CS341: Algorithms **Spring 2020**

Lecture 3: May 19

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 3.1. **Big-O notation**: $f(n) \in O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

Definition 3.2. **Big-Omega notation**: $f(n) \in \Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$

Definition 3.3. **Theta notation**: $f(n) \in \Theta(g(n))$ if there exists constants $c_1 > 0, c_2 > 0$ and $n_0 > 0$ such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$

Definition 3.4. **Little o-notation**: $f(n) \in o(g(n))$ if for any constants $c > 0$, there exists a $n_0 > 0$ such that $|f(n)| < c|g(n)|$ for all $n \geq n_0$

Definition 3.5. **Little w-notation**: $f(n) \in \omega(g(n))$ if for any constants $c > 0$, there exists a $n_0 > 0$ such that $|f(n)| > c|g(n)|$ for all $n \geq n_0$

Theorem 3.6. $f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

Theorem 3.7. Let $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$, Then: $\begin{cases} f(n) \in o(g(n)) & \text{if } L = 0 \\ f(n) \in \Theta(g(n)) & \text{if } 0 < L < \infty \\ f(n) \in \omega(g(n)) & \text{if } L = \infty \end{cases}$

Theorem 3.8. Some Mathematical Results:

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{n} = \Theta(n^2)$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{n6} = \Theta(n^3)$
- $\sum_{i=1}^{n} i^d = \Theta(n^d) \quad \forall d \geq 1$
- $\sum_{i=1}^{n} c^i = c + c^2 + \cdots + c^n = \frac{c^{n+1}-1}{c-1} = \begin{cases} \Theta(n) & \text{for } c = 1 \\ \Theta(1) & \text{for } c < 1 \\ \Theta(c^n) & \text{for } c > 1 \end{cases}$
- $\sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \leq \int_1^n \frac{1}{x} dx = \ln x \in \Theta(\log n)$
- $\log(n!) = \sum_{i=1}^{n} \log(i) = n \log(n) - \Theta(n) = \Theta(n \log n)$

Example 3.9. Analyze the following loop:

```
for i = n down to 1
    j = i
    while (j ≤ n)
        j = 2j
```

If we take $\log(n!) = \Theta(n \log n)$, the above runtime would be $\Theta(n \log n)$. However, if we take $\log(n!)$ as $n \log n - \Theta(n)$, the above runtime would be $\Theta(n)$.

Theorem 3.10. Prove

$$a^{\log_b n} = n^{\log_b a}$$

Proof 3.11.

$$\begin{aligned} LHS &= a^{\log_b n} \\ &= a^{\frac{\log_a n}{\log_a b}} \\ &= (a^{\log_a n})^{\frac{1}{\log_a b}} \\ &= n^{\frac{1}{\log_a b}} \\ &= n^{\frac{1}{\frac{log_b b}{\log_b a}}} \\ &= n^{\frac{\log_b a}{log_b b}} \\ &= n^{\log_b a} \\ &= RHS \end{aligned}$$

□

CS341: Algorithms **Spring 2020**

Lecture 4: May 21

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 4.12. Divide and Conquer Algorithm: Algorithms that are MergeSort-like. It has the following template:

```
DC-Alg-Template:
    DC-Alg (P)
        base case...
        sol_1 = DC-Alg(subP_1)
        sol_2 = DC-Alg(subP_2)
        ...     sol_a = DC-Alg(subP_a)
        return combine (sol_1, sol_2, ..., sol_a)
```

Definition 4.13. **Recurrence**: an equation or an inequality, that describes a function $T(n)$(in our case, the runtime of DC-Alg), in terms of $T's$ values on smaller inputs and a base case.

Example 4.14. MergeSort: $T(n) \leq 2T(\frac{n}{2}) + 7n$; $T(2) \leq 2$

There are three methods to solve an recurrence:

1. Proof by Induction: substitution method or guess & check method
2. Recursion Tree: already saw. Write down the algorithm and analyze each pieces.
3. Master Method: easiest to use if $T(n)$ looks like $T(n) \leq aT(\frac{n}{b}) + O(n^d)$

Example 4.15. Median-of-Medians: $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + n$; $T(1) = 1$. We cannot use Master Method. We will prove by induction.

Guess: $T(n) \leq 10n \quad \forall n \geq 1$

Proof by Induction:

Base case: $n = 1$, $T(1) = 1 \leq 10 \times 1$. Correct.

Inductive Hypothesis: Assume $T(k) \leq 10k \quad \forall k \leq n-1$, we need to prove that $T(n) \leq 10n$

$$\begin{aligned} T(n) &\leq \frac{10n}{5} + \frac{10 \times 7n}{10} + n \\ &\leq 2n + 7n + n \\ &= 10n \end{aligned}$$

□

Recursion Tree Method: draw out all levels, and sum everything together. Probably will use the result from previous lecture. There would be three cases and Master Theorem captures these three possibilities:

1. work at each level stay same.
2. work at root dominates.
3. work at leaves dominate.

Theorem 4.16. **Master Theorem**: A "blackbox" for solving recurrence of a specific type: if all subproblems are of equal size

Or more formally: **If**

$$\begin{aligned} &T(n) \leq aT(\frac{n}{b}) + O(n^d) \\ &T(O(1)) = \Theta(1) \\ &a = \text{ number of recursive calls} \\ &b = \text{ input size shrinkage factor} \\ &d = \text{ exponent in the runtime of combine step.} \end{aligned}$$

Then $$T(n) = \begin{cases} \Theta(n^d \log(n)) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(a^{\log_b(n)}) = \Theta(n^{log_b(a)}) & \text{if } a > b^d \end{cases}$$

Proof 4.17. Assume, for simplicity, n is a power of b.

Drawing the trees we can see that the number of leaves is $a^{\log_b(n)}$.

Pick an arbitrary level j, the number of sumproblems is a^j. The size of input to the subproblem would be $\frac{n}{b^j}$. The work done by each subproblem is $(\frac{n}{b^j})^d = \frac{n^d}{(b^d)^j}$.

The total work at level j is at most $a^j \cdot \frac{n^d}{(b^d)^j} = n^d \cdot (\frac{a}{b^d})^j$.

The total work of the algorithm is

$$\sum_{j=0}^{log_b n} n^d \cdot (\frac{a}{b^d})^j = n^d \sum_{j=0}^{log_b n} (\frac{a}{b^d})^j$$

Applying the result from last lecture completes the proof (summing a geometric series, depending on the size of $\frac{a}{b^d}$). □

Example 4.18. For MergeSort: $T(n) = 2T(\frac{n}{2}) + 7n$. Hence $a = 2, b = 2, d = 1$. So by Master Theorem, the runtime is in $\Theta(n \log n)$.

CS341: Algorithms **Spring 2020**

Lecture 5: May 26

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 5.19. Problem from Computational Geometry: **2D Maxima**:
Input: a set P of n 2D points.
Output: All maximal points. Maximal point is defined as follows: p is maximal if no point q dominates p, i.e. $\nexists\ p.x < q.x \ \&\ p.y < q.y$

We have a Naive Algorithm, running in $\Theta(n^2)$:

```
Naive Algo:
    for each p ∈ P:
        for each q ∈ P s.t. p ≠ q:
            check if q dominates p
        if p is not dominated
            output p
```

We can have a better solution by the principle of Divide and Conquer: Suppose we divide the points into L and R according to point with median x value.

Observation 1 : maximal point in R is globally maximal.

Observation 2 : maximal point in L is maximal if and only if $p.y > p*.y$ where $p*$ is the highest y-valued point in R.

```
DC Algo:
    sort P by x-axis
    DC-Maxima(P)
DC-Maxima(P) (P is sorted by x axis)
    if P.size() = 1 return P
    ML = DC-Maxima(P[1, ..., n/2])
    MR = DC-Maxima(P[n/2 + 1, ..., n])
    let p* = highest y-valued point in P[n/2 + 1, ..., n]   ⇒ O(n)
    out = MR
    for p ∈ ML   ⇒ O(n)
        if each (p.y ≥ p*.y) then
            out.add(P)
    return out
```

Overall, the runtime would be $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

Exercise: after the initial sort on x value, find a non-recursive algorithm that takes $O(n)$ time.

Definition 5.20. Problem from Computational Geometry: **Closest 2D Pair**.
Input: a set P of n 2D points.
Output: Find a pair (p, q) with the closest Euclidean distance.

$$dist(p, q) = \sqrt{(p.x - q.x)^2 + (p.y + q.y)^2}$$

What if it is a 1D problem? Sort it and traverse once. Solved within $\Theta(n \log n)$ time. We have a Naive Algorithm, running in $\Theta(n^2)$:

Naive Algo:
for each pair (p, q), compute $dist(p, q)$ and keep a running minimum.

We can have a better solution by the principle of Divide and Conquer: First, we claim that the closest pair $(p*, q*)$ must satisfy one of the following:

1. Both in L
2. Both in R
3. $p* \in L$ and $q* \in R$

We should also see that

Observation 1 : Let $\delta = \min\{dist(pairL), dist(pairR)\}$
If $dist(pairL) < \delta$, then $pairS$ lies within 2δ-strip around the median x value becuase any point outside this region is already at a distance $> \delta$ to the point in the L and R.

Observation 2 : Consider the point p' with the lowest $y - value$ in the 2δ-strip. If $S = (p', q)$ is the winner, i.e. $dist(pairS) < \delta$, then $q.y \leq p.y + \delta$. Hence, we will start with the lowest p' point, and looking for the points within a $2\delta \times \delta$ box.

```
DC Algo:
        sort P by x-axis
        DC-CP(P)
DC-CP(P) (P is sorted by x axis)
        if P.size() = 1 return P
        PairL = DC-Maxima(P[1, ..., n/2])
        PairR = DC-Maxima(P[n/2 + 1, ..., n])
        δ = min{dist(PairL), dist(PairR)}
        PairS = findSpanningPair(P)
        return min{pairL, pairR, pairS} by their distance
findSpanningPair(P) (P is sorted by x axis)
        S = select p ∈ P such that |P[n/2] − p.x| ≤ δ   ⇒ O(n)
        sort S by y values.    ⇒ O(n log n)
        minDist = +∞
        minPair = NULL
        for i = 1 ... |S|.length    ⇒ O(n)
            j = i+1
            while (S[j].y ≤ S[i].y + δ)
                if (dist(S[i], S[j]) < minDist)
                    minDist =dist(S[i], S[j])
                    minPair =(S[i], S[j])
                j++
        return minPair
```

Overall, the runtime would be $T(n) = 2T(\frac{n}{2}) + O(n \log n) = O(n \log^2 n)$. Can we improve this to be $O(n \log n)$? Yes. At the very beginning, sort P by y as well. When calling `findSpanningPair`, we pass an extra P_y, so that we can select points from P_y directly to construct S. This avoids sorting on $y - value$ each time, and hence will reduce the runtime to be $O(n \log n)$.

CS341: Algorithms **Spring 2020**

Lecture 6: May 28

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 6.21. Problem of **Integer Multiplication**:
Input: 2 n-digit integers X and Y.
Output: $Z = XY$. We will work with base 10, but the same algorithm and analysis for base 2 (or any other base).

We have an Naive Algorithm, that we normally learned from primary school.

> Primary School Alg:
> Multiply each digit in Y by X, shift one digit each time, and suming all of them up.

The number of operations, if we see multiply/add 1 digit number is 1 operation, shift a digit is 1 operation (treat this as multiply by 10), will be: $\Theta(n^2)$. There are $1+2+\cdots+(n-1)$ shifts which is in $\Theta(n^2)$ already.

We might do something better, using the principle of Divide and Conquer. For example, maybe we could have a DC algorithm that divides X, Y into 2 ints of $\frac{n}{2}$-digits.

