

CS 341 Spring 2023:

Lecture Notes

1	Introduction	3
1.1	Asymptotic Review	3
1.2	Types of Algorithms	5
1.3	Recurrence Relations	6
2	Divide and Conquer	8
2.1	Examples	8
3	Graph Algorithms	13
3.1	Graph Theory Review	13
3.2	Breadth-First Search	14
3.3	Shortest Path by BFS	15
3.4	Bipartiteness by BFS	17
3.5	Depth-First Search	17
3.6	Cut Vertices by DFS	19
3.7	Directed Graphs	21
4	Greedy Algorithms	24
4.1	Introduction	24
4.2	Basic Greedy Examples	24
4.3	Shortest Paths: Dijkstra's Algorithm	27
4.4	Minimum Spanning Trees: Kruskal's Algorithm	28
5	Dynamic Programming	32
5.1	Introduction	32
5.2	Interval Scheduling	33
5.3	Knapsack Problem	34
5.4	Subsequence Problems	35
5.5	Graph Algorithms	37
5.6	Shortest Path Revisited: Bellman–Ford	38
5.7	All Shortest Paths: Floyd–Warshall	39
6	Complexity Theory	40
6.1	Introduction	40
6.2	Sample Reductions	41

6.3 NP-completeness	42
7 Final Review	46
Back Matter	47
List of Problems	47
List of Named Results	48
Index of Defined Terms	50

Lecture notes taken, unless otherwise specified, by myself during section 001 of the Spring 2023 offering of CS 350, taught by Armin Jamshidpey.

Lectures

Lecture 1	(05/09)	3
Lecture 2	(05/11)	4
Lecture 3	(05/16)	7
Lecture 4	(05/18)	9
Lecture 5	(05/25)	11
Lecture 6	(05/30)	15
Lecture 7	(06/01)	17
Lecture 8	(06/06)	19
Lecture 9	(06/08)	21
Lecture 10	(06/13)	22
Lecture 11	(06/15)	24
Lecture 12	(06/20)	25
Lecture 13	(06/22)	27
Lecture 14	(06/27)	28
Lecture 15	(06/29)	28
Lecture 16	(07/04)	32
Lecture 17	(07/06)	35
Lecture 18	(07/11)	36
Lecture 19	(07/13)	38
Lecture 20	(07/18)	39
Lecture 21	(07/20)	40
Lecture 22	(07/25)	41
Lecture 23	(07/27)	43
Lecture 24	(08/01)	46

Chapter 1

Introduction

1.1 Asymptotic Review

Recall from CS 240, that given a problem with instances I of size n :

*Lecture 1
(05/09)*

Definition 1.1.1 (runtime)

The runtime of an instance I is $T(I)$.

The worst-case runtime is $T(n) = \max_{\{I: |I|=n\}} T(I)$.

The average runtime is $T_{\text{avg}}(n) = \frac{\sum_{\{I: |I|=n\}} T(I)}{|\{I: |I|=n\}|}$

Recall also the asymptotic comparison of functions $f(n)$ and $g(n)$ with values in $\mathbb{R}_{>0}$:

Definition 1.1.2 (big- O)

$f(n) \in O(g(n))$ if there exists $C > 0$ and n_0 such that $n \geq n_0 \implies f(n) \leq Cg(n)$.

Definition 1.1.3 (big- Ω)

$f(n) \in \Omega(g(n))$ if there exists $C > 0$ and n_0 such that $n \geq n_0 \implies f(n) \geq Cg(n)$. Equivalently, $g(n) \in O(f(n))$.

Definition 1.1.4 (big- Θ)

$f(n) \in \Theta(g(n))$ if there exists $C, C' > 0$ and n_0 with $n \geq n_0 \implies Cg(n) \leq f(n) \leq C'g(n)$. Equivalently, $f(n) \in O(g(n)) \cap \Omega(g(n))$.

Recall also that if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ is finite, then $f(n) \in \Theta(g(n))$.

Definition 1.1.5 (little- o)

$f(n) \in o(g(n))$ if for all $C > 0$, there exists n_0 such that $n \geq n_0 \implies f(n) \leq Cg(n)$.

Equivalently, $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Definition 1.1.6 (little- ω)

$f(n) \in \omega(g(n))$ if for all $C > 0$, there exists n_0 such that $n \geq n_0 \implies f(n) > Cg(n)$.

Equivalently, $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ or $g(n) \in o(f(n))$.

Also, recall that any polynomial of degree k is in $\Theta(n^k)$.¹

We write $n^{O(1)}$ to mean at most polynomial (i.e., $O(n^k(n))$ where $k \in O(1)$)

Exercise 1.1.7. Is 2^{n-1} in $\Theta(2^n)$?

Proof. Notice that $2^{n-1} = \frac{1}{2}2^n$. If we let $C = \frac{1}{2} = C'$, $n_0 = 1$, notice that for $n \geq n_0$, we have $C2^n = 2^{n-1} \leq 2^{n-1} \leq 2^{n-1} = C'2^n$. That is, $2^{n-1} \in \Theta(2^n)$. \square

Exercise 1.1.8. Is $(n-1)!$ in $\Theta(n!)$?

Solution. No. Notice that $\lim_{n \rightarrow \infty} \frac{(n-1)!}{n!} = \lim_{n \rightarrow \infty} \frac{1}{n} = 0$. Therefore, $(n-1)! \in o(n!)$, which contradicts $(n-1)! \in \Theta(n!)$. \square

Consider now multivariate functions $f(n, m)$ and $g(n, m)$ with values in $\mathbb{R}_{>0}$. Then,

Definition 1.1.9 (multivariate big- O)

$f(n, m)$ is in $O(g(n, m))$ if there exist C, n_0, m_0 such that $f(n, m) \leq Cg(n, m)$ for $n \geq n_0$ **or** $m \geq m_0$.

We similarly define the other asymptotic analysis functions. We could alternatively define using $n \geq n_0$ **and** $m \geq m_0$ but they both give the same results.

Notice that all basic operations are not equal. For example, multiplication may take $O(b)$ time for a b -bit word.

Warning: big- O is only an upper bound, so $1 \in O(n^2)$ and $n \in O(n)$, but we know that $1 \ll n$.

Asymptotic notation hides constants. Any $\Theta(n^2)$ algorithm will beat a $\Theta(n^3)$ algorithm eventually. A galactic algorithm is practically irrelevant because the crossing point is stupidly large.

¹As long as n is eventually increasing, i.e., the n^k term dominates.

1.2 Types of Algorithms

Problem 1.2.1 (contiguous subarrays)

Given an array $A[1..n]$, find a contiguous subarray $A[i..j]$ that maximizes the sum $A[i] + \dots + A[j]$.

Consider the brute-force attempt

Algorithm 1.2.2 BRUTEFORCE(A)

```

1:  $opt \leftarrow 0$ 
2: for  $i \leftarrow 1, \dots, n$  do
3:   for  $j \leftarrow i, \dots, n$  do
4:      $sum \leftarrow 0$ 
5:     for  $k \leftarrow i, \dots, j$  do
6:        $sum \leftarrow sum + A[k]$ 
7:     if  $sum > opt$  then
8:        $opt \leftarrow sum$ 
9: return  $opt$ 

```

which has a runtime $\Theta(n^3)$. This is inefficient. We are recomputing the same sum in the j loop, so if we instead keep the running sum:

Algorithm 1.2.3 BETTERBRUTEFORCE(A)

```

1:  $opt \leftarrow 0$ 
2: for  $i \leftarrow 1, \dots, n$  do
3:    $sum \leftarrow 0$ 
4:   for  $j \leftarrow i, \dots, n$  do
5:      $sum \leftarrow sum + A[j]$ 
6:     if  $sum > opt$  then
7:        $opt \leftarrow sum$ 
8: return  $opt$ 

```

we get $\Theta(n^2)$.

We can develop a divide-and-conquer algorithm by noticing that the optimal subarray (if not empty) is either (1) completely in $A[1..n/2]$, (2) completely in $A[n/2 + 1..n]$, or (3) contains $A[n/2]$ and $A[n/2 + 1]$.

Each of MAXIMIZEUPPERHALF and MAXIMIZELOWERHALF have runtime $\Theta(n)$, so DIVIDEANDCONQUER has runtime $2T(n/2) + \Theta(n) \in \Theta(n \log n)$.

Finally, notice that we can instead solve the problem in nested subarrays $A[1..j]$ of sizes $1, \dots, n$. The optimal subarray is either a subarray of $A[1..n-1]$ or contains $A[n]$.

Write $M(j)$ for the maximum sum for subarrays of $A[1..j]$. Then,

$$M(n) = \max(M(n-1), \bar{M}(n)) = A[n] + \max(\bar{M}(n-1), 0)$$

Algorithm 1.2.4 DIVIDEANDCONQUER(A)

```

1: if  $n = 1$  then return  $\max(A[1], 0)$ 
2:  $opt_{lo} \leftarrow \text{DIVIDEANDCONQUER}(A[1..n/2])$ 
3:  $opt_{hi} \leftarrow \text{DIVIDEANDCONQUER}(A[n/2 + 1..n])$ 
4: function MAXIMIZELOWERHALF()
5:    $opt \leftarrow A[n/2]$ 
6:    $sum \leftarrow A[n/2]$ 
7:   for  $i \leftarrow n/2 - 1, \dots, 1$  do
8:      $sum \leftarrow sum + A[i]$ 
9:     if  $sum > opt$  then  $opt \leftarrow sum$ 
10:  return  $opt$ 
11: function MAXIMIZEUPPERHALF()
12:  ...
13:  $opt_{mid} \leftarrow \text{MAXIMIZELOWERHALF}() + \text{MAXIMIZEUPPERHALF}()$ 
14: return  $\max(opt_{lo}, opt_{hi}, opt_{mid})$ .

```

where $\bar{M}(j)$ is the maximum sum for subarrays of $A[1..j]$ that include j . Notice that the optimal subarray containing $A[n]$ is either $A[i..n]$ for $i \leq n - 1$ or exactly $[A[n]]$.

Algorithm 1.2.5 DYNAMICPROGRAMMING(A)

```

1:  $\bar{M} \leftarrow A[1]$ 
2:  $M \leftarrow \max(\bar{M}, 0)$ 
3: for  $i = 2, \dots, n$  do
4:    $\bar{M} \leftarrow A[i] + \max(\bar{M}, 0)$ 
5:    $M \leftarrow \max(M, \bar{M})$ 
6: return  $M$ 

```

which has runtime $\Theta(n)$. We cannot do better than this (proof beyond the scope of the course, but intuitively notice that we cannot find a max without knowing the entire array).

1.3 Recurrence Relations

Recall merge sort.

The recurrence relation is $T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$

If we let c and d be the constants, we get $T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + cn & n > 1 \\ d & n = 1 \end{cases}$

Equivalently, we can sloppily remove floors and ceilings to get $T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & n > 1 \\ d & n = 1 \end{cases}$

Construct now a recursion tree, assuming $n = 2^j$. Notice that we will end up with j layers where layer i has 2^i nodes where each node takes cn time (the last layer nodes take d time).

Theorem 1.3.1 (master theorem)

Suppose $a \geq 1$ and $b > 1$. Consider the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y)$$

in sloppy or exact form. Let $x = \log_b(a)$. Then,

$$T(n) = \begin{cases} \Theta(n^x) & y < x \\ \Theta(n^y \log n) & y = x \\ \Theta(n^y) & y > x \end{cases}$$

Proof. Let $a \geq 1$ and $b \geq 2$. Then, let $T(n) = aT(\frac{n}{b}) + cn^y$ and $T(1) = d$. Also, write for convenience $n = b^j$. We can now consider the recurrence tree.

The i^{th} row in the tree (except the bottom) will have a^i subproblems of size n/b^i which each have cost $c(n/b^i)^y = cn^y b^{-iy}$. The j^{th} row will have a^j nodes with cost d . Then,

$$T(n) = da^j + cn^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i$$

We know that $x = \log_b a$ which gives $b^x = a$. Assume $r = \frac{a}{b^y} = \frac{b^x}{b^y} = b^{x-y}$. Then, we have

Lecture 3
(05/16)

$$\begin{aligned} da^{\log_b n} + cn^y \sum_{i=0}^{j-1} r^i &= dn^{\log_b a} + cn^y \sum_{i=0}^{j-1} r^i \\ &= dn^x + cn^y \begin{cases} j & r = 1 \\ \Theta(1) & r < 1 \\ \frac{r^j - 1}{r - 1} \in \Theta(r^j) & r > 1 \end{cases} \\ &= \begin{cases} dn^x + cn^y \log_b n \in \Theta(n^y \log n) & x = y \\ dn^x + c'n^y \in \Theta(n^y) & x < y \\ dn^x + c''n^x \in \Theta(n^x) & x > y \end{cases} \end{aligned}$$

noting that $r^j = r^{\log_b n} = n^{\log_b r} = n^{x-y}$, so in the latter case $cn^y \Theta(r^j) \in \Theta(n^x)$. \square

When n^x dominates, we call it “heavy leaves”. When n^y dominates, we call it “heavy top”. Otherwise, we call it “balanced”.

