CS 341 Spring 2023:
Lecture Notes

1 Introduction **2**
- 1.1 Asymptotic Review 2
- 1.2 Types of Algorithms 3
- 1.3 Recurrence Relations 5

2 Divide and Conquer **7**
- 2.1 Examples 7

3 Breadth First Search **12**
- 3.1 Graph Theory Review 12
- 3.2 Breadth-First Exploration of a Graph 13

Back Matter **14**
- List of Problems 14
- List of Named Results 14
- Index of Defined Terms 16

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) 2
Lecture 2 (05/11) 3
Lecture 3 (05/16) 6
Lecture 4 (05/18) 8
Lecture 5 (05/25) 10

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 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)}$ 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, $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

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, $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ 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 1.2.1 (*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 1.2.1.1 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 1.2.1.2 BetterBruteForce(A)

```
1: opt ← 0
2: for i ← 1, … , n do
3:     sum ← 0
4:     for j ← i, … , n do
5:         sum ← sum + A[j]
6:         if sum > opt then
7:             opt ← 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 $A[n/2+1]$.

Algorithm 1.2.1.3 DivideAndConquer(A)

```
1: if n = 1 then return max(A[1], 0)
2: opt_lo ← DivideAndConquer(A[1..n/2])
3: opt_hi ← DivideAndConquer(A[n/2 + 1..n])
4: function MaximizeLowerHalf()
5:     opt ← A[n/2]
6:     sum ← A[n/2]
7:     for i ← n/2 − 1, … , 1 do
8:         sum ← sum + A[i]
9:         if sum > opt then opt ← sum
10:    return opt
11: function MaximizeUpperHalf()
12:    …
13: opt_mid ← MaximizeLowerHalf() + MaximizeUpperHalf()
14: 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 $i \le n-1$ or exactly $[A[n]]$.

Algorithm 1.2.1.4 DynamicProgramming(A)

```
1: M̄ ← A[1]
2: M ← max(M̄, 0)
3: for i = 2, … , n do
4:     M̄ ← A[i] + max(M̄, 0)
5:     M ← max(M, M̄)
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 (*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 ith 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 jth 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 2.1.1 (*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 2.1.1.1 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 ← 1; j ← n/2 + 1
3: for k ← 1, …, n do
4:     if i > n/2 then A[k] ← S[j++]
5:     else if j > n then
6:         A[k] ← S[i++]
7:         c ← c + n/2
8:     else if S[i] < S[j] then
9:         A[k] ← S[i++]
10:        c ← c + j − (n/2 + 1)
11:    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 2.1.2 (***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 $F = F_0 + F_1 x^{n/2}$ and $G = G_0 + G_1 x^{n/2}$. 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 (*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 2.1.3 (***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 $\Theta(n^{3 \log_{\ell m p} k})$ time.

Problem 2.1.4 (*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 2.1.5 (*selection*)

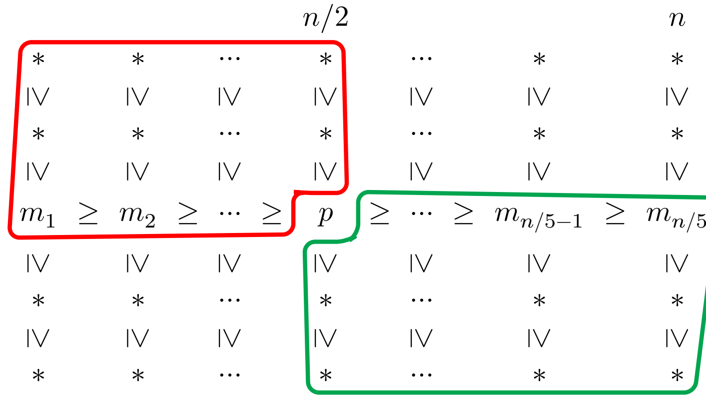
> Given $A[0..n-1]$, find the entry that would be at index k if A were sorted.

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 2.1.5.1 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, $T(n) \le T(n/5) + T(7n/10) + O(n)$.

Lecture 5 (05/25)

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

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

We will show that $T(n) \le cn$ for large enough c and all $n > 0$. We know that there exists a sufficiently large c such that $T(n) \le cn$ 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,

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

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

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

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

Therefore, we can say that $T(n) \le cn$, 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 $O(n \log n)$. $\square$

Chapter 3

Breadth First Search

3.1 Graph Theory Review

Recall graph theory from MATH 239, specifically:

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 Exploration of a Graph

Problem 3.2.1

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

Algorithm 3.2.1.1 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

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, $s \rightsquigarrow v_i$. □

List of Problems

1.2.1 Problem (contiguous subarrays) 3

2.1.1 Problem (counting inversions) 7
2.1.2 Problem (polynomial multiplication) 8
2.1.3 Problem (matrix multiplication) 8
2.1.4 Problem (closest pairs) 9
2.1.5 Problem (selection) 9

List of Named Results

1 Theorem (master theorem) 5

1 Lemma (Karatsuba's identity) 8

Index of Defined Terms

adjacency list, 12
adjacency matrix, 12

big-Ω, 2
big-Θ, 2
big-O, 2

connected component, 13
connected graph, 13
cycle, 13

edges, 12

galactic algorithm, 3
graph, 12

inversion, 7

little-ω, 2
little-o, 2

path, 13

root, 13
rooted tree, 13
runtime
- average, 2
- of an instance, 2
- worst-case, 2

sloppy recurrence, 5
subgraph, 13

tree, 13

vertices, 12