- $X = a10^{\frac{n}{2}} + b$
- $Y = c10^{\frac{n}{2}} + d$
- If $X = 2345, Y = 6789$, then $a = 23, b = 45, c = 67, d = 89$.
- Then

$$\begin{aligned} XY &= (a10^{\frac{n}{2}} + b)(c10^{\frac{n}{2}} + d) \\ &= ac10^n + ad10^{\frac{n}{2}} + bc10^{\frac{n}{2}} + bd \\ &= ac10^n + (ad + bc)10^{\frac{n}{2}} + bd \end{aligned}$$

DC-Mult1(X, Y n digit)
  if $|X|, |Y| = 1$ **return** XY
  define a, b, c, d as before
  $v_{ac} =$ `DC-Mult1`(a, c)
  $v_{ad} =$ `DC-Mult1`(a, d)
  $v_{bc} =$ `DC-Mult1`(b, c)
  $v_{bd} =$ `DC-Mult1`(b, d)
  combine:
    $tmp1 = v_{ac} \cdot 10^n \quad \Rightarrow O(n)$
    $tmp2 = v_{ad} + v_{bc} \Rightarrow O(n)$
    $tmp3 = tmp2 \cdot 10^{\frac{n}{2}} \quad \Rightarrow O(n)$
  return $tmp1 + tmp3 + v_{bd} \quad \Rightarrow O(n)$

The runtime of the above algorithm is $T(n) = 4T(\frac{n}{2}) + \Theta(n)$, $a = 4, b = 2, d = 1$, by Master Theorem, $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$.

We can do something better. In the above procedure, we don't actually need ad and bc. All we need is their sum. Can we reduce to make only 3 recursive calls? Yes. We will compute

(1) $a \times c$

(2) $b \times d$

(3) $(a + b) \times (c + d) = ac + ad + bc + bd$

And then we compute (3) - (2) - (1) will give us what we want $(ad + bc)$

DC-Mult2(X, Y n digit) (aka **Karatsuba-Ofman Algorithm**)
  if $|X|, |Y| = 1$ **return** XY
  define a, b, c, d as before
  $v_{ac} =$ `DC-Mult2`(a, c)
  $v_{bd} =$ `DC-Mult2`(b, d)
  $tmp =$ `DC-Mult2`$(a + b, c + d) \quad \Rightarrow O(n)$
  return $v_{ac}10^n + (tmp - v_{ac} - v_{bd})10^{\frac{n}{2}} + v_{bd} \quad \Rightarrow O(n)$

The runtime of the above algorithm will be $T(n) = 3T(\frac{n}{2}) + \Theta(n)$, $a = 3, b = 2, d = 1$, by Master Theorem, $T(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1.59})$. □

Fact: we can use Karatsuba-Ofman Algorithm to divide X and Y into k ints of size $\frac{n}{k}$, and as a result, we will get the runtime reduce to $O(n^{\log_k(2k-1)}) \Rightarrow O(n^{1+\epsilon})$ for any $\epsilon > 0$. The best/fastest algorithm for this problem is done in $O(n \log n \times \log(\log n))$.

Definition 6.22. Problem of **Matrix Multiplication**:
Input: 2 $n \times n$ matrices A and B.
Output: $C = A \times B$, as an $n \times n$ matrix. The multiplication is defined by $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$.

$$\begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$

If we stick on the operation in the definition to compute C, we will have an algorithm of $O(n^3)$ since for each c_{ij} we need n multiplications and n additions, and hence we will compute each c_{ij} in $\Theta(n)$.

```
Standard Alg (A, B)
    for i = 1...n
        for j = 1...n
            C[i][j] = 0
            for k = 1...n
                C[i][j] = A[i][k] + B[k][j]
    return C
```

We can design a Divide and Conquer algorithm. We can divide A, B into four $\frac{n}{2} \times \frac{n}{2}$ matrices. A fact from linear algebra is that:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

such that each block behaves as an atomic element, i.e.

$$C_{ij} = (A_{i1}B_{ij} + A_{i2}B_{2j})$$

in which each operation is matrix multiplication/addition.

Also recall that multrix addition is defined as follows

$$\begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$

in which

$$c_{ij} = a_{ij} + b_{ij} \quad \Rightarrow \Theta(1)$$

We will rely on this fact to design a new algorithm

```
DC-MM1 (A, B n × n matrices)
    C11 = DC-MM1(A11, B11) + DC-MM1(A12, B21)   ⇒ n^2/4 for additions
    C12 = DC-MM1(A11, B12) + DC-MM1(A12, B22)   ⇒ n^2/4 for additions
    C21 = DC-MM1(A21, B11) + DC-MM1(A22, B21)   ⇒ n^2/4 for additions
    C22 = DC-MM1(A21, B22) + DC-MM1(A22, B22)   ⇒ n^2/4 for additions
    return [C11 C12; C21 C22]                   ⇒ Θ(n^2)
```

The runtime of the above algorithm is $T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$, $a = 8, b = 2, d = 2$, by Master Theorem, $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$.

Similar to the problem with integer multiplication, when compute $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$, we don't care about the individual elements. All we need is the sum. The fact is that we don't have to make 8 recursive calls. We will see some magic.

DC-MM2 (A, B $n \times n$ matrices) (aka **Strassen's Algorithm**)

$Z_1 =$ `DC-MM2`$(A_{11}, B_{12} - B_{22})$
$Z_2 =$ `DC-MM2`$(A_{11} + A_{12}, B_{22})$
$Z_3 =$ `DC-MM2`$(A_{21} + A_{22}, B_{11})$
$Z_4 =$ `DC-MM2`$(A_{22}, B_{21} + B_{11})$
$Z_5 =$ `DC-MM2`$(A_{11} + A_{22}, B_{11} + B_{22})$
$Z_6 =$ `DC-MM2`$(A_{12} - A_{22}, B_{21} + B_{22})$
$Z_7 =$ `DC-MM2`$(A_{11} - A_{21}, B_{11} + B_{12})$
combine: $\Rightarrow \Theta(n^2)$
$C_{11} = Z_5 + Z_4 - Z_2 + Z_6$
$C_{12} = Z_1 + Z_2$
$C_{21} = Z_3 + Z_4$
$C_{22} = Z_5 + Z_1 - Z_3 - Z_7$
return $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ $\Rightarrow \Theta(n^2)$

To somehow prove this actually work, let's take a look at C_{12}, which follows exactly the definition.

$$\begin{aligned} C_{12} &= Z_1 + Z_2 \\ &= A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{22} + A_{12}B_{22} \\ &= A_{11}B_{12} + A_{12}B_{22} \end{aligned}$$

The runtime of the above algorithm is $T(n) = 7T(\frac{n}{2}) + O(n^2)$, $a = 7, b = 2, d = 2$, by Master Theorem, $T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$.

The history of Matrix Multiplication:

- until 1969, the best we have is $\Theta(n^3)$
- in 1969, we get Strassen's Algorithm which runs in $\Theta(n^{2.807})$
- in 1978, we get to $\Theta(n^{2.796})$
- in 1979, we get to $\Theta(n^{2.780})$
- in 1981, we get to $\Theta(n^{2.522})$
- in 1982, we get to $\Theta(n^{2.517})$
- in 1983, we get to $\Theta(n^{2.496})$
- in 1986, we get to $\Theta(n^{2.379})$
- in 1989, we get to $\Theta(n^{2.376})$. This is the best we get right now.
- After that, we use a better/more accurate runtime evaluation. The algorithm did not change at all, but the runtime is better.
- in 2010, we get to $\Theta(n^{2.374})$
- in 2011, we get to $\Theta(n^{2.37286642})$
- in 2019, we get to $\Theta(n^{2.3728639})$

CS341: Algorithms **Spring 2020**

Lecture 7: June 2

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 7.23. Greedy Algorithm: Algorithms that iteratively make "short-sighted", locally optimum looking decisions with the hope that they output an optimal solution.

Example 7.24. When doing coin changing, we are greedily looking for notes/coins with larger values.

Greedy Algorithms	Divide and Conquer Algorithms
easy to design	difficult to design
easy to analyze	difficult to analyze
difficult to prove correctness	easy to prove correctness

We have two main proof techniques:

1. Greedy-stays-ahead-arguments: argue that the greedy solution S_g is better than an arbitrary solution s at each iteration.
2. Exchange Argument: argue that an arbitrary solution s can be transformed, step by step and without getting worse, into the greedy solution S_g.

Definition 7.25. Problem of **Activity Selection**
Input: 1 resource and n requests, where each request i has a start time $s(i)$ and a finishing time $f(i)$
Output: the maximum number of non-overlapping requests.

There could be several "natural" algorithms (in which the fourth one is correct):

1. Greedy Alg 1: looking for the earliest start time.
2. Greedy Alg 2: looking for the shortest request.
3. Greedy Alg 3: looking for minimum overlapping requesr first.
4. Greedy Alg 4: looking for the earliest finishing time.

```
Greedy Alg 4(R):
    while there are still some requests in R
        Pick the earliest finishing request, add it to the result
        Remove all the overlapping requests in R.
```

The intuition behind this algorithm is that, if we don't pick the one with the earliest finishing time, we lose some time for no reason. Given that we can have one more request, why would we waste some time to take the one that will be finished later? The proof of correctness is as below:

Proof 7.26. Let $r_{g_1}, ..., r_{g_k}$ be the k requests that the above algorithm picks.

Key Claim: the earliest time that any schedule can fulfill i requests is $f(r_{g_i})$, i.e. no requests can be finished before $f(r_{g_i})$.

By induction on i, we see that:

Base Case: $i = 1$, holds by construction, $f(r_{g_i})$ is the earliest finishing time of all requests.

Inductive Hypothesis: Suppose the claim holds for all $i = t$.

Inductive Conclusion: Try to prove for the case of $i = t + 1$.

Let $r_{h_1}, ..., r_{h_t}, r_{h_{t+1}}$ be the first $t + 1$ requests executed by an arbitrary solution s.

Suppose for contradiction that $f(r_{h_{t+1}}) < f(r_{g_{t+1}})$. But this is impossible because our algorithm would have picked $f(r_{h_{t+1}})$ over $f(r_{h_{g+1}})$. Or more formally, we have $f(r_{g_t}) \leq f(r_{h_t}) \leq s(r_{h_{t+1}}) \leq f(r_{h_{t+1}}) < f(r_{gt+1})$, where the first part comes from Inductive Hypothesis, the second part comes from the fact that start times are earlier than finishing times for any requests, and the last part comes from the assumption of the contradiction. This is indeed a contradiction because we are picking a later time $r_{h_{t+1}}$ is not overlapping with r_{g_t} or (anything our algorithm picked earlier). So the algorithm will pick $f(r_{h_{t+1}})$ over $f(r_{h_{g+1}})$ by the assumption finishes later than $r_{h_{t+1}}$. □

Corollary 7.27. Greedy Alg 4(R) is optimal.

Proof 7.28. Similar to the above proof.

Let $S_g = r_{g_1}, ..., r_{g_k}$.

Suppose for contradiction that there is another solution s with $k+1$ requests, in which $s = r_{h_1}, ..., r_{h_k}, r_{h_{k+1}}$. But this is a contradiction because we know by the claim above that $f(r_{g_k}) \leq f(r_{h_k})$, so r_{h+1} does not overlap with $r_{g_1}, ..., r_{g_k}$, and our algorithm would have picked it. □

The full algorithm is:

```
Greedy EFT(R):
R: a set of n requests
        Sort R by finishing times of requests    ⇒ Θ(n log n)
        S_g = {R[0]}
        latestFT = R[0].ft
        for r = 1, ..., n    ⇒ O(n)
            if (R[i].start > latestFT)
                S_g.add(R[i])
                latestFT = R[i].ft
```

The runtime is $\Theta(n \log n)$

CS341: Algorithms **Spring 2020**

Lecture 8: June 3

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 8.29. Problem of **Job Scheduling1**
Input: a set of n jobs $J_1, ..., J_n$, each with length l_i for $1 \leq i \leq n$
Output: a schedule of jobs on a processor such that $\sum_{j=1}^{n} C_j$ is minimized, where C_j is the completion time of job J_j

Intuition: jobs with shorter completion time have less impact on the completion times of future jobs.

```
Greedy-Shortest-Job-First(J)
J: the set of n jobs
        return (Sort J)    ⇒ Θ(n log n)
```

Proof 8.30. Proof of optimality of the above algorithm:

For a schedule S, let $S[i]$ be the i-th jov executed.

Let S_g be the greedy schedule.

Let $subcost(S, j) = \sum_{i=1}^{j} C_{S[i]}$. So by definition, $cost(S) = subcost(S, n)$.

