CS 341: ALGORITHMS (S18) — LECTURE 2
BIG-O NOTATION AND REDUCTIONS

ERIC BLAIS

1. Big-O notation

We ended the first lecture with definitions for algorithms, for solving a problem, and for the model of computation (Word RAM) that we will use to measure the run time of an algorithm on a given input. The next ingredient that we need to determine the time complexity of an algorithm is a decision on how to measure this complexity *globally* instead of on a per-instance basis.

Definition 2.1 (Worst-case time complexity)**.** The *worst-case time complexity* of an algorithm A is the function $T_A : \mathbb{N} \to \mathbb{N}$ obtained by letting $T_A(n)$ be the maximum time complexity of A over any input of size n.

Remark 2.2. Worst-case is not the only way we can measure the time complexity of an algorithm as a function of its input size. (One could take the *average* time complexity over some distribution on inputs of length n instead, for example). But this is a model that is particularly useful and the one we will focus on throughout this course.

We now come to one of the central ideas in the analysis of algorithms: what we care about, when we measure the time complexity of an algorithm, is not the *exact* expression for this time complexity, but rather its *asymptotic* rate of growth as the inputs get larger. This is best done using big-O notation.

Definition 2.3 (Big-O notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 1}$ satisfy $f = O(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \leq c\, g(n)$.

Remark 2.4. Here and throughout,

- $\mathbb{N} = \{1, 2, 3, \ldots\}$ is the set of natural numbers,
- $\mathbb{R}^+$ be the set of positive real numbers, and
- $\mathbb{R}^{\geq 1}$ be the set of real numbers that have value at least 1.

Big-O notation is so useful in part because it lets us simplify even very complicated expressions and only worry about the "most significant" aspects of time complexity. For example, we have the following fact.

Proposition 2.5. *The function $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ satisfies $f = O(n^7)$.*

Proof. For every $n \geq 4$,

$$f(n) = 4n^7 + 100n^3 + \tfrac{1}{3}n^2 + \pi \leq 4n^7 + 100n^3 + n^2 + n \leq 106n^7$$

so $f = O(n^7)$ by the definition with $c = 106$ and $n_0 = 4$. □

One word of warning: since $n^7 \leq n^{100}$ for every $n \geq 1$, it is also correct to say that the function f defined in the proposition satisfies $f = O(n^{100})$. To describe asymptotics of a function more precisely, we need the big-Ω and the big-Θ notation. The big-Ω notation is used to give lower bounds on the asymptotic growth of a function.

Definition 2.6 (Big-Ω notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \Omega(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \geq c\, g(n)$.

The big-Θ notation is used to show that we have matching upper and lower bounds on the asymptotic growth of a function.

Definition 2.7 (Big-Θ notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

We can use this notation to strengthen our last proposition.

Proposition 2.8. *The function $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ satisfies $f = \Theta(n^7)$.*

Proof. We have already seen that $f = O(n^7)$. We also have that for every $n \geq 1$, $f(n) \geq 4n^7$ so by definition $f = \Omega(n^7)$ (with $n_0 = 1$ and $c = 4$) and, therefore, $f = \Theta(n^7)$ as well. □

There are also situations where we want to argue that a function grows *asymptotically slower* than another reference function. We can state this formally with the little-o notation.

Definition 2.9 (little-o notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = o(g)$ if for every $c \in \mathbb{R}^+$, there exists $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) < c\, g(n)$.

Remark 2.10. Equivalently, $f = o(g)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. This definition can be easier to work with; feel free to do so.

Similarly, we can say that a function f grows asymptotically *faster* than another function g using the little-ω notation.

Definition 2.11 (little-ω notation)**.** Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \omega(g)$ if and only if $g = o(f)$.

We will be using this notation extensively throughout the course, so it is critical that you are all very comfortable working with it. We will be covering some exercises related to this notation in this week's tutorial and in the first assignment, but you can also find much more information about it and many other exercises to work on in the textbook.

