the `makeset`, `find_set` and `union` functions of the disjoint sets data structure given the union-by-rank heuristic.**

   - `makeset(x)`: create a singleton set containing just x. Initially each node is in a component by itself.
   - `find(x)`: to which set does x belong? We repeatedly test pairs of nodes to see if they belong to the same set.

- `union(x, y)`: merge the sets containing x and y. And wherever we add an edge, we are merging two components.
- The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis. For each node, we maintain a rank that is an upper bound on the height of the node. In union by rank, the root with smaller rank is made to point to the root with larger rank during a UNION operation.

8. **What is the running time of these operations?**

   - `makeset(x)`

     $$O(1)$$

   - `find(x)`

     $$O(1)$$

   - `union(x, y)`

     $$O(n)$$

9. **How does the `find_set` function change when you also use the path compression heuristic?**

```
Find-Set(x) {
    if x != p[x]
        then p[x] = Find-Set(p[x])
    return p[x];
}
```

   - This amortized cost turns out to be just barely more than O(1), down from the earlier O(log n).

10. **What is the new running time of the `find_set` and `union` functions in this case?**

    $$O(1)$$

    $$O(n)$$

11. **Provide Prim's algorithm.**

```
MST-Prim(G, w, r)
    for ech u in V[G]
        do key[u] = inf
            pi[u] = nil
    key[r] = 0
    Q = V[G]
    while Q != {}
        do u = extract-min(Q)
            for each v in Adj[u]
                do if v in Q and w(u, v) < key(u)
                    then pi[v] = u
                        key[v] = w(w, v)
```

12. **What is the running time of Prim's algorithm given an array implementation of a priority queue?**

13. **What is the running time of the algorithm given a binary heap implementation of a priority queue?**

3.3 The class of NP problems - Examples of NP complete problems

3.3.1 Chapter 8.1 from DPV: Search Problems

- Satisfiability is a problem of great practical important.
  - Here's an instance:

$$(x \vee y \vee z)(x \vee \neg y)(y \vee \neg z)(z \vee \neg x)(\neg x \vee \neg y \vee \neg z)$$

- This is *Boolean formula in conjunctive normal form* (CNF)
  - There are *clauses*
  - There are *literals*
- Is there are solution to this last case?

Search problem A *search problem* is specified by an algorithm C that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$. We say S is a solution to I if and only if $C(I, S) = true$.

- Yet, interestingly, there are two natural variants of SAT for which we do have good algo- rithms. If all clauses contain at most one positive literal, then the Boolean formula is called a Horn formula, and a satisfying truth assignment, if one exists, can be found by the greedy algorithm of Section 5.3.
- Alternatively, if all clauses have only two literals, then graph theory comes into play, and SAT can be solved in linear time by finding the strongly connected components of a particular graph constructed from the instance
  - In fact, in Chapter 9, we'll see a different polynomial algorithm for this same special case, which is called 2SAT.
- On the other hand, if we are just a little more permissive and allow clauses to contain three literals, then the resulting problem, known as 3SAT (an example of which we saw earlier), once again becomes hard to solve!

The traveling salesman problem

- In the TSP, we are given n vertices $1, ..., n$ all $n(n-1)/2$ distances between them, as well as a *budget* b. We are asked to find a you, a cycle that passes through every vertex exactly once, of total cost b or less – or to report that no such tour exists.

$$d_{p(1),p(2)} + d_{p(2),p(3)} + ... + d_{p(n),p(1)} \leq b$$

- Notice that this defines TSP as a search problem, where given an instance, find a tour within the budget (or report that no such one exists).

  - Why address it like this when it actually a *optimization problem*?

- Turning an optimization problem into a search a problem does not chance its difficulty at all, because the two version *reduce to one another*. Any algorithm that solves the optimization TSP also readily solves the search problem: find the optimum tour and if it within budget, return it, otherwise return no solution.

- Conversely, an algorithm for the search problem can also be used to solve the optimization problem.

  - To see why, first suppose that we show how knew the *cost* of the optimum tour; then we could find this tour by calling the algorithm for the search problem, using the optimum cost as the budget.
  - Fine, but how do we find the optimum cost? Easy: binary search.

- There is subtlety here: Why do we have to introduce a budget? Isn't any optimization problem also a search problem in the sense that we are searching for a solution that has the property of being optimal?

