CS 341 Term Notes

Shale Craig
sakcraig@uwaterloo.ca

October 28, 2013

Contents

Preface

This course was taken in Winter 2013 at the University of Waterloo under Timothy Chan.

These notes come with no guarantee of correctness, veracity, nor semblance of course materials. Go to class, your profs are right experts.

Compiled with love by:

- Shale Craig - `http://shalecraig.com`

Please submit corrections, omissions, and requests for change to `http://github.com/shalecraig/Notes`

Chapter 1

Introduction

1.1 Bentley's Problem

Find the consecutive sub-array B of a given array A with the maximal value.

i.e: $A = [1, 2, -3, 4, -4, 0]$, $B = [1, 2, -3, 4]$

$O(n)$-time solution exists:

```
max_subarray(A):
    max_ending_here = max_so_far = 0
    for x in A:
        max_ending_here = max(0, max_ending_here + x)
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far
```

Basically, we keep track of the maximal subarray ending "here" (including the empty array), and then compare that to the max so far.

1.2 The 3SUM problem

Given a set of integers A and an integer k, find 3 different numbers in A that sum to k.

i.e. $A = [1, 2, -3, 4, -4, 0]$, $k = 3$. $2 + 4 - 3 = k$

$O(n^2)$-time solution exists:

```
sort(A);
for i=0 to n-3 do
    a = A[i];
    j = i+1;
    l = n-1;
```

```
        while (j<l) do
            b = A[j];
            c = A[l];
            if (a+b+c == k) then
                output a, b, c;
                exit;
            else if (a+b+c > k) then
                l = l - 1;
            else
                j = j + 1;
            end
        end
    end
```

Basically, we sort values in A, pick one value of a, then try to find values of $A[j]$ and $A[l]$ that equal k. We change the values of l and j according to the value of the sum compared to k.

1.3 Math review

1.3.1 Asymptotic notation

We use O, o, Ω, ω, and Θ to denote the runtimes of different algorithms as the input sizes tend to ∞. This isn't perfect for talking about small n, but performing well for small n is less an algorithms task and more an algebra task.

Here's a table:

Symbol	Condition
$f(n) \in O(g(n))$	$\|f(n)\| \leq kg(n)$ for some positive k
$f(n) \in \Omega(g(n))$	$\|f(n)\| \geq kg(n)$ for some positive k
$f(n) \in \Theta(g(n))$	$k_1 g(n) \geq \|f(n)\| \geq k_2 g(n)$ for some positive k_1, k_2
$f(n) \in o(g(n))$	$\|f(n)\| \leq \epsilon g(n)$ for all positive ϵ
$f(n) \in \omega(g(n))$	$\|f(n)\| \geq \epsilon g(n)$ for all positive ϵ

1.4 Summations

Summations are useful, and we only use a few in this course.

Here they are:

$$\sum_{i=0}^{n} i = \frac{n^2+n}{2}$$
$$\sum_{i=0}^{n} i^2 = \frac{2n^3+3n^2+n}{2}$$
$$\sum_{i=0}^{n} a^i = \frac{1-a^{n-1}}{1-a}$$
$$\sum_{i=0}^{n} \frac{1}{n} = O(\log n + \gamma)$$

Note: I'm not sure this is complete.

1.5 Recurrences [Sec 4.4,4.5,4.3]

Recurrence relationships are often found in recursive code. We probably want to solve these to see the asymptotic behaviour of our algorithm.

1.5.1 The Recursion Tree Method

The basic idea of this approach is we break down values as they go down to child nodes, and determine a function for the number of child nodes at each level, and another the cost of each node. By summing these together, we get the overall cost of the function with respect to n.

i.e.

$$f(n) = \begin{cases} 12f\left(\frac{n}{12}\right) + 14n & n \geq 100 \\ n & \text{otherwise} \end{cases}$$

(Drawing not provided - if you supply one, I'll add it.)

By drawing this out, we see:

1. In level i, there are 12^i nodes.
2. In level i, our cost is $14\frac{n}{12^i}$ per node.
3. The tree height is (approximately) $\log_{12} n$.