Claim: Take an arbitrary solution S, $subcost(S_g, j) \leq subcost(S, j)$ for all $j = 1, ..., n$.

Proof: By induction on j.

Base Case: $j = 1$. It is true for the base case since $subcost(S_g, 1) = length(S_g[1]) \leq subcost(S, 1)$ because $length(S_g[1])$ is globally the shortest length among all n jobs.

Inductive Hypothesis: Suppose for $j = k$, the claim holds, i.e.$subcost(S_g, k) \leq subcost(S, k)$

Inductive Conclusion: We need to prove for $j = k + 1$, i.e. $subcost(S_g, k + 1) \leq subcost(S, k + 1)$.

Note that $subcost(S_g, k + 1) = subcost(S_g, k) + \sum_{i=1}^{k+1} length(S_g[i]) \leq subcost(S, k) + \sum_{i=1}^{k+1} length(S[i])$. This is true because $subcost(S_g, k) \leq subcost(S, k)$ by inductive hypothesis and $\sum_{i=1}^{k+1} length(S_g[i]) \leq \sum_{i=1}^{k+1} length(S[i])$ by the algorithm as in S_g they are sorted by length, and we are summing up the $k + 1$ shortests jobs.

Hence, $subcost(S_g, k + 1) \leq subcost(S, k + 1)$, so $cost(S_g) \leq cost(S)$, and hence S_g is optimal. □

Definition 8.31. Problem of **Job Scheduling2**
Input: a set of n jobs $J_1, ..., J_n$, each with length l_i and weight w_i for $1 \leq i \leq n$
Output: a schedule of jobs on a processor such that $\sum_{j=1}^{n} w_j C_j$ is minimized, where C_j is the completion time of job J_j, and w_j is the weight of job J_j.

If the weights are the same, then this problem can be simplified to **Job Scheduling1**. If the lengths are the same, similarly, this problem can be simplified to **Job Scheduling1**, ordering by decreasing weights.

The general strategy is to combine l and w into a single score f that captures both intuitions and sort by increasing f. So combined scoring function $f(l_i, w_i)$ should satisfy:

1. If weights are the same, shorter length jobs get lower scores
2. If lengths are the same, higher weight jobs get lower scores.

Possible combined scoring function could be $f_1(l_i, w_i) = l_i - w_i$, $f_2(l_i, w_i) = \frac{l_i}{w_i}$

```
Greedy-Combined-Score-Low-First(J)
J: the set of n jobs
        return (Sort J by the score l_i/w_i)    ⇒ Θ(n log n)
```

Theorem 8.32. Greedy Sort by $\frac{l_i}{w_i}$ is optimal.

Proof 8.33. Proof with Exchange Argument: Argue that an arbitrary schedule S can be transformed into S_g step by step without getting worse.

Notational change: Let's rename jobs so that $S_g = J_1 J_2 \ ... \ J_n$, i.e. J_1 has the lowest $\frac{l}{w}$ score, and J_2 has the second lowest $\frac{l}{w}$ score, and so on.

Let S be any arbitrary schedule such that $S \neq S_g$. Then it must be the case that in S, there is a job J_k right after a job J_i such that $i > k$.

Let's exchange J_i and J_k. The cost of S will change. Recall that the cost of a schedule is $cost(S) = \sum_{j=1}^{n} w_j C_j$. Note that the contributions of prefix and suffix jobs do not change.

$\Delta cost(J_k) = -w_k l_i$, and $\Delta cost(J_i) = w_i l_k$. Overall, the change in the cost of S is $-w_k l_i + w_i l_k \leq 0$ since $k < i$, and hence $\frac{l_k}{w_k} \leq \frac{l_i}{w_i} \quad \Rightarrow \quad w_k l_i \geq w_i l_k$. Hence the $cost(S') \leq cost(S)$. So the exchange can only improve the cost of S.

Therefore, if we keep finding such "out of order consecutive pairs" and performed exchanges, at some point we have to stop (i.e. there can no longer be out of order consecutive pairs). There exists $\binom{n}{2}$ possible pairs we can swap, and once a pair J_i, J_k gets swapped, they can never get swapped again. At some point, the swaps must stop.

We will eventually arrive at S_g. Hence $cost(S_g) \leq cost(S)$. S_g is the optimal schedule. □

CS341: Algorithms **Spring 2020**

Lecture 9: June 9

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 9.34. Problem of **Stable Matching**
Input: n interns and n companies, and each intern/company has a preference list over companies/interns
Output: a matching M between interns and companies that is stable. By stable, we mean that there is no pairs (c, i) and (c', i'). such that for c: $i' > i$ and for i': $c > c'$. That is, c and i' prefer each other to i and c', respectively. Also, the matching has to be "perfect": every intern/company is matched.

Gale and Shapley's Greedy Algorithm(n companies, n interns, and their rankings)
Initially all companies and interns are unmatched
while (there exists unmatched company who hasn't proposed to all interns)
c = pick one such company arbitrarily
i = highest ranked intern to whom c has not yet proposed
c proposes to i:
if (i is free) **then**
(c, i) are "tentatively matched"
else
c' = the company i is currently matced to
if(i prefer c over c') **then**
(c, i) are matched and c' is unmatched
return all matched pairs

1. Does this algorithm terminate? Yes, at each iteration, we wish to pick an unmatched company who hasn't yet proposed to every n intern. We can pick the same company at most n times, and there are n companies. Hence, there are at most n^2 interations.
2. Does it always return a matching?
   - For a matched intern i, it will never unmatched from now on
   - So yes, it will be matched at some point when both a company and a intern is unmatched.
3. Is that matching stable?
   - $i's$ sequence of companies get better and better over time
   - A matched company can get unmatched later on. And the sequence of interns c get worse and worse over time.
   - So yes. The company will propose to an ideal candidate, then regardless the candidate accepted or rejected the offer, this candidate will not get matched with a worse company.
4. Is there always a stable matching? Yes, because our algorithm returns one.

5. Does it always return the same matching? Yes.

   We define that: A valid intern i for a company c is an intern such that there exists a stable matching SM_j in which c is matched to i. An optimal intern i for a company c is $c's$ most preferred valid intern.

   We claim that each company gets their optimal intern, and each intern get matched with their worst (pessimal) company, i.e. this algorithm is extremely favoring the company.

   Suppose during the execution of the algorithm, a company c is rejected by $optimal(c)$. Consider the first time this happens and without loss of generality it happens to company 1 when proposing to intern D, i.e. suppose optimal (1) is D, but D rejected 1 (say for 3).

   By definition of "optimal intern", there exists a stable matching $S*$ with $(1, D)$ is matched. Then D prefers 3 over 1. Also, 3 prefers D over C since 1 was the first company c rejected by $optimal(c)$. And hence this cannot be a stable matching, and we reach a contradiction.

CS341: Algorithms **Spring 2020**

Lecture 10: June 11

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 10.35. Problem of **Linear Independent Set**:
Input: undirected line graph $G = (V, E)$, which is a chain, and weights on vertices
Output: The max weight independent set in G, i.e. a set of mutually non-adjacent vertices with the maximum total weight.

Naive Algorithm:
Search all 2^n possible subsets: check with one has the highest weight and is independent. $\Rightarrow \Theta(2^n)$

Greedy Algorithm:
Pick next independent vertex greedily. But this algorithm is incorrect.

Dynamic Programming Approach: Let $S*$ be the optimal solution. Reason about what $S*$ looks like in terms of optimal solutions to sub-problem will be like follows:

Suppose we have a vertex set V of size n, in which $V = \{v_1, ..., v_n\}$, with weights $w_1, ..., w_n$ respectively. Consider v_n, we can make a simple claim that: there are only two possibilities, either in S^* or not in S^*.

Denote $G_n = G$, in which $S^*_n = S^*$ is the optimal solution. Similarly $G_{n-1} = G \setminus \{v_n\}$, in which S^*_{n-1} is the optimal solution. And we have until $G_1 = \{v_1\}$, in which $S^*_1 = \{v_1\}$ is the optimal solution.

If $v_n \notin S^*$, then $S^*_n = S^*_{n-1}$ because that means for all $v \in S^*_n$, $v \in \{v_1, ..., v_{n-1}\}$, and hence S^* is a solution to G_{n-1}, and it must be the max-weight solution to G_{n-1}, otherwise S^*_{n-1} is not the optimal solution for G_{n-1}.

If $v_n \in S^*_n$, then $S^*_n = S^*_{n-2} \cup \{v_n\}$, $S^*_n \setminus \{v_n\} = S'^*_n$, is an optimal solution to G_{n-2}, otherwise weight of $weight(S^*_{n-2}) + w_n > weight(S'^*_n) + w_n = weight(S^*_n)$, contradicting the fact that S^*_n is the max weight independent set in G_n.

Hence, if $v_n \notin S^*_n$, then $S^*_n = S^*_{n-1}$; if $v_n \in S^*_n$, then $S^*_n = S^*_{n-2} \cup \{v_n\}$. Here is an recursive algorithm based on above thoughts:

```
LIS-Rec(G = (V, E) and W):
    if V.size() = 1, then
        return V
    S_1 = LIS-Rec(G_{n-2}) ∪ {v_n}
    S_2 = LIS-Rec(G_{n-1})
    return max{S_1, S_2}
```

The runtime of the above algorithm is $T(n) = T(n-1) + T(n-2) + O(1) = \Theta(2^n)$. This is slow because we repeatedly calculated a lot of results, pretty much like the naive way to calculate Fibonacci Sequence.

The first type of fix is to cache the results, or also called memoization. This improves the runtime to be $\Theta(n)$.

```
LIS-DP(G = (V, E) and W):
A: a solution array of size n + 1.
A[i] = max weight independent set to G_i
        A[0] = 0
        A[1] = w_1
        for i = 2 to n
            A[i] = max{A[i − 2] + w_i, A[i − 1]}
        return A[n]
```

The runtime would simply be $\Theta(n)$. Arguing for correctness, we just need to use the arguments in our thought process. Or more rigorously, prove it by induction.

Space requirement would be $\Theta(n)$, we can improve this by storing only recent 3 results. The cases/results before $i-2$ will not be reused again in the future. We just need two temp variables.

How to we reconstruct the actual independent set? The first option is, for each $A[i]$, we store the independent set for G_i. The space requirement would be $\Omega(n^2)$.

The second option is to backtrack. Given the result array A, and the weight array W, we can infer that if v_n is in S_n^* or not. We campare $A[n]$ with $A[n-1]$. If $v_n \in S_n^*$, then $A[n] = A[n-1]$. If not, $A[n] = A[n-2]+w_n$. If both are equal, then it implies that there are multiple solutions.

```
LIS-Reconstruct Algorithm (G = (V, E), weights, A)
A is the solution array
        let S* = ∅
        i = n
        while i > 0
            if (A[i − 2] + w_i ≥ A[i − 1]) then
                S* = S* ∪ {v_i}
                i = i − 2
            else
                i = i − 1
        return S*
```

The runtime would be $\Theta(n)$ time.

The recipe of a Dynamic Algorithm:

1. Identify a small number of subproblems. In the linear independent set problem, we have n subproblems.
2. Represent the solution to subproblems in a 1 or multidimensional array.
3. Quickly solve larger subproblems using solutions to smaller subproblems, establishing a recurrence relationship.
4. Solve all subproblems and quickly compute the final solution to the original problem.

CS341: Algorithms **Spring 2020**

Lecture 11: June 16

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 11.36. Problem of **Weighted Activity Selection**
Input: 1 resource (lecture room) and n requests (e.g. events) where as r_i has a start time $s(i)$ finish time $f(i)$ and value v_i.
Output: accept a set of non-overlapping requests with max value.
Mathematically, select a set S of requests such that for all (i, j), we have either $f(i) \leq s(j)$ or $f(j) \leq s(i)$, and we have $\sum_{i \in S} v_i$ is maximized overall possible S.

Assume, without loss of generality, r_i are ordered by non-decreasing finishing time, i.e. we have $f(r_1) \leq f(r_2) \leq \ldots \leq f(r_n)$.

Let $P(k)$ be the subproblem containing $r_1, \ldots, r_k$. Considider r_n, and call optimal solution S^*. There are two possibilities, either $r_n \in S^*$ or $r_n \notin S^*$.