  - The catch is that the solution to a search problem should be easy to recognition, or as we it early, polynomial time checkable. Given a potential solution to the TSP, it is easy to check the properties "is a tour" and "has total length less than or equal to budget"
  - But how could you check that the property "is optimal"?

- As with SAT, there are no known polynomial time algorithms for the TSP.

  - In Section 6.6 we saw a faster, yet still exponential, dynamic programming algorithm.

Euler and Rudrata

- *Euler*: When can a graph be drawn without lifting the pencil from paper?

  - If and only if:
    1. The graph is connected.
    2. Every vertex with the possible exception of two vertices has even degree. (Degree is number of connected neighbors)
  - Can be solved in polynomial time.

- *Rudrata*: Can one visit all the squares of the chessboard without repeating any square, in one long walk that ends at the starting esquire and at eachs yep makes a legal knight move?

  - Indeed, define the RUDRATA PATH problem to be just like RUDRATA CYCLE, except that the goal is now to find a path that goes through each vertex exactly once.

Cuts and bisections

- A cut is a set of edges whose removal leaves a graph disconnected.

  - It is of interest to find small cuts, and this **minimum cut** problem is given a graph and a budget b, to find a cut with at most b edges.

- the *BALANCED CUT* problem is this: given a graph with n vertices and a budget b, partition the vertices into two sets S and T such that $|S|, |T| \geq n/3$ and such that there are at most b edges between S and T.

3.3.2 Chapters 34.2 from DPV: Polynomial-time verification

- We now look at algorithms that verify membership in languages. For instance, suppose you have a grammar of the decision problem PATH, we are also given a path p from u to v.
  - We can *easily* check whether the length of p is at most k, and if so, we can view p as a "certification" that the instance indeed belongs to PATH.

3.3.3 Practice questions

1. **What is the satisfiability problem?**

   The satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula

2. **What is the worst-case running time of the best known algorithm for this problem?**

3. **What is the running time of checking whether a candidate solution is truly solving the problem or not?**

4. **Are there versions of the general satisfiability problem that can be solved in polynomial time? Describe two of them.**

5. **What is the traveling salesman problem?**

   The travelling salesman problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

   TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (i.e. each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

6. **What is a dynamic programming approach for the traveling salesman approach?**

   Denote the cities by $1, ..., n$, the salesman's hometown being 1, and let $D = (d_{ij})$ be the matrix of intercity distances.

   The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length.

   The brute force approach takes $O(n!)$, but dynamic programming is better.

   Here is the appropriate subproblem:

   > For a subset of cities $S \in \{1, 2, ..., n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

   Now lets express $C(S, j)$ in terms of smaller subproblems. We need to state at 1 and end at j; what should we pick as the secon-to-last city? It has to be some $i \in S$, so the overall path lentth is the distance from 1 to i, namely $C(S - \{j\}, i)$, plus the length of the final edge, d_{ij}. We must pick the best such i:

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

The problems are ordered by S. Here's the code:

```
C({1}, 1) = 0
for s = 2 to n:
    for all subsets S in {1, 2, ..., n} of sizes s and containing 1:
        C(S, 1) = infinity
        for all j in S, j != 1:
            C(S, j) = min {C(s - {j}, i) + d_ij : i in S, i != j}
return min_j C({1, ..., n}, j) + d_j1
```

7. **What is the running time of this approach?**

   There are at most $2^n \times n$ subproblems, and each one takes linear time to solve. The total running time is therefore:

   $$O(n^2 2^n)$$

8. **Show that any optimization problem can be reduced to a search problem.**

9. **Show that any search problem can be reduced to an optimization problem.**

10. **Why do we typically prefer to work with search problems when studying the computational complexity of algorithms?**

11. **What is an Eulerian tour?**

    In graph theory, an Eulerian trail (or Eulerian path) is a trail in a graph which visits every edge exactly once. Similarly, an Eulerian circuit or Eulerian cycle is an Eulerian trail which starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. Mathematically the problem can be stated like this:

    > Given the graph on the right, is it possible to construct a path (or a cycle, i.e. a path starting and ending on the same vertex) which visits each edge exactly once?

    Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published posthumously in 1873 by Carl Hierholzer.

    Every edge once.