2. The 2SUM problem

Now that we have all our definitions in place, let's put them to practice.

Definition 2.12 (2SUM)**.** An instance of the *2SUM problem* is an array $A \in \mathbb{Z}^n$ of n integers and an integer $m \in \mathbb{Z}$. The valid solution to an array A is True if there exist 2 indices $i, j \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] = m$, and False otherwise.

The following algorithm solves the 2SUM problem.

Algorithm 1: SIMPLE2SUM $(A[1..n], m)$

```
for i = 1, ..., n − 1 do
    for j = i, ..., n do
        if A[i] + A[j] = m return True;
return False;
```

The claim that SIMPLE2SUM solves the 2SUM problem requires a (simple) proof. But let's focus on the time complexity of this algorithm.

Proposition 2.13. *The* SIMPLE2SUM *algorithm has time complexity* $\Theta(n^2)$.

Proof. What we really care about in terms of time complexity for this problem is the number of basic operations (addition, incrementing, and comparison) on integers. We do this in the Word RAM model by setting the width w of words to be large enough that each integer in a problem instance can be stored with w bits.

And we measure time complexity in the *worst-case*, so we consider input arrays for which the inner loop is not interrupted by a "return True" command. In this case, the number of additions performed by the algorithm is

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n-1)}{2}.$$

For every $n \geq 2$, we have $\frac{n^2}{4} \leq \frac{n(n-1)}{2} \leq n^2$ so the algorithm performs $\Theta(n^2)$ additions and the same bound also holds for the total number of increments and compare operations. □

With this analysis in hand, we can now answer the central question in algorithm design: can we do better? Indeed we can! Let's break down what the algorithm is doing in detail. For each index $i \in \{1, 2, \ldots, n\}$, the inner loop is trying to determine if there is an integer $A[j]$ in A for which $A[i] + A[j] = m$, or, to rephrase the same statement: if the integer $m - A[i]$ is in the array.

As it turns out, we know a very efficient method for determining whether an integer (such as $m - A[i]$) is in an array: binary search! This, of course, only works when the array is sorted, but that's something we can do in the algorithm as well. The resulting algorithm is as follows.

Algorithm 2: 2SUM $(A = (A[1..n], m)$

```
SORT(A);
for i = 1, ..., n − 1 do
    if FIND(A, m − A[i]) then return True;
return False;
```

What we have just done looks simple, perhaps even obvious, yet it is a great example of an extremely powerful algorithmic technique: reduction.

Reduction idea: Use known algorithms to solve new problems.

We'll see many other examples where the reduction technique proves very useful—and it will even return with a starring role in the last module of this class, when we tackle

NP-completeness. But for now, let's continue our work and establish the correctness and time complexity of the 2SUM algorithm.

Proposition 2.14. *The* 2SUM *algorithm solves the 2SUM problem.*

Proof. Consider first the case where there exist indices i^*, j^* for which $A[i^*] + A[j^*] = m$. We can rewrite the identity as $A[j^*] = m - A[i^*]$, so for the iteration of the for loop where $i = i^*$, the integer $m - A[i]$ is in A, which means that the FIND algorithm returns True and so does 2SUM.

Consider now the case where for every indices i, j we have $A[i] + A[j] \neq m$. then for every index i, we have that for every index j, $A[j] \neq m - A[i]$ or, in other words, the integer $m - A[i]$ is not in A and FIND returns False. Therefore, 2SUM also correctly returns False. □

Proposition 2.15. *When* SORT *has time complexity* $\Theta(n \log n)$ *and* FIND *is a binary search algorithm with time complexity* $\Theta(\log n)$*, the* 2SUM *algorithm has time complexity* $\Theta(n \log n)$*.*

Proof. The initial call to SORT has time complexity $\Theta(n \log n)$ and the total time complexity of the for loop, in the worst case, is $n \cdot \Theta(\log n) = \Theta(n \log n)$. □

Can we do even better? It takes $\Theta(n)$ time just to read the integers stored in the array A, so the best we can probably hope for is to remove the extra $\log n$ term in our runtime. But instead of trying to do this, let's see if we can use the Reduction idea to solve even more complex problems.