Chapter 2

Divide and Conquer

In general, we want to:

- divide: split a problem into subproblems;
- conquer: solve the subproblems recursively; and
- combine: use subproblem results to derive problem result

This is possible when:

- the original problem is easily decomposable into subproblems;
- combining solutions is not costly; and
- subproblems are not overly unbalanced

2.1 Examples

Problem 2.1.1 (counting inversions)

Given an unsorted array $A[1..n]$, find the number of inversions in it, i.e., pairs (i, j) such that $A[i] > A[j]$.

Example 2.1.2. Given $A = [1, 5, 2, 6, 3, 8, 7, 4]$, we get $(2,3)$, $(2,5)$, $(2,8)$, $(4,5)$, $(4,8)$, $(6,7)$, $(6,8)$, and $(7,8)$.

The naive algorithm checks all pairs and takes $\Theta(n^2)$ time. We can do better.

Let c_ℓ be the number of inversions in $A[1..n/2]$, c_r be the number of inversions in $A[n/2 + 1..n]$, and c_t be the number of transverse inversions, i.e., inversions where i is on the left and j is on the right.

We can find c_ℓ and c_r by recursion.

To find c_t , we must count the number of left indices greater than each right index. This can be done by sorting and then binary searching, since the binary search result index gives exactly what we want. The sort takes $O(n \log n)$ and each of the n binary searches takes $O(\log n)$.

This gives us $T(n) \leq 2T(n/2) + O(n \log n) = O(n \log^2 n)$.

We can instead modify MERGESORT and find c_t using a modified MERGE:

Algorithm 2.1.3 Modified MERGE($A[1..n]$) (additions in green)

Require: both halves of A are sorted

```

1: copy  $A$  into a new array  $S$ ;  $c = 0$ 
2:  $i \leftarrow 1$ ;  $j \leftarrow n/2 + 1$ 
3: for  $k \leftarrow 1, \dots, n$  do
4:   if  $i > n/2$  then  $A[k] \leftarrow S[j++]$ 
5:   else if  $j > n$  then
6:      $A[k] \leftarrow S[i++]$ 
7:      $c \leftarrow c + \frac{n}{2}$ 
8:   else if  $S[i] < S[j]$  then
9:      $A[k] \leftarrow S[i++]$ 
10:     $c \leftarrow c + j - (\frac{n}{2} + 1)$ 
11:  else  $A[k] \leftarrow S[j++]$ 

```

Here, every time we merge in an element from the left, we add to c the number of elements on the right which are greater than it. This will run in $\Theta(n \log n)$ time because the modified MERGE is still $\Theta(n)$.

Problem 2.1.4 (polynomial multiplication)

Given $F = f_0 + \dots + f_{n-1}x^{n-1}$ and $G = g_0 + \dots + g_{n-1}x^{n-1}$, calculate $H = FG$.

The naive algorithm takes $\Theta(n^2)$ time to expand.

Notice that we can split $F = F_0 + F_1x^{n/2}$ and $G = G_0 + G_1x^{n/2}$. Then, we have $H = F_0G_0 + (F_0G_1 + F_1G_0)x^{n/2} + F_1G_1x^n$. If we divide and conquer, we make 4 recursive calls with size $n/2$ and $\Theta(n)$ extra work for the additions.

However, $T(n) = 4T(n/2) + \Theta(n) \in \Theta(n^2)$ which is not an improvement.

Lemma 2.1.5 (Karatsuba's identity)

$$(x + y)(a + b) - xa - yb = xb + ya$$

But if we already have F_0G_0 and F_1G_1 , we can use Karatsuba's identity to instead calculate $(F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1 = F_0G_1 + F_1G_0$. That is, we will calculate:

$$\begin{aligned}
 H &= (F_0 + F_1x^{n/2})(G_0 + G_1x^{n/2}) \\
 &= F_0G_0 + ((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1)x^{n/2} + F_1G_1x^n
 \end{aligned}$$

This means we only need to make 3 recursive calls instead of 4.

Then, $T(n) = 3T(n/2) + \Theta(n) \in \Theta(n^{\lg 3})$ which is an improvement.

Lecture 4
(05/18)

Based on this observation, Toom–Cook created a family of algorithms that for $k \geq 2$ make $2k - 1$ recursive calls in size n/k , i.e., $T(n) \in \Theta(n^{\log_k(2k-1)})$ which gets arbitrarily close to linear (but with increasingly massive constants).

If $F, G \in \mathbb{C}[x]$, then we can use FFT to get $T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$ time.

Problem 2.1.6 (matrix multiplication)

Given $A = [a_{i,j}] \in M_{n \times n}$ and $B = [b_{j,k}] \in M_{n \times n}$, calculate $C = AB$.

The naive algorithm takes inputs of size $\Theta(n^2)$ in $\Theta(n^3)$ time.

Consider instead breaking into block matrices: $A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$ and $B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$.

Then, $C = \begin{pmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{pmatrix}$

This makes 8 recursive calls of size $n/2$ and $\Theta(n^2)$ additions, which resolves to $\Theta(n^3)$ (no improvement). However, due to Strassen, we can reduce this to 7, giving $\Theta(n^{\lg 7})$ time.

We can generalize to do k multiplications of $\ell \times \ell$ matrices in $\Theta(n^{\log_\ell k})$ time and k multiplications of $\ell \times m$ by $m \times p$ in $\Theta(n^{3 \log_{\ell mp} k})$ time.

Problem 2.1.7 (closest pairs)

Given n distinct points (x_i, y_i) , find a pair (i, j) that minimizes the distance

$$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Equivalently, minimize $d_{i,j}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$.

Separate the space of points into L and R halfspaces based on the median x value. The closest pair is either entirely in L , entirely in R , or transverse.

We can recursively find minimum distances δ_L and δ_R . Then, if we let $\delta = \min\{\delta_L, \delta_R\}$, any closer transverse points must be within δ units of the median x value.

Now, if we start from the bottom point $P \in L$ by y -value in that band, we only have to compare P with points $Q \in R$ with $y_P \leq y_Q < y_P + \delta$.

We can only have a maximum of 8 points inside the $2\delta \times \delta$ rectangle of possible Q points, because the points must be at least δ apart.

Therefore, we are doing $\Theta(1)$ work for each P , so we do $\Theta(n)$ work to find transverse pairs.

For this to work, we must first sort the points by x and by y (in $O(n \log n)$ time). We can find the median in $O(1)$ time. We split the sorted points in $O(n)$ time for the two recursive calls and find the δ band in $O(n)$ time. Again, it takes $O(n)$ time to find transverse pairs. Therefore, $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

Problem 2.1.8 (selection)

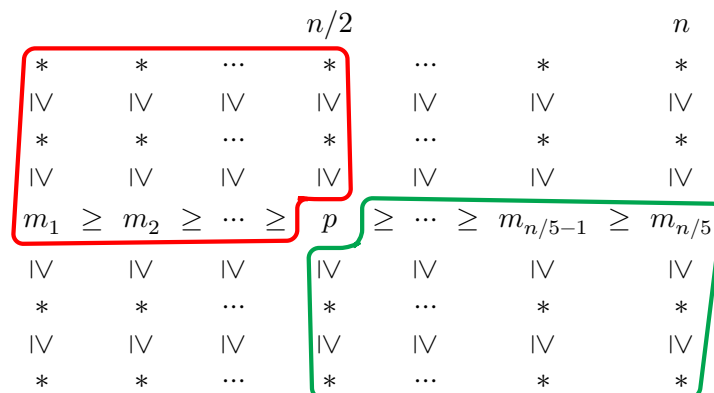
Given $A[0..n-1]$, find the entry that would be at index k if A were sorted.

Recall from CS 240 that selection by sorting takes $O(n \log n)$ time or $O(n)$ randomized expected time using $\text{QUICKSELECT}(A, k)$:

Algorithm 2.1.9 $\text{QUICKSELECT}(A, k)$

- 1: $p \leftarrow \text{CHOOSEPIVOT}(A)$
- 2: $i \leftarrow \text{PARTITION}(A, p)$ $\triangleright i$ is the correct index of p
- 3: **if** $i = k$ **then return** $A[i]$
- 4: **else if** $i > k$ **then return** $\text{QUICKSELECT}(A[0..i-1], k)$
- 5: **else return** $\text{QUICKSELECT}(A[i+1..n-1], k-i-1)$

Consider splitting A into groups G_i of size 5. Then, find the medians m_i of each group. We can choose the pivot p as the median of medians:



Then, we are guaranteed to have $3n/10$ elements **above** and **below** $p = A[i]$, so the recursive calls to $A[0..i-1]$ and $A[i+1..n-1]$ have size at most $7n/10$ (with equality when i is exactly $3n/10$ or $7n/10$).

Therefore, $T(n) \leq T(n/5) + T(7n/10) + O(n)$.

Claim 2.1.10. $T(n/5) + T(7n/10) + O(n) \in O(n)$

Lecture 5
(05/25)

Proof. Proceed by induction. Note that $T(n) \leq \begin{cases} O(1) & n < 120 \\ T(\frac{n}{5}) + T(\frac{7}{10}n + 6) + O(n) & n \geq 120 \end{cases}$

We will show that $T(n) \leq cn$ for large enough c and all $n > 0$. We know that there exists a sufficiently large c such that $T(n) \leq cn$ for $n < 120$ because $T(n)$ is just $O(1) \subsetneq O(n)$.

Choose a constant a to write $O(n)$ as an .

Suppose $T(m) \in O(m)$ for all $0 < m < n$. Then,

$$\begin{aligned}
 T(n) &\leq \frac{cn}{5} + c\left(\frac{7n}{10} + 6\right) + an \\
 &\leq c\frac{n}{5} + c\frac{7n}{10} + 6c + an \\
 &= 9c\frac{n}{10} + 6c + an \\
 &= cn + \left(-c\frac{n}{10} + 6c + an\right)
 \end{aligned}$$

We can show that the latter term is non-positive:

$$\begin{aligned}
 -c\frac{n}{10} + 6c + an \leq 0 &\iff c\left(6 - \frac{n}{10}\right) + an \leq 0 \\
 &\iff c\left(6 - \frac{n}{10}\right) \leq -an \\
 &\iff c\left(\frac{n}{10} - 6\right) \geq an \\
 &\iff c \geq 10a\frac{n}{n-60}
 \end{aligned}$$

Now, if $\frac{n}{n-60} \leq 2$, i.e., $n \geq 120$, then we can say that $c \geq 20a$.

Therefore, we can say that $T(n) \leq cn$, i.e., $T(n) \in O(n)$. □

Example 2.1.11. What does $T(n) = T(\frac{2}{3}n) + T(\frac{n}{3}) + n$ resolve to?

Solution. Notice that if we draw a tree, each layer sums to n (this makes sense inductively since we pass $\frac{2}{3}$ of n to the left and $\frac{1}{3}$ of n to the right). There will be $O(\log_{3/2} n)$ layers in the tree, so it should resolve to $O(n \log n)$. □

Chapter 3

Graph Algorithms

3.1 Graph Theory Review

Recall graph theory from MATH 239, specifically: ms

Definition 3.1.1 (graph)

A graph G is a pair (V, E) where V is a finite set of vertices and E is a set of unordered pairs of distinct vertices, called edges. By convention, we write $n = |V|$ and $m = |E|$.

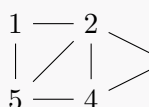
Now, we can define some structures on a graph:

Definition 3.1.2 (adjacency list)

An array $A[1..n]$ such that $A[v]$ is a linked list containing all edges connected to v . This contains $2m$ list cells with total size $\Theta(n + m)$ but takes more than $O(1)$ time to test if an edge exists.

Definition 3.1.3 (adjacency matrix)

A matrix $M \in M_{n \times n}(\{0, 1\})$ such that $M[v, w] = 1$ if and only if $\{v, w\} \in E$. Size is $\Theta(n^2)$ but testing if an edge exists is $O(1)$.

