CS341: Algorithms

Felix Zhou

Fall 2019, University of Waterloo
from Professor Eric Blais' Lectures

Contents

1 A Whirlwind Tour: The Convex Hull Problem

1.1 Naive Solution

$O(n^3)$ time

1.2 Jarvis Walk

$O(n^2)$ time

1.3 Divide & Conquer

$O(n \log n)$ time

1.4 Chan's Algorithm

$O(n \log h)$ time where h is the size of the returned convex hull.

2 Introduction

2.1 Basic Definitions

Definition 2.1.1 (Algorithm)
An algorithm is a description of a process that is

- unambiguous (always know what step to do next)
- effective (each step is a basic operation)
- finite

Definition 2.1.2 (Solution)
An algorithm solves a problem if for every instance of the problem, when that instance is the input to the algorithm, it produces a valid solution as output.

2.2 Models of Algorithm Analysis

Definition 2.2.1 (Line-Cost Model)
Each line run takes 1 time step.

Definition 2.2.2 (Bit-Cost Model)
Each operation on a single bit takes 1 time step.

2.2.1 Case Study

Example 2.2.1 (Tower)

```
tower(n):
    k = 1k
    for i=1,..., n:
        k = 2^k
    return
```

Analysis with the line-cost model shows us that the algorithm runs in linear time which is simply ridiculous.

Example 2.2.2

```
ALG(n):
    ...
    ...
    c = a*b
```

Observe that the naive multiplication algorithm runs in $O(\log a \cdot \log b)$ time.

To make matters worse, there is a fast multiplication algorithm which runs in $O(\log a(\log b)^c)$ where c is a small constant less than 1.

2.2.2 CS341 Model

Definition 2.2.3 (Word RAM Model)
For an algorithm running on inputs of size n,

1. the memory will consist of words of length $w(n)(= \log n)$.
2. all operations on individual words take 1 time step.

Note that this is NOT the only model that is possible. Existing variations including those whose basic instructions have a weight.

However, this is almost always the basic model which is used to analyze algorithms. We come to some result and then "deal" with the necessary complications.

2.3 Time Complexity

Let us write

$$T_A(I) = \text{number of time steps } A \text{ takes on input } I$$

Definition 2.3.1 (Worse-Case Time Complexity)
$T_A(n) = \max\{T_A(I) : I \text{ is an input of size } n\}$

Definition 2.3.2
Two functions $f, g : \mathbb{N} \to \mathbb{R}_{\geq 1}$ satisfy $f = O(f)$ if

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 \implies f(n) \leq c\dot{g}(n)$$

Proposition 2.3.1
$f(n) = 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi = O(n^7)$

Proof
$c = 300, n_0 = 1000$

Definition 2.3.3
$f = \Omega(g)$ if there is some $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that for all $n \geq n_0$

$$f(n) \geq cg(n)$$

Proposition 2.3.2
$n^7 + o(n^7) = \Omega(n^7)$

Definition 2.3.4
$f = \Theta(g)$ if $f = O(g) \wedge f = \Omega(g)$

Note that to be 100 percent correct, we should write $f \in O(g), f \in \Omega(g), f \in \Theta(g)$.

Definition 2.3.5
$f = o(g)$ if for every $c \in \mathbb{R}^+$ there is some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$

$$f(n) < c \cdot g(n)$$

Definition 2.3.6
$f = \omega(g)$ if for every $c \in \mathbb{R}^+$ there is some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$

$$f(n) > c \cdot g(n)$$

3 Reduction

3.1 2 SUM

Definition 3.1.1 (2 SUM Problem)
Given an array A of n integers and $m \in \mathbb{Z}$, return

- $(i, j) \in [n]^2$ such that
$$A[i] + A[j] = m$$
if such a pair exists
- $\perp$ if no such pair exists

3.1.1 Naive Algorithm

```
SIMPLE2SUM(A, m):
    for i in [n]:
        for j in i, ..., n:
            if A[i] + A[j] = m:
                return (i, j)

    return None
```

$\Theta(n^2)$

3.1.2 Binary Sort

```
2SUM(A, m):
    B = sorted(A)
    for i in [n]:
        j = BSearch(B, m-B[i])
        if j is not None:
            return (Scan(A, B[i]), Scan(A, B[j]))
```

$O(n \log n) + O(n \log n) + 2O(n)$

3.2 3SUM

Definition 3.2.1 (3 SUM Problem)
Given an array A of n integers and $m \in \mathbb{Z}$, return

- $(i, j, k) \in [n]^3$ such that

$$A[i] + A[j] + A[k] = m$$

  if such a tuple exists

- $\perp$ if no such tuple exists

3.2.1 Naive Algorithm

```
SIMPLE2SUM(A, m):
    for i in [n]:
        for j in i, ..., n:
            for k in j, ..., n:
                if A[i] + A[j] + A[k] = m:
                return (i, j, k)

    return None
```

$\Theta(n^3)$

3.2.2 2SUM Reduction

Definition 3.2.2 (Reduction)
Use known algorithms to solve new problems.

```
3SUM(A, m):
    for i in [n]:
        (j, k) = 2SUM(A, m-A[i])
        if (j, k) is not None:
            return (i, j, k)
    return None
```

$n^2 \log n$

3.2.3 Follow-up

Can we solve 3SUM in time $O(n^2)$?

Can we solve 3SUM in time $o(n^2)$?

$$O\left(\frac{n^2}{\log^2 n}\right) \ldots O(n^{1.99999})$$

4 Recurrences

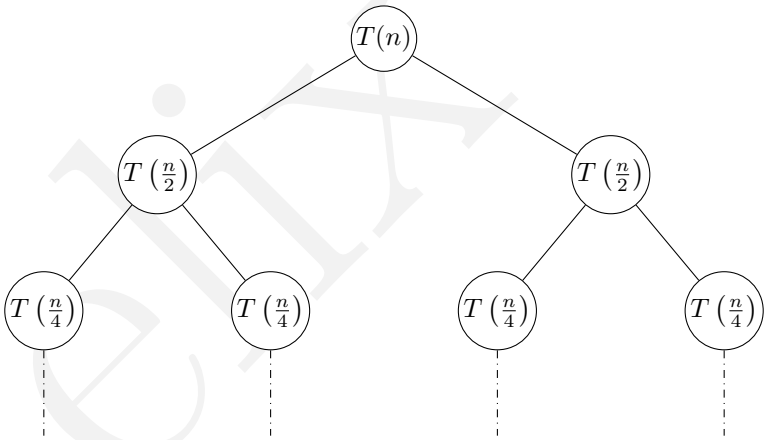
4.1 Merge Sort

```
MERGESORT(A):
    if n == 1:
        return A

    A_1 = MERGESORT(A[1, ..., n/2])
    A_2 = MERGESORT(A[n/2+1, ..., n])

    return MERGE(A_1, A_2)
```

4.1.1 Time Complexity

$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

4.1.2 Recurrence Tree



There are $\log n$ levels and each level requires a linear time operation.

This shows that the overall run time is

$$O(n \log n)$$

4.1.3 Magic Merge

Suppose we can merge in constant time, then the time complexity is a geometric sequence

$$1 + 2 + \cdots + 2^{\log n}$$

which gives us $O(n)$ time.

4.2 Tri-Merge Sort

```
MERGESORT(A):
    if n == 1:
        return A

    A_1 = MERGESORT(A[1, ..., n/3])
    A_2 = MERGESORT(A[n/3+1, ..., 2n/3])
    A_3 = MERGESORT(A[2n/3+1, ..., n])

    return MERGE(A_1, A_2, A_3)
```

4.2.1 Time Complexity

$T(n) = 3T\left(\frac{n}{3}\right) + O(n)$

The recurrence stays the same.

4.3 Master Theorem

Theorem 4.3.1 (Master Theorem)
Let $a, b \geq 1$ be constant and a function $f : \mathbb{N} \to \mathbb{R}_+$.
Suppose we have the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

(1) $f(n) \in O(n^{\log_b a - \epsilon}) \implies T(n) \in O(n^{\log_b a})$

(2) $f(n) \in \Theta(n^{\log_b a}) \implies T(n) \in \Theta(n^{\log_b a} \log_b n)$

(3) $f(n) \in \Omega(n^{\log_b a + \epsilon})$ and in addition if $af\left(\frac{n}{b}\right) \leq \alpha f(n)$ for some $0 \leq \alpha < 1$, then $T(n) \in \Theta(f(n))$

Lemma 4.3.2
$a^{\log_b n} = n^{\log_b a}$

Proof

$$\begin{aligned}
\frac{\ln n}{\ln b}(\ln a) &= \frac{\ln a}{\ln b}(\ln n) \\
(\log_b n)(\ln a) &= (\log_b a)(\ln n) \\
\ln a^{\log_b n} &= \ln n^{\log_b a} \\
a^{\log_b n} &= n^{\log_b a}
\end{aligned}$$

Proof (Master Theorem)

$$\begin{aligned}
T(n) &= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \dots \\
&= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) + a^{\log_b n} f(1) \\
&= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) + O(n^{\log_b a}) && \text{by the lemma}
\end{aligned}$$

If $f(n) \in O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$ then the summation is a geometric series bounded above by its infinite sum so $T(n) \in n^{\log_b a}$.

