Algorithms Lecture Notes
UC Berkeley

Kelvin Lee

September 4, 2020

These are course notes for UC Berkeley's CS170 Efficient Algorithms and Intractable Problems, instructed by Professor Avishay Tal and Umesh Vazirani.

Contents

§1 Introduction

§1.1 Asymptotic Notation

Let $f, g : \mathbb{R} \to \mathbb{R}$ for the following definitions.

§1.1.1 O Notation

The O notation describes upper bounds for function.

Definition 1.1 (O Notation)
If there exists a constant $c > 0$ and N such that for $x > N, |f(x)| \leq c|g(x)|$, we say that

$$f(x) = \mathcal{O}(g(x)).$$

§1.1.2 Ω Notation

The Ω notation describes lower bounds for functions.

Definition 1.2 (Ω Notation)
If there exists a constant $c > 0$ and N such that for $x > N, |f(x)| \geq c|g(x)|$. This indicates that

$$f(x) = \Omega(g(x))$$

§1.1.3 Θ Notation

The Θ notation describes both upper and lower bounds for functions.

Definition 1.3 (Θ Notation)
If there exist constants $c_1, c_2 > 0$, and N such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for $x > N$, we have $f(x) = O(g(x)) = \Omega(g(x))$, which implies

$$f(x) = \Theta(g(x)).$$

Theorem 1.4 (Asymptotic Limit Rules)
If $f(n), g(n) \geq 0$:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = \mathcal{O}(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some $c > 0 \implies f(n) = \Theta(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \Omega(g(n))$.

§2 Divide and Conquer

Definition 2.1 (Divide and Conquer)

The **divide-and-conquer** strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

§2.1 Karatsuba's Algorithm

Suppose we are multiplying two n-bit integers, x and y. Split up x into half, x_L and x_R, so that $x = 2^{n/2}x_{\mathrm{L}} + x_{\mathrm{R}}$.

$$\begin{aligned} xy &= \left(2^{n/2}x_L + x_R\right) + \left(2^{n/2}y_L + y_R\right) \\ &= 2^n \left(x_L y_L\right) + 2^{n/2}\left(x_R y_L + x_L y_R\right) + x_R y_R \end{aligned}$$

- The additions take linear time, as do the multiplications by powers of 2 (left-shifts).
- It requires 4 multiplications on $n/2$ bit numbers, we get the recurrence relation

$$T(n) = 4T(n/2) + \mathcal{O}(n).$$

However, this can be improved by using 3 multiplications:

- only need $x_L y_L, x_R y_R$, and $(x_L + x_R)(y_L + y_R)$ because

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

- The improved running time would then be

$$T(n) = 3T(n/2) + \mathcal{O}(n).$$

- The constant factor improvement occurs at every level of the recursion, which dramatically lowers time bound of $O\left(n^{\log_2 3}\right)$.

Algorithm 2.2 Karatsuba's Algorithm

function $\textsc{Mult}(x, y)$
  $P_1 \leftarrow \mathrm{Mult}(x_L, y_R)$
  $P_2 \leftarrow \mathrm{Mult}(x_R, y_L)$
  $P_3 \leftarrow \mathrm{Mult}(x_L + x_R, y_L + y_R)$
  return $2^n P_1 + 2^{n/2}\left(P_3 - P_1 - P_2\right) + P_3$

§2.2 Master Theorem

A divide-and-conquer algorithm might be described by

$$T(n) = aT\left(\frac{n}{b}\right) + \mathcal{O}(n^d)$$

Theorem 2.3 (Master Theorem)
If $T(n) = aT(n/b) + \mathcal{O}\left(n^d\right)$ for $a > 0, b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} \mathcal{O}\left(n^d\right) & \text{if } d > \log_b a \\ \mathcal{O}\left(n^d \log n\right) & \text{if } d = \log_b a \\ \mathcal{O}\left(n^{\log_b a}\right) & \text{if } d < \log_b a \end{cases}$$

Proof. Let $T(n) = aT\left(n/b\right) + \mathcal{O}\left(n^d\right)$. For simplicity, assume that $T(1) = 1$ and that n is a power of b. From the definition of big-O, we know that there exists constant $c > 0$ such that for sufficiently large $n, T(n) \leq aT\left(n/b\right) + cn^d$.

Suppose we have a recursive tree with $\log_b n + 1$ level. Consider level j. At level j, there are a^j subproblems. Each of size $\frac{n}{b^j}$, and will take time at most $c\left(\frac{n}{b^j}\right)^d$ to solve (this only considers the work done at level j and does not include the time it takes to solve the subsubproblems). Then the total work done at level j is at most $a^j \cdot c\left(n/b^j\right)^d = cn^d\left(\frac{a}{b^d}\right)^j$, where a is the branching factor and b^d is the shrinkage in the work needed (per subproblem). Summing over all levels, the total running time is at most $cn^d \sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j$. Consider each of the three cases:

1. $(a < b^d)$: $\frac{a}{b^d} < 1$, then

$$\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty}\left(\frac{a}{b^d}\right)^j = \frac{1}{1 - \frac{a}{b^d}} = \frac{b^d}{b^d - a}.$$

   Hence, $T(n) = cn^d \cdot \frac{b^d}{b^d - a} = O\left(n^d\right)$. Intuitively, in this case the shrinkage in the work needed per subproblem is more significant, so the work done in the highest level "dominates" the other factors in the running time.