3. The 3SUM problem

Definition 2.16 (3SUM)**.** An instance of the *3SUM problem* is an array $A \in \mathbb{Z}^n$ of n integers and an integer m. The valid solution to an array A is True if there exist 3 indices $i, j, k \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] + A[k] = m$, and False otherwise.

We can again define a simple algorithm to solve the 3SUM problem.

Algorithm 3: SIMPLE3SUM $(A[1..n], m)$

```
for i = 1, ..., n do
    for j = i, ..., n do
        for k = j, ..., n do
            if A[i] + A[j] + A[k] = m return True;
return False;
```

This algorithm is not so efficient.

Proposition 2.17. *The* SIMPLE3SUM *algorithm has time complexity* $\Theta(n^3)$*.*

Proof. The total number of additions performed by the algorithm is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=j}^{n} 2 = \Theta(n^3)$$

and this is also asymptotically the total number of elementary operations performed by the algorithm. □

Let's again try to use the Reduction technique to improve our algorithm. We can perhaps see how again an efficient FIND algorithm could help us. Or, since we just designed an efficient 2SUM algorithm, perhaps that algorithm can help us? Let's find out by again fixing an index i and seeing what the two inner loops are doing. They're trying to find out if there are indices j, k for which

$$A[i] + A[j] + A[k] = m \quad \iff \quad A[j] + A[k] = m - A[i].$$

This is an instance of the 2SUM problem!

Algorithm 4: 3SUM $(A[1..n], m)$

for $i = 1, \ldots, n$ **do**
  if 2SUM $(A, m - A[i])$ then **return** True;
return False;

And the analysis of the time complexity is again simple, yielding the following result.

Theorem 2.18. *The 3SUM problem can be solved by an algorithm with time complexity* $\Theta(n^2 \log n)$.

Proof. When the 3SUM algorithm calls the algorithm 2SUM with time complexity $\Theta(n \log n)$ that we designed in the previous section, its total time complexity is $n \cdot \Theta(n \log n) = \Theta(n^2 \log n)$. □

Can you do even better? With some work, you may be able to design an algorithm that solves the 3SUM problem in time $\Theta(n^2)$. Determining whether we can do even better is one of the major open problems in algorithms research today: it was only very recently that researchers were able to show that 3SUM can be solved by an algorithm with time complexity $o(n^2)$, and whether or not there is an algorithm that solves 3SUM with time complexity $O(n^\gamma)$ for some $\gamma < 2$ remains unknown.

CS 341: ALGORITHMS (S18) — LECTURE 3

SOLVING RECURRENCES

ERIC BLAIS

Many of the algorithms that we will have to analyze in this course have a recursive structure. To complete this analysis, we need to develop tools for solving recurrences. This is what we do today.

1. Recurrence trees

Recall that the MergeSort algorithm is defined as follows.

Algorithm 1: MergeSort $(A = (A[1], \ldots, A[n]))$

if $n = 1$ **return**;
MergeSort $(A[1, \ldots, \lfloor n/2 \rfloor])$;
MergeSort $(A[\lfloor n/2 \rfloor + 1, \ldots, n])$;
Merge $(A[1, \ldots, \lfloor n/2 \rfloor], A[\lfloor n/2 \rfloor + 1, \ldots, n])$;