Let $c < \log_b a$

$$\begin{aligned}
T(n) &= n^c \sum_{i=0}^{\log_b n - 1} \left(\frac{a^i}{b^{ic}}\right) + O(n^{\log_b a}) \\
&= O(n^c) + O(n^{\log_b a}) \\
&= O(n^{\log_b a})
\end{aligned}$$

If $f(n) \in \Theta(n^{\log_b a})$ then each entry in the sum is constant and there are $\log_b n$ of them so $T(n) \in n^{\log_b a} \log_b n$.

Let $c = \log_b a$

$$\begin{aligned} T(n) &= n^c \sum_{i=0}^{\log_b n - 1} \left(\frac{a^i}{b^{ic}}\right) + O(n^{\log_b a}) \\ &= n^c(\log_b n - 1) + O(n^c) \\ &= \Theta(n^{\log_b a} \log_b n) \end{aligned}$$

If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, the lower bound is trivial.

Suppose now that there is some constant $1 > \alpha \in \mathbb{R}_+$ such that

$$af\left(\frac{n}{b}\right) \leq \alpha f(n)$$

note that then

$$a^i f\left(\frac{n}{b^i}\right) \leq \alpha^i f(n)$$

so

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) + O(n^{\log_b a}) \\ &\leq \sum_{i=0}^{\infty} \alpha^i f(n) + O(n^{\log_b a}) \\ &\in O(f(n)) \end{aligned}$$

Proposition 4.3.3
For the same setup as the Master Theorem:

(1) $f(n) \in \Theta(n^{\log_b a - \epsilon}) \implies T(n) \in \Theta(n^{\log_b a})$

(2) $f(n) \in \Theta(n^{\log_b a}) \implies T(n) \in \Theta(n^{\log_b a} \log_b n)$

(3) $f(n) \in \Theta(n^{\log_b a + \epsilon}) \implies T(n) \in \Theta(n^{\log_b a + \epsilon})$

Proof
(1), (2) is the exact same proof.

The proof of (3) simply requires the leverage of finite geometric series.

4.4 Proving Recurrences by Induction

Let $T(n)$ be the running time of some algorithm.

We can prove an (ideally tight) upper bound of $T(n)$ by guessing an upper bound and proving

its correct through induction.

Example 4.4.1
$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n.$

With $T(1) = 1$ and the base cases

$$T(2) = 2T(1) + 2 \leq c2\log 2 \iff c \geq 2$$
$$T(3) = 2T(1) + 3 \leq c3\log 3 \iff c \geq 2$$

Now for the induction step

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2c\left\lfloor \frac{n}{2} \right\rfloor \log\left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq cn\log\frac{n}{2} + n \\ &= cn\log n - cn + n \\ &\leq cn\log n \end{aligned}$$

4.4.1 Warnings

We can guess the wrong upper bound so that the induction step almost works, but it must be strictly correct.

It is also possible that we guess an upper bound which is simply not tight.

4.5 Changing Variables

Consider

$$T(n) = 2T(\sqrt{n}) + \log n$$

and note that

$$T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$$

We can take

$$S(m) := T(2^m)$$

so that

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

But we know that

$$S(m) \in O(m\log m)$$

So

$$T(2^m) \in O(\log m \log\log m) \implies T(n) \in O(\log n \log\log n)$$

5 Divide & Conquer

The essence of the divide and conquer approach lies in 3 steps:

1) divide the problem into smaller subproblems
2) conquer each smaller subproblem seperately
3) combine the results to solve the original problem

5.1 Counting Inversions

Definition 5.1.1 (Inversion)
A pair $i < j$ is an inversion in the seqeunce

$$a_1, \ldots, a_n$$

if $a_i > a_j$.

5.1.1 Problem

Given a sequence $a_1, \ldots, a_n$, compute the number of inversions in the sequence.

Note that the brute force algorithm solves this in $O(n^2)$ time.

5.1.2 Divide & Conquer

Divide the array into two halves.

The total number of inversions is

$$r_L + r_R + r_A$$

the number of inversions in the left, right subarray, and the number of inversions across the division line.

Modify MergeSort so that in the merge step, the number of elements left in the left subarray is the exact number of inversions with one element in the pair being fixed in the right subarray.

```
def sortAndCountInv(A[1,...,n]):
    if n == 1:
        return
```

```
(L, r_L) = sortAndCountIng(A[1,...,n/2])
(R, r_R) = sortAndCountIn(A[n/2+1,...,n])

r, S = 0, []
while L or R:
    if R[0] < L[0]:
        S.append(R[0])
        R = R[1:]
        r += len(L)
    else:
        S.append(L[0])
        L = L[1:]

return S, r_L + r_R + r
```

$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$

5.2 Binary Multiplication

5.2.1 Problem

Given two n-bit integers x, y, compute their product xy.

5.2.2 Grade School Algorithm

The grade school algorithm $\Theta(n^2)$.

5.2.3 Karatsuba's Algorithm

$$\begin{aligned} xy &= \left(2^{\frac{n}{2}} x_L + x_R\right)\left(2^{\frac{n}{2}} y_L + y_R\right) \\ &= 2^n (x_L + y_L) + 2^{\frac{n}{2}} \left((x_L - x_R)(y_L + y_R) - x_L y_L + x_R y_R\right) + x_R y_R \end{aligned}$$

$T(n) = 3T\left(\frac{n}{2}\right) + O(n) \in O(n^{\log_2 3})$

5.3 Fast Matrix Multiplication

5.3.1 Problem

Let $A, B \in \mathbb{R}^{m \times n}$, we wish to compute C such that

$$C_{i,j} := \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

5.3.2 Brute Force

$O(n^3)$ time.

5.3.3 Strassen's Algorithm

Write

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

By block matrix multiplication

$$C_{ij} = A_{i1} B_{1j} + A_{i2} B_{2j}$$

This does 8 calls to matrices of size $\frac{n}{2}$ and $O(n^2)$ work with addition.

Solving

$$T(n) := 8\left(\frac{n}{2}\right) + n^2 \in O(n^3)$$

which is not better.

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

then

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

We have

$$T(n) := 7T\left(\frac{n}{2}\right) + n^2 \in O(n^{\log_2 7}) \approx O(n^{2.81})$$

The best algorithms run in

$$O(n^{2.37})$$

time and it is conjectured that the theoretical lower bound of

$$\Omega(n^2)$$

can be obtained.

5.4 Closest Pair of Points on the Line

5.4.1 Problem

Let $\{(x_k, y_k)\} \subseteq \mathbb{R}^2$ be a finite set of points, we wish to find

$$\delta := \min \|\vec{x}_i - \vec{x}_j\|$$

for all $i \neq j$.

5.4.2 Naive Algorithm

Enumerate all points and keep a running minimum

$$T(n^2) \in \Theta(n^2)$$

5.4.3 Divide & Conquer

1. Initially sort the list with respect to the x-coordinates
2. Recursively call the algorithm on two halves, split by x-coordinate, call them Q, R
3. return smallest distance between points in Q, R, and between points in left and right.

Note for the combine step, we only need to consider points crossing some line L seperating points of Q, R which are less than

$$\delta := \min \delta_Q, \delta_R$$

Let S be the set of points which are at most δ from L.

Proposition 5.4.1
Any point outside of S are distance greater than δ from each other.

Proposition 5.4.2
For $(x^*, y^*) \in S$, there must be at most 8 points $(x', y') \in S$ such that

$$y^* \leq y' \leq y^* + \delta$$

Proof
We can subdivide the points with 8 squares of side length $\frac{\delta}{2}$ and note that each square can only contain one point, else it has distance closer than δ and also resides in either Q, R.

```
delta_Q := closestPair(Q)
delta_R := closestPair(R)
delta := min(delta_Q, delta_R)

S = []
for i=1, ..., n:
if x_n/2 - delta <= x_i x_n/2 + delta:
S.append(x_i, y_i)

S.sort(key = lambda x, y: y)
for 1 <= i < j <= len(S), j-i <= 7:
    delta = min(delta, norm( (x_i, y_i), (x_j, y_j) ))

return delta
```

This solves to

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

The recurrence will be an assignment question

5.4.4 An Optimization

We can presort to get the runtime

$$T(n) = 2T\left(\frac{n}{2}\right) + n \in O(n \log n)$$

6 Greedy Algorithms

6.1 Motivation: Finding Change

1. breaks down problem into individual steps (sequential)
2. each choice for individual steps is made according to a local "decision"

> **Example 6.1.1 (Finding Change)**
> Does a greedy algorithm always return change with minimum number of coins?
>
> In general no! Consider the following:
>
> Coin Value: 1, 100, 101
>
> Change 200

> **Example 6.1.2 (Huffman Encoding)**

6.2 Proving Correctness

6.2.1 Always Ahead

At each point of the algorithm, the "decision" made by the greedy algorithm is "as good as" any the decision made by any other algorithm.