12. **When does a graph have an Eulerian tour?**

    An undirected graph has an Eulerian cycle if and only if every vertex has even degree, and all of its vertices with nonzero degree belong to a single connected component.

    If and only if (a) the graph is connected and (b) every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has even degree.

13. **What is the Rudrata cycle/path problem?**

    Let us define the Rudrata cycle search problem to be the following: given a graph, find a cycle that visits each vertex exactly once – or report that no such cycle exists.

    This problem is ominously reminiscent of the TSP, and indeed no polynomial algirhtm is known for it.

    a Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once.

    Every vertex once.

14. **Is there a polynomial time algorithm for this problem?**

15. **What is the independent set problem?**

    an independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set I of vertices such that for every two vertices in I, there is no edge connecting the two. Equivalently, each edge in the graph has at most one endpoint in I. The size of an independent set is the number of vertices it contains. Independent sets have also been called internally stable sets.

    The independent set problem is the search for a subset of nodes which contains no edges between them.

16. **Describe a dynamic programming solution for computing an independent set on trees.**

    Start by rooting the tree at any node r. Now, each node defines a subtree – the one hanging from it. This immediately suggests subproblems:

    $$I(u) = \text{size of largest independent set of subtree hanging from } U$$

    Our final goal is $I(r)$.

    Dynamic programming proceeds as always from smaller subproblems to larger ones, that is to say, bottom-up in the rooted tree. Suppose we know the largest independent sets for all subtrees below a certain node u; in other words, suppose we know $I(w)$ for all descendants w of u. How can we compute $I(u)$? Lets split the computation into two cases: any independent set include u or it doesn't.

    $$I(u) = \max \left\{ 1 + \sum_{\text{granchildren w of u}} I(w), \sum_{\text{children w of u}} I(w) \right\}$$

    If the independent sets includes u, then we get one point for it, but we aren't allowed to include the children of u – therefore, we move on to the grandchildren. This is the first case in the formula. On the other hand, if we don't include u, then we don't get a point for it, but we can move on to its children.

17. **What is the running time of this solution?**

    The number of subproblems is exactly the number of vertices. With a little care, the running time can be made linear.

    $$O(|V| + |E|)$$

18. **Is there a polynomial time algorithm for this problem on general graphs?**

19. **What is the vertex cover problem?**

    A vertex cover (sometimes node cover) of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.

    A vertex cover of a graph $G = (V, E)$ is a subset of vertices $S \subset V$ that includes at least one endpoint of every edge in E.

20. **What is the clique problem?**

    The clique problem refers to any of the problems related to finding particular complete subgraphs ("cliques") in a graph, i.e., sets of elements where each pair of elements is connected.

21. **Are there polynomial time algorithms for these problems?**

3.4 Additional examples of NP complete problems - Reductions

3.4.1 Chapter 8.2 from DPV: NP-complete problems

Hard problems, NP Complete

3SAT Like the satisfiability problem for arbitrary formulas, determining the satisfiability of a formula in conjunctive normal form where each clause is limited to at most three literals is NP-complete also; this problem is called 3-SAT, 3CNFSAT, or 3-satisfiability.

Traveling Salesman The travelling salesman problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Longest path the longest path problem is the problem of finding a simple path of maximum length in a given graph.

3D matching a 3-dimensional matching is a generalization of bipartite matching (a.k.a. 2-dimensional matching) to 3-uniform hypergraphs.

Knapsack Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Independent set an independent set or stable set is a set of vertices in a graph, no two of which are adjacent.

Integer linear programming a mathematical optimization or feasibility program in which some or all of the variables are restricted to be integers.

Rudrata path a Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once.

Balanced cut given a graph with n vertices and a budget b, partition the vertices into two sets S and T such that $|S|, |T| \leq n/3$ and such that there are at most b edges between S and T.

Easy problems (in P)

2SAT the problem of determining whether a collection of two-valued (Boolean or binary) variables with constraints on pairs of variables can be assigned values satisfying all the constraints.

Minimum spanning tree Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together

Shortest path problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

Bipartite matching a matching or independent edge set in a graph is a set of edges without common vertices.

- Unary knapsack
- Independent set on trees
- Linear programming

- Euler path

Minimum cut a minimum cut of a graph is a cut (a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge) whose cut set has the smallest number of edges (unweighted case) or smallest sum of weights possible.