Example 3.1.4. Given the graph  , the adjacency list is:

1 \rightarrow 2 \rightarrow 5
2 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5
3 \rightarrow 2 \rightarrow 4
4 \rightarrow 2 \rightarrow 3 \rightarrow 5
5 \rightarrow 1 \rightarrow 2 \rightarrow 4

and the adjacency matrix is

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Definition 3.1.5 (graph terminology)

We also recall some terms from MATH 239:

- A path is a sequence of vertices v_1, \dots, v_k such that $\{v_i, v_{i+1}\} \in E$ for all i . If a path from v to w exists, we write $v \rightsquigarrow w$.
- A connected graph has $v \rightsquigarrow w$ for all $v, w \in V$.
- A cycle is a path $v \rightsquigarrow v$ of length at least 3 with all elements pairwise distinct.
- A tree is a graph with no cycles.
- A rooted tree is a tree with a vertex chosen to be the root.
- A subgraph of $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$, $E' \subseteq E$, and $u, v \in V'$ for all $uv \in E'$.
- A connected component of G is a connected subgraph of G that is not a subset of any other connected subgraph.

3.2 Breadth-First Search

Problem 3.2.1

Search a graph G starting from a vertex s in order of distance from s .

Algorithm 3.2.2 BFS(G, s)

```

1: let  $Q$  be an empty queue
2: let  $\text{visited}$  be a boolean array of size  $n$  with all entries set to  $\perp$ 
3: ENQUEUE( $s, Q$ )
4:  $\text{visited}[s] \leftarrow \top$ 
5: while  $Q$  is not empty do
6:    $v \leftarrow \text{DEQUEUE}(Q)$ 
7:   for  $w$  neighbours of  $v$  do
8:     if  $\text{visited}[w] = \perp$  then
9:       ENQUEUE( $w, Q$ )
10:     $\text{visited}[w] \leftarrow \top$ 
```

Each vertex is enqueued at most once and dequeued at most once, which has cost $O(n)$. Therefore, each adjacency list is read at most once. The cost for the for loop is $O(\sum \deg v) = O(m)$ by the Handshaking Lemma.

Therefore, the total cost of BFS is $O(n + m)$.

Lemma 3.2.3

`visited`[v] is true for some vertex v if and only if $s \rightsquigarrow v$ in G .

Proof. Let $s = v_0, \dots, v_K$ be the vertices with `visited` $v_i = \top$, in order of discovery. By induction, we show that $s \rightsquigarrow v_i$.

For $i = 0$, $v_0 = s$, so trivially $s \rightsquigarrow s$.

Otherwise, suppose $s \rightsquigarrow v_j$ for all $j < i$. We are currently in the for loop for some vertex w already visited. Therefore, by assumption, $s \rightsquigarrow w$. But since v_i is a neighbour of w , $s \rightsquigarrow v_i$. \square

Exercise 3.2.4. For a connected graph, $m \geq n - 1$.

Lecture 6
(05/30)

Proof. Recall from MATH 239 that if a graph G is connected, then it has a spanning tree T . The spanning tree of n vertices has exactly $n - 1$ edges. Then, since the spanning tree is a subgraph of G , $m \geq |E(T)| = n - 1$, as desired. \square

3.3 Shortest Path by BFS

Problem 3.3.1

What is the shortest path from s to v in G ?

Consider now how we can keep track of parents (predecessors) and levels (depths):

Algorithm 3.3.2 BFS(G, s) with parents and levels

```

1: let  $Q$  be an empty queue
2: let parent be an array of size  $n$  with all entries set to  $\perp$ 
3: let level be an array of size  $n$  with all entries set to  $\infty$ 
4: ENQUEUE( $s, Q$ )
5: parent[ $s$ ]  $\leftarrow s$ 
6: level[ $s$ ]  $\leftarrow 0$ 
7: while  $Q$  is not empty do
8:    $v \leftarrow$  DEQUEUE( $Q$ )
9:   for  $w$  neighbours of  $v$  do
10:    if parent[ $w$ ] =  $\perp$  then
11:      ENQUEUE( $w, Q$ )
12:      parent[ $w$ ]  $\leftarrow v$ 
13:      level[ $w$ ]  $\leftarrow$  level[ $v$ ] + 1
```

We can define a BFS tree T as the subgraph of G made of all w such that `parent`[w] $\neq \perp$ and all edges $\{w, \text{parent}[w]\}$ between those vertices.

Claim 3.3.3. The BFS tree T is in fact a tree.

Proof. Proceed by induction on the vertices for which $\text{parent}[v]$ is not \perp .

When we set $\text{parent}[s] \leftarrow s$, we have one vertex and no edges.

Suppose T is a tree and we are adding $\text{parent}[w] \leftarrow v$. Then, v must have already been in T because it came from Q , so we are extending T by adding (1) the vertex w and (2) the edge $\{v, w\}$. This does not create a cycle because $\text{parent}[w] = \perp$, so T remains a tree.

Therefore, by induction, at the end of BFS, T is a tree. \square

Claim 3.3.4. The levels in the queue Q are non-decreasing.

Proof. Exercise (TODO). \square

Claim 3.3.5. For all vertices u and v , if there is an edge $\{u, v\}$, then $\text{level}[v] \leq \text{level}[u] + 1$.

Proof. Suppose that u and v are adjacent and visited.

If we dequeue v before u , then $\text{level}[v] \leq \text{level}[u] + 1$ by Claim 3.3.4.

If u is dequeued before v , then the parent of v is either u or something else before u . This is because while visiting u , we must either have enqueued v or already visited v . Therefore, v 's parent must be at or before u . Then, by Claim 3.3.4, $\text{level}[\text{parent}[v]] \leq \text{level}[u]$.

That is, $\text{level}[v] = \text{level}[\text{parent}[v]] + 1 \leq \text{level}[u] + 1$. \square

Lemma 3.3.6

For all v in G , there is a path $s \rightsquigarrow v$ in G if and only if there is a path $s \rightsquigarrow v$ in T . If so, the path in T is a shortest path and $\text{level}[v]$ is the distance from s to t .

Proof. By Lemma 3.2.3, $s \rightsquigarrow_G v$ if and only if v is visited. That is, all such v are in T . But T is connected as a tree, therefore $s \rightsquigarrow_G v \iff s \rightsquigarrow_T v$.

Let δ be the distance from s to v . We must show $\text{level}[v] \leq \delta$ and $\delta \leq \text{level}[v]$.

Trivially, $\delta \leq \text{level}[v]$ because $\text{level}[v]$ is the length of the path $s \rightsquigarrow_T v$.

We will prove by induction that for all i , if there is a path of length i from s to v , then $\text{level}[v] \leq i$. For the base case $i = 0$, there are no such paths.

Suppose this is true for $i - 1$, and consider a path $P = s \cdots uv$ with length i . Then, we can decompose P as $P' = s \cdots u$ and uv . But P' has length $i - 1$, so $\text{level}[u] \leq i - 1$. Then, by Claim 3.3.5, $\text{level}[v] \leq \text{level}[u] + 1 \leq i$.

Therefore, since this is true for all i , it is true for $i = \delta$.

Finally, we have that $\delta = \text{level}[v]$ and $s \rightsquigarrow_T v$ is a shortest path. \square

3.4 Bipartiteness by BFS

Definition 3.4.1 (bipartite)

A graph $G = (V, E)$ is bipartite if there exists a partition $U_1 \sqcup U_2 = V$ such that for every $uv \in E$, $u \in U_1$ and $v \in U_2$ (or vice versa).

Problem 3.4.2

Is G bipartite?

Lemma 3.4.3

Suppose G is connected and we run $\text{BFS}(G, s)$ for some s . Let V_1 and V_2 be vertex sets with odd and even level respectively. Then, G is bipartite if and only if all edges have one end in V_1 and one end in V_2 .

Proof. Suppose all edges have one end in V_1 and one end in V_2 . Then, G is bipartite by definition.

Suppose G has bipartition (W_1, W_2) . Then, wLOG say that $s \in W_2$. Since $s \rightsquigarrow v$ for all $v \in V$ and all paths alternate between W_1 and W_2 , odd depth vertices will fall in $W_1 = V_1$ and even ones in $W_2 = V_2$. \square

Lecture 7
(06/01)

This is nice because we can test in $O(m)$ time.

3.5 Depth-First Search

Analogous to BFS, but we use a stack (implicitly with recursion, or explicitly with a stack data structure) to follow neighbours until we cannot.

Algorithm 3.5.1 DFS(G)

Require: G is a graph on n vertices given by adjacency lists

```

1: visited  $\leftarrow$  array of size  $n$  initialized to  $\perp$ 
2: procedure EXPLORE( $v$ )
3:   visited[ $v$ ]  $\leftarrow \top$ 
4:   for  $w$  neighbour of  $v$  do
5:     if visited[ $w$ ] =  $\perp$  then
6:       EXPLORE( $w$ )
7: for  $v \in G$  do
8:   if visited[ $v$ ] =  $\perp$  then
9:     EXPLORE( $v$ )

```

Lemma 3.5.2 (white path lemma)

When we start exploring v , any w connected to v by an unvisited path will be visited during $\text{EXPLORE}(v)$.

Proof. Let $v_0 = v \cdots v_k = w$ be a path $v \rightsquigarrow w$ with v_1, \dots, v_k all not visited. We prove all v_i are visited before $\text{EXPLORE}(v)$ is finished.

Obviously holds for $i = 0$. Suppose it holds for $i < k$. When we visit v_i , $\text{EXPLORE}(v)$ is not finished and v_{i+1} is one of the neighbours.

If $\text{visited}[v_{i+1}]$ is already true (because $v \rightsquigarrow v_{i+1}$ by some other path), we are done. Otherwise, we are going to visit it now, which is before $\text{EXPLORE}(v)$ is finished.

Therefore, v_{i+1} is visited during $\text{EXPLORE}(v)$, as desired. \square

Corollary 3.5.3. After we call EXPLORE at v_1, \dots, v_k , we have visited exactly the connected components containing v_1, \dots, v_k .

Note: we cannot find shortest paths using a DFS tree without customization. For example, the DFS tree for a cycle will be a path even though the root and leaf are adjacent.

The runtime is still $O(n + m)$.

Definition 3.5.4

Let T_1, \dots, T_k be a DFS forest with vertices u and v . Then, u is an ancestor of v if $u, v \in T_i$ for some i and u is on the path from the root of T_i to v . Equivalently, we write that v is a descendant of u .

Lemma 3.5.5 (key property)

All edges in G connect a vertex to one of its descendants or ancestors.

Proof. Let $\{v, w\}$ be an edge and suppose WLOG we visit v first.

Then, when we visit v , (v, w) is an unvisited path $v \rightsquigarrow w$, so by the [white path lemma](#), w must become a descendant of v . \square

Definition 3.5.6 (back edge)

An edge in G connecting an ancestor to a descendant which is not in the DFS forest.

Corollary 3.5.7. All edges are either tree edges or back edges.

Proof. Equivalent statement of the [3.5.5](#). \square

We can extend DFS with **start** and **finish** arrays:

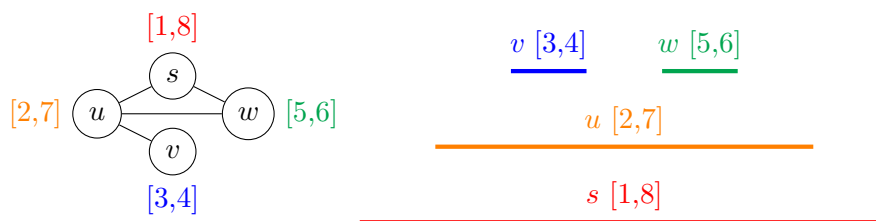
Algorithm 3.5.8 DFS(G) with timing

Require: G is a graph on n vertices given by adjacency lists

```

1: visited  $\leftarrow$  array of size  $n$  initialized to  $\perp$ 
2: start, finish  $\leftarrow$  array of size  $n$ 
3:  $t \leftarrow 1$ 
4: procedure EXPLORE( $v$ )
5:   visited[ $v$ ]  $\leftarrow \top$ 
6:   start[ $v$ ]  $\leftarrow t$ ;  $t++$ 
7:   for  $w$  neighbour of  $v$  do
8:     if visited[ $w$ ] =  $\perp$  then
9:       EXPLORE( $w$ )
10:  finish[ $v$ ]  $\leftarrow t$ ;  $t++$ 
11: for  $v \in G$  do
12:   if visited[ $v$ ] =  $\perp$  then
13:     EXPLORE( $v$ )
  
```

For example, we can draw a graph with $[\text{start}[v], \text{finish}[v]]$ labelled:



Notice that the intervals shrink with depth and follow a structure similar to the well-formed parenthesis problem. We can in fact prove:

Lemma 3.5.9 (parentheses theorem)

If $\text{start}[u] < \text{start}[v]$, then either $\text{finish}[u] < \text{start}[v]$ or $\text{finish}[u] < \text{finish}[v]$.