6.2.2 Exchange Method

Given any optimal solution, show that the greedy algorithm achieves it.

6.3 Interval Scheduling

6.3.1 Problem

We are given n intervals

$$(s_1, f_1), \ldots, (s_n, f_n)$$

Find the maximum (= largest) set

$$I \subseteq [n]$$

of non-overlapping intervals.

6.3.2 Greedy Algorithm

1. Pick the shortest interval first
2. earliest finish time
3. earliest start time
4. fewest overlaps with other intervals

Proposition 6.3.1
For every valid solution

$$I^* = \{i_1, \ldots, i_k\}$$

the set

$$I' = \{j_1, i_2, \ldots, i_k\}$$

is also a maximum set of non-overlapping intervals sorted by finish time with distinct finish times.
Where j_1 is the first interval returned by the earliest finish time algorithm.

Proof
I' is a set of non-overlapping intervals because I^* is and for any $m \geq 2$ and

$$f_{j_1} \leq f_{i_1} < s_{i_m}$$

by the greedy algorithm.

Remark that $|I'| = |I^*|$ since

$$j_1 \notin \{i_2, \ldots, i_k\}$$

so I' is a valid solution.

Proposition 6.3.2
If

$$I^* = \{j_1, \ldots, j_{m-1}, i_m, \ldots, i_k\}$$

is a valid solution then so is

$$I' = \{j_1, \ldots, j_m, i_{m+1}, \ldots, i_k\}$$

Proof
We argue by induction.

The base case has been handled in the proposition above.

Now, by the definition of the greedy algorithm, we have

$$f_{j_m} \leq f_{i_m} < s_{i_k}, k \geq m$$

which shows the proof.

Proposition 6.3.3
The greedy algorithm determined by earliest finish time solves the interval scheduling problem.

Proof
The previous proposition shows that any valid solution I^* can be transformed to the greedy solution I' with possible smaller size. so it suffices to show that

$$|I'| = |I^*|$$

But the greedy algorithm only returns if all intervals after the last returned index intersect with at least one of its returned intervals.

By the above, we conclude the proof.

6.4 Minimizing Lateness

Given n tasks with processing times

$$p_1, \ldots, p_n$$

and deadline

$$d_1, \ldots, d_n$$

find an ordering σ that minimizing

$$\max_i L_i^\sigma = \max_i \left\{ \left[\sum_{j:\sigma(j)\leq\sigma(i)} p_j \right] - d_i \right\}$$

6.4.1 Greedy Algorithm

1. earliest deadline
2. longest task first
3. shortest task first
4. shortest slack
5. tasks past deadline last

Consider the earliest deadline algorithm.

Proposition 6.4.1
The greedy algorithm characterized by earliest deadline solves the minimizing lateness problem.

Proof
Consider any arbitrary input and sort the deadlines so $d_1 \leq d_2 \leq \cdots \leq d_n$.

Let σ be the return values of some optimal solution on an arbitrary input.

Furthermore, suppose that σ is not the same permutation as the returned permutation of the greedy algorithm, which means there is some $\sigma(j) < \sigma(i)$ but $d_i \leq d_j$.

Let σ' be the permutation equal to σ but with i, j flipped.

We claim flipping the ordering does not change overall lateness.

Clearly, $L_i^{\sigma'} \leq L_i^{\sigma}$ since the deadlines stay constant and we complete task i earlier in σ'.

Now, to see that $L_j^{\sigma'} \leq L_i^{\sigma}$, we notice that $d_i \leq d_j$ and

$$\sum_{k:\sigma(k)\leq\sigma(i)} p_k = \sum_{l:\sigma'(l)\leq\sigma'(j)} p_l$$

as task i, j finish at the exact same time in σ, σ' respectively.

So the overall lateness did not increase.

We can repeat this procedure at most $O(n^2)$ times (think BubbleSort) to get the permutation returned by the greedy algorithm, demonstrating its correctness.

6.5 Interval Coloring

6.5.1 Problem

Given n intervals $(s_1, f_1), \ldots, (s_n, f_n)$, find a coloring of each interval such that no two overlapping intervals get the same color and we use as few colors as possible.

6.5.2 Interval Scheduling Repeatedly

Take the maximum non-overlapping?

6.5.3 Greedy

Sort by start time.

For every interval i, if there is a color that was used previously and does not overlap i, use it. Otherwise, use a new color.

$O(n \cdot d)$.

where d is the maximum number of colors necessary.

Theorem 6.5.1
Let d be the maximal number of intervals that cover any point.
Then at least d colors are required and the greedy algorithm uses at most d colors.

Proof
All d intervals that cover a point overlap each other and need distinct colors.

Suppose that at most d intervals intersect at any point. Given some index of interval i, then there can be at most $d-1$ intervals indexed by

$$1, \ldots, i-1$$

that cover the point s_i.

Hence there is at least one color available for i.

6.6 Fractional Knapsack

6.6.1 Problem

Given a knapsack capacity W and n items with weights

$$w_1, \ldots, w_n$$

and values

$$v_1, \ldots, v_n$$

find $x_1, \ldots, x_n$ satisfying

$$0 \leq x_i \leq w_i$$

such that

$$\sum x_i \leq W$$

which maximizes

$$\sum \frac{v_i}{w_i} x_i$$

6.6.2 Greedy Algorithm: Highest Value-Weight Ratio First

Theorem 6.6.1
The proposed greedy algorithm is correct.

Proof
Sort the items so that

$$\frac{v_1}{w_1} \geq \cdots \geq \frac{v_n}{w_n}$$

Let $x_1, \ldots, x_n$ be the output of the greedy algorithm.

Let $y_1, \ldots, y_n$ be an optimal solution.

We want to construct a new solution

$$y'_1, \ldots, y'_n$$

that is optimal and closer to $x_1, \ldots, x_n$.

where closer is defined by the

$$m = |i \leq n : x_i \neq y_i|$$

If $m = 0$, $x = y$ so the greedy algorithm is optimal.

Else, we want y' to have less than m values different than x.

Let k be the first index where $x_k \neq y_k$.

Then $x_k > y_k$ by the greedy algorithm.

Since $\sum x_i = \sum y_i$, there must be $l > k$ such that $x_l < y_l$.

Increase y_k and decrease y_l by

$$\min\{x_k - y_k, y_l - x_l\}$$

Then m decreases as we force either $x_k = y'_k$ or $x_l = y'_l$ and since the ratio of k is at least the ratio of l, The net change in overall value is nonnegative.

We can repeat this procedure at most $\frac{m}{2}$ times so that $x = y^{(\alpha)}$ for some $1 \leq \alpha \leq \frac{m}{2}$.

6.7 Offline Cache

6.7.1 Problem

For any n page requests, minimize the number of page faults.

6.7.2 LRU Cache

6.7.3 FIFO

6.7.4 LFU

6.7.5 LIFO

6.7.6 LFD: Least Forward Distance

Which element do we NOT need for as long as possible

7 Dynamic Programming

(1) breakdown problem into subproblem

(2) solve the subproblems

(3) reuse solutions already computed

7.1 Fibonacci Numbers

$$F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

The time complexity for naive recursion is $O(2^n)$.

7.1.1 DP

```
F[0] = 1
F[1] = 1
for k=2, ..., n:
    F[k] = F[k-1] + F[k-2]
return F[n]
```

7.2 Text Segmentation

Given a string s, can we "split" s into substrings such that each substring is is a "valid" word given by a dictionary.

7.2.1 Greedy Algorithm

Does not work.

7.2.2 DP

```
def(T[1,...,n]):
    S[k] = [False]*n
    for k=1, ..., n:
        S[k] = any(S[j] and isWord(T[j+1, ..., k]) for j=k-1,...,1)
```

```
    return S[n]
```

$O(n^2)$ time assuming `isWord` is runs in constant time.