Based on this, we get the summation:

$$\begin{aligned} f(n) &= \sum_{i=0}^{\log_{12} n} (12^i)(14\frac{n}{12^i}) \\ &= \sum_{i=0}^{\log_{12} n} 14n \\ &= 14n \log_{12} n \end{aligned}$$

i.e we have $f(n) = 14n \log_{12} n$

In this example I don't count the leaf nodes, but probably should.

TODO: evaluate the leaf nodes.

1.5.2 Master Method

The master method is often referred to as a "cookbook" or a "lookup" based method. We basically know that $f(n)$ is of a certain form, we can substitute it into the answer.

Algorithms expressed in the form of $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ can be solved using the master theorem. We didn't talk about other forms in class.

We define $c = \log_b a$, and require that $\epsilon > 0$.

There are three cases that we can apply it in:

1. If $f(n) \in O(n^{c-\epsilon})$, it follows that $T(n) \in \Theta(n^c)$.
2. If $f(n) \in \Theta(n^c \log^k n)$ for $k \geq 0$, it follows that $T(n) \in \Theta(n^c \log^{k+1} n)$.
3. If $f(n) \in \Omega(n^{c+\epsilon})$, it follows that $T(n) \in \Theta(n^c)$.

1.5.3 Guess-And-Check Method

In this method, we guess a recursion and substitute into the recurrence to check for veracity. If our guess is incorrect we start again, using what we learned to re-build our guess.

i.e. Guess $T(n) = c_1 n + c_2$:

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + T(n/8) + n \\ T(n) &= c_1 n + c_2 \\ T(n/2) + T(n/4) + T(n/8) + n &= c_1 n \frac{4+2+1}{8} + 3c_2 + n \\ &= \frac{7nc_1}{8} + 3c_2 \end{aligned}$$

From this, we get that $\frac{7c_1}{8} + 1 = c_1$, so $c_1 = -8$. Similarily, $c_2 = 0$.

Chapter 2

Divide and Conquer

Divide and conquer algorithms break up problems into smaller sections that they solve recursively. They can take advantage of the fact they're solving smaller problems to provide a better runtime than naive solutions.

They generally consist of three steps:

1. Break up input to smaller parts.
2. Solve each part.
3. Merge the solutions together.

2.1 The Maxima Problem

TODO: It was this problem.

Given a set of points P, find the set of points M that are a maxima of P. A maximal point $p \in M$ is one where there is no other point r in P where $r.x > p.x$ and $r.y > p.y$.

Conceptually, we can break the set of points P into P_1 and P_2, each roughly half the size of the other. Then we can solve to get M_1 and M_2. We can merge M_1 and M_2 together, and return that.

We need to remember to consider the base case in writing the solution.

```
def maxima(P):
    if len(P) <= 2:
        return P
    sortByX(P)
    ->P1 is the 'top-left half'
    P1 = P[0 : len(P)/2]
    P2 = P[len(P)/2+1 : -1]
    M1 = maxima(P1)
    M2 = maxima(P2)
    i1 = 0
```

```
    firstM2 = M2[0]
    M = []
    while (M1[i1].y > firstM2.y):
        M.addOne(M1[i1])
    M.addAll(M2)
    return M
```

This algorithm takes $T(n) = 2T(n/2) + n \log n$ worst-case time. We can use the master method to solve this.

By pre-sorting all the points instead of doing it in every reduction, we only need to do it once, so we can say that $T(n) = 2T(n/2) + n$.

$a = 2$, $b = 2$, $f(n) = n$, $c = \log_b a = 1$.

We know $f(n) \in \Theta(n^1 \log^0 n)$, so rule 2 must apply for $k = 0$. Thus, $T(n) \in \Theta(n^1 \log n)$.

2.2 The Closest Pair Problem (Shamos' Algorithm)

Given a set P of points, find the distance betwwen the closest pair of points $\langle p, q \rangle$ in P.

