G22.1170.01: Fundamental Algorithms I
Hand Out 1: Analysis of Algorithms
(Wednesday, February 19, 1986)

1 Introduction

In general, there can be several algorithms to solve a problem; and one is faced with the problem of choosing an algorithm for use. One way to differentiate between two competing algorithms is to assign a complexity measure to the algorithms. Having done this, one may choose the algorithm with the lowest complexity.

There are two kinds of complexity measures:

- *Static*: A complexity measure is static, if it does not depend on the input values. An example of such a measure is the *program length*. This measure is relevant only if the algorithm is to be used once or only a few times.

- *Dynamic*: A complexity measure is dynamic, if it varies with the input values. Some examples of such a measure are *running time* or *storage space* complexity of an algorithm as a function of the input values. This measure is relevant, when the algorithm is to be run several times on 'large' input values.

In most practical applications, the dynamic complexity measure is significant, since all production softwares are assumed to run a large number of times, and the cost of producing and maintaining the software (which depend on static complexities) are amortized rather fast. We shall use running time as our complexity measure, since almost all the algorithms we consider have a space bound that is a linear function of the input size. Furthermore, while analyzing running times, we will ignore constant factors, and will concentrate only on the orders of growth. There are two reasons for doing so:

1. This allows us to ignore details of the machine model, (such as, the hardware of the machine, the instruction sets of the computer, the memory structure of the computer, the quality of code generated by the compiler, etc.) thus giving us a *machine-independent* complexity measure.

2. For large enough problem sizes the relative efficiencies of two algorithms depend on the running times as an *asymptotic* function of input size, independent of constant factors.

We shall generally measure the running time, $T(n)$, as a function of the *worst-case* input data of size n; that is, $T(n)$ is the maximum, over all inputs of size n, of the running time on that input. The worst-case analysis provides a performance guarantee, but may be overly pessimistic, if the worst-case inputs occur seldom.

An alternative is an *average-case* analysis: we measure the running time, $T_{\text{avg}}(n)$, as the average, over all possible inputs of size n, of the running time on that input. However, such an analysis is frequently mathematically intractable. Furthermore, we must take care that our probability distribution is realistic—which may be much harder to accomplish.

Faced with these problems, algorithm-designers have suggested other average-case complexity measures; but these measures have not achieved wide-spread acceptance, and requires mathematics, outside the scope of this class. Hence, we shall restrict ourselves to worst-case complexity analysis, and occasionally, try to do the average-case analysis.

2 Big Omicron, Big Omega, Big Theta

We talk about the orders of growth of the function $T(n)$ using the functions: $O(\cdot)$ (known as *big-omicron* or more popularly, *big-oh*), $\Omega(\cdot)$ (*big-omega*) and $\Theta(\cdot)$ (*big-theta*). They are defined as follows:

- $T(n) = O(f(n))$, if there are two positive constants C and n_0 such that

$$T(n) \leq C \cdot f(n), \text{for all } n \geq n_0.$$

- $T(n) = \Omega(f(n))$ [1], if there are two positive constants C and n_0 such that

$$T(n) \geq C \cdot f(n), \quad \text{for all } n \geq n_0.$$

[1]Some define this differently: $T(n) = \Omega(f(n)$, if there is a positive constant C such that $T(n) \geq C \cdot f(n)$ infinitely often (for infinitely many values of n). This is a weaker definition, and seems more useful for lower-bound proofs—but, for all practical purposes, the stronger definition works pretty well.

- $T(n) = \Theta(f(n))$, if there are three positive constants C, C' and n_0 such that
$$C \cdot f(n) \leq T(n) \leq C' \cdot f(n), \quad \text{for all } n \geq n_0.$$

3 Algebra on O

Many of the rules of algebra work with O-notations, but some don't. Here, the main problem is that, in the expression $T(n) = O(f(n))$, '=' is a *one-way equality* (*i.e.* , it does not have the symmetry property of the equality.) For instance: $\frac{1}{2}n^2 + n = O(n^2)$ but not $O(n^2) = \frac{1}{2}n^2 + n$. (Why not?)