If $r_n \notin S^*$, then S^* is optimal for $P(n-1)$ since S^* is a valid solution for $P(n-1)$. If $r_n \in S^*$, then $S^* \setminus \{r_n\}$ is optimal for $P(i^*)$, in which i^* is the largest index of a request r_{i^*} that doesn't intersect with r_n.

For each request r_j, we define $z(j) = i$ to be the largest index of a request r_i that doesn't intersect with r_j. And $z(1) = 0$.

```
Weighted-Activity-Selection(R)
    R: n requests, each with a start time s(i) finish time f(i) and value v_i
    compute Z for each 1 ≤ i ≤ n
    let A be an array of size n
    A[0] = 0, A[1] = {v_1}
    for i = 2 to n
        A[i] = max{A[i − 1], A[z(i)] ∪ {v_i}}
    return A[n]
```

This algorithm runs in $O(n)$ time, if R is sorted already. Computing Z could be done in linear time as well.

Definition 11.37. Problem of **Sequence Alignment**
Input: 2 DNA Stands X, Y (i.e. two strings containing A, T, G, C), and penalties $\delta > 0$ for gaps $(-)$, and $\alpha(i, j)$ for aligning nucleotides i and j. Assume $\alpha(i, i) = 0$, $X = x_1 x_2 ... x_m$, $Y = y_1 y_2 ... y_n$
Output: the minimum penalty alignment of X and Y.
An alignment: make X and Y the same length by (possibly) adding gaps to X and Y, Then put them on top of each other.

Note that there is a maximum number of gaps to be added. After that number, adding more gaps would be of no good.

Let's say $X^m = x_1...x_m$, $X^{m-1} = x_1...x_{m-1}$, ..., $X^1 = x_1$, and similarly for Y we have $Y^n = y_1...y_m$, $Y^{n-1} = y_1...y_{n-1}$, ..., $Y^1 = y_1$ Suppose we have an optimal solution $A^* = (X^*, Y^*)$, what could be matched in the last position? There are three cases:

1. (x_m, y_n). In this case, $IA = (X^* - x_m, Y^* - y_n)$ is an alignment for X^{m-1} and Y^{n-1}, and IA must be optimal, otherwise we reach a contradiction. Proof by contradiction: suppose IA is not optimal. Let $penalty(IA) = p$, then there exists another alignment IA' for X^{m-1} and Y^{n-1} such that $penalty(IA') < p$. Then consider alignment $A' = IA' \cup (x_m, y_n)$. A' is a alignment for $X^m = X$ and $Y^n = Y$ in which $penalty(A') = penalty(IA') + \alpha(x_m, y_n) < penalty(A^*) = penalty(IA) + \alpha(x_m, y_n)$. Contradicting the fact that A^* is optimal.

2. $(x_m, -)$ In this case, $IA = (X^* - x_m, Y^*)$ is an alignment for X^{m-1} and Y^n. This is optimal as well.

3. $(-, y_n)$ In this case, $IA = (X^*, Y^* - y_n)$ is an alignment for X^m and Y^{n-1}. This is optimal as well.

Hence

$$A^* = optimal(X^m, Y^n) = \min\{optimal(X^{m-1}, Y^{n-1}) + \alpha(x_m, y_n), optimal(X^{m-1}, Y^n) + \delta, optimal(X^m, Y^{n-1}) + \delta\}$$

```
Sequence-Alignment (X, Y, δ, α(i, j))
X, Y (i.e. two strings containing A, T, G, C)
δ: penalty for gaps
α(i, j): penalty for mismatches
        A be the 2-D solution array of size m × n.
        (A[i][j] is the penalty is the optimal alignment of X^i and Y^j)
        A[0][0] = 0
        for i = 1 to m
            A[i][0] = iδ
        for j = 1 to n
            A[0][j] = jδ
        for i = 1 to m
            for j = 1 to n
                A[i][j] = min{A[i − 1][j − 1] + α(x_i, y_j), A[i − 1][j] + δ, A[i][j − 1] + δ}
        return A[m][n]
```

The runtime of this algorithm would be $\Theta(nm)$.

Given the final penalty and the solution array A, we will do backtracking as follows:

```
Sequence-Alignment-Backtracking(X, Y, δ, α(i, j), A)
X, Y (i.e. two strings containing A, T, G, C)
δ: penalty for gaps
α(i, j): penalty for mismatches
A: the 2-D solution array of size m × n.
        let i = m, j = n
        let B be an empty linked list
        while (i > 0 and j > 0)
            if (A[i][j] was filled by case 1) then
                B.append((x_i, y_j))
                i − −, j − −
            else if (A[i][j] was filled by case 2) then
                B.append((x_i, −))
                i − −
            else if (A[i][j] was filled by case 3) then
                B.append((−, y_j))
                j − −
        return B
```

CS341: Algorithms **Spring 2020**

Lecture 12: June 18

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 12.38. Problem of **Matrix Multiplication Order**
Input: dimensions of n rectangular matrices, where column dimension of A_i = row dimension of A_{i+1}
Output: cost of min-cost parenthesization of multiplying the given n matrices. Assume given A, B of dimensions $p \times q$ and $q \times r$, the cost of computing AB is $p \times q \times r$. (we are deciding which matrices to multiply first)

Let's introduce some notations. Let $A_{i...j}$ be the result matrix of multiplying $A_iA_{i+1}...A_j$. Let $OPT_{i...j}$ be the cost of and min-cost parenthesization of multiplying $A_iA_{i+1}...A_j$.

Given the notation $OPT_{1...n}$. Consider $OPT_{1...n}$, and think about the very last multiplication. There are several cases:

1. We have $OPT_{1...n} = (A_1)(A_{2...n})$.
2. We have $OPT_{1...n} = (A_1...2)(A_{3...n})$.
3. ...
4. We have $OPT_{1...n} = (A_{1...(n-1)})(A_n)$.

There are $n-1$ cases in total. Suppose we are in case k, that is, we have $OPT_{1...n} = (A_{1...k})(A_{(k+1)...n})$. Then we can see that $A_{1...k}$ and $A_{(k+1)...n}$ must perform the optimal solution for each subproblem, otherwise $OPT_{1...n}$ is not optimal. And we can find the cost based on this parenthesization.

The recurrence relationship is as follows:

$$OPT_{1...n} = \min \begin{cases} OPT_{1...1} + OPT_{2...n} + d_0d_1d_n \\ OPT_{1...2} + OPT_{3...n} + d_0d_2d_n \\ ... \\ OPT_{1...k} + OPT_{(k+1)...n} + d_0d_2d_n \\ ... \\ OPT_{1...n-1} + OPT_{n...n} + d_0d_{n-1}d_n \end{cases}$$

There are $\binom{n}{2}$ subproblems, one for each (i, j) such that $i \leq j$.

```
Matrix-Multiplication-Order(d_0, d_1, ..., d_n)
d_0, d_1, ..., d_n: dimensions of matrices. The first matrix is a d_0 × d_1 matrix, and so on.
    S[n][n]: the solution array of n × n, in which S[i][j] is the cost of optimal way of multiplying S_i...S_j.
    for i = 1 to n
        S[i][i] = 0
        S[i][i + 1] = d_{i-1} d_i d_{i+1}
    for i = 1 to n
        for j = i + 1 to n
            min_ij = ∞
            for k = i + 1 to j − 1
                min_ij = min{min_ij, S[i][k] + S[k + 1][j] + d_{i-1} d_k d_j}
            S[i][j] = min_ij
    return S[1][n]
```

But this does not work. We will be using some elements that has not been computed yet. To fix this, we will change the order of the loop. We can either compute diagonal by diagonal or row by row from bottom. Here is how we compute diagonal by diagonal.

```
Matrix-Multiplication-Order(d_0, d_1, ..., d_n)
d_0, d_1, ..., d_n: dimensions of matrices. The first matrix is a d_0 × d_1 matrix, and so on.
    S[n][n]: the solution array of n × n, in which S[i][j] is the cost of optimal way of multiplying S_i...S_j.
    for i = 1 to n
        S[i][i] = 0
        S[i][i + 1] = d_{i-1} d_i d_{i+1}
    for len = 3 to n
        for r = 1 to n − len
            c = r + len − 1
            min_cr = ∞
            for k = 1 to len
                j = r + k − 1
                min_cr = min{min_cr, S[r][j] + S[j + 1][c] + d_{r-1} d_k d_c}
            S[c][r] = min_cr
            c++
    return S[1][n]
```

To backtrack, we should record the index k that gives the minimum value for all cases (i, j), and look up the table to reconstruct the parenthesization.

CS341: Algorithms **Spring 2020**

Lecture 13: June 23

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 13.39. Problem of **0/1 Knapsack**
Input: n items $\{o_1, ..., o_n\}$, values $\{v_1, ..., v_n\}$, weights $\{w_1, ..., w_n\}$, knapsack capacity W.
Output: a subset $S \subseteq \{o_1, ..., o_n\}$ such that $\sum_{i \in S} v_i$ is maximized while keeping $\sum_{i \in S} w_i \leq W$

Our intuition might lead us to a greedy algorithm. There are several of them, they are incorrect:

1. Highest value first
2. Lightest first
3. Highest value per weight first. Quite similar to the job scheduling problem.

Let's assume W, and w_i for all $1 \leq i \leq n$ are all positive integers.

Suppose we have an optimal solution S^*. Consider the last object, with index n. There are two cases: $o_n \in S^*$ or $o_n \notin S^*$.

1. if $o_n \notin S^*$, then S^* is optimal for the subproblem of $\{o_1, ..., o_{n-1}\}$, with the same capacity W. This can be proved simply by contradiction.
2. if $o_n \notin S^*$, then $S^* \setminus \{o_n\}$ is optimal for the subproblem of $\{o_1, ..., o_{n-1}\}$, with the capacity $W - w_n$. Similarly, this can be proved simply by contradiction.

Let's define $OPT_{i,c}$ be the optimal solution for the subproblem that contains $\{o_1, ..., o_i\}$ with capacity c.

In terms of recurrence relationship, $OPT_{i,c} = \max \begin{cases} OPT_{i-1,c} \\ OPT_{i-1,c-w_i} \cup \{o_i\} \end{cases}$

Hence the algorithm would be

```
0/1 Knapsack(O, V, w, W)
O: the n items{o_1, ..., o_n}
V: values for each item {v_1, ..., v_n}
w: weights for each item {w_1, ..., w_n}
W: Capacity
    Suppose we have a solution array A of dimension n × W, initially 0
    for i = 1 to n
        for c = 1 to W
            A[i][c] = max{A[i − 1][c], A[i − 1][c − w_i] + c_i}
    return A[n][W]
```

The runtime of this algorithm is clearly $O(nW)$. Did we prove this algorithm can solve "knapsack with integer weights" tractible (i.e. in polynomial time)? No. Actually $n \times W$ is not in polynomial time. Specifically, it has to be in polynomial time with respect to input size, i.e. number of bits to represent the input.

The input size for this problem will be: n values for $v_1, ..., v_n$, this can be represented in $n \log(max\_value)$ bits; n values for $w_1, ..., w_n$, this can be represented in $n \log(max\_weight)$ bits. However, we need $\log W$ bits to represent W. This is not polynomial, it's actually exponential in $\log W$. So $O(nW)$ is not in poly-time, it's in pseudo poly-time.

Definition 13.40. a **graph** G is a pair of V, a set of nodes, and E, a set of edges. $G = (V, E)$. It's a natural way to represent connected data (which is the case for many applications).

Terminology:

- directed: edges have directions
- undirected: edges do not have directions
- simple graph: all pair of vertices can have at most 1 edge between them
- multigraph: parallel edges can exist
- cyclic: the graph contains a cycle
- acyclic: there are no cycles
- connected graph: "in one piece"
- unconnected graph: can have "multiple disconnected pieces"

Conventions

- $|V| = n$
- $|E| = m$

If graph $G = (V, E)$ is undirected, the maximum number of edges is $\binom{n}{2} = O(n^2)$. If G is directed, then maximum number of edges is $2\binom{n}{2} = n(n-1) = O(n^2)$. If G is undirected and connected, the minimum number of edges is $n - 1$.