Conceptually, we can solve this problem using a similar (but different!) approach than before. We will split the points into left and right halves and solve them independently. We will then merge them, and only return the best answer between the two. All that remains is to evaluate pairs of points that cross the separating boundary between the two halves, which can be done by creating a "moving elevator" that goes up to iterate through all points that are possibly valid inside this elevator.

```
def getClosestPair(Px, Py):
    if len(P) == 2
        return dist(P[0], P[1])
    Pxl = Px[0 : len(Px)/2]
    Pxr = Px[len(Px)/2+1 : -1]
    Pyl = Py[p is in Pxl]
    Pyl = Py[p is in Pyl]
    bestLeft = getClosestPair(Pxl, Pyl)
    bestRight = getClosestPair(Pxr, Pyr)
    best = min(bestLeft, bestRight)
    windowRight = Pl[-1].x + best
    windowLeft = Pr[0].x - best
    window = Pyl[p.x > windowLeft] + Pyr[p.x < windowRight]
    for (i=0...len(window)-1):
        k = i+1
        while k < len(window)-1 and window[k].y-window[i].y < best:
            if dist(window[k], window[i]) < best:
                best = dist(window[k], window[i])
            k = k+1
    return best
```

We take P_x and P_y as input (points sorted according to x and y). We can express the runtime of this algorithm as $T(n) = 2T(n/2) + 6n\log(n) + 3$. Using the master method, we can solve the recurrence as $T(n) = \Theta(n\log^2 n)$.

2.3 Multiplication of Large Numbers (Karatsuba and Ofman's Algorithm)

This can be found at [KT, Sec 5.5 or BB, Sec 7.1]

We want to multiply integers A and B (expressed as n bits) more efficiently than the grade-school "multiply-everything" approach.

Conceptually, Karatsuba used a shortcut where instead of multiplying A with B directly, he performed three multiplications of smaller size, which is a bit faster. If you use the algorithm recursively, it's asymptotically faster.

Given integers x and y we want to compute xy. Choose a base B, and $m = \log(x)/2$

$$\begin{aligned} x &= x_1B^m + x_2 \\ y &= y_1B^m + y_2 \\ xy &= z_2B^{2m} + z_0 + z_1B^m \\ z_2 &= x_1y_1 \\ z_0 &= x_0y_0 \\ z_1 &= x_1y_0 + x_0y_1 \\ &= (x_1 + x_0)(y_1 + y_0) - z_2 - z_0 \end{aligned}$$

It takes $T(n) = 3T(n/2) + cn + d$ (c and d are constants) to calculate the value of AB. Solved by the master theorem, this becomes $T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^1.58)$.

2.4 Matrix Multiplication (Strassen's Algorithm)

This can be found at [CLRS, Sec 4.2]

Given matrices A and B, find the product C of A and B.

Conceptually, Strassen realized that (like Karatsuba) you can do these multiplications in fewer operations that what is explicitly obvious.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$
$$B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$
$$C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$
$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$
$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$
$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

If we define an intermediate matrix M, we can calculate C in 7 multiplications instead of the 8 above.

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$
$$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$$
$$M_4 = A_{2,2}(B_{2,1} - B_{1,1})$$
$$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$$
$$M_6 = (A_{2,1} + A_{1,1})(B_{1,1} + B_{1,2})$$
$$M_7 = (A_{1,2} + A_{2,2})(B_{2,1} + B_{2,2})$$

We can now calculate C in terms of M:

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$
$$C_{1,2} = M_3 + M_5$$
$$C_{2,1} = M_2 + M_4$$
$$C_{2,2} = M_1 + M_2 + M_3 + M_6$$

Since we calculate this with 7 multiplications of sub-problems of size $n/2$, this will take us $T(n) = 7T(n/2)+in$ (i is for the number of additions). By master method, we arrive at $T(n) = \Theta(n^{\log_2(7)})$ as the running time of the algorithm.

2.5 Subsection [Sec 9.2-9.3]

Basic problem statement: Find the ith element of an array A.

2.5.1 Randomized Quickselect

Basic idea is to pick a random element, seperate the elements into subsets that are larger and smaller than the element, then recurse into the subset.

Best runtime is $O(1)$, but worst runtime is $O(n^2)$ since it is random. Average runtime is $O(n)$, but we can do better.

2.5.2 Blum, Floyd, Pratt, Rivest, and Tarjan's Algorithm

Also called "Median of Five".

