CS 341
Lecture Notes
Winter 2025

Collin Roberts

March 28, 2025

Contents

1 Lecture 01 - Introduction, review of asymptotics **8**
- 1.1 Course Intro . . . 8
- 1.2 Slide 09 . . . 8
- 1.3 Slide 10 . . . 8
- 1.4 Slide 13 Exercise . . . 9
- 1.5 Slide 14 Exercise . . . 9
- 1.6 Slide 16 . . . 9
- 1.7 Slide 17 . . . 10
- 1.8 Slide 18 . . . 10
- 1.9 Slide 19 . . . 10
- 1.10 Slide 21 . . . 10
- 1.11 Slide 22 . . . 10
- 1.12 Slide 24 . . . 10
- 1.13 Notes and Tasks from the Lecture . . . 11

2 Lecture 02 - Solving recurrences **12**
- 2.1 Slide 03 . . . 12
- 2.2 Slide 07 . . . 13
- 2.3 Slide 08 . . . 14
- 2.4 Slide 10 . . . 14
- 2.5 Slide 11 . . . 14

2.6 Slide 12 . . . 15
2.7 Notes and Tasks from the Lecture . . . 15

3 Lecture 03 - Divide and conquer I 16
3.1 Slide 04 . . . 16
3.2 Slide 06 . . . 16
3.3 Slide 08 . . . 16
3.4 Slide 09 . . . 18
3.5 Slide 10 . . . 18
3.6 Slide 11 . . . 18
3.7 Slide 12 . . . 18
3.8 Slide 13 . . . 19
3.9 Slide 14 . . . 20
3.10 Slide 16 . . . 20
3.11 Slide 17 . . . 21
3.12 Notes and Tasks from the Lecture . . . 21

4 Lecture 04 - Divide and conquer II 22
4.1 Slide 03 . . . 22
4.2 Slide 05 . . . 22
4.3 Slide 07 . . . 23
4.4 Slide 08 . . . 23
4.5 Slide 09 . . . 23
4.6 Slide 12 . . . 23
4.7 Notes and Tasks from the Lecture . . . 24

5 Lecture 05 - Divide and conquer III 25
5.1 Rough Plan, To Be Fleshed Out . . . 25
5.2 Notes and Tasks from the Lecture . . . 25

6 Lecture 06 - Graphs algorithms I - breadth first search 25
6.1 Global . . . 25
6.2 Slide 04 . . . 25
6.3 Slide 09 . . . 25
6.4 Slide 10 . . . 26
6.5 Slide 12 . . . 26
6.6 Slide 13 . . . 26
6.7 Slide 14 . . . 26

6.8 Slide 16 . . . 27
6.9 Slide 17 . . . 27
6.10 Slide 18 . . . 28
6.11 Slide 20 . . . 28
6.12 Slide 21 . . . 28
6.13 Notes - Éric in F24 . . . 29
6.14 Notes and Tasks from the Lecture . . . 30

7 Lecture 07 - Graph algorithms II - depth-first search 30
7.1 Global . . . 30
7.2 Slide 03 . . . 30
7.3 Slide 04 . . . 31
7.4 Slide 05 . . . 31
7.5 Slide 07 . . . 32
7.6 Slide 09 . . . 33
7.7 Slide 10 . . . 33
7.8 Slide 12 . . . 34
7.9 Slide 15 . . . 34
7.10 Slide 16 . . . 34
7.11 Slide 17 . . . 35
7.12 Slide 18 . . . 35
7.13 Notes - Éric in F24 . . . 36
7.14 Tasks . . . 36
7.15 Notes and Tasks from the Lecture . . . 36

8 Lecture 08 - Graph algorithms III - Directed graphs 37
8.1 Global . . . 37
8.2 Slide 02 . . . 37
8.3 Slide 03 . . . 37
8.4 Slide 04 . . . 37
8.5 Slide 05 . . . 37
8.6 Slide 06 . . . 38
8.7 Slide 07 . . . 38
8.8 Slide 08 . . . 38
8.9 Slide 09 . . . 38
8.10 Slide 10 . . . 39
8.11 Slide 12 . . . 39
8.12 Slide 15 . . . 39

8.13 Slide 18 . . . 40
8.14 Notes and Tasks from the Lecture . . . 40

9 Lecture 09 - Graph algorithms IV - Dijkstra's algorithm 40
9.1 Slide 03 . . . 40
9.2 Slide 04 . . . 41
9.3 Slide 06 . . . 41
9.4 Slide 07 . . . 41
9.5 Slide 10 . . . 41
9.6 Slide 11 . . . 41
9.7 Slide 12 . . . 41
9.8 Slides 13-15: Proof that Dijkstra's Algorithm is Correct . . . 41
9.9 Notes and Tasks from the Lecture . . . 43

10 Lecture 10 - Graph algorithms V - Minimum Spanning Trees 43
10.1 Slide 03 . . . 43
10.2 Slide 04 . . . 43
10.3 Slide 05 . . . 44
10.4 Slide 06 . . . 44
10.5 Slide 07 . . . 44
10.6 Slide 08 . . . 45
10.7 Slide 09 . . . 45
10.8 Slide 10 . . . 45
10.9 Slide 12 . . . 46
10.10 Slide 13 . . . 46
10.11 Notes and Tasks from the Lecture . . . 46

11 Lecture 11 - Greedy algorithms I 47
11.1 Slide 04 . . . 47
11.2 Slide 08 . . . 47
11.3 Slide 09 . . . 47
11.4 Slide 10 . . . 47
11.5 Slide 13 . . . 48
11.6 Slide 14 . . . 48
11.7 Slide 16 . . . 48
11.8 Notes and Tasks from the Lecture . . . 48

12 Lecture 12 - Snow Day **49**
12.1 Global 49

13 Lecture 13 - Greedy algorithms II **49**
13.1 Global 49
13.2 Slide 06 49
13.3 Slide 07 49
13.4 Slide 08 50
13.5 Slide 10 (Fractional Knapsack Problem) 50
13.6 Slide 11 51
13.7 Slide 12 51
13.8 Slide 13 52
13.9 Slide 14 52
13.10Slide 15 53
13.11Notes and Tasks from the Lecture 53

14 Lecture 14 - Dynamic Programming I **54**
14.1 Global 54
14.2 Slide 03 54
14.3 Slide 10 54
14.4 Slide 11 55
14.5 Slide 13 55
14.6 Slide 14 55
14.7 Slide 15 55
14.8 Slide 16 55
14.9 Notes and Tasks from the Lecture 56

15 Lecture 15 - Dynamic Programming II **56**
15.1 Global 56
15.2 Slide 2 56
15.3 Slide 3 56
15.4 Slide 4 56
15.5 Slide 5 57
15.6 Notes and Tasks from the Lecture 57

16 Lecture 16 - Dynamic programming III **58**
16.1 Global 58
16.2 Edit Distance 58

16.3 Optimal Binary Search Trees 60
16.4 Maximum Independent Sets In Trees 64
16.5 Notes and Tasks from the Lecture 65

17 Lecture 17 - Dynamic programming IV **66**
17.1 Global 66
17.2 Bellman-Ford Algorithm 66
17.3 Floyd-Warshall Algorithm 70

18 Lecture 18 - Polynomial Time Reductions **71**
18.1 Global 71
18.2 Slide 03 71
18.3 Slide 04 71
18.4 Slide 05 71
18.5 Slide 06 71
18.6 Slide 10 72
18.6.1 Proof that $Clique =_P Independent\ Set$ 72
18.6.2 Proof that $VC =_P Independent\ Set$ 73
18.7 Slide 11 74
18.7.1 Proof that $HP =_P HC$ 74
18.8 Slide 15 75
18.8.1 Proof that $3SAT \leq_P IS$ 75
18.9 Notes - Éric Lecture 19 in F24 77

19 Lecture 19 - Reductions, P, NP, co-NP **77**
19.1 Global 77
19.2 Slide 03 77
19.3 Slide 04 78
19.4 Slide 05 78
19.5 Slide 08 78
19.5.1 Explanation for the argument that Circuit-Sat is NP-complete 78
19.5.2 Explanation for how the argument in the slides follows this template 79
19.6 Slide 09 79
19.6.1 Explanation for Circuit-Sat $\leq_P$ 3SAT 79

20 Lecture 20 - NP-completeness I **80**
20.1 Global . . . 80
20.2 Notes - Éric Lecture 21 in F24 . . . 80

21 Lecture 21 - NP-completeness II **81**
21.1 Global . . . 81
21.2 Slide 02 . . . 81
21.3 Slides 03-08 . . . 81
21.3.1 Explanation for 3SAT $\leq_P$ DirectedHamiltonianCycle . . . 81
21.4 Slides 09-10 . . . 83
21.4.1 Explanation for DirectedHamiltonianCycle $\leq_P$ HamiltonianCycle . . . 83
21.5 Notes - Éric Lecture 23 in F24 . . . 84
21.6 Notes and Tasks from the Lecture . . . 84

22 Lecture 22 - NP-completeness III **85**
22.1 Global . . . 85
22.2 3-Dimensional Matching (Slides 03-08) . . . 85
22.2.1 Global . . . 85
22.2.2 Explanation for 3SAT $\leq_P$ 3DMatching . . . 85
22.3 Subset Sum (Slides 09-12) . . . 88
22.3.1 Global . . . 88
22.3.2 Explanation for 3DM $\leq_P$ SubsetSum . . . 88
22.4 Notes and Tasks from the Lecture . . . 89

23 Lecture 23 - NP-Completeness **89**
23.1 Notes - Éric Lecture 23 in F24 . . . 89

24 Lecture 24 - Misc **90**
24.1 Notes - Éric Lecture 24 in F24 . . . 90

25 Lecture 25 - Max flow **90**
25.1 Max Flow . . . 90

26 Lecture 26 - Max flow = Min cut **91**

27 Lecture 27 - Applications of Flows and Cuts **92**

Global Tasks:

1. When you make your own slide deck, include sections / subsections to make navigation easier for you, and for the students.

1 Lecture 01 - Introduction, review of asymptotics

1.1 Course Intro

1. ISC: Sylvie Davies.
2. **Textbooks**
   (a) CLRS = **Introduction to Algorithms** by Cormen, Leierson, Rivest, Stein
   (b) KT = **Algorithm Design** by Kleinberg, Tardos
   (c) DPV = **Algorithms** by Dasgupta, Papadimitriou, Vazirani

1.2 Slide 09

1. Bullet 3 is the **Limit Rule**, say from CS 240.

1.3 Slide 10

Examples True or False?

1. $2^{n-1} \in \Theta(2^n)$?
   True.
   (a) $2^{n-1} \in O(2^n)$: $c = 1, n_0 = 1$ works.
   (b) $2^{n-1} \in \Omega(2^n)$: $c = \frac{1}{2}, n_0 = 1$ works.
   Alternatively, just apply a Lemma from the CS 341 Background Information.
2. $(n-1)! \in \Theta(n!)$?
   False.
   (a) $(n-1)! \in O(n!)$ holds: $c = 1, n_0 = 1$ works.
   (b) $(n-1)! \in \Omega(n!)$ does not hold: Towards a contradiction, suppose that constants c and n_0 satisfy the definition. Choose an arbitrary

n such that $n > n_0$ and $n > \frac{1}{c}$. Then we have

$$\begin{aligned} & c \cdot n! \\ = \; & c \cdot n \cdot (n-1)! \\ > \; & c \cdot \frac{1}{c} \cdot (n-1)! \\ = \; & (n-1)!, \end{aligned}$$

which is a contradiction.

1.4 Slide 13 Exercise

1. Cost of the Sum Routine:
   (a) The for loop executes n times.
   (b) Each loop iteration requires $\underbrace{O(1)}_{\text{access } A[i]} + \underbrace{O(1)}_{+}$ time.
   (c) So we get $O(n)$ in total.

1.5 Slide 14 Exercise

1. Cost of the Product Routine:
   (a) If multiplication is a basic operation, then this is the same as the sum routine.
   (b) If multiplication is not basic, but must instead be implemented using addition, then it will be $O(n^2)$.

1.6 Slide 16

1. The problem stated here is solved (partially - we only return the sum, not the bounds that created it) in the following ways on the subsequent slides. Note that the run time improves as we go. We will explain each of these techniques, later in the course.
   (a) Brute force: 17-19
   (b) Divide-and-conquer: 20-22
   (c) Dynamic Programming: 23-25
2. We adopt the stated Convention to keep our notation as clean as possible in what follows, and not need to handle empty cases separately.

1.7 Slide 17

1. Per Armin's note, Slide 17 is not actually a solution. this is a useless pseudocode which does nothing. They have potentially seen this in CS240 as is. In the first module of CS240, this was used to show them how they can find the runtime of nested loops. There exists a reference if you look at my lecture plan.

1.8 Slide 18

1. Should all the matrix entries be negative, this algorithm will return 0. This is correct: a sum of 0 is realized by the empty sub-array.
2. The $\Theta(n^3)$ runtime is clear from the structure of the code.

1.9 Slide 19

1. The $\Theta(n^2)$ runtime is clear from the structure of the improved code.

1.10 Slide 21

1. This entire slide is to handle Case 3 from the previous slide; Cases 1 and 2 are trivial. This explains why the right boundary entry are included in `MaximizeLowerHalf` (and, symmetrically, why the left boundary entry would be included in `MaximizeUpperHalf`).

1.11 Slide 22

1. I tried, and failed, to understand Beidl's "bare hands" proof that the Divide-And-Conquer version of Maximum Subarray's worst case run time (same as MergeSort's worst case run time) lies in $\Theta(n \log n)$.
2. Every other source, including CS 341 itself, relies on a recursion tree.
3. From now on, so shall I.

1.12 Slide 24

1. The boxed pseudo-code computes $\overline{M}(n)$.

1.13 Notes and Tasks from the Lecture

1. Notes
   (a) Answer to the Question, is the W25 offering the same as the F24 offering: The topics will be mostly the same. The one exception is that the topic **max-flow/min-cut** was included during F24 but will be omitted during W25.
   (b) Slide 12 Explain better why the $\wedge$-rule is less strict than the $\vee$-rule. It is to do with the choice of C:
      i. The first version ($\vee$) makes it more difficult to fix a C, hence it is more strict.
      ii. The second version ($\wedge$) makes it easier to fix a C, hence it is less strict.
2. Tasks
   (a) Get Piazza set up and populated for the W25 term, if it's not done already (touch base with Sylvie).
   (b) Add a link to the unsecured website, to the LEARN site.
   (c) Fix my screen timeout settings!
   (d) Bring treats to class, from now on!
   (e) Consistently include or exclude the Ericson textbook everywhere (It's mentioned in Armin's slides, but not elsewhere, I think).
   (f) Post to the course website:
      i. Lecture Notes
      ii. CS 341 Background Information
   (g) Turn the Exercises into Clicker Questions, where possible.
   (h) Start L02 with the problem stated on Slide 16, and its many solutions.
   (i) Announce: no tutorials on January 10; first tutorials will be on January 17.
   (j) When we start into dynamic programming later on, recall this last example: it is a great example where, by adding some storage, and remembering work already done, we can effectively cut down our run-time.

2 Lecture 02 - Solving recurrences

2.1 Slide 03

Exercise: Prove that $T^w(n) \leq T(n)$ and $T(n)$ is increasing (an easy induction).

Solution:

1. Proof that $T^w(n) \leq T(n)$, for all $n \geq 1$:
   (a) The proof is by induction on $n \geq 1$.
   (b) Base $n = 1$:
      i. $T^w(1) = d = T(1)$.
   (c) Induction $n > 1$:
      i. The induction hypothesis is that $T^w(m) \leq T(m)$, for all $m < n$.
      ii. Then

$$\begin{aligned} & T^w(n) \\ \leq\ & T^w\left(\left\lceil \frac{n}{2} \right\rceil\right) + T^w\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn \\ \underbrace{\leq}_{I.H.}\ & T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn \\ =\ & T(n). \end{aligned}$$

2. Proof that $T(n)$ is increasing, for all $n \geq 1$:
   (a) We show that, for all $n \geq 1$, $T(n+1) > T(n)$.
   (b) The proof is by induction on $n \geq 1$.
   (c) Base $n = 1$:
      i.

$$\begin{aligned} T(1) &= d \\ T(2) &= T(1) + T(1) + cn \\ &= d + d + cn \\ &= 2d + cn \\ &> T(1), \end{aligned}$$

      since all quantities are positive.
   (d) Induction $n > 1$:
      i. The induction hypothesis is that $T(m) > T(\ell)$, for all $m > \ell$.

ii. Then

$$\begin{aligned} & T(n) \\ = \quad & T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn \\ \underbrace{>}_{I.H.} \quad & T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + c(n-1) \\ = \quad & T(n-1). \end{aligned}$$

2.2 Slide 07

1. We do the proof for n a power of b; the result holds for $n \in \mathbb{R}_{\geq 0}$.
2. As on the following slides, $T(1) = d$ (for some $d > 0$) should be part of the definition here too.
3. We should add here that $c > 0$.
4. Checking that the Master Theorem implies that, for MergeSort $T(n) \in \Theta(n \log n)$:
   (a) Let

$$\begin{aligned} a &= 2 \\ b &= 2 \\ y &= 1 \\ x &= \log_b a, \text{ so that} \\ b^x &= a. \end{aligned}$$

   This gives us that

$$x = \log_2 2 = 1,$$

   which, applying the Master Theorem, says that

$$T(n) \in \Theta(n \log n),$$

   as desired.
5. The statement of the Theorem should be clarified, to say that, when $x = y$, we get $T(n) \in \Theta(n^y \log_b n)$ (i.e. state the base explicitly - it depends on b - it is not always 2).

