Algorithms

Andi Gu

May 2018

1 Algorithm Analysis

We analyze our programs via three different functions:

- Big-O notation. If we say that a program has $O(f(n))$, this means that above a certain n_0, we can find a constant c such that $cf(n)$ is greater than the number of basic operations the program must execute for all $n > n_0$. This gives us an 'upper bound' on the program's running time.
- Big-Omega notation: If we say that a program has $\Omega(f(n))$, this means that above a certain n_0, we can find a constant c such that $cf(n)$ is less than the number of basic operations the program must execute for all $n > n_0$. This gives us an 'lower bound' on the program's running time.
- Big-Theta notation: If we say that a program has $\Theta(f(n))$, this means that above a certain n_0, we can find two constants c_1 and c_2 such that the number of basic operations the program must execute is **between** $c_1 f(n)$ and $c_2 f(n)$ for all $n > n_0$. This gives us an 'average' for the program's running time.

Often the proofs that a certain function has $O(f(n))$ are quite straightforward.

Theorem. *Let $g(n) = \sum_{i=0}^{m} a_i n^i$. If g is upper bounded by $O(f(n))$, we can say that $f(n) = n^{m+1}$.*

Proof. We set $cf(n)$ to be greater than $g(n)$ and see if the inequality holds above a certain n_0:

$$cn^{m+1} > \sum_{i=0}^{m} a_i n^i$$

We prove an even stronger theorem by changing increasing all n^i to n^m since $n^m \geq n^i$ for all $n > 1$.

$$cn^{m+1} > \sum_{i=0}^{m} a_i n^m$$

$$cn^{m+1} > n^m \sum_{i=0}^{m} a_i$$

$$cn > \sum_{i=0}^{m} a_i$$

The sum of all the coefficients a_i is simply a constant.

Now, we see if we set c to $1 + \sum_{i=0}^{m} a_i$, for all $n > 1$, $cf(n) > g(n)$. □

2 Divide and Conquer

The divide and conquer paradigm handles problems by breaking them down into smaller subproblems, solving those recursively, and cleverly recombining the results to solve the larger problem.

2.1 Karatsuba Multiplication

We study the problem of multiplying two n-digit numbers. The grade school method is $O(n^2)$, which can be easily verified. We try to obtain a better running time by splitting the two numbers as follows:

$$x = \underbrace{x_1 x_2 \dots x_{n/2}}_{a} \underbrace{x_{n/2+1} x_{n/2+2} \dots x_n}_{b}$$

$$y = \underbrace{y_1 y_2 \dots y_{n/2}}_{c} \underbrace{y_{n/2+1} y_{n/2+2} \dots y_n}_{d}$$

$$x \cdot y = a \cdot c \cdot 10^n + a \cdot d \cdot 10^{n/2} + b \cdot c \cdot 10^{n/2} + b \cdot d$$

But, if we make a clever observation, we can see that $a \cdot d + b \cdot c = (a + b) \cdot (c + d) - a \cdot c - b \cdot d$. So in reality, we are concerned with only three products:

$$p_1 = a \cdot c$$

$$p_2 = b \cdot d$$

$$p_3 = (a + b) \cdot (c + d) - p_1 - p_2$$

Actually, this algorithm can be generalized to numbers with any number of digits. Without loss of generality, if x has n_1 digits and y has n_2 digits (where $n_1 > n_2$), we simply pad y with leading 0's until it has n_1 digits as well. The psuedocode looks something like:

Algorithm 1 Karatsuba Multiplication

function MULTIPLY(x, y)
  $n \leftarrow \max(\text{len}(x), \text{len}(y))$
  if $n = 1$ **then**
    return $x * y$
  else
    $px, py \leftarrow \text{pad}(x, n), \text{pad}(y, n)$ ▷ Pads to n digits with leading 0's
    $nby2 \leftarrow \text{int}(n/2)$
    $a, b \leftarrow \text{split}(x, nby2)$ ▷ First half gets $0 : nby2$ inclusive
    $c, d \leftarrow \text{split}(y, nby2)$
    $p_1 \leftarrow$ MULTIPLY(a, c)
    $p_2 \leftarrow$ MULTIPLY(b, d)
    $p_3 \leftarrow$ MULTIPLY$(a + b, c + d) - p_1 - p_2$
    return $p_1 * 10^{nby2*2} + p_3 * 10^{nby2} + p_2$
  end if
end function

2.2 Counting Inversions

It is often useful to find the number of element pairs that are 'out of order' in an array. More concretely, if we have two elements in an array A at index i and j, where $i < j$ and $A[i] > A[j]$, we say the two are inverted. If we want the total number of inversions, we might take a naive approach and loop through every possible element pairing, which is $O(n^2)$ time. However, it is possible to get a better running time by clever divide and conquer.