The way this works is pretty cool. We pick the median by finding the median of groups of five elements (recursive call 1). Then we use the knowledge of the value of the median to do a quickselect-style recursion (call 2) into the half that contains our element.

Through this algorithm, finding the median takes $O(n)$ time (awesome), and the overall algorithm takes $O(n)$ time.

Pseudocode:

```
select(A, k):
    if (|A| == 1):
        return A[0]
    split A into groups G, each of 5 elements
    create B = median of each g in G
    x = select(B, |B|/2)
    L = {a < x}
    R = {a > x}
    if (l <= k):
        return select(L, k)
    else
        return select(R, k-|L|)
```

Chapter 3

Greedy Algorithms

Put simply, greedy algorithms solve decision problems in short amounts of time. They don't backtrack, and rely on a heuristic to make their decisions. They don't work in all circumstances, but are speedy in some. Greedy is (sometimes) good.

We (almost always) need to prove greedy algorithms to be correct. Proving them wrong is not very hard.

3.1 Coin Changing

Problem definition: You are asked to make \$1.34 into the fewest coins possible. Given a dollar value and a set of coin values, what are the minimum coins that make up that value?

Simple algorithm that everybody uses: pick the highest value coin that is less than the difference between the sum of coins you have chosen and the dollar value remaining.

3.2 Disjoint intervals

Problem definition: You are given a set A of intervals where some of them overlap. Choose the largest subset S of A such that elements in S do not overlap.

Here are a bunch of heuristics:

- Pick the shortest range. This fails.
- Pick the range that conflicts with the fewest ranges. This fails.
- Pick the range that finishes first. This succeeds.

3.3 Fractional knapsack

Problem definition: You are given a set of items I and a backpack size b. Each item has a weight $i.w$ and a size $i.s$. You are allowed to choose fractional items. Choose backpack items such that the value of your

bag is maximized while not being oversized.

Correct Heuristic: While there is room in your knapsack, pick the remaining item that has the highest weight/size ratio, and put it in your knapsack. If it doesn't fit, put a fraction of it in your knapsack.

3.4 Stable marriage

Problem definition: You are given a set of men and women, and lists of their preferences for each other. Provide a "matching" of them such that no two people prefer each other over their "match".

Correct Heuristic: While there are unmatched men, send an unmatched man to the next woman on his list that he hasn't proposed to yet. If that woman is already paired, she chooses between the two men, and the less preferred suitor becomes unmatched.

Proof of correctness:

- Claim 1: A woman exists that he hasn't proposed to yet.

  Some women are unmatched. All women previously proposed to by him are matched. Therefore, some matched woman hasn't been proposed to by him.

- Claim 2: The resultant matching is stable.

  (By way of contradiction) Suppose we have matched pairs $\langle c, e\rangle$ and $\langle c', e'\rangle$ where c prefers e' over e and e' prefers c over c'. Then we must have had that e' made an offer to c before c'. Later on, e' made an offer to c, and c preferred e' over e. Since that is impossible, the solution must be stable.

Chapter 4

Dynamic Programming

Basically, Dynamic Programming is solving smaller problems first before solving larger problems, then using the smaller problems as building blocks for the larger problems.

Problems where recursive functions are called repetitively with the same arguments can be transformed to a dynamic programming problem. We take the recursive formula, and build up to it (instead of down).

We use an array to "cache" results of recursive functions, and effectively loop our code from "early parameters" to higher parameters. We work towards our solution.

In solutions, we need to specify:

1. Recursive formula
2. Initial values
3. Iteration order
4. A sub-problem the solution is composed of
5. The answer in terms of an array

4.1 Binomial coefficients

We want to calculate $\binom{n}{k}$ in the fewest number of calculations possible.

Define our recursive formula:

$$C(n,k) = \begin{cases} 1 & n = k \text{ or } k = 0 \\ C(n-1,k-1) + C(n-1,k) & \text{otherwise} \end{cases}$$

Implemented in code, this would look like:

```
def C(i, j):
    if (i == j || j == 0):
        return 1
```

```
    return C(i-1, j-1) + C(i-1, j)
```

This is a valid algorithm but in this implementation, the same values of C will be calculated many times over and over.