The problems in P for a variety of reasons: dynamic programming, network flow, graph search, greedy, etc.

The problems in NP are hard for the same reason! At their core, they are all the same problem, but just different disguises. They are *equivilent*: as we shall see in 8.3, they can be reduced to one another and back.

P and NP

- We know what a search problem is: it's defining characteristic is that any proposed solution can be quickly check for correctness, in the sense that there is an efficient checking algorithm C that takes as input the given instance I and outputs `true` if and only if S really is a solution to instance I.
  - Moreover, the running tim of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instnace.
  - We denote the class of all search problems by **NP**.
- We've seen many examples NP search problems that are solvable in polynomial time.
  - There is an algorithm that takes as input an instance I and has a running time polynomial in $|I|$.
  - If I has a solution, the algorithm returns such a solution: and if I has no solution, the alogirhtm correctly reports so.
  - The class of all search problems that can be solved in polynomial time is denote by P.

Reductions, again

- Even is P is not NP, what about the specific problems on the left side of the table? On the basis of what evidence we do believe that these particular problems have no efficient algorithms.
  - Such evidence is provided by *reductions*, which translate one search problem into another.
  - What they demonstrate is that they problems on the left side are *exactly the same problem*, except that they are stated in different languages.

  A search problem is NP-complete if all other search problems reduce to it.

NP Class of computational problems for which solutions can be computed by a non-deterministic Turing machine in polynomial time (or less). Or, equivalently, those problems for which solutions can be checked in polynomial time by a deterministic Turing machine.

NP-hard Class of problems which are at least as hard as the hardest problems in NP. Problems in NP-hard do not have to be elements of NP, indeed, they may not even be decision problems.

NP-complete Class of problems which contains the hardest problems in NP. Each element of NP-complete has to be an element of NP.

NP-easy At most as hard as NP, but not necessarily in NP, since they may not be decision problems.

NP-equivalent Exactly as difficult as the hardest problems in NP, but not necessarily in NP.

3.4.2 Practice questions

1. **What is the minimum cut problem?**

   In graph theory, a minimum cut of a graph is a cut (a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge) whose cut set has the smallest number of edges (unweighted case) or smallest sum of weights possible. Several algorithms exist to find minimum cuts.

2. **Is there a polynomial time algorithm for this problem?**

   Yes – Krager's algorithm.

3. **Consider the following approach for computing a minimum cut: run Kruskal's algorithm on an unweighted graph and remove the last edge of the resulting minimum spanning tree to define a cut (randomize the selection of the edges among multiple equivalent choices). Show that the probability of this cut being the minimum cut is at least $\frac{1}{n^2}$**

   So let us see why the cut found in each iteration is the minimum cut with probability at least 1/n2. At any stage of Kruskal's algorithm, the vertex set V is partitioned into connected components. The only edges eligible to be added to the tree have their two endpoints in distinct components. The number of edges incident to each component must be at least C, the size of the minimum cut in G (since we could consider a cut that separated this component from the rest of the graph). So if there are k components in the graph, the number of eligible edges is at least kC/2 (each of the k components has at least C edges leading out of it, and we need to compensate for the double-counting of each edge). Since the edges were randomly ordered, the chance that the next eligible edge in the list is from the minimum cut is at most $C/(kC/2) = 2/k$. Thus, with probability at least $1 - 2/k = (k-2)/k$, the choice leaves the minimum cut intact. But now the chance that Kruskal's algorithm leaves the minimum cut intact all the way up to the choice of the last spanning tree edge is at least

   $$\frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times ... \frac{2}{4} \times \frac{1}{4} = \frac{1}{n(n-1)}$$

4. **What is the running time n^2 of the randomized approach that computes the minimum cut with high probability through Kruskal's algorithm?**

   This means that repeating the process O(n2) times and outputting the smallest cut found yields the minimum cut in G with high probability: an O(mn2 log n) algorithm for unweighted minimum cuts. Some further tuning gives the O(n2 log n) minimum cut algorithm, invented by David Karger, which is the fastest known algorithm for this important problem.

5. **What is the balanced cut problem?**

   given a graph with n vertices and a budget b, partition the vertices into two sets S and T such that $|S|, |T| \leq n/3$ and such that there are at most b edges between S and T.