Proof. If $\text{start}[u] < \text{start}[v]$, we push v on the stack while u is still there, so we pop v before we pop u since stacks are FIFO. \square

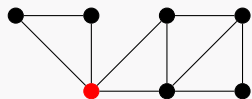
3.6 Cut Vertices by DFS

We define a cut vertex analogous to a bridge edge from MATH 239.

Lecture 8
(06/06)

Definition 3.6.1 (cut vertex)

Given a connected graph G , a vertex $v \in V(G)$ is a cut vertex (or articulation point) if removing v and its edges makes G disconnected.

Example 3.6.2.

has a cut vertex in red.

Problem 3.6.3Which of the vertices in G are cut vertices?Consider a rooted DFS tree T with known parent and level.**Proposition 3.6.4**The root s is a cut vertex if and only if it has more than one child.

Proof. Suppose s has one child v . Then, $T - s$ is a rooted DFS tree with root v (i.e., it remains connected).

Suppose s has subtrees S_1, \dots, S_k . Let $u \in S_i$ and $v \in S_j$ for $i \neq j$. Then, there does not exist a path $u \rightsquigarrow v$ in $T - s$ by the **key property** since it would involve a non-tree, non-back cross edge. Therefore, the subtrees are disconnected in $T - s$. \square

Proposition 3.6.5

Let $a(v) = \min\{\text{level}[w] : vw \in E(G)\}$ and $m(v) = \min\{a(w) : w \text{ descendant of } v\}$. Any non-root vertex v is a cut vertex if and only if it has a child w with $m(w) \geq \text{level}[v]$.

Proof. Let w be a child of v with subtrees T_w and T_v , respectively.

Suppose $m(w) < \text{level}[v]$ and we have removed v . Then, there is a vertex w' in T_w to some vertex v' above v . That is, for any vertex $u \in V(T_w)$, we have that $u \rightsquigarrow w \rightsquigarrow w' \rightsquigarrow v' \rightsquigarrow s$ and T_w is still connected.

Therefore, for v to be a cut vertex, we must have $m(w) \geq \text{level}[v]$.

Suppose $m(w) \geq \text{level}[v]$. Then, by the **key property**, all edges from T_w end in T_v . They are either the tree edge vw or a back edge going to an ancestor at or below v . Therefore, removing v will cause T_w to be disconnected and v is a cut vertex. \square

Therefore, we can solve the cut vertex problem by calculating $m(v)$ for every vertex.

We can compute $a(v)$ in $O(\deg v)$. Notice that if v has children w_1, \dots, w_k , then $m(v) = \min\{a(v), m(w_1), \dots, m(w_k)\}$. Then, if we have the m of the children, we get $m(v)$ in $O(\deg v)$.

By traversing the DFS tree, we get every $m(v)$ in $O(n + m) = O(m)$ (since G connected). Then, we can test the cut vertex condition for each vertex v and each of its children in $O(\deg v)$.

Therefore, we can test all vertices in $O(m)$ time.

Algorithm 3.6.6 FINDCUTVERTICES(G, s)

```

1:  $T \leftarrow \text{DFS}(G, s)$   $\triangleright$  DFS tree for  $G$  with root and level
2:  $a, m \leftarrow$  arrays of size  $|V(G)|$  initialized to  $\infty$ 
3:  $\text{cut} \leftarrow$  array of size  $|V(G)|$  initialized to  $\perp$ 
4: procedure EXPLORE( $v$ )
5:   for  $w$  child of  $v$  do
6:      $a[v] \leftarrow \min\{a[v], \text{level}[w]\}$ 
7:     EXPLORE( $w$ )
8:      $m[v] \leftarrow \min\{m[v], a[w]\}$ 
9:    $m[v] \leftarrow \min\{a[v], m[v]\}$ 
10:  for  $w$  child of  $v$  do
11:    if  $m[w] \geq T.\text{level}[v]$  then  $\text{cut}[v] \leftarrow \top$ 
12: EXPLORE( $T.\text{root}$ )

```

3.7 Directed Graphs

We can define a directed graph similar to an ordinary graph:

Definition 3.7.1 (directed graph)

A graph $G = (V, E)$ where edges are *ordered* pairs (u, v) .

If G has no cycles, it is a directed acyclic graph (DAG).

Note that we allow loops (v, v) . Paths and cycles have the ordinary meaning.

Definition 3.7.2 (topological ordering)

An ordering $<$ of V in a DAG such that $(a, b) \in E$ implies $a < b$.

Proposition 3.7.3

A directed graph is acyclic if and only if there is a topological ordering on it.

Lecture 9
(06/08)

Proof. The backwards direction is clear.

Assume we have a DAG. There exists at least one vertex with in-degree 0, because otherwise there would be a cycle. We can inductively remove the vertex with in-degree 0 to get a topological ordering.

In fact, if run DFS and we order V with the ordering $v < w \iff \text{finish}[w] < \text{finish}[v]$, then we can show that $<$ is a topological order.

Suppose that $(v, w) \in E$.

If we discover v before w , then w is a descendant of v by the [white path lemma](#) so we must finish exploring it before we finish v .

Otherwise, if we discover w before v , then there cannot exist a path $w \rightsquigarrow v$ because otherwise $w \rightsquigarrow vw$ is a cycle. Therefore, $\text{finish}[w] < \text{start}[v] < \text{finish}[v]$.

Therefore, $<$ is a topological order whose existence is necessary and sufficient for a DAG. \square

Definition 3.7.4 (strong connectivity)

A directed graph G is strongly connected if for all v and w in G , there is a path $v \rightsquigarrow w$ (and $w \rightsquigarrow v$)

Corollary 3.7.5. G is strongly connected if and only if there exists s such that for all w there exist paths $s \rightsquigarrow w$ and $w \rightsquigarrow s$.

Proof. The forwards direction is trivial. In the backwards direction, notice that for any two vertices v and w , we have $v \rightsquigarrow s \rightsquigarrow w$ and $w \rightsquigarrow s \rightsquigarrow v$. \square

Problem 3.7.6

How can we test if a graph is strongly connected?

Solution. Call EXPLORE twice, starting from the same vertex s . On the second run, reverse all the edges. Then, if every vertex v is explored in both runs, we know that $s \rightsquigarrow v$ and $v \rightsquigarrow s$, i.e., the graph is strongly connected.

We can reverse the edges using an adjacency list in $O(n + m)$ time, so this algorithm runs in $O(n + m)$ time. \square

Proposition 3.7.7

Contracting the strongly connected components of a directed graph forms a DAG.

Proof. Suppose not. Then there exists a cycle of strongly connected components. However, this means that any vertex from any of these can be reached from any other. Therefore, the strongly connected component is not maximal. \square

Problem 3.7.8

What are the strongly connected components and their respective DAG?

Lecture 10
(06/13)

This has time complexity $O(n + m)$.

Proposition 3.7.10

For any vertices v and w , TFAE: v and w are in the same SCC; and v and w are in the same DFS tree of G^\top (sorted by decreasing finish time).

Algorithm 3.7.9 Kosaraju's algorithm for strongly connected components

```

1: procedure SCC( $G$ )
2:   run DFS( $G$ ) augmented with finish times
3:   sort the vertices by decreasing finish time
4:   run DFS( $G^\top$ )
5:   return the trees in the DFS forest of  $G^\top$ 

```

Proof. Suppose $v, w \in C \in SCC(G)$ and let s be the first vertex visited in C . Then, $s \rightsquigarrow v$ within C and the path is white when visiting s by supposition. By the [white path lemma](#), v will be in the DFS tree. Likewise for w .

Suppose v and w are in a DFS tree T for G^\top rooted at s . That is, among the vertices in T , s has the highest finish time. Let $t \in T$. As a descendent, $s \rightsquigarrow_{G^\top} t$, so $t \rightsquigarrow_G s$.

Claim that t descends from s in G , so we get a path $s \rightsquigarrow_G t$.

Proceed by structural induction on t and its children. Let u be a child of t in T . Suppose $\text{start}[s] \leq \text{start}[t] < \text{finish}[t] \leq \text{finish}[s]$. Since $\text{finish}[u] < \text{finish}[s]$, we have by the [parentheses theorem](#) that either $[s(u)]$ or $(u)[s]$. But the second option is impossible because if $tu \in E(T) \subseteq E(G^\top)$, then $ut \in E(G)$, which means that $u \rightsquigarrow t$ and by the [white path lemma](#), t should be a descendant of u , not s . Therefore, u is a descendant of s , as desired.

Finally, because $s \rightsquigarrow_G t$ and $t \rightsquigarrow_G s$, t is in the strongly connected component of s . □

Problem 3.7.11

Does a graph G contain a Hamiltonian path (i.e., a path P with $P(V) = V$)?

For an undirected graph G , this is one of the canonical NP-complete problems.

For a DAG G , we can do this in linear time with a topological ordering.

Proposition 3.7.12

A DAG G has a Hamiltonian path if and only if it has a topological ordering $v_1 < \dots < v_k$ such that $v_i v_{i+1} \in E(G)$ for all i .

Proof. Let G have a Hamiltonian path $P = v_1 \dots v_k$. Define an ordering $v_1 < \dots < v_k$. Suppose $v_i v_j \in E(G)$. If $i > j$, then $v_i v_j v_{j+1} \dots v_i$ is a cycle. However, G is a DAG, so we must have $i < j$. Therefore, $<$ is a topological ordering as desired.

Suppose G has a topological ordering $v_1 < \dots < v_k$ with $v_i v_{i+1} \in E(G)$ for all i . Then, we immediately get a Hamiltonian path given by $v_1 \dots v_k$. □

Chapter 4

Greedy Algorithms

4.1 Introduction

Suppose we are solving a combinatorial optimization problem, i.e., a problem with a large (but finite) domain \mathcal{D} such that we are trying to find an optimal solution $E \in \mathcal{D}$ that maximizes/minimizes some sort of cost function.

*Lecture 11
(06/15)*

We will build E step-by-step by taking the locally best solution. Usually, it is very hard to prove correctness/optimality but easy to find a counterexample.

For example, recall the Huffman encoding from CS 240. We build the binary code tree by joining trees with the least frequencies. This actually minimizes the length of the encoding.

4.2 Basic Greedy Examples

Problem 4.2.1 (interval scheduling)

Suppose we have n intervals $[s_i, f_i]$. What is the subset of disjoint intervals with maximum length?

We can show that a few naive greedy algorithms are wrong by drawing counterexamples:

- Choose $\min_i s_i$:
- Choose $\min_i \{f_i - s_i\}$:
- Choose minimum conflicts:

However, we can prove that the greedy algorithm taking the earliest finish time is optimal.

Algorithm 4.2.2 INTERVALSCHEDULING($I = [[s_1, f_1], \dots, [s_n, f_n]]$)

```

1:  $S \leftarrow \emptyset$ 
2:  $I \leftarrow$  sort  $I$  by finish time
3: for  $[s_i, f_i] \in I$  do
4:   if  $[s_i, f_i]$  has no conflicts in  $S$  then
5:      $S \leftarrow S \cup \{[s_i, f_i]\}$ 

```

Proposition 4.2.3

Suppose O is optimal. Then, $|S| = |O|$ where S is generated by Algorithm 4.2.2.

Proof. Let i_1, \dots, i_k be the intervals in S ordered by their addition and likewise j_1, \dots, j_m be the intervals in O ordered by increasing finish time.

We prove the claim that for all $r \leq k$, $f_{i_r} \leq f_{j_r}$. Proceed by induction on r .

For $r = 1$ this is true since i_1 is the interval with the earliest finish time.

Suppose $r > 1$ and it is true for $r - 1$. Then, $f_{i_{r-1}} \leq f_{j_{r-1}}$ by assumption and $f_{j_{r-1}} < s_{j_r}$ by the order we set on O . Therefore, $f_{i_{r-1}} < s_{j_r}$.

That is, at the time the greedy algorithm chose i_{r-1} , j_r was an option. Since the greedy algorithm picks the earliest finish time, $f_{i_r} \leq f_{j_r}$.