So how do you manipulate expressions with O's? *Very carefully.* Here are some of the simple operations you can do with the O-notation:

$$\begin{aligned}
f(n) &= O(f(n)), \\
c \cdot O(f(n)) &= O(f(n)), \qquad c \text{ is a constant}, \\
O(f(n)) + O(f(n)) &= O(f(n)), \\
O(O(f(n)) &= O(f(n)), \\
O(f((n))\, O(g(n)) &= O(f(n)\, g(n)), \\
O(f(n)\, g(n)) &= f(n)\, O(g(n)).
\end{aligned}$$

Using these, we can derive many other useful relations: Let $T_1(n) = O(f(n))$ be the running time of the program P_1, and $T_2(n) = O(g(n))$, the running time of P_2. The following program P

procedure $P(N)$;
begin
  $P_1(N); P_2(N)$
end.

has running time $T_1(n)+T_2(n)$. It is easy to show that $f(n) = O(\max(f(n), g(n)))$ and $g(n) = O(\max(f(n), g(n)))$. Using the above identities(?), we see that

$$\begin{aligned}
T_1(n) + T_2(n) &= O(f(n)) + O(g(n)) \\
&= O(O(\max(f(n), g(n)))) + O(O(\max(f(n), g(n)))) \\
&= O(\max(f(n), g(n))) + O(\max(f(n), g(n))) \\
&= O(\max(f(n), g(n)))
\end{aligned}$$

4 Analysis of Algorithms and Recurrence Relations

In general, we are interested in analyzing algorithms that is made of a set of a *mutually recursive* algorithms. Non-recursive algorithms are usually easy to analyze, and require very little mathematical sophistication. In what follows, we will concentrate on *self-recursive* algorithms, since these illustrate all the main ideas, and generalization to mutually recursive algorithms is straightforward.

Let us write self-recursive algorithm in the following schematic form:

procedure $P(N)$;
begin
  $Statement{-}1$;
  $P(N_1)$;
  $Statement{-}2$;
  $P(N_2)$;
  $Statement{-}3$;
  $\vdots$
  $Statement{-}l$;
  $P(N_l)$;
  $Statement{-}(l+1)$;
end.

A general program may be transformed into this form after resolving **if-then-else**'s and unrolling **while-loop**'s. Hence, l itself can be function of $Size(N)$. Such schemas are somewhat simplistic, but not overly restrictive. Let $Size(N) = n$, $Size(N_1) = n_1$, ..., $Size(N_l) = n_l$. Assume that the time spent by $Statement{-}1$, $Statement{-}2$, ..., $Statement{-}(l+1)$ is bounded from above by $f(n)$. Then we can write the complexity of the algorithm as follows:

$$T(n) \leq \sum_{i=1}^{l} T(n_i) + f(n).$$

Such an inequality is called a *recurrence relation.*

What we present here are, in fact, techniques to solve *recurrence equations*

of the form

$$T(n) = \sum_{i=1}^{l} T(n_i) + f(n).$$

Under a rather mild assumption, you can show that replacing the inequality by an equality does not create any problem. (This will be a home work problem.)

5 Solving the Recurrence Equations

The subject of solving recurrence equations (also called difference equations) arise in many other areas too: combinatorics, probability theory, discrete-time control theory, economics—to name a few. There are several powerful mathematical techniques available to solve these—such as, *summing factors*, *generating functions*, *z-transformations*, *operator methods*, etc. But most of these techniques are beyond our scope. We develop only a few techniques—though simple, they are sufficient for our purpose.

5.1 Guess-Work

The idea is to guess a solution, and then verify the guess by a mathematical induction over the integer. This method requires you to know the solution in order to get the solution, and does not work. Use it only when you have no other way of solving the equation—try several guesses; if you are lucky, one of them will work.

Example. 5.1 *Consider the following recurrence equation:*

$$T(n) = \begin{cases} c_1, & \text{if } n = 1; \\ 2\,T(\frac{n}{2}) + c_2 n, & \text{if } n > 1. \end{cases} \tag{1}$$

Suppose we guess that $T(n) = c_2\, n \lg n + c_1\, n$, then by mathematical induction over all positive integers, we can show the validity of our guess. Hence $T(n) = O(n \lg n)$. It is left as an exercise for you to do the steps of the inductive proof. □

5.2 Expansion

Another way to solve a recurrence relation is to repeatedly expand the terms as many times as possible. When this process stops, we will be left with a sum consisting of functions of $f(\cdot)$ and lower values of $T(\cdot)$. In many cases the summation can be evaluated to obtain a close form solution. This method is rather messy, and prone to error—avoid it, in all but a few simple cases.

