Background

Acknowledgement
This notes is compiled based on the following textbooks and a variety of Internet resources, such as Wikipedia for example.

- Introduction to Algorithms by Cormen, Lerserson, Rivest and Stein.

1 Introduction

Definition 1 *An algorithm is a step by step procedure for solving a problem. Computationally, an algorithm is a well-defined computaional procedure that takes some value or a set of values, as input and produce some value, or a set of values as output.*

The word algorithm originates from the Persian mathematician Abu Abdallah Muammad ibn Musa al-Khwarizmi (AD 780-850), whose name's Latin translation is Algoritmi or Algaurizin (see Figure 1). Among his many contributions, his book "On the Calculation with Hindu Numerals" spread HinduArabic numeral system throughout the Middle East and Europe. The Latin translation of the book is "Algoritmi de numero Indorum", which led to the word "algorithm". His book "The Compendious Book on Calculation by Completion and Balancing", with the Latin translation as "Liber Algebr et Almucabola" established the mathematical discipline algebra.

Figure 1: Statue of al-Khwarizmi at Amirkabir University of Technology, Tehran

Definition 2 *A data structure is a way to store and organize data for efficeint access and modification.*

In this course, unless specified otherwise, we shall assume the underlying computation model is a **random access machine (RAM)**. In RAM model, instructions are executed sequentially, i.e., with no concurent operations. There is no memory hierarchy. Each standard data type (char, int, float, double, etc) occupies one memory cell. Each memory access takes one time step, and there is no shortage of memory. Each simple operation (e.g., addition, comparison, multiplication, etc) takes one time step. For any given problem the **running time** of an algorithms is the number of time steps. The space used by an algorithm is the number of memory cells.

In the RAM model, the **input size** of a particular input is the number of memory cells occupied. The **running time** of an algorithm of a particular input of size n is the number of time steps it takes to produce the desired output, and is usually denoted by $T(n)$, i.e., a functino of n. Since running time and space of an algorithm depends on the particular input, in order for the analysis to be meaningful, we can either focus on **worst-case running time** or **average-case running time**. The worst-case running time is the most number of steps for any input of size n, while the average-case running time is the expected running time of the algorithm over all possible input.

We'll generally be pessimistic and perform worst-case running time analysis for algorithms. You will be surprised that in most situations, the worst-case running time is actually a very good indication of the real-world performance of the algorithm, since the "average case" is generally as bad as the worst case. There are situations when the average-case is much better than the worst-case, we will talk about these situations when we discuss randomized algorithms. As you can imagine, average-case analysis is more involing and generally requires the computation of the expectation of the running time as a random variable.

Like the running time, the **space** or **space complexity** of the algorithm is the number of memory cells (including input) the algorithm uses.

Example 1 *Given an array A of n integers. The integers are from $1, 2, ..., n+1$. There are no duplicates. One of the integers is missing. Can you develop an algorithm to determine which integer is missing?*

Solution:

We will present three algorithms for solving the problem.

One observation is that the missing integer must equal to $\sum_{j=1}^{n+1} - \sum_{j=1}^{n} A[j]$. This immediately yields the following algorithm 1 with the following pseudo-code:

Algorithm 1

Input: An array A of n integers from $1, 2, ..., n+1$ with one mising.

Output: The missing integer.

Line 1: Initialize S to be 0.

Line 2: for $j = 1$ to n do:

Line 3: $S = S + A[j]$

Line 4: return $\frac{(n+1)(n+2)}{2} - S$.

Similarly if getting the sum works, getting the product should also work, i.e., $\frac{\Pi_{j=1}^{n+1}}{\Pi_{j=1}^{n} A[j]}$, where Π is the product symbol. This immediately yields the following algorithm 2 with the following pseudo-code:

Algorithm 2

Input: An array A of n integers from $1, 2, ..., n+1$ with one mising.

Output: The missing integer.

Line 1: Initialize P to be 1.

Line 2: for $j = 1$ to n do:

Line 3: $P = P \cdot A[j]$

Line 4: return $\frac{(n+1)!}{P}$.