Now, suppose for a contradiction that S is not optimal, i.e., $|S| < |O|$. Then, there must be a j_{k+1} . But by the above claim, $f_{i_k} \leq f_{j_k} < s_{j_{k+1}}$. This means j_{k+1} was an option for the greedy algorithm, so it would not have stopped at i_k and instead added j_{k+1} .

Therefore, S must be optimal. □

We call proofs of this kind, i.e., contradicting that greedy could not have chosen an optimal solution, greedy stays ahead.

We can also greedily solve a similar problem:

Lecture 12
(06/20)

Problem 4.2.4 (interval colouring)

Suppose we have n intervals $[s_i, f_i]$. Use the minimum number of colours to colour the intervals, so that each interval gets one colour and any overlapping intervals get different colours.

Consider the algorithm:

Algorithm 4.2.5 INTERVALCOLOURING($I = [[s_1, f_1], \dots, [s_n, f_n]]$)

```

1:  $c \leftarrow$  empty colouring
2:  $I \leftarrow$  sort  $I$  by start time
3: for  $[s_i, f_i] \in I$  do
4:    $c(i) \leftarrow$  minimum  $c$  such that there are no conflicts

```

which we do not bother analyzing the time complexity of. We show correctness:

Proposition 4.2.6

Suppose that Algorithm 4.2.5 uses k colours. There is no way to colour I with $k - 1$ colours.

Proof. Suppose interval ℓ is the first to use k . Then, the algorithm must have found $k - 1$ overlapping intervals with colours $1, \dots, k - 1$. Let these be intervals i_1, \dots, i_{k-1} . By the initial sorting, we have $s_{i_j} < s_\ell$ for $j = 1, \dots, k - 1$. Also, since they overlap, we have $f_{i_j} > f_\ell$. Therefore, s_ℓ is a point with k intervals, meaning that it is impossible to colour with $k - 1$ colours. \square

Problem 4.2.7 (minimize total completion time)

Suppose we have n jobs each requiring processing time p_i , and we are adding one job each step (e.g., the first step runs just one job, the fifth step runs five jobs, the last step runs all jobs). Order the jobs such that the total processing time is minimized.

The setup is a bit weird, so we can construct an example.

Example 4.2.8. For $n = 5$ and $\mathbf{p} = [2, 8, 1, 10, 5]$, we can construct tables to find the total processing time if we do not order the jobs and if we order the jobs by increasing processing time:

2	8	1	10	5	Σ	1	2	5	8	10	Σ
2	2	2	2	2	10	1	1	1	1	1	5
		8	8	8	24		2	2	2	2	8
			1	1	3			5	5	5	15
				10	20				8	8	16
					5					10	10
2	10	11	21	26	70	1	3	8	16	26	54

to find that the total processing times are 70 and 54, respectively.

Proposition 4.2.9

The total processing time is minimized when $e(i)$ is a permutation of $[n]$ such that $(p_{e(i)})$ is non-decreasing.

Proof. Suppose there is an optimal permutation e that is not non-decreasing. That is, there exists an i such that $p_{e(i)} > p_{e(i+1)}$.

The total processing time is:

$e(1)$	$e(2)$...	$e(i)$	$e(i+1)$...	$e(n)$	Σ
$p_{e(1)}$	$p_{e(1)}$		$p_{e(1)}$	$p_{e(1)}$		$p_{e(1)}$	$np_{e(1)}$
	$p_{e(2)}$		$p_{e(2)}$	$p_{e(2)}$		$p_{e(2)}$	$(n-1)p_{e(2)}$
			\vdots	\vdots		\vdots	\vdots
			$p_{e(i)}$	$p_{e(i)}$		$p_{e(i)}$	$(n-i+1)p_{e(i)}$
				$p_{e(i+1)}$		$p_{e(i+1)}$	$(n-i)p_{e(i+1)}$
						\vdots	\vdots
						$p_{e(n)}$	$p_{e(n)}$

Suppose we swap $e(i)$ and $e(i+1)$. Then, we have removed one copy of $p_{e(i)}$ and added one copy of $p_{e(i+1)}$. But by assumption, $p_{e(i+1)} - p_{e(i)} < 0$, so this swap decreases the total processing time, and the solution was not optimal. \square

4.3 Shortest Paths: Dijkstra's Algorithm

Recall that we can define a weight function on a graph.

Lecture 13
(06/22)

Definition 4.3.1 (weight function)

Given a graph $G = (V, E)$, a function $w : E \rightarrow \mathbb{R}$. We call (G, w) a weighted graph.

Then, we define the weight for a path $P = v_0 \cdots v_k$ by $w(P) = \sum_{i=1}^k w(v_{i-1}v_i)$.

Remark 4.3.2. A shortest path exists in any directed weighted graph with no negative-weight cycles.

Problem 4.3.3 (single-source shortest path)

Given $G = (V, E)$ with weight $w : E \rightarrow \mathbb{R}_{\geq 0}$ and a source $s \in V$, find a shortest path from s to each $v \in V$.

We denote the length of the shortest path $s \rightsquigarrow v$ by $\delta(s, v)$.

Remark 4.3.4. If $v_0 \cdots v_k$ is a shortest path from v_0 to v_k , then $v_0 \cdots v_i$ is a shortest path from v_0 to v_i for all $0 \leq i \leq k$.

Proof. Suppose not. Then, use the shorter path to get to v_i and continue onward to v_k to get a shorter path to v_k . \square

Dijkstra's algorithm is built on this observation.

For each vertex, we maintain an estimate of the distance $d[v]$ and a predecessor in that path estimate $\pi[v]$. We start with all vertices in a set Q and pop vertices one at a time in order of d , adding them to a set C . When a vertex moves from Q to C , update the distance estimates and predecessors of its neighbours.

Algorithm 4.3.5 DIJKSTRA(G, w, s)

```

1: for each vertex  $v \in V$  do
2:    $d[v] \leftarrow \infty$ 
3:    $\pi[v] \leftarrow \perp$ 
4:  $d[s] \leftarrow 0$ 
5:  $C \leftarrow \emptyset$ 
6:  $Q \leftarrow V$ 
7: while  $Q \neq \emptyset$  do
8:    $u \leftarrow \text{EXTRACTMIN}(Q)$ 
9:    $C \leftarrow C \cup \{u\}$ 
10:  for neighbours  $v \in \text{Adj}[u]$  do
11:    if  $d[v] > d[u] + w(uv)$  then
12:       $d[v] = d[u] + w(uv)$ 
13:       $\pi[v] = u$ 

```

Notice that we implement Q and C as heaps since then we get time complexity $O(|V| \log |V| + |E| \log |V|)$ (for the extractions and the updates, respectively) instead of $O(|V|^2 + |E|)$ with arrays.

Claim now that Dijkstra's is correct. Proceed by a greedy stays ahead proof.

Lecture 14
(06/27)

Proposition 4.3.6

For each vertex $v \in V$, $d[v] = \delta(s, v)$ at the time v is added to C .

Proof. Suppose for a contradiction that $u \in V$ is the first vertex for which $d[u] \neq \delta(s, u)$ when it is added to C . Denote the iteration when u is added to C as time t .

Let P be a shortest path $s \rightsquigarrow u$. Since $s \in C$ and $u \in V - C$, there exists a pair of vertices $x \in C$ and $y \in V - C$ with $xy \in P$. We claim that at time t , $d[y] = \delta(s, y)$.

Since u is the first vertex with $d[u] \neq \delta(s, u)$, for all vertices w in C at time t , $d[w] = \delta(s, w)$. In particular, $d[x] = \delta(s, x)$. When x was added to C , the edge xy was considered. By Remark 4.3.4, since P is a shortest path, the parts from s to x and y are also shortest paths. Therefore, $\delta(s, y) = d[x] + w(x, y)$ which is exactly what $d[y]$ is set to during that iteration.

However, we can also say that $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$. But at time t , Dijkstra's chose u from the min-heap, so we must have $d[u] \leq d[y]$.

Therefore, $d[y] = d[u]$ so we must have tight $d[y] = \delta(s, y) = \delta(s, u) = d[u]$, which is our contradiction. \square

4.4 Minimum Spanning Trees: Kruskal's Algorithm

Recall again some MATH 239 content:

Lecture 15
(06/29)

Definition 4.4.1 (spanning tree in G)

Given a connected graph $G = (V, E)$, a tree $T = (V, A)$ where $A \subseteq E$.

Suppose we have a weighted graph. Then,

Problem 4.4.2 (minimal spanning tree)

Given a connected graph G with edge weights w , what is the spanning tree with minimal weight $w(T) = \sum_{e \in A} w(e)$?

Recall that a tree is a graph with no cycles. Kruskal's algorithm greedily selects the lowest-weight edge available that does not create a cycle:

Algorithm 4.4.3 GREEDYMST(G)

```

1:  $A \leftarrow$  empty graph
2: sort edges by increasing weight
3: for  $e \in E$  do
4:   if  $e$  does not create a cycle in  $A$  then
5:      $\quad$  append  $e$  to  $A$ 

```

Now, we prove that this indeed works.

Claim 4.4.4. Let $G = (V, E)$ be a connected graph and $A \subseteq E$. If (V, A) has no cycles and $|A| < n - 1$, then there is always an edge $e \in E - A$ such that $(V, A \cup \{e\})$ still has no cycles.

Proof. In any graph, $|V| - c \leq |E|$ where c is the number of components $C_i = (V_i, E_i)$ in (V, E) . Since each component is connected, $|E_i| \geq |V_i| - 1$. Then,

$$|V| - c = \sum_{i=1}^c |V_i| - \sum_{i=1}^c 1 = \sum_{i=1}^c (|V_i| - 1) \leq \sum_{i=1}^c |E_i| = |E|$$

In particular, for (V, A) , we have $n - c < n - 1 \implies c > 1$. This means there exists an edge e that connects two components of (V, A) .

Since e is a bridge of $(V, A \cup \{e\})$, it does not add a cycle. □

Claim 4.4.5. If the output of Algorithm 4.4.3 is $A = [e_1, \dots, e_r]$, then (V, A) is a spanning tree and $r = n - 1$.

Proof. By construction, (V, A) has no cycles (i.e., it is a tree). Suppose (V, A) is not spanning. Then, it has at least two components and there exists an edge $e \notin A$ such that $(V, A \cup \{e\})$ has no cycles because G is connected. Now, either:

- $w(e) < w(e_1)$. This cannot happen because e_1 is the edge with the smallest weight.

- $w(e_i) < w(e) < w(e_{i+1})$ for some i . This also does not work because if e was considered after inserting e_i and rejected, then that means that e creates a cycle in $(V, \{e_1, \dots, e_i\})$. But by construction, e does not create a cycle in (V, A) .
- $w(e) > w(e_r)$. This also does not make sense, because the algorithm would have included it in A after selecting e_r .

Therefore, no such e exists, which means that (V, A) is spanning. Since (V, A) is a spanning tree, we also get that $r = n - 1$. \square

Claim 4.4.6. Let (V, A) and (V, T) be two spanning trees, and let e be an edge in T but not in A .

Then, there exists an edge e' in $A - T$ such that $(V, T + e' - e)$ is still a spanning tree. Also, e' is on the cycle that e creates in A .

(This is Theorem 5.2.4 from MATH 239)

Proof. Let $e = uv$. Then, $(V, A + e)$ contains a cycle $C = vw \dots v$.

Since T is a tree, $(V, T - e)$ has two connected components T_1 and T_2 . WLOG, suppose $v \in T_1$. Then, C starts in $V(T_1)$, immediately crosses to $V(T_2)$, and at some point later crosses back into $V(T_1)$. Therefore, there exists an edge $e' \in E(C) \subseteq A + e$ connecting T_1 and T_2 that is in A but not in T .

It follows that $(V, T + e' - e)$ is a spanning tree by reconnecting those components. \square

Now, proceed by an exchange argument.

Proposition 4.4.7

Let A be the output of Algorithm 4.4.3 and (V, T) be any spanning tree. Then, (V, A) is a spanning tree with $w(A) \leq w(T)$.

Proof. We know that (V, A) is a spanning tree from Claim 4.4.5.

Claim that $w(A) \leq w(T)$. We induct on the size of $T - A$. If $T - A = \emptyset$, then $A = T$ and naturally $w(A) = w(T)$.

Suppose $A \neq T$. Then, let $e \in T - A$. By Claim 4.4.6, there exists an edge $e' \in A - T$ such that $(V, T + e' - e)$ is a spanning tree and e' is on the cycle that e creates in A .