2.3 Slide 08

1. We should add here that $y \in \mathbb{Z}$ and $j \geq 0$.
2. The final size number, namely $\frac{n}{b^j}$, equals 1, because $n = b^j$.
3. Also the number of levels, namely $\log_b n$, equals j, because $\log_b n = \log_b(b^j) = j$.

2.4 Slide 10

1. Suggested revisions for Armin: "is a geometric sequence" $\mapsto$ "involves a geometric series"

2.5 Slide 11

Setup

1. x, y are integers.
2. $a \geq 1$ and $b \geq 2$ are integers, with $a = b^x$, equivalently $x = \log_b a$.
3. The geometric series has first term $a = 1$ and common ratio $r = \frac{a}{b^y} = \frac{b^x}{b^y} = b^{x-y}$.
4. $n = b^j$, equivalently $j = \log_b n$.
5. $a^j = (b^x)^j = (b^j)^x = n^x$.
6. Simplify r^j as much as possible:

$$r^j = \left(\frac{a}{b^y}\right)^j = \frac{a^j}{(b^j)^y} = \frac{n^x}{n^y} = n^{x-y}.$$

Cases of the proof, explained more fully

1. $r < 1$ equivalently $x < y$:
   (a) Per the CS 240 geometric series summary, $\sum_i r^i \in \Theta(1)$.
   (b) This shows that $T(n) \in \Theta(n^y)$ (since $x < y$, the second term dominates the first).
2. $r = 1$ equivalently $x = y$:
   (a) Per the CS 240 geometric series summary, $\sum_i r^i \in \Theta(j) \underbrace{=}_{j=\log_b n} \Theta(\log_b n)$.
   (b) This shows that $T(n) \in \Theta(n^y \log_b n)$ (since $x = y$, the second term dominates the first).
3. $r > 1$ equivalently $x > y$:
   (a) Per the CS 240 geometric series summary, $\sum_i r^i \in \Theta(r^{j-1})$.

(b) By a Lemma from CS 240 recalled in the Background Information document, this says that $\sum_i r^i \in \Theta(r^j) \underbrace{=}_{above} \Theta(n^{x-y})$.

(c) This shows that both terms lie in $\Theta(n^x)$, so that by the sum rule, we have $T(n) \in \Theta(n^x)$.

2.6 Slide 12

1. Example: $T(n) = 2T\left(\frac{n}{2}\right) + n, T(1) = 0, n$ a power of 2.
   (a) In the notation of the Master Theorem:

$$\begin{aligned} a &= 2 \\ b &= 2 \\ y &= 1 \\ x &= \log_b a \\ &= \log_2 2 \\ &= 1 \\ x &= y, \text{ equivalently} \\ r &= 1, \text{ so that} \\ T(n) &\in \Theta(n^y \log_b n) \\ &= \Theta(n \log n). \end{aligned}$$

2. CR to type up the notes on the guess-and-check approach to solving this example.

2.7 Notes and Tasks from the Lecture

1. Notes
   (a) Our course convention is (as it was in CS 240) that the base of log is 2, unless otherwise specified.
2. Tasks
   (a) Correct the suggested readings on the course website, in the second half of the term.

3 Lecture 03 - Divide and conquer I

3.1 Slide 04

1. **Examples:** Amazon, YouTube, etc where you are a member of a group who are all interested in some stuff.
2. We are not trying to solve the collaborative filtering problem. What we are trying to solve is one of the many tools which might be useful in collaborative filtering.
3. The Padlet question here is just to give them some time to think about the problem and hopefully convinces them what we are doing has some applications.
4. **Answer to Exercise:** Something like "compare the similarity of two rankings" is a good answer.
5. Counting inversion is related to the answer. It is counting the places in two rankings which are different.

3.2 Slide 06

1. Notation:
   - c_ℓ: # of inversions in $A\left[1, \ldots, \frac{n}{2}\right]$
   - c_r: # of inversions in $A\left[\frac{n}{2}+1, \ldots, n\right]$
   - c_t: # of **transverse** inversions $i \leq \frac{n}{2}, j > \frac{n}{2}$.
2. Example: $A = [1, 5, 2, 6, 3, 8, 7, 4], n = 8$. Then

$$\begin{aligned} c_\ell &= 1 - Swap : (2, 5) \\ c_r &= 3 - Swap : (8, 7), (8, 4), (7, 4) \\ c_t &= 4 - Swap : (6, 3), (6, 4), (5, 3), (5, 4) \end{aligned}$$

   Note, this accounts for all of the 8 inversions we listed earlier, on Slide 05.

3.3 Slide 08

1. **Claim:** $T(n) = 2T\left(\frac{n}{2}\right) + cn \log n$ gives $T(n) \in \Theta(n \log^2 n)$.

Proof. Sketchy proof that $T(n) \in O(n \log^2 n)$

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \log n \\
&= 2\left[2T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right)\right] + cn \log n \\
&= 4T\left(\frac{n}{4}\right) + cn \log\left(\frac{n}{2}\right) + cn \log n \\
&\cdots \\
&= cn\left[\underbrace{\log 2 + \log 4 + \cdots + \log n}_{\log n \text{ terms}}\right] \\
&\leq cn\left[\underbrace{\log n + \log n + \cdots + \log n}_{\log n \text{ terms}}\right] \\
&= cn \log^2 n.
\end{aligned}$$

Proof that $T(n) \in \Omega(n \log^2 n)$

This proof follows the technique of substitution outlined in CLRS. Suppose that there exists a constant $d > 0$ and an n_0 such that, for all $n \geq n_0$,

$$d\left(\frac{n}{2}\right)\log^2\left(\frac{n}{2}\right) \leq T\left(\frac{n}{2}\right).$$

Then

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \log n \\
&\geq 2\left[d\left(\frac{n}{2}\right)\log^2\left(\frac{n}{2}\right)\right] + cn \log n \\
&= dn\left(\log n - \log 2\right)^2 + cn \log n \\
&= dn\left(\log n - 1\right)^2 + cn \log n \\
&= dn\left(\log^2 n - 2\log n + 1\right) + cn \log n \\
&= dn \log^2 n + dn(1 - 2\log n) + cn \log n \\
&\geq dn \log^2 n,
\end{aligned}$$

provided $dn(1 - 2\log n) + cn \log n \geq 0$, which will hold provided

$$d \leq \frac{c \log n}{2 \log n - 1}.$$

No boundary conditions are given. We could work through the boundary conditions as in CLRS, if needed. □

3.4 Slide 09

1. Recall the notation: c_t denotes the number of **transverse** inversions, with $i \leq \frac{n}{2}, j > \frac{n}{2}$.

3.5 Slide 10

1. The array A in this example is the same as in the previous example. Hence the counts of inversions are also the same.
2. How to Compute c_t:
   (a) Keep a running total.
   (b) Each time we insert $S[i]$ into A, count how many new transverse inversions have been carried out since the previous $S[i]$-insertion.
   (c) line 5: j has gotten to big; all right-hand entries are already inserted. Hence the i^{th} entry must be transversely inverted with all of the right hand entries, $\frac{n}{2}$ of them. This gives $c = c + \frac{n}{2}$.
   (d) line 6: j is still in bounds; the i^{th} entry must be transversely inverted with the right hand entries inserted to date, $j - \left(\frac{n}{2} + 1\right)$ of them. This gives $c = c + j - \left(\frac{n}{2} + 1\right)$.
3. We showed in Lecture 02 (Slide 07) that Mergesort has $T(n) \in O(n \log n)$. The merge then contributed $d_n \in O(n)$ then; this part is the same here.

3.6 Slide 11

1. No divide and conquer yet. It's coming on the next slide.
2. The first, brute force approach is in $\Theta(n^2)$.

3.7 Slide 12

1. Assume that n is even.
2. F_0 captures the low-order terms of F (and G_0 does the same for G).
3. F_1 captures the high-order terms of F (and G_1 does the same for G).
4. Exercise: Want: $F_0G_1+F_1G_0$, using only one polynomial multiplication, starting from $F_0, F_1, G_0, G_1, F_0G_0, F_1G_1$.

$$\begin{aligned} & (F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1 \\ = \; & F_0G_0 + F_0G_1 + F_1G_0 + F_1G_1 - F_0G_0 - F_1G_1 \\ = \; & F_0G_1 + F_1G_0, \end{aligned}$$

3.8 Slide 13

1. Check the identity:

$$\begin{aligned} & (F_0 + F_1 x^{\frac{n}{2}})(G_0 + G_1 x^{\frac{n}{2}}) \\ = \; & F_0 G_0 + (F_0 G_1 + F_1 G_0) x^{\frac{n}{2}} + F_1 G_1 x^n, \end{aligned}$$

   so that we will be done if we can confirm that the middle coefficient equals $(F_0 + F_1)(G_0 + G_1) - F_0 G_0 - F_1 G_1$. But this is exactly the exercise from the previous slide, no?
2. Analysis: 3 recursive calls, each in size $\frac{n}{2}$:
   (a) $F_0 G_0$
   (b) $(F_0 + F_1)(G_0 + G_1)$
   (c) $F_1 G_1$
3. $T(n) = 3T\left(\frac{n}{2}\right) + cn$, analyzed using the Master Theorem:

$$\begin{aligned} a &= 3 \\ b &= 2 \\ y &= 1 \\ x &= \log_b a \\ &= \log_2 3 \\ &= \frac{\ln 3}{\ln 2} \\ &\approx 1.58 \\ x &> y, \text{ so that} \\ r &> 1, \text{ and therefore} \\ T(n) &\in \Theta(n^x) \\ &= \Theta(n^{\log_2 3}). \end{aligned}$$

3.9 Slide 14

1. Gets close to exponent 1, as $k \to \infty$. Check:

$$\begin{aligned} & \lim_{k\to\infty} \log_k(2k-1) \\ = \quad & \lim_{k\to\infty} \frac{\ln(2k-1)}{\ln k} \\ \underbrace{=}_{L'Hopital} \quad & \lim_{k\to\infty} \frac{\frac{2}{2k-1}}{\frac{1}{k}} \\ = \quad & \lim_{k\to\infty} \frac{2k}{2k-1} \\ = \quad & 1.\checkmark \end{aligned}$$

2. FFT stands for **Fast Fourier Transforms**.

3.10 Slide 16

1. $T(n)$, analyzed using the Master Theorem:

$$\begin{aligned} a &= 8 \\ b &= 2 \\ y &= 2 \\ x &= \log_b a \\ &= \log_2 8 \\ &= 3 \\ x &> y, \text{ so that} \\ r &> 1, \text{ and therefore} \\ T(n) &\in \Theta(n^x) \\ &= \Theta(n^3). \end{aligned}$$

3.11 Slide 17

1. $T(n)$, analyzed using the Master Theorem:

$$\begin{aligned}
a &= 7 \\
b &= 2 \\
y &= 2 \\
x &= \log_b a \\
&= \log_2 7 \\
&= \frac{\ln 7}{\ln 2} \\
&\approx 2.807 \\
x &> y, \text{ so that} \\
r &> 1, \text{ and therefore} \\
T(n) &\in \Theta(n^x) \\
&= \Theta(n^{\log_2 7}).
\end{aligned}$$

3.12 Notes and Tasks from the Lecture

1. Notes
   (a) Our course standard will be to number our arrays starting from 1, not from 0. We will explicitly state if any particular example deviates from this standard.
   (b) Slide 2: If possible, remove the extraneous page down at the end of the page. Ask Armin.
   (c) Slide 18: Do the results quoted here sit on top of the approach taught in CS 370? Ask Armin/Mark.
   (d) Slide 19: Should this blank page at the end be removed? Ask Armin/Mark.
2. Tasks
   (a) Update the website:
      i. Post office hours, and start holding them this week.
   (b) Slides 7-8: Make it clearer where we are talking about entries, not indices. Where appropriate, change i into $A[i]$. Suggest to Armin to revise the slides accordingly.
   (c) Document, for Exams:
      i. Reference Sheets

ii. Study Guide (what you will need to memorize, and what you won't)
iii. Practice Problems, about topics covered by the exam but not yet by any assignment.

4 Lecture 04 - Divide and conquer II

4.1 Slide 03

1. Brute-force: $\Theta(n^2)$.
2. Goal: $\Theta(n \log n)$, using a Divide-and-Conquer approach.
3. See §33.4 in CLRS:
   (a) Divide: Find a vertical line which bisects the point set into L and R, of equal sizes (see the following pictures).
   (b) Conquer: Make two recursive calls, one to handle each of the subsets created above. This returns δ_L and δ_R, both of which are needed as described below.
   (c) Combine: Take the minimum over the three possibilities arising from the setup:
      i. min in L
      ii. min in R
      iii. min is transverse

4.2 Slide 05

1. $dist(P, R)$ and $dist(Q, L)$ are horizontal distances. In this example, this is where the white band comes from. $\delta = 4$, so the white band covers all points at $dist \leq 4$ from the other side.
2. I suggest the more clear notation $y_P \leq y_Q < y_P + \delta$, instead of $y_P \leq y < y_P + \delta$. We have already restricted to the one point of interest on the left, labelled P at the previous step: it is the only point in the white band created in the previous step.
3. One small confusing point: we were looking for **transverse** pairs just a moment ago, but the constructed rectangle contains points on the left.

4.3 Slide 07

1. A square on the left contains at most one point from L. Reason: If some square contained two points, then the distance separating them would be $\leq \frac{\delta}{2} < \delta$, contradicting the definition of δ.
2. The same argument shows that the square on the right contains at most one point from R.

4.4 Slide 08

1. The reason for $O(n \log n)$ runtime for initialization: sort the points twice, with respect to x and y (c.f. **kd-trees**, in Module 8 of CS 240).
2. Finding the x-median is easy, because we have already sorted the points by x-co-ordinate, when we initialized.
3. Run time: Recursive calls: all to justify the $\Theta(n)$ term in the recursive formula.

4.5 Slide 09

1. We should standardize our notation here. In Lecture 03, our arrays were indexed $1 \ldots n$. Here our arrays are indexed $0 \ldots n-1$.
2. I also suggest that we create a new line for the heading "Known Results". Talk to Armin.
3. Reason why a randomized algorithm has expected run time in $\Theta(n)$: Refer to CS 240, Module 03, Section on Randomized Algorithms.
4. Assumption: All the $A[i]$s are distinct.

4.6 Slide 12

1. Explanation for $\frac{3n}{10}$:
   (a) $\frac{1}{2}$ of the m_is are $> p$.
   (b) There are $\frac{n}{5}$ m_is.
   (c) So the number of m_is that are $> p$ is $\left(\frac{1}{2}\right)\left(\frac{n}{5}\right) = \frac{n}{10}$.
   (d) Each m_i is the median of a set of size 5; hence there are 3 entries in that set of size 5 which are $\geq m_i$.
   (e) Each of these 3 entries is $\geq m_i > p$, by transitivity.
   (f) Hence the total number of entries which are $> p$ is $3\left(\frac{n}{10}\right) = \frac{3n}{10}$.

2. Why "same thing for $n-i-1$" is correct: swap less / greater throughout: the analysis still works the same way.
3. If time permits, you can (make sure you tell the students this is optional, since it's not part of the W25 slide deck) Slide 13 from Éric's slide deck.
4. This (almost) completes our section on divide-and-conquer.
5. We will actually finish it at the end of Lecture 05, using the remaining time for additional examples and techniques.

4.7 Notes and Tasks from the Lecture

1. Notes
   (a) Slide 04
      i. Explain that the point on the vertical boundary is another choice for P, to be handled at a different time.
   (b) Slide 05
      i. Label the RH point as Q? Ask Armin.
      ii. Why is it enough to
         A. draw the rectangle with P at its bottom, i.e.
         B. only consider points with $y_P \leq y \leq y_P + \delta$?
   (c) Slide 07
      i. Explain why the maximum distance between two points in one of the small squares is $< \delta$: The diagonal distance for a square with side length $\frac{\delta}{2}$ equals $\left(\frac{\sqrt{2}}{2}\right)\delta < \delta$.
   (d) Slide 08
      i. Explain where the recursion stops: for any set containing ≤ 2 points, no recursive calls are needed - just handle transverse pairs.
   (e) Slide 11
      i. Why 5? So far, it appears arbitrary. The analysis happens to work out.
   (f) Slide 12
      i. **Note:** In this example, **we do not actually divide**: We just create one smaller instance to process at each level of recursion!
2. Tasks
   (a) Run a poll (on Piazza?) to discover whether there is any appetite

for virtual office hours.

5 Lecture 05 - Divide and conquer III

5.1 Rough Plan, To Be Fleshed Out

1. Method of Substitution, from CLRS.
   (a) Rigourous proof that $T(n) = 2T\left(\frac{n}{2}\right) + dn \log n$ gives $T(n) \in \Theta(n \log^2 n)$.
2. Method of Change of Variables, from CLRS.
3. Correctness Proof(s), skipped earlier, from CLRS.

5.2 Notes and Tasks from the Lecture

1. Notes
   (a) Question: Is there any improved version of the **Master Theorem**, which can handle recursions like $T(n) = 2T\left(\frac{n}{2}\right) + dn \log n$?
2. Tasks
   (a) Post the clicker questions to the course website.

6 Lecture 06 - Graphs algorithms I - breadth first search

6.1 Global

1. Refer to §20.2 in CLRS.

6.2 Slide 04

1. Both representations of the provided graph are worked out in CLRS.

6.3 Slide 09

1. s is a chosen **source** vertex.
2. Note that we are using the adjacency list representation of the graph here (as stated in the pseudocode).