Example. 5.2 *Consider the following recurrence equation:*

$$T(n) = \begin{cases} c_1, & \text{if } n = 1; \\ 2\,T(\frac{n}{2}) + c_2 n, & \text{if } n > 1. \end{cases} \tag{2}$$

First rearrange the recursive part as

$$T(n) = c_2 n + 2\,T\left(\frac{n}{2}\right).$$

Expanding the recursive definition of $T(\frac{n}{2})$ shows that

$$T(n) = c_2 n + 2\left[c_2 \frac{n}{2} + 2\,T\left(\frac{n}{4}\right)\right].$$

Multiplying and canceling yields

$$T(n) = c_2 n + c_2 n + 4\,T\left(\frac{n}{4}\right).$$

Applying the recursive definition of $T(\frac{n}{4})$ to above shows that

$$T(n) = c_2 n + c_2 n + c_2 n + 8\,T\left(\frac{n}{8}\right).$$

This process of expansion and cancellation can be iterated $\lg n$ times to yield

$$T(n) = \left[\sum_{i=1}^{\lg n} c_2 n\right] + n \cdot T(1) = c_2\, n \lg n + c_1\, n. \qquad \square$$

5.3 Recursion Tree

Another way to solve the recurrence equation of the form

$$T(n) = \begin{cases} T(n_0), & \text{if } n = n_0; \\ T(n_1) + T(n_2) + \cdots + T(n_l) + f(n), & \text{if } n > n_0, \end{cases} \tag{3}$$

is by expanding the terms of it in a tree like manner. The tree corresponding to this expansion is called a *recursion tree*, and is defined as follows. The recursion tree for the equation **??** at the value n is:

- *a single node with the value $T(n_0)$, if $n = n_0$;*
- *otherwise, a node with the value $f(n)$ and l sons such that the i^{th} son is the root of a recursion tree of the equation* **??** *at n_i.*

It is easy to see that $T(n)$ is just the sum of values at the nodes.

Example. 5.3 *Consider the following recurrence equation:*

$$T(n) = \begin{cases} c_1, & \text{if } n = 1; \\ 2\,T(\frac{n}{2}) + c_2 n, & \text{if } n > 1. \end{cases} \tag{4}$$

The recursion tree for the equation **??** is as shown below.

You may prove the following facts about the above recursion tree:

1. The tree has a depth of $\lg n + 1$. The sum of values at all the internal nodes at level i ($0 \leq i < \lg n$) is $c_2 n$. Thus

$$\sum_{j=\text{internal}-\text{node}} \text{value}(j) = c_2\, n \lg n.$$

2. The tree has n leaves. The value of each leaf node is c_1.

$$\sum_{j=\text{leaf}-\text{node}} \text{value}(j) = c_1 n.$$

Hence

$$T(n) = \sum_{j=\text{node}} \text{value}(j) = c_2\, n \lg n + c_1\, n. \quad \square$$

5.4 Telescoping

The idea of telescoping or summing factors is rather simple, and is developed step-by-step in this and the next section. We start out by showing how to solve recurrence equations of very simple form using this technique. But this technique, when combined with the transformation techniques of the next section, is quite powerful and handles almost all equations. Please try to master this technique by trying to solve several recurrence equations—*this is the official technique for this course*.

Consider the recurrence equation

$$T(n) = \begin{cases} 1, & \text{if } n = 0; \\ T(n-1) + 1, & \text{if } n \geq 1. \end{cases} \tag{5}$$

We can write down the above equation as follows:

$$\begin{array}{llllll} T(n) - & T(n-1) & & & & = 1 \\ & T(n-1) - & T(n-2) & & & = 1 \\ & & & \vdots & & \\ & & & T(1) - & T(0) & = 1 \\ \hline T(n) & & & & -T(0) & = n \end{array}$$

The simplification is obtained by cancelling $-T(i)$ of the current line by the $T(i)$ of the next line. Thus when we added up all the equations most of the

terms canceled out. We say the sum *telescopes*. Note that the final answer is $T(n) = n + T(0) = n + 1$.