Since we rejected e , it must have created a cycle in A . Therefore, it appeared in the sorted list after all elements in its induced cycle, in particular, $w(e') \leq w(e)$.

Then, if we let $T' = T + e' - e$, we have $w(T') = w(T) + w(e') - w(e) \leq w(T)$.

Also, since T' has one more edge in common with A , $|T' - A| < |T - A|$. By the inductive hypothesis, $w(A) \leq w(T') \leq w(T)$.

Therefore, A is optimal. \square

Now, consider how we can implement this optimally. Every vertex is initially in a component of just itself. Then, as edges are added, the components grow. To check if a cycle is formed, check if both ends of an edge are in the same component.

We need a data structure that allows us to quickly identify a set given an element belongs to ($\text{FIND}(e)$) and also join two sets ($\text{UNION}(S_1, S_2)$). With these operations, we can rewrite Kruskal's as:

Algorithm 4.4.8 GREEDYMSTUNIONFIND(G)

```

1:  $T \leftarrow []$ 
2:  $U \leftarrow \{\{v_1\}, \dots, \{v_n\}\}$ 
3: sort edges by increasing weight
4: for  $e \in E(G)$  sorted do
5:   if  $U.\text{FIND}(e_k.1) \neq U.\text{FIND}(e_k.2)$  then
6:      $U.\text{UNION}(U.\text{FIND}(e_k.1), U.\text{FIND}(e_k.2))$ 
7:    $T.\text{APPEND}(e_k)$ 

```

We can implement this with U as an array of linked lists paired with an array of indices X such that $e \in U[X[e]]$. Then, we can run FIND in $O(1)$ time and UNION in $O(n)$ time.

There will be $O(m)$ FIND 's and $O(n)$ UNION 's. The total time complexity is then $O(m \log m + n^2)$.

We can optimize this slightly. First, only traverse the smaller list when unioning. Also, have each linked list in U keep track of its size and have an optional pointer to a continuation. Then, by setting the pointer, we can get concatenation in $O(1)$ even though the overall union is still $O(n)$ (because of the updates to X).

However, for any vertex v , the size of the list containing v at least doubles every time we update $X[v]$. That means that $X[v]$ is updated $O(\log n)$ times, meaning the overall cost per union per vertex is $O(\log n)$ for $O(n \log n)$ total.

Therefore, we can do Kruskal's in $O(m \log m)$.

Chapter 5

Dynamic Programming

5.1 Introduction

Recall the Fibonacci numbers F_n defined by $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ with the naive algorithm

*Lecture 16
(07/04)*

Algorithm 5.1.1 FIB(n)

```
1: if  $n = 0$  then return 0
2: else if  $n = 1$  then return 1
3: elsereturn FIB( $n - 1$ ) + FIB( $n - 2$ )
```

Assuming we count additions as unit cost, the runtime is $T(n) = F_{n+1} + 1 \in \Theta(\varphi^n)$ which is bad.

Notice that we are recomputing small F_n a bunch of times, but we actually only need each one once. We can instead cache:

Algorithm 5.1.2 FIBCACHED(n)

Require: $T \leftarrow [0, 1, \perp, \dots, \perp]$ global array of size n

```
1: if  $T[n] = \perp$  then
2:    $T[n] \leftarrow$  FIBCACHED( $n - 1$ ) + FIBCACHED( $n - 2$ )
3: return  $T[n]$ 
```

Also, notice that the dependency graph of which subproblems require each other is a DAG. Therefore, we can take an order on the dependencies and iterate: This is our dynamic programming

Algorithm 5.1.3 FIBITERATIVE(n)

```
1:  $T \leftarrow [0, 1, \perp, \dots, \perp]$  0-indexed array of size  $n$ 
2: for  $i = 2, \dots, n$  do
3:    $T[i] \leftarrow T[i - 1] + T[i - 2]$ 
4: return  $T[n]$ 
```

algorithm. In fact, we can optimize even more by noticing that we can discard all but the last two

elements of the array, giving a constant-space algorithm:

Algorithm 5.1.4 FIBOPTIMAL(n)

```

1:  $(u, v) \leftarrow (0, 1)$ 
2: for  $i = 2, \dots, n$  do
3:    $(u, v) \leftarrow (v, u + v)$ 
   return  $v$ 

```

All these improved algorithms run in $O(n)$ time, a significant improvement.

We can give a general recipe for dynamic programming algorithms:

1. **Identify the subproblem:** We are retaining solutions in an array. What are the dimensions of the array? What does each entry represent? Where will the final answer be in the array?
2. **Establish DP-recurrence:** How does a subproblem contribute to a larger subproblem? What is the dependency between cells in the array?
3. **Set base cases:** Initialize the array with some non-recursively defined base cases.
4. **Specify the order of computation:** Clarify the DAG of subproblem dependencies. How does the algorithm maintain this order?
5. **Recover the solution (if needed):** What subproblem answers provide the problem solution? How, if necessary, do we traceback the solution from the subproblems?

We can often convert a DP algorithm into iterative loop(s). Distinguish divide and conquer algorithms which do not always solve subproblems and are not easily rewritten iteratively.

5.2 Interval Scheduling

Problem 5.2.1 (weighted interval scheduling)

Recall Problem 4.2.1. Now, add a weight w_i to each interval. We choose a subset $T \subseteq [n]$ which maximizes $W = \sum_{i \in T} w_i$.

Example 5.2.2. Let $I = [[2, 8], [2, 4], [5, 6], [7, 9]]$ with weights $[6, 2, 1, 2]$.

Solution. By inspection, since the weight $w_1 = 6 > 5 = w_2 + w_3 + w_4$, the solution is $T = [1]$ with $W = 6$. □

Notice that we can split on whether we accept the last interval I_n and write for example that the optimal weight

$$W(I_1, \dots, I_n) = \begin{cases} w_n + W(I_{m_1}, \dots, I_{m_s}) & \text{if we choose } I_n \\ W(I_1, \dots, I_{n-1}) & \text{if we do not} \end{cases}$$

where I_{m_1}, \dots, I_{m_s} are the $s < n$ intervals not intersecting I_n .

Suppose we sort the intervals by finish time, i.e., $f_i \leq f_{i+1}$ for all i . Then, we have $m_1, \dots, m_s = 1, \dots, j$ where $j = \max\{i : f_i < s_n\}$ because I_n is the last interval with the latest finish time, so we only need to compare its start time with earlier intervals' finish times. (If j does not exist just return $w_n + 0$.)

We need to calculate the j -values for every i :

Algorithm 5.2.3 FINDJS($A, s_1, \dots, s_n, f_1, \dots, f_n$)

```

1:  $j \leftarrow$  array of size  $n$ 
2:  $f_0 \leftarrow \infty$ 
3:  $i \leftarrow 1$ 
4: for  $k = 0, \dots, n$  do
5:   while  $i \leq n$  and  $f_k \leq s_{A[i]} < f_{k+1}$  do
6:      $j[i] \leftarrow k$ 
7:      $i++$ 
8: return  $j$ 

```

where A is a sorting permutation such that $(s_{A[i]})$ is non-decreasing. This runs in $O(n \log n) + O(n) = O(n \log n)$ time.

Now, for the main procedure, we define $W[i]$ as the maximal weight possible with the intervals I_1, \dots, I_i .

Then, for $W[0] = 0$ and $i \geq 1$, $W[i] = \max\{W[i-1], w_i + W[j[i]]\}$.

Since $W[i]$ depends only on entries in W before it, we can just iterate on $i = 1, \dots, n$ in $O(n)$ time.

Algorithm 5.2.4 INTERVALSCHEDULING($s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Require: intervals are sorted by finish time

```

1:  $A \leftarrow$  sorting permutation of  $s_1, \dots, s_n$ 
2:  $j \leftarrow$  FINDJS( $A, s_1, \dots, s_n, f_1, \dots, f_n$ )
3:  $W \leftarrow$  0-indexed array of size  $n$ 
4:  $W[0] \leftarrow 0$ 
5: for  $i = 1, \dots, n$  do
6:    $W[i] \leftarrow \max\{W[i-1], w_i + W[j[i]]\}$ 
7: return  $W[n]$ 

```

This gives a total time for the algorithm of $O(n \log n) + O(n) = O(n \log n)$.

5.3 Knapsack Problem

Problem 5.3.1 (0/1 knapsack)

Suppose we have items with weights w_1, \dots, w_n and values v_1, \dots, v_n but our knapsack has capacity W . We want to select items $S \subseteq \{1, \dots, n\}$ satisfying $\sum_{i \in S} w_i \leq W$ and maximizes $\sum_{i \in S} v_i$.

Example 5.3.2. $\mathbf{w} = [3, 4, 6, 5]$, $\mathbf{v} = [2, 3, 1, 5]$, $W = 8$.

Solution. The optimal $S = \{1, 4\}$ with weight $3 + 5 = 8$ and value $2 + 5 = 7$. \square

For each item n , we can either choose it or we can not. Let $O[W, n]$ be best value for a knapsack of capacity W and considering only the items $1, \dots, n$. Then, $O[W, n]$ is either $v_n + O[W - w_n, n - 1]$ or $O[W, n - 1]$.

Lecture 17
(07/06)

We can initialize $O[0, i] = 0$ for all i and $O[w, 0] = 0$ for all w . To be able to calculate $O[W, n]$, we must have already calculated $O[W - w_n, n - 1]$. In particular, if we iterate on n first, we can guarantee that the entire row $O[, n - 1]$ exists before considering $O[W, n]$:

Algorithm 5.3.3 01KNAPSACK($v_1, \dots, v_n, w_1, \dots, w_n, W$)

```

1:  $O \leftarrow$  0-indexed array of size  $(n + 1) \times (W + 1)$ 
2:  $O[0, :] \leftarrow \mathbf{0}$ ;  $O[:, 0] \leftarrow \mathbf{0}^\top$ 
3: for  $i = 1, \dots, n$  do
4:   for  $w = 1, \dots, W$  do
5:     if  $w_i > w$  then
6:        $O[w, i] \leftarrow O[w, i - 1]$ 
7:     else
8:        $O[w, i] \leftarrow \max\{v_n + O[W - w_n, n - 1], O[W, n - 1]\}$ 
9: return  $O[W, n]$ 
```

The runtime here is obviously $\Theta(nW)$. We call this pseudo-polynomial because it is polynomial in n (the size of the input) but also in W (the *value* of an input). It is not polynomial because the size parameters are n and $\lg W$, but we have $n2^{\lg W}$.

5.4 Subsequence Problems

Problem 5.4.1 (longest increasing subsequence)

Find the longest (potentially discontinuous) increasing subsequence of an array $A[1..n]$ of integers.

Example 5.4.2. Given $A = [7, 1, 3, 10, 11, 5, 19]$, the longest increasing subsequence is $[1, 3, 10, 11, 19]$.

Notice that there are $\Theta(2^n)$ subsequences, so brute force is very bad here.

Suppose we try doing DP and storing $\ell[i]$ as the longest increasing subsequence of $A[1..i]$. This doesn't work, since we can't immediately deduce $\ell[i + 1]$ from just $\ell[i]$ and A .

We could instead store into $L[i]$ a pair of the length and the last entry (ℓ, c) . Then, we can add on the next element $L[i] \leftarrow (\ell + 1, A[i])$, but what is $L[i]$ if we do not select $A[i]$?

Alternatively, let $L[i]$ be the length of the longest increasing subsequence of $A[1..i]$ that ends with $A[i]$. Then, $L[1] = 1$. The longest increasing subsequence S_i ending at $A[i]$ either looks like $[\dots, A[j], A[i]] = [\dots S_j, A[i]]$ for some j or just $[A[i]]$.

Algorithm 5.4.3 LONGESTINCREASINGSUBSEQUENCE($A[1..n]$)

```

1:  $L \leftarrow$  array of size  $n$ 
2:  $L[1] \leftarrow 1$ 
3: for  $i = 2, \dots, n$  do
4:    $L[i] \leftarrow 1$ 
5:   for  $j = 1, \dots, i - 1$  do
6:     if  $A[j] < A[i]$  then
7:        $L[i] \leftarrow \max\{L[i], L[j] + 1\}$ 
8: return  $\max L$ 

```

$\triangleright S_i = [A[i]]$
 $\triangleright S_i = [\dots S_j, A[i]]$

This algorithm runs in $\Theta(n^2)$ time which is much faster than $\Theta(2^n)$. Note that we don't return the actual sequence here, only its length, but it is trivial to find the sequence from the array L .