The basic idea is that we split the array into two halves and classify inversions into three separate cases: left inversions, right inversions, and split inversions. Left inversions are inversions between elements that are purely members of the left half of the array, right inversions are the same for the right half, and split inversions are inversions **across** the two halves.

Algorithm 2 Inversion Counting

```
function countInversions(A)
    n ← len(A)
    if n = 1 then
        return A, 0
    else
        l, inv_l ← countInversions(A[: n/2])
        r, inv_r ← countInversions(A[n/2 :])
        i, j ← 0, 0
        inv ← inv_l + inv_r
        for k ← 1, n do
            if j > n/2 or (i < n/2 and l[i] ≤ r[j]) then
                A[k] ← l[i]
                i ← i + 1
            else
                A[k] ← r[j]
                j ← j + 1
                inv ← inv + n/2 − i          ▷ Elements right of i in l are inverted
            end if
        end for
    end if
end function
```

2.3 The Master Method

We have not analyzed the running time of the above algorithms yet, but it turns out that we have a general rule for determining the running time of any divide and conquer algorithm. If we denote running time T as a function of n, our general recurrence will look like:

$$T(n) = a \cdot T\Big(\frac{n}{b}\Big) + cn^d$$

We begin an intuitive analysis by seeing that there are two 'forces' in this recurrence – one is the bad force, increasing the number of subproblems per level. This is represented by a. The 'good force' is b, which represents splitting up the problem into smaller and smaller pieces.

We proceed in our formal analysis. If we call the root of the recurrence tree level 0, we see that level i will have a^i subproblems each with size $\frac{n}{b^i}$. So the running time of the level itself (without counting the recursive calls it may spawn) is:

$$a^i c\Big(\frac{n}{b^i}\Big)^d = cn^d \cdot \Big(\frac{a}{b^d}\Big)^i$$

Since there are at most $\log_b n$ levels, we sum from $i = 0$ to $i = \log_b n$:

$$T(n) = \sum_{i=0}^{\log_b n} cn^d \cdot \Big(\frac{a}{b^d}\Big)^i$$

For brevity, we call the ratio $\frac{a}{b^d}$ the simple r.

$$= cn^d \sum_{i=0}^{\log_b n} r^i$$

Now, we analyze three separate cases. If $r < 1$, we see that the sum is upper-bounded by $\frac{1}{1-r}$. This is because if the sum went to $i = \infty$, the sum would be exactly $\frac{1}{1-r}$. So it is bounded by some constant, which means that the running time when $r < 1$ is on the order of n^d. When $r = 1$, the running time is simply $n^d \log_b n$ – however, we drop the base since this results in a constant, and simply write it as $n^d \log n$.

Finally, when $r > 1$, we reverse the scenario so that i starts at $-\infty$ and runs to $\log_b n$, and the result of this sum still upper bounds the sum we have in the expression. This larger sum from $i = -\infty$ to $i = \log_b n$ is $\frac{k^{\log_b n}}{1-1/r}$. However, we drop the constants and are left with $r^{\log_b n}$. There is some simplifying to be done here:

$$\begin{aligned} n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n} &= n^d \cdot b^{-d \log_b n} a^{\log_b n} \\ &= n^d \cdot n^{-d} \cdot a^{\log_b n} \\ &= a^{\log_a n / \log_a b} \\ &= n^{1/\log_a b} \\ &= n^{\log_b a} \end{aligned}$$

So we have finally arrived at our master method for three different cases of k:

$$O(n) = \left\{ \begin{array}{ll} n^d, & \text{for } \frac{a}{b^d} < 1 \\ n^d \log n, & \text{for } \frac{a}{b^d} = 1 \\ n^{\log_b a}, & \text{for } \frac{a}{b^d} > 1 \end{array} \right\}$$

The intuitive explanation for this is that for $\frac{a}{b^d} < 1$, the bulk of the work is done at the **root**, so the running time depends on n^d (work done at the first level). For $\frac{a}{b^d} > 1$, the bulk of the work is done at the **leaves**, which explains the term $a^{\log_b n}$ that appears before we simplify to $n^{\log_b a}$. Of course, $a^{\log_b n}$ represents the number of leaves of the recurrence tree, where each leaf is doing a constant amount of work independent of n (this is the base case).

We are now prepared to find the running time for Karatsuba multiplication and inversion counting. For Karatsuba multiplication, a was 3, b was 2, and d was 1. So this is the third case, and the complexity is thus $O(n) = n^{\log_2 3} \approx n^{1.585}$, which is better than the naive complexity of n^2. For inversion counting, this is the second case so complexity is $n \log n$.