Finding the solution above was almost trivial. Sometimes however we have to use a little trickery to make the sum of successive equations telescope. Suppose that our recurrence is

$$\begin{array}{ll} a_0 T(0) & = c_0; \\ a_n T(n) & = b_n T(n-1) + c_n, \quad \text{if} n \geq 1. \end{array} \tag{6}$$

where a_n, b_n and c_n are given. If we try to add to successive copies of this equation, for example,

$$\begin{array}{llll} a_n T(n) - & b_n T(n-1) & & = c_n \\ & a_{n-1} T(n-1) - & b_{n-1} T(n-2) & = c_{n-1} \end{array}$$

then the $T(n-1)$ terms do not cancel since b_n is not equal to a_{n-1}. Note however that we could multiply the equations by some factor which would make the $T(n-1)$ terms cancel and the sum will telescope. Let us multiply the n^{th} equation by some (so far unspecified) factor f_n to obtain

$$\begin{array}{llll} f_n a_n T(n) - & f_n b_n T(n-1) & & = f_n c_n \\ & f_{n-1} a_{n-1} T(n-1) - & f_{n-1} b_{n-1} T(n-2) & = f_{n-1} c_{n-1} \end{array}$$

In order to guarantee that the $T(n-1)$ terms cancel we require that

$$f_n b_n = f_{n-1} a_{n-1}$$

or

$$f_n = f_{n-1} \left(\frac{a_{n-1}}{b_n} \right).$$

By unrolling the above equation we get:

$$f_n = f_{n-1} \left(\frac{a_{n-1}}{b_n} \right) = f_{n-2} \left(\frac{a_{n-2}}{b_{n-1}} \right) \left(\frac{a_{n-1}}{b_n} \right) = \cdots = f_0 \frac{\prod_{i=0}^{n-1} a_i}{\prod_{i=1}^{n} b_i}.$$

Fixing $f_0 = 1$, we get

$$\begin{array}{ll} f_0 & = 1; \\ f_n & = \frac{\prod_{i=0}^{n-1} a_i}{\prod_{i=1}^{n} b_i}, \quad \text{if } n \geq 1. \end{array} \tag{7}$$

f_n is called the *summing factor* for the equation **??**. Let us write

$$R(n) = f_n a_n T(n).$$

Since

$$f_n a_n T(n) = f_n b_n T(n-1) + f_n c_n = f_{n-1} a_{n-1} T(n-1) + f_n c_n,$$

this yields

$$\begin{array}{ll} R(0) & = c_0; \\ R(n) & = R(n-1) + c_n f_n, \quad \text{if } n \geq 1. \end{array}$$

Now the telescopy will work.

$$\begin{array}{llll} R(n)- & R(n-1) & & = c_n f_n \\ & R(n-1)- & R(n-2) & = c_{n-1} f_{n-1} \\ & & \vdots & \\ & & R(1)- \quad R(0) & = c_1 f_1 \\ \hline R(n) & & -R(0) & = \sum_{j=1}^{n} c_j f_j \end{array}$$

Hence

$$R(n) = f_n a_n T(n) = \left[c_0 + \sum_{j=1}^{n} c_j f_j \right],$$

and

$$T(n) = \frac{1}{f_n a_n} \left[c_0 + \sum_{j=1}^{n} c_j f_j \right]$$

simplifying,

$$T(n) = \left(\frac{\prod_{i=1}^{n} b_i}{\prod_{i=0}^{n} a_i} \right) \left[c_0 + \sum_{j=1}^{n} \frac{c_j \prod_{i=0}^{j-1} a_i}{\prod_{i=1}^{j} b_i} \right]. \tag{8}$$

Example. 5.4 *Consider the following recurrence equation:*

$$T(n) = \begin{cases} 1, & \text{if } n = 0; \\ 2\,T(n-1) + 2^n, & \text{if } n \geq 1. \end{cases} \tag{9}$$