6. **Is there a polynomial time algorithm for this problem?**

   Not unless $P = NP$

7. **What is the knapsack problem?**

   The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

8. **Is there a polynomial time algorithm for this problem?**

   Not unless $P = NP$

9. **What is the subset sum problem and how does it relate to the knapsack problem?**

   In computer science, the subset sum problem is an important problem in complexity theory and cryptography. The problem is this: given a set (or multi-set) of integers, is there a non-empty subset whose sum is zero? For example, given the set $\{-7, -3, -2, 5, 8\}$, the answer is yes because the subset $\{-3, -2, 5\}$ sums to zero. The problem is NP-complete.

10. **What is the class of NP problems?**

    In computational complexity theory, NP is one of the most fundamental complexity classes. The abbreviation NP refers to "nondeterministic polynomial time."

    Intuitively, NP is the set of all decision problems for which the instances where the answer is "yes" have efficiently verifiable proofs of the fact that the answer is indeed "yes". More precisely, these proofs have to be verifiable in polynomial time by a deterministic Turing machine. In an equivalent formal definition, NP is the set of decision problems where the "yes"-instances can be accepted in polynomial time by a non-deterministic Turing machine. The equivalence of the two definitions follows from the fact that an algorithm on such a non-deterministic machine consists of two phases, the first of which consists of a guess about the solution, which is generated in a non-deterministic way, while the second consists of a deterministic algorithm that verifies or rejects the guess as a valid solution to the problem.

    The complexity class P is contained in NP, but NP contains many important problems, the hardest of which are called NP-complete problems, whose solutions are sufficient to deal with any other NP problem in polynomial time. The most important open question in complexity theory, the P = NP problem, asks whether polynomial time algorithms actually exist for NP-complete, and by corollary, all NP problems. It is widely believed that this is not the case.

11. **What is the class of P problems?**

    In computational complexity theory, P, also known as PTIME or DTIME(nO(1)), is one of the most fundamental complexity classes. It contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.

    Cobham's thesis holds that P is the class of computational problems that are "efficiently solvable" or "tractable"; in practice, some problems not known to be in P have practical solutions, and some that are in P do not, but this is a useful rule of thumb.

12. **What is the relation between these two classes of problems?**

    If it turned out that P does not equal NP, it would mean that there are problems in NP (such as NP-complete problems) that are harder to compute than to verify: they could not be solved in polynomial time, but the answer could be verified in polynomial time.

    Consider the subset sum problem. Given an arbitrary set, do any of the set's subsets' elements sum to zero? If such a solution was given, it would be easy to verify. But no known non-polynomial time algorithm can solve it, so it's in NP. It's *quickly checkable* but not *quickly solvable*.

13. **For instance, is P = NP?**

    It is not known, but the evidence is not suggestive of it.

14. **What does it mean that you can reduce a search problem A to a search problem B?**

    Two problems can be reduced to one another if their instances can be easily rephrased as instances of the other. For example, the problem of solving linear equations in a indeterminate x reduces to the problem of solving quadratic equations.

We say that a language is polynomial time reducible to another language if there exists a polynomial time computable function such that you can make the elements of one to the other.

15. **What do you need to do in order to provide such a reduction? (you can provide a drawing to explain your answer)**

16. **What is the class of NP-complete problems?**

    A decision problem L is NP-complete if it is in the set of NP problems and also in the set of NP-hard problems.

17. **What is the class of NP-hard problems?**

    NP-hard problems are problems that are at least as hard as the hardest problems in NP.

18. **When is a problem in the class of co-NP problems?**

    We can define the complexity class **co-NP** as the set of languages L such that the inverse of L is in NP.

    The question of whether NP is closed under complement can be rephrased as whether NP = co-NP.

19. **Provide an example of a co-NP problem.**

    the subset sum problem: given a finite set of integers, is there a non-empty subset that sums to zero? To give a proof of a "yes" instance, one must specify a non-empty subset that does sum to zero. The complementary problem is in co-NP and asks: "given a finite set of integers, does every non-empty subset have a non-zero sum?"