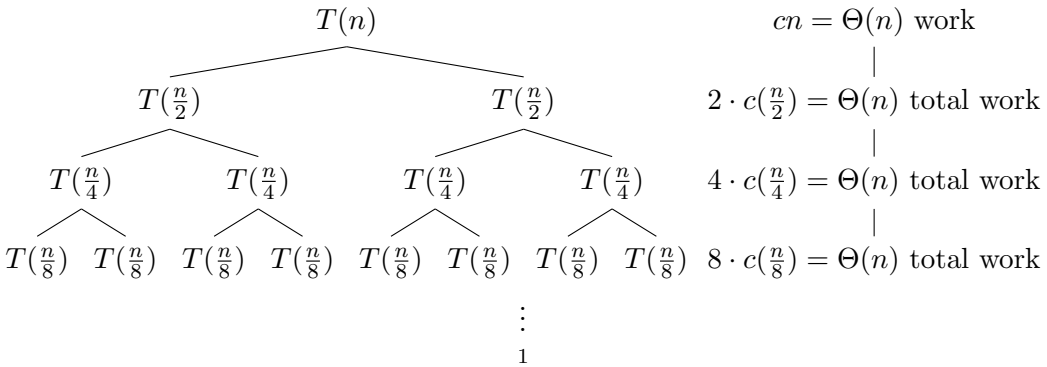
When Merge is an algorithm that merges the two input arrays in time $\Theta(n)$, what is the time complexity of MergeSort? If we let $T(n)$ denote the time complexity of the algorithm on arrays with n entries, we find that

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n).$$

To simplify the analysis, let's assume that n is a power of 2. Then we have

$$T(n) = 2\,T(\tfrac{n}{2}) + \Theta(n).$$

To determine the solution to this recursion, the most natural approach is to draw the recursion tree and write down how much time is spent on each level of the tree:



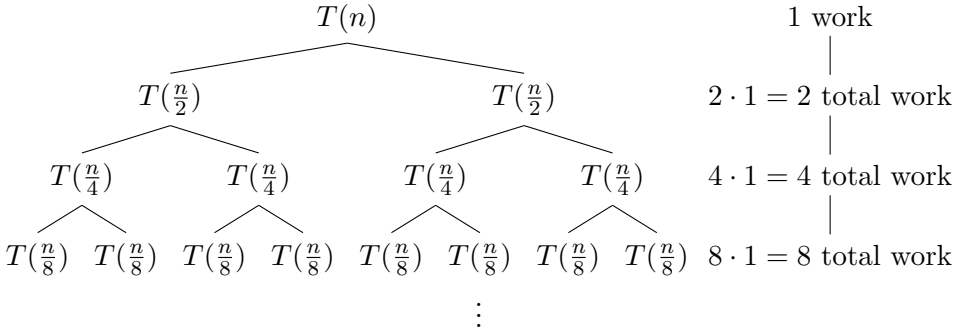
The recursion tree has depth $\log n$ and $\Theta(n)$ time is spent at each level of the tree on completing the merges. The total time complexity of the MergeSort algorithm is therefore

$$T(n) = \Theta(n \log n).^1$$

Many other recursions can also be solved using the same recursion tree approach. For example, what if we (magically) allowed the Merge algorithm to run in a single unit of time? Then the recursion defining the time complexity of the MergeSort algorithm would be

$$T(n) = 2\,T(\tfrac{n}{2}) + 1$$

and the recursion tree would now look like this:



This tree again has depth $\log n$, so the total time complexity in this case is

$$T(n) = 1 + 2 + 4 + 8 + \cdots + \tfrac{n}{2} + n = 2n - 1 = \Theta(n).$$

One more variant: what if we proceed recursively but now divide the array in three instead of two pieces?