6.4 Slide 10

1. Explanation of $O(n+m)$ run time for BFS, from CLRS
   (a) Each vertex is enqueued at most once, and hence dequeued at most once.
   (b) The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(n)$.
   (c) Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once.
   (d) Since the sum of the lengths of all n adjacency lists is $\Theta(m)$, the total time spent in scanning adjacency lists is $O(n+m)$.
   (e) The overhead for initialization is $O(n)$, and thus the total running time of the BFS procedure is $O(n+m)$.
   (f) Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.
2. One useful further comment from Armin: $O(nm)$ would mean that we are looking at all edges for each dequeued node, which is not happening here.

6.5 Slide 12

1. The proof is by induction **on** i.
2. "by assumption" $\mapsto$ "by the induction hypothesis"
3. Reason why "$\{v_j, v_i\}$ is in E": From the algorithm, we discovered v_i as a neighbour of v_j.

6.6 Slide 13

1. The proof is by induction **on** i.
2. We should keep our induction approach similar across these proofs. In the previous proof, we used strong induction for $i > 0$. In this proof we use weak induction to get from i to $i+1$.

6.7 Slide 14

1. Exercise:

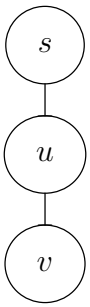
(a) The contrapositive is that $m < n - 1$ implies that G is not connected.
(b) **Claim:** With $m \geq 1$ edges, at most $m + 1$ vertices can be connected.
(c) This can be proved by a straightforward induction on $m \geq 1$.
(d) The claim clearly implies the above contrapositive.

6.8 Slide 16

1. The algorithm set s to be its own parent. This is why we need to exclude the edge from s back to itself in the setup here.
2. Explanation for why T remains a tree when we set $parent[w] \leftarrow v$
   (a) The new graph is connected, because the old graph was.
   (b) We add a vertex, which has only one edge incident with it. There is no possibility of creating a cycle by doing this.
3. For a rigourous proof of Sub-Claim 1, see Lemma 20.3 in CLRS, 4th Edition (Equivalently, Lemma 22.3 in CLRS, 2nd Edition).

6.9 Slide 17

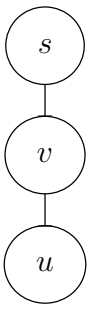
1. Consider these two examples, corresponding with the two cases of the proof that follows:
   (a)



$$\begin{aligned} level[u] &= 1 \\ level[v] &= 2 \end{aligned}$$

u is dequeued before v.

(b)



$$\begin{aligned} level[u] &= 2 \\ level[v] &= 1 \end{aligned}$$

v is dequeued before u.

2. Improved Proof:
   (a) If we dequeue v before u, then $level[v] \leq level[u]$. which implies $level[v] \leq level[u] + 1$.
   (b) Otherwise we dequeue u before v. Then the parent of v is either u, or was dequeued before u.
      i. If the parent of v is u, then $level[v] = level[u] + 1$. This implies $level[v] \leq level[u] + 1$.

   Otherwise the parent of v was dequeued before u. This implies that $level[v.parent] \leq level[u]$. Also, $level[v] = level[v.parent] + 1$. Putting it all together gives $level[v] = level[v.parent] + 1 \leq level[u] + 1$, as required.

6.10 Slide 18

1. The induction is on i.

6.11 Slide 20

1. We can swap W_1 and W_2, as needed.

6.12 Slide 21

1. Use an integer to keep track of the "colours" that identify each component.

2. Start BFS at a vertex v.
3. When it finishes, all vertices that are reachable from v are colored (i.e., labeled with a number).
4. Loop through all vertices which are still unlabeled and call BFS on those unlabeled vertices to find other components.

6.13 Notes - Éric in F24

1. To start, no edge can connect to itself, so every edge is defined by a pair of **distinct** nodes.
2. Convince yourself that, given a graph, the m mentioned in the definitions is constant.
3. **Good Student Question:** Given a tree, does it matter which node we choose to be the root?
   A: No! Parent-child relationships will change, but no properties that we will need will change, if we make a different choice of root!
4. Convince yourself that Éric's statement that induction and contradiction are really the same thing (to prove POMI is correct, we argue by contradiction), is actually correct!
5. Correctness 1 needs strong induction; Correctness 2 needs only simple induction.
6. To test whether there is a walk from v to w, run BFS from v, then test whether $visited(w) = true$.
7. To test whether a graph is connected, run BFS from anywhere, then test that $m = n - 1$.
8. A given vertex comes out of the queue at most once (it can only go into the queue once; it might never come out).
9. d_v denotes the **degree** of vertex v (i.e. the number of edges emanating from it).
10. **Keeping track of parents and levels**: Now, to test whether a node was visited, we check whether its parent is not NIL.
11. **Graph Convention:** The distance between two nodes which are not connected, is **infinite**.
12. Shortest paths from the BFS tree:
    (a) Let $v_0 \to v_1 \to \cdots \to v_{i-1} \to v_i \to \cdots \to v \to v_k$ be a shortest path $s \to v$.
    (b) $level(v) \leq dist(s, v) = k$.
    (c) For all $i, level(i) \leq i$.

(d) The level of the parent of v_i is either v_{i-1} or a node that came before v_{i-1}.
(e) $level(parent(v_i)) \leq level(v_{i-1})$

6.14 Notes and Tasks from the Lecture

1. Notes
   (a) **Q:** In the adjacency list example, does it matter that the linked lists are sorted?
   A: No. I just ordered them in the example to be systematic, and make sure that I didn't miss anything.
   (b) **Q:** Does Sub-Claim 1 imply that the constructed graph has no cycles?
   A: Collin to answer, ASAP.
   (c) All the content of Sub-Claims 1 and 2 happens within the context of the BFS Tree.
   (d) Where to Start L07: L05, Slide 18 (the 2-part claim about the BFS tree, for a graph G).
2. Tasks
   (a) Better explain the statement and proof of Sub-Claim 2: $level[v] \leq level[u] + 1$.

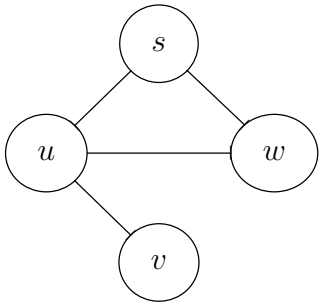
7 Lecture 07 - Graph algorithms II - depth-first search

7.1 Global

1. Refer to §20.3 in CLRS.

7.2 Slide 03

Example: Perform depth-first-search on the graph



Solution:

1. Start from s.
2. Visit s's first child, namely u.
3. Visit u's first child, namely v.
4. We cannot go any deeper: v has no children. We must now back up to u.
5. Visit u's next child, namely w.
6. We cannot go any deeper: w has no children. We must now back up to u.
7. u has no more children. We must now back up to v.
8. v has no more children. We are now done.

7.3 Slide 04

1. This first version of the algorithm simply records which nodes have been visited.
2. As is pointed out in the final bullet, we could easily enhance this to add a parent array, as in BFS.

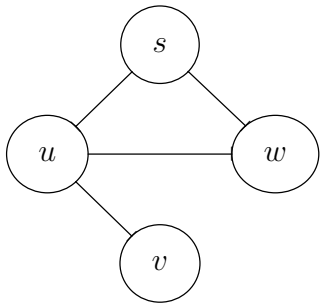
7.4 Slide 05

1. The proof is by induction on $i \geq 0$.
2. Improved Induction Step ($i > 0$):
   (a) Induction Hypothesis: The result is true for some $i < k$, i.e. we visit v_i during $explore(v)$.
   (b) We will show that the result is then true for $i + 1$.
   (c) By our assumption, $explore(v)$ is not yet finished.
   (d) Because there is a path $v = v_0, \ldots, v_i, v_{i+1}, \ldots, v_k = w$, therefore v_{i+1} is a neighbour of v_i.

(e) 2 cases for when we reach v_{i+1} at step 3 of *explore*:
   i. We have already visited v_{i+1}: the desired result holds.
   ii. We have not already visited v_{i+1}: we visit it now; again the desired result holds.

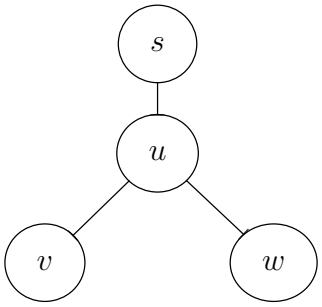
7.5 Slide 07

1. Explanation for Why Run Time is in $\Theta(n+m)$ See pp 566-567 of CLRS!
   (a) The loop on line 2 of DFS takes $\Theta(n)$ time, exclusive of the time to execute the calls to *explore*.
   (b) As we did for breadth-first search, we use aggregate analysis.
   (c) The procedure *explore* is called exactly once for each vertex $v \in V$, since the vertex u on which *explore* is invoked must not be visited yet, and the first thing *explore* does is set vertex u to visited.
   (d) During an execution of *explore*, the loop on line 2 executes $|Adj(v)|$ times.
   (e) Since $\sum_{v \in V} |Adj(v)| \in \Theta(m)$ and *explore* is called once per vertex, the Depth-first search total cost of executing *explore* is $\Theta(n+m)$.
   (f) The running time of DFS is therefore in $\Theta(n+m)$.
2. Example to show DFS need not find the shortest path: Consider the example that started the lecture:



   (a) Building a tree from this graph starting from s and using DFS

yields:



(b) In the original graph, $dist(s, w) = 1$.
(c) But, in the constructed tree, these nodes are at distance of 2 from each other.

7.6 Slide 09

1. Improved Proof:
   (a) Let $\{v, w\}$ be any edge in G.
   (b) W.L.O.G. Suppose we visit v first (If not, just swap).
   (c) Since we visit v before w, therefore at the time we visit v, $\{v, w\}$ is an unvisited path of G.
   (d) Therefore, by the White Path Lemma, w will be visited during $explore(v)$.
   (e) Hence there is a path $v \rightsquigarrow w$.
   (f) Hence v is an ancestor of w.

7.7 Slide 10

1. Explanation for Observation: CLRS Theorem 20.10.
   (a) Let $\{u, v\}$ be an arbitrary edge of G, and suppose without loss of generality that u is visited before v. Then, during $explore(u)$, the search must discover and finish v before it finishes u, since v is on u's adjacency list.
      i. If the first time that the search explores edge $\{u, v\}$, it is in the direction from u to v, then v is undiscovered until that time, for otherwise the search would have explored this edge already in the direction from v to u. Thus, $\{u, v\}$ becomes a tree edge.

ii. If the search explores $\{u, v\}$ first in the direction from v to u, then $\{u, v\}$ is a back edge, since there must already be a path of tree edges from u to v (by assumption, u was visited before v).

7.8 Slide 12

1. Explanation:
   (a) First bullet of Observation:
      i. first (red) case: the intervals are distinct
      ii. second (blue) case: the interval for v is contained in the interval for u
   (b) If $finish[u] < start[v]$, then there is nothing to prove.
   (c) Otherwise, it must be that $start[v] < finish[u]$. In this case, observing that we pop v before we pop u implies that $finish[v] < finish[u]$.

7.9 Slide 15

1. The first bullet is a proof of the contrapositive of "if s is a cut vertex, then s has > 1 child".
2. The second bullet is a proof of "if s has > 1 child, then s is a cut vertex".

7.10 Slide 16

1. The black edges are tree edges.
2. The green edges are back edges.
3. The levels are determined when the DFS tree is created, and are not changed after that.
4. The descendants are determined by the DFS tree structure.
5. Why $a(v) = 1$: Level 1 is reachable via an edge, from v.
6. Why $m(v) = 0$: The bottom-left node is a descendant of v; its $a(\cdot)$ equals 0.
7. The test on slide 17 correctly identifies v as a cut vertex: its right child has $m(\cdot) = 2 \geq level[v]$.

7.11 Slide 17

1. Improved Proof
   (a) Take a child w of v.
   (b) Let T_w be the subtree rooted at w.
   (c) Let also T_v be the subtree rooted at v.
   (d) We have these cases:
      i. For all children w of v, $m(w) < level[v]$
         A. Then there is an edge from T_w to a vertex above v.
         B. Therefore after removing v, T_w remains connected to the root.
         C. Hence v is not a cut vertex.
         D. We have proved "If v does not have a child w with $m(w) \geq level[v]$, then v is not a cut vertex." This is contrapositive of "If v is a cut vertex, then v has a child w with $m(w) \geq level[v]$".
      ii. v has a child with $m(w) \geq level[v]$
         A. I claim that all edges originating from T_w end in T_v (so that v is a cut vertex). **Proof:** Consider any edge originating from $x \in T_w$. Then since $m(w) \geq level[v]$, it follows that this edge from x ends at a level at least $level[v]$. By the Key Property, this edge connects x to one of its ancestors or descendants. This proves the claim.

7.12 Slide 18

1. Why $m(v) = \min\{a(v), m(w_1), \ldots, m(w_k))\}$: The minimum level reachable from v is either reachable directly from v, or reachable from a descendant of v.
2. How to compute all $m(w_1), \ldots, m(w_k)$ in $O(m)$:
   (a) Use recursion (with memoization) to control which node we are at in the tree.
   (b) Update a globally available array of m values, for the current node.
   (c) The base case is a leaf, obviously. Since a leaf has no children, its $m(\cdot)$ simply equals its $a(\cdot)$ value.
   (d) Do the above min-check for all internal nodes.
   (e) Compute $a(\cdot)$ as usual for the node (using its adjacency list).

7.13 Notes - Éric in F24

1. Connected components are the equivalence classes under the equivalence relation: $a \sim b$ if and only if there exists a path from a to b.
2. DFS is BFS, with the queue replaced by a stack.
3. DFS is much more natural to define, using recursion.
4. CLRS colour scheme:
   (a) white - not started visiting yet
   (b) grey - visiting in process (on the stack)
   (c) black - visiting is completed

7.14 Tasks

1. Look up the odd-cycle lemma (I think), from MATH 239?

7.15 Notes and Tasks from the Lecture

1. Notes
   (a) **Q:** Is the worst-case run-time for solving a maze the same for BFS as for DFS?
   A: Intuitively, I think so. I will confirm ASAP.
   (b) **Slide 09**
      i. Up to re-labelling of the first vertex visited, this is correctly stated in the context of G (and not in the constructed DFS tree).
      ii. The proof (I think) ignores the case where $\{v, w\}$ was already visited when constructing the tree. Or does it? Check it again; explain it better.
2. Tasks
   (a) Before the mid-term and final exams, suggest some CLRS exercises, useful for preparation.
   (b) Produce a better graph example that exercises both inductive cases of the proof of the White Path Lemma, at different times.
   (c) Start L08 at **Cut Vertices**.

8 Lecture 08 - Graph algorithms III - Directed graphs

8.1 Global

1. Refer to §20.4 and §20.5 in CLRS.
2. For an example of DFS carried out on a directed graph, see CLRS, Figure 20.4

8.2 Slide 02

1. The left-hand example at the bottom of the slide is cyclic.
2. The right-hand example at the bottom of the slide is acyclic (all edges pass from left to right).

8.3 Slide 03

1. The adjacency lists now depend on the directions of the edges.

8.4 Slide 04

1. The last bullet refers to edges in G, not in the DFS tree of G.
2. There can exist edges **in** G connecting the trees T_i.
3. We will call these **cross edges**, starting on the next slide.

8.5 Slide 05

1. The left-hand subgraph is constructed first.
2. The right-hand subgraph is constructed second.
3. This is why the cross edge goes from right to left.
4. **Why no cycle can contain two cross edges:**
   (a) A cross edge can be found only after one subtree (the LH one in this example) is finished, then we find an edge going back to the finished tree later on.
   (b) The second cross edge which would form a cycle (going L-¿R in this example) would instead have become a tree edge.

8.6 Slide 06

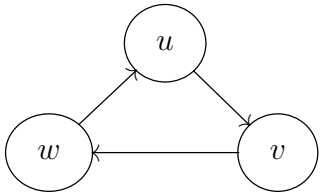
1. See Armin's hand-written notes for brief explanations of the 4 cases. Type these up when time permits.

8.7 Slide 07

1. Reason Why Acyclicity is in $O(n+m)$: Because DFS is, and because the start / end times we need to record in order to detect back edges (Slide 06) can be found in constant time while constructing the DFS tree.

8.8 Slide 08

1. A **topological order** is useful to find an order of doing tasks. E.g. a DAG might be used to model dependency relations. E.g. we can model course pre-requisites using a DAG. Then to find an order of taking courses, we find a topological ordering on this graph.
2. The example on the slide is from a recipe.
3. No such order **is possible** if G contains a cycle. item E.g. no topological ordering is possible on

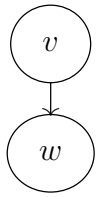


4. A topological order would require $u < v < w < u$. By the transitivity of $<$, this is impossible.
5. The prototype for an ordering, $<$, is the usual $<$ on $\mathbb{R}$: any two non-equal elements can be compared.

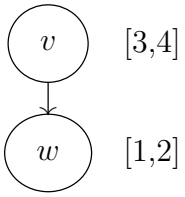
8.9 Slide 09

1. We want to use DFS to find a topological ordering.
2. We can simply think of start and finish times.

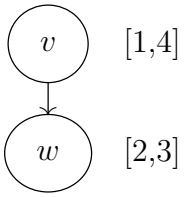
3. Consider a simple DAG:



4. Starting DFS on the bottom vertex gives:



5. Starting DFS on the top vertex gives:



8.10 Slide 10

1. This Proposition affords us a linear time algorithm for finding a topological order, using DFS.

8.11 Slide 12

1. Reason Why Strong Connectivity is in $O(n+m)$: Because DFS is.

8.12 Slide 15

1. Explanation for how to efficiently get the vertices in reverse order of finishing from step #1, as input for step #2: Create an array `reversefinish`, of size n. Fill this array up, from the end to the beginning, as we finish processing vertices: first finished is last in the array; last finished is first in the array. When we start on step #2, process the vertices in the order captured in forward order by the `reversefinish` array.