3.5 Examples of reductions between NP complete problems

3.5.1 Practice questions

1. **Show that the general satisfiability (SAT) problem reduces to the 3-SAT problem.**

   The trick is as follows. Expand the ellipses in the following formulation as "$y_1, y_2, y_3...$"

   $$(a_1 \vee a_2 \vee ... \vee a_k)$$

   Is equivalent to the following 3SAT formulation:

   $$(a_1 \vee a_2 \vee y_1)(\neg y_1 \vee a_3 \vee y_2)(y_{k-3} \vee a_{k-1} \vee a_k)$$

   Suppose all the clauses on the right are satisfied. Then at least on of the literals a_1 through a_k must be true, which would in turn force y_2 to be true, and so on, eventually falsifying the last clause.

   Conversly, if $(a_1 \vee a_2 \vee ... \vee a_k)$ is satisfied, then some a_1 must be true. Set $y_1, \cdots, y_{i-2}$ to true and the rest to false. This ensures that the clauses on the right are satisfied.

2. **Show that the 3-SAT problem reduces to the Independent Set problem.**

   - Graph G has a triangle for each clause with vertices labelled by the clauses' literal,s, and has edges between opposite literals.
   - The goal g is set to the number of clauses.

3. **Show that the Independent Set problem reduces to the Vertex cover problem.**

   To reduce independent set to vertex cover we just need to notice that a set of nodes S is a vertex confer of graph $G = VE$ if and only if the remaining nodes, $V - S$, are an independent set of G.

   Therefore, to solve an instance (G, g) of independent set, simply look for a vertex cover of G with $|V| - g$ nodes. If such a vertex cover exists, then take all the nodes NOT in it. If no such vertex cover exists, then G cannot possibly have an indepandt set of size g.

4. **Show that the Independent Set problem reduces to the Clique problem.**

   Define the complement of a graph $G = (V, E)$ to be $\bar{G} = (V, \bar{E})$, where $\bar{E}$ contains precisely those unordered pairs of vertices that are not in E. Then, a set of nodes S is an independent set of G if and only if S is a clique of $\bar{G}$. TO paraphrase, these nodes have no edges between them if and only if they have all possible edges between them in $\bar{G}$

5. **Show that the Circuit SAT problem reduces to the SAT problem.**

   Notice that SAT asks for a satisfying truth assignment of a circuit that has this simple structure: a bunch of AND gates at the top join the clauses, and the result of this big AND is the oupt. Each clause is the OR of the literals.

6. **What is the importance of this reduction?**

   Naturally, for any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs. If the polynomial algorithm in question solves a problem that requires a yes or no answer (as is the situation with C: "Does S encode a solution to the instance encoded by I?"), then this answer is given at the output gate.

   We conclude that, given any instance I of problem A, we can construct in polynomial time a circuit whose known inputs are the bits of I, and whose unknown inputs are the bits of S, such that the output is true if and only if the unknown inputs spell a solution S of I . In other words, the satisfying truth assignments to the unknown inputs of the circuit are in one-to-one correspondence with the solutions of instance I of A. The reduction is complete.

3.6 Intelligent Exhaustive Search, Intro to Approximation Algorithms

3.6.1 Practice questions

1. **What is the idea behind backtracking search in order to solve NP-complete problems?**

   Backtracking is based on the observation that it is often possible to reject a solution by looking at just a small portion of it.

2. **Describe a general framework for backtracking search.**

   A backtracking algorithm requires a test that looks at a subproblem and quickly declares one of three outcomes:

   1. Failure: the subproblem has no solution.
   2. Success: a solution to the subproblem is found.
   3. Uncertainty.

3. **Describe the application of backtracking search to the satisfiability problem, i.e., which sub-problems are expanded at each iteration and which branching variable is considered?**

For a Boolean clause $\phi(a, b, ..., c)$, branch on any variable, and seek clauses that are violated by assignments of `true` or `false`. It makes sense to choose the subproblem that contains the smallest clause and to then branch on a variable in that clause. If this clause happens to be a singleton, then at least one of the resulting branches will be terminated.

4. **Describe the general branch-and-bound approach for optimization problems.**

   As before, we will deal with partial solutions, each of which represents a subproblem, namely, what is the (cost of the) best way to complete this solution? And as before, we need a basis for eliminating partial solutions, since there is no other source of efficiency in our method. To reject a subproblem, we must be certain that its cost exceeds that of some other solution we have already encountered. But its exact cost is unknown to us and is generally not efficiently computable. So instead we use a quick lower bound on this cost.