7.3 Longest Increasing Subsequence

```
def LIS(A[1,...,n]):
    for k=1, ..., n:
        S[k] = max(S[j] + 1 for j=k-1,...,1 if A[j] < A[k])

    return S[n]
```

$O(n^2)$ time but can be optimized with binary search for $O(n \log n)$ time.

7.4 Longest Common Subsequence

Given strings x, y find the length of the longest string that is a subsequence of both x, y.

7.4.1 Remark

Let $x[i]$ denote the i-th character of x with $x[0] = \varnothing$ being the empty string by convention.

7.4.2 DP

Let $M[i, j]$ be the length of the longest common subsequence in $x[1, \ldots, i], y[1, \ldots, j]$.

```
M[1, 1] = 1 if x[1] == y[1] else 0
# M[0, j] = M[i, 0] = 0 for every i, j

for i= 1, ..., m:
    for j=1, ..., n:
        M[i, j] = max(M[i-1, j], M[i, j-1])

        if x[i] == y[j]:
            M[i, j] = max(M[i, j], M[i-1, j-1] + 1)

return M[m, n]
```

7.4.3 Producing the Longest Subsequence

Backtrack

(i) if $x[i] \neq y[j]$ then we must have taken the max of the left and above.

(ii) if $x[i] = y[j]$ then we must have added a letter

(iii) otherwise choose above or left arbitrarily

7.5 Edit Distance

Definition 7.5.1 (Edit Distance)
Edit distance between string x, y is the minimal number of operations needed to transform $x \to y$.

(1) add a letter to x

(2) delete any letter from x

(3) substitute one letter for a different one

7.5.1 DP

```
M[1, 1] = 1 if x[1] == y[1] else 0
# M[0, j] = j for every j
# M[i, 0] = i for every i

for i= 1, ..., m:
    for j=1, ..., n:
        M[i, j] = min(M[i-1, j], M[i, j-1]) + 1

        if x[i] == y[j]:
            M[i, j] = min(M[i, j], M[i-1, j-1] + 1)
        else:
            M[i, j] = min(M[i, j], M[i-1, j-1])

return M[m, n]
```

7.6 Weighted Interval Scheduling

Solution to the previous problem.

7.6.1 DP

Let us solve the WIS Problem by solving it up to the first k-th interval.

```
W.sort(key=lambda (c, d): d)

M[1] = W[1]

for i= 2, ..., n:
    M[i] = M[i-1]

    (a, b) = W[i]
    j = bisect_right(W, a, key=lambda (c, d): d)

    M[i] = min(M[i], M[j] + W[i])

return M[n]
```

7.7 Optimal Binary Search Trees

7.7.1 Problem

Given items $1, \ldots, n$ and probabilities $p_1, \ldots, p_n$, construct a binary search tree T which minimizes the search cost

$$\sum_{i=1}^{n} p_i \cdot \text{depth}_T(i)$$

Note that this is different than the Huffman Encoding as we must still ensure that the items are produced in-order if we apply an in-order traversal.

7.7.2 DP

We wish to build trees from the bottom up.

For $1 \leq i \leq j \leq n$, let

$$M[i, j]$$

is the minimum search cost of binary search items $i, i+1, \ldots, j$.

We solve the subproblems by increasing values of $j - i$.

(1) identity root node $k \in \{i, \ldots, j\}$

(2) find search cost of subtrees for items $i, \ldots, k-1$ and $k+1, \ldots, j$

(3) compute search cost of optimal subtrees for the new tree rooted at k.

Note that the third item is simply

$$C = C_L + \sum_{l=i}^{k-1} p_l \cdot 1 + C_R + \sum_{l=k+1}^{j} p_l \cdot 1 + p_k \cdot 1$$

This is due to the fact that we "shifted" the left and subtrees down one level, increasing the depth of all nodes by 1, and we further account for the root node k.

```
for i=1, ..., n:
    for j=i, ..., n:
        if i==j:
            M[i, i] = p_i
            M[i, i-1] = 0
        else:
            M[i, j] = float('inf')

for d = 1, ..., n-1:
    for i = 1, ..., n-d:
        j = i+d

        p = sum(p_i, ..., p_j)

        for k=i+1, ..., j:
            M[i, j] = min(M[i, j], M[i, k-1] + M[k+1] + p)

return M[1, n]
```

$O(n^3)$ time but can be optimized to $O(n^2)$ time.

7.8 Knapsack

7.8.1 Problem

An instance of the 0-1 knapsack problem is a set of n items that have positive integer weights

$$w_1, \ldots, w_n$$

and values

$$v_1, \ldots, v_n$$

as well as a maximum weight capacity W of the knapsack.

A valid solution is a subset

$$S \subseteq 1, \ldots, n$$

such that

$$\sum_{i \in S} w_i \leq W$$

maximizing

$$\sum_{i \in S} v_i$$

Note that the greedy solution to the previous knapsack variant allowing for fractional items does not work anymore.

7.8.2 DP

Let

$$M(k, w)$$

denote the total value of a valid solution to the problem up to the first k items and capacity w.

We either take or choose not to take the k-th item, giving the recursion

$$M(k, w) = \begin{cases} M(k-1, w), & w_k > w \\ \max\{M(k-1, w), v_k + (k-1, w-w_k)\}, & w_k \leq w \end{cases}$$

The proof of correctness is by induction on k, w and follows intuition.

$O(nW)$ run-time.

To find the optimal set of items, we can backtrack to find the items put in the knapsack based on $M[k, w]$.

8 Graph Algorithms

8.1 Definition

Definition 8.1.1 (Graph)
A graph $G = (V, E)$ is a pair consisting of a vertex set V and a set

$$E \subseteq V \times V$$

of edges connecting pairs of vertices.

By convention, we will let

$$n = |V|, m = |E|$$

Definition 8.1.2 (Unordered Graph)
A graph with undirected edges.

Definition 8.1.3

- adjacent
- incidence
- in/out degree
- cycle
- connectedness
- tree
- tree
- connected component

Definition 8.1.4 (Path)
A graph theoretic walk.

Definition 8.1.5 (Simple Path)
A graph theoretic path.

8.2 Graph Representations

We either represent graphs with an

Definition 8.2.1 (Adjacency Matrix)
$M \in \mathbb{R}^{n \times n}$ such that

$$M_{i,j} = \begin{cases} 1, & ij \in E \\ 0, & ij \notin E \end{cases}$$

or

Definition 8.2.2 (Adjacency List)
n linked lists, one for each node in $v \in V(G)$, which stores all the neighbours of v.

8.2.1 Tradeoffs

Notice that the adjacency matrix representation takes $\Theta(n^2)$ space but allows us to check adjacency in constant time. The tradeoff is that identifying all neighbours of a vertex v takes $O(n)$ time.

On the other hand, the adjacency list approach takes $\Theta(n+m)$ space but requires $O(n)$ time to check for adjacency of two vertices. However, identifying all th eneighbours of v takes $\Theta(\deg(v))$ time.

8.3 Graph Exploration Problem

8.3.1 Problem

Given a graph $G = (V, E)$ and a vertex $s \in V$, a valid solution to this instance is a list of all the vertices in the connected component of G containing s.

8.3.2 Breadth-First Search

```
for v in V-s:
    status[v] = undiscovered
status[s] = discovered

Q = set([s])
L = set()

while Q:
    v = Q.pop()

    for w in adj(G, v):
        if status[w] == undiscovered:
            status [w] = discovered
            Q.push(w)

    L.append(v)
    status[v] = explored

return L
```

8.3.3 Running Time Analysis

Theorem 8.3.1
The time complexity of BFS is $O(n+m)$.

Proof
By only adding vertices in the queue when they are undiscovered and by changing their status to discovered when we do so, we guarantee that each vertex is added to the queue at most once.

This means the total number of queue operations is $O(n)$ and since those operations take constant time, the time complexity of those operations is $O(n)$.

Furthermore, since the adjacency list of each vertex is visited at most once ad a constant amount of time is spent for each elemet of those lists, the total amount of time spent in the inner for loop is $O(m)$.

The initialization has linear time complexity.

The total time complexity of the BFS is

$$O(n) + O(n) + O(m) = O(n + m)$$

8.4 Single-Source Shortest Path

8.4.1 Problem

We wish to find the shortest path between a special source vertex s and all the other vertices in the graph.

8.4.2 DFS

```
for v in V-s:
    status[v] = undiscovered
    dist[v] = 'inf'
status[s] = discovered
dist[s] = 0

Q = set([s])

while Q:
    v = Q.pop()

    for w in adj(G, v):
        if status[w] == undiscovered:
            status [w] = discovered
            dist[w] = dist[v] + 1
            Q.push(w)

    status[v] = explored

return dist
```

8.4.3 Correctness

Theorem 8.4.1
Correctness of BFS.