2. $(a = b^d)$: The amount of work done at each level is the same: cn^d. since there are $\log_b n$ levels, $T(n) = \left(\log_b n + 1\right) cn^d = O\left(n^d \log n\right)$.

3. $(a > b^d)$: We have

$$\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j = \frac{\left(\frac{a}{b^d}\right)^{\log_b n + 1} - 1}{\frac{a}{b^d} - 1}.$$

   Since a, b, c, d are constants,

$$T(n) = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \cdot \frac{a^{\log_b n}}{b^d \log_b n}\right) = O\left(n^d \cdot \frac{n^{\log_b a}}{n^d}\right) = O\left(n^{\log_b a}\right)$$

   Intuitively, here the branching factor is more significant, so the total work done at each level increases, and the leaves of the tree "dominate".

$\square$

§3 Matrix multiplication

§3.1 Naïve algorithm

If $A \in k^{n \times n}$ and $B \in k^{n \times n}$, then $AB \in k^{n \times n}$, where $(AB)_{ij} = A[i,:] \cdot B[:,j]$. There are n^2 entries that take n time each, so computing AB naïvely takes $O(n^3)$ time.

§3.2 Divide-and-conquer algorithm

Divide the matrices into blocks.

$$\begin{pmatrix} X_{ij} \end{pmatrix} \begin{pmatrix} Y_{ij} \end{pmatrix} = \begin{pmatrix} (XY)_{ij} \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} \tag{2}$$

Algorithm 3.1 Efficiently multiply two square matrices.

function $\textsc{Mult}(X, Y \in k^{n \times n})$
  $P_1 \leftarrow AE$
  $\ddots$ (6 steps omitted; see algebra above)
  $P_8 \leftarrow DH$
  return $\begin{pmatrix} P_1 + P_2 & P_3 + P_4 \\ P_5 + P_6 & P_7 + P_8 \end{pmatrix}$

Runtime analysis:

$$T[n] = 8T\left[\frac{n}{2}\right] + O(n^2) \tag{3}$$

Apply Master Theorem.

$$= O(n^3) \tag{4}$$

It turns out there's a way to end up with the results of those 8 submatrix multiplications using only 7 multiplications.

$$T[n] = \mathbf{7}T\left[\frac{n}{2}\right] + O(n^2) \tag{5}$$

$$= O(n^{2.81})? \tag{6}$$

§4 Median

The median of an n-set of numbers is the $n/2$th-largest number.

§4.1 Naïve algorithm

Sort the list. $O(n \log n)$.

§4.2 Divide-and-conquer algorithm

Dividing into two lists and computing their medians doesn't always get you the median.

It is helpful to generalize median to selection.

Algorithm 4.1 Attempt at a divide-and-conquer algorithm for finding the median of a list of numbers n items long.

function MEDIAN($S = \{a_1, \dots, a_n\}$)
  Choose $v \in S$
  $S_L \leftarrow \{a \in S \mid a < v\}$
  $S_V \leftarrow \{a \in S \mid a = v\}$
  $S_R \leftarrow \{a \in S \mid a > v\}$
  return ?? to be continued

Algorithm 4.2 Select kth smallest number in a set.

function SELECT($S = \{a_1, \dots, a_n\}$, k)
  Choose $v \in S$
  $S_L \leftarrow \{a \in S \mid a < v\}$
  $S_V \leftarrow \{a \in S \mid a = v\}$
  $S_R \leftarrow \{a \in S \mid a > v\}$
  if $|S_L| > k$ **then**
    return Select(S_L, k)
  if $k > |S_L|$ and $k \leq |S_L| + |S_V|$ **then**
    return v
  if $k > |S_L| + |S_V|$ **then**
    return Select($S_R, k - |S_L| - |S_V|$)

§4.3 Runtime analysis

In the worse case, this algorithm chooses an extreme every single call. So $\Theta(n)$ times you have to separate $\Theta(n)$ items and the runtime is $\Theta(n^2)$.

How shall we judge the average case? (It is not interesting to consider only the best case, which is $\Theta(1)$.) Instead, call pivots *good* which are within $n/4$ places of the median. The following inequalities follow:

$$|S_L| \leq \frac{|S|}{4} \tag{7}$$

$$|S_R| \leq \frac{|S|}{4} \tag{8}$$

A "good" pivot reduces the problem size by $\frac{3}{4}$ per stack frame.

Now to consider probability. Any given step,

$$\Pr\left(v \text{ is a good pivot}\right) = \frac{1}{2} \tag{9}$$

Rewrite the runtime expression.

$$\begin{aligned} T[n] &= \text{time after first good pivot} \\ &\quad + \text{time until first good pivot} && (10) \\ &\leq T\left[\frac{3}{4}n\right] + n\mathrm{E}\left[\#\text{ of pivots}\right] && (11) \end{aligned}$$

"trust me, this is 2."

$$\begin{aligned} &\leq T\left[\frac{3}{4}n\right] + 2n && (12) \\ &\leq O(n) && (13) \end{aligned}$$

Algorithm 4.3 Put your caption here

```
procedure ROY(a, b)                              ▷ This is a test
    System Initialization
    Read the value
    if condition = True then
        Do this
        if Condition ≥ 1 then
            Do that
        else if Condition ≠ 5 then
            Do another
            Do that as well
        else
            Do otherwise
    while something ≠ 0 do                       ▷ put some comments here
        var1 ← var2                              ▷ another comment
        var3 ← var4
```