So for the problems that take G as connected and simple graphs, we have $n - 1 \leq m \leq n^2$. So $\log n \leq \log m \leq 2 \log n$. So $\log m \in \Theta(\log n)$.

We have a convention that if a runtime is run in $\Theta(n \log m)$ time, we usually say it runs in $\Theta(n \log n)$ time for simplicity.

Definition 13.41. **Degree** of a vertex v:
out-deg(v) is the number of outgoing edges from v
in-deg(v) is the number of incoming edges to v
For undirected graph, we simply use deg(v), as the number of edges connected with v

Graph Storage Formats:

- Adjacency Matrix: a $n \times n$ matrix. If (i, j) is 1, this means that there is an edge from i to j, or simply there is an edge between i and j for undirected graph. For example, $\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ this means that: there is an edge from 1 to 2, 3, 4, respectively; there is an edge from 2 to 1, and there is an edge from 3 to 4. This method takes $O(n^2)$ space, but as we look up to see if (u, v) is in E, it's essentially 1 operation. If we want to iterate over u's out/in edges, it will take $O(n)$ time.
- Adjacency List Format: We will have a list of vertices, which vertex corresponds to a linked list which contains all vertices that this vertex is adjacent to. The space would be $O(n + m)$. To look up (u, v), this takes $deg(u)$ operations. Iterating over u will $deg(u)$ time.

Theorem 13.42. HandShaking Lemma:

$$\sum_{v=1}^{n} out - deg(v) = m$$

If graph is undirected, then

$$\sum_{v=1}^{n} deg(v) = 2m$$

CS341: Algorithms **Spring 2020**

Lecture 14: June 25

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 14.43. BFS: **Breadth-First Traversal**

- starts from s and traverses the graph in waves
- $s \to s'$ first degree n_1 branches $\to$ s's second degree n_2 branches $\to \ldots$

Definition 14.44. DFS: **Depth-First Traversal**

- from s tries to go "as far as" it can "as fast as" is can
- backtrack when stuck

```
BFS: Breadth-First Traversal (G = (V, E))
        mark all vertices as not-visited
        for each v ∈ V
            if v is not visited then
                do BFS(G, v)

BFS(G, s):
        let Q be an empty queue
        mark s visited
        enqueue(s, Q)
        while(Q is not empty)
            let v = dequeue(Q)
                for each neighbour u of v
                    if (u is not visited) then
                        mark u as visited
                        enqueue(u, Q)
                    mark v as finished
```

This algorithm runs in $O(n + m)$ time.

```
DFS: Depth-First Traversal (G = (V, E))
        mark all vertices as not-visited
        for each v ∈ V
            if v is not visited then
                do DFS(G, v)

DFS(G, s):
        mark s visited
        for each neighbour v of s
            if (u is not visited) then
                DFS(G, v)
            mark v as finished
```

Properties of BFS and DFS

- both run in linear time
- starting with vertex v, will reach all vertices t such that there is an $s - t$ path in G.

Definition 14.45. **Parent of** v, $p(v)$, is the vectex u that was being traversed when v was first seen/visited.

Definition 14.46. **BFS Tree** (V_T, E_T), is the graph such that

- $V_T = V$
- $E_T = \{(p(v), v) : v \in V_T\}$

This algorithm runs in $O(n + m)$ time as well.

Application of BFS:

1. unweighted single source shortest paths
2. bypartiteness, or aka 2 coloring
3. Connected Component in undirected graphs
4. Topological Sort

Definition 14.47. Problem of **unweighted single source shortest paths**
Input: $G = (V, E)$ (directed or undirected), source vertex $s \in V$
Output: for all $v \in V$, output the shortest $s - v$ paths, and shortest distances $dist(s, v)$

Just run BFS is sufficient to solve this problem. The distance would be the level where we reach v. Runtime would be $O(n+m)$. We may construct a path using the BFS tree, just by following the parent of each node.

Definition 14.48. Problem of **Bipartite checking/2 coloring**
Input: $G = (V, E)$ (undirected)
Output: true or false to the following question: Can V be partitioned as V_1 (red) and V_2 (green) such that for all $(u, v) \in E$, u is red and v is green.

Just run BFS and see if there are any cross edges in the graph. The runtime is still $O(n + m)$

```
Bipartite-Checking(G = (V, E))
G is an undirected graph
        run BFS(G(V, E))
        given even levels red, and odd levels green
        if there exists (u, v) ∈ E such that u and v are the same color then
            return false
        else
            return true
```

Definition 14.49. Problem of **Undirected (weakly) Connected Components**
Input: $G = (V, E)$ (undirected)
Output: $CC_1, CC_2, ..., CC_k$ where each CC_i is a maximal subset of V such that: each s, t in CC_i has a path to each other, and each $v \in V$ is part of one CC_i.

Do BFS with labeling, keeping track of which are visited to record the connected component that each vertex belongs to.

CS341: Algorithms **Spring 2020**

Lecture 15: June 30

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Example 15.50. **Directed Acyclic Graphs**: DAGs. As suggested by name, directed graph without cycles.
Examples include: job scheduling in Operating Systems; CS course dependencies; Terminologies:

- source vertex: has no incoming edges
- sink vertex: has no outgoing edges

Properties:

- Every DAG has at least 1 source
- Every DAG has at least 1 sink

Definition 15.51. Problem of **Topological Sorting of a DAG**
Input: a $G = (V, E)$, which is a DAG
Output: vertices in sorted order such that each vertex appears after its dependencies.
Or equivalently, the problem is that, given an input DAG $G = (V, E)$, order the noddes such that if $(u, v) \in E$, then u appears before v.

```
Topological-Sorting-Algo-1 (G = (V, E))
G = (V, E) a DAG, input to the problem
    let result = anemptylist
    while G is not empty
        let v be a source node in G
        addv to result
        remove v from G
    return result/reverse
```

We can also pick a sink iteratively. The runtime for this algorithm is $O(n^2)$ since we need to find source by looping over V. However, this could actually be done in $O(n + m)$, because we create new sources by removing vertex. We know what we are removing, so we can know which vertices might be new sources.

Topological-Sorting-Algo-2 ($G = (V, E)$) (This uses DFS)
$G = (V, E)$ a DAG, input to the problem
run `DFS(`G`)`
put each finished vertex into an array in reverse order.
(We need to keep track of the finish time/the time get stucked and sort them in reverse order)

The runtime of this algorithm runs in $O(n + m)$ time.

```
DFS with Finish Time Keeping:
global variable t = 1
DFS(G = (V, E)):
    f: an array of size n, initialized to null.
    for i = 1 to n
        if V[i] is not yet visited then
            DFS(G, i)

DFS(G = (V, E), u):
    mark u as visited
    for all neighbors v of u
        if v is not yet visited then
            DFS(G, v)
    f[u] = t
    t++
```

Theorem 15.52. In a DAG, if $u \to v$ (there is an edge from u to v), then $f[u] > f[v]$

Proof 15.53. There are two cases:

1. if u is visited first, then: `DFS(`v`)` call will be made before `DFS(`u`)` finishes, because DFS call is made on every edge from u. So $f[u] > f[v]$.
2. if v is visited first, then: Since the graph is acyclic, then `DFS` cannot discover u from v. So v has to finish before even discovering u. Hence, $f[v] < s[u] < f[u]$ (where $s[u]$ is a discovery time of u). So $f[u] > f[v]$.

□

Corollary 15.54. In a DAG, if $u \rightsquigarrow v$ (there is an path from u to v), then $f[u] > f[v]$

Proof 15.55. Similar proof to the above theorem, with induction on the vertices in the path. □

`Topological-Sorting-Algo-2` yields the correct result. By the theorem, we know that if (u, v) exists, then $f[u] > f[v]$. We order according to decreasing f values, therefore u will be ordered before v.

Definition 15.56. **Strongly Connected Components**: SCC
Given a directed graph $G = (V, E)$, we say that u and v are strongly connected if and only if $u \rightsquigarrow v$ and $v \rightsquigarrow u$ (There is a path from u to v and there is a path from v to u).
A SCC of G is a set $C \subseteq V$ such that

- for all $(u, v) \in C$, u and v are strongly connected
- C is non-empty and maximal, i.e. for all v' in $V \setminus C$, v' is not strongly connected with any vertex in C.

Definition 15.57. **SCC graph of G**:
Given a directed graph $G = (V, E)$, we say that G^{SCC} as the meta-graph of its SCCs, or SCC graph of G.

Theorem 15.58. G^{SCC} is a DAG.

Proof 15.59. Suppose there is a cycle, then this cycle would be a new big SCC, and the original SCCs that forms the cycle is not maximal. Contradiction. □

Can BFS/DFS identify SCCs? We cannot blindly apply them. It would be dependent on the starting vertex. If we start on some sink SCCs (or more rigorously, start a BFS/DFS from vertices in reverse topological order of G^{SCC}), then BFS/DFS would give the correct result.

Let's define that "the finishing time of an SCC_i" in DFS traversal be the maximum finishing time of the vertices in SCC_i, i.e.

$$f[SCC_i] = \text{ max over } v \in SCC_i f[v]$$

Theorem 15.60. in G^{SCC}, if $SCC_i \to SCC_j$, then $f[SCC_i] > f[SCC_j]$.

Proof 15.61. Let u be the first vertex visited in SCC_i or SCC_j. There are two cases:

1. u is in SCC_i, then by extended key lemma of Topological Sort (Theorem 15.52), $f[u]$ will be greater than any vertex in SCC_j.

2. u is in SCC_j, then by extended key lemma of Topological Sort (Theorem 15.52), $f[u] > f[y]$ for any vertex in SCC_j, and $f[u] < f[x]$ for any vertex in SCC_i, and hence $f[SCC_i] > f[SCC_j]$.

□

CS341: Algorithms **Spring 2020**

Lecture 16: July 2

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Continuing from the last lecture, we need to do some preprocessing to guarantee that we start a traversal from vertex from a sink SCC.

It seems like we are able to determine the source SCC by their finishing time. But our goal is to identify sink SCCs. So, if we reverse the graph, SCC stays there as the same. Sink SCCs becomes source SCCs. And we will have only one sink SCC, which was our original source SCC. The propertoes of G_{rev} are:

1. G_{rev} and G have the same SCCs
2. The $(G_{rev})^{SCC}$ is the reverse of G^{SCC}

Kosaraju's Algorithm ($G = (V, E)$)
$G = (V, E)$ is a directed graph
    run `DFS` on G_{rev} and keep finishing times f
    run `DFS/BFS` in G:
        pick start vertices by decreasing f times
        propagate the ID of each new start vertex during traversal
(The labels will implicitly indentify SCCs.)

The runtime of the above algorithm is $O(n + m)$. For correctness, recall Theorem 15.60. We pick `BFS/DFS` sources in reverse topological order of the SCCs in G, so we peel off exactly one sink SCC with each `BFS/DFS`.

Definition 16.62. **Tree**: a undirected graph $G = (V, E)$ that is

- acyclic
- connected

Or from MATH239, a undirected connected graph contains no cycle.

Properties of trees include:

Lemma 16.63. Breaking a tree Lemma: Removing any edge from a tree T disconnects T.

Proof 16.64. Suppose not, and removing a edge (u, v) to get T'. If T' is still connected, then there exists a $u - v$ path in T, then with this removed edge (u, v), we get a cycle in the original T. Contradiction. □

Lemma 16.65. Breaking a cycle Lemma: If a undirected connected graph $G = (V, E)$ has a cycle $C \in E$, then removing an edge (u, v) from C cannot disconnect G.

Proof 16.66. Suppose not, for the sake of contradiction. Let's say $C = w_1...w_n w_1$ is the cycle contained in G. Consider any two nodes s, t and a $s - t$ path p. If p does not contain $w_1, ..., w_n$, then p still exists. If p contains any of $w_1, ..., w_n$, let's say we removed (w_i, w_{i+1}), then we can still use the remaining part of the cycle as the alternative path p' which still connects s and t. □

Lemma 16.67. Cycle Creation Lemma: Adding a new edge into a tree T creates a cycle.

Proof 16.68. Because T is a tree, there is a path p from u to v already in T. Adding a (u, v) edge closes the cycle. □

Theorem 16.69. Every tree of n vertices contains exactly $n - 1$ edges.