5. **Describe an application of branch-and-bound to the traveling salesman problem.**

6. **How is the approximation ratio of an approximation algorithm computed?**

   Suppose now that we have an algorithm A for our problem which, given an instance I, returns a solution with value $A(I)$. The *approximation ratio* of algorithm A is defined to be:

$$\alpha_A = \max_i \frac{A(I)}{\text{OPT}(I)}$$

   In other words, α_A measures by the factor by which the output of the algorithm A exceeds the optimal solution, on the worst-case input. The approximation ration can also be defined for maximization problems, such as independent set, in the same way – except that to get a number larger than 1 we take the reciprocal.

7. **Provide an approximation algorithm for the vertex cover problem. What approximation ratio does it achieve?**

3.7 Approximation Algorithms, Local Search Heuristics

3.7.1 Chapters 9.2 from DPV: Approximation algorithms

- For every instance I, let us denote $\text{OPT}(I)$ the value (benefit or cost) of the optimum solution. It makes the math simpler to *assume that this is always a positive energy.*
- For any instance I of size n, we showed that this greedy algorithm is guaranteed to quickly find a set cover of cardinality of at most $\text{OPT}(I) \log n$.
  - This factor is known as the approximation guarantee of the algorithm.
- More generally, consider any minimization problem. Suppose now that we have an algorithm A for our problem which, given instance I, returns a solution with value $A(I)$. This *approximation ratio* of algorithm A is defined to be:

$$\alpha_A = \max_i \frac{A(I)}{\text{OPT}(I)}$$

- In other words, α_A measures by the factor by which the output of algorithm A exceeds the optimal solution, on the worst-case input.
  - The approximation ratio can also be defined for maximization problems, such as independent set, in the same way – except that to a number larger than one we take the reciprocal.

Clustering

- We have some data that we want to divide into groups.
  - It is useful to define "distances" between these data points – numbers that capture how close or far they are from one another.
  - Assume that we have such distances and that they satisfy the usual *metric* properties:
    1. $\forall x, yd(x, y) \geq 0$
    2. $d(x, y) = 0$ if an only if $x = y$.
    3. $d(x, y) = d(y, x)$
    4. (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$
- We would like to partition the data points into groups that are compact in the sense of having small diameter.

  k-CLUSTER

  Input: Points $X = \{x_1, \cdots, x_n\}$ with underlying metric $d(\cdot, \cdot)$; integer k

  Output: A partition of the points into k clusters $C_1, \cdots C_k$.

  Goal: Minimize the diameter of the clusters,

  $\max_j \max_{x_a, x_b \in C_j} d(x_a, x_b)$

- One way to visualize this task is to imagine n points in space, which are to be convered by k spheres of equal size. What is the smallest possible diameter of the sphere?

TSP

- if the distances between cities satisfy the metric properties, then there is an algorithm that outputs a tour of length at most 1.5 times optimal

$$\text{TSP cost} \leq \text{cost of this path} \leq \text{MST cost}$$

- We somehow need to use the MST to build a traveling salesman tour. If we can use each edge *twice*, then by following the shape of the MST, we end up with a tour that visits all the cities, some of them more than once.
  - Therefore, this tour has a length of at most twice the MST cost, which as we've already seen is at most twice the TSP cost.
- This is the result we wanted, but we aren't quite done because our tour visits come cities multiple times and is therefore not legal.
  - To fix the problem, the tour should simply skip any city it is about to revisit, and instead move directly the *new* city in its list.

3.7.2 Chapter 9.3 from DPV: Local search heuristics

Graph partitioning

- The problem of graph partitioning arises in a diversity of application.
  - We saw a special case in Balanced Cut.

  Graph partitioning

  Input: An undirected graph $G = (V, E)$ with nonnegative edge weights; a real number $\alpha \in (o, 1/2]$

  Output: A partition of the vertices into two groups, A and B, each of size at least $\alpha|V|$.