8.13 Slide 18

1. I dislike the induction setup here. t is the thing about which we are making our argument; therefore it would be better not to also use it as an ingredient in the setup of the induction case. Also, we should explain the base case, trivial as it is: $t = s$.
2. Explanation for the bullet stating "if (2), with our induction assumption, we get $start[u] < start[t]$":
   (a) The assumption of case (2) gives $start[u] < finish[u] < start[s] < finish[s]$.
   (b) Above, the induction assumption gives $start[s] \leq start[t] < finish[t] \leq finish[s]$.
   (c) Therefore we have the chain of inequalities $start[u] < finish[u] < start[s] \leq start[t]$, in other words $start[u] < start[t]$, as claimed.

8.14 Notes and Tasks from the Lecture

1. Notes
   (a) **Q:** What (if anything) is the analog of cut vertices in the directed case?
   A: We are not very interested in this, actually. We are only interested in the definition of weak connectivity versus strong connectivity.
   (b) Slide 12: Does the choice of s matter? I am fairly certain that it doesn't - If one vertex works, then any vertex works. Check it!
2. Tasks
   (a) Start L09 at Armin's L07 slide deck, Slide 10, the proof that $<$ constitutes a topological order.

9 Lecture 09 - Graph algorithms IV - Dijkstra's algorithm

9.1 Slide 03

1. This, and all subsequent questions, refer to the weights. All shortest paths are in terms of the weights.

9.2 Slide 04

1. Make the question into a Clicker question; the correct answer is: true.

9.3 Slide 06

1. For all vertices v, ...
2. The estimate $d[v]$ will become the actual value $d[v]$ by the end of the algorithm.
3. **Q for Armin:** Can we make all the letters lower case (here and on the next slide), for consistency?

9.4 Slide 07

1. CLRS calls the process of updating the edges emanating from u **relaxation**.
2. Collin to add the CLRS notion of "relaxing" of a vertex.

9.5 Slide 10

1. $Extract-Min(Q)$ removes u from Q, which makes the while loop work correctly.

9.6 Slide 11

1. Think of Q as a min-priority queue, replacing the ordinaty queue from BFS.
2. Line 8, array imp: explanation for why is it $O(|V|^2)$: We need to extract n times, each time searching an array of size $n-i$ for the smallest degree element.

9.7 Slide 12

1. First bullet: when u is added to C.

9.8 Slides 13-15: Proof that Dijkstra's Algorithm is Correct

1. **Convergence Property (CLRS Lemma 22.14):** Assuming that

$s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and that $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v). Then $d[v] = \delta(s, v)$ at all times afterward.

2. The proof is based on Theorem 22.6 in CLRS: When Dijkstra's Algorithm terminates, we have $d[u] = \delta(s, u)$, for all vertices $v \in V$.
3. We will show that at the start of each iteration of the while loop of lines 7-13, we have $d[v] = \delta(s, v)$ for all $v \in C$. This suffices, because the algorithm terminates when $C = V$, so that then, $d[v] = \delta(s, v)$ for all $v \in V$.
4. The proof is by induction on the number of iterations of the while loop, which equals $|C|$ at the start of each iteration. There are two bases:
   (a) for $|C| = 0$, so that $C = \emptyset$ and the claim is trivially true, and
   (b) for $|C| = 1$, so that $C = \{s\}$ and $d[s] = 0 = \delta(s, s)$.
5. For the inductive step, the inductive hypothesis is that $d[v] = \delta(s, v)$ for all $v \in C$. The algorithm extracts vertex u from $V \setminus C$.
6. Because the algorithm adds u into C, we need to show that $d[u] = \delta(s, u)$ at that time.
7. If there is no path from s to u, then $d[s] = \infty = \delta(s, s)$.
8. If there is a path $s \rightsquigarrow u$, then, (as CLRS Figure 22.7 shows), let y be the first vertex on a shortest path from s to u that is not in C, and let $x \in C$ be the predecessor of y on that shortest path. (We could have $y = u$ or $x = s$ or both.)
9. Because y appears no later than u on the shortest path and all edge weights are nonnegative, we have $\delta(s, y) \leq \delta(s, u)$.
10. Because the call of `EXTRACT-MIN` in line 8 returned u as having the minimum d value in $V \setminus C$, we also have $d[u] \leq d[y]$, and the upper-bound property gives $\delta(s, u) \leq d[u]$.
11. Since $x \in C$, the inductive hypothesis implies that $d[x] = \delta(s, x)$.
12. During the iteration of the while loop that added x into C, the edge (x, y) was relaxed.
13. By the convergence property, $d[y]$ received the value of $\delta(s, y)$ at that time.
14. Thus, we have $\delta(s, y) \leq \delta(s, u) \leq d[u] \leq d[y]$ and $d[y] = \delta(s, y)$, so that $\delta(s, y) = \delta(s, u) = d[u] = d[y]$.
15. Hence, $d[u] = \delta(s, u)$, and by the upper-bound property, this value never changes again.

9.9 Notes and Tasks from the Lecture

1. Notes
   (a) At the start of the Lecture 09 Slide deck, point out that we are now back to the undirected case.
   (b) L09, Slide 04: Jump to the end of the slide deck for run-time analysis (it won't all make sense yet, though).
   (c) L09, Slide 05: The motivation to modify, is to prove the correctness of **Kruskal's Algorithm**.
   (d) **Q for Armin:** Does **weakly connected** mean that there exists a source vertex, s, from which we can reach any other vertex, but not necessarily navigate back to s? Does this definition go by some other similar name?
   (e) A strongly connected graph will always contain a cycle, if it contains enough vertices.
2. Tasks
   (a) Make slides for the proof of the correctness of Dijkstra's Algorithm.
   (b) **Where to Start L10:** L09 deck, Slide 11 (analysis of Dijkstra's Algorithm)
   (c) Modify the lecture notes to change $r \mapsto c$, to match Slide 05 (inequalities).
   (d) Announce mid-term exam coverage, per the February 5 instructor meeting.

10 Lecture 10 - Graph algorithms V - Minimum Spanning Trees

10.1 Slide 03

1. At each step, select the minimum weight edge which does not create a cycle.

10.2 Slide 04

1. Kruskal's algorithm is another greedy algorithm (as described above).

10.3 Slide 05

1. # con. comp. means the number of connected components.
2. explanation for why $\#vertices - \#con.comp. \leq \#edges$:
   (a) As usual, suppose that our graph has n vertices and m edges.
   (b) Suppose that our graph has $r \geq 1$ connected components.
   (c) Let the i^{th} connected component have n_i vertices and m_i edges.
   (d) Hence $n = \sum_{i=1}^{r} n_i$ and $m = \sum_{i=1}^{r} m_i$ edges.
   (e) Because the i^{th} connected component is connected, therefore $m_i \geq n_i - 1$, for all $1 \leq i \leq r$.
   (f) Then we compute

$$\begin{aligned} & \#vertices - \#con.comp. \\ = \; & n - r \\ = \; & \left(\sum_{i=1}^{r} n_i \right) - r \\ = \; & \sum_{i=1}^{r} (n_i - 1) \\ \leq \; & \sum_{i=1}^{r} m_i \\ = \; & m \\ = \; & \#edges, \end{aligned}$$

   as claimed.

10.4 Slide 06

1. We never demand that all weights are distinct. Therefore here we need to replace some $<$ with $\leq$, so that we will actually cover **all** possible cases for where $w(e)$ can fall within $w(e_1), \ldots, w(e_r]$).

10.5 Slide 07

1. I suggest to change the notation here to replace T by B throughout, for better symmetry with the A notation already in use.
2. Because e is not in A, we know that $(V, A + e)$ must contain a cycle.

3. Because T is a spanning tree, therefore removing e from T splits T into two connected components, T_1, T_2. (Include Armin's diagram showing this situation here, somehow.)
4. Explanation for why e' is in A: e' is some edge in the constructed cycle, in $A + e$. By construction e' cannot equal e. Therefore e' is one of the other edges of the cycle, all of which are in A.
5. Explanation for why e' is not in T: Recall that T_1, T_2 were both connected components. $e \in T$ goes from T_1 to T_2. e' goes from T_2 back to T_1. All of this shows that if e' is in T, then T contains a cycle (namely the cycle found above). Becuase T is a (spanning) tree, therefore this cannot happen.

10.6 Slide 08

1. All other edges in A have weights $\leq w(e)$.
2. Meaning of "keep going": After finitely many steps, we will turn T into A. Since $\leq$ holds at each step, we get what we want by the end.

10.7 Slide 09

1. CLRS has a section (19.1) on the `Disjoint-Set` data structure.
   (a) The operations on this data structure which we need here are `Union, Find`.
2. We do NOT cover this data structure in CS 240. We should!

10.8 Slide 10

1. This slide treats `Disjoint-Set` as an ADT (i.e. no implementation details yet).
2. On line 5,
   (a) $e_k.1$ is the first vertex defining the edge, and $e_k.2$ is the second vertex.
   (b) The `if` statement is meant to determine whether the selected edge connects two disjoint sets together.
3. *U.Union* replaces two distinct sets by their union.

10.9 Slide 12

1. This slide starts to discuss the data structure.
2. `Find` is constant time: $O(1)$. This uses the array X from Slide 11. With this X, it is clear that we can carry out `Find` in constant time.
3. Union takes more time: traverse the first list before we know where to append the second: worst case $O(n)$.
4. Kruskal
   (a) Sort once $O(m \log m)$ (need a general sort because we don't yet have additional information about how our nodes are organized)
   (b) Each find is constant; $2m$ vertices in total: $O(m)$.
5. Reason for this modified approach: efficiency!
6. Colours indicate disjoint sets; when we merge into a set, we make the colours the same.
7. Motivation for Slide 13: beat the n^2 that appeared at the bottom of Slide 12: make it $n \log n$ instead, if possible.

10.10 Slide 13

1. Maintain additional information to make the algorithm more efficient, as described on the slide:
   (a) size of each set
   (b) pointer to the tail of each list
2. Total cost of union per vertex: $O(\log n)$, as near the bottom of Slide 13.
3. Therefore the total cost of all unions: $O(n \log n)$, as at the bottom of Slide 13.

10.11 Notes and Tasks from the Lecture

1. Notes
   (a) **Where to Start L11:** You found it on the fly.
2. Tasks
   (a) Expand the Background Information Document, to include more details that would otherwise need to be looked up in the CS 240 course materials.

11 Lecture 11 - Greedy algorithms I

11.1 Slide 04

1. This is an example of a Greedy Algorithm, that we have already seen in CS 240.

11.2 Slide 08

1. Explanation for why this algorithm lies in $O(n \log n)$:
   (a) Sorting a list of size n (e.g. using MergeSort) is in $O(n \log n)$.
   (b) The for-loop is clearly in $O(n)$. Because of the pre-sorting, to check for an overlap, it suffices to compare the candidate's start time against the last finish time included. This can be done in constant time.
   (c) So by the max-rule, we get $O(n \log n) + O(n) = O(n \log n)$.

11.3 Slide 09

1. Since O is assumed to be optimal, therefore $m \geq k$.
2. Hence we need to rule out $m > k$, to establish $m = k$.
3. See Slide 11 for the confirmation of the above fact: Towards a contradiction, we assume that $|S| < |O|$. Because $|S| = k$ and $|O| = m$, this is the same as assuming that $k < m$.

11.4 Slide 10

1. The notation here is inconsistent with earlier. But there is no way to avoid this, given that we are using is for S, and js for O.
2. The induction is on r.
3. Why The Base Case ($r = 1$) Holds: Want: $f(i_1) \leq f(j_1)$, where i_1 is the first interval in the constructed S, and j_1 is the first interval in the optimal O. Because of the sorting carried out during the construction of S, $f(i_1)$ is guaranteed to the the earliest possible finish time for any provided interval.

11.5 Slide 13

1. Time permitting, include the failed attempts from Éric's slides.
2. If you can, change ... to $\cdots$.

11.6 Slide 14

1. Explanation for s_ℓ is a time contained in k intervals: We just argued that s_ℓ is contained in all of the intervals $[s_{i_1}, f_{i_1}], \ldots, [s_{i_{k-1}}, f_{i_{k-1}}]$ (note, there are $k-1$ of them). By construction, s_ℓ is also contained in $[s_\ell, f_\ell]$.

11.7 Slide 16

Detailed Computation for $T(L') - T(L)$:

$$
\begin{aligned}
& T(L') - T(L) \\
= \; & nt(e_1) + (n-1)t(e_2) + \cdots + (n-i+1)t(e_{i+1}) + (n-i)t(e_i) + \cdots + 2t(e_{n-1}) + t(e_n) \\
& - [nt(e_1) + (n-1)t(e_2) + \cdots + (n-i+1)t(e_i) + (n-i)t(e_{i+1}) + \cdots + 2t(e_{n-1}) + t(e_n)] \\
= \; & t(e_{i+1}) - t(e_i) \\
< \; & 0,
\end{aligned}
$$

because $t(e_{i+1}) < t(e_i)$.

11.8 Notes and Tasks from the Lecture

1. Notes
   (a) Stuff.
2. Tasks
   (a) Indicate to which slide deck you are referring when you mention slide numbers.
   (b) **Where to Start L12:** L11 deck, Slide 10, Proof of the Lemma.

12 Lecture 12 - Snow Day

12.1 Global

The university campus was closed on Thursday, February 13, when we were supposed to deliver Lecture 12.

13 Lecture 13 - Greedy algorithms II

13.1 Global

1. The slide deck for this lecture is Éric's Lecture 06 deck, from F24.
2. If Collin can obtain the source for this deck, then he will recompile it to match the current W25 offering, and correct typos.

13.2 Slide 06

1. Attempt #2
   (a)

$$\begin{aligned} \ell(1) &= 0 \\ \ell(2) &= 5 \end{aligned}$$

   The smaller slack is first; this choice is not optimal.
   (b)

$$\begin{aligned} \ell(1) &= 0 \\ \ell(2) &= 0 \end{aligned}$$

   The larger slack is first; this choice is optimal.

13.3 Slide 07

1. Observation, Explained:
   (a) Suppose $d(i) = d(j)$.
   (b) Claim: The two orderings $[1, \ldots, i, j, \ldots, n], [1, \ldots, j, i, \ldots, n]$ have the same max-lateness.
   (c) "Proof": The only difference is i, j versus j, i. So focus on the lateness contributed by these two pairs.

(d) Look at the diagram. The black vertical line is the common deadline.

(e) The finish time for the pair is always the same: $f_i + f_j = f_j + f_i$.

2. Next Bullet: Since i, j were arbitrary, therefore we can conclude that all orderings in non-decreasing deadline order have the same max-lateness.

13.4 Slide 08

1. Checking the Argument:
   (a) Suppose that L is **not** non-decreasing with respect to deadlines. With this assumption, we want to show that $max-lateness(L) \geq max - lateness(L_{greedy})$.
   (b) By definition, there exists $1 \leq i \leq n$ such that $d(e_i) \geq d(e_{i+1})$.
   (c) Let L' be the permutation $[1, \ldots, e_{i+1}, e_i, \ldots, n]$.
   (d) Consider $max - lateness(L')$.
   (e) The new lateness of e_{i+1} cannot increase compared to L: we now do it earlier than before. So it is at most $max - lateness(L)$.
   (f) The new lateness of e_i is at most the old lateness of e_{i+1}. So it is at most $max - lateness(L)$.
   (g) Nothing else changes. Hence $max-lateness(L') \leq max-lateness(L)$.
   (h) L' removes one inversion that was present in L. $(d(e_i)) > d(e_{i+1}) \Leftrightarrow (d(e_{i+1})) < d(e_i)$.
   (i) Keep going. What is the maximum number of swaps needed to correctly sort the list? The worst case is when the list is sorted in the reverse of the needed, greedy order.

1.		$n-1$	swaps
2.		$n-2$	swaps
$\vdots$		$\vdots$	
$n-1$.		1	swap
$\sum_{i=1}^{n-1}$	$=$	$\frac{n(n-1)}{2}$	swaps

13.5 Slide 10 (Fractional Knapsack Problem)

1. Remark: This is **dynamic programming**, which is the topic of our next unit.

13.6 Slide 11

1. Trivial Solution: if $\sum_i w_i < w$ (all $e_i = 1$).
2. So for a non-trivial solution, assume $\sum_i w_i \geq w$.
3. Observation: Yes, under the assumption $\sum_i w_i \geq w$ which we just made.
4. Consequence: Yes, under the assumption $\sum_i e_i w_i \leq w$ from the definition.

13.7 Slide 12

1. Attempt 1 Example setup ($W = 50$)

i	1	2	3
w_i	10	30	20
v_i	60	90	100

   decreasing order of v_i : $3, 2, 1$. Right: $20 + 30 = 50$, full already, no room left for #1.
2. Attempt 2 Example setup ($W = 10$)

i	1	2
w_i	10	5
v_i	100	1

   (a) w_2, w_1 gives: $e_2 = 1, e_1 = \frac{1}{2}$, total value $= \left(\frac{1}{2}\right) 100 + (1)1 = 51$.
   (b) w_1, w_2 gives: $e_1 = 100, e_2 = 0$, total value $= (1)100 + (0)1 = 100$.
3. Attempt 3 non-increasing "value per kilo" $\frac{v_i}{w_i}$.
   (a) Confirming that the first example yields $[6, 3, 5]$, again with $w = 50$

i	1	2	3
w_i	10	30	20
v_i	60	90	100
$\frac{v_i}{w_i}$	6	3	5

   This gives the order $1, 3, 2$. This gets $\left[1, 1, \frac{2}{3}\right]$, for a value $(1)60 + (1)90 + \left(\frac{2}{3}\right) 100 = 150 + \frac{200}{3} = \frac{450+200}{3} = \frac{650}{3} = 216\frac{2}{3} > 190$.
   (b) Confirming that the second example yields $\left[10, \frac{1}{5}\right]$, again with $w = 10$

i	1	2
w_i	10	5
v_i	100	1
$\frac{v_i}{w_i}$	10	$\frac{1}{5}$

This gives the order $1, 2$. This gets $[1, 0]$, for a value $(1)100+(0)1 = 100$. This agrees with my earlier guess at the optimal choice.