If we make C an array instead of a function and determine an iteration order, we only need to calculate values of C once.

For the iteration order, we can iterate from $i = 0...n$, and for each of those we can iterate $j = 0...k$.

We also need to determine initial values of C. In this case we have $C[i, 0] = 1$, $C[0, i] = 1$.

The code would look like this:

```
def C(n, k):
    C = new array[n,k]
    for i = 0...n:
        C[i, 0] = 1
    for i = 0...min(n, k):
        C[0, i] = 1
    for i = 0...n:
        for j=0...k:
            C[i, j] = C[i-1, j-1] + C[i-1, j]
    return C[n, k]
```

The array C is nk in size. Every element is calculated once, and takes $O(1)$ time to calculate. Thus, this algorithm takes $O(nk)$ time overall.

4.2 Coin Changing

Given a set of coins N and a target value w, what is the minimum number of coins in N summing to w.

Sub-problem: We define $C[i, j]$ as the minimum number of coins in $\{N_0...N_i\}$ that sum to j.

Solution: Return $C[|N|, w]$, the minimum number of coins in N that sum to w.

Base cases:

$$C[i, 0] = 0$$
$$C[0, j] = \infty$$

Recursive formula:

$$C[i, j] = \begin{cases} \min(C[i-1, j], C[i-1, j-1] + 1) & j \geq D_i \\ C[i-1, j] & \text{otherwise} \end{cases}$$

Iteration order: Iterate through all numbers $i = 1...n$, for each of those iterate through $j = 1...w$.

Code:

```
def minGrouping(D, w):
    for i = 0...n:
        C[i, 0] = 0
    for j = 1...w:
        C[0, j] = infinity
    for i = 1...n:
        for j=1...w:
            if j < D[i]:
                C[i, j] = min(C[i-1, j], C[i-1, j-1] + 1)
            else:
                C[i, j] = C[i-1, j]
    return C[|D|, w]
```

The array C takes $O(w|N|)$ space. Each element in C is calculated exactly once and takes $O(1)$ time to calculate, so the entire algorithm takes $O(w|N|)$ time total.

4.3 0-1 Knapsack

Problem definition: Given a set of n items with values V and weights W, maximize the total value without the weight going over w.

Subproblem definition: $C[i, j]$ is the highest value possible with the first i elements, at a weight limit of j.

Answer: $C[n, w]$

Base cases: $C[0, j] = C[i, 0] = 0$

Recursive formula:

$$C[i,j] = \begin{cases} \max(C[i-1,j], C[i-1,j-W_i] + V_i) & j \geq w_i \\ C[i-1,j] & \text{otherwise} \end{cases}$$

Code:

```
def C(V, W, w):
    n = |V|
    for i=0...n:
        C[i, 0] = 0
    for j=0...w:
        C[0, j] = 0
    for i=1...n:
        for j=1...w:
            if (j >= W[i]):
                C[i, j] = max(C[i-1, j], C[i-1, j-W[i]] + V[i])
            else:
                C[i, j] = C[i-1, j]
    return C[n, w]
```

4.4 Longest common subsequence

Reference for this can be found at [Sec 15.4]

Problem statement: Given strings A and B, return the longest subsequence of characters that appear in both strings.

i.e. A = "ALGORITHM", B = "LOGARITHM".

The longest common substring is "LORITHM".

Subproblem: $C[i, j]$ is the length of the longest substring of the first i characters of A and the first j characters of B.

Answer: $C[m, n]$, where $m = |A|$, $n = |B|$.

Base case: $C[i, 0] = C[0, j] = 0$ ($i = 0...m$, $j = 0...n$).

Recursive formula:

$$C[i,j] = \begin{cases} \max(C[i-1,j], C[i,j-1]) & A_i \neq B_j \\ \max(C[i-1,j], C[i,j-1], C[i-1,j-1]+1) & A_i = B_j \end{cases}$$

i.e. If the characters at i and j don't match, use the max of $\langle i-1, j\rangle$ and $\langle i, j-1\rangle$. If they do match, use the max of $\langle i-1, j\rangle$, $\langle i, j-1\rangle$, and $\langle i-1, j-1\rangle + 1$.