The third observation is that if we can use a Boolean array B of size $n + 1$. We iterate through all the entries of A, for each entry $A[j]$, we will mark $B[A[j]]$ as $True$. In the end the entry in B that equals to $False$ is the missing integer. This yeilds the following algorithm 2 with the pseudo-code:

Algorithm 3

Input: An array A of n integers from $1, 2, ..., n + 1$ with one mising.

Output: The missing integer.

Line 1: Initialize a Boolean array B of size $n + 1$.

Line 2: Set all entries of B to be $False$.

Line 3: for $j = 1$ to n do:

Line 4: $B[A[j]] = True$

Line 5: for $k = 1$ to $n + 1$ do

Line 6: if $B[k] = False$, return k.

So, what is the strength and weakness of the three algorithms? Which algorithm runs faster? Which algorithm uses less space? Which algorithm is easier to implement?

Recall that under the RAM model, each simple instruction takes one unit time step and each standard data type takes one unit memory cell.

The running time of the first algorithm is: $2n + 5$. This is because there are n iterations, and in each iterration, we spend one unit of time incrementing j and one unit of time for addition; we spend 4 units of time for calculating $\frac{(n+1)(n+2)}{2}$; and we spend 1 unit of time for the final subtraction. The space used is $n + 1$, where the 1 comes from storing the sum S.

The running time of the second algorithm is: $3n + 2$. This is because there are n iterations, and in each iterration, we spend one unit of time incrementing j and one unit of time for multiplication; we spend n units of time for calculating $(n + 1)!$; and we spend 1 unit of time for the final division. The space used is $n + 1$, where the 1 comes from storing the product P.

The running time of the third algorithm is: $5n + 2$, where there is a $n + 1$ coming from initialing all the entries of B to be $False$, a $3n$ coming from scanning through the array A and marking the corresponding entry in B, and a $n + 1$ coming from scanning through the array B. The space used is $n + (n + 1)$, since the array B will take $n + 1$ memory cells.

Clearly from our RAM model analysis, algorithm 1 is superior to algorithms 2 and 3 in both time and space. under the RAM model. □

How does our RAM model analysis of the three algorithms fair in the practical situations?

We shall start with the easier topic on space first. Clearly, algorithms 1 and 2 uses less space. Using the RAM model, the third algorithm will use twice the space of the first two algorithms because of the use of the Boolean array B. Since space are valuable resources, for extremely large arrays, twice the space could be the difference of whether being able to store the entire data set in the main memory or not. Since a virual memory page fault could be 1,000,000 times slower than the memory access, doubling the space could be fatal. On the other hand, in practical situations, the array B will not double the amount of space used. If each entry in an array B is implemented using a bit, then storing array B will only take $\frac{n+1}{8}$ Bytes. Assuming each entry in array A is a standard integer, which takes 4 Bytes, then the space occupied by A will be $4n$ Bytes. The using B will only increase the storage by $\frac{1}{32}$, which is about 3%.

Now let's examine running time more carefully. In our RAM model, the running time of of addition and multiplication are the same. However, multiplication is obviously much more expensive than addition. As a rule of thumb, multiplication usually requires at least four times the number of CPU cycles in comparison to addition. Thus, in terms of the acutal performance, the running time of algorithm 2 will be much slower than algorithm 1.

The other problem with algorithms 1 and 2 is overflow. For ease of explanation, let's assume the standard type integer takes 32 bit. Assuming the architecture uses first bit to indicate whether the number is positive or negative, then the range of the integers is $-(2^{31}-1), ..., +2^{31}-1$, i.e., $2147483647, ..., +2147483647$. Overflow means that the result of an arithmetic operation doesn't fit in the number of bits of a standard type used for the operation. (This is usually handled by an overflow bit.) It's obvious that algorithm 2 will have immediate overflow problem. (Note that $13! = 6227020800$ is already larger than 2147483647.) However, even algorithm 1 can incur over flow problems for larger arrays, for example, if $n = 65535$, which is not very big, $\frac{(n+1)(n+2)}{2} = 2147516416$ already overflows. Algorithm 3 on the other hand doesn't have any potential overflow problem. In fact, since algorithm 3 doesn't involve addition, it could be the most efficient in terms of actual running times. Is there a way to develop an algrithm that neither has potential overflow problem nor uses extra storage? The answer is using XOR.