Algorithm 2: TriMergeSort $(A = (A[1], \ldots, A[n]))$

if $n = 1$ **return**;
TriMergeSort $(A\big[1, \ldots, \lfloor n/3 \rfloor\big])$;
TriMergeSort $(A\big[\lfloor n/3 \rfloor + 1, \ldots, \lfloor 2n/3 \rfloor\big])$;
TriMergeSort $(A\big[\lfloor 2n/3 \rfloor + 1, \ldots, n\big])$;
Merge ($A\big[1, \ldots, \lfloor n/3 \rfloor\big]$, $A\big[\lfloor n/3 \rfloor + 1, \ldots, \lfloor 2n/3 \rfloor\big]$, $A\big[\lfloor 2n/3 \rfloor + 1, \ldots, n\big]$);

Let Merge again be an algorithm with time complexity $\Theta(n)$. When n is a power of 3, the time complexity of the TriMergeSort algorithm is

$$T(n) = 3\,T(\tfrac{n}{3}) + \Theta(n).$$

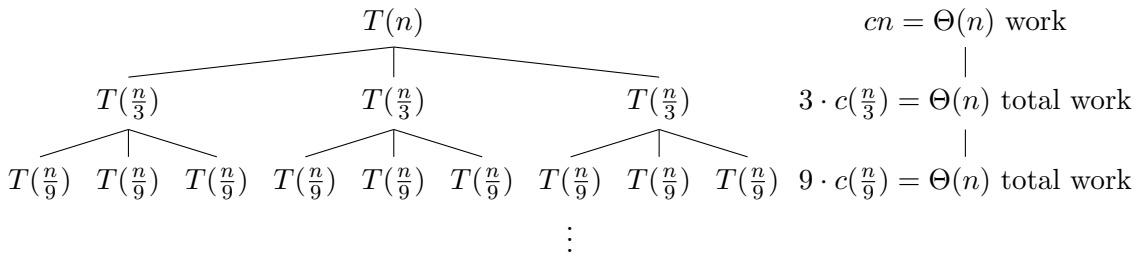
The recursion tree for this recurrence is:

The tree has depth $\log_3 n$, so the time complexity of TriMergeSort is

$$T(n) = \Theta(n \log_3 n) = \Theta(n \log n),$$

the same (asymptotically) as MergeSort!

[1] Here and throughout this course, logarithms without subscripts are over base 2.



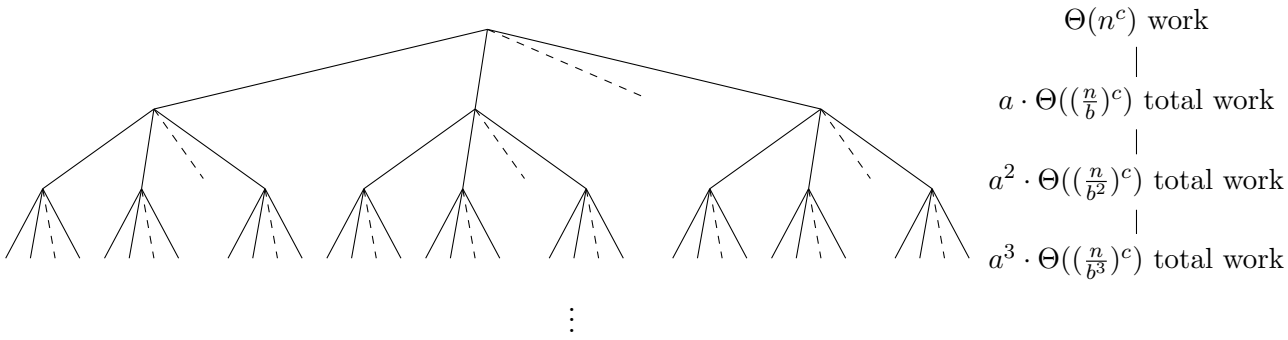
2. Master theorem

We could continue exploring various different examples using the recursion tree method. But let's be a bit more ambitious and try to consider a general scenario that captures many individual examples at once. Let T be defined by the recurrence

$$T(n) = aT(\tfrac{n}{b}) + \Theta(n^c)$$

for some constants $a > 0$, $b > 1$, and $c \geq 0$.

We can again use the recursion tree approach to solve for T in this case.



We can therefore express T as

$$T(n) = \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n}\right) \cdot \Theta(n^c).$$

To determine T, we must simply understand the geometric sequence with ratio $\frac{a}{b^c}$. There are three cases to consider.

