Recursion for computation

A computation model is *Turing complete* when it can compute everything that can be computed by a Turing machine.

Pragmatically, a model (or a language) is Turing complete if it can do
- sequence
- branch
- repetition (either as loop or as recursion)

Recursion
- is as *powerful* as iteration in establishing a Turing complete model.
- is proof-friendly for proving correctness of algorithms. (Thus promoted in functional programming languages, such as ML).
  - Why?
- is also efficient.
  - Myth: Loop is much faster than recursion
  - Truth: recursion can be as efficient as iteration.

Note: any algorithm using recursion can be converted to using iterations, and vice versa.
Fibonacci numbers

population dynamics: “how quickly would population of rabbits expand under appropriate conditions?”

conditions:
- a pair of rabbits has a pair of children every year
- These children too young to have their own children until two years later
- Rabbits never die

F(1) = 1 (start with one pair)
F(2) = 1 (too young to have children first year)
F(3) = 2 (in 2nd year, one pair of children)
F(4) = 3 (in 3rd year, another pair of children)
F(5) = 5 (first pair of grandchildren, plus another pair of children)
...

In general, F(n) = F(n-1) + F(n-2), namely, all rabbits at previous year are still there, plus one pair of children for every pair of rabbits we had two years ago.

Algorithm 3.1

1 int fib(int n)
2 if (n<= 2) return 1
3 else return fib(n-1) + fib(n-2)

Time complexity (a local view)

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

Recursive tree for fib(5) call

Two lines code executed for each internal node (only one when on leaves)
Theorem 3.2 In a computation without while or for loops, but possibly with recursive procedure calls, the total computation time is $\Theta(C)$, where $C$ is the total number of procedure calls.

**Proof:** $C$ is the number of internal nodes plus the number of leaves in the recursive tree. □

Example: Algorithm 3.1 has time complexity

$T(n) = 2 \times \# \text{ of internal nodes} + 1 \times \# \text{ of leaves}$

# of leaves = $\text{fib}(n)$, just the $n$-th fibonacci number!
# of internal nodes = # of leaves -1, a property of binary trees.

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

---

**Iterations can be converted as recursions**

For example, Sequential Search can be implemented recursively

```c
int seqSearchRec(int[] E, int m, int num, int K)
{
    int ans;
    if (m >= num)
        ans = -1;
    else if (E[m] == K)
        ans = m;
    else
        ans = seqSearchRec(E, m+1, num, K);
    return ans;
}
```
Recurrence Equations and Master Theorem

General scheme for time complexity analysis

\[ T(n) = \]
1. For a sequence of blocks, add the individual costs
2. With alternation, take the cost of the most costly branch
3. If recursive procedure called, add \( T(n') \), where \( n' \) is the size at call.

For example, the recursive Sequential Search can be analyzed using this scheme

```c
int seqSearchRec(int[] E, int m, int num, int K)
int ans;
1  if (m >= num)
2    ans = -1;
3  else if (E[m] == K)
4    ans = m;
5  else
6    ans = seqSearchRec(E, m+1, num, K);
7  return ans;
```

Let \( n = \text{num} - m \) as the initial size

\[ T(n) = (0 + (1 + \max(0, T(\text{num}-(m+1)))) + 0 = T(n-1) + 1 \]
Recurrence Equations and Master Theorem

Divide-and-Conquer

The cost of solving a problem of size $n$ is shared by the cost of $b$ subproblems of size $n/c$, plus non-recursive overhead cost $f(n)$:

$$T(n) = b \cdot T\left(\frac{n}{c}\right) + f(n)$$

This is a recurrence equation.

How to evaluate the cost $T(n)$?

An example

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$
Observations
1. \( T(n) \) is the sum of the nonrecursive costs of all nodes in the tree, which is the sum of the row-sums;
2. Depth of the tree is \( D = \frac{\lg(n)}{\lg(c)} \);
3. Number of leaves is approximately \( L = n^E \) where \( E = \frac{\lg(b)}{\lg(c)} \);
4. If the row-sums remain about constant at all depth, then \( T(n) \in \Theta(f(n) \log(n)) \);
5. If the row-sums grow fast, the cost at the leaves would dominate, therefore \( T(n) \in \Theta(n^E) \);
6. If the row-sums decrease fast, the cost at the root would dominate, therefore \( T(n) \in \Theta(f(n)) \);
7. And more formally,

