CS 341 Spring 2023:
Lecture Notes

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

Lectures

Chapter 1

Introduction

1.1 Asymptotic Review

Lecture 1 (05/09)

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

Definition (*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 (*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 (*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 (*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 $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite, then $f(n) \in \Theta(g(n))$.

Definition (*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, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Definition (*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, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ or $g(n) \in o(f(n))$.

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

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

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

Exercise 1.1.1. 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, $2^{n-1} \in \Theta(2^n)$. □

Exercise 1.1.2. Is $(n-1)!$ in $\Theta(n!)$?

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

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

Definition (*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.

Lecture 2 (05/11)

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.

1.2 Types of Algorithms

Problem 1.1 (*contiguous subarrays*)

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

Consider the brute-force attempt

Algorithm 1.2.1 BruteForce(A)

```
1: opt ← 0
2: for i ← 1, ..., n do
3:     for j ← i, ..., n do
4:         sum ← 0
5:         for k ← i, ..., j do
6:             sum ← sum + A[k]
7:         if sum > opt then
8:             opt ← 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 1.2.2 BetterBruteForce(A)

```
opt ← 0
for i ← 1, ..., n do
    sum ← 0
    for j ← i, ..., n do
        sum ← sum + A[j]
        if sum > opt then
            opt ← sum
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 $A[n/2+1]$.

Algorithm 1.2.3 DivideAndConquer(A)

```
if n = 1 then return max(A[1], 0)
opt_lo ← DivideAndConquer(A[1..n/2])
opt_hi ← DivideAndConquer(A[n/2 + 1..n])
function MaximizeLowerHalf()
    opt ← A[n/2]
    sum ← A[n/2]
    for i ← n/2 − 1, ..., 1 do
        sum ← sum + A[i]
        if sum > opt then opt ← sum
    return opt
function MaximizeUpperHalf()
    ...
opt_mid ← MaximizeLowerHalf() + MaximizeUpperHalf()
return max(opt_lo, opt_hi, opt_mid).
```

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 $1, \ldots, n$. 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,

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

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 1.2.4 DynamicProgramming(A)

```
M̄ ← A[1]
M ← max(M̄, 0)
for i = 2, ..., n do
    M̄ ← A[i] + max(M̄, 0)
    M ← max(M, M̄)
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 (*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 $cn^y \Theta(r^j) \in \Theta(n^x)$. □

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 (*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 2.1.1. 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 2.1.1 Modified MERGE($A[1..n]$) (additions in green)

Require: both halves of A are sorted

```
copy A into a new array S; c = 0
i ← 1; j ← n/2 + 1
for k ← 1, ..., n do
    if i > n/2 then A[k] ← S[j++]
    else if j > n then
        A[k] ← S[i++]
        c ← c + n/2
    else if S[i] < S[j] then
        A[k] ← S[i++]
        c ← c + j − (n/2 + 1)
    else A[k] ← 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 2.2 (*polynomial multiplication*)

Given $F = f_0 + \cdots + f_{n-1}x^{n-1}$ and $G = g_0 + \cdots + g_{n-1}x^{n-1}$, 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 2.1.1 (*Karatsuba's identity*)

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

Lecture 4 (05/18)

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.

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 2.3 (*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: $A = \left(\begin{array}{c|c} A_{1,1} & A_{2,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right)$ and $B = \left(\begin{array}{c|c} B_{1,1} & B_{2,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right)$.