Proof
We argue by induction on the following invariants

(i) All nodes at distance at most d from s have their distance correctly set.

(ii) all other nodes have distance ∞

(iii) the set of nodes that are in the queue (set of nodes at state discovered) is the vertices at distance exactly d from s.

Corollary 8.4.1.1
BFS solves the Graph Exploration Problem.

8.5 Testing Bipartiteness

8.5.1 Problem

Given a connected graph G, determine if it is bipartite or not.

8.5.2 BFS

```
for v in V-s:
    status[v] = undiscovered
status[s] = discovered
partition[s] = 0

Q = set([s])

while Q:
    v = Q.pop()

    for w in adj(G, v):
        if status[w] == undiscovered:
            status [w] = discovered
            partition[w] = 1 - partition[v]
            Q.push(w)
        elif partition[v] == partition[w]:
            return False

    status[v] = explored

return True
```

8.6 Spanning Tree

Definition 8.6.1 (Spanning Tree)
A subgraph of G which has $n-1$ edges and contains all vertices of G.

8.6.1 Problem

Given a graph G, output a spanning tree.

8.6.2 BFS

We can represent a rooted tree with an arry of n nodes, the entry for node v is its parent in the tree with the entry for the root being $\varnothing$.

```
for v in V-s:
    status[v] = undiscovered
status[s] = discovered
parent[s] = None

Q = [s]

while Q:
    v = Q.pop()

    for w in adj(G, v):
        if status[w] == undiscovered:
            status [w] = discovered
            parent[w] = v
            Q.push(w)

    status[v] = explored

return parent
```

8.7 Depth-First Search

Essentially a BFS with a stack instead of a queue.

8.7.1 The Algorithm

```
def DFS(G, s):
    def explore(G, v, parent):
        visited[v] = True

        previsit(G, v, parent) # some function to call to solve another problem

        for w in adj(G, v):
            if not visited[w]:
                explore(G, w)

        postvisit(G, v, parent) # same here

    for v in V(G):
        visited[v] = False

    explore(G, s, None)
```

8.7.2 Iterative Version

```
def DFS(G, s):
    visited = [False for v in V(G)]

    previsit(G, s, None)
    visited[s] = True

    stack = [s]
    while stack:
        v = stack.pop()

        for w in adj(G, v):
            if not visited[w]:
                previsit(G, w, v) # same as before

                visited[w] = True
                stack.push(w, v)

                postvisit(G, w, v) # same as before

    postvisit(G, s, None)
```

8.7.3 Remarks

Lemma 8.7.1
A non-DFS tree edge for a connected graph always connects a vertex to one of its ancestors in the DFS tree.

Proof
Let uv be a non-tree edge.

Let u be discovered before v in the DFS.

If v occurs in the subtree below u, then it is a descendant of u.

Elsewise, the DFS would have discovered v by the uv edge before starting a new subtree, which contradicts the definition of the algorithm.

8.8 Cut Vertex

Definition 8.8.1 (Cut Vertex)
A cut vertex is a separator of a graph.

Definition 8.8.2 (Block)
A connected graph with no cut vertex.

Lemma 8.8.1
The root of the DFS tree has at least 2 children if and only if it is a cut vertex.

Lemma 8.8.2
The non-root vertex v in a DFS tree is a cut-vertex if and only if it contains a sub-tree that itself has no edges to an ancestor.

8.8.1 DFS

```
def DFS(G, s):
    t = 1

    def explore(G, v, parent):
        visited[v] = True
```

```
        pre[v] = t
        low[v] = pre[v]
        t += 1

        for w in adj(G, v):
            if not visited[w]:
                explore(G, w, v)
                low[v] = min(low[v], low[w])

                if parent is not None and low[w] >= pre[v]:
                    return "found cut vertex"
            else if w != parent:
                low[v] = min(low[v], pre[w])

        if p is None and |adj[v]| >= 2:
            return "found cut vertex"

        post[v] = t
        t += 1

    for v in V(G):
        visited[v] = False

    explore(G, s, None)
```

8.9 Dicycle Detection

Definition 8.9.1 (Dicycle)

Consider the DFS tree of a directed graph $D = (N, A)$. There are 3 types of non-tree edges

(i) backward edge to ancestor

(ii) forward edge from ancestor to to descendant

(iii) cross edge from later subtree to a previous subtree rooted at some vertex.

Lemma 8.9.1
A directed graph has a cycle if and only if it has a backwards arc.

Proof
The backward direction is trivial so we show the forwards direction.

Enumerate our cycle

$$v_0, \ldots, v_k, v_0$$

where v_0 is the first to be "discovered" by our DFS algorithm. Then clearly $v_k v_0$ is a backward arc.

Definition 8.9.2 (Directed-Acyclic Graph)
A directed graph with no cycles.

8.9.1 DFS

```
def DFS(G, s):
    t = 1

    def explore(G, v, parent):
        visited[v] = True

        pre[v] = t
        t += 1

        for w in adj(G, v):
            if not visited[w]:
                explore(G, w)
            elif post[w] is None:
                return "Cycle!"

        post[v] = t
        t += 1

    for v in V(G):
        visited[v] = False

    explore(G, s, None)
```

8.10 Topological Sorting

Definition 8.10.1 (Topological Sort / Linearization)
of a DAG is an ordering

$$v_1, \ldots, v_n$$

of the vertices such that every edge $v_i v_j$ is such that $i \leq j$.

Theorem 8.10.1
For a DAG $D = (N, A)$, after DFS: `post[u] > post[v]` for every $uv \in A$.

Proof
Postvisit will only be called when we have finished exploring all vertices that are reachable from u (including v).

8.10.1 DFS

```
def DFS(G, s):
    result = []
    t = 1

    def explore(G, v, parent):
        visited[v] = True

        pre[v] = t
        t += 1

        for w in adj(G, v):
            if not visited[w]:
                explore(G, w)

        post[v] = t
        t += 1
        result.append(v)

    visited = [False for v in V(G)]
    explore(G, s, None)

    result.reverse()
    return result
```

8.11 Strong Connectivity

Definition 8.11.1
$D = (N, A)$ is strongly connected if for all $s, v \in N$ there is both a sv-dipath and a vs-dipath.

Remark that if s, v is strongly connected, v, w are strongly connected, then s, w are strongly connected. So strong connectivity partitions a directed graph into equivalence classes we call strongly-connected components.

Definition 8.11.2 (Reversal)
The reversal D^R of a directed graph D is $D^R = (N, A^R)$ defined as

$$A^R := \{vu : uv \in A\}$$

We can test for strong connectivity by running an exploration algorithm on D, D^R, the reversal of D.

8.12 Minimum Spanning Tree

Definition 8.12.1 (Minimum Spanning Tree)
A (MST) of G is a spanning tree with minimum total weight (sum of total edges).

8.12.1 Kruskal's Algorithm

```
def kruskal:
    E.sort(lambda e: e.weight)

    for uv in E:
        if component(u) != component(v):
            T.add(u, v)

    return T
```

Sorting takes $n \log n$ time.

We can perform the component equality check in $O(m \log m)$ using Union-Find data structures.

8.12.2 Cut Property Lemma

Definition 8.12.2 (Cut)
A cut is a partition $(S, \bar{S})$ of V into two sets.
An edge crosses the cut $(S, \bar{S})$ if it connects a vertex in S to a vertex in $\bar{S}$.

Lemma 8.12.1 (Cut Property Lemma)
Let X be a subset of the edges of a MST of G where no edge of X crosses the cut $(S, \bar{S})$.
Then the edge e with minimum weight among all the edges that cross the cut $(S, \bar{S})$ satisfies the following statement:
$X \cup \{e\}$ is a subset of a MST of G.

Proof
Let T be a MST of G that includes X. T must have an edge e' that crosses the cut $(S, \bar{S})$.

If $e' = e$, then we are done.

Else, $T' := (T - e') \cup \{e\}$ is a spanning tree of G.

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

so T is still a MST.

8.12.3 Prim's Algorithm

```
def prim:
    S = {s}

    while S != V:
        uv = min(uv : u in S, v not in S)
        S.add(v)
        T.add(u, v)

    return T
```

$O(m \log m)$

8.12.4 Proof of Correcness

We argue by induction on $|X|$ that it is a subset of a MST of G.

If $X = \varnothing$ is a subset of all MST of G.

Now, suppose inductively that X is a subset of a MST of G. We claim that $X \cup \{uv\}$ is also a subset of a MST of G. This follows directly from the cut property lemma.

8.13 Shortest Paths in Nonnegative Edge Weighted Graphs

Notice that a MST does NOT always contain the shortest paths.