Theorem 16.70. Every graph of $n - 1$ edges and n nodes that contains no cycle, is a tree.

Theorem 16.71. Every graph with $n - 1$ edges that connect n nodes is acyclic.

Theorem 16.72. The two out of three theorem: Let $G = (V, E)$ be a graph. Any two of the following imply the remaining one:

- G is connected
- G is acyclic

- The number of edges is exactly one fewer than the number of vertices.

Definition 16.73. Problem of **Finding Minimum Spanning Tree**:
Input: A undirected connected graph $G = (V, E)$ with arbitrary edge weights W_e
Output: a tree T^* of V such that $w(T^*) \leq T$ for any other tree T of V, where $w(T) = \sum_{e \in T} w_e$
Note that "spanning" is redundant since by definition trees are connected, but it is used in the name of the problem to emphasize that we are connecting these n nodes.

Assumptions are:

- G is connected already
- Edge weights are distinct

CS341: Algorithms **Spring 2020**

Lecture 17: July 7

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Prim's Algorithm to find MST ($G = (V, E)$):
$G = (V, E)$: an undirected connected graph
    choose any $s \in V$
    let $L = \{s\}$, $R = V \setminus \{s\}$
    let $T = \emptyset$ for storing final tree
    for $i = 1$ to $n - 1$
        let (u, v) be the minimum cost edge crossing L to R
        remove v from R and add to L
        add (u, v) to T
    return T

Prim's Algorithm ($G = (V, E)$):
$G = (V, E)$: an undirected connected graph
    choose any $s \in V$
    let $L = \{s\}$, $R = V \setminus \{s\}$
    let $T = \emptyset$ for storing final tree
    let MWE be a priority queue, storing that for all $v \in R$, the minimum weight of the edges (u, v) connecting v to L
    for $i = 1$ to $n - 1$
        $(v, (u, v)) = MWE.extractmin$ $\Rightarrow O(n \log n)$
        R.remove(v) $\Rightarrow O(n)$
        L.add(v) $\Rightarrow O(n)$
        for each neightbour z of V in R $\Rightarrow O(m \log n)$
            if $w(v, z) <$ z's weight in MWE **then**
                z.decrementkey($w(v, z)$)
        remove v from R and add to L
        add (u, v) to T
    return T

The runtime of this algorithm is $O(m \log n)$. The proof of correctness will be presented after we introduce some important results.

Definition 17.74. **Cut**: a cut of $G = (V, E)$ is a partition of V into 2 non-empty sets.

Lemma 17.75. Empty Cut Lemma: a graph $G = (V, E)$ is disconnected if and only if there exists a cut (X, Y) with no crossing edges.

Lemma 17.76. Double Cut Crossing Lemma: Suppose a cycle $C \subseteq$ has a edge e crossing a cut (X, Y), then there is another edge $e' \in C$ that is also crossing the same cut (X, Y)

Corollary 17.77. Lonely Cut Edge Corrolary: If there is a cut (X, Y) with only one edge crossing it, then e is not part of the cycle.

Lemma 17.78. MST Cut Lemma: Consider any cut (X, Y) in $G = (V, E)$, and let e be the min-cost edge crossing (X, Y), then e belongs to every MST.

Proof 17.79. Suppose there exists n MST T^* that does not contain $e^* = (u, v)$. Let $T^{*'} = T^* \cup \{(u, v)\}$, then by Cycle Creation Lemma (Lemma 16.67), $T^{*'}$ has a cycle that contains (u, v). Then by Double Cut Crossing Lemma (Lemma 17.76), there exists e' that is also crossing (X, Y).

Then let's remove e' from $T^{*'}$, yielding $T^{**} = T^{*'} \setminus \{e'\}$. By Breaking a Cycle Lemma (Lemma 16.65), T^{**} is still connected, and it has exactly $n - 1$ edges. Therefore, by two out of three theorem (Theorem 16.72), T^{**} is another tree, but since $w(e^*) < w(e')$, we have $w(T^{**}) < w(T^*)$, contradicting that T^* was an MST, completing the proof that e^* is in a MST. □

Proof 17.80. Prim's algorithm's correctness proof: we need to prove:

- T_{prim} is a tree connecting V
- T_{prim} is a MST

For part one, let $T_{prim,i}$ be the set of edges after i iterations of the algorithm. Let L_i be the set of vertices in L after i iterations. We may claim that $T_{prim,i}$ connects the vertices in L_i, prove by induction. Therefore, at termination, $T_{prim} = T_{prim,n-1}$, connects $L_{n-1} = V$ and contains exactly $n - 1$ edges. By the two out of three theorem (Theorem 16.72), T_{prim} is a tree connecting V.

For part two, we will use the MST Cut Lemma. T_{prim} is an MST, follows directly from the lemma and the construction process in the algorithm. By the lemma, all the edges we found is in an MST, and we precisely have $n - 1$ edges, and as part one we proved that T_{prim} is a tree. So T_{prim} must be a MST. □

CS341: Algorithms **Spring 2020**

Lecture 18: July 9

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Kruskal's Algorithm to find MST ($G = (V, E)$):
$G = (V, E)$: an undirected connected graph
    Without loss of generality, let $w(e_1) < w(e_2) < ... < w(e_m)$
    (We may need to sort the edges beforehand)
    let $T = \emptyset$ for storing final tree
    let $CC =$ be a union-find structure, initialized to in different components
    for $i = 0$ to m
        if $T \cup \{e_i = (u, v)\}$ does not create a cycle **then**
        (i.e. **if** $find(u) \neq find(v))$ $O(\log n)$
            $T = T \cup \{e_i = (u, v)\}$
            (i.e. $union(u, v)$)
    return T

The runtime of this algorithm depends on the efficiency in checking if adding an edge creates a cycle. We will use a union-find data structure to keep track of the connected components that we are creating in T. We will need two operations supported: $find(x)$ gives the component that x belongs to, and $union(x, y)$ merges the sets C_x and C_y. We are essentially using a tree to do that. If we union two components, we merge two trees: smaller set points to the larger set.

Proof 18.81. Kruskal's algorithm's correctness proof: we need to prove:

- T_{krusk} is a spanning tree
- T_{krusk} is a MST

For part one, T_{krusk} does not contain a cycle by construction. It is spanning V by the Empty Cut Lemma (Lemma 17.75): we will argue that no matter which cut (X, Y) we pick, there is a crossing edge.

Consider an arbitrary cut (X, Y). Since the graph is connected, there must be some crossing edges between X and Y. Let e^* is the min-cost edge crossing (X, Y), then we claim that $e^* \in T_{krusk}$. Since when we encounter this edge, we won't create a cycle, and we will be adding this into T_{krusk}.

For part two, let (u, v) be any of the $n - 1$ edges in T_{krusk}. We want to show that (u, v) is a min-cost edge crossing some cuts. Consider the point in time when Kruskal's algorithm add (u, v) to T_{krusk}. We know that at this point, u and v belong to different components. The edge we picked would be of min-cost since none of the crossing edges are in T_{krusk} (not inspected by Kruskal yet, they have larger weight). Adding (u, v) would not create a cycle, and it is of min-cost by construction. By the MST Cut Lemma(Lemma 17.78), (u, v) belongs to every MST. So every edge in T_{krusk} belongs to every MST. Therefore, T_{krusk} is the MST. □

Definition 18.82. Problem of **Single Source Shortest Paths**
Version 1: In DAG, with arbitrary edge weights
Input: A DAG $G = (V, E)$ with arbitrary edge weights and a source s
Output: Shortest paths (and distances) from s to all $v \in V$

We may use dynamic programming to solve this problem. Consider the last edge (u, v)

$$SD(e) = \min \begin{cases} SD(b) + w(b, e) \\ SD(c) + w(c, e) \\ SD(d) + w(d, e) \end{cases}$$

Or in general, $SD(v) = \min_{(u,v)} SD(u) + w(u, v)$

```
DP-Shortest-Path Algorithm(G = (V, E), w, v)
        sort G topologically      ⇒ O(n + m)
        let SD[n] = ∞
        let SD[s] = 0
        for v ∈ G in topological order(start from left)    ⇒ O(n + m)
              SD[v] = min_(u,v) SD[u] + w(u, v)
        return SD
```

We can do backtracking to reconstruct the actual shortest path.

CS341: Algorithms **Spring 2020**

Lecture 19: July 14

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 19.83. Problem of **Single Source Shortest Paths**
Version 2:
Input: A directed or undirected $G = (V, E)$ with non-negative edge weights and a source s
Output: Shortest paths (and distances) from s to all $v \in V$
Also assume that s, v-path exists ($s \rightsquigarrow v$) for all $v \in V$.

Dijkstra's Algorithm $(G = (V, E), s)$:
- **let** $L = \{s\}$, $R = V \setminus \{s\}$
- **let** SD =array of length n, initialized to NULL
- **let** $parent$ =array of length n, initialized to NULL
- **let** $DistSoFar$ = a priority queue
- $DistSoFar[s] = 0$
- $DistSoFar[v] = \infty$ for $v \neq s$
- **for** $i = 1$ to $n - 1$ $\quad \Rightarrow \sum_{v \in V} deg(v) = O(m)$
  - **let** $v^* = DistSoFar.extractmin()$
  - $R.remove(v^*)$ $\quad \Rightarrow O(n \log n), O(n)$ times, $O(\log n)$ each time
  - $L.add(v^*)$
  - **for** each (v^*, z) such that $z \in R$
    - $newDistToz = SD[v^*] + w(v^*, z)$
    - **if** $newDistToz \leq DistSoFar[z]$ **then**
      - $DistSoFar.decrement(z, newDistToz)$ $\quad \Rightarrow O(m \log n), O(m)$ times, $O(\log n)$ each time
      - $parent[z] = v^*$
- **return** $(SD, parent)$

```
Dijkstra's Algorithm (G = (V, E), s): Correct version from wikipedia
    let Q be a priority queue
    for each v ∈ V
        dist[v] = ∞
        prev[v] = UNDEFINED
        Q.add(v, dist[v])
    dist[s] = 0
    while Q is not empty
        let u = Q.pop() (the vertex with minimum dist[u])
        for each neighbour v ∈ Q of u
            let alt = dist[u] + w(u, v)
            if alt < dist[v] then
                dist[v] = alt (update in Q)
                prev[v] = u
    return dist[], prev[]
```

The runtime of the above algorithm is $O(m \log n)$.

Proof 19.84. Correctness of Dijkstra's Algorithm: by induction on the number of iterations i.

We claim that at each iteration i:

1. For all $v \in L$, $SD[v]$ and $parent[v]$ are correct
2. For all $v \in R$, $DisSoFar[v]$ is the distance of the shortest s, v-path that only uses the nodes in L as intermediate nodes. And $parent[v]$ is also correct.

Base Cases: $i = 0$, $L = \{s\}$, $SD[s] = 0$, so (1) is true. And we loop over s's neighbours and update their distance to the $w(s, z)$ and keep the minimum. So we inspect all possible paths from s to z, which are the only possible paths from s to z that use only s as intermediate nodes, so (2) is also true.

Inductive Hypothesis: The claim holds for $i \leq k$.

Inductive Conclusion: we need to prove the claim holds for $i = k + 1$. Let $v^* \in R$ be the vertex picked at iteration $i = k + 1$, i.e. $DistSoFar[v^*] \leq DistSoFar[v]$ for all $v \in R$.

Let $parent[v^*] = u^*$. Consider any other path $P' = s \rightsquigarrow \ell \to r \rightsquigarrow v^*$, suppose the path we have right now is $P = s \rightsquigarrow u^* \to v^*$. We know that

$$\begin{aligned} cost(P') &= cost(s \rightsquigarrow \ell \to r) + cost(r \rightsquigarrow v^*) \\ &\geq DistSoFar[r] \quad \text{because } r \in R, \text{ and by IH } DistSoFar[r] \text{ is the shortest } s, r\text{-path that only uses} \\ &\quad \text{nodes in } L \text{ as intermediate nodes}, cost[s \rightsquigarrow \ell \to r] \geq DistSoFar[r], \text{ and } cost(r \rightsquigarrow v^*) \geq 0 \text{ by assumption} \\ &\geq DistSoFar[v^*] \\ &= cost(P) \end{aligned}$$

So $cost(P') \geq cost(P)$, and Dijkstra's algorithm correctly sets the $SD[v^*] = DistSoFar[v^*]$ and updates the costs and neighbours of v^*. □

Definition 19.85. Problem of **Single Source Shortest Paths**
Version 3:
Input: A directed or undirected $G = (V, E)$ with negative edge weights and a source s. But G does not have negative weight cycles.
Output: Shortest paths (and distances) from s to all $v \in V$
Also assume that s, v-path exists ($s \rightsquigarrow v$) for all $v \in V$.