Then, $C = \left(\begin{array}{c|c} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ \hline A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{array}\right)$

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 2.4 (*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 2.5 (*selection*)

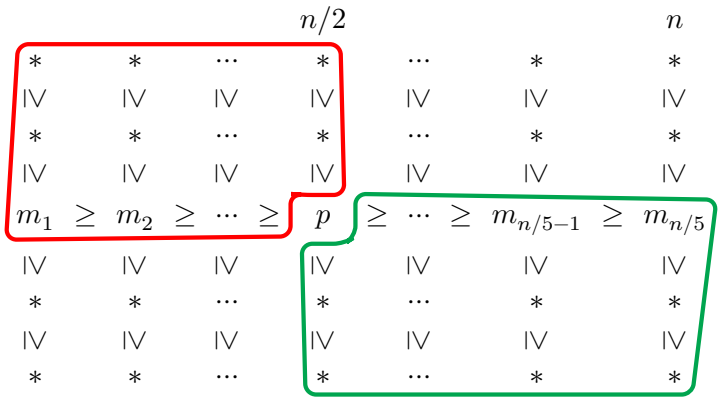
> Given $A[0..n-1]$, find the entry that would be at index k if A were sorted.

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

Algorithm 2.1.2 QuickSelect(A, k)

```
1: p ← ChoosePivot(A)
2: i ← Partition(A, p)                    ▷ i is the correct index of p
3: if i = k then return A[i]
4: else if i > k then return QuickSelect(A[0..i − 1], k)
5: else return 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, $T(n) \leq T(n/5) + T(7n/10) + O(n)$.

Lecture 5 (05/25)

Claim 2.1.1. $T(n/5) + T(7n/10) + O(n) \in O(n)$

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 2.1.2. 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 (*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 $m = |E|$.

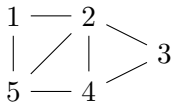
Now, we can define some structures on a graph:

Definition (*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 (*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 3.1.1. Write the adjacency list and matrix for 

Solution. The adjacency list is:

$1 \to 2 \to 5$
$2 \to 1 \to 3 \to 4 \to 5$
$3 \to 2 \to 4$
$4 \to 2 \to 3 \to 5$
$5 \to 1 \to 2 \to 4$

and the 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 (*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 3.1

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

Algorithm 3.2.1 BFS(G, s)

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

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 3.2.1

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, $s \rightsquigarrow v_i$. □

Lecture 6 (05/30)

Exercise 3.2.1. For a connected graph, $m \geq n - 1$.

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. □

3.3 Shortest Path by BFS

Problem 3.2

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 3.3.1 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 ⊥
3:  let level be an array of size n with all entries set to ∞
4:  ENQUEUE(s, Q)
5:  parent[s] ← s
6:  level[s] ← 0
7:  while Q is not empty do
8:      v ← DEQUEUE(Q)
9:      for w neighbours of v do
10:         if parent[w] = ⊥ then
11:             ENQUEUE(w, Q)
12:             parent[w] ← v
13:             level[w] ← level[v] + 1
```

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

Claim 3.3.1. The BFS tree T is in fact a tree.

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

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

Suppose T is a tree and we are adding $\mathsf{parent}[w] \leftarrow v$. 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 $\mathsf{parent}[w] = \perp$, so T remains a tree.

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

Claim 3.3.2. The levels in the queue Q are non-decreasing.

Proof. Exercise (TODO). □

Claim 3.3.3. For all vertices u and v, if there is an edge $\{u, v\}$, then $\mathsf{level}[v] \leq \mathsf{level}[u] + 1$.

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

If we dequeue v before u, then $\mathsf{level}[v] \leq \mathsf{level}[u] + 1$ by Claim 3.3.2.

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 3.3.2, $\mathsf{level}[\mathsf{parent}[v]] \leq \mathsf{level}[u]$.

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

Lemma 3.3.1

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 $\mathsf{level}[v]$ is the distance from s to t.

Proof. By Lemma 3.2.1, $s \rightsquigarrow_G v$ 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 $\mathsf{level}[v] \leq \delta$ and $\delta \leq \mathsf{level}[v]$.

Trivially, $\delta \leq \mathsf{level}[v]$ because $\mathsf{level}[v]$ 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 $\mathsf{level}[v] \leq i$. 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 $\mathsf{level}[u] \leq i-1$. Then, by Claim 3.3.3, $\mathsf{level}[v] \leq \mathsf{level}[u] + 1 \leq i$.