Problem 5.4.4 (longest common subsequence)

Given two arrays of characters (strings) $A[1..n]$ and $B[1..m]$, find the maximum length of a (potentially discontinuous) subsequence common to both A and B .

Example 5.4.5. For $A = \text{blurry}$ and $B = \text{burger}$, we should return burr for $k = 4$.

As with Problem 5.3.1, we have to work in a 2D problem space. Let $M[i, j]$ be the longest subsequence length between $A[1..i]$ and $B[1..j]$. Zero out $M[0, \cdot]$ and $M[\cdot, 0]$. Then, $M[i, j]$ will be the greatest of either (1) ignoring $B[j]$, (2) ignoring $A[i]$, or (3) adding $A[i] = B[j]$:

Algorithm 5.4.6 LONGESTCOMMONSUBSEQUENCE($A[1..n], B[1..m]$)

```

1:  $M \leftarrow$  0-indexed array of size  $n + 1 \times m + 1$ 
2:  $M[0, \cdot] \leftarrow \mathbf{0}$ ;  $M[\cdot, 0] \leftarrow \mathbf{0}^\top$ 
3: for  $i = 1, \dots, n$  do
4:   for  $j = 1, \dots, m$  do
5:      $M[i, j] \leftarrow \max\{M[i, j - 1], M[i - 1, j]\}$ 
6:     if  $A[i] = B[j]$  then
7:        $M[i, j] \leftarrow \max\{M[i, j], 1 + M[i - 1, j - 1]\}$ 
8: return  $M[n, m]$ 

```

Notice that because we iterate by i first, $M[i - 1, 0..m]$ will have values. Also, since we are iterating by increasing j , $M[i, 1..j - 1]$ will be calculated. Therefore, this algorithm works and runs in $\Theta(nm)$ time.

Problem 5.4.7 (edit distance)

Given two arrays of characters $A[1..n]$ and $B[1..m]$, what is the minimum number of **add**, **delete**, or **change** operations are required to turn A into B ?

Lecture 18
(07/11)

Example 5.4.8. For $A = \text{snowy}$, $B = \text{sunny}$, the edit distance is 3.

Both $\text{snowy} \rightarrow \text{sunny}$ and $\text{snowy} \rightarrow \text{sunny}$ each take 3 operations.

Let $D[i, j]$ be the edit distance between $A[1..i]$ and $B[1..j]$. Then, $D[0, j] = j$ for all j (add j characters to the empty string) and $D[i, 0] = i$ for all i (add i characters to the empty string). Otherwise, the value of $D[i, j]$ is the minimum of:

- Changing the last character: $D[i - 1, j - 1] + \delta_{A[i]B[j]}$
- Deleting $A[i]$ and matching $A[1..i - 1]$ with $B[1..j]$: $D[i - 1, j] + 1$
- Adding $B[j]$ and matching $A[1..i]$ with $B[1..j - 1]$: $D[i, j - 1] + 1$

Computing all values of D takes $\Theta(mn)$ time.

5.5 Graph Algorithms

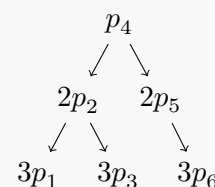
Problem 5.5.1 (optimal BST)

Given objects indexed by $1, \dots, n$ with probabilities of access p_1, \dots, p_n (with $\sum p_i = 1$), what is the optimal binary search tree that minimizes the expected access time?

This problem is similar to the greedy Huffman tree problem (where frequencies are given instead of probabilities) and finding an optimal ordering for a linked list (greedily sort by probability).

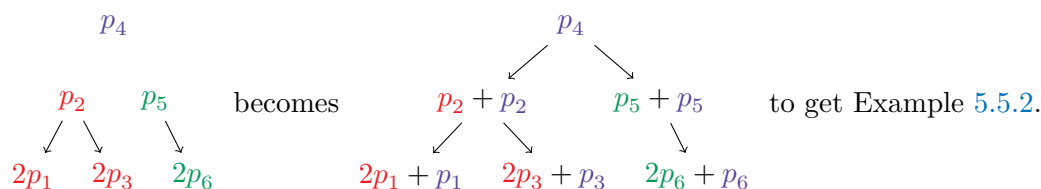
The expected access time is $\sum_{i=1}^n p_i(\text{depth}(i) + 1)$.

Example 5.5.2. A possible BST with $n = 6$ with the cost of each node:



We will split into subproblems based on where we place the root of a subtree. Let $M[i, j]$ be the minimal cost for a tree of the items $\{i, \dots, j\}$ and $M[i, j] = 0$ when $j < i$. If we select k as the root of the i, j -subtree, then the cost of the tree is the cost of the $i, k - 1$ and $k + 1, j$ subtrees plus p_k .

When we place a subtree beneath k , the depth of every node increases by 1. That is, we can take the weight from the M array and add on one more copy of p_ℓ for each node ℓ in the subtree:



Then,

$$\begin{aligned} M[i, j] &= \min_{i \leq k \leq j} \left(\textcolor{red}{M}[i, \textcolor{red}{k} - 1] + \sum_{\ell=i}^{\textcolor{blue}{k}-1} \textcolor{blue}{p}_\ell + \textcolor{blue}{p}_k + \textcolor{green}{M}[\textcolor{green}{k} + 1, j] + \sum_{\ell=k+1}^{\textcolor{violet}{j}} \textcolor{violet}{p}_\ell \right) \\ &= \min_{i \leq k \leq j} (\textcolor{red}{M}[i, \textcolor{red}{k} - 1] + \textcolor{green}{M}[\textcolor{green}{k} + 1, j]) + \sum_{\ell=i}^{\textcolor{violet}{j}} \textcolor{violet}{p}_\ell \end{aligned}$$

Notice that $\sum_{\ell=i}^j p_\ell = \sum_{\ell=1}^j p_\ell - \sum_{\ell=1}^{i-1} p_\ell$. We can cache these sums in $O(n)$ time. We now have our algorithm:

Algorithm 5.5.3 OPTIMALBST(p_1, \dots, p_n)

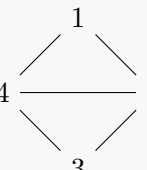
```

1:  $S[0] \leftarrow 0$ 
2: for  $i = 1, \dots, n$  do
3:    $S[i] \leftarrow S[i-1] + p_i$ 
4:   for  $i = 1, \dots, n+1$  do
5:      $M[i, i-1] \leftarrow 0$ 
6:     for  $d = 0, \dots, n-1$  do  $\triangleright d = j - i$ 
7:       for  $i = 1, \dots, n-d$  do
8:          $j \leftarrow d + i$ 
9:          $M[i, j] \leftarrow \min_{i \leq k \leq j} (M[i, k-1] + M[k+1, j] + S[j] - S[i-1])$ 
```

which runs in $O(n^3)$ time.

Problem 5.5.4 (largest independent set of a tree)

Given a tree $T = (V, E)$, what is the largest independent set (i.e., a set $S \subseteq V$ such that $S^2 \cap E = \emptyset$)?

Example 5.5.5. If $G =$  , then $S = \{1, 3\}$ and we return 2.

Let $I[v]$ be the size of the largest independent set of the subtree rooted at v . We can either include v in the independent set or exclude it. If we include it, take the sum of the $I[u]$ for each grandchild u . Otherwise, we take the sum of the $I[u]$ for the children of u .

We will have to calculate I bottom-up, but otherwise this gives us an $O(n)$ algorithm.

5.6 Shortest Path Revisited: Bellman–Ford

Recall Problem 4.3.3 (single-source shortest path) that we solved with Dijkstra's.

Lecture 19
(07/13)

5.7 All Shortest Paths: Floyd–Warshall

Lecture 20
(07/18)

Chapter 6

Complexity Theory

6.1 Introduction

We consider in general decision problems.

*Lecture 21
(07/20)*

Definition 6.1.1 (decision problem)

A map that takes a problem instance and returns a truth value. In general, a map $A : \mathcal{I}(A) \rightarrow \{0, 1\}$. We call an instance where $A(I) = 1$ a yes-instance (otherwise a no-instance).

Note that we can consider a problem instance $I \in \mathcal{I}(A)$ as a natural number (formally) since we typically represent them as some sort of binary string.

Proposition 6.1.2

Almost all decision problems are undecidable.

Proof. Notice that we can represent the map $A : \mathbb{N} \rightarrow \{0, 1\}$ as a binary string $A(0)A(1)\dots$. Then, assuming there is no structure here, we can perform a diagonal argument to show that the problem is uncountably infinite.

However, solutions are a finite bit string, which are only countably infinite. □

Definition 6.1.3 (P)

The class of decision problems that can be solved in polynomial time, i.e., in $O(n^p)$ time for $p > 0$.

Definition 6.1.4 (polynomial reduction)

A problem A is reducible to a problem B if there exists a function $f : \mathcal{I}(A) \rightarrow \mathcal{I}(B)$ such that yes-instances are mapped to yes-instances (no-instances to no-instances) and f is computable in $O(n^p)$ for some p . We write $A \leq_P B$.

As with normal orders, we say that $A =_P B \iff A \leq_P B \wedge B \leq_P A$.

Suppose that $A \leq_P B$ and $B \in P$. Then, given an instance $I \in \mathcal{I}(A)$, we can solve it by solving $f(I)$ (which is computed in polynomial time) as an instance of B (which is polynomial).

Lemma 6.1.5 (transitivity of polynomial reduction)

If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

Proof. By definition, there exists $f : \mathcal{I}(A) \rightarrow \mathcal{I}(B)$ and $g : \mathcal{I}(B) \rightarrow \mathcal{I}(C)$. Then, $g \circ f : \mathcal{I}(A) \rightarrow \mathcal{I}(C)$ and preserves yes-no. \square

Lemma 6.1.6 (proving hardness)

Suppose $A \leq_P B$ and we know that $A \notin P$. Then, $B \notin P$.

Proof. By contradiction. If $B \in P$, then we could solve A in polynomial time. \square

6.2 Sample Reductions

Recall from graph theory that a clique is a set of vertices $S \subseteq V(G)$ such that for all $u, v \in S$, $uv \in E(G)$. Similarly, an independent set is a set S of vertices such that for all $u, v \in S$, $uv \notin E(G)$.

Proposition 6.2.1

Consider the problems **Clique** (does a graph G have a clique of size k ?) and **IS** (does a graph G have an independent set of size k ?). Then, $\text{Clique} =_P \text{IS}$.

Proof. \square

Proposition 6.2.2

Consider the problems **HC** (does a graph G have a Hamiltonian cycle?) and **HP** (does a graph G contain a Hamiltonian path?). Then, $\text{HC} =_P \text{HP}$.

Proof. Consider first the Hamiltonian s, t -path problem (does G contain a Hamiltonian path $s \rightsquigarrow t$?) and that we are given an instance (G, s, t) . Then, (G, s, t) is a yes-instance if and only if $G + x + sx + tx$ is a yes-instance of **HC**.

Likewise, a Hamiltonian cycle $s \rightsquigarrow ts$ exists if and only if $(G - st)$ is a yes-instance of the Hamiltonian s, t -path problem.

Lecture 22
(07/25)

Since both of those operations are polynomial time removals/additions of vertices/edges, we have that the Hamiltonian s, t -path problem $=_P$ HC.

Now, consider the actual problem HP. We can show $HP \leq_P HC$ in the same way. Consider a graph G and add a vertex x adjacent to all vertices. Then, this new graph G' has a Hamiltonian cycle if and only if G has a Hamiltonian path.

Finally, to show $HC \leq_P HP$, consider a graph G with a Hamiltonian cycle $s \rightsquigarrow t \rightsquigarrow s$. Create G' by splitting an arbitrary vertex t into two vertices t and t' such that $s \rightsquigarrow t$ and $t' \rightsquigarrow s$. Then, because of the way we divided the edges of t , a Hamiltonian path in G' must start at t and end at t' . It follows that G has a Hamiltonian cycle if and only if G' has a Hamiltonian path.

how do we divide the edges of t ?

Therefore, $HC = HP$. \square

Recall from CS 245 that a binary function of n variables is satisfiable if there exists an assignment of truth values to variables that makes the expression true. Recall also that we may write any binary function in conjunctive normal form, i.e., as a conjunction of a finite set of m disjunctions of literals (either variables or their negations).

Proposition 6.2.3

Consider the problem 3SAT (is a CNF formula of at most 3 literals per clause satisfiable?). Then, $3SAT \leq_P IS$.

Proof. The reduction will take advantage of the fact that to make the whole formula true, we must select at least one literal from each disjunction to make true.

