

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
	Back Matter	10
	List of Problems	10
	List of Named Results	10
	Index of Defined Terms	12

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

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

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

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

¹As 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, $2^{n-1} \in \Theta(2^n)$. \square

Exercise 1.1.2. 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 (*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.

*Lecture 2
(05/11)*

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 $A[i] + \dots + A[j]$.

Consider the brute-force attempt

Algorithm 1.2.1.1 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 1.2.1.2 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 $A[n/2 + 1]$.

Algorithm 1.2.1.3 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  $\max(opt_{lo}, opt_{hi}, opt_{mid})$ 

```

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,

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

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 1.2.1.4 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 (*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

Problem 2.0.1 (*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 2.0.1. 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 2.0.1.1 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:  $i \leftarrow 1$ ;  $j \leftarrow n/2 + 1$ 
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  $A[k] \leftarrow S[j++]$ 

```

which will run in $\Theta(n \log n)$ time because the modified MERGE is still $\Theta(n)$.

Problem 2.0.2 (*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 (*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:

Lecture 4
(05/18)

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

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 2.0.3 (*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: $A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$ and $B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$.

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

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 2.0.4 (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 2.0.5 (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 2.0.5.1 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 QUICKSELECT( $A[0 : i - 1], k$ )
5: else return QUICKSELECT( $A[i + 1 : n - 1], k - i - 1$ )

```

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

Then, we can show that there are at least $3n/10$ elements below and above p , so we can cap the recursive calls at $7n/10$ size.

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

List of Problems

1.2.1 Problem (contiguous subarrays)	3
2.0.1 Problem (counting inversions)	7
2.0.2 Problem (polynomial multiplication)	8
2.0.3 Problem (matrix multiplication)	8
2.0.4 Problem (closest pairs)	9
2.0.5 Problem (selection)	9

List of Named Results

1	Theorem (master theorem)	5
1	Lemma (Karatsuba's identity)	8

Index of Defined Terms

big- Ω , [2](#)

big- Θ , [2](#)

big- O , [2](#)

galactic algorithm, [3](#)

inversion, [7](#)

little- ω , [2](#)

little- o , [2](#)

runtime

average, [2](#)

of an instance, [2](#)

worst-case, [2](#)

sloppy recurrence, [5](#)