3 Randomized Algorithms

It is often useful to introduce **randomness** into our algorithms. Although this may seem counter-intuitive, it sometimes proves to be useful.

3.1 Quicksort

We begin our study of the quick-sort algorithm. This is a divide-and-conquer algorithm, but it is a special case. It is actually randomized and has worst case running time $O(n^2)$, however, it can be shown that the algorithm runs in $\Theta(n \log n)$ time. First, a description of the algorithm.

Algorithm 3 Quicksort

```
function PARTITION(A, p l, r)
    i ← r                                  ▷ Index of left-most element larger than pivot
    swap(A[l], A[p])                       ▷ Now A[l] is the pivot element
    for j ← l + 1, r do
        if A[j] < A[l] and i ≠ r then
            swap(A[j], A[i])
            i ← i + 1
        else if A[j] > A[l] and i = r then
            i ← j
        end if
    end for
    if A[i] > A[l] then
        i ← i − 1
    end if
    swap(A[i], A[l])
    return A, i
end function

function QUICKSORT(A, l, r)
    if r > l then
        A, p ← PARTITION(A, randint(l, r), l, r)
        QUICKSORT(A, l, max(p − 1, l))
        QUICKSORT(A, min(p + 1, r), r)
    end if
    return A
end function
```

The advantages of this sort over merge sort is that it is in-place, meaning there is $O(1)$ extra memory required.

The key to the algorithm is to chose a pivot. Ideally, if we could find the median of any set of data very quickly, we would set the pivot to this element and each subproblem would be of size $n/2$. In the worst case, if we imagine we are given a sorted array and we pick the first element of the subarray every time as our pivot, our subproblem size will decrease by only one each recursive call, with a linear amount of work being done per call (in the partition step) – so overall running time will be $O(n^2)$. However, instead of cleverly selecting a pivot, it turns out we can get away with simply randomly selecting one, and we still still achieve an average running time of $\Theta(n \log n)$.

3.1.1 Analysis

We start by introducing some notation. We let z_i denote the i^{th} smallest element in the array so that z_1 represents the smallest element. We let $X_{i,j}$ be the random variable that represents the number of times the two elements z_i and z_j are compared at some point in the quicksort where $i \neq j$.

Theorem. *The variable $X_{i,j}$ can only take on 2 values: 0 and 1.*

Proof. In our algorithm, any pair of numbers are compared if and only if one of them is the pivot. However, from then on, they will never be compared again since we ignore the pivot element in further recursive calls. So in any given call of quicksort, if the pivot happens to be exactly z_i or z_j, then z_i and z_j will be compared exactly once. Then the pivot will be ignored in all further recursive calls and the two will never be compared again. On the other hand, it is possible that z_i and z_j never get compared – this occurs if the pivot's value is **between** z_i and z_j. Then the two will end up on different 'sides' of the array, and will never meet again. Thus they will have been compared 0 times. □

So if we call the number of comparisons c (which is a random variable), we can find its expected value:

$$\begin{aligned} E[c] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 0 \cdot p[X_{i,j} = 0] + 1 \cdot p[X_{i,j} = 1] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p[X_{i,j} = 1] \end{aligned}$$

So the expected number of comparisons we make in quicksort is simply the sum of probabilities that we compare any two elements exactly once. Now, our goal is to find an expression for $p[X_{i,j} = 1]$ in terms of i and j. Since we have set this up so that $i < j$, we imagine the series

$$z_i, z_{i+1}, z_{i+2}, \ldots, z_j$$

Throughout quicksort, this 'group' will stay together until one of these elements is selected as the pivot. Then we must ask what the probability that either z_i or z_j are selected as the pivot, since this means that elements z_i and z_j will be compared with one another. It is trivial to show that this probability is $\frac{2}{j-i+1}$ (the factor of 2 is due to the fact that either z_i or z_j works) if there is an equal probability of selecting any element.

Now, finding $E[c]$ is simple.