This is same as

$$\begin{array}{ll} 1 \cdot T(0) & = 1; \\ 1 \cdot T(n) - 2 \cdot T(n-1) & = 2^n, \quad \text{if} n \geq 1. \end{array}$$

This is an example of **??** with

$$a_n = 1, \quad b_n = 2.$$

By **??** we may choose

$$\begin{array}{ll} f_0 & = 1; \\ f_n & = \frac{1}{2^n}, \quad \text{if } n \geq 1. \end{array}$$

Multiplying by f_n $(\frac{1}{2^n})$ gives the new equation

$$\begin{array}{ll} T(0) & = 1; \\ \frac{T(n)}{2^n} - \frac{T(n)}{2^{n-1}} & = 1, \quad \text{if } n \geq 1. \end{array}$$

Using telescopy we get

$$\begin{array}{lllll} \frac{T(n)}{2^n} - & \frac{T(n-1)}{2^{n-1}} & & & = 1 \\ \frac{T(n-1)}{2^{n-1}} - & \frac{T(n-2)}{2^{n-2}} & & & = 1 \\ & & \vdots & & \\ & & \frac{T(1)}{2} - & \frac{T(0)}{1} & = 1 \\ \hline \frac{T(n)}{2^n} & & & -T(0) & = n \end{array}$$

Hence

$$T(n) = (n+1)2^n. \quad \square$$

5.5 Range and Domain Transformations

Sometimes it is useful to apply a transformation to a sequence in order to make it appear in a more desirable form. The function T maps the integer n in its *domain* to a real $T(n)$ in its *range*. We call a transformation on the values of the sequence $T(n)$ a *range transformation* and a transformation on the indices n a *domain transformation*. We illustrate the ideas using examples:

Domain Transformation

Example. 5.5 *Consider the following recurrence equation:*

$$T(n) = \begin{cases} c_1, & \text{if } n = 1; \\ 2\,T(\frac{n}{2}) + c_2 n, & \text{if } n > 1. \end{cases} \tag{10}$$

Let us use the following domain transformation first

$$n = 2^k, \quad (k = \lg n) \quad \text{hence } S(k) = T(n) = T(2^k).$$

The above equation becomes

$$S(k) = \begin{cases} c_1, & \text{if } k = 0; \\ 2\,S(k-1) + c_2 2^k, & \text{if } k \geq 1. \end{cases}$$

Now we can use telescoping to get the following solution:

$$S(k) = T(2^k) = 2^k \left(c_2 k + c_1\right).$$

Back-substituting n for 2^k and $\lg n$ for k, we conclude that

$$T(n) = n(c_2 \lg n + c_1). \quad \square$$

Range Transformation

Example. 5.6 *Consider the following (pseudo-non-linear) recurrence equation:*

$$T(n) = \begin{cases} 1, & \text{if } n = 0; \\ 3\,\left(T(n-1)\right)^2, & \text{if } n \geq 1. \end{cases} \tag{11}$$

Let us use the following range transformation first:

$$S(n) = \log_3 T(n), \quad \left(3^{S(n)} = T(n)\right).$$

We may rewrite the recurrence as

$$S(n) = \begin{cases} 0, & \text{if } n = 0; \\ 2\,S(n-1) + 1, & \text{if } n \geq 1. \end{cases}$$

Using multiplier $f_n = \frac{1}{2^n}$, we get

$$\begin{array}{ll} \frac{S(0)}{2^0} & = 0 \\ \frac{S(n)}{2^n} - \frac{S(n-1)}{2^{n-1}} & = \frac{1}{2^n} \end{array}$$

Using telescopy, we get

$$\frac{S(n)}{2^n} - \frac{S(0)}{2^0} = \sum_{i=1}^{n} \left(\frac{1}{2^i}\right) = \frac{(1/2) - (1/2)^{n+1}}{1 - (1/2)} = 1 - \left(\frac{1}{2}\right)^n$$

Hence,

$$S(n) = 2^n - 1.$$

Hence,

$$T(n) = 3^{S(n)} = 3^{2^n - 1}. \quad \square$$

6 Examples

6.1 Example: Divide-and-Conquer

An important class of recurrence equations results from algorithms based on the '*divide-and-conquer*' paradigm; such algorithms consist of three steps:

1. DIVIDE: Break the problem of size n into a smaller subproblems each of size $\frac{n}{b}$, using no more than $g(n)$ time.
2. RECUR: Recursively solve each of the a subproblems using $a \cdot T(\frac{n}{b})$ time.
3. MARRY: Combine the solutions to the subproblems to get a solution to the original problem, using no more than $h(n)$ time.