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

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

3.4 Bipartiteness by BFS

Definition (*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 3.3

Is G bipartite?

Lemma 3.4.1

Suppose G is connected and we run 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.

Lecture 7 (06/01)

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 $W_2 = V_2$. □

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

```
visited ← array of size n initialized to ⊥
procedure EXPLORE(v)
    visited[v] ← ⊤
    for w neighbour of v do
        if visited[v] = ⊥ then
            EXPLORE(v)
for v ∈ G do
    if visited[v] = ⊥ then
        EXPLORE(v)
```

Lemma 3.5.1 (*white path lemma*)

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

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

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

If 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 EXPLORE(v) is finished.

Therefore, v_{i+1} is visited during EXPLORE(v), as desired. □

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

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

Let $T_1, \ldots, T_k$ 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 3.5.2 (*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. □

Definition (*back edge*)

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

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

Proof. Equivalent statement of the 3.5.2. □

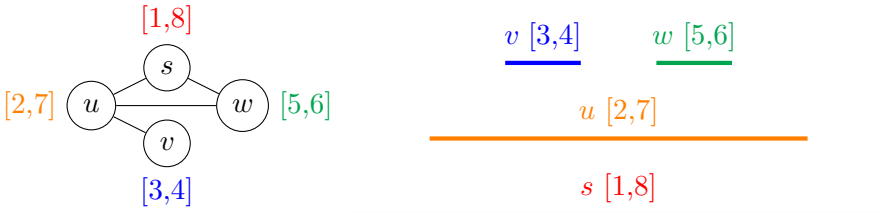
We can extend DFS with start and finish arrays:

Algorithm 3.5.2 DFS(G) with timing

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

```
visited ← array of size n initialized to ⊥
start, finish ← array of size n
t ← 1
procedure EXPLORE(v)
    visited[v] ← ⊤
    start[v] ← t; t++
    for w neighbour of v do
        if visited[v] = ⊥ then
            EXPLORE(v)
    finish[v] ← t; t++
for v ∈ G do
    if visited[v] = ⊥ then
        EXPLORE(v)
```

For example, we can draw a graph with $[\mathsf{start}[v], \mathsf{finish}[v]]$ 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 3.5.3 (*parentheses theorem*)

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

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

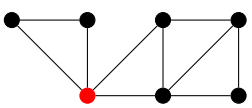
3.6 Cut Vertices by DFS

Lecture 8 (06/06)

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

Definition (*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 3.6.1.  has a cut vertex in red.

Problem 3.4

Which of the vertices in G are cut vertices?

Consider a rooted DFS tree T with known parent and level.

Proposition 3.6.1

> 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 $T - s$. □

Proposition 3.6.2

> Let $a(v) = \min\{\mathsf{level}[w] : vw \in E(G)\}$ 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 $m(w) \geq \mathsf{level}[v]$.

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

Suppose $m(w) < \mathsf{level}[v]$ 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 $m(w) \geq \mathsf{level}[v]$.

Suppose $m(w) \geq \mathsf{level}[v]$. 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. □

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 3.6.1 FindCutVertices(G, s)

```
1:  T ← DFS(G, s)                          ▷ DFS tree for G with root and level
2:  a, m ← arrays of size |V(G)| initialized to ∞
3:  cut ← array of size |V(G)| initialized to ⊥
4:  procedure Explore(v)
5:      for w child of v do
6:          a[v] ← min{a[v], level[w]}
7:          Explore(w)
8:          m[v] ← min{m[v], a[w]}
9:      m[v] ← min{a[v], m[v]}
10:     for w child of v do
11:         if m[w] ≥ T.level[v] then cut[v] ← ⊤
12: Explore(T.root)
```

3.7 Directed Graphs

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

Definition (*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 (*topological ordering*)

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

Lecture 9 (06/08)

Proposition 3.7.1

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

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 $v < w \iff \mathsf{finish}[w] < \mathsf{finish}[v]$, 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, $\mathsf{finish}[w] < \mathsf{start}[v] < \mathsf{finish}[v]$.