The XOR (or $\oplus$) operator is defined using the following truth table:

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

For two multi-bit integer x and y, $x \oplus y$ is the integer obtained by performing XOR for each pair of corresponding bits. XOR has many important properties, here we shall use the following four properties: (1) $x \oplus 0 = x$, (2) $x \oplus x = 0$, (3) $x \oplus y = y \oplus x$ (i.e., commutative), and (4) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ (i.e., associative). This gives us the following algorithm 4:

Algorithm 4

Input: An array A of n integers from $1, 2, ..., n+1$ with one mising.

Output: The missing integer.

Line 1: S = 0.

Line 2: for $j = 1$ to n do:

Line 3: $S = S \oplus A[j]$

Line 2: for $j = 1$ to $n + 1$ do:

Line 3: $S = S \oplus j$

Line 4: return S.

Note: A popular interview question is to use the XOR operation to swap two varaibles without intruducing any additional memory. Show how this can be done.

Another important thing to remember when solving an algorithmic problems is that one should always try to take advantage if the data is given in certain order. As an example, consider the following problem:

Example 2 *Given a* **sorted** *array of A of n integers. The integers are from* $1, 2, ..., n + 1$. *There are no duplicates. One of the integers is missing. Can you develop an algorithm to determine which integer is missing?*

Solution: Consider the entry $A[\frac{n}{2}]$. Observe that if the missing integer is in the range $1, 2, ..., \frac{n}{2}$, then $A[\frac{n}{2}] > \frac{n}{2}$, thus we don't need to search the part of the array $A[\frac{n}{2} + 1, ..., n]$. As a result, we can recursively search the part of the array $A[1, ..., \frac{n}{2}]$. This immediately yields the following algorithm 5, which recurively using the routine called find_missing.

Find_Missing

Input: An array A, a starting index i and an ending index j. The contents of $A[i, ..., j]$ store the integers $i, i + 1, ..., j + 1$ with one integer missing.

Output: The missing integer.

Line 1: $k = \lfloor \frac{i+j}{2} \rfloor$

Line 2: if $A[k] = k$, Find_Missing (A, i, k)

Line 3: else if $A[k] > k$ Find_Missing (A, k, j)

Line 4: else return k.

Algorithm 5

Input: An array A of n integers from $1, 2, ..., n + 1$ with one mising.

Output: The missing integer.

Line 1: return Find_Missing (A, 1, n)

The above algorithm is a **divide and conquer** type of algorithm. The running time analysis usually comes down to solving a recurrence relation. Let $T(n)$ denotes the running time. Then $T(n)$ will satisify the following recurrent relation: $T(n) = 1+T(\lceil \frac{n}{2} \rceil)$. This recurrence relation can be solved using the **brute-force** method.

Assume that $n = 2^k$, then $T(n) = 1 + T(\lceil \frac{n}{2} \rceil)$ becomes $T(2^k) = 1 + T(2^{k-1})$.

Expanding this recursion, we obtain: $T(2^k) = 1 + T(2^{k-1}) = 1 + (1 + T(2^{k-2})) = 2 + T(2^{k-2}) = 2 + (1 + T(2^{k-3})) = 3 + T(2^{k-3}) = ... = k + T(2^{k-k}) = k + 1 = \log_2 n + 1$, which is much more efficient than the previous four algorithms due the fact that the numbers are given sorted. □

2 Big O Notations

In this course, unless otherwise noted, we shall use the RAM model for analyzing time and space. As you have already seen that the running time coming out the RAM model is not the most accurate, however, it is very close in terms of the order of of magnitude if we ignore the constant muliplicative coefficient. This gives rise to the **Growth Rate**, also referred to as the **Big O notations** or the **Asymptotic Notations**. In a nutshell, the growth rate of a function $f(n)$ is its leading growth term with the constant multiplicative factor stripped off. (Recall that we say a function $g(n)$ grows faster **assympototically** than another function $h(n)$, if $\lim_{n\to\infty} \frac{g(n)}{h(n)} = \infty$.) As an example, the growth rate of the function $f(n) = 42n^3 + 25n^{\frac{3}{2}} + 32n \log n + 10^100$ is n^3, we in this situation, we say the function $f(n)$ is **Big O** of n^3 and write $f(n) = O(n^3)$.