8.13.1 Dijkstra's Algorithm, 1959

Let $G = (V, E)$ be a graph and $s \in V$ a source vertex. We will to find the shortest s, v-path in G for every $v \in V$.

```
B = {s}
parent = [None]*len(V)

while B != V:
    xy, d = min(xy, d(s, x) + w(x, y) for xy in E, x in B, u not in B)
    d(x, y) = d
    parent(y) = x
    B.add(y)
```

Proposition 8.13.1
We claim that $d(s, y)$ is the minimum distance from $s \to y$.

Proof
Any sv-path π intersects $\delta(B)$. Let $\pi_1 := \pi[B]$. Furthermore, let $e = uv$ be the first edge leaving B and π_2 be the rest of the path.

$$w(\pi) \underbrace{\geq}_{w(\pi_2) \geq 0} w(\pi_1) + w(u, v) \geq d(s, u) + w(u, v) \geq d$$

We can argue more rigorously with inductive on $|B|$.

8.13.2 Implementational Details for Djikstra's Algorithm

We want to tchoose edge leaving B which minimizes some value. We can make a heap of edges $xy, x \in B, y \notin B$ where $v(x,y) = d(s,x) + w(x,y)$. This heap has size $O(m)$.

A more efficient method is to use a heap of vertices. We keep a "tentative distance" $d(v)$ for all $v \notin B$. $d(v)$ is the minimum weight sv-path with all but last edge in B.

```
d = ['inf']*len(V)
d[s] = 0

parent = [None]*len(V)

B = set()
while len(B) < len(V):
    y = heappop(heap)
    B.add(y) # d[y] is true distance
    for yz in E:
        if d[y] + w(y, z) < d(z):
            d[z] = d[y] + w(y, z)
            # update heap
            parent[z] = y
```

We store at most n values in the heap. Modifying a value in our heap (`decrease-key` or delete and add) takes $O(\log m)$ time. We do this at most m times

We also pop from the heap n times, each pop takes $O(\log n)$ time.

The total run-time is given by

$$O(n \log n) + O(m \log n) = O(m \log n)$$

assuming G is connected.

There is an optimization with the Fibonacci Heap which gives

$$O(n \log n + m)$$

running time.

8.14 Shortest Paths in General Edge Weighted Graphs

Djikstra's Algorithm can be modified to some degree to support this but then we would lose the greedy aspect of it. Let us take a step back.

For now, let us ignore the existence of negative dicyles.

8.14.1 Bellman-Ford

Since the greedy algorithm fails, it is natural to consider dynamic programming as our next option.

We defined

$$\texttt{dist[v]} = \text{length of shortest path from } s \text{ to } v$$

We can obtain simpler subproblems if we do not consider all the sv-paths but only paths that use at most

$$1, 2, \ldots$$

edges.

Specifically, we want to compute

$$d_i[v]$$

which is the lenght of the shortest sv-path using at most i edges.

The base case is

$$d_1[v] = \begin{cases} w(s,v), & (s,v) \in E \\ \infty, & \text{else} \end{cases}$$

The $d_i[v]$ is also easy once we have $d_{i-1}[v]$

$$d_i[v] = \min \begin{cases} d_{i-1}[v] \\ d_{i-1}[u] + w(u,v), & (u,v) \in E \end{cases}$$

8.14.2 The Algorithm

```
for v in V:
    if sv in E:
        d[1][v] = w(s, v)
    else:
        d[1][v] = 'inf'

for i=2,3,...,n-1:
    for v in V:
        d[i][v] = d[i-1][v]
        for u in V:
            if uv in E and d[i-1][u] + w(u, v) < d[i][v]:
                d[i][v] = d[i-1][u] + w(u, v)

return d[n-1]
```

Time complexity $O(n^3)$ when edge queries can be performed in constant time. there is a slight reformulation of Bellman-Ford which runs in $O(nm)$ if we store the graph with the adjacency list model.

8.15 All-Pairs Shortest Path (APSP)

8.15.1 The Problem

Given a weighted graph G, find the shortest path between every pair of vertices $u, v \in V$.

8.15.2 Bellman-Ford Reduction

```
for u in V:
    d[u, *] = bellman_ford(G, u)

return d
```

Time Complexity $O(n^2 m) \subseteq O(n^4)$

8.15.3 Bellman-Ford APSP

```
for u, v in V:
    d[0][u, v] = 'inf'
for u in V:
    d[0][u] = 0

for i=1,2,...,n:
    for u, v in V:
        d[i][u, v] = d[i-1][u, v]
    for u in V:
        for xv in E:
            d[i][u, v] = min(d[i][u, v], d[i-1][u, x] + w(x, v))

return d[n]
```

Time Complexity $O(n^3 + n^2 m)$

8.15.4 Floyd-Warshall Algorithm

Let $d_k[u, v]$ be the shortest uv-path which only uses vertices $v_1, \ldots, v_k$.

```
for u, v in V:
    if uv in E:
        dist[0][u, v] = w(u, v)
    elif u == v:
        dist[0][u, v] = 0
    else:
        dist[0][u, v] = 'inf'

for k=1,2,...,n:
    for u, v in V:
        dist[k][u, v] = min(d[k-1][u, v], d[k-1][u, v[k]] + d[i-1][v[k], v])

return d[n]
```

Time Complexity $O(n^3)$ with best $O(n^2)$ space.

9 Exhaustive Search

If none of our previous techniques works

- Approximation Algorithm
- Heuristic
- Exhaustive Search
- Randomized Algorithms
- Quantum Algorithms

9.1 Subset Sum

9.1.1 The Problem

Given an array A of n positive integers and a target T, is there a subset $S \subseteq [n]$ such that

$$\sum_{i \in S} A[i] = T$$

9.1.2 Backtracking

Let S, R denote the accumulator set and remaining set of choices to include (or not).

Consider the decision tree where at each vertex, the left child is the decision to include the first element of the remaining set, and the right child is the decision to NOT include the first element of the remaining set.

At the root, $S = \varnothing, R = [n]$.

At each configuration, we can check

(i) If $\sum_{i \in S} A[i] = T$, we are done!
(ii) If $\sum_{i \in S} A[i] > T$, this is a dead end.
(iii) If $\sum_{S \cup R} A[i] < T$, this is a dead end.

9.1.3 Generalized Backtracking Template

```
init A

while A:
    C = next(A)
    if C is Solution:
        return True
    if C is not DeadEnd:
        A.add(expand(C))

return False
```

Time Complexity $O(2^n)$

9.2 Traveling Salesman Problem

Minimum cost Hamiltonian Circuit in a DIRECTED graph.

9.2.1 Brute Force

$O(n!) = 2^{O(n \log n)}$

9.2.2 Branch-and-Bound

The same configuration idea applies, except we do not return once we find a valid solution but compare with our currect best. If at any point, our configuration cannot beat the currect best, we stop (bound) the current search and check the next configuration.

```
init set A active configs
best = 'inf'

while A:
    C = next(A)
    if C is Solution and C.value < best:
        best = C.value
    elif C is not DeadEnd and valueLB(C) < best:
        A.add(expand(C))

return best
```

Dead-End Checks

- $G(V, E \setminus X)$ is NOT 2-connected
- I contains a cycle of length less than n
- more than 2 edges is incident with the same vertex
- size of I is more than n

Lower Bounds

- the weight of the current path
- $(n - |I|) \cdot$ minimum weight$(E \setminus X)$

10 Computation Complexity

10.1 Introduction to P

10.1.1 What is Efficiency?

- better then target
- better than brute force
- run time $O(n^2), O(n \log n), O(n), O(n \log n), O(n^3)$
- memory/space $O(n)$
- terminates / correct
- fast parallel
- sequential
- cache-friendly

Definition 10.1.1 (Polynomial-Time Algorithm)
An algorithm is polynomial time if it has time complexity $O(n^k)$ on inputs of size n for some $k \geq 0$.

Remark that we always measure input by the size and NOT value.

10.1.2 Motivations for the Definition

- Robustness
- Strength of conclusion: we can conclude that if a problem cannot be solved in polynomial time, then no algorithm can even come close to being "efficient"
- Applicability: many problems encountered in real life does NOT have a polynomial time solution

10.2 The Class P

Definition 10.2.1 (Decision Problem)
An answer is True of False

Definition 10.2.2 (Optimization Problem)
We wish to find the best feasible solution

Definition 10.2.3 (Search Problem)
We wish to find the solution itself

Definition 10.2.4 (Enueration Problem)
We wish to find all feasible problems

We will work mostly with decision problems. Consider the following reductions from search problems to decison problems

For Integer Multiplication Decision, we can ask whether the i-th bit is 1?

For Interval Scheduling Decision, we can ask is there a set of k non-overlapping intervals in I?