Theorem 3.17 (Master Theorem) The solution to the recurrence equation
\[ T(n) = b \frac{T(n/c)}{c} + f(n), \]
where \( b \geq 1 \) and \( c > 1 \), has the following forms:
1. If \( f(n) \in O(n^{E-\varepsilon}) \) for \( \varepsilon > 0 \), then \( T(n) \in \Theta(n^E) \) where \( E = \frac{\lg(b)}{\lg(c)} \) is called critical exponent.
2. If \( f(n) \in O(n^E) \), then \( T(n) \in \Theta(f(n) \log(n)) \).
3. If \( f(n) \in \Omega(n^{E+\delta}) \) for \( \delta \geq \varepsilon \), and \( f(n) \in O(n^{E+\delta}) \) for \( \delta \geq \varepsilon \), then \( T(n) \in \Theta(f(n)) \).
Example 1

\[ T(n) = 7T(n/2) + n^2 \]

1. Recognize \( b, c, \) and \( f(n) \):
   \[ b = 7, \ c = 2, \ \text{and} \ f(n) = n^2 \]
2. Compute \( E = \log(b)/\log(c) = \log(7) \)
3. Compare \( f(n) \) and \( n^E \) asymptotically
   \[ f(n) = n^{\log(7)/\log(2)} = n^{\log_2(7)} \]
   \[ = n^{0.8} = O(n^{0.8}) \]
4. Apply appropriate case of Master Theorem
   case 1 applies: \( T(n) = \Theta(n^{\log_2(7)}) \)

Example 2

\[ T(n) = 4T(n/2) + n^2 \log(n) \]

1. Recognize \( b, c, \) and \( f(n) \):
   \[ b = 4, \ c = 2, \ \text{and} \ f(n) = n^2 \log(n) \]
2. Compute \( E = \log(b)/\log(c) = \log(4)/\log(2) = 2 \)
3. Compare \( f(n) \) and \( n^E \) asymptotically
   \[ f(n)/n^E = n^2 \log(n)/n^2 = \log(n) \]
4. Determine appropriate case of Master Theorem and apply
   case 1: \( f(n)/n^E = \log(n) = \Theta(n^{\epsilon}) \) for some \( \epsilon > 0 \) \ NO
   case 2: \( f(n)/n^E = \log(n) = \Theta(1) \) \ NO
   case 3: \( f(n)/n^E = \log(n) = \Omega(n^{\epsilon}) \) for some \( \epsilon > 0 \) \ NO

Note: \( \log(x) \) is faster than \( \Theta(1) \) but slower than \( x^\epsilon \) for any \( \epsilon > 0 \) (Exercise).

Lesson: There are gaps between cases in Master Theorem, therefore Master Theorem does not cover all recurrence equations of that form.
**Example 3**

\[ T(n) = T(n/4) + T(n/2) + n^2 \]

Exercise: \( T(n) = n^2(1 + 5/16 + (5/16)^2 + \ldots) = (16/11) n^2 = \Theta(n^2) \)

---

**Induction Proofs**

A mechanic procedure with mainly 3 steps

Step 1: prove base case(s), e.g., \( n=0 \).

Step 2: assume the goal is true for arbitrary \( n \), say \( n=k \).

Step 3: then prove it is also true for \( n=k+1 \).
Example: $\sum_{i=1}^{n} i = n(n+1)/2$

Base case $n = 1$
LHS = 1 and RHS = $1(1+1)/2 = 1$
Note: we can do this manually for $n = 2, 3, \ldots$

Let's assume it holds for arbitrary $n \geq 1$, we now prove it also holds for $n+1$.

$LHS(n+1) = \sum_{i=1}^{n+1} i = \left( \sum_{i=1}^{n} i \right) + (n+1)$

\[
= n(n+1)/2 + (n+1) \\
= [n(n+1) + 2(n+1)]/2 \\
= (n+1)[n+2]/2 \\
= RHS(n+1)
\]

Since we have proved manually it is true when $n=1$. Now we know if it is true for $n=1$ it must be true for $n=2$, and if it is true for $n=2$ it must be true for $n=3$, and on and on.

Note: such a procedure is like to unravel a recursive call in a reversed order, i.e., from base case to more general cases.

---

A recursive implementation to compute the sum of arithmetic series

```c
int sum_one_to_n(int n) {
    int sum;
    if (n == 1) 
        sum = 1;
    else if (n > 1)
        sum = n + sum_one_to_n(n-1); // this recurrence is called on a smaller input.
    return sum;
}
```