13.8 Slide 13

1. Runtime: $O(n \log n)$. The pre-sort is in $O(n \log n)$; the rest is clearly in $O(n)$.

13.9 Slide 14

1. Assumption, not stated but should be: We have pre-sorted by non-decreasing $\frac{v_i}{w_i}$.
2. Reason for $s_{i_j} \le s_\ell$: the pre-sorting that is done at the start of the algorithm.
3. Change "because their weights are the same" to "because $\sum e_i w_i = w = \sum s_i w_i$".
4. Problem: We are using i in two different ways here. To correct this, choose i, j as described; index by k instead of i afterwards.
5. Question: Why is $\sum s'_i w_i = w$?
6. Add this definition of s'_k, for all $1 \le s'_k \le n$ to the slides.

$$s'_k = \begin{cases} s_i + \frac{\alpha}{w_i} & \text{if} \quad k = i \\ s_j - \frac{\alpha}{w_j} & \text{if} \quad k = j \\ s_k & \text{otherwise} \end{cases}$$

7. Claim: $\sum_k s_{k'} w_k = w$. (Have $\sum_k s_k w_k = w$)
8. Proof:

$$\begin{aligned} & \sum_k s_{k'} w_k \\ = & \sum_k s_k w_k + \left(\frac{\alpha}{w_i}\right) w_i - \left(\frac{\alpha}{w_j}\right) w_j, \text{ by earlier definition of } s_{k'} \\ = & \sum_k s_k w_k + \alpha - \alpha \\ = & \sum_k s_k w_k, \text{ by earlier assumption with corrected notation} \\ = & \ w, \end{aligned}$$

as claimed.

9. Claim: $value(S') \geq value(S)$.
10. Proof:

$$\begin{aligned} value(S') &= \sum_k s_k v_k \\ value(S) &= \sum_k s_{k'} v_k \\ &= \sum_k s_k v_k + \left(\frac{\alpha}{w_i}\right) v_i - \left(\frac{\alpha}{w_j}\right) v_j \\ &= \sum_k s_k v_k + \alpha \left(\frac{v_i}{w_i} - \frac{v_j}{w_j}\right) \\ &= value(S) + +\alpha \left(\frac{v_i}{w_i} - \frac{v_j}{w_j}\right). \end{aligned}$$

    Recall that we pre-sorted by non-increasing $\frac{v_i}{w_i}$, and $j > i$. Hence $\left(\frac{v_i}{w_i} - \frac{v_j}{w_j}\right) \geq 0$. By assumption $\alpha > 0$. This shows that $\alpha \left(\frac{v_i}{w_i} - \frac{v_j}{w_j}\right) \geq 0$. So finally, this shows that $value(S') \geq value(S) \geq 0$, as claimed.
11. Change "choose the first α such that" to "choose the smallest α such that either $s'_i = e_i$ or $s'_j = e_j$ or both".
12. Change "one more common entry with" to "at least one more common entry with".
13. After finitely many steps (the list is finitely long, hence can have at most finitely many differences), it is guaranteed to equal E.

13.10 Slide 15

1. Include the example of computing finish times, from Éric's L05 slide deck.

13.11 Notes and Tasks from the Lecture

1. Notes
   (a) If possible, correct the typo on Éric's L06 slide deck, Slide 06: red $t(1) \mapsto t(2)$.
   (b) If possible, fix the colours too.
2. Tasks

(a) Announce my modified office hours for Feb 27 and 28: Feb 27 hours are cancelled; Feb 28 hours will run from 14:00 to 16:00.

14 Lecture 14 - Dynamic Programming I

14.1 Global

1. This lecture uses Armin's Lecture 11 slide deck?

14.2 Slide 03

1. Explanation for $T(n) = F(n+1) - 1$:
   (a) Proof by (strong) induction on $n \geq 0$.
   (b) Base ($n = 0$):
      i. $T(0) = 0$.
      ii. $F(0+1) - 1 = F(1) - 1 = 1 - 1 = 0$✓
   (c) Base ($n = 1$):
      i. $T(1) = 0$.
      ii. $F(1+1) - 1 = F(2) - 1 = 1 - 1 = 0$✓
   (d) Induction ($n > 1$):
      i. I.H. $T(n-1) = F(n) - 1$ and $T(n-2) = F(n-2) - 1$.
      ii.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\underbrace{=}_{I.H.} [F(n) - 1] + [F(n-1) - 1] + 1 \\ &= F(n) + F(n-1) - 1 \\ &\underbrace{=}_{Fibonacci\ definition} F(n+1) - 1. \end{aligned}$$

2. Explanation for $T(n) \in \Theta(\varphi^n)$, where $\varphi = \frac{1+\sqrt{5}}{2}$, the **Golden Ratio**: Refer to Background Information document.

14.3 Slide 10

1. all indices $< n \mapsto$ all indices $t < n$.

14.4 Slide 11

1. increasing end time $\mapsto$ non-decreasing end time.

14.5 Slide 13

1. Definition of $M[j]$: from the two cases mentioned earlier:
   (a) where we exclude interval j, and
   (b) where we include interval j: w_j is from including interval j, and $M[p_j]$ is from all intervals that don't overlap with interval j.
2. Exercise: recover the optimum set, not only $M[n]$, for extra $\Theta(n)$.
   (a) I think we just need to add an indicator array of size n, and indicate in that array for each interval j, whether we have included interval j or not, as we go through the main procedure.
   (b) Then at the end, make one pass through the array to list off which intervals we included.
   (c) Check all of this with Mark, when time permits.

14.6 Slide 14

1. $$S \subset \{1, \ldots, n\} \mapsto S \subseteq \{1, \ldots, n\}.$$
2. While the above is mathematically more correct, the problem will be trivial if we can include everything!

14.7 Slide 15

1. we choose item n or not $\mapsto$ we include item n, or we don't
2. "choose" $\mapsto$ "include" through the rest of the bullets also.
3. Indent the list of two items, of which we take the max.

14.8 Slide 16

1. The array O is two-dimensional!
2. Explanation for why the run time is in $\Theta(nW)$:
   (a) The outer loop runs n times.
   (b) The inner loop runs W times.
   (c) The work inside the inner loop is all in $\Theta(1)$.

14.9 Notes and Tasks from the Lecture

1. Notes
   (a) Stuff.
2. Tasks
   (a) Stuff.

15 Lecture 15 - Dynamic Programming II

15.1 Global

1. This lecture uses Armin's Lecture 12 slide deck.

15.2 Slide 2

1. The subsequence does not need to be contiguous.
2. There are 2^n subsequences from a sequence of length n (each entry is included, or not).

15.3 Slide 3

1. We want to solve this problem using dynamic programming. Hence we first need to decide which subproblems we can solve, to make it possible to solve the main problem.
2. Attempt #2: The example provided us shows us the difficulty with this attempt, in the case where $i = 6$ and we cannot add $A[7]$ to form a longer increasing subsequence.

15.4 Slide 4

1. `cat` here means **concatenation**.
2. In the provided example (sequence $[7, 1, 3, 10, 11, 5, 19]$), we get

i	$L[i]$	increasing subsequence witnessing $L[i]$
1	1	$[7]$
2	1	$[1]$
3	2	$[1, 3]$
4	3	$[1, 3, 10]$
5	4	$[1, 3, 10, 11]$
6	3	$[1, 3, 5]$
7	5	$[1, 3, 10, 11, 19]$

15.5 Slide 5

1. We should state up front that the array L has entries $1, \ldots, n$.
2. **Explanation for the Algorithm:**
   (a) Line 3: At a minimum, each entry constitutes an increasing sequence by itself.
   (b) Lines 5-6: If $A[j] < A[i]$, then $A[j]$ can fit into an increasing subsequence ending with $A[i]$ - revise $L[i]$ if appropriate (i.e. when $L[j] + 1$ is at least as big as $L[i]$).

15.6 Notes and Tasks from the Lecture

1. Notes
   (a) Slide 2: The subsequence must actually be strictly increasing, even though some entries in the provided sequence can be equal.
   (b) Question: What run-time would result from brute force, without dynamic programming, for the longest increasing subsequence problem? Would it be worse than n^2?
   (c) Slide 6: **Remark:** This example makes $m = n$, which is NOT TRUE IN GENERAL.
   Remark: This length 0 is possible, as is confirmed on Slide 7.
2. Tasks
   (a) Include in some form Éric's improved algorithm for the longest increasing subsequence problem, which achieves $O(n \log n)$.

16 Lecture 16 - Dynamic programming III

16.1 Global

1. This lecture uses Armin's Lecture 13 slide deck.

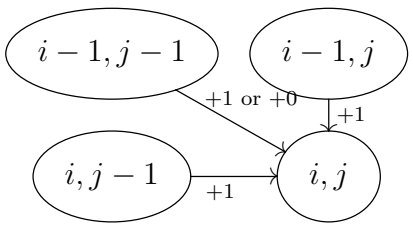
Outline:

1. Edit Distance
2. Optimal Binary Search Trees
3. Maximum Independent Sets In Trees

16.2 Edit Distance

1. Assume we want to design a spell checker.
2. We need to find a way to deal with mis-spelling.
3. So given an input word (string), we want to be able to find "close" words.
4. It is up to us to define closeness ear.
5. When do mis-spellings occur?
   (a) A character is typed in error.
   (b) A character is omitted during typing.
   (c) An extra character is typed.
6. One way to measure closeness to other words is to try to align the input with other words.
7. We try to see to what extent we can align two words.
8. In other words, try to write one word on top of another.
9. Let's be more formal.
   (a) Our inputs are $A[1 \dots n], B[1 \dots m]$.
10. Example with $A = snowy, B = sunny$, as in the slides.
11. **Remark:** This example should be improved to demonstrate that $m = n$ will not hold in general.
12. The character _ is called a **gap** and can be used as many times as needed.
13. Count the number of operations $\{add, delete, change\}$ which are needed to turn A into B.
14. An **alignment** of words A, B writes A on top of B, and shows what changes are needed to turn A into B.
15. The **cost** of an **alignment** is the number of columns in which the letters are not the same.

16. The **edit distance** is the cost of the best possible alignment.
17. We want to compute the edit distance between $A[1 \dots n]$ and $B[1 \dots m]$.
18. **Goal:** Compute the edit distance between A and B.
19. Define an array D of size $(n+1) \times (m+1)$, where for any $i \leq n, j \leq m$, $D[i,j]$ is the edit distance between $A[1 \dots i]$ and $B[1 \dots j]$.
20. Initialization:
    (a) The edit distance between an empty word and $A[1 \dots i] = i$, so $D[i,0] = i$ for all i.
    (b) The edit distance between an empty word and $B[1 \dots j] = j$, so $D[0,j] = j$ for all j.
21. Three cases may happen.
    (a) Write $A[i]$ on top of $B[j]$.
    $$\begin{array}{cc} \cdots & A_{i-1} \quad A_i \\ \cdots & B_{j-1} \quad B_j \end{array}$$
    This last column will
    i. add 1 to the edit distance, if $A[i] \neq B[j]$, and
    ii. add 0 to the edit distance, if $A[i] = B[j]$.
    The edit distance of the remaining columns will equal the edit distance between $A[1 \dots i-1]$ and $B[1 \dots j-1]$.
    (b) Write $A[i]$ on top of _.
    $$\begin{array}{cc} \cdots & A_{i-1} \quad A_i \\ \cdots & B_j \quad \_ \end{array}$$
    In this case, the edit distance will equal 1 plus the edit distance between $A[1 \dots i-1]$ and $B[1 \dots j]$.
    (c) Write _ on top of $B[j]$.
    $$\begin{array}{cc} \cdots & A_i \quad \_ \\ \cdots & B_{j-1} \quad B_j \end{array}$$
    In this case, the edit distance will equal 1 plus the edit distance between $A[1 \dots i]$ and $B[1 \dots j-1]$.
22. This can be generalized to other indices.
23. Computing any $D[i,j]$:



Take the minimum of the three possibilities.

24. The desired answer is then $D[n, m]$.
25. Example from DVP

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

16.3 Optimal Binary Search Trees

1. In CS 240, you have seen re-ordering items in linked lists and arrays.
2. The move-to-front heuristic suggests to (ad-hoc) bring the last searched (or inserted) item to the front of the list.
3. We also saw that, if we know probabilities of enquiries of items in advance, then we can find an optimal ordering which gives the best expected cost of accessing entries.
4. What we did for linked lists can be done if we have a BST.
5. Idea: Keep the items with higher search probabilities in places in the BST which are easier to access.
6. So assume he have n items, say numbers, or comparable other objects. Each of these items has a probability of access:

i	1	2	$\cdots$	n
p_i	p_1	p_2	$\cdots$	p_n

   with $\sum_i p_i = 1$.
7. **Goal:** Construct a BST with the least expected cost of access.
8. The cost of accessing a node at depth i is $i + 1$.

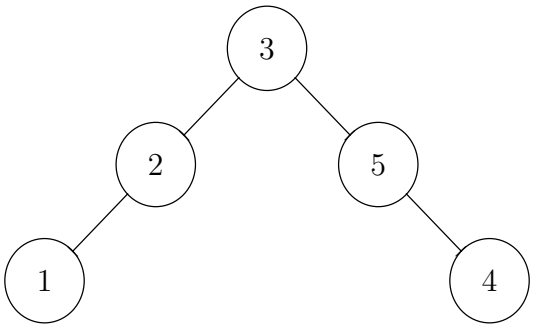
9. Hence the expected costs of access is

$$\sum_{i=1}^{n} p_i(depth(i)+1)$$

10. **Example:**

i	1	2	3	4	5
p_i	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$

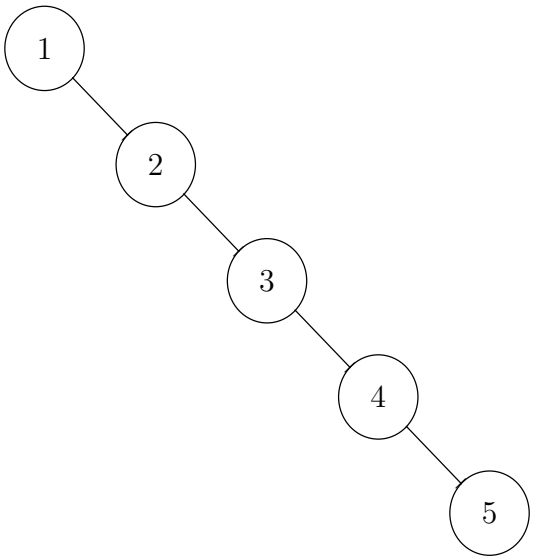
(a)



Expected Cost:

$$1 \cdot \left(\frac{1}{5}\right) + 2 \cdot 2 \cdot \left(\frac{1}{5}\right) + 2 \cdot 3 \cdot \left(\frac{1}{5}\right) = \frac{11}{5}.$$

(b)



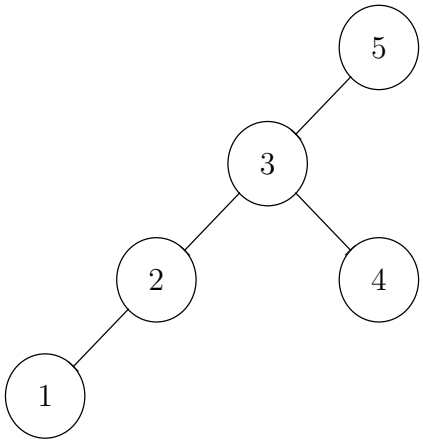
Expected Cost:

$$\frac{1}{5}(1+2+3+4+5) = \frac{15}{5} = 3.$$

11. Observation: In the linked list example, we saw that a greedy strategy worked.
12. **Example to Demonstrate that a Greedy Strategy Does Not Work:**

i	1	2	3	4	5
p_i	0.1	0.2	0.25	0.05	0.4

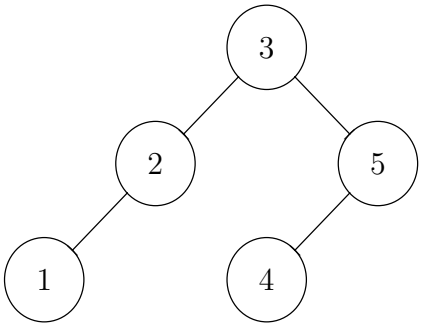
(a) Greedy: Put the key with the highest probability at the root.



Expected Cost:

$$(0.4) \cdot 1 + (0.25) \cdot 2 + (0.2 + 0.05) \cdot 3 + (0.1) \cdot 4 = 2.05$$

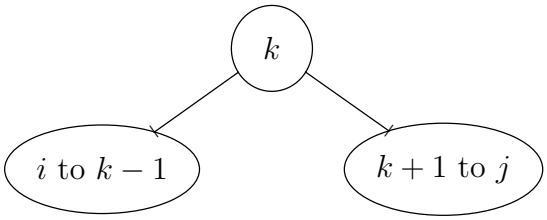
(b) Better Than Greedy:



Expected Cost:

$$(0.25) \cdot 1 + (0.2 + 0.4) \cdot 2 + (0.1 + 0.05) \cdot 3 = 1.9$$

13. Define $M[i, j]$ to be the minimal cost for items i to j.
14. Having already fixed $1 \leq i \leq j \leq n$, take an arbitrary $i \leq k \leq j$, and put item k at the root.