Definition 10.2.5 (P)
P is the collection of decision problems that can be solved by polynomial-time algorithms.

Example 10.2.1
3SUM $\in P$

10.3 Reductions

Definition 10.3.1 (Polynomial-Time Reducible)
The decision problem A is polynomial-time reducible to the decision problem B

$$A \leq_P B$$

if there is a polynomial-time algorithm F that transforms A's input I_A into an input I_B to B that has the same answers.

For the assignment, we will use the following definition

Definition 10.3.2 (Polynomial-Time Reducible)
The decision problem A is polynomial-time reducible to the decision problem B

$$A \leq_P B$$

if a polynomial time algorithm for B leads to a polynomial time algorithm for A.

Example 10.3.1 (LIS $\leq_P$ LCS)
Given a sequence z on n numbers in $[d]$ and $k \geq 0$, does z contain an increasing subsequence of length k?

Given two sequences x, y on n numbers in $[\ell]$ and $k \geq 0$, do x, y have a common subsequence of length k?

We can sort the sequence and ask for the longest subsequence between the original sequence and sorted sequence.

Example 10.3.2 (CLIQUE $\leq_P$ INDEPSET)
Given G, k, does G contain a clique of size k?

Given G, k does G contain an independent set of size $\geq k$?

We can take the "complement" of a graph (edge if and only if no edge in original) and run clique or independent set to get the other.

Proposition 10.3.3
If $A \leq_P B$ then

- $B \in P \implies A \in P$
- $A \notin P \implies B \notin P$

10.3.1 Reducible Problems

Definition 10.3.3 (CLIQUE)
Given G, k, does G have a clique of size at least k?

Definition 10.3.4 (INDEPSET)
Given G, k, does G have an independent set of size at least k?

Definition 10.3.5 (NONEMPTY)
Does G have any edges?

Definition 10.3.6 (VERTEXCOVER)
Does G have a vertex cover?

We know INDEPSET$\leq_P$CLIQUE and CLIQUE$\leq_P$INDEPSET.

Proposition 10.3.4
NONEMPTY$\leq_P$CLIQUE

Proof
F takes input G and transforms it to $(G, 2)$ for CLIQUE

Proposition 10.3.5
CLIQUE$\leq_P$NONEMPTY
This is an open problem!

Proposition 10.3.6
INDEPSET$\leq_P$VERTEXCOVER

Proof
F takes input (G, k) to INDEPSET and generates $(G, n - k)$.

Definition 10.3.7 (SETCOVER)
Given a collection of subsets of $[m]$ and a number k, are there k sets

$$S_1, \ldots, S_k \in \mathcal{S}$$

such that

$$S_1 \cup \cdots \cup S_k = [m]$$

Proposition 10.3.7
VERTEXCOVER$\leq_P$SETCOVER

Proof
F takes input (G, k) and transforms it to $(\mathcal{S}, m, k)$ where $m = |E|, \mathcal{S} = \{S_v : v \in V\}$

$$S_v = \{vw \in E\}$$

10.3.2 Facts

Proposition 10.3.8
If $A \leq_P B$ and $B \leq_P C$ then

$$A \leq_P C$$

Proposition 10.3.9
There are decision problems A, B where

$$A \leq_P B, B \leq_P A$$

Theorem 10.3.10
If HARD is not in P and HARD$\leq_P$B, then $B \notin P$.

Note that if $A \leq_P$HARD, then we cannot conclude anything.

10.4 Polynomial-Time Verifier & NP

Definition 10.4.1 (Verifier)
A verifier for decision problem X is an algorithm A that takes some input x to X and a certificate y such that

(i) when the answer to X on x is yes, there is a certificate y that causes A to accept

(ii) when the answer to X on x is no, then for every certificate y, A rejects

Definition 10.4.2 (Polynomial Time Verifier)
A verifier that runs in polynomial tiem in the length of x (only).
In other words, it takes certificates of length polynomial in x.

Definition 10.4.3 (NP)
The set of all decision problems that have polynomial time verifiers.

Proposition 10.4.1
CLIQUE$\in$NP.

Proof
Take the certificate $S \subseteq V$.

First, we check $|S| \geq k$

We verify by checking for each $u \neq v \in S$ check $uv \in E$. Accept if and only if all checks pass.

Consider the phony "certificate" b

$$b := \begin{cases} 1, & G \text{ has size at least } k \\ 0, & \text{otherwise} \end{cases}$$

Accept $\iff b = 1$.

This is NOT a valid certificate / verifier since when the answer to X is no, then for every certificate y, we reject.

Proposition 10.4.2
SUBSETSUM∈NP

Proof
Certificate $S \subseteq [n]$.

We verify by checking $\sum_{i \in S} A[i] = t$.

10.4.1 P vs. NP

If we can verify a problem in polynomial time, can we also solve it in polynomial time?

10.5 NP-Completeness

10.5.1 Definitions & Basic Results

Definition 10.5.1 (NP-Complete)
The decision problem X is NP-complete if

(i) $X \in NP$

(ii) $\forall A \in NP, A \leq_P X$

Theorem 10.5.1
If X is NP-Complete, then

(i) If $X \in P$ then $P = NP$

(ii) If $X \notin P, P \neq NP$

Theorem 10.5.2
The problem X is NP-complete if

(i) $X \in NP$

(ii) there is an NP-complete A such that

$$A \leq_P X$$

Proof
For every $B \in NP$ then

$$B \leq_P A \leq_P X$$

so $B \leq_P X$.

10.5.2 Examples

Definition 10.5.2 (3-SAT)
Given a boolean formula φ on n variables in CNF (AND of ORs, conjunction of clauses of literals) with m clauses that each contain at most 3 literals, determine whether φ is satisfiable.

Notice the size of φ is $O(m \log n)$ and the size of an assignment certificate is $O(n)$.

Theorem 10.5.3
3SAT $\in$ NP.

Theorem 10.5.4 (Cook-Levin, 1971, 1973)
3SAT $\in$ NP-Complete.

Theorem 10.5.5
INDEPSET is NP-Complete.

Proof
We saw INDEPSET is in NP.

We show there is a reduction from 3SAT to INDEPSET.

Let F take formula φ with m clauses and output (G, m) defined below.

$G = (V, E)$ has 1 vertex for each literal in each clause and edges uv for each

(i) u, v are literals in the same clause

(ii) u, v are negations of each other (conflicting/contradicting literals)

Theorem 10.5.6
CLIQUE is NP-Complete.

Proof
CLIQUE is in NP.

Moreover

$$\text{INDEPSET} \leq_P \text{CLIQUE}$$

and INDEPSET is NP-complete.

10.5.3 Corollaries

(1) VERTEXCOVER

(2) SETCOVER

10.5.4 The Hamiltonian Path Problem is NP-Complete

We will first show that DirHamPath is NP-Complete.

Theorem 10.5.7
DirHamPath is NP-Complete.

Definition 10.5.3 (Gadget)
Let φ be the formulat that is the input to 3-SAT.
A graph which corresponds to the idea of assigning True / False values to each of the variables

$$x_1, \ldots, x_n$$

in the original formulat φ.
Let P_i be a path with doubled arcs such that traversing right means an assignment of True to x_i and left is False.
Let s, t be source and sink vertices and connect ends of the path P_i to the ends of the path P_{i+1}.
Each path has $3(m+1)$ nodes. We need 2 per clause but also an intermidiary node between those taken up by clauses.
Add a vertex $c_j, 1 \leq j \leq m$ for each clause C_j. c_j has an arc FROM the $3j$-th node on P_i to the $3j+1$-st node on P_i if x_i appears in C_j. Symmetrically, c_j has an arc TO the $3j$-th node on P_i from the $3j+1$-st node on P_i if $\neg x_i$ appears in C_j.

Proof (Sketch)
The problem is clearly in NP since a verifier which demands the path as a certificate exists.

We now show that

$$3\text{SAT} \leq_P \text{DIRHAMPATH}$$

Let G be the gadget corresponding to φ, an arbitrary input for 3-SAT. Clearly, every valid assignment corresponds to a Hamiltonian Path. We want to show the converse.

The argument goes as: If there is no valid assignment, any path starting at s (it must since s has no in-arcs) will be "stuck" at a variable.

Theorem 10.5.8
HAMPATH is NP-Complete.

Proof
The same argument for the directed version shows that it is in NP.

Let us now show that

$$\text{DIRHAMPATH} \leq_P \text{HAMPATH}$$

Let $D = (N, A)$ be a directed graph. Let $G = (V, E)$ be an undirected graph obtained

from D as follows

$$V := \{v_i, v, v_o : v \in N\}$$
$$E := \{v_i v, v v_o : v \in N\} \cup \{u_o v_i : uv \in A\}$$

Clearly, any Hamiltonian Path in D is a Hamiltonian Path in G. It is an exercise that an Hamiltonian Path in G is a Hamiltonian Path in D.