The recurrence equation is of the form:

$$T(n) = a \cdot \left(\frac{n}{b}\right) + f(n), \quad \text{where } f(n) = g(n) + h(n). \tag{12}$$

Use the following domain-transformation first

$$n = b^m \quad (m = \log_b n).$$

The above equation becomes

$$T(b^m) = a \cdot T(b^{m-1}) + f(b^m).$$

Using the multiplying factor $f_m = (\frac{1}{a^m})$ we get

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \frac{f(b^m)}{a^m}.$$

Using telescopy

$$\begin{array}{llll}
\frac{T(b^m)}{a^m} - & \frac{T(b^{m-1})}{a^{m-1}} & & = \frac{f(b^m)}{a^m} \\
\frac{T(b^{m-1})}{a^{m-1}} - & \frac{T(b^{m-2})}{a^{m-2}} & & = \frac{f(b^{m-1})}{a^{m-1}} \\
& & \vdots & \\
& & \frac{T(b)}{a} - \quad \frac{T(1)}{1} & = \frac{f(b)}{a} \\
\hline
\frac{T(b^m)}{a^m} & & -T(1) & = \sum_{i=1}^{m} \frac{f(b^i)}{a^i}
\end{array}$$

Hence

$$T(b^m) = a^m \left[T(1) + \sum_{i=1}^{m} \frac{f(b^i)}{a^i} \right].$$

ASSUMPTION f is a multiplicative function; that is, f has the following property

$$f(m\,n) = f(m)\,f(n).$$

Example of a multiplicative function is $f(n) = n^c$. Hence

$$f(b^i) = (f(b))^1.$$

Rewrite the previous function as

$$T(b^m) = a^m \left[T(1) + \sum_{i=1}^{m} \left(\frac{f(b)}{a} \right)^i \right].$$

Let

$$\alpha = \left(\frac{f(b)}{a} \right).$$

Then

$$T(n) = a^m \left[T(1) + \sum_{i=1}^{m} \alpha^i \right] = a^m \left[T(1) + \alpha \frac{\alpha^m - 1}{\alpha - 1} \right].$$

There are three cases to consider

- CASE.1 *$a > f(b)$; that is, $\alpha < 1$. Then*

$$T(n) = a^m \cdot O(1) = O\left(a^m\right) = O\left(a^{\log_b n}\right) = O\left(n^{\log_b a}\right).$$

- CASE.2 *$a = f(b)$; that is, $\alpha = 1$. Then*

$$T(n) = a^m \cdot O(m) = O\left(a^m m\right) = O\left(a^{\log_b n} \log_b n\right) = O\left(n^{\log_b a} \log_b n\right).$$

- CASE.3 *$a < f(b)$; that is, $\alpha > 1$. Then*

$$T(n) = a^m \cdot O\left(\alpha^m\right) = O\left(f(b)^m\right) = O\left(f(b)^{log_b n}\right) = O\left(n^{\log_b f(b)}\right).$$

6.2 Example: Randomized Divide-and-Conquer

Let us consider a randomized algorithm (*i.e.* algorithm involving coin-tosses), also based on the '*divide-and-conquer*' paradigm; such an algorithm consists of three steps:

1. RANDOM-DIVIDE: Break the problem of size n into 2 smaller subproblems as follows: first choose a number i ($1 \leq i \leq n-1$) with probability $\frac{1}{n-1}$; let the first subproblem be the one consisting of the first i elments, and the second, consisting of the remaining $n-i$ elements. The division step takes no more than $g(n)$ amount of time.

2. RECUR: Recursively solve each of the 2 subproblems using $T_{\text{avg}}(i) + T_{\text{avg}}(n-i)$ time.

3. MARRY: Combine the solutions to the subproblems to get a solution to the original problem, using no more than $h(n)$ time.

