Introduction to Recurrences

- **The question:** Why do we care about solving recurrences in a course about algorithms?
- **The answer:**
  - As we have seen, many algorithms can be implemented recursively.
  - Because of this, the run-time of these algorithms can be described in terms of a recurrence relation.
  - Thus, we will need to be able to solve recurrences in order to determine the run-time of many algorithms.

Example: Factorial

- We have seen the *factorial* function before:

$$n! = \begin{cases} 1 & \text{when } n = 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

- We have also seen the C++ implementation:

```
int factorial(int n) {
   if (n==1)
      return 1;
   else
      return n*factorial(n-1);
   }
```

- It is not hard to see that the run-time of factorial, $T(n)$ is given by

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

- The notation T_n is equivalent to $T(n)$.

What is a Recurrence Equation?

- As we have seen, a *recursive* function is one that is defined in terms of itself.
- Similarly, a *recurrence* is an equation which is defined in terms of itself.
- In the previous example, *factorial* is a recursive function while the time complexity of this function is expressed in terms of a recurrence:

$$n! = \begin{cases} 1 & \text{when } n = 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

- We can observe that both have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces.
- The boundary condition terminate the recurrence.

Understanding Recurrences

- Recurrences have 2 types of terms: The *recursive* term(s) and the *non-recursive* terms.
  - The *recursive* terms are those terms that refer to the recurrence.
  - The *non-recursive* terms are those terms that do not refer to the recurrence.
  - **Example:** In the recurrence

    $$R(n) = 2R(n/2) + n^2 + 3,$$

    the recursive term is $2R(n/2)$, and the non-recursive term is $n^2 + 3$.
- How do these terms relate to algorithms?
  - Recursive terms come from when a recursive function calls itself.
  - Non-recursive terms come from the other work done in the subroutine, including any splitting or combining of data that must be done.

Example: Binary Search

- We want to find a value v in a sorted array of size n.
  - We compare the middle value m of the array to v.
  - If the $m = v$, we are done.
  - Else if $m < v$, we search the left half of the array.
  - Else ($m > v$), we search the right half of the array.
  - Now, we have the same problem, but only half the size.
- To solve an instance of binary search, we need to do a compare operation, followed by an instance of binary search of half the size.
- Thus, the recurrence for the (worst-case) run-time of this algorithm is

$$T(n) \leq 1 + T(\lfloor \frac{n}{2} \rfloor)$$

Recurrence Solutions

- To compute the nth value of the recurrence, we usually need to compute the first $n-1$ terms first.
- A *solution* to a recurrence is an equivalent equation which is not expressed in terms of itself. Given a solution, we can compute the nth term directly.
- **Example:**
  - Consider the recurrence equation:

    $$T(n) = 2 * T(n-1) + 1,$$

    with boundary condition $T(0) = 0$.
  - Some small values of $T(n)$ can easily be computed:

n	0	1	2	3	4	5	6	7
$T(n)$	0	1	3	7	15	31	63	127

  - It is not too hard to see that $T(n) = 2^n - 1$. This is a solution to the recurrence.

Solving Recurrences

There is no general method to solve recurrences. There are several strategies one can try:

1. The *Characteristic Equation* for homogeneous linear recurrences of the form

   $$a_0 T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = 0$$

   for which the characteristic equation is:

   $$a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0$$

2. The *Substitution Method*.

3. The *Iteration Method*.

4. The *Master Method* which applies to recurrences of the form

   $$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

   where $a \geq 1$ and $b > 1$ are constants.

The Substitution Method

- This would be better called the *guess and prove it by induction* method.
- **Example:** Consider the recurrence

$$S(n) = \begin{cases} 1 & \text{when } n = 1 \\ S(n-1) + n & \text{otherwise} \end{cases}$$

Then the solution is $S(n) = \dfrac{n(n+1)}{2}$.

- **Proof:** When $n = 1$, $S(1) = 1(2)/2 = 1$.
Assume that for $n < k$, $S(n) = \frac{n(n+1)}{2}$. Then

$$\begin{aligned} S(k) &= S(k-1) + k = \frac{(k-1)(k)}{2} + k \\ &= \frac{k^2 - k}{2} + k = \frac{k^2 - k + 2k}{2} \\ &= \frac{k^2 + k}{2} = \frac{k(k+1)}{2} \end{aligned}$$

Thus, $S(n) = \frac{n(n+1)}{2}$ for all $n \geq 1$.

