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.

Chapter 1

Introduction

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. 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 1.2. 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 $1 << n$.

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.

Exercise 2.1. 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 2.1 BruteForce(A)

```
opt ← 0
for i ← 1, …, n do
    for j ← i, …, n do
        sum ← 0
        for k ← i, …, j do
            sum ← sum + A[k]
        if sum > opt then
            opt ← sum
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 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 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 $1, \dots, n$. 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 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).

Chapter 2

Solving Recurrences

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,

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

□

Index of Defined Terms