(a) Cost of left subtree:

$$M[i, k-1] + \sum_{\ell=i}^{\ell=k-1} p_\ell$$

Explanation for the second term: All items are shifted down by one level, so the cost of access increases by one in each case.

(b) Cost of right subtree (similar):

$$M[k+1, j] + \sum_{\ell=k+1}^{\ell=j} p_\ell$$

(c) Cost of root:

$$p_k$$

15. Hence the total cost for the choice k is

$$\begin{aligned} & M[i, k-1] + \sum_{\ell=i}^{\ell=k-1} p_\ell + M[k+1, j] + \sum_{\ell=k+1}^{\ell=j} p_\ell + p_k \\ = \quad & M[i, k-1] + M[k+1, j] + \sum_{\ell=1}^{\ell=j} p_\ell. \end{aligned}$$

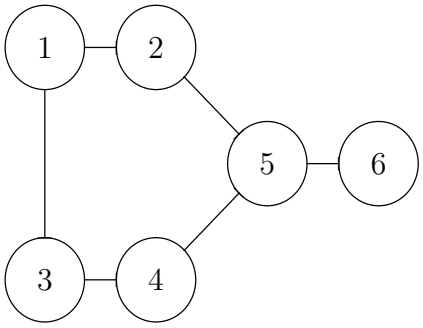
16. The above formula is for any choice of k. Now we have to choose k so that it gives minimal value. Try for all k. This gives

$$M[i, j] = \min_{i \leq k \leq j} \left(M[i, k-1] + M[k+1, j] \right) + \sum_{\ell=1}^{\ell=j} p_\ell.$$

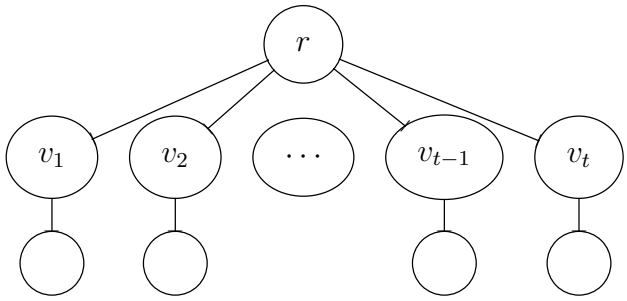
17. We also define $M[i, j] = 0$ for $j < i$.
18. To compute $\sum_{\ell=1}^{\ell=j} p_\ell$, we define
    $S[\ell] = \sum_{t=1}^{t=\ell} p_t$, so that
    $\sum_{\ell=i}^{\ell=j} p_\ell = S[j] - S[i-1]$, with $S[0] = 0$.
19. We can compute all S in $O(n^2)$.

16.4 Maximum Independent Sets In Trees

1. An **independent set** of a graph $G = (V, E)$ is $S \subseteq V$ such that $u, v \in S$ implies $(u, v) \notin E$.
2. The **maximum independent set** problem is to find an independent set of maximum cardinality.
3. **Example:**



   The maximum independent set in this example: $\{2, 3, 6\}$.
4. It is believed that the maximum independent set problem is intractable.
5. However, if our graph is a tree, then we can use dynamic programming. But what sub-problems do we want?
6. Given a tree,



   we have two cases, for whether the root r belongs to the maximum independent set, S, or not.

(a) $\underline{r \in S}$:
   i. Since $r \in S$, therefore no child of r can be in S.
   ii. Hence the remainder of S must come from the subtrees rooted at the grandchildren of r.

(b) $\underline{r \notin S}$:
   i. Since $r \notin S$, therefore all of the elements of S are from subtrees rooted at the children of r.

7. Based on these observations, the subproblems should be based on subtrees.
8. Define $I(v)$ = size of the largest independent set in the subtree rooted at v.
9. We are done if we can compute $I(r)$, where r is the root.
10. Based on the above argument for the root, which holds for any subtree, we have the following recurrence relation:

$$I(v) = \max\left\{1 + \sum_{u \text{ a grandchild of } v} I(u), \sum_{u \text{ a child of } v} I(u)\right\}.$$

Remark: The first option comes from the case where v is in the maximum independent subset; the second option comes from the case where v is not in the maximum independent subset.

11. Using the above recurrence relation, one can solve the problem in $O(|V|)$ time. See the notes referenced below for the details; pull those details in here.
12. Note that there are only n subproblems to solve: one per vertex.

16.5 Notes and Tasks from the Lecture

1. $\underline{\text{Notes}}$
   (a) In the third edit distance example, check with Armin that two changes are needed, and not one, correct?
   (b) For Maximum Independent Sets in Trees, is the runtime in $O(|V|)$, instead of in $O(|V| + |E|)$, as in Armin's hand-written notes?
   $\underline{\text{Armin's answer:}}$ I think it should be just $O(|V|)$. This is a tree so $|E| = |V| - 1$. The Top-Down implementation is discussed in Lap Chi's notes: `https://cs.uwaterloo.ca/~lapchi/cs341/notes/L13.pdf` For a bottom-up implementation, we need to have references to parents as well.

2. Tasks
   (a) Make your own slides for this lecture. Include all details from these notes there instead.

17 Lecture 17 - Dynamic programming IV

17.1 Global

1. This lecture uses Armin's Lecture 14 slide deck.

17.2 Bellman-Ford Algorithm

1. We have seen Dijkstra's algorithm, which solves the single-source shorted path problem, when the input has no negative weights.
2. Dijkstra's algorithm used a greedy strategy.
3. Here we look at another algorithm, which is one of the earliest algorithms which used the dynamic programming approach.
4. The input may have negative weights, and the algorithm can detect negative cycles.
5. We have already seen that distance is well-defined only if there are no negative cyles.
6. If there are no negative cycles, then Bellman-Ford finds all distances $\delta(s, v)$, for a source s and a vertex v.
7. The good news about this algorithm is that it can detect negative cycles.
8. All these good properties come with a price.
9. The cost is worse than Dijkstra.
10. Simple informal observation A path $s \rightsquigarrow v$ contains at most $n - 1$ edges, unless it contains a cycle. If a path contains $\geq n$ edges, then by pigeonhole principle, a vertex appears more than once, hence there exists a cycle.
11. Assume there are no negative cycles.
12. **Recall:** $\delta(s, v)$ is the length of a shortest path $s \rightsquigarrow v$, as before.
13. **Definition:** For all $0 \leq i \leq n - 1$, $\delta_i(s, v)$ is the length of a shortest path $s \rightsquigarrow v$ having $\leq i$ edges. If no such path exists, then $\delta_i(s, v) = \infty$.
14. **Observations (assuming there are no negative cycles):**
    (a) $\delta_0(s, s) = 0$.

(b) $\delta_0(s, v) = \infty$, for all $v \neq s$.
(c) Since there are no negative cycles, therefore $\delta_{n-1}(s, v) = \delta(s, v)$ (shortest paths are simple).
(d) $\underline{\delta(s, v) \leq \delta_i(s, v), \text{ for all } 0 \leq i \leq n-1 \text{ and for all } v}$
   i. If $\delta_i(s, v) = \infty$, then it is obvious.
   ii. If $\delta_i(s, v) < \infty$, then there exists a path $s \rightsquigarrow v$. So $\delta(s, v)$ is either the length of that path, or the length of a shorter path. So $\delta(s, v) \leq \delta_i(s, v)$.
(e) $\underline{\delta_{n-1}(s, v) = \delta(s, v)}$
   i. If $\delta(s, v) = \infty$, then it is obvious.
   ii. If $\delta(s, v) < \infty$, then there exists a path $s \rightsquigarrow v$. So $\delta(s, v)$ is the weight of a shortest path, P, $s \rightsquigarrow v$. We claim that this path has at most $n-1$ edges. If not, then the path has $\geq n$ edges. This implies (by pigeonhole principle) that the path contains a cycle. By our assumption, this cycle cannot be negative. The presence of this positive cycle contradicts the fact that $\delta(s, v)$ is the weight of a shortest path (remove the positive cycle to produce a path which is strictly shorter). Since P has $\leq n-1$ edges, therefore $\delta_{n-1}(s, v)$ is the length of P, which by definition of δ equals $\delta(s, v)$.

15. Now let's find a recurrence relation.
16. Assume $\delta_i(s, v) < \infty$.
17. Then $\delta_i(s, v)$ is the length of a path with $\leq i$ edges, which has a finite weight.
18. The path either has exactly i edges, or $\leq i-1$ edges.
   (a) If it has $\leq i-1$ edges, then we use $\delta_{i-1}(s, v)$.
   (b) Otherwise (i.e. length equals i), the path can be decomposed into a sub-path $s \rightsquigarrow u$, having $i-1$ edges, and one edge (u, v).
   (c) Hence $\delta_i(s, v) = \delta_{i-1}(s, u) + w(u, v)$.
19. For finding the shortest path, we must consider all edges (u, v), as the path may pass through any neighbour u of v.
20. Hence we have

$$\delta_i(s, v) = \min \left\{ \delta_{i-1}(s, v), \min_{(u,v) \in E} \{\delta_{i-1}(s, u) + w(u, v)\} \right\}.$$

21. Define $d[v], d_i[v]$ as in the algorithms from the slides.
22. Explanation for line 3 of the improved algorithm: Bellman-Ford is basically $n-1$ rounds of relaxation. The counter i makes sure we have

done $n-1$ rounds of relaxation.

23. **We have already seen the correctness of the first version, i.e. Dijkstra (assuming no negative cycle is present)**:

$$\begin{aligned} d_i[v] &= \delta_i(s,v) \text{ for all } 0 \le i \le n-1, \text{ since this implies} \\ d_{n-1}[v] &= \delta_{n-1}(s,v) \\ &= \delta(s,v), \text{ by the last observation above.} \end{aligned}$$

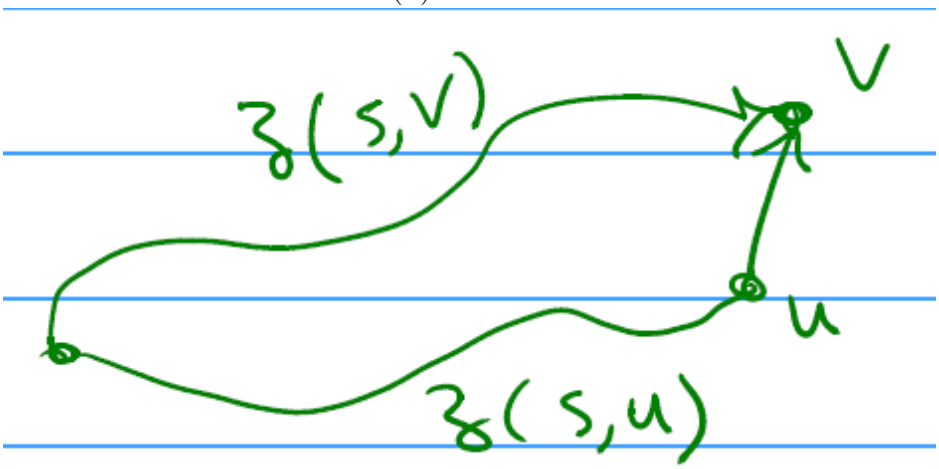
24. **Remark:** We have to find edges (u,v) for the given vertex v, which is time consuming.
25. **Saving Time and Space:** We can use a single array d instead of n of them. We can also modify the algorithm to go with edges not toward specific vertices, v.
26. **Recall:** Lines 5 to 7 of the improved algorithm are called **relaxation**.
27. **Correctness (Improved version), Part 1 - Claim:** For all $0 \le i \le n-1$ and for all v, after iteration i, $d[v] \le d_i[v]$. Proof by induction on i:
    (a) Base ($i = 0$):
        i. Clear from the initialization of the d_0 and d arrays.
    (b) Induction ($i > 0$):
        i. I.H.: Suppose that the statement is true up to $i-1$.
        ii. At the beginning of the i^{th} iteration, by the I.H. we have $d[v] \le d_{i-1}[v]$.
        iii. $d[v]$ can only decrease, so $d[v] \le d_{i-1}[v]$ remains true through the loop.
        iv. $d[v]$ is replaced by $\min\{d[v], \min_{(u,v)\in E}\{d[u]+w(u,v)\}\}$, where by the I.H. we have $d[u] \le d_{i-1}[u]$.
        v. So at the end of iteration i, by how the first algorithm updates d_i, we have $d[v] \le d_i[v]$. Explanation:
            A. In the original version, $d_i[v] \leftarrow d_{i-1}[v]$ to start.
            B. $d_i[v]$ is updated only if relaxation strictly improves path length.
            C. In this case, $d_i[v] \leftarrow d_{i-1}[u] + w(u,v)$. In the improved version, $d[v] \leftarrow d[u] + w(u,v)$. So the desired inequality must hold at the end of the i^{th} iteration.
28. **Claim:**
    (a) $d[v]$ can only decrease through a relaxation, and

(b) if $\delta(s,u) \leq d[u]$ and $\delta(s,v) \leq d[v]$ before relaxation, then $\delta(s,v) \leq d[v]$ after relaxation.

Proof of Claim:

(a) The first item is obvious from the shape of the algorithm.

(b)



(c) Before Relaxation:

$$\begin{aligned} \delta(s,v) &\leq \delta(s,u) + w(u,v), \text{ by triangle inequality, so that} \\ \delta(s,v) &\leq d[u] + w(u,v), \text{ because } \delta(s,u) \leq d[u] \text{ by assumption.} \end{aligned}$$

(d) After relaxation, $d[v]$ gets the value of

$$\min \left\{ \underbrace{d[v]}_{\geq \delta(s,v) \text{ before relaxation}}, \min_{(u,v)\in E} \left\{ \underbrace{d[u] + w(u,v)}_{\geq \delta(s,v) \text{ before relaxation}} \right\} \right\}$$

Hence $d[v] \geq \delta(s,v)$.

29. **Correctness (Improved version), Part 2 - Claim:** For all $0 \leq i \leq n-1$ and for all v, after iteration i, $\delta(s,v) \leq d[v] \leq \delta_i(s,v)$. Proof of Claim:
    (a) Correctness Part 1: $d[v] \leq d_i[v] \underbrace{=}_{\text{earlier Dijkstra observation}} \delta_i(s,v)$.
    (b) Previous Claim: $\delta(s,v) \leq d[v]$.
30. Why This Suffices To Establish Correctness: The above identity is for all i, in particular for $i = n-1$. So taking $\delta_{n-1}(s,v) = \delta(s,v)$. Hence $\delta(s,v) \leq d[v] \leq \delta(s,v)$, which implies $d[v] = \delta(s,v)$, for all v.

31. **Summary:**
    (a) If no negative cycle is present, then
        i. At the end of the algorithm, for all v we have $d[v] = \delta(s, v)$.
        ii. So by the triangle inequality, we obtain

$$\begin{aligned} \delta(s,v) &\leq \delta(s,u) + w(u,v), \text{ so that} \\ d[v] &\leq d[u] + w(u,v), \text{ for any edge } (u,v). \end{aligned}$$

        iii. This says that we can detect the presence of a negative cycle, by exhibiting a vertex v and an edge, $(u, v) \in E$ such that $d[v] > d[u] + w(u, v)$.
    (b) If there is a negative cycle, say $v_0, \ldots, v_k = v_0$ with $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$, then
        i. I claim that there exists an edge (v_{i-1}, v_i) with $d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)$. Proof:
            A. Towards a contradiction, suppose that for all i, $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$.
            B. Sum the inequality around the cycle:

$$\begin{aligned} & \sum_{i=1}^{k} d[v_i] \\ \leq & \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ \leq & \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

            C. Because $v_0 = v_k$, we have $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$.
            D. So subtracting these equal quantities from the above chain of inequalities, we obtain

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

            which is a contradiction.

17.3 Floyd-Warshall Algorithm

1. **Explanation for the Recurrence Relation on Slide 15:**

(a) Let v_j, v_k be any two vertices.
(b) To define $D_i(v_j, v_k)$, let P be a shortest path $v_j \rightsquigarrow v_k$ with all intermediate vertices in $\{v_1, \ldots, v_i\}$.
(c) We have these cases for whether $v_i \in P$ or not:
    i. $\underline{v_i \notin P}$: Take $D_i(v_j, v_k) = D_{i-1}(v_j, v_k)$.
    ii. $\underline{v_i \in P \text{ (only once, since } P \text{ is a shortest path)}}$: Partition P into $v_j \underbrace{\rightsquigarrow}_{P_1} v_i \underbrace{\rightsquigarrow}_{P_2} v_k$
        A. $P_1 : v_j \rightsquigarrow v_i$ (a shortest path, because P is)
        B. $P_2 : v_i \rightsquigarrow v_k$ (a shortest path, because P is)
    iii. Take $D_i(v_j, v_k) = D_{i-1}(v_j, v_i) + D_{i-1}(v_i, v_k)$.
(d) These are the two cases of the recursion.

18 Lecture 18 - Polynomial Time Reductions

18.1 Global

1. This lecture uses Armin's Lecture 15 slide deck.

18.2 Slide 03

1. The provided definition is equivalent to requiring
    (a) all "yes" answers for A to map to "yes" answers for B, and
    (b) all "no" answers for A to map to "no" answers for B.