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

Definition (*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. 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 $w \rightsquigarrow s \rightsquigarrow v$. □

Problem 3.5

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. □

Proposition 3.7.2

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. □

Lecture 10 (06/13)

Problem 3.6

What are the strongly connected components and their respective DAG?

Algorithm 3.7.1 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^⊤)
5:     return the trees in the DFS forest of G^⊤
```

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

Proposition 3.7.3

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).

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 $\mathsf{start}[s] \leq \mathsf{start}[t] < \mathsf{finish}[t] \leq \mathsf{finish}[s]$. Since $\mathsf{finish}[u] < \mathsf{finish}[s]$, 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

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 3.7.4

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

Proof. Let G have a Hamiltonian path $P = v_1 \cdots v_k$. Define an ordering $v_1 < \cdots < v_k$.

Suppose $v_iv_j \in E(G)$. If $i > j$, then $v_iv_jv_{j+1} \cdots v_i$ 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 $v_1 < \cdots < v_k$ with $v_iv_{i+1} \in E(G)$ for all i. Then, we immediately get a Hamiltonian path given by $v_1 \cdots v_k$. □

Chapter 4

Greedy Algorithms

4.1 Introduction

Lecture 11 (06/15)

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.

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.

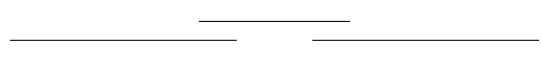
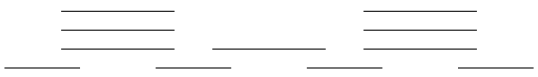
For example, recall the Huffman encoding from CS 240. We build the binary code tree by joining trees with the least frequencies. This actually minimizes the length of the encoding.

4.2 Basic Greedy Examples

Problem 4.1 (*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 4.2.1 IntervalScheduling($I = [[s_1, f_1], \dots, [s_n, f_n]]$)

```
1: S ← ∅
2: I ← sort I by finish time
3: for [s_i, f_i] ∈ I do
4:     if [s_i, f_i] has no conflicts in S then
5:         S ← S ∪ {[s_i, f_i]}
```

Proposition 4.2.1

Suppose O is optimal. Then, $|S| = |O|$ where S is generated by Algorithm 4.2.1.

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.

Lecture 12 (06/20)

We can also greedily solve a similar problem:

Problem 4.2 (*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 4.2.2 IntervalColouring($I = [[s_1, f_1], \dots, [s_n, f_n]]$)

```
c ← empty colouring
I ← sort I by start time
for [s_i, f_i] ∈ I do
    c(i) ← minimum c such that there are no conflicts
```

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

Proposition 4.2.2

Suppose that Algorithm 4.2.2 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. □

Problem 4.3 (*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 4.2.1. For $n = 5$ and $\mathbf{p} = [2, 8, 1, 10, 5]$, what is the total processing time if we do not order the jobs and if we order the jobs by increasing processing time?

Solution. Construct tables:

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

1	2	5	8	10	Σ
1	1	1	1	1	5
	2	2	2	2	8
		5	5	5	15
			8	8	16
				10	10
1	3	8	16	26	54

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

Proposition 4.2.3

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
$p_{e(1)}$	$p_{e(1)}$		$p_{e(1)}$	$p_{e(1)}$		$p_{e(1)}$	$np_{e(1)}$
	$p_{e(2)}$		$p_{e(2)}$	$p_{e(2)}$		$p_{e(2)}$	$(n-1)p_{e(2)}$
			⋮	⋮		⋮	⋮
			$p_{e(i)}$	$p_{e(i)}$		$p_{e(i)}$	$(n-i+1)p_{e(i)}$
				$p_{e(i+1)}$		$p_{e(i+1)}$	$(n-i)p_{e(i+1)}$
						⋮	⋮
						$p_{e(n)}$	$p_{e(n)}$

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. □

4.3 Shortest Paths: Dijkstra's Algorithm

4.4 Minimum Spanning Trees: Kruskal's Algorithm

Lecture 14 (06/22)

Lecture 15 (06/29)

Chapter 5

Dynamic Programming

5.1 Introduction

Lecture 16 (07/04)

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

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 ← [0, 1, ⊥, ... , ⊥] global array of size n
1: if T[n] = ⊥ then
2:     T[n] ← 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

Algorithm 5.1.3 FIBITERATIVE(n)