Code:

```
def LCS(A, B):
    n = |A|
    m = |B|
    C = []
    for i = 0...m:
        C[i, 0] = 0
    for j = 0...n:
        C[0, j] = 0
    for i = 0...m:
        for j = 0...n:
            if (A[i] == B[j]):
                C[i, j] = max(C[i-1, j], C[i, j-1], C[i-1, j-1]+1)
            else:
                C[i, j] = max(C[i-1, j], C[i, j-1])
    return C[n, m]
```

4.5 Minimum-length triangulation

Reference for this can be found at [CLR (1st ed.), Sec 16.4]

Given a convex polygon P with n vertices, find the triangulation with minimum length of chords.

Brute force takes $\Omega\left(\frac{4^n}{n^{3/2}}\right)$ possible triangulations. DP does better than this.

Sub-problems: $(1 \leq i \leq j \leq n)$

$C[i, j] =$ length of the minimum-length triangulation of the sub-polygon with the vertices running from i to j (then l

Answer: $C[1, n]$

Base-cases: $C[i, i+1] = C[i, i+2] = 0$

Recursive formula:

$$C[i, j] = \min_{k \in \{i+1 \ldots j-1\}} \{C[i, k] + C[k, j] + d(V_i, V_k) + d(V_j, V_k)$$

i.e. $C[i, j]$ is the minimum length for any way you split the polygon, which is the cost of the left half plus the right half plus the distance of the halves.

Iteration order was not written down, but I'm sure it was complicated.

Chapter 5

Graph Algorithms

5.1 BFS/DFS

Skipped writing this one, it's in my notes, and seems to be pretty trivial. The only thing to watch out for is to mark vertices as discovered because "this is a graph" and not a tree.

5.2 Connectedness

Skipped this class?

5.3 Cycle Detection

Skipped this class?

5.4 2-Coloring

Skipped this class?

5.5 Topological Sorting

Problem description: For a directed graph $G = (V, E)$, return a vertex order such that for all edges, the "from vertex" appears before the "to vertex".

Conceptually, we visit all vertices using BFS. If we reach any vertex we have already touched, then we know there is a cycle. Otherwise, we are able to put elements into L in the order they are explored.

```
def topologicalSort(V, E):
```

```
    L = []
    while there are unmarked vertices:
        n = an unmarked vertex
        visit(n)
    return L
def visit(n):
    if n has a temporary mark, we have found a cycle
    if n is not marked:
        mark n temporarily
        for each node m with an edge from n to m:
            visit(m)
        mark n permanently
        add n to L
```

5.6 Strongly Connected Components (Kosaraju and Sharir's Algorithm)

Given a directed graph $G = (V, E)$, partion V into components such that for all u, v in some component, there exists a path from u to v and there exists a path from v to u.

i.e. we want to simplify/condense a directed graph into a DAG.

1. Run DFS(G), number vertices in the order that they finish.
2. Form G^T (the transpose of G - arrows are reversed).
3. Run DFS(G^T), preferring higher-numbered vertices.
4. Return the vertices from each DFS tree as components.

Proof: Take a DFS tree T of G^T. Let r be a root, u be any Vertex of T.

1. r has a higher number than u (if not, pick u first).
2. There exists a path $u \to r$ in G, since there exists a path $r \to u$ in G^T.
3. There exists a path $r \to u$ in G.
   By way of contradiction, assume that there is no such path:
   If u was discovered first, then r finished before u, which means they would've been ordered differently.
   If r was discovered first, $r \not\to u$ by assumption, r finished before u.
4. For all u, v in T, there exists a path $u \leftrightarrow v$, since there exist paths $u \leftrightarrow r$ and $r \leftrightarrow v$.
5. There exists a path $u \leftrightarrow v$ in G implies $\langle u, v \rangle$ are in the same tree.

5.7 Minimum Spanning Trees

A spanning tree is a subgraph of a graph that is a tree and connects to all vertices.