Since the running times of most algorithms are **positive increasing** functions, i.e., the running time increases as the input size increases, we will focus on this special type of functions. Now we can define rigously the asympotic notiations for two **positive increasing** functions $f(n)$ and $g(n)$ with $n = 1, 2, ...$:

Definition 3 *We say $f(n)$ is $O(g(n))$ (or assymptotically less than or equal) if there exists positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.*

Definition 4 *We say $f(n)$ is $\Omega(g(n))$ (or assymptotically greater than or equal) if there exists positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.*

Definition 5 *We say $f(n)$ is $\Theta(g(n))$ (or assymptotically equal) if there exists positive constants c_1 and c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. Clearly, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Definition 6 *We say $f(n)$ is $o(g(n))$ (or assymptotically smaller) if* **for any** *positive constant c there exists $n_0 > 0$ such that $f(n) < cg(n)$ for all $n \geq n_0$. Note that the key difference between Big O and small o notiation is that Big O notation only requires the inequality $f(n) < cg(n)$ for some constant c, while in the small o notation, the inequality must hold for all positive constants c. Because of this, $f(n) = o(g(n))$ implies $f(n) = O(g(n))$, but not vice versa.*

Definition 7 *We say $f(n)$ is $\omega(g(n))$ (or assymptotically larger) if for any positive constant c there exits $n_0 > 0$ such that $f(n) > cg(n)$ for all $n \geq n_0$. Similarly, $f(n) = \omega(g(n))$ implies $f(n) = \Omega(g(n))$, but not vice versa.*

The asymptotic notations are instrinsically connected with the definition of limit. Recall that we say the limit of a sequence x_n approaches x^* as n goes to infinity if for every real number $\epsilon > 0$, there exists a natural number N such that for all $n \geq N$, we have $|x_n - x^*| \leq \epsilon$. Natually, $\lim_{n\to\infty} x_n = 0$ if for any $\epsilon > 0$, there exists a natural number N such that for all $n \geq N$, we have $|x_n| \leq \epsilon$. Further, we say the limit of a sequence x_n approaches ∞ as n goes to infinity if for every real number $M > 0$, there exists a natural number N such that for all $n \geq N$, we have $x_n \geq M$.

Observe that we can essentially view positive functions such as $f(n)$ and $g(n)$ defined on $n = 1, 2, ...,$ as a sequence. Now assume $\lim_{n\to\infty} \frac{f(n)}{g(n)} = a$, for some positive constant a. Using the definition of limit, this means, for every $\epsilon > 0$, there exists an N, such that for all $n \geq N$, $\left|\frac{f(n)}{g(n)} - a\right| \leq \epsilon$. Set $\epsilon = \frac{a}{2}$, we have there exists $N(\frac{a}{2})$, such that such that for all $n \geq N(\frac{a}{2})$, $\left|\frac{f(n)}{g(n)} - a\right| \leq \frac{a}{2}$. This implies, $-\frac{a}{2} \leq \frac{f(n)}{g(n)} - a \leq \frac{a}{2}$, which is $\frac{a}{2} \leq \frac{f(n)}{g(n)} \leq \frac{3a}{2}$, which is $\frac{a}{2}g(n) \leq f(n) \leq \frac{3a}{2}g(n)$ for all $n \geq N(\frac{a}{2})$. Now observe that by setting the c_1, c_2, and n_0 in Definition 5 as: $c_1 = \frac{a}{2}$, $c_2 = \frac{3a}{2}$, and $n_0 = N(\frac{a}{2})$, we have for $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$, and therefore $f(n) = \Theta(g(n))$. Thus, to show a function $f(n) = \Theta(g(n))$, it suffices to show that the limit of $\frac{f(n)}{g(n)}$ approaches some positive constant as n goes to infinity.