10.5.5 NP-Completeness of the Hamiltonian Cycle Problem

Remark that there are graphs with a Hamiltonian Path but NO Hamiltonian Cycles!

Lemma 10.5.9
HamCycle is NP-Complete.

Proof
Clearly, HamCycle is in NP.

We show that HamPath $\leq_P$ HamCycle.

Let G be the input to the Hamiltonian Path Problem. Let G' be the graph obtained from G by adding a new vertex s and edges to every existing vertex of G.

If G has Hamiltonian Path P then sPs is certainly a Hamiltonian Cycle in G'. Conversely, if G' has Hamiltonian Cycle C, then $C - s$ is certainly a Hamiltonian Path in G.

10.5.6 NP-Completeness of the Subset Sum Problem

We can also use a reduction from 3Sat to show that SubsetSum is NP-hard. The key insight fo rthis reudction is that we will want to use REALLY big numbers in the reduction.

Theorem 10.5.10
SubsetSum is NP-complete.

Proof
We already saw in the last lecture that it is in NP.

We show

$$\text{3-Sat} \leq_P \text{SubsetSum}$$

Indeed, given an instance φ of the 3-Sat problem on n variables with m clauses, we

want to construct a set of numbers that we will turn into an instance of the SubsetSum problem.

Let the numbers be of form

$$\chi_{x_i} C_1 C_2 C_3 \dots$$

where the i-th bit is 1 if the first segment correponds to x_i. Moreoever let the bit corresponding to C_j be 1 if the assignment of x_i satisfies C_j. We do the exact same for numbers of the form

$$\chi_{\neg x_i} C_1 C_2 C_3 \dots$$

except C_j is 1 if the assignment of $\neg x_i$ satisfies C_j.

Let S be a subset of the numbers. We want the sum of the characteristic bits to sum to 1 for each bit. This corresponds to only one choice of value for each variable. Furthermore, we want the sum of the clause bits to sum to between 1 and 3 for each bit. This means each clause has at least one variable satified.

But is is difficult to express a range for the clause bits. Instead we add 2 numbers per clause. They are 0 on the characteristic bits and 1 and 2 on the bit for each C_j (0 elsewise). This allows us to shoot for the target sum of

$$\underbrace{11\dots1}_{\text{characteristic bits}} \quad \overbrace{44\dots4}^{\text{clause bits}}$$

Clearly, φ is satisfiable if and only if a subset S of the numbers described above sum to the target above.

10.5.7 Closing Remarks on NP-Completeness

There are many NP-Complete problems every

(I) graph edge-colourability

(II) super mario

(III) integer programming

(IV) number theory

An open question is whether NP-complete is the same as NP-hard

Definition 10.5.4 (NP-Hard)
The decision problem X is NP-hard if every problem $A \in NP$ satisfies

$$A \leq_P X$$

There are other reductions than the many-one reduction we have been primarily been working with

Definition 10.5.5 (Cook Reduction)
A cook reduction from problems A to B is a polynomial time algorithm F that, using a polynomial-time algorithm for B as a black-box, solves A in polynomial time.

There are problems that cannot be verified efficiently

Definition 10.5.6 (CoClique)
For a graph G and $k \in \mathbb{Z}_+$, is the largest clique of G of size less than k?

It is easy to check the answer is no but very difficult to check the answer is yes.

There are even problems which are harder than the ones in NP! Undecidable problems are the ones that cannot be solved by ANY algorithm. More details to come during the last two lectures of this class.

11 Approximation Algorithms

11.1 Metric TSP

Given a complete weighted graph G with non-negative edge weights that satisfy

$$w(u, v) \leq w(u, x) - w(x, v)$$

find the length ℓ_{TSP} of the shortest TSP TSP tour through G.

Theorem 11.1.1
MetricTSP is NP-Complete.

Can we find a polytime algorithm that identifies a TSP tour of length $\leq 2\ell_{\text{TSP}}$.

If we remove the condition that we need a tour, one idea is to find a MST of G and simply traverse the tree (use edge at most twice).

Proposition 11.1.2
tour length is $2 \cdot w(T)$.

Proposition 11.1.3
$w(T) \leq \ell_{\text{TSP}}$.

Proof
Any TSP tour contains a spanning tree. Hence by the previous proposition

$$2 \cdot w(T) \leq 2\ell_{\text{TSP}}$$

Proposition 11.1.4
The length of the shortcut tour is at most the length of the tour.

11.1.1 The Algorithm

```
def MetricTSPALG(G):
    T = MST(G)
    tour = traversal of T
    tour_prim = shortcut version of tour

    return tour_prime
```

11.2 Vertex Cover

Design a poly-time algorithm that returns a vertex cover of size

$$\leq 2 \cdot |C_{\text{OPT}}|$$

11.2.1 The Algorithm

```
def VertexCover(G):
    S = set()
    for each uv in E:
        if u not in S and v not in S:
            S.add(u)
            S.add(v)

    return S
```

Proposition 11.2.1
$|S| \leq 2 \cdot |C_{\text{OPT}}|$

Proof
Naive matching algorithm.

11.3 TSP

Given a complete weighted graph G with positive edge weights $\bar{w}$, find a TSP tour of G of length at most $c\ell_{\text{TSP}}$.

Theorem 11.3.1
If $P \neq NP$ then there is no poly-time algorithm that returns a k-approximation to ℓ_{TSP} for any constant $k \geq 1$.

Proof
We argue by contradiction.

Assume A is a poly-time k-approximation algorithm to TSP. Let G be the input to HamCycle

Give the edges weight 1 and make it complete by adding edges of weight kn. If G has a Hamiltonian cycle, then G' has a tour of length at most n. A will return a tour of length at most kn.

Elsewise, if G has NO Hamiltonian Cycles, then G' has a TSP tour of length greater than kn.

12 (Very) Difficult Computational Problems

There are problems even harder than NP-complete problems.

12.1 Impossible Problems

Definition 12.1.1 (Halting Problem)
Given the binary encoding of an algorithm, determine if it terminates.

Theorem 12.1.1 (Turing)
The Halting problem is undecidable.

Proof
Diagonalization argument.

Notice that if the Halting Problem is decidable, then many conjectures such as the Collatz Conjecture and Goldbach Conjecture are trivial corollaries.

Theorem 12.1.2
If X is a decision problem for which the reduction

$$\textsc{Halting} \leq X$$

holds, then there is no algorithm that solves X.

12.1.1 Very Difficult Problems

Even when we restrict to decidable problems, there are still problems harder than the NP-complete ones.

Definition 12.1.2 (Totally Quantified Boolean Formula)
TQBF asks that given a totally quantified Boolean formula

$$\exists x_1 \forall x_2 \exists x_3 \ldots \phi(x_1, \ldots, x_n)$$

determine if the problem is true or not.

"Is there a first move M_1 such that Alice can make such that no matter what move M_2 Bob

makes, there is a response M_3 such that no matter ..."

Notice that unlike SAT, TQBF is hard to even verify!

Definition 12.1.3 (PSPACE)
The collection of problems which can be solved with polynomial space.

Whether or not TQBF can be solved in polynomial time is equivalent to the problem of determining whether

$$P = PSPACE$$

There are also problems which are known to be unsolvable in polynomial time.

Definition 12.1.4 (HaltIn*k*Steps Problem)
Given an algorithm A and an input x to A, does A halt in at most k steps?

This can be solved in roughly $O(k)$ steps but the input has size $O(\log k)$.

Finally, consider the problem of determining whether two regular expression are equivalent. This problem cannot be solved in

$$O(2^{2^{\cdots^{2^n}}})$$

for any constant tower.

12.2 Rather Difficult Problems

Recall the Clique problem. The only known lowerbound is

$$\Omega(n+m)$$

which is the time to read input.

The 3Sum Conjecture says that for every $\epsilon > 0$, any algorithm which solves 3Sum has time complexity

$$\Omega(n^{2-\epsilon})$$

Definition 12.2.1 (Colinear Problem)
Given some points in Euclidean space, do at least 3 of them lie on a line?

There are

$$O(n^3), O(n^2 \log n), O(n^2)$$

algorithms? But we can never do better than $O(n^2)$ by a reduction to the 3Sum Conjecture.

Theorem 12.2.1
If the 3Sum Conjecture is true, there is no algorithm that solves Colinear in $O(n^c)$ time for any constant

$$c < 2$$

Proof
Reduction as indicated above with points

$$(a, a^3)$$

for each a in the input of 3Sum.