Construct a graph of all the literals. Attach each literal in the same clause. Attach any two complementary literals.

Then, ask if there is an independent set of size at least m .

For correctness, suppose that G has an independent set S of size at least m . Assign each variable $x = \top$ if $\exists x \in S$ and $x = \perp$ if $\exists \bar{x} \in S$. Since literals are adjacent to their complements, x will either be set to true or false (or neither, in which case we just assign \top arbitrarily). Also, since literals are adjacent in G if they are from the same clause, the m elements of the independent set must come from each of the m clauses by the pigeonhole principle. Therefore, this assignment satisfies G .

Conversely, suppose there exists a satisfying assignment. Then, simply construct the according independent set. Because edges exist only between contradictory literals or between clauses, the set will indeed be independent.

this feels wrong

Therefore, $3SAT \leq_P IS$. \square

6.3 NP-completeness

Consider the **SubsetSum** problem, where we are given a set of integers S and must find a subset $T \subseteq S$ such that $\sum_{x \in T} x = 0$. This problem is hard.

However, suppose an oracle gives us T and claims it has sum 0. We can write a helper function

$\text{VERIFY}(I, C)$ which returns **yes** if I is a yes-instance and C is a valid certificate that proves I is a yes-instance. In our example, $\text{VERIFYSUBSETSUM}(S, T)$ checks that indeed every element in T is also in S and also that they sum to 0.

Definition 6.3.1 (NP)

The class of decision problems with yes-instances that can be verified in polynomial time. Equivalently, the class of decision problems that can be solved by a non-deterministic algorithm in polynomial time.

For example, 3SAT is in NP because evaluating the clauses given a valid variable truth-value assignment can be done in polynomial time.

*Lecture 23
(07/27)*

Not all decision problems are in NP: for example, consider whether a graph is non-Hamiltonian. The no-instances are easily verifiable, but the yes-instances are hard.

Definition 6.3.2 (co-NP)

The class of decision problems with no-instances that can be verified in polynomial time.

Clearly, all problems in P are also in NP and co-NP: simply solve the problem in polynomial time. Therefore, $P \subseteq NP$ and $P \subseteq \text{co-NP}$. This leads to the most famous problem in computer science.

Conjecture 6.3.3

$P \stackrel{?}{=} NP$

To make deciding whether $NP \subseteq P$ easier, we create a notion of the “hardest” problems in NP.

Definition 6.3.4 (NP-complete)

A problem $X \in NP$ is NP-complete if for all $Y \in NP$, we have $Y \leq_P X$. Then, we write $X \in NPC$.

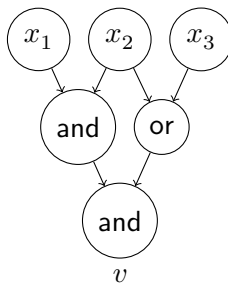
Then, it immediately follows that $P = NP \iff \exists X \in NPC, X \in P$.

Theorem 6.3.5 (Cook–Levin)

$3SAT \in NPC$

This is a useful theorem, since once we have on NP-complete problem, we can just show that any other problem, for example IS, is NP-complete because $3SAT \leq_P IS$.

Proof (sketch). Consider the CircuitSAT problem. We are given a DAG with labelled vertices. Inputs are marked with x_1, \dots, x_n . Internal vertices are marked by Boolean operators **and**, **or**, and **not**. For example,



For a vertex v , is there some truth-value assignment to the x_i 's that makes v true?

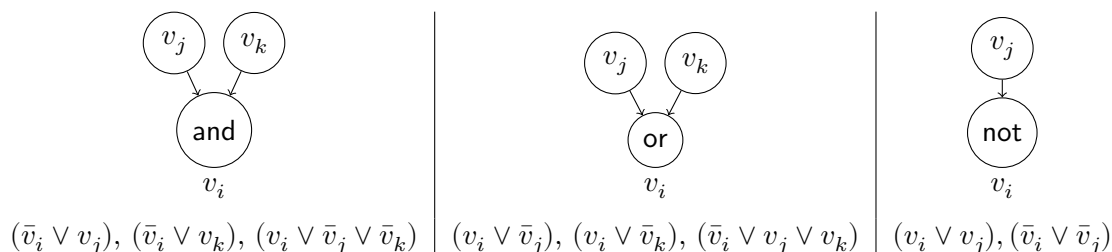
We will show that **CircuitSAT** is NP-complete, and then that $\text{CircuitSAT} \leq_P 3\text{SAT}$.

Let $A \in \text{NP}$ and S be a yes-instance of A . We want to find an algorithm that checks of a certificate t can prove that S is indeed a yes-instance.

This verification algorithm is a Boolean function (i.e., it takes in t as input and outputs a Boolean value), so we can write it as a circuit (recall from CS 245 that \wedge , \vee , and \neg are sufficient to write any Boolean function). Therefore, we can just call **CircuitSAT** to find t , which is as good as solving S .

Therefore, $A \leq_P \text{CircuitSAT}$, and $\text{CircuitSAT} \in \text{NPC}$.

Now, notice that we can transform the **CircuitSAT** DAG into a set of conjunctive clauses with at most three literals:



Therefore, $\text{CircuitSAT} \leq_P 3\text{SAT}$, which means that $3\text{SAT} \in \text{NPC}$. □

For reference, a list of NP-complete problems:

- 3SAT, SAT
- independent set, vertex cover, clique
- (directed) Hamiltonian cycle, Hamiltonian path
- travelling salesman
- subset sum
- 0/1 knapsack

We will show NP-completeness for Hamiltonian cycles and paths.

Theorem 6.3.6

$3\text{SAT} \leq_P \text{DirectedHamiltonianCycle} \leq_P \text{HamiltonianCycle}$

Proof. We begin by showing $3\text{SAT} \leq_P \text{DirectedHamiltonianCycle}$. That is, given a formula, we must

create a directed graph such that the formula is satisfiable if and only if the graph is Hamiltonian.

To do this, we will first create a graph with 2^n Hamiltonian cycles corresponding to each possible truth value assignment. Then, we will add vertices to constrain the valid cycles to those consistent with the given clauses. \square

Chapter 7

Final Review

Exercise 7.0.1. Suppose we want to schedule people be on call between time S and T . If each of the n people are available from s_i to t_i , give a greedy algorithm to assign the minimum number of people.

Lecture 24
(08/01)

Assume that the input is already sorted by start time $S \leq s_1 \leq \dots \leq s_n \leq T$.

Solution. Since we know we must start at S , consider all the intervals that start at S . Then, pick the one with the latest end time.

For each subsequent selection, consider all the intervals with start times before the last chosen end time and then select the one with the latest end time.

Algorithm 7.0.2 GREEDYSCHEDULEASSIGN($S, T, [s_1, t_1], \dots, [s_n, t_n]$)

```
1:  $O \leftarrow \emptyset$ 
2:  $s \leftarrow S, o \leftarrow 1$ 
3: for  $i = 1, \dots, n$  do
4:   if  $s_i \leq s$  then
5:     if  $t_i \geq t_o$  then
6:        $o \leftarrow i$ 
7:   else
8:      $O \leftarrow O \cup \{o\}$ 
9:      $s \leftarrow t_o$ 
10:   $o \leftarrow i$ 
```

This runs in $O(n)$ time. □

Exercise 7.0.3. An $i \times j$ rectangle is worth $P[i, j]$. Given an $n \times m$ rectangle, give a dynamic programming algorithm to find the optimal way to cut the rectangle into smaller rectangles.

Solution. Let $M[i, j]$ be the optimal value of a rectangle after considering subdivisions. Then, $M[i, j]$ is either:

- $P[i, j]$, the value without cutting;

- $M_V(i, j) = \max \bigcup_{1 \leq k \leq i} \{M[k, j], M[i - k, j]\}$, the maximum value of a vertical cut; or
- $M_H(i, j) = \max \bigcup_{1 \leq k \leq j} \{M[i, k], M[i, j - k]\}$, the maximum value of a horizontal cut.

Finally, we just iterate. Each of the $O(nm)$ iterations takes $O(n + m)$ time, so we have $O(n^2m + nm^2)$. \square

Exercise 7.0.4. Consider the problem **ModifiedSTPath**: given an edge-weighted directed graph, is there a simple s, t -path (i.e., with no repeated vertices) with total weight at most k . What complexity class is **ModifiedSTPath** in?

Solution. We can find shortest paths in polynomial time, so find a shortest path via Dijkstra and check if its weight is at most k . This means **ModifiedSTPath** \in P. Therefore, it is also in NP and co-NP. \square

Exercise 7.0.5. Show that **ModifiedSTPath** is NP-complete.

Proof. We will show that **DirectedHamiltonianPath** \leq_P **ModifiedSTPath**.

Suppose we have a directed graph $G = (V, E)$ as input to **DirectedHamiltonianPath**. Notice that a Hamiltonian path is just a simple path with $|V| - 1$ edges.

If we give every edge in G a weight of -1 , then we can just count the edges that a path traverses. Let $G' = (V \cup \{s, t\}, E \cup \{sv, tv : v \in V\})$ and $w(e) = -1$. Then, we can call **MODIFIEDSTPATH**($G', w, s, t, -(|V| + 1)$) which will respond “yes” exactly when there is a path of maximum length $|V| + 1$ and “no” otherwise.

Therefore, **DirectedHamiltonianPath** \leq_P **ModifiedSTPath**, which means **ModifiedSTPath** is NP-complete, as desired. \square

List of Problems

1.2.1 Problem (contiguous subarrays)	5
2.1.1 Problem (counting inversions)	8
2.1.4 Problem (polynomial multiplication)	9
2.1.6 Problem (matrix multiplication)	10
2.1.7 Problem (closest pairs)	10
2.1.8 Problem (selection)	11
4.2.1 Problem (interval scheduling)	24
4.2.4 Problem (interval colouring)	25
4.2.7 Problem (minimize total completion time)	26
4.3.3 Problem (single-source shortest path)	27
4.4.2 Problem (minimal spanning tree)	29
5.2.1 Problem (weighted interval scheduling)	33
5.3.1 Problem (0/1 knapsack)	34
5.4.1 Problem (longest increasing subsequence)	35
5.4.4 Problem (longest common subsequence)	36
5.4.7 Problem (edit distance)	36
5.5.1 Problem (optimal BST)	37
5.5.4 Problem (largest independent set of a tree)	38

List of Named Results

1.3.1 Theorem (master theorem)	7
2.1.5 Lemma (Karatsuba’s identity)	9
3.5.2 Lemma (white path lemma)	18
3.5.5 Lemma (key property)	18
3.5.9 Lemma (parentheses theorem)	19
6.1.5 Lemma (transitivity of polynomial reduction)	41
6.1.6 Lemma (proving hardness)	41
6.3.5 Theorem (Cook–Levin)	43

Index of Defined Terms

- NP, [43](#)
- NP-complete, [43](#)
- P, [40](#)
- co-NP, [43](#)

- adjacency list, [13](#)
- adjacency matrix, [13](#)
- ancestor, [18](#)
- articulation point, [19](#)

- back edge, [18](#)
- BFS tree, [15](#)
- big- Ω , [3](#)
- big- Θ , [3](#)
- big- O , [3](#)
- bipartite, [17](#)

- certificate, [43](#)
- clique, [41](#)
- combinatorial optimization, [24](#)
- conjunctive normal form, [42](#)
- connected component, [14](#)
- connected graph, [14](#)

- cycle, [14](#)
- decision problem, [40](#)
- descendant, [18](#)
- directed acyclic graph, [21](#)
- directed graph, [21](#)

- edges, [13](#)

- Fibonacci numbers, [32](#)

- galactic algorithm, [4](#)
- graph, [13](#)
- greedy stays ahead, [25](#)

- independent set, [38](#), [41](#)
- inversion, [8](#)

- literals, [42](#)
- little- ω , [4](#)
- little- o , [4](#)

- no-instance, [40](#)

- path, [14](#)
- polynomial reduction, [41](#)

- problem instance, [40](#)
- pseudo-polynomial, [35](#)

- root, [14](#)
- rooted tree, [14](#)
- runtime
 - average, [3](#)
 - of an instance, [3](#)
 - worst-case, [3](#)

- satisfiable, [42](#)
- sloppy recurrence, [6](#)
- spanning tree in G , [29](#)
- strongly connected, [22](#)
- subgraph, [14](#)

- topological ordering, [21](#)
- tree, [14](#)

- vertices, [13](#)

- weight function, [27](#)
- weighted graph, [27](#)

- yes-instance, [40](#)