If a graph does contain negative weight cycles, how can we define shortest paths?

- We may allow paths to contain cycles. But we can always make paths shorter by taking more laps of the negatively weighted cycles. This problem is not well defined
- We may disallow paths to contain cycles. This problem is well-defined by intractible, i.e. NP-hard.
- So, let's just assume that the graph does not have negative weight cycles

The algorithm to solve this problem will do two things:

1. If the input G does not have negative weight cycles, it returns the correct shortest paths from s to every other node
2. Otherwise, it errors and detects the negative weight cycle.

CS341: Algorithms **Spring 2020**

Lecture 20: July 16

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Consider an arbitrary vertex $v \in V$, and the shortest path from s to v, $P^* = s \rightsquigarrow u \rightarrow v$. The maximum number of edges P^* can have is $n-1$.

Let $P^*_{v,i}$ be the shortest distance from s to v that uses at most i edges. If no such path exists, denote it as ∞. So, $P^*_{v,n-1}$ is the shortest distance from s to v.

There are three possibilities for the number of edges in $P^*_{v,i}$

1. There is no path from s to v that uses at most i edges
2. The number of edges in $P^*_{v,i}$ is smaller than i
3. The number of edges in $P^*_{v,i}$ is exactly i

Suppose we are in case 1 or 2, then $P^*_{v,i} = P^*_{v,i-1}$. Suppose we are in case 3, then $P^*_{v,i} = \min_{(u,v)\in E}(P^*_{u,i-1} + w(u,v))$. So the final recurrence is

$$P^*_{v,i} = \min \begin{cases} P^*_{v,i-1} \\ \min\limits_{(u,v)\in E}(P^*_{u,i-1} + w(u,v)) \end{cases}$$

Bellman Ford's Algorithm($G = (V, E), w$):
$G = (V, E)$ the graph with arbitrary edge weights
    let P be a $n \times n-1$ solution array.
    Base Cases:
    $P[s][0] = 0$, $P[v][0] = \infty$ for $v \neq s$
    for $i = 1$ to $n-1$ $\Rightarrow n$ iterations
        for $v \in V$ $\Rightarrow m$ operations overall
            $P[v][i] = P[v][i-1]$
            for $(u, v) \in E$
                $P[v][i] = \min(P[v][i], P[u][i-1] + w(u, v))$
    return P ($P[v][n-1]$ is the length of the shortest path from s to v)

The runtime is $O(mn)$. Several optimizations are

- We can reduce the number of rows of the solution array to be 2, instead of $n-1$. We only need the result for $i-1$, so 2 rows are sufficient.
- Suppose at iteration i, for all $v \in V$, $P[v][i] = P[v][i-1]$, then the algorithm has converged since our recurrence only depends on the previous iteration.

How do we detect negative cycles? We can this algorithm for one more iteration (i.e. $i = n$), and then check if $P[v][n] = P[v][n-1]$ for all $v \in V$. If not, then there is a negative weight cycle, because if `Bellman Ford's Algorithm` stablizes at some iteration, then there cannot be a negative weight cycle.

Definition 20.86. Problem of **All pairs Shortest Paths**
Input: A directed $G = (V, E)$ with arbitrary edge weights.
Output: Shortest paths (and distances) for all possible pairs (u, v) in which $u, v \in V$

We will order vertices from 1 to n, and we will only use the first i vertices in each subproblem. Let $V = \{1, 2, ..., n\}$, $V^k = \{1, 2, ..., k\}$. Let $P_{(i,j,k)}$ be the shortest i, j-path that uses only V^k as intermediate nodes, excluding i and j. Finally, we will return the result as $P_{(i,j,n)}$ for all possible pairs (i, j).

There are two possibilities to be considered, either the vertex $k \in P_{(i,j,k)}$ or $k \notin P_{(i,j,k)}$.

If $k \in P_{(i,j,k)}$, then all the internal nodes are from $\{1, ..., k-1\}$, and hence $P_{(i,j,k)} = P_{(i,j,k-1)}$, easily proved by contradiction. If $k \notin P_{(i,j,k)}$, there there is one, and only one, internal nodes that is k. Let's divide the path to be P_1 and P_2, in which $P_1 = i \rightsquigarrow k$ and $P_2 = k \rightsquigarrow j$. We know that P_1 and P_2 only contain intermediate nodes $\{1, ..., k-1\}$, and we know that P_1 and P_2 are optimal in their own cases, otherwise $P_{(i,j,k)}$ is not optimal. So we know that $P_1 = P_{(i,k,k-1)}$, and $P_2 = P_{(k,j,k-1)}$.

Floyd-Warshall Algorithm($G = (V, E), w$):
$G = (V, E)$ the graph with arbitrary edge weights
    let A be a $n \times n \times n$ solution array.
    $A[i][j][k]$ is the shortest i, j-path with V^k as intermediate nodes
    Base Cases:
    $A[i][i][0] = 0$ for all $i = 1$ to n

$$A[i][j][0] = \begin{cases} C_{i,j} & \text{if } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$$

    DP process:
    for $k = 1$ to n
        for $i = 1$ to n
            for $j = 1$ to n
                $A[i][j][k] = \min\{A[i][j][k-1], A[i][k][k-1] + A[k][j][k-1]\}$
    return $A[i][j][n]$

Trivially, this algorithm runs in $O(n^3)$ time.

To detect negative weight cycles, we just check $A[i][i][n]$ for each i. We should get 0 for each case.

Summary of shortest path problems (unweighed shortest path algorithm not shown here, for that problem, just run BFS):

	Single Source Shortest Path, DAG	Disjkstra	Bellman-Ford	Floyd-Warshall
Single Source Or All pairs	Single Source	Single Source	Single Source	All pairs
Runtime	$O(n+m)$	$O(m \log n)$	$O(mn)$	$O(n^3)$
Negative Edges	Yes	No	Yes	Yes
Negative Cycles	No	No	No but can detect	No but can detect

CS341: Algorithms **Spring 2020**

Lecture 21: July 21

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Definition 21.87. **Theory of NP-Completeness**:

The high level goal is to identify intractble, i.e. very difficult to solve, computational problems.

We will look at decision version of problems. Actually, every optimization problem has a natural decision version (and vice versa). The decision version of the problem for (let's say find MST) would be is it possible to reach the target k for the problem (in the MST case, k would be the target weight).

Definition 21.88. **Class P**: A computational probelm $C \in P$ if there is an algorithm solving C in $O(n^k)$ time for some constant k, where n is the number of bits to represent the input.

For example, $O(n), O(n \log n), O(n^3), O(n^{1000})$

The above definition is simple, and very successful in identifying tractible problems, but it also allows impractical runtime, such as $O(n^{1000})$.

Definition 21.89. Problem of **Travelling Salesman Problem**

Input: Complete graph $G = (V, E)$ of n nodes, with arbitrary edge weights, and a target tour distance k.

Output: Is there a tour with cost less than or equal to k? A tour is a cycle which traverses all nodes exactly once.

In 1965, a conjecture is made by Jack Edmonds, claiming that there is no efficient algorithmes to solve the above problem. This is equivalent to today's conjecture of $P \neq NP$. This conjecture is still open. We still cannot settle the conjecture, but if we believe this problem is a uniquely difficult problem to solve, how else can we make a cases that it is very difficult?

We may argue that TSP (Travelling Salesman Problem) is difficult in a relative sense, i.e. TSP is "as hard as" a large class of other problems.

Our tool to prove that C_2 is "as hard as" C_1 is "**Reductions**".

Definition 21.90. C_1 reduces to C_2 ($C_1 \leq_P C_2$) if given a polynomial time algorithm solving C_2, we can also solve C_1 in polynomial time.

To use this definition, we want to use a problem C_1 that we know is hard, i.e. we already know that C_1 does not have a polynomial time solution, and if we show that C_1 can be reduced to C_2, then we know that C_2 is as hard as C_1.

Note that reductions are transitive: if ($C_1 \leq_P C_2$) and ($C_2 \leq_P C_3$), then ($C_1 \leq_P C_3$), and we will use reductions to "spread hardness".

Definition 21.91. **$\mathfrak{C}$-Completeness**: Let $\mathfrak{C}$ be a set of problems, and let $c_i \in \mathfrak{C}$.

If for all $c_k \in \mathfrak{C}$, $c_k \leq_P c_i$, then c_i is $\mathfrak{C}$-Complete.

Our goal is to show that TSP is $\mathfrak{C}$-complete for some larget set $\mathfrak{C}$- of computational problems. We need to pick an appropriate set $\mathfrak{C}$ of problems. It cannot be the set of all problems, since there are undecidable problems, such as halting, but TSP is decidable. In the worst case, we can brute-force to solve TSP.

We will have an alternative ambitious set: the set of brute-force solvable problems. And we call this set $\mathfrak{C} = NP$.

Definition 21.92. A problem $C \in NP$ if:

- correct solution to the problem has polynomial length
- instances with an answer YES can be verified within polynomial time (possibily using a correct solution)

Actually, every computational probelm we've seen so far is in NP.

So, an NP-complete problem is the hardest problem in NP. There are thousands of NP-complete problems in the real life.

Our goal is to show TSP is NP-complete, i.e. if we find a magic algorithm solves TSP in polynomial time, then all NP problems can be solved in polynomial time.

We know that $P \subseteq NP$, but we don't know if $P = NP$. It is still a conjecture. Most people believe $P \neq NP$.

CS341: Algorithms **Spring 2020**

Lecture 22: July 23

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Theorem 22.93. Decision and optimization version of problems have, computationally, almost the same cost. Specifically, if one can solve the decision version of a problem, one can solve the optimization version almost as efficiently (and vice versa).

Using a decision version algorithm to solve the optimization version problem, we can do a binary search algorithm, which takes $\log(v^*)$ (some parameters in the input) more time (so maybe the result is $O(n^k \log v^*)$).

Definition 22.94. C^* is NP-Complete if:

- $C^* \in NP$
- Every NP problem $C \leq_P C^*$, i.e. this problem is NP-Hard.

There are two methods to show a problem C^* is NP-Hard:

1. Direct argument from the definition of class NP, that an arbitrary $C \in NP$ reduces to C^*. However, in 1971, Steven Cook proved that the CIRCUIT-SAT problem cannot be reduced.
2. NPC reduction: show that only already known NPC problem $C^{**} \leq_P C^*$, since we already know $C \leq_P C^{**}$.

To show $C_1 \leq_P C_2$, we need to have

- take an instance of C_1, $\Pi_1 \in C_1$
- have a poly-time instance converter, which converts $\Pi_1 \in C_1$ to $\Pi_2 \in C_2$
- have a magical poly-time algorithm, that outputs $S_2 \in C_2$, gives a YES or NO, for C_2
- Map the result back to $S_1 \in C_1$, which has to be a bijection (if and only if relationships).

Definition 22.95. Problem of **Hamiltonian Cycle**

Input: a undirected graph $G = (V, E)$

Output: YES if G contains a Hamiltonian Cycle, and NO otherwise.

a Hamiltonian Cycle is a cycle that contains every node $v \in V$ exactly once.

We know, as a fact, that HAM-CYCLE is NP-Complete.

Example 22.96. Suppose we know HAM-CYCLE is NP-Complete, and we want to show TSP is NP-Complete as well.

First of all, TSP $\in$ NP, this is proved in the last lecture.

Now we want to prove HAM-CYCLE $\leq_P$ TSP. There are two things I need to do: provide a poly-time instance converter, and give the final map of outputs.