Case 1. When $\frac{a}{b^c} = 1$, then each term in the sequence is 1 and so

$$T(n) = \log_b(n) \cdot \Theta(n^c) = \Theta(n^c \log n).$$

Case 2. When $\frac{a}{b^c} < 1$, then the geometric series $1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n} = \Theta(1)$ is a constant and so

$$T(n) = \Theta(n^c).$$

Case 3. When $\frac{a}{b^c} > 1$, then we have an increasing geometric series that is dominated by the last term, so that

$$T(n) = \Theta\left(n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n}\right) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}).$$

This result establishes what is known as the *Master Theorem*.

Theorem 3.1 (Master theorem)**.** *If $T(n) = aT(\frac{n}{b}) + \Theta(n^c)$ for some constants $a > 0$, $b > 1$, and $c \geq 0$, then*

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } c > \log_b a \\ \Theta(n^c \log n) & \text{if } c = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } c < \log_b a. \end{cases}$$

What is most important to remember of this theorem is not the statement of the theorem itself but rather the method that we used to obtain it: with the recursion tree method, you can easily recover the Master Theorem itself and also solve recursions that are not of the form covered by the theorem.

3. Geometric series

The different cases of the Master Theorem were handled rather quickly in the section above. It's worth slowing down a bit to see exactly how the asymptotic results about geometric series are obtained. The starting point for those results is the fundamental identity for geometric series.[2]

Fact 3.2. *For any $r \neq 1$ and any $n \geq 1$,*

$$1 + r + r^2 + r^3 + \cdots + r^n = \frac{1 - r^{n+1}}{1 - r}.$$

We can now use this lemma to give the big-Θ closed form expressions for geometric series when $r \neq 1$. Let's start with the case where $r < 1$.

Theorem 3.3. *For any $0 < r < 1$, the function $f : n \mapsto 1 + r + r^2 + \cdots + r^n$ satisfies $f = \Theta(1)$.*

Proof. From the geometric series identity,

$$1 + r + r^2 + r^3 + \cdots + r^n = \frac{1 - r^{n+1}}{1 - r}$$

we observe that for every $n \geq 1$, $f(n) \leq \frac{1}{1-r}$ so, setting $n_0 = 1$ and $c = \frac{1}{1-r}$ we have that for all $n \geq n_0$, $f(n) \leq c \cdot 1$ and, therefore, $f = O(1)$.

Similarly, if we let n_0 be the smallest integer for which $r^{n_0} \leq \frac{1}{2}$ (which is the value $n_0 = \lceil \log_{1/r}(2) \rceil$), then for every $n \geq n_0$ we have $\frac{1-r^{n+1}}{1-r} \geq \frac{1}{2(1-r)}$ and, taking the constant $c = \frac{1}{2(1-r)}$, $f \geq c \cdot 1$ so $f = \Omega(1)$.

Since $f = O(1)$ and $f = \Omega(1)$, then also $f = \Theta(1)$. □

The analysis of geometric series when $r > 1$ is similar.

Theorem 3.4. *For any $r > 1$, the function $f : n \mapsto 1 + r + r^2 + \cdots + r^n$ satisfies $f = \Theta(r^n)$.*

Proof. The geometric series identity can also be expressed as

$$1 + r + r^2 + r^3 + \cdots + r^n = \frac{r^{n+1} - 1}{r - 1}.$$

With $n_0 = 1$ and $c = \frac{r}{r-1}$, this identity implies that for all $n \geq n_0$, $f(n) \leq \frac{r^{n+1}}{r-1} = c \cdot r^n$ so $f = O(r^n)$.

[2] Can you see how to prove this identity? *Hint:* Multiply the left-hand side of the identity by $1 - r$ and remove the terms that cancel out.