A minimum spanning tree is a subgraph of a graph that has the least (or equal to the lowest) sum of weighted edges in the spanning tree compared to all other spanning trees of the graph.

5.7.1 Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm to find the minimum spanning tree of a graph.

Basic idea of algorithm is as follows:

- Create a set of trees F from all vertices in V.
- Create a set of edges S from all edges in E.
- While S is nonempty and F is not spanning:
  - Remove an edge e with minimum weight from S.
  - If e connects two trees in F, remove the two trees from F, join them by e and add that to F.
- Return F.

5.7.2 Prim's Algorithm

Prim's Algorithm is a greedy algorithm to find the minimum spanning tree of a graph.

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

- Initialize $V_{seen} = \{x\}$ (x) is an arbitrary node.
- Initialize $E_{mst} = \{\}$.
- While $V_{seen}! = V$:
  - Choose an edge $\langle u, v\rangle$ with minimal weight so u is in V_{seen} and v is not.
  - Add v to V_{seen} and $\langle u, v\rangle$ to E_{mst}.
- Return E_{mst}.

If we set $n = |V|$ and $m = |E|$, we can get it to be $O(n^2 + m)$ using no fast data structures, $O(m \log n)$ using a heap, and $O(n \log n + m)$ using a Fibonacci Heap.

5.8 Shortest Paths

Given a weighted directed graph $G = (V, E)$, find a path P from s to t such that we minimize the "weight" of the path.

5.8.1 DP for DAG case

We can solve this problem using DP.

Sub-Problem: $\delta[v] = w($ shortest path $s \to v)$.

Answer: $\delta[t]$

Base case: $\delta[s] = 0$

Recursive formula:

$$\delta[i] = \min_{u \in V : \langle u,v \rangle \in E} \{\delta[u] + w(u, v)\}$$

Iteration order: Topological sort order.

Analysis: $O(n + \sum_{v \in V} \text{inDeg}(v)) = O(m + n)$ time.

5.8.2 Dijkstra's Algorithm

Dijkstra's Algorithm assumes a positive weight for all edges in the graph.

We compute d[v] = shortest path weight from s to v

```
// Assumes positive weight
S = {s}
delta[s] = 0
while S != V:
    pick edge(u,v) with u in S, v in V-S that minimizes delta[u] + w(u, v)
    insert v to S
    delta[v] = delta[u] + w(u, v)
```

In fact, Dijkstra's algorithm is similar to Prim's algorithm, but the selection criteria changes from pick the shortest "edge" from a u to a v to pick the shortest "edge + delta[v]".

It runs in $O(n \log n + m)$ time, where m is the number of edges.

Correctness: Claim: If $\langle u, v \rangle$ is the edge with the shortest $delta[u] + w(u, v)$, then the shortest weight from s to v is equal to $delta[u] + w(u, v)$.

Proof: There is a path from s to v of weight $delta[u] + w(u, v)$ Consider another path P from s to v. P contains edge $\langle u', v' \rangle$ from S to $S - V$. Then $w(p) \geq delta[u'] + w(u', v') + 0 \geq delta[u] + w(u, v)$.

5.8.3 All-Pairs Shortest Paths

Problem statement: For every pair of vertices $\langle u, v \rangle$, find the shortest path u to v.

Dijkstra Repeatedly

Runs in $O(n(n \log n + m))$, but assumes all edges are positive.

DP #1

Subproblems $D[i, j, k]$ is the minimum weight over all paths from i to j of length $\leq k$.

Answers: $D[i, j, n-1]$ for all i, j

Base Case:

$$D[i, j, 0] = \begin{cases} 0 & l = j \\ \infty & \text{otherwise} \end{cases}$$

Recursive Formula:

$$D[i, j, k] = \min_{\ell \in \{1 \ldots n\}} D[i, \ell, k-1] + w(i, j)$$

Evaluate in increasing k.

```
for i = 1...n
    for j = 1...n
        for k = 0...n-1
            ...
```

There are $\Theta(n^3)$ entries, each one takes $\Theta(n)$ time. Overall, it is $\Theta(n^4)$ time.