Now, what if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$? Using the definition of limit, we have for every $\epsilon > 0$, there exists an N, such that for all $n \geq N$, $\frac{f(n)}{g(n)} \leq \epsilon$. Observe that $\frac{f(n)}{g(n)} \leq \epsilon$ implies $\frac{f(n)}{g(n)} < 2\epsilon$. Hence we have $f(n) < 2\epsilon g(n)$. Now set the n_0 and c in Definition 6 as: $n_0 = N$ and $c = 2\epsilon$, we have for all $n \geq n_0$, $f(n) < cg(n)$, hence $f(n) = o(g(n))$. Similarly, one can show that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f(n) = \omega(g(n)$.

Observation 1 *For* **positive increasing** *functions $f(n)$ and $g(n)$ defined on $n = 1, 2, ...,$:*

$$\begin{cases} \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 & f(n) = o(g(n)), f(n) = O(g(n)) \\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = a, 0 < a < \infty & f(n) = \Theta(g(n)), f(n) = O(g(n)), f(n) = \Omega(g(n)) \\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty & f(n) = \omega(g(n)), f(n) = \Omega(g(n)) \end{cases}$$

From the above definition, it is clear that given two positive increasing functions f and g, to figure out their assympototic relations, one only needs to calculate the limit. This requires L'Hospital rule:

Theorem 1 *Let $f(x)$ and $g(x)$ be differentiable functions, such that $\lim_{x\to\infty} f(x) = \infty$ and $\lim_{x\to\infty} g(x) = \infty$, then*

$$\lim_{n\to\infty} \frac{f(x)}{g(x)} = \lim_{n\to\infty} \frac{f'(x)}{g'(x)}$$

Example 3 *Calculate $\lim_{x\to\infty} \frac{\sqrt{x}}{\log x}$.*

Solution:

$$\lim_{n\to\infty} \frac{\sqrt{x}}{\log x} = \lim_{n\to\infty} \frac{(\sqrt{x})'}{(\log x)'} = \lim_{n\to\infty} \frac{\frac{1}{2}x^{-\frac{1}{2}}}{\frac{1}{x}} = \lim_{n\to\infty} \frac{\sqrt{x}}{2} = \infty$$

Thus, polynomial in n grows faster than polynomial in $\log n$. □

Example 4 *Calculate $\lim_{x\to\infty} \frac{e^x}{x^2}$*

Solution:

$$\lim_{x\to\infty}\frac{e^x}{x^2}=\lim_{x\to\infty}\frac{(e^x)'}{(x^2)'}=\lim_{x\to\infty}\frac{e^x}{2x}=\lim_{x\to\infty}\frac{(e^x)'}{(2x)'}=\lim_{x\to\infty}\frac{e^x}{2}=\infty$$

Thus, exponetial grows faster than polynomial. □

3 Expoentiation and Exponential Functions

Definition 8 *Let a be a real number, and n be a positive integer, then*

$$a^n=\underbrace{a\cdot a\cdot ...\cdot a}_{n\ of\ a}$$

where a is called the base and n is called the exponent.

Below are some important properties of exponentiation. Let a be a real number, m, n be positive integers. Then

- $a^{m+n}=a^m a^n$
- $(a^m)^n=a^{mn}$
- For $a\neq 0$, define $a^0=1$. Since $1=a^0=a^{1-1}=a^1a^{-1}=aa^{-1}$, we have $a^{-1}=\frac{1}{a}$. Similarly, $a^{-n}=(a^n)^{-1}=\frac{1}{a^n}$.
- Since $a=a^1=a^{\frac{1}{n}n}=(a^{\frac{1}{n}})^n=\underbrace{a^{\frac{1}{n}}\cdot a^{\frac{1}{n}}\cdot ...\cdot a^{\frac{1}{n}}}_{\text{n}}$, thus $a^{\frac{1}{n}}$ is the $n-$th real root of the equation $x^n=1$. To avoid ambiguities, if a is positive and n is even, the equation $x^n=a$ has two real roots, one positive and one negative, the positive root is denoted by $a^{\frac{1}{n}}$. If a is positive, and n is odd, the equation has only one real root and is denoted by $a^{\frac{1}{n}}$. If a is negative, and n is odd, the equation has one negative solution and is denoted by $a^{\frac{1}{n}}$. If a is negative, and n is even, the equation has no real solution. (This case can be handled using complex roots, which will be ignored here.) With this, we can define $a^{\frac{m}{n}}=(a^{\frac{1}{n}})^m$.
- To generalize expoentiation to an irrational number x^*, we have to use the fact that any irrational number can be expressed as the limit of a sequence of rational numbers $x_1, x_2, ..., x_n, ...$, and define $a^x=\lim_{n\to\infty}a^{x^n}$.