And when we choose n_0 to be the smallest integer that satisfies $r^{n_0} \geq 2$ (or, equivalently, $n_0 = \lceil \log_r(2) \rceil$) and $c = \frac{r}{2(r-1)}$ then we have that for all $n \geq n_0$, $f(n) = \frac{r^{n+1}-1}{r-1} \geq \frac{r^{n+1}-\frac{1}{2}r^{n+1}}{r-1} = c \cdot r^n$ so $f = \Omega(r^n)$.

Since $f = O(1)$ and $f = \Omega(1)$, then also $f = \Theta(1)$. □

CS 341: ALGORITHMS (S18) — LECTURE 4
FAST MULTIPLICATION

ERIC BLAIS

The MergeSort algorithm we studied in the last lecture is a classic example of the power of the *divide and conquer* technique. This technique is a general approach for solving problems with a three-step approach:

Divide: the original problem into smaller subproblems,
Conquer: each smaller subproblem separately, and
Combine: the results back together.

With MergeSort, we divided the initial array in two, conquered the sorting problem on the two smaller arrays with recursive calls to MergeSort, and combined the results using Merge. In the next few lectures, we will see other problems where this approach is applicable.

1. Integer multiplication

Let's consider one of the most fundamental arithmetic problems around: how to multiply two positive integers.

Definition 4.1 (Integer multiplication problem)**.** Given two n-bit positive integers x and y, output their product xy.

Humans invented an algorithm for this problem thousands of years ago. It is now known as the grade school algorithm. With this algorithm, we multiply x by y_i and shift the result $n-i$ positions to the left. As an example, when we run this algorithm to solve 13×11, we obtain

$$\begin{array}{r} 1101 \\ \times\ 1011 \\ \hline 1101 \\ 1101 \\ 0000 \\ +\ 1101 \\ \hline 10001111 \end{array} \qquad (= 143)$$

Notice that in the computations, we produces $O(n^2)$ intermediate bits, and this is indeed the time complexity of this algorithm. This begs the question: can we use the Divide & Conquer approach to obtain a more efficient algorithm?

To apply the Divide & Conquer approach, we need to first determine how we can break up the original multiplication problem into problems on smaller inputs. The natural way to do this is to split the n-bit integer x into two $n/2$-bit integers x_L (containing the $n/2$

most-significant bits) and x_R (containing the least-significant bits) and to do the same with y. Then

$$x = 2^{n/2}x_L + x_R,$$
$$y = 2^{n/2}y_L + y_R,$$

and

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

This is exactly what we were hoping to see! We have divided up the problem of multiplying two n-bit integers into four instances of the smaller problem of multiplying two $n/2$-bit integers. The resulting algorithm is as follows.

Algorithm 1: MULTIPLY $(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_n)$

if $n = 1$ **return** xy;
$(x_L, x_R) \leftarrow$ SPLIT (x);
$(y_L, y_R) \leftarrow$ SPLIT (y);
$P_{LL} \leftarrow$ MULTIPLY (x_L, y_L);
$P_{LR} \leftarrow$ MULTIPLY (x_L, y_R);
$P_{RL} \leftarrow$ MULTIPLY (x_R, y_L);
$P_{RR} \leftarrow$ MULTIPLY (x_R, y_R);
return $2^n P_{LL} + 2^{n/2}(P_{LR} + P_{RL}) + P_{RR}$;

At first, it might be worrisome to see that we have also added extra multiplications by 2^n and $2^{n/2}$, but in fact these are just shifts (by n and $n/2$ bits, respectively), so these operations have time complexity $O(n)$ and the total time complexity of the MULTIPLY algorithm is defined by the recursion

$$T(n) = 4\,T(n/2) + O(n).$$

We can apply the Master Theorem with parameters $a = 4$, $b = 2$, and $c = 1$ and the observation that $1 = c < \log_b a = 2$ to obtain

$$T(n) = O(n^2).$$

This is exactly the same as the grade school algorithm! So while the Divide & Conquer approach gives an elegant algorithm, it does not appear to have led to any efficiency improvement.