  Goal: Minimize the capacity of the cut (A, B)

- This variant is NP hadd.
- Focus on the special case $\alpha = 1/2$, in which A and B are forced to contain exactly half the vertices. The apparent loss of generality is purely cosmetic, as Graph partitioning reduces to this particular case.
- We need to decide upon a neighborhood structure for our problem, and there is one one obvious way.
  - Let (A, B) with $|A| = |B|$ be a candidate solution; we will define its neighbors to be all solutions obtainable by swapping one pair of vertices across the cut, that is, all solution of the form $(A - \{a\} + \{b\}, B - \{b\} + \{a\})$ where $a \in A$ and $b \in B$.
- There is still a lot of room for improve in terms of the quality of the solution produced. The search space includes some local optima that are quite far from the global solution.

3.7.3 Practice questions

1. **When does a distance function satisfy metric properties?**

   1. $\forall x, y d(x, y) \geq 0$
   2. $d(x, y) = 0$ if an only if $x = y$.
   3. $d(x, y) = d(y, x)$
   4. (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$

2. **What is the k-clustering NP-complete problem?**

   k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

3. **Provide a simple approximation scheme for the k-clustering problem.**

   The idea is to pick k of the data points as cluster centers and to then assign each of the remaining points to the center closest to it, thus creating k clusters. The centers are picked one at a time, using an intuitive rule: always pick the next center to be as far as possible from the centers chosen so far

   ```
   Pick any point u_1 in X as the first cluster center
   for i = 2 to k:
       let u_i be the point in X that is farthest from u_1, ..., u_{i - 1}
   create k clusters: C_i = {all x in X whose closest center is in u_i}
   ```

4. **What is the approximation ratio that it achieves?**

   the resulting diameter is guaranteed to be at most twice optimal.

5. **Prove it.**

   Let $x \in X$ be the point farthest from $u_1, ..., u_k$ (in other words the next center we could have chose, if we wanted $k+1$ of them), and let r be its distance to its closest center. Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

6. **Provide an approximation algorithm for the traveling salesman problem given that the underlying graph has edge weights that satisfy metric properties.**

   1. Construct a minimum spanning tree T for G.
   2. Duplicate all edges of T. That is, wherever there is an edge from u to v, add a second edge from v to u. This gives us an Eulerian graph H.
   3. Find an Eulerian circuit C in H. Clearly, its span is twice the span of the tree.
   4. Convert the Eulerian circuit C of H into a Hamiltonian cycle of G in the following way: walk along C, and each time you are about to come into an already visited vertex, skip it and try to go to the next one (along C).

7. **What is the approximation ratio for this algorithm? Prove it.**

   if the distances between cities satisfy the metric properties, then there is an algorithm that outputs a tour of length at most 1.5 times optimal.

8. **Describe the general local search framework.**

   By its incremental process of introducing small mutations, trying them out, and keeping them if they work well. This paradigm is called *local search* and can be applied to any optimization task.

   ```
   let s be any initial solution
   while there is some solution s' in the neighborhood of s
       for which cost(s') < cost(s): replace s by s'
   return s
   ```

   On each iteration, the current solution is replaced by a better one close to it, in its *neighborhood*. This neighborhood structure is something we impose upon the problem and is the central designing decision in local search.

9. **Provide a local search approach for the traveling salesman problem.**

   Assume we have all inter point distances between n cities, giving a search space of $(n-1)!$ different tours. What is a good notion of neighborhood?

   The most obvious notion is to consider two tours as being close if they different in just a few edges, but they cannot differ in just one. We define the *2-change* neighbor of tour s as being the set of tours that can be obtained by removing two edges of s and then putting in two other edges.

   We now have a well-defined local search procedure. How does it measure up under our two standard criteria for algorithms – what is its overall running time and does it always return the best solution?

   Neither of those questions have a satisfactory answer. Each iteration is fast, because a tour has only $O(n^2)$ neighbors. However, it is not clear how many iterations will be needed: perhaps there might be an exponential number. Likewise, all we can easily say about the final tour is that it is *locally optimal*

– that is, it is superior to the tours in its immediate neighborhood. There might be better solutions further away.

To combat this, 3-change, 4-change ...

10. **Provide a local search strategy for the graph partitioning problem.**

    Let (A, B) with $|A| = |B|$ be a candidate solution; we will define its neighbors to be all solutions obtainable by swapping one pair of vertices across the cut, that is, all solution of the form $(A - \{a\} + \{b\}, B - \{b\} + \{a\})$ where $a \in A$ and $b \in B$.

11. **How can you deal with local optimal in the context of local search?**

    - Randomization and restarts
    - Simulated annealing