18.3 Slide 04

1. Work through this (straightforward) exercise during the lecture.

18.4 Slide 05

1. Type up this proof.
2. Do not spend time on this proof in class.

18.5 Slide 06

1. Create a Clicker Question based on this.
2. Write up an explanation for the correct answer in these notes.

18.6 Slide 10

18.6.1 Proof that *Clique* $=_P$ *Independent Set*

Remark: This result is not even stated in Armin's slides. However it is worth proving, so we prove it here.
Definition: Given a graph $G = (V, E)$, its **complement** is $\overline{G} = (V, \overline{E})$ ($(u, v) \in \overline{E}$ if and only if $(u, v) \notin E$).
Lemma: $S \subseteq V$ is a clique in G (having $\geq k$ vertices) if any only if S is an independent set in $\overline{G}$ (having $\geq k$ vertices).

Proof.
1. Forward:
   (a) Assume that S is a clique in G (having $\geq k$ vertices).
   (b) I claim that S is an independent set in $\overline{G}$ (having $\geq k$ vertices).
   (c) Towards a contradiction, suppose that S is not an independent set in $\overline{G}$.
   (d) Then there is a pair of vertices $u, v \in S$, with $(u, v) \in \overline{E}$.
   (e) By definition of the graph complement, $(u, v) \notin E$.
   (f) This contradicts the fact that S is a clique in G.
2. Backward:
   (a) Assume that S is an independent set in $\overline{G}$ (having $\geq k$ vertices).
   (b) I claim that S is a clique in G (having $\geq k$ vertices).
   (c) Towards a contradiction, suppose that S is not a clique in G.
   (d) Then there is a pair of vertices $u, v \in S$, with $(u, v) \notin E$.
   (e) By definition of the graph complement, $(u, v) \in \overline{E}$.
   (f) This contradicts the fact that S is an independent set in $\overline{G}$.

□

1. Proof that *Clique* $\leq_P$ *Independent Set*
   (a) We must exhibit a polynomial time algorithm, F, to transform inputs for Clique into inputs for IS, preserving "yes" answers and "no" answers.
   (b) Let (G, k) be an arbitrary instance for Clique.
   (c) Return the instance $(\overline{G}, n - k)$ for Independent Set.
   (d) Clearly this can be done in polynomial time.
   (e) By the Lemma, this reduction is correct.
2. Proof that *Independent Set* $\leq_P$ *Clique*
   (a) We must exhibit a polynomial time algorithm, F, to transform inputs for IS into inputs for Clique, preserving "yes" answers and

"no" answers.

(b) Let (G, k) be an arbitrary instance for Independent Set.

(c) Return the instance $(\overline{G}, n - k)$ for Clique.

(d) Clearly this can be done in polynomial time.

(e) By the Lemma, this reduction is correct.

18.6.2 Proof that $VC =_P$ *Independent Set*

1. To see the connection between VC and Independent Set, we need this **Lemma:** S is a vertex cover in G if and only if $V \setminus S$ is an independent set in G.

   Proof. (a) Forward:
   i. Assume that S is a vertex cover in G.
   ii. We claim that $V \setminus S$ is an independent set in G.
   iii. Towards a contradiction, suppose that $V \setminus S$ is not an independent set.
   iv. Then there are some vertices $x, y \in V \setminus S$ such that $(x, y) \in E$.
   v. By the definition of a vertex cover, either $x \in S$ or $y \in S$.
   vi. But this contradicts the fact that $x, y \in V \setminus S$.
   vii. This shows that $V \setminus S$ is an independent set in G.

   (b) Backward:
   i. Assume that $V \setminus S$ is an independent set in G.
   ii. We claim that S is a vertex cover in G.
   iii. Towards a contradiction, suppose that S is not a vertex cover.
   iv. Then there is an edge $(x, y) \in E$ such that $x \notin S$ and $y \notin S$.
   v. In other words, $x, y \in V \setminus S$, and there is an edge between them.
   vi. This contradicts the fact that $V \setminus S$ is an independent set.
   vii. This shows that S is a vertex cover in G.

   □

2. Now we want to show that $VC \leq_P$ *Independent Set* and *Independent Set* $\leq_P VC$.
3. The previous lemma shows that
   G has a vertex cover of size $\leq k$, if and only if G has an independent set of size $\geq n - k$.
4. **Proof that** *Independent Set* $\leq_P VC$

(a) We need to exhibit a polynomial reduction from Independent Set to VC.
(b) Given (G, k) for Independent Set, return $(G, n - k)$ for VC.
(c) This runs in polynomial time.
(d) By the observation following from the Lemma, the reduction is correct.

5. **Proof that** $VC \leq_P$ *Independent Set*
   (a) We need to exhibit a polynomial reduction from VC to Independent Set.
   (b) Given (G, k) for VC, return $(G, n - k)$ for Independent Set.
   (c) This runs in polynomial time.
   (d) By the observation following from the Lemma, the reduction is correct.
6. The above results plus transitivity imply that $Clique =_P$ *Independent Set* $=_P VC$.

18.7 Slide 11

18.7.1 Proof that $HP =_P HC$

1. Proof that $HP \leq_P HC$
   (a) Given an arbitrary instance $G = (V, E)$ for HP, we must produce an instance $G' = (V', E')$ for HC.
   (b) Construct G' as follows:
      i. Add a new vertex s to V: $V' = V \cup \{s\}$.
      ii. Add new edges (s, v), for all $v \in V$.
   (c) It is easy to see that this algorithm, F, runs in polynomial time.
   (d) Claim: G has a Hamiltonian path if and only if G' has a Hamiltonian cycle.
   Proof:
      i. Forward:
         A. Assume P is a Hamiltonian path in G with end points a, b.
         B. Then $(s, a) + P + (b, s)$ is a Hamiltonian cycle in G'.
      ii. Backward:
         A. Assume that C' is a Hamiltonian cycle in G'.
         B. Then there must be two incident edges in S; name them $(a, s), (b, s)$.

C. Construct C by removing $(a, s), (b, s)$ from C'.
D. Then C is a Hamiltonian path in G.

2. Proof that $HC \leq_P HP$
   (a) Given an arbitrary instance $G = (V, E)$ for HC, we must produce an instance $G' = (V', E')$ for HP.
   (b) Construct G' as follows:
      i. Choose an arbitrary vertex $x \in V$.
      ii. Add a duplicate x' of x (connect x' to exactly the vertices of G to which x connects).
      iii. Add degree 1 (see next step) vertices t, t'.
      iv. Add edges $(t, x), (t', x')$.
   (c) This is a polynomial time construction.
   (d) Claim: G has a Hamiltonian cycle if and only if G' has a Hamiltonian path.
   Proof:
      i. Forward:
         A. Assume that P is a Hamiltonian cycle in G.
         B. Then P must contain edges $(y_1, x), (y_2, x)$ for some vertices y_1, y_2.
         C. Then $(P \setminus (y_1, x)) + (t', x') + (x', y_1) + (x, t)$ is a Hamiltonian path in G'.
      ii. Backward:
         A. Assume P is a Hamiltonian path in G'.
         B. Since t, t' are the only two degree 1 vertices in G', therefore these must be the endpoints of P.
         C. x' has two neighbours: t' and some other vertex y.
         D. Since x' is a copy of x, therefore (x, y) is an edge in G.
         E. Then $(P \setminus \{(t', x'), (x', y), (t, x)\}) + (x, y)$ is a Hamiltonian cycle in G'.

18.8 Slide 15

18.8.1 Proof that $3SAT \leq_P IS$

1. Let an arbitrary instance for 3SAT be given (i.e. a formula in CNF in which each disjunctive clause has ≤ 3 literals).
2. Construct a graph for each clause (containing edges of the first type), as follows:

(a) For each clause having 3 literals, form a triangle labelling each vertex with a literal of the clause.
(b) If there are only 2 literals, form a linesegment, labelling the two endpoints.
(c) If there is only 1 literal, simply include it alone in the graph we are constructing.

3. To force exactly one choice from each clause, we set k to the number of clauses.
4. We need to avoid choosing opposite literals $x, \neg x$ in different clauses. So add an edge (first type) between any two vertices that correspond to opposite literals.
5. This construction takes polynomial time.
6. Claim: Suppose that the formula contains k clauses. Then the formula is satisfiable if and only if there is an independent set of size k in the constructed graph.

Proof. (a) Forward:
i. Suppose that the formula is satisfiable (i.e. there is a truth valuation which satisfies every clause).
ii. Select one literal that is satisfied, from each clause. Include the corresponding vertex in the constructed independent set.
iii. Since there are k clauses, therefore our constructed vertex set contains k vertices.
iv. Why the constructed set is independent:
A. We choose only one literal from each clause; hence no edge of the first type joins a pair of vertices in the constructed set.
B. Because we have a satisfying truth valuation, therefore no edge of the second type joins a pair of vertices in the constructed set (we cannot simultaneously select a literal plus its negation).

(b) Backward:
i. Suppose that the constructed graph has an independent set of size k.
ii. An independent set can select at most one vertex from each clause (because of the presence of the edges of the first type).
iii. Since there are k clauses, therefore an independent set must then choose exactly one vertex from each clause.

iv. Because of the presence of edges of the second type, therefore for each variable we select at most one literal involving it (possibly neither; never both).
v. How to determine $(x_i)^t$:
   A. If the independent set selected x_i, then set $(x_i)^t = 1$;
   B. if the independent set selected $\neg x_i$, then set $(x_i)^t = 0$;
   C. if the independent set did not select either of x_i or $\neg x_i$, then set $(x_i)^t$ does not matter.
vi. By the consistency edges, this truth valuation is well-defined.
vii. We selected a literal from every clause to satisfy.
viii. Therefore we have constructed a satisfying truth valuation for the formula.

□

18.9 Notes - Éric Lecture 19 in F24

Now from Lecture 20, I think

1. Slide 4 Correct "conjonctive" to "conjunctive"!

19 Lecture 19 - Reductions, P, NP, co-NP

19.1 Global

1. This lecture uses Armin's Lecture 16 slide deck.

19.2 Slide 03

1. Recall from the last lecture, that (minimum) VC asks, given a graph $G = (V, E)$ and a positive integer k, Does there exists a vertex cover for G of size $\leq k$?
2. Correction: Example 1 (the first one): $Alg_v(S, t)$: go through all E and check if t covers the edges and $|t| \leq k$ (aboslute value bars missing in the Slides).
3. Correction: Example 1 (the second one): Rename it to Example 2.
4. **Exercises:**
   (a) Clique:
      i. S: a graph $G = (V, E)$

ii. t: a vertex subset of V
iii. $Alg_v(S,t)$: go through pairs of vertices and check that each pair is connected by a vertex in E.

(b) IS:
i. S: a graph $G = (V, E)$
ii. t: a vertex subset of V
iii. $Alg_v(S,t)$: go through edges and check that no edge has both vertices in t.

(c) HC:
i. S: a graph $G = (V, E)$
ii. t: an ordering of the vertex list V
iii. $Alg_v(S,t)$: go through t in order, verifying that each candidate edge is in E, and that the end is connected back to the beginning.

(d) HP:
i. S: a graph $G = (V, E)$
ii. t: an ordering of the vertex list V
iii. $Alg_v(S,t)$: go through t in order, verifying that each candidate edge is in E.

(e) Subset-Sum: See Slide 11 in this slide deck.

19.3 Slide 04

1. The question of whether an arbitrary graph is non-Hamiltonian is in NP, is still open.

19.4 Slide 05

1. Correction: There is no unique hardest problem in NP. Hence I suggest to change this statement from "the hardest problem in NP" to "a hardest problem in NP".

19.5 Slide 08

19.5.1 Explanation for the argument that Circuit-Sat is NP-complete

1. Use the definition of NP-completeness:
   (a) Let A be an arbitrary problem in NP.

(b) Construct a polynomial-time reduction from A to Circuit-Sat.
(c) This reduction witnesses that $A \leq_P$ Circuit-Sat.
(d) Since A was arbitrary, therefore Circuit-Sat satisfies the definition of NP-completeness.

19.5.2 Explanation for how the argument in the slides follows this template

1. Since $A \in$ NP, there exists a verification algorithm $Alg_A(S, t)$, which runs in polynomial time.
2. Use $Alg_A(S, t)$ to construct a circuit, with t as its input. Treat t as a binary string, as in the original definition of NP. This means that $Alg_A(S, t)$ can be expressed as a boolean function, with inputs the bits of t. Construct the corresponding boolean circuit. Because $Alg_A(S, t)$ runs in polynomial time, therefore this construction runs in polynomial time.
3. To summarize, this construction
   (a) takes an arbitrary instance S for A as input,
   (b) outputs an instance for Circuit-Sat as output, and
   (c) runs in polynomial time.
4. To conclude that this is a correct reduction, we still need to verify that it preserves "yes" and "no" answers.
   (a) If S is a "yes" for A, then a certificate, t, exists for S. Hence the constructed instance for Circuit-Sat will also be a "yes".
   (b) If S is a "no" for A, then no certificate, t, exists for S. Hence the constructed instance for Circuit-Sat will also be a "no".
5. The reduction is correct; this completes the argument that $A \leq_P$ Circuit-Sat.

19.6 Slide 09

19.6.1 Explanation for Circuit-Sat $\leq_P$ 3SAT

1. Let S be an arbitrary instance for Circuit-Sat.
2. Consider pairs of input bits x_i, x_{i+1}, one pair at a time.
3. Each pair combines in some y_i, via $\wedge$ or $\vee$, yielding $(y_i \leftrightarrow (x_i \wedge x_{i+1}))$ or $(y_i \leftrightarrow (x_i \vee x_{i+1}))$.
4. Note that each formula involves at most 3 inputs.

5. Iteratively handle the next level up the tree in the same way.
6. Construct the conjunction of all formulas created in this way.
7. The constructed formula is an instance for 3SAT.
8. It also follows that the constructed formula will be a "yes" for 3SAT if and only if S was a "yes" for Circuit-Sat.
9. Hence we have a correct reduction from Circuit-Sat to 3SAT.
10. This shows that Circuit-Sat $\leq_P$ 3SAT, as claimed.

20 Lecture 20 - NP-completeness I

20.1 Global

1. This lecture continues to use Armin's Lecture 16 slide deck.
2. By the end of this lecture, we finished covering this slide deck.

20.2 Notes - Éric Lecture 21 in F24

Still from Lecture 20, I think

1. Global: To verify a decision problem lies in NP: it must have a polynomial size certificate and a polynomial time verification algorithm.
2. Slide 16: Stuff.

Now from Lecture 21, I think

1. Slide 9:
   (a) certification: are at least 2 y_is 1?
   (b) Darn! Too slow!
   (c) Hey, he mentioned that students see Turing machines in CS 245!
2. Given an arbitrary instance $x \in PROB \in$ NP, build circuit from $B(x, \cdot)$. Input to the circuit = certificate, y.
3. He waved his hands over constructing the circuit. Still, polynomial size.
4. Slide 12:
   (a) To prove $3SAT \leq Independent - Set$.
   (b) We know $I.S. \leq Clique$, $I.S. \leq Vertex{-}Cover$, so $I.S., Clique, Vertex{-}Cover$ are all NP-complete.
   (c) Exercise: explain (English, pseudo-code not required) why the provided construction is polynomial time.

21 Lecture 21 - NP-completeness II

21.1 Global

1. This lecture uses Armin's Lecture 17 slide deck.
2. See also the KT Textbook.
3. See also Lap Chi's L19 notes, especially for the diagrams.

21.2 Slide 02

1. Because we have already proved that 3SAT is NP-Complete, the Theorem stated on the slide will establish that both of DirectedHamiltonianCycle and HamiltonianCycle are NP-complete.

21.3 Slides 03-08

21.3.1 Explanation for 3SAT $\leq_P$ DirectedHamiltonianCycle

1. It is easy to see that DirectedHamiltonianCycle is in NP. (Certificate: a candidate directed Hamiltonian cycle.)
2. Let an arbitrary instance for 3SAT be given (i.e. an arbitrary CNF formula, in which each disjunctive clause has ≤ 3 literals).
3. Let the formula contain variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$.
4. We need to construct a directed graph, $G = (V, E)$, such that the formula is satisfiable if and only if G has a directed Hamiltonian cycle.
5. We need to start by creating some graph structures for the variables and the truth valuation, t.
6. Idea: Associate a long "two-way" path to each variable, x_i:
   (a) L-R if $(x_i)^t = 1$ (true),
   (b) R-L if $(x_i)^t = 0$ (false).
7. The "base" graph (Slide 03) has a one-to-one correspondence between the 2^n possible truth valuations, and the 2^n directed Hamiltonian cycles in the graph.
   (a) There are n two-way paths, one per variable.
   (b) Each path contains $3m$ vertices, having 3 vertices per clause.
   (c) The two end points of the path P_i connect to the two start points of P_{i+1}.
   (d) There is a source vertex s that connects the two start points of P_1.

(e) There is a sink vertex t that connects the two end points of P_n.
(f) The vertex t connects back to s.