Definition 9 *The Euler's number e is defined as $\lim_{n\to\infty}(1+\frac{1}{n})^n$, and $e\approx 2.718$.*

Definition 10 *The natural expoential function or simply expoential function is e^x.*

The following are some of the properties that we will be using in this course:

- Using Taylor expansion, we have $e^x=1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+....$ Thus for x sufficient small, $e^x\approx 1+x$. Similarly, since $e^{-x}=1-x+\frac{x^2}{2!}-\frac{x^3}{3!}+...$, we have $e^{-x}\approx 1-x$ for x small.
- For n large, $e^{\frac{x}{n}}\approx 1+\frac{x}{n}$.
- For n large, $e^{-\frac{1}{n}}\approx 1-\frac{1}{n}$.

- $(1-\frac{1}{n})^n = (e^{-\frac{1}{n}})^n = e^{-1}$

Example 5 *Compare the assymptotic behavior of* 2^n *and* 3^n*.*

Solution:

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} (\frac{2}{3})^n = 0$$

Thus, $2^n = o(3^n)$. □

4 Logarithmic Functions

Definition 11 *Let* a *and* x *be positive real numbers,* $a \neq 1$*. Then if* $a^y = x$*, we say that the* **log-base-**a *of* x *is* y*, and write* $\log_a x = y$*.*

Below are some important properties of logarithmic functions:
Let a, b, x, y, p be real numbers, and $a, x, y > 0$, $a, b \neq 1$ then

- $a^{\log_a x} = x$
- $\log_a (xy) = \log_a x + log_a y$
- $\log_a (x^p) = p \log_a x$
- $\log_a x = \frac{log_b x}{\log_b a}$
- Using Taylor expansion, $\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - ....$ Thus, for x succficient small, $\ln(1-x) \approx -x$.

Example 6 *Compare the assymptotic behavior of* $\log_2 n$ *and* $\log_3 n$*.*

Solution:

$$\lim_{n\to\infty} \frac{\log_2 n}{\log_3 n} = \lim_{n\to\infty} \frac{\log_2 n}{\frac{\log_2 n}{\log_2 3}} = \log_2 3$$

Thus, $\log_2 n = \Theta(\log_3 n)$. □

5 Solving Recurrences

A lot of algorithms in this course will involve recursions. As seen from the Introudction section, for recursive algroithms, their running time is often represented as a recurrence, and finding their running time involves solving recurrences.

Definition 12 *A recurrence relation is a sequence, e.g.,* $a_1, a_2, ..., a_n, ...$ *that is defined recursively, i.e., an element is defined involving the previous element or elements, along with the initial (or exit) conditions*

Example 7 *Suppose you are given a* **sorted** *array* A *of* n *real numbers, and a real number* x*. Determine if* x *is* A *or not.*

Solution: This is classic binary search problem. The first step of the algroithm is to compare the element $A[\lfloor \frac{n}{2} \rfloor]$ with x. If $A[\lfloor \frac{n}{2} \rfloor] = x$, we have found x in A. Else if $A[\lfloor \frac{n}{2} \rfloor] < x$, then we shoud recursively search $A[\lfloor \frac{n}{2} \rfloor, ..., n]$. Else we recursively search $A[1, ..., \lfloor \frac{n}{2} \rfloor]$. This immediately yields the running time $T(n)$ as: $T(n) = 1 + T(\lfloor \frac{n}{2} \rfloor)$. □

An important topic in studying the recursive functions is its solution, i.e., finding the analytic representation of each term in the sequence.

5.1 Brutefore Expansion

We have discussed the bruteforce method in the introduction section. Let's look at another example.