Let us assume that $g(n) + h(n) = c \cdot n$.

Hence the recurrence equation will be

$$T_{\text{avg}}(n) = \sum_{i=1}^{n-1} \left(\frac{1}{n-1} \left[T_{\text{avg}}(i) + T_{\text{avg}}(n-i) \right] \right) + c \cdot n.$$

This can be rwritten as

$$\begin{aligned} T_{\text{avg}}(n) &= \tfrac{1}{n-1} \textstyle\sum_{i=1}^{n-1} \left(T_{\text{avg}}(i) + T_{\text{avg}}(n-i) \right) + c \cdot n \\ &= \tfrac{1}{n-1} \left(\textstyle\sum_{i=1}^{n-1} T_{\text{avg}}(i) + \sum_{i=1}^{n-1} T_{\text{avg}}(n-i) \right) + c \cdot n \\ &= \tfrac{1}{n-1} \left(\textstyle\sum_{i=1}^{n-1} T_{\text{avg}}(i) + \sum_{j=1}^{n-1} T_{\text{avg}}(j) \right) + c \cdot n \\ &= \tfrac{2}{n-1} \left(\textstyle\sum_{i=1}^{n-1} T_{\text{avg}}(i) \right) + c \cdot n. \end{aligned}$$

Or

$$(n-1)\, T_{\text{avg}}(n) = 2 \left(\sum_{i=1}^{n-1} T_{\text{avg}}(i) \right) + c \cdot n(n-1). \tag{13}$$

Substituting $n+1$ for n, we get

$$n\, T_{\text{avg}}(n+1) = 2 \left(\sum_{i=1}^{n} T_{\text{avg}}(i) \right) + c \cdot n(n+1). \tag{14}$$

Subtracting equation **??** from equation **??**,

$$n\,T_{\text{avg}}(n+1) - (n-1)\,T_{\text{avg}}(n) = 2\,T_{\text{avg}}(n) + 2c \cdot n.$$

Simplifying the above equation

$$n\,T_{\text{avg}}(n+1) = (n+1)\,T_{\text{avg}}(n) + 2c \cdot n.$$

Or

$$\frac{T_{\text{avg}}(n+1)}{n+1} - \frac{T_{\text{avg}}(n)}{n} = \frac{2c}{n+1}.$$

Using telescopy we get

$$\begin{array}{llll}
\frac{T_{\text{avg}}(n+1)}{n+1} - & \frac{T_{\text{avg}}(n)}{n} & & = \frac{2c}{n+1} \\
\frac{T_{\text{avg}}(n)}{n} - & \frac{T_{\text{avg}}(n-1)}{n-1} & & = \frac{2c}{n} \\
& \vdots & & \\
& \frac{T_{\text{avg}}(2)}{2} - & \frac{T_{\text{avg}}(1)}{1} & = \frac{2c}{2} \\
\hline
\frac{T_{\text{avg}}(n+1)}{n+1} & & -T(1) & = 2c\left\{\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1}\right\}
\end{array}$$

Hence

$$T_{\text{avg}}(n+1) = (n+1)\left[T_{\text{avg}}(1) + 2c\,(H_{n+1} - 1)\right].$$

Or,

$$T_{\text{avg}}(n) = n\left[T_{\text{avg}}(1) + 2c\,(H_n - 1)\right].$$

The approximate size of the n^{th} harmonic number H_n is a well-known quantity

$$H_n = \ln n + \gamma + \frac{1}{2n} - O\left(n^{-2}\right).$$

Here $\gamma = 0.57721\ 56649\ldots$ is *Euler's constant*. Hence

$$\begin{aligned}
T_{\text{avg}}(n) &= n\left[2c\left(\ln n + \gamma + \tfrac{1}{2n} - O\left(n^{-2}\right)\right) + T_{\text{avg}}(1) - 2c\right] \\
&= (2\ln 2)c\,n\,\lg n + (2c(\gamma - 1) + T_{\text{avg}}(1))n + c - O(n^{-1}) \\
&= 1.39c\,n\,\lg n + \left[T_{\text{avg}}(1) - 0.85c\right]n + c - O(n^{-1}) \\
&= O(n \lg n).
\end{aligned}$$