8. There are 2^n directed Hamiltonian cycles in this directed graph, since we must use each path P_i in exactly one direction, as the intermediate vertices of these paths are not connected to anything else.
9. This is a good start, with a one-to-one correspondence between truth valuations and Hamiltonian cycles.
10. Next, we will add some clause structures to "kill" all of the Hamiltonian cycles that do not correspond to satisfying truth valuations.
11. Recall that the two-way paths are all of length $3m$. The first three vertices belong to the first clause, and in general the three vertices between $v_{3i+1}, v_{3i+2}, v_{3i+3}$ belong to the i^{th} clause.
12. Suppose we have a clause, say $x_1 \vee x_2 \vee \neg x_3$. Then to satisfy the clause, we want the Hamiltonian cycles to
    (a) go L-R in P_1, P_2 (to satisfy x_1, x_2),
    (b) go R-L in P_3 (to satisfy $\neg x_3$).
13. To this end, create a new vertex c_j, for each clause C_j.
14. Label the vertices in P_i as $v_{i,1}, v_{i,2}, \ldots, v_{i,3m}$.
15. If the literal x_i appears in C_j, then add the directed edges $(v_{i,3j-1}, c_j)$ and $(c_j, v_{i,3j})$.
16. Otherwise, if the literal $\neg x_i$ appears in C_j, then add the directed edges $(c_j, v_{i,3j-1})$ and $(v_{i,3j}, c_j)$. item Do this for each clause, C_j.
17. Note that the edges for $C_j, C_k (k \neq j)$ don't share any vertices. This is why we created the long paths in the first place.
18. This is the whole construction. Clearly it can be done in polynomial time.
19. It remains to prove that the formula is satisfiable if and only if the constructed graph has a directed Hamiltonian cycle.
20. <u>Forward:</u>
    (a) Assume that there is a satisfying assignment.
    (b) As above,
        i. if $(x_i)^t = 1$, then we visit P_i, L-R;
        ii. otherwise if $(x_i)^t = 0$, then we visit P_i, R-L.
    (c) For clause C_j, say $(x_a \vee x_b \neg \vee x_c)$, say, at least one literal is true under t.
    (d) If $(x_a)^t = 1$ (true), then when we follow P_a L-R, we "detour" to visit c_j during the clause j region in the path P_a.
    (e) Similarly, if $(x_a)^t = 0$ (false), then when we follow P_a R-L, we

"detour" to visit c_j during the clause j region in the path P_a.

(f) Since t satisfies every clause, therefore we visit every clause vertex c_j following these directions and detours, and hence form a Hamiltonian cycle in the graph.

21. Backward:
    (a) Assume that the constructed graph has a directed Hamiltonian cycle.
    (b) We need to argue that the directed Hamiltonian cycle must look like those paths and detours above, which correspond with a satisfying truth valuation. But why must this be the case?
    (c) Crucial observation: If we use the directed edge $(v_{a,3j-1}, c_j)$, then we must also use the edge $(c_j, v_{a,3j})$ immediately after it; otherwise the vertex $v_{a,3j}$ will not be reachable, and there will be no way to complete the cycle. (Task: Insert the diagram from Lap Chi's notes here.)
    (d) Since a Hamiltonian cycle visits every vertex, therefore at least one variable path is going in the correct direction.
    (e) All the paths go L-R or R-L (as it must come back immediately after each "detour" to a clause vertex. as observed above).
    (f) Therefore this corresponds to a satisfying truth valuation, as intended.

21.4 Slides 09-10

21.4.1 Explanation for DirectedHamiltonianCycle $\leq_P$ HamiltonianCycle

1. Given an arbitrary directed graph $G = (V, E)$ for DHC, we construct an undirected graph G', in which we create three new vertices v_{in}, v_{mid}, v_{out} for each vertex $v \in V$.
2. Create edges of G' as follows:
    (a) For every $v \in V$, add edges (v_{in}, v_{mid}) and (v_{mid}, v_{out}).
    (b) For each directed edge $(u, v) \in E$, add an undirected edge (u_{out}, v_{in}) in E'.
3. This is the construction. It can clearly be done in polynomial time.
4. It remains to prove that G has a directed Hamiltonian cycle if and only if G' has a Hamiltonian Cycle.
    (a) Forward:

i. Assume that G has a directed Hamiltonian cycle.
ii. By following the cycle and replacing each directed edge (u, v) by (u_{out}, v_{in}) and using the paths $v_{in} \to v_{mid} \to v_{out}$ for all $v \in V$, we get a Hamiltonian cycle in G'.

(b) Backward:
i. Assume that G' has a Hamiltonian Cycle.
ii. In the cycle, start with a vertex v_{in}, then v_{mid} must be a neighbour of v_{in} in the Hamiltonian cycle (otherwise v_{mid} will not be reachable, since it is of degree 2), and then v_{out} must be a neighbour of v_{mid} in the Hamiltonian cycle.
iii. By construction, the the cycle must go to w_{in} for some w, and then w_{mid}, w_{out} as above.
iv. So following the undirected Hamiltonian cycle of $G_{\prime}$, it must be of the form described in the previous direction, hence it corresponds to a directed Hamiltonian cycle in G.

21.5 Notes - Éric Lecture 23 in F24

Still from Lecture 21.

1. Slide 18:
   (a) input size = $\underbrace{\ell}_{\#\ of\ clauses} \cdot \underbrace{\log n}_{\#\ of\ bits\ needed\ to\ write\ indices\ in \{1,\ldots n\}}$
   (b) $x_{1000} \vee x_{1001} \vee \overline{x_{1000}}$
   (c) (becomes)
   $x_1 \vee x_2 \vee \overline{x_1}$

Now from Lecture 22

1. Slide 4:
   (a) We all agree to quietly forget the Euclidean Travelling Salesman Problem.
2. Slide 6:
   (a) $k = 0$: if and only if there exist no vertices. Silly, but correct.

21.6 Notes and Tasks from the Lecture

1. Notes
   (a) **Q:** Is HamiltonianCycle $\leq_P$ DirectedHamiltonianCycle?
   A: Yes, I think so: just make every undirected edge into two directed edges.

2. Tasks
   (a) Add a note to the proof that 3SAT $\leq_P$ DirectedHamiltonianCycle: It is crucial that we select a unique literal that satisfies each clause, in case there are multiple choices. The reason is that for the constructed graph to have a directed Hamiltonian cycle, we must visit each vertex exactly once. If we have several literals that could satisfy a clause, then without the note above, we could visit the clause vertex multiple times. Write this up properly, when time permits.

22 Lecture 22 - NP-completeness III

22.1 Global

1. This lecture uses Armin's Lecture 18 slide deck.
2. See also the KT Textbook.
3. See also Lap Chi's L20 notes, especially for the diagrams.

22.2 3-Dimensional Matching (Slides 03-08)

22.2.1 Global

1. See the KT Textbook, §8.6.
2. This is a generalization of the **bipartite matching problem**.

22.2.2 Explanation for 3SAT $\leq_P$ 3DMatching

1. First, note that 3DMatching is clearly in NP (certificate: a candidate set of hyperedges).
2. The above result will establish that 3DMatching is NP-complete, since 3SAT is.
3. Let an arbitrary instance for 3SAT be given (i.e. an arbitrary CNF formula, in which each disjunctive clause has ≤ 3 literals).
4. Let the formula contain variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_s$.
5. We are done if we can construct an instance H for 3DMatching such that F is satisfiable if and only if H admits a perfect 3D matching (and obviously the reduction must take polynomial time).

6. As in the Hamiltonian cycle problem, we create the "fidget spinner gadgets" to capture the truth valuations of the variables.
7. For any variable $v_i (1 \leq i \leq n)$, we create the gadget pictured, with
   (a) Vertices
      i. $2s$ core vertices $v_{i,1}, \ldots, v_{i,2s}$ (only used in the gadget),
      ii. $2s$ tip vertices $z^T_{i,1}, z^F_{i,1}, \ldots, z^T_{i,s}, z^F_{i,s}$ (will connect to clauses).
   (b) Hyperedges for $1 \leq j \leq s$
      i. $z^T_{i,j}, v_{i,2j-1}, v_{i,2j}$
      ii. $z^F_{i,j}, v_{i,2j}, v_{i,2j+1}$
8. By construction, the core vertices $v_{i,j}$ are not used in any other hyperedges. We will preserve this fact as we add vertices and hyperedges to handle the clauses.
9. This implies that there are only two possibilities for choosing the hyperedges in the variable gadget to cover the core vertices, corresponding to setting the gadget's corresponding variable to 0 (false) or 1 (true):
   (a) 0 (false): Include the F tip vertices, via hyperedges of the form $z^F_{i,j}, v_{i,2j}, v_{i,2j+1}$
   (b) 1 (true): Include the T tip vertices, via hyperedges of the form $z^F_{i,j}, v_{i,2j-1}, v_{i,2j}$

   Note, the first choice determines all the subsequent choices. Hence there are two choices per variable.
10. This captures the binary decision for each variable, as we have one gadget per variable.
11. This also gives us 2^n choices in total, as there are n variables.
12. It remains to add some clause structures to the 3DM instance so that only satisfying truth valuations "survive".
13. For any clause C_j,
   (a) Add two new vertices: a_j, b_j.
   (b) For any literal x_i in C_j, add a hyperedge $(a_j, b_j, z^T_{i,j})$.
   (c) For any literal $\neg x_i$ in C_j, add a hyperedge $(a_j, b_j, z^F_{i,j})$.
14. Note that the hyperedges for different clauses are disjoint, because they use different tips in the variable gadgets.
15. Each clause (s of them) covers one tip. There are $2ns$ tips in total. There will be $2ns - s = (2n-1)s$ tips left over at this point (i.e. not covered yet).
16. Create
   (a) $(2n-1)s$ pairs of " dummy" vertices d_k, e_k, and

(b) all hyperedges $(z^T_{i,j}, d_k, e_k)$ and $(z^F_{i,j}, d_k, e_k)$ for every tip $z_{i,j}$ not yet covered, in every variable gadget There are $(2ns - s)(2ns)$ of these new hyperedges.

17. In total, we have just added
    (a) $2(2n-1)s$ new dummy vertices, and
    (b) $(2n-1)s \cdot (2ns) < 4n^2s^2$ new hyperedges.
18. The construction is now finished. It is wasteful, but it can clearly be done in polynomial time.
19. It remains to prove that the original formula is satisfiable if and only if the constructed 3DM instance admits a perfect 3D-matching.
    (a) Forward:
        i. Assume that there is a satisfying truth valuation, t, for the orignal formula.
        ii. For each variable $x_i (1 \leq i \leq n)$, cover its gadget according to $(x_i)^t$.
            A. $(x_i)^t = 1$ (true): cover the T tips, or
            B. $(x_i)^t = 0$ (false): cover the F tips.
        iii. Since t is a satisfying truth valuation, therefore it satisfies every clause C_j. Every clause has a literal x_i or $\neg x_i$ in it.
        iv. By construction, each variable x_i has a tip in the gadget with the right F/T value. This tells us which hyperedge to use to cover a_j, b_j.
        v. For these remaining $(2n-1)s$ uncovered tips, we use the dummy hyperedges to cover them all.
        vi. This is a perfect 3D matching.
    (b) Backward:
        i. Assume that the constructed 3DM instance has a perfect 3D-matching.
        ii. For each variable x_i's gadget, there are only two ways to cover all of the core vertices $v_{i,j}$.
            A. If these hyperedges don't cover the F tips, then set $(x_i)^t = 1$ (true), and otherwise
            B. if these hyperedges don't cover the T tips, then set $(x_i)^t = 0$ (false).
        iii. It remains to argue that this t satisfies the original formula.
        iv. Let C_j be an arbitrary clause in the formula.
        v. The 3D matching picked exactly one of the hyperedges
            A. $(a_j, b_j, z^T_{i,j})$ iff x_i is in C_j, or

B. $(a_j, b_j, z^F_{i,j})$ iff $(\neg x_i)$ is in C_j.

vi. This says that the above choice for $(x_i)^t$ satisfies C_j.

vii. Since C_j was arbitrary, therefore t is a satisfying truth valuation for the formula.

22.3 Subset Sum (Slides 09-12)

22.3.1 Global

1. See the KT Textbook, §8.8.

22.3.2 Explanation for 3DM $\leq_P$ SubsetSum

1. First, note that SubsetSum is clearly in NP (certificate: a choice of subset).
2. The above result will establish that SubsetSum is NP-complete, since 3DMatching is.
3. Let an arbitrary instance (X, Y, Z all of size n; m hyperedges $E \subset X \times Y \times Z$) for 3DM be given.
4. We are done if we can construct an instance $a_1, \ldots, a_n, K$ for SubsetSum such that the first instance admits a perfect 3D matching if and only if the second instance has some subset $S \subseteq \{1, \ldots, n\}$ such that $K = \sum_{i \in S} a_i$ (and obviously the reduction must take polynomial time).
5. By Slide 10, we have reduced 3DMatching to

   > Given m 0-1 vectors in dimension $3n$, does there exist a subset that sums to the 1-vector in dimension $3n$?

6. This shows that this new decision problem is NP-complete.
7. Hence we will be finished if we can reduce this new decision problem, to SubsetSum.
8. Idea: Think of the $0-1$ vector as the (backwards) binary representation of a number: $(x_u, y_v, z_w) \mapsto 2^{u-1} + 2^{n+v-1} + 2^{2n+w-1}$.
9. With this mapping, if there is a subset of hyperedges that constitute a perfect 3D matching, then their corresponding numbers would add up to $\sum_{i=0}^{3n-1} 2^i$, which corresponds with the 1-vector.
10. However a subset whose numbers sum to $\sum_{i=0}^{3n-1} 2^i$ may not correspond with a perfect 3D matching, because of the possibility of carrying during binary addition.
11. We need a way to solve this carrying problem.

12. There are at most m numbers in any chosen subset.
13. So select our base for addition to be $b = m + 1$.
14. Final Construction:
    (a) Map each triple $(x_u, y_v, z_w) \mapsto b^{u-1} + b^{n+v-1} + b^{2n+w-1}$.
    (b) Define $K = \sum_{i=0}^{3n-1} b^i$ (the 1-vector in base b). This can be done in polynomial time.
15. Claim: There is a perfect 3D matching if and only if there is a subset with sum = K.
    (a) Forward:
        i. Assume that there is a perfect 3D matching.
        ii. We have already explained this direction above.
    (b) Backward:
        i. Assume that there is a subset with sum = K.
        ii. Because the choice of b prevents carrying, for each position ℓ in the base-b representation, the subset sum records how many times we have covered the ℓ^{th} vertex in the 3DM instance.
        iii. As the target number, K, has 1 in each digit, therefore the subset must correspond to a perfect 3D matching.

22.4 Notes and Tasks from the Lecture

1. Notes
    (a) Stuff.
2. Tasks
    (a) In the proof that 3SAT $\leq_P$ 3DM, clarify how the indices for the hyperedges loop back to the start (it's clear from the pictures, of course).

23 Lecture 23 - NP-Completeness

23.1 Notes - Éric Lecture 23 in F24

Still from Lecture 22.

1. Slide 17:
    (a) Per variable, $2s$ tips $\rightarrow 2ns$ total.
    (b) ns covered in pink
    (c) s covered (at least) by clauses

(d) So we get $ns - s$ tips ($= 4$) uncovered ???

Now from Lecture 23

1. Slide 4:
   (a) Stuff.
2. Slide 6:
   (a) Stuff.

24 Lecture 24 - Misc

24.1 Notes - Éric Lecture 24 in F24

Still from Lecture 23.

1. Slide 4:
   (a) $\log t$, because we express the bound on the run-time, in binary form.
2. Slide 7:
   (a) The Halting Problem is NP-hard, but not in NP.

25 Lecture 25 - Max flow

25.1 Max Flow

1. Slide 5:
   (a) The edge in the first sum is named e.
2. Slide 6:
   (a) Not clearly a flow problem yet, but it is "close enough".
   (b) See the graph at the bottom of the slide, where the labels indicate capacities.
3. Slide 7:
   (a) The algorithm might not be polynomial. It might only be pseudo-polynomial.
4. Slide 8:
   (a) Modify the provided flow, to increase its value from 3 to 4.
5. Slide 10:
   (a) Explanation for why we want a **minimal** value of all capacities on γ in G_f:

i. It is the most conservative choice, hence the least likely to violate any flow constraints after we have modified the graph as in the algorithm.

(b) Why the new flow is improved: As on the slide itself!

6. Slide 11: Why we still have a flow afterwards: Let f be the new flow.
   (a) For all integers $0 \leq f'(e) \leq c(e)$
   (b) Suppose e is blue: $f'(e) = f(e) + x$.
   (c) Hence $f'(e) \geq 0$ because $x \geq 0$.
   (d) Also, $\underbrace{x}_{min\ capacity} \leq \underbrace{c(e) - f(e)}_{capacity\ of\ e\ in\ G_f}$ so $\underbrace{f(e) + x}_{f'(e)} \leq c(e)$.
   (e) Now, 1 of 4 possible cases: blue-in, red-out, I think
   red edge got decreased by x.
   blue edge got increased by x.
   Things work out in this case.
   (f) The other 3 cases are similar
   (g) Now suppose e is red? Maybe I missed this case.
   (h) Check these details, ASAP!
7. Slide 13
   (a) After 200000 steps, we will terminate and return the max flow.
   (b) I think that he said this is true polynomial time.
   (c) We can do better at choosing our augmented graph; he did not explain how.
8. Slide 14
   (a) Check that $r^2 = 1 - r$.
   (b) This implies (multiplying through by r^i) $r^{i+2} = r^i - r^{i+1}$.
9. Slide 18
   (a) No need to know how the example was created.
   (b) **Moral:** If we stick to integers, the algorithm will terminate, finding the maximum flow.
   (c) Next Lecture: proof of correctness.

26 Lecture 26 - Max flow = Min cut

Stuff.

27 Lecture 27 - Applications of Flows and Cuts

1. General I think that he said he proved in Lecture 17 that max-flow equals min-cut. Check it!
2. Slide 4
   (a) I think we need a bit more care in the "loop" case: What if we loop back to the source???
   (b) I think the induction step is is (quietly) a proof by contradiction. Check it!
   (c) Recall that the **value** of a flow is the total amount leaving the source node.
   (d) Check all of this, and generate questions for Éric, ASAP.
   (e) Stuff.
3. Slide 11
   (a) Stuff.