```
1: T ← [0, 1, ⊥, ... , ⊥] 0-indexed array of size n
2: for i = 2, ... , n do
3:     T[i] ← T[i − 1] + T[i − 2]
4: return T[n]
```

dynamic programming 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)

```
(u, v) ← (0, 1)
for i = 2, ..., n do
    (u, v) ← (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 5.1 (*weighted interval scheduling*)

Recall Problem 4.1. 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 5.2.1. 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 5.2.1 FindJs$(A, s_1, \dots, s_n, f_1, \dots, f_n)$

```
j ← array of size n
f_0 ← ∞
i ← 1
for k = 0, ..., n do
    while i ≤ n and f_k ≤ s_{A[i]} < f_{k+1} do
        j[i] ← k
        i++
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 5.2.2 IntervalScheduling$(s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n)$

Require: intervals are sorted by finish time

```
A ← sorting permutation of s_1, ..., s_n
j ← FindJs(A, s_1, ..., s_n, f_1, ..., f_n)
W ← 0-indexed array of size n
W[0] ← 0
for i = 1, ..., n do
    W[i] ← max{W[i − 1], w_i + W[j[i]]}
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 5.2 (*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 5.3.1. $\mathbf{w} = [3, 4, 6, 5]$, $\mathbf{v} = [2, 3, 1, 5]$, $W = 8$.

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

Lecture 17 (07/06)

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 $O[W, n-1]$.

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 $O[W - w_n..W, n-1]$. 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 5.3.1 01KnapSack$(v_1,\dots,v_n,w_1,\dots,w_n,W)$

```
O ← 0-indexed array of size (n+1) × (W+1)
O[0,] ← 0; O[,0] ← 0^⊤
for i = 1,...,n do
    for w = 1,...,W do
        if w_i > w then
            O[w,i] ← O[w,i−1]
        else
            O[w,i] ← max{v_n + O[W−w_n, n−1], O[W,n−1]}
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 5.3 (*longest increasing subsequence*)

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

Example 5.4.1. 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 5.4.1 LongestIncreasingSubsequence$(A[1..n])$

```
L ← array of size n
L[1] ← 1
for i = 2,...,n do
    L[i] ← 1                                   ▷ S_i = [A[i]]
    for j = 1,...,i−1 do                       ▷ S_i = [... S_j, A[i]]
        if A[j] < A[i] then
            L[i] ← max{L[i], L[j]+1}
return max L
```

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 (*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 5.4.2. For $A = \mathsf{blurry}$ and $B = \mathsf{burger}$, we should return burr for $k = 4$.

As with Problem 5.2, 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 $M[0,]$ and $M[, 0]$. 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 5.4.2 LongestCommonSubsequence($A[1..n], B[1..m]$)

```
M ← 0-indexed array of size n + 1 × m + 1
M[0, ] ← 0; M[, 0] ← 0^⊤
for i = 1, ..., n do
    for j = 1, ..., m do
        M[i, j] ← max{M[i, j − 1], M[i − 1, j]}
        if A[i] = B[j] then
            M[i, j] ← max{M[i, j], 1 + M[i − 1, j − 1]}
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.

List of Problems

List of Named Results

Index of Defined Terms