DP # 2

Same setup as DP # 1, but only $k = \{1, 2, 4, 8, ...(n-1)$'s next power of $2\}$

We choose the middle vertex ℓ, then join things together:

Recursive formula:

$$D[i, j, k] = \min_{\ell \in \{1 \ldots n\}} \{D[i, \ell, k/2] + D[\ell, j, k/2]\}$$

There are $\Theta(n^2 \log n)$ entries, each one takes $\Theta(n)$ time to compute. Overall, $\Theta(n^3 \log n)$ time.

Floyd-Warshall Algorithm

This is another DP variant.

Subproblems: $D[i, j, k]$ minimum weight over all paths i to j with only intermediate vertices in $\{1 \dots k\}$.

Answers: $D[i, j, n]$ for any vertices $\langle i, j \rangle$.

Base Case:

$$D[i, j, 0] = \begin{cases} w(i, j) & i \neq j \\ 0 & i = j \end{cases}$$

Recursive formula:

$$D[i, j, k] = \min\{D[i, j, k-1], D[i, k, k-1] + D[k, j, k-1]\}$$

i.e. the minimum of the path that doesn't use k and the path from i to k plus the minimal path k to j.

The iteration order will be weird (increasing k). I don't have it written down, but would need to think about it.

Every edge takes $O(1)$ time to compute, there are $O(n^3)$ edges. I suppose this takes $O(n^3)$ time overall.

Chapter 6

NP/P/etc

6.1 Theory of NP-Completeness

All I have written down is that proving problems is hard.

I'll describe NP-Completness later in the NP-Completness section.

6.2 Problem Specifications

6.2.1 Decision Problem

Decision Problems are problems that output "yes/no" answers.

6.2.2 Search Problems

Search Problems are problems that determine the existence of at least one solution to a given problem.

6.3 Definition of P

Class P is the class of "easy"/"tractable" problems.

P is the set of problems that are solveable in worst-case polynomial time i.e. $O(n^d)$ for some constant d. We require that n is a reasonable representation of the input size, measured in bits.

6.4 Definition of NP

Class NP is the class of "hard"/"intractable" problems.

NP is the set of problems that have solutions that are verifiable in $O(n^d)$ time for some constant d. We require that n is a reasonable representation of the input size, measured in bits.

All decision problems that can be expressed in the form:

- Intput $\langle x \rangle$
- Output "yes" iff there exists an object y such that property $R(x, y)$ holds.
  Where:
  1. y has polynomial size
  2. $R(x, y)$ can be verified in polynomial time.

  We call y the certificate, and evaluating R verification.
  NP $\leftrightarrow$ "Non-Deterministic Polynomial"

6.5 P vs NP vs EXPTIME

$$P \subseteq NP \subseteq EXPTIME$$

EXPTIME is the set of problems solveable in $O(2^{n^d})$ time.

Proving (or disproving) that $P = NP$ is a million dollar proof.

6.6 Polynomial-time reductions

To prove that problems are equivalently hard, we want to provide polynomial-time "reductions" from one type of problem to another.

We can say that L_1 polynomial-time reduces to L_2 if arbitrary instances of problem L_1 can be solved using polynomial time computational steps, plus polynomial calls to an oracle that solves L_2.

We say that $L_1 \leq_p L_2$ if there is a polynomial time function f so that L_1 on x is a "yes" iff output of L_2 on $f(x)$ is "yes".

6.7 NP-Completeness

NP-Complete problems are a set of NP problems that all polynomial-reduce to each other. The implications of proving that one of these problems is in P would mean that $P = NP$.

The requirement for NP-Completeness seems pretty trivial, which makes it easier for us to prove.

$$L \in NP$$
$$L_0 \leq_p L$$

If L_0 is known to be NP-Complete, then L is NP-Complete too.

6.7.1 Proving NP-Completeness

6.8 SAT (Cook-Levin theorem)

6.9 SAT to 3SAT

6.10 3SAT to independent set/vertex cover/clique

6.11 Vertex cover to Hamiltonian cycle/traveling salesman

6.12 Vertex cover to subset sum/knapsack

6.13 Beyond NP

6.13.1 Halting Problem

6.13.2 Turing's theorem