The instance converter would be: $\Pi_{HC}(G = (V, E)) \to \Pi_{TSP}$, Π_{TSP} would be a complete graph G' with edge weights, and a target cost k.

Let G' be the graph that adds every missing edge in G. For all original edges in G, we give its weight as 0, and for all missing edges, we give its weight as 1, i.e. $w(u, v) = \begin{cases} 0 & \text{if edge } (u, v) \in G \\ 1 & \text{if edge } (u, v) \notin G \end{cases}$

The target k would be 0. So that we can force the algorithm to use all edges in the original graph. This instance converter is clearly poly-time.

Now we claim that if there is a hamiltonian cycle in G if and only if there is a tour in G' with cost less than or equal to 0.

The proof of forward direction: if there is a hamiltonian cycle in G, all edges in C get cost 0, then C is a tour in G' with cost less than or equal to 0. So we would output YES in this case.

The proof of backward direction: if there is a tour C' in G' with cost less than or equal to 0, then all edges in C' must have weight 0, so they all exists in G. Thereforce C' is a hamiltonian cycle in G. So we would output YES in this case.

CS341: Algorithms **Spring 2020**

Lecture 23: July 28

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Example 23.97. Example of Reduction, from independent set to vertex covering.

Recall the problem of independent set:

Input: a undirected graph $G = (V, E)$ and a target independent set of size k

Output: YES if G contains an independent set of size greater than or equal to k, i.e. there exists a subset $S \subseteq V$ such that no pair of vertices have an edge between them, and NO otherwise.

Recall the problem of vertex covering:

Input: a undirected graph $G = (V, E)$ and a target k

Output: YES if there exists a subset $S \subseteq V$ of size at most k such that every edge is covered, i.e. for all $(u, v) \in E$, either $u \in S$ ot $v \in S$, and NO otherwise.

We claim that the problem of vertex covering(VC) is NP-Complete.

Proof:

Step 1: To prove $VC \in NP$. It is easy to verify the solution in polynomial time because solutions are subsets $S \subseteq V$, which can be encoded in poly-length. If (G, k) is YES, then given a S of size less than or equal to k that cover each edge in E, we can use $O(mk)$ time to verfiy that for each edge $e = (u, v) \in E$, either u or v is in S.

Step 2: an poly-time instance converter from IS to VC: We just change the target value to be $n - k$. The input of IS is $G = (V, E), k$, and the input of VC would be $G = (V, E), n - k$.

Step 3: Prove that if there exists an IS of size k in G if and only if there is a VC of size $n - k$ in G. The key idea is that $S = IS$ means that no edge $e = (u, v) \in E$ is between vertices in S, so either u or v must be touching $V \setminus S$, which is of size $n - k$, and vice versa.

Example 23.98. Example of Reduction, from independent set to clique.

Recall the problem of independent set:

Input: a undirected graph $G = (V, E)$ and a target independent set of size k

Output: YES if G contains an independent set of size greater than or equal to k, i.e. there exists a subset $S \subseteq V$ such that no pair of vertices have an edge between them, and NO otherwise.

The problem of clique:

Input: a undirected graph $G = (V, E)$ and a target k

Output: YES if there exists a subset $S \subseteq V$ of size at least k such that for all $u \in S, v \in S$, $(u, v) \in E$, i.e. S is a complete graph (clique), and NO otherwise.

We claim that the problem of clique is NP-Complete.

Proof:

Step 1: To prove clique$\in NP$. It is easy to verify the solution in polynomial time because solutions are subsets $S \subseteq V$, we can check if a solution S is a clique in $O(n|S|^2)$ time using a brute force search.

Step 2: an poly-time instance converter from IS to clique: We just change the graph to be its complement. The input of IS is $G = (V, E), k$, and the input of VC would be $G^C = (V, E^C), k$. This can be done in $O(n^2)$ time, obviously.

Step 3: Prove that if there exists an IS of size at least k in G if and only if there is a clique of size at least k in G^C (S itself is a clique in G^C).

Example 23.99. Example of Reduction, from 3-CNF SAT (Boolean satisfiability problem) to indepentdent set.

The problem of 3-CNF SAT:

Input: a boolean formula ϕ consisting of

- n boolean variables $x_1, x_2, ..., x_n$, which are true or false
- m clauses connectives, which are $\wedge(AND), \vee(OR), \neg(NOT)$ such that
  - each clause C_j has 3 different literals, i.e. a variable x_i or its neg $\neg x_i$
  - ϕ is in "conjecture normal form", i.e. each clause is an "OR" or three literals, and the entire formula is m of these clauses.

Output: YES if there exists an assignment of True/False to X_i such that ϕ is true, and NO otherwise.

For example, a ϕ could be $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$. This is satisfiable by taking $x_1 = T, x_2 = F, x_3 = F$.

Recall the problem of independent set:

Input: a undirected graph $G = (V, E)$ and a target independent set of size k

Output: YES if G contains an independent set of size greater than or equal to k, i.e. there exists a subset $S \subseteq V$ such that no pair of vertices have an edge between them, and NO otherwise.

We claim that the problem of independent set is NP-Complete.

Proof:

Step 1: To prove independent set $\in NP$. It is easy to verify the solution in polynomial time because solutions are subsets $S \subseteq V$, we can check if a solution S is an independent set (check none of the edges between vertices in S exists) in polynomial time using a brute force search.

Step 2: an poly-time instance converter from 3-CNF SAT to IS:

For example, if we have a $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)$, then each clause C_i gives us a "triangle gadget". Then for each node pairs, x_i and $\neg x_i$, have an edge between them (inter-triangle edges), call them "consistency edges". This can be done clearly in poly-time. Let the target for IS to be m.

Step 3: Prove that ϕ is satisfiable if and only if there is an independent set S of size at least m in G.

Could there be an independent set of size greater than m in G? No, because we have m triangles, by construction we can pick at most 1 node from each triangle gadget.

For the proof of the forward direction: if ϕ is satisfiable, then there is an independent set S of size exactly m in G. This is because there exists an assignment $A = (x_1, ..., x_n)$ where $x_i = T/F$ for all i, such that each clause C_i is satisfied. Then to construct S, pick one and only one of the literals ℓ that make C_i true, put the node corresponding node for L into S. Then we claim that S is independent and of size m.

It is clearly of size m since we are picking one variable from each triangle. By construction, we cannot have picked an x_i and $\neg x_i$ in S because any assignment gives one truth value to x_i.

For the proof of the backward direction: there is an independent set S of size m in G, then ϕ is satisfiable. Then one vertex from each gadget must be picked in S. Moreover, no x_i and $\neg x_i$ can have benn picked in S. So make $x_i = \begin{cases} T & \text{if } x_i \text{ was picked in some gadgets} \\ F & \text{if } \neg x_i \text{ was picked in some gadgets} \\ T/F & \text{otherwise} \end{cases}$. Then for each C_i, there is at least one literal that makes C_i true, so ϕ is satisfiable.

CS341: Algorithms **Spring 2020**

Lecture 24: July 30

Lecturer: Semih Salihoglu *Noted By: Haochen Wu*

This lecture's notes tend to be supplementary (add-on notes) of the course notes provided.

Example 24.100. Example of Reduction, from Subset Sum to 0/1 Knapsack.

The problem of Subset Sum:

Input: $X = \{x_1, ..., x_n\}$ of integers and a target sum t

Output: YES if there exists a subset $S \subseteq X$ thay sum exactly to t, and NO otherwise.

Input: n items $\{o_1, ..., o_n\}$, values $\{v_1, ..., v_n\}$, weights $\{w_1, ..., w_n\}$, knapsack capacity W.
Output: svser $S \subseteq \{o_1, ..., o_n\}$ such that $\sum_{i \in S} v_i$ is maximized while keeping $\sum_{i \in S} w_i \leq W$

The problem of 0/1 Knapsack:

Input: n items $O = \{o_1, ..., o_n\}$, values $\{v_1, ..., v_n\}$, weights $\{w_1, ..., w_n\}$, target knapsack capacity W, target value V

Output: YES if there exists a subset $S \subseteq \{1, ..., n\}$ such that $\sum_{i \in S} v_i \geq V$, and $\sum_{i \in S} w_i \leq W$and NO otherwise.

We claim that the problem of 0/1 Knapsack is NP-Complete.

Proof:

Step 1: To prove 0/1 Knapsack $\in NP$. It is easy to verify the solution in polynomial time because solutions are subsets $S \subseteq \{1, ..., n\}$, we can sum the values to check if it is greater than or equal to V and sum the weights to check if it is less than or equal to W.

Step 2: an poly-time instance converter from Subset Sum to 0/1 Knapsack: We have $X = \{x_1, ..., x_n\}$, t. We just make $\{v_1 = x_1, v_2 = x_2, ..., v_n = x_n\}$ and $\{w_1 = x_1, w_2 = x_2, ..., w_n = x_n\}$, and make $W = t, V = t$. We set the values and weights to be the same.

Step 3: Prove that there exists a subset S that sums to t if and only if there exists a subset S in 0/1 Knapsack whose sum of values is greater than or equal to $V = t$, and whose sum of weights is smaller than or equal to $W = t$.

This is because each value has $v_i = w_i = x_i$. So S itself is the 0/1 knapsack subset whose $value = weight = t$, and vice versa.

Undecidability:

Are NP-Complete problems the most difficult problems that exist? No, there are harder problems, such as Co-NP, Polynomial Hierarchy, P-Space problems. All of them are solvable. There are problems with well defined inputs and outputs, that cannot be solved by any algorithm.

Definition 24.101. Problem of **Halting Problem** (Introduced by Church and Turing in 1936):

Input: Binary representation of an algorithm A, and an input D to A

Output: YES if A halts on D and NO otherwise (i.e. A goes into an infinite loop)

Example 24.102. We claim that Halting is uncomputable/undecidable, because the existence of an algorithm solving Halting is a paradox, i.e. self-contradictory.

A common method to construct paradoxes is self-referential statements.

Suppose such an algorithm "does Halt" exists that solves Halting.

```
does-Halt(A, D):
    if A halts on D then
        return YES
    else
        return NO
```

But them we can develop the folloing algorithm:

```
oppo-Halt(A, D)
    if does-Halt(A, D)
        go into infinite loop
    else
        halt and return -1
```

we can also develop the folloing algorithm:

```
oppo-Halt-S(A)
    if does-Halt(A, A)
        go into infinite loop
    else
        halt and return -1
```

However, Does `oppo-Halt-S` halt on itself? What does `does-Halt`$(oppo-Halt-S, oppo-Halt-S)$ return?

- if `oppo-Halt-S` return YES, then `oppo-Halt-S` will go into infinite loop, so `does-Halt` makes a mistake
- if `oppo-Halt-S` return NO, then `oppo-Halt-S` halts, so `does-Halt` makes a mistake

Therefore, `does-Halt` cannot exist. If it exists, at least on $(A = oppo - Halt - S, D = oppo - Halt - S)$ it makes a mistake.

Definition 24.103. Problem of **Halting Problem** (Introduced by Church and Turing in 1936):

Input: Binary representation of an algorithm A

Output: YES if A halts on every input X and NO otherwise

Example 24.104. Reducing one undecidable problem to another: Reducing Halting to Halting All.

We claim that Halting All is undecidable:

Proof:

The instance converter (does not have to be in poly-time): the input of the Halting problem has input A, D. The input of Halting All would be an algorithm

```
B(X):
    A(D)
    return -1
```

We claim that Halting All will halt on all input X if and only if A halts on D. This is because B ignores its input and only calls $A(D)$.

In undecidability reductions, the instance converter do not have to take polynomial time.

Definition 24.105. Post Correspondence Problem:

Input: A list of words $L_1 = \{w_{11}, w_{12}, ..., w_{1n}\}$ of from an alphabet with at least 2 characters. A list of words $L_2 = \{w_{21}, w_{22}, ..., w_{2n}\}$ of from an alphabet with at least 2 characters.

Output: is there a finite sequence $(i_1, i_2, ..., i_k)$ for some k such that $w_{1,i_1} w_{1i_2} ... w_{1i_n} = w_{2i_1} w_{2i_2} ... w_{2i_n}$.

Note that k is not limited.

Post Correspondence Problem is undecidable because Halting problem reduces to it.