The Iteration Method

- With the iteration method, we expand the recurrence and express it as a summation dependent only on n and initial conditions. Then we evaluate the summation.
- **Example:** Solve the recurrence

$$R(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2R(n/2) + n/2 & \text{otherwise} \end{cases}$$

- **Solution:** We have

$$\begin{aligned} R(n) &= 2R(n/2) + n/2 \\ &= 2(2R(n/4) + n/4) + n/2 = 4R(n/4) + n \\ &= 4(2R(n/8) + n/8) + n = 8R(n/8) + 3n/2 \\ &\vdots \\ &= 2^k R(n/(2^k)) + kn/2 \\ &= 2^{\log_2 n} R(n/(2^{\log_2 n})) + (\log_2 n)n/2 \\ &= nR(n/n) + (\log_2 n)n/2 \\ &= nR(1) + (\log_2 n)n/2 = O(n \log n) \end{aligned}$$

The Iteration Method: Example 2

- Solve the recurrence

$$H_n = \begin{cases} 1 & \text{when } n = 1 \\ 2H_{n-1} + 1 & \text{otherwise} \end{cases}$$

- **Solution:**

$$\begin{aligned} H_n &= 2H_{n-1} + 1 \\ &= 2(2H_{n-2} + 1) + 1 = 2^2 H_{n-2} + 2 + 1 \\ &= 2^2(2H_{n-3} + 1) + 2 + 1 = 2^3 H_{n-3} + 2^2 + 2 + 1 \\ &\vdots \\ &= 2^{n-1} H_1 + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

- Thus, $H_n = 2^n - 1$.

The Master Method

- Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

where n/b can mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$
   for some constant $\epsilon > 0$,
   then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$,
   then $T(n) = \Theta(n^{\log_b a} \log n)$.

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$
   for some constant $\epsilon > 0$, and
   if $af(n/b) \leq cf(n)$ for some constant $c < 1$
   and all sufficiently large n,
   then $T(n) = \Theta(f(n))$.

Example 1 of Master Method

- Solve the recurrence

$$T(n) = 4T(n/2) + n.$$

- **Solution:** We have $a = 4, b = 2$, and $f(n) = n$. Thus, $\log_b a = \log_2 4 = 2$. Can we find $\epsilon > 0$ such that

$$n = O(n^{\log_b a - \epsilon}) = O(n^{2-\epsilon})?$$

  If we pick $\epsilon = .5$, it is clear that

$$n = O(n^{2-\epsilon}) = O(n^{1.5}).$$

  Therefore case 1 applies and

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

Example 2 of Master Method

- Solve the recurrence

$$T(n) = 4T(n/2) + n^2.$$

- **Solution:** We have $a = 4, b = 2$, and $f(n) = n^2$. Thus, $\log_b a = \log_2 4 = 2$. Can we find $\epsilon > 0$ such that

$$n^2 = O(n^{\log_a b - \epsilon}) = O(n^{2-\epsilon})?$$

Absolutely not. However,

$$n^2 = \Theta(n^2) = \Theta(n^{\log_b a}),$$

so case 2 applies, and

$$T(n) = \theta(n^2 \log n).$$

Example 3 of Master Method

- Solve the recurrence

$$T(n) = 4T(n/2) + n^3.$$

- **Solution:** Here, $a = 4, b = 2$, and $f(n) = n^3$. From the previous 2 examples, it should be clear that cases 1 and 2 won't apply, so we'll try case 3. We need to show two things:
  - We need to find $\epsilon > 0$ such that

$$n^3 = \Omega(n^{\log_2 4+\epsilon}) = \Omega(n^{2+\epsilon}).$$

    We can pick any $0 < \epsilon \leq 1$.
  - Now, we need to find $c < 1$ such that

$$4(n/2)^3 = \frac{1}{2}n^3 \leq cn^3$$

    for sufficiently large n.
    This is true for for all n if we pick $c = 1/2$.

  So case 3 applies, and

$$T(n) = \Theta(n^3).$$