2. Fast integer multiplication

...And yet if that was the end of the story, we probably wouldn't be covering the integer multiplication problem at this point in the class! Recall that

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

There's a deceptively simple observation that can be traced back to Gauss which will be immensely useful to us: the middle term $x_L y_R + x_R y_L$ satisfies the identity

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

We can use this identity to express the term $x_L y_R + x_R y_L$ with a difference of three multiplications. This would be a step back for us (we only need 2 multiplications to compute the same term directly!) except that two of the multiplications, $x_L y_L$ and $x_R y_R$ are multiplications that we already need to do in the multiplication algorithm anyways! As a result, we can now implement a Divide & Conquer multiplication algorithm that performs only 3 multiplications instead of 4.

Algorithm 2: FastMultiply $(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_n)$

if $n = 1$ **return** xy;
$(x_L, x_R) \leftarrow$ Split (x);
$(y_L, y_R) \leftarrow$ Split (y);
$P_{LL} \leftarrow$ FastMultiply (x_L, y_L);
$P_{RR} \leftarrow$ FastMultiply (x_R, y_R);
$P_{\text{sum}} \leftarrow$ FastMultiply $(x_L + x_R, y_L + y_R)$;
return $2^n P_{LL} + 2^{n/2}(P_{\text{sum}} - P_{LL} - P_{RR}) + P_{RR}$;

The time complexity of this algorithm now satisfies

$$T(n) = 3\,T(n/2) + O(n).$$

We can again apply the Master Theorem, this time with parameters $a = 3$, $b = 2$, and $c = 1$. Since $1 = c < \log_b a = \log_2 3 < 1.59$, we obtain

$$T(n) = O(n^{1.59}).$$

This result is a great illustration of the power of algorithmic thinking: despite countless mathematicians (and students of all ages) using multiplication algorithms over thousands of years, it was only in 1960 that Karatsuba showed with the above algorithm that it was possible to solve the integer multiplication in subquadratic time.

3. Fast matrix multiplication

A similar approach can also be used to obtain a fast matrix multiplication algorithm. Given two $n \times n$ matrices A and B, their product is the matrix $C = AB$ whose entries satisfy

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}.$$

The algorithm that uses this definition directly has time complexity $O(n^3)$. Again, we can try to use the Divide & Conquer method to obtain a more efficient algorithm. In this case, it is natural to divide each matrix in four:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \quad \text{and} \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

We then observe that the 4 submatrices of C can each be obtained by taking products of the submatrices of A and B:

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}. \end{aligned}$$

The Divide & Conquer algorithm obtained using these identities makes 8 multiplications on $\frac{n}{2} \times \frac{n}{2}$ matrices and requires $O(n^2)$ extra work (since the matrices it adds together have n^2 entries) so its time complexity satisfies

$$T(n) = 8\,T(n/2) + O(n^2).$$

With the Master Theorem, we see that this time complexity is $T(n) = O(n^3)$, the same as the naïve matrix multiplication algorithm.

As was the case with the integer multiplication problem, however, it is possible to be more clever in how we choose the multiplications of the submatrices so that we compute the product C with only 7 (instead of 8) multiplications. Doing so yields a time complexity

$$T(n) = 7\,T(n/2) + O(n^2),$$

which is $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$. The resulting algorithm is known as *Strassen's algorithm* and has been extremely influential in practice as well as in theory: there are numerous computational problems today that require multiplication of enormous matrices, and Strassen's algorithm is provides significant efficiency improvements over the naïve algorithms in those settings.

As a good algorithm designer, however, there is one question that you should be asking yourself at this point: can we do even better than Strassen's algorithm? Indeed we can. At the moment, the best known algorithm for matrix multiplication has time complexity $O(n^{2.37286\ldots})$. It is conjectured that matrix multiplication can be done in time $O(n^2)$ which, if true, would clearly be optimal since there are n^2 entries in an $n \times n$ matrix.