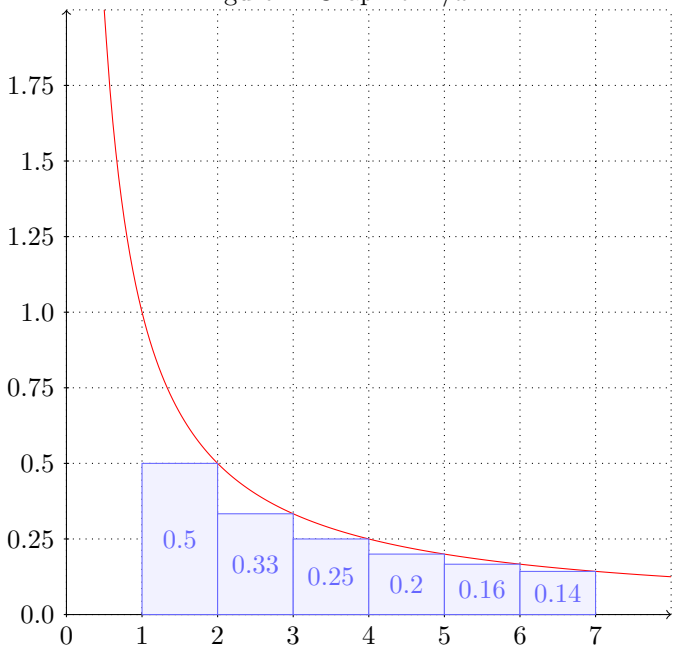
$$E[c] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

We now upper bound $E[c]$ by flattening the outer sum and essentially setting $i = 1$ for n iterations.

$$E[c] \leq 2n \sum_{j=2}^{n} \frac{1}{j}$$

Now, this sum $\sum_{j=2}^{n} \frac{1}{j}$ is actually upper bounded by $\ln x$. We can show this by imagining a graph of $\frac{1}{x}$:

Figure 1: Graph of $1/x$



So the sum is related to the area under the graph of $1/x$. In fact, the sum is upper bounded by the integral of $1/x$ from 1 to n. Finally:

$$\begin{aligned} E[c] &\leq 2n \int_1^n \frac{1}{x}\,\mathrm{d}x \\ &\leq 2n(\ln n - \ln 1) \\ &\leq 2n \ln n \end{aligned}$$

So the expected number of comparisons made is upper bounded by $2n \log n$. Since comparisons are the dominant work done in the algorithm (swaps occur at most once per comparison), the average running time of quicksort is $\Theta(n \log n)$.

3.2 Median Selection

It is also possible to use a randomized algorithm that finds the median of an array, and then using this median as the pivot, continue the quick sort. We show that a randomized algorithm for median selection will give us an average running time of $\Theta(n)$. The algorithm again makes use of the partition method. We describe a more general algorithm that can find z_i of an array (which we call the i^{th} order statistic).

Algorithm 4 Randomzied Median Selection

```
function SELECT(A, l, r, i)
    if l = r then
        return A[l]
    else
        A, p ← PARITION(A, randint(l, r), l, r)
        if p = i then                                  ▷ Got lucky and pivot is z_i
            return A[p]
        else if p < i then                             ▷ Pivot to the left of z_i
            return SELECT(A, p + 1, r, i)
        else                                           ▷ Pivot to the right of z_i
            return SELECT(A, l, p − 1, i)
        end if
    end if
end function
```

In the analysis of the running time for this algorithm, we separate its progression into 'phases'. We say that the algorithm is in phase j if the size of the subproblem it is working on is between $\left(\frac{3}{4}\right)^j n$ and $\left(\frac{3}{4}\right)^{j+1} n$ where n is the original size of the array. So phase 0 is when the size of the subarray we are looking at is between n and $\frac{3n}{4}$, phase 1 is between $\frac{3n}{4}$ and $\frac{9n}{16}$, and so on. The variable of interest here is the amount of iterations the algorithm spends in each phase j – we call this random variable k_j. Now, in given phase j, the amount of work done is upper bounded by $E[k_j] \cdot c\left(\frac{3}{4}\right)^j n$ since the work partition does is linear. Now, the expected running time is upper bounded by:

$$\sum_{j=0}^{\infty} E[k_j] \cdot c\left(\frac{3}{4}\right)^j n$$

We now show that $E[k_j]$ is 2 for all j. Here, we see that if we select a pivot that is in between the 25th and 75th percentile statistic in the array, the subproblem will be reduced by at least $\frac{3}{4}$ and we will progress to the next phase. So at worst, on any given iteration, we have a 50% chance of progressing to the next iteration. Now, we can express the expected value of k_j using this fact since the probability that we will get to the i^{th} iteration without progressing to the next phase is $\frac{1}{2^i}$. So:

$$E[k_j] = \sum_{i=1}^{\infty} i \cdot \frac{1}{2^i} = 2$$

Finally, we can solve for our expected running time:

$$\begin{aligned}
&\sum_{j=0}^{\infty} E[k_j] \cdot c\left(\frac{3}{4}\right)^j n \\
&= 2cn \sum_{j=0}^{\infty} \left(\frac{3}{4}\right)^j \\
&= 2cn \cdot \frac{1}{1-\frac{3}{4}} \\
&= 8cn
\end{aligned}$$

Thus we have shown the expected running time is $\Theta(n)$.

