Lecture 7-Recurrence Relation & Summation

Mathematical basis

A **logarithm** of base b for value y is the power to which b is raised to get y. Normally, this is written as $\log_b y = x$. Thus, if $\log_b y = x$ then $b^x = y$, and $b^{\log_b y} = y$.

Logarithms are used frequently by programmers. Here are two typical uses.

Example 2.6 Many programs require an encoding for a collection of objects. What is the minimum number of bits needed to represent n distinct code values? The answer is $\lceil \log_2 n \rceil$ bits. For example, if you have 1000 codes to store, you will require at least $\lceil \log_2 1000 \rceil = 10$ bits to have 1000 different codes (10 bits provide 1024 distinct code values).

Example 2.7 Consider the binary search algorithm for finding a given value within an array sorted by value from lowest to highest. Binary search first looks at the middle element and determines if the value being searched for is in the upper half or the lower half of the array. The algorithm then continues splitting the appropriate subarray in half until the desired value is found. (Binary search is described in more detail in Section 3.5.) How many times can an array of size n be split in half until only one element remains in the final subarray? The answer is $\lceil \log_2 n \rceil$ times.

Summations and Recurrences: Most programs contain loop constructs. When analyzing running time costs for Programs with loops, we need to add up the costs for each time the loop is executed. This is an example of a summation. Summations are simply the sum of costs for some function applied to a range of parameter values. Summations are typically written with the following "Sigma" notation:

$$\sum_{i=1}^{n} f(i).$$

This can also be written f(1)+ f(2)+···+ f(n−1)+ f(n). Within a sentence. Given a summation, you often wish to replace it with a direct equation with the same value as the summation. This is known as a closed-form solution, and the process of replacing the summation with its closed-for m solution is known as solving the summation. The following is a list of useful summations, along with their closed-form solutions.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \quad (2.1)$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} = \frac{n(2n+1)(n+1)}{6}. \quad (2.2)$$

$$\sum_{i=1}^{\log n} n = n \log n. \quad (2.3)$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ for } 0 < a < 1. \quad (2.4)$$