Example 8 *Solve the following recurrence relation:*

$$\begin{cases} T(n) = T(n-1) + n & \text{for } n \geq 2 \\ T(1) = 1 \end{cases}$$

Solution: Note that this recurrence relation can be used to model the running time of recursive insertion sort algorithm.

$$T(n) = T(n-1)+n = (T(n-2) + (n-1))+n = T(n-2)+(n-1)+n = (T(n-3) + (n-2))+(n-1)+n$$

$$= T(n-3)+(n-2)+(n-1)+n = ... = T(1)+2+3+...+n = 1+2+3+...+n = \frac{n(n+1)}{2} = \Theta(n^2)$$

□

5.2 Recursion Tree Method

Another method for bounding the asymptptic behavior of a recurrence relation is the recursion tree method.

Example 9 *Solve the following recurrence relation:*

$$\begin{cases} T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + n & \text{for } n \text{ large} \\ T(n) = 1 & \text{for } n \text{ constant} \end{cases}$$

Solution: We an use the recursion tree method to estimate the asympotic behavior of a recurrence relation. The recursion is sessnetially a tree representing an expansion of the recurrence relation. Each node in this tree represents the cost of a single recursion. Once the recursion tree is built, we will sum up the costs of the nodes for each level to obtain a per-level cost, and then sum up the cost for each level to obtain the cost of the entire recursion. Typicall, we don't need to build the entire tree. Once a few levels of the recursion is built, the per-level cost of the tree will either be aparent or can be easily inferred.

Figure 2 shows the recursion tree of the given recurrence relation. As can be inferred, the per-level costs are n, $\frac{9}{10}n$, $\left(\frac{9}{10}\right)^2 n$, ..., which are expoentially decreasing. Thus $T(n) \leq \sum_{j=0}^{\infty} \left(\frac{9}{10}\right)^j n = \frac{1-\left(\frac{9}{10}\right)^{\infty}}{1-\frac{9}{10}} n = 10n$, and $T(n) = O(n)$.

On the other hand, since $T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + n \geq n$, $T(n) = \Omega(n)$.
We end up with tight asymptotic bound $T(n) = \Theta(n)$.

□

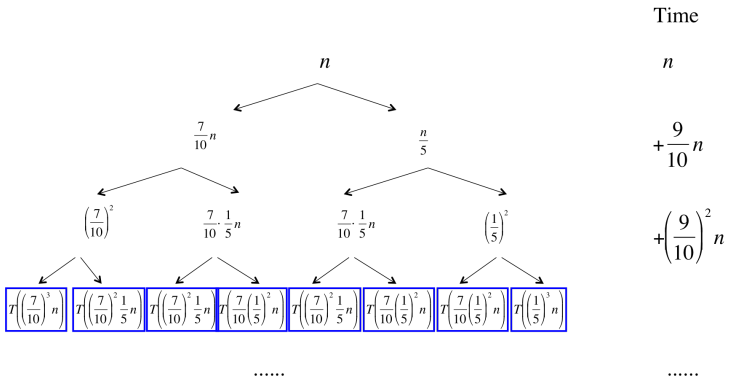

Figure 2: Illustrating the recursion tree method.

5.3 Guess and Substitution Method

The best way to illustrate the guess and substitution method is through an example.

Example 10

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + n & \text{for } n \geq 2 \\ T(n) = 1 & \text{for } n = 1 \end{cases}$$

Suppose that we guess the solution to be $T(n) = O(n \log n)$. This means there exists some positive constant c, such that $T(n) \leq cn \log n$ for all n sufficiently large (i.e., $n \geq n_0$ for some positive constant n_0. Note that this is now a claim with respect to natural numbers, and we can prove this claim using mathematical induction.

Claim 1 *For some c and n_0, we have for all $n \geq n_0$, $T(n) \leq cn \log n$.*

Proof:

Basis: $T(2) = 4 \leq c2 \log 2$ provided $c \geq 2$.

I. H.: Assume that for all $2 \leq k < n$, $T(n) \leq c \log n$.

I. S.:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq 2c\frac{n}{2}\log\frac{n}{2} + n$$
$$= cn\left(\log n - \log 2\right) + n = cn \log n - cn + n \leq cn \log n$$

The induction step holds as long as $c \geq 1$, we will have

Conclusion: Hence the entire proof goes through as long as $c \geq 2$ and $n \geq 2$.

□