Now, if we apply this algorithm to our previous quicksort implementation, we can **expect** (on average) now do a linear amount of work on each recursive call, on the partition and select subroutines. However, we are now guaranteed to split the problem size in half every time since the select subroutine gives us a median. The expected running time of quicksort is still $\Theta(n \log n)$.

3.3 Graph Contraction Algorithm

A cut of a graph is defined as a partition of a graph into two non-empty subgraphs, where edges going between the two partitions are called 'crossing edges'. The minimum cut of a graph is defined as the cut in which the number of crossing edges is minimized.

We introduce an algorithm called the 'graph contraction' algorithm. The basic premise is that we select a random edge from one of the m edges in the graph. We then 'contract' this edge so that the two vertices it connected u and v are now merged into a singular vertex uv. This new vertex uv is connected to all the nodes that u and v were previously connected to, and we **allow** parallel edges (there may be more than one edge from vertex uv to another vertex w). We simply continue this process, randomly selected edges and contracting them until we have 2 vertices left. This gives us a cut of the graph – not guaranteed to be a minimum cut.

So the algorithm is not deterministic and is not certain to give us the correct answer every time. But we may ask, what is its probability of success? We examine the chance that it gets **one** of the minimum cuts. If this minimum cut has k crossing edges, the only way for the algorithm to terminate correctly is to never select any of the k edges. In this analysis, we denote the event where the algorithm chooses a non crossing on the i^{th} iteration as A_i. We can say that A_1 is $1 - k/m$. However, we will find it more useful to continue our analysis in terms of the number of **nodes** in the graph rather than number of edges. Our key observation here is that no node can have degree less than k (otherwise the cut with just this node would be the minimum cut and k would be lower). By the handshake lemma, $m \geq \frac{nk}{2}$. So now we can say:

$$\begin{aligned}
P(A_1) &\geq 1 - \frac{k}{\frac{nk}{2}} \\
&\geq 1 - \frac{2}{n}
\end{aligned}$$

The next term we are interested in is $P(A_2 \cap A_1)$ (the probability that we don't screw up on the

second iteration, given that we didn't screw up on the first one). We write

$$\begin{aligned} P(A_2 \cap A_1) &= P(A_2 \mid A_1) \cdot P(A_1) \\ &\geq \left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n}\right) \\ P(A_3 \cap A_2 \cap A_1) &= P(A_3 \mid A_1 \cap A_2) \cdot P(A_1 \cap A_2) \\ &\geq \left(1 - \frac{2}{n-2}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n}\right) \end{aligned}$$

In general now,

$$\begin{aligned} P\left(\bigcap_{i=1}^{n-3} A_i\right) &\geq \prod_{i=0}^{n-3} 1 - \frac{2}{n-i} \\ &\geq \prod_{i=0}^{n-3} \frac{n-i-2}{n-i} \end{aligned}$$

It turns out that this is a telescoping series. Only the first two denominators and the last two numerators are left after cancelling.

$$\begin{aligned} &\geq \frac{(n-(n-4)-2) \cdot (n-(n-3)-2)}{(n-0) \cdot (n-1)} \\ &\geq \frac{2}{n(n-1)} \end{aligned}$$

So the probability that the algorithm gives us the correct answer is actually quite small. However, if we analyze the algorithm when we run it a large number of times N, we see that the probability that we find the correct answer just **once** can actually be quite high. We do this by finding the probability that N calls to the algorithm fail to return the correct answer every time. We call this probability P'.

$$P' \leq \left(1 - \frac{2}{n(n-1)}\right)^N$$

We make a sloppy reduction for ease of analysis.

$$P' \leq \left(1 - \frac{1}{n^2}\right)^N$$

We use the fact that $1 + x \leq e^x$ for all x. If we set $x = -\frac{1}{n^2}$, we can reduce this to:

$$P' \leq e^{-\frac{N}{n^2}}$$

So the probability of success (finding the correct minimum cut at least once) is:

$$P \geq 1 - e^{-\frac{N}{n^2}}$$

So if we have a desired probability of success p, we can simply solve:

$$\begin{aligned} 1 - e^{-\frac{N}{n^2}} &\geq p \\ 1 - p &\geq e^{-\frac{N}{n^2}} \\ \frac{N}{n^2} &\geq -\ln(1-p) \\ N &\geq -n^2 \ln(1-p) \end{aligned}$$

If we want the probability of success to be 99%, we can set N to just $4.61n^2$.