Example2

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = 0; j < i; j++)
      count++;
```

Lets see how many times **count++** will run.

When $i = 0$, it will run 0 times.

When $i = 1$, it will run 1 times.

When $i = 2$, it will run 2 times and so on.

Total number of times **count++** will run is $0+1+2+\ldots+(N-1) = \frac{N*(N-1)}{2}$. So the time complexity will be $O(N^2)$.

The running time for a recursive algorithm is most easily expressed by a recursive expression because the total time for the recursive algorithm includes the time to run the recursive call(s). A recurrence relation defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the recursive definition for the factorial function:

$n! = (n-1)!\cdot n$ for $n > 1$; $1! = 0! = 1$.

Another standard example of a recurrence is the Fibonacci sequence:

$\mathrm{Fib}(n) = \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2)$ for $n > 2$; $\mathrm{Fib}(1) = \mathrm{Fib}(2) = 1$.

From this definition we see that the first seven numbers of the Fibonacci sequence are 1,1,2,3,5,8, and 13. Notice that this definition contains two parts: the general definition for Fib(n) and the base cases for Fib (1) and Fib (2). Likewise, the definition for factorial contains a recursive part and base cases. Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by function fact of for an input of size n will be zero when n = 0or n = 1(the base cases), and it will be one plus the cost of calling faction a value of n−1. This can be defined using the following recurrence: T(n) = T(n−1) +1 for n > 1; T(0) = T(1) = 0. As with summations, we typically wish to replace the recurrence relation with a closed-form solution. One approach is to expand the recurrence by replacing any occurrences of T on the right-hand side with its definition.

Example 2.8 If we expand the recurrence $\mathbf{T}(n) = \mathbf{T}(n-1) + 1$, we get

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-1) + 1 \\ &= (\mathbf{T}(n-2) + 1) + 1.\end{aligned}$$

We can expand the recurrence as many steps as we like, but the goal is to detect some pattern that will permit us to rewrite the recurrence in terms of a summation. In this example, we might notice that

$$(\mathbf{T}(n-2) + 1) + 1 = \mathbf{T}(n-2) + 2$$

and if we expand the recurrence again, we get

$$\mathbf{T}(n) = \mathbf{T}(n-2) + 2 = \mathbf{T}(n-3) + 1 + 2 = \mathbf{T}(n-3) + 3$$

which generalizes to the pattern $\mathbf{T}(n) = \mathbf{T}(n-i) + i$. We might conclude that

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-(n-1)) + (n-1) \\ &= \mathbf{T}(1) + n - 1 \\ &= n - 1.\end{aligned}$$

Solving Recurrences with the Iteration/Recursion-tree Method

- In the iteration method we iteratively "unfold" the recurrence until we "see the pattern".
- The iteration method does not require making a good guess like the substitution method (but it is often more involved than using induction).
- Example: Solve $T(n) = 8T(n/2) + n^2$ $(T(1) = 1)$

$$\begin{aligned}
T(n) &= n^2 + 8T(n/2) \\
&= n^2 + 8(8T(\frac{n}{2^2}) + (\frac{n}{2})^2) \\
&= n^2 + 8^2T(\frac{n}{2^2}) + 8(\frac{n^2}{4})) \\
&= n^2 + 2n^2 + 8^2T(\frac{n}{2^2}) \\
&= n^2 + 2n^2 + 8^2(8T(\frac{n}{2^3}) + (\frac{n}{2^2})^2) \\
&= n^2 + 2n^2 + 8^3T(\frac{n}{2^3}) + 8^2(\frac{n^2}{4^2})) \\
&= n^2 + 2n^2 + 2^2n^2 + 8^3T(\frac{n}{2^3}) \\
&= \dots \\
&= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \dots
\end{aligned}$$

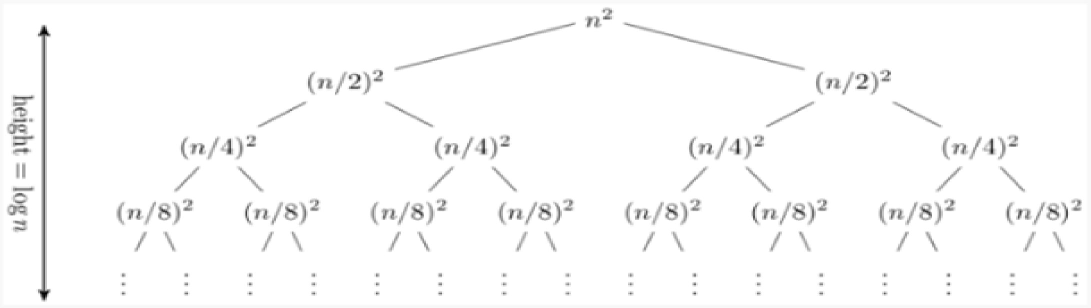
  - Recursion depth: How long (how many iterations) it takes until the subproblem has constant size? i times where $\frac{n}{2^i} = 1 \Rightarrow i = \log n$

A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

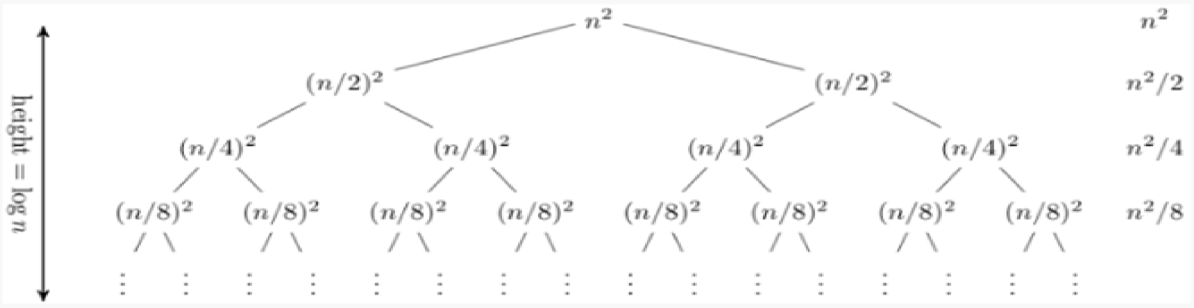
For instance, consider the recurrence

$T(n) = 2T(n/2) + n^2$.

The recursion tree for this recurrence has the following form:



In this case, it is straightforward to sum across each row of the tree to obtain the total work done at a given level:



This a geometric series, thus in the limit the sum is $O(n^2)$. The depth of the tree in this case does not really matter; the amount of work at each level is decreasing so quickly that the total is only a constant factor more than the root.