Recursive Algorithms

- An algorithm/program/function/procedure/method is ***recursive*** if it calls *itself* as a subroutine.

- If the problem is small/simple enough, just solve it directly.

  Otherwise, ***divide and conquer***:

  - Reduce the problem to one or more *smaller/simpler* subproblems.
  - Solve each subproblem recursively.
  - Combine the subsolutions into the final solution.

  Just like induction!

- Proof of correctness by induction:

  - Base case(s) in proof = direct parts of algorithm
  - Inductive case(s) in proof = recursive parts of algorithm

- Running time computed by setting up and solving a ***recurrence***:

  - Base case(s) of recurrence = direct parts of algorithm
  - Recursive case(s) of recurrence = recursive parts of algorithm

Analyzing Algorithms

- Most instructions take $\Theta(1)$ time.

  addition, subtraction, multiplication, division, comparisons, assignments, logical operations, array lookups, pointer traversals, memory allocation[1]

- Loops become sums:

$$\boxed{\begin{array}{l} \text{for } i \leftarrow 1 \text{ to } n \\ \qquad \textbf{\textit{Something taking } } \boldsymbol{f(i)} \textbf{\textit{ time}} \end{array}} \implies \sum_{i=1}^{n} f(i) \;+\; \Theta(n)$$

- Subroutine calls become functions:

$$\boxed{\begin{array}{l} \text{FOO}(n/2) \\ \text{for } i \leftarrow 1 \text{ to } n \\ \qquad \text{BAR}(i) \\ \text{FOO}(n/2) \end{array}} \implies 2 \cdot T_{\text{FOO}}(n/2) + \sum_{i=1}^{n} T_{\text{BAR}}(i) + \Theta(n)$$

- Recursive calls become recurrences:

$$\boxed{\begin{array}{l} \underline{\text{SQUEE}(n)\text{:}} \\ \quad \text{for } i \leftarrow 1 \text{ to } n-1 \\ \qquad \text{SQUEE}(i) \end{array}} \implies T_{\text{SQUEE}}(n) = \sum_{i=1}^{n} T_{\text{SQUEE}}(i) + \Theta(n)$$

[1] Dynamic memory management requires some nontrivial work behind the scenes, but in practice, we can *usually* pretend it's free. However, dealing with multilevel caches and virtual memory is a *lot* more complicated.

Binary Search

Suppose $A[1\,..\,n]$ is a *sorted* array of numbers, and x is another number.

```
BINARYSEARCH(A[lo .. hi], x):
  if lo > hi
      return "none"
  else
      mid ← ⌊(hi + lo)/2⌋
      if x = A[mid]
          return mid
      else if x < A[mid]
          return BINARYSEARCH(A[lo .. mid − 1], x)
      else
          return BINARYSEARCH(A[mid + 1 .. hi], x)
```

[demo]

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47

Non-recursive binary search

```
BINARYSEARCH(A[lo .. hi], x):
  while lo ≤ hi
      mid ← ⌊(hi + lo)/2⌋
      if x = A[mid]
          return mid
      else if x < A[mid]
          hi ← mid − 1
      else
          lo ← mid + 1
  return "none"
```

Here we transformed ***tail recursion*** into a simple loop.

Whenever possible, ***think*** recursively, but ***code*** non-recursively.

Merge Sort

To sort an array:
  If the array is short enough, there's nothing to do. Relax.
  Otherwise:
    Recursively sort the left half of the array.
    Recursively sort the right half of the array.
    Merge the two sorted halves.

[demo]

Merging two sorted lists

Main idea: Move the smallest element into the output list and repeat.

```
MERGE(A[1 .. n], B[1 .. m]):
  i ← 1
  j ← 1
  for k ← 1 to n + m
      if i > n
          C[k] ← B[j];  j ← j + 1
      else if j > m
          C[k] ← A[i];  i ← i + 1
      else if A[i] < B[j]
          C[k] ← B[j];  j ← j + 1
      else
          C[k] ← A[i];  i ← i + 1

  return C[1 .. n + m]
```

- ***Correctness:*** induction on the number of loop iterations

  After the kth iteration of the loop, the k smallest elements of $A \cup B$ are stored in $C[1..k]$ in increasing order.

- ***Running time:*** $\Theta(1)$ per iteration, so $\Theta(m+n)$ overall.

```
MergeSort(A[1 .. n]):
  if n ≥ 2
      m ← ⌈n/2⌉
      MergeSort(A[1 .. m])
      MergeSort(A[m + 1 .. n])
      B[1 .. n] ← Merge(A[1 .. m], A[m + 1 .. n])
      A[1 .. n] ← B[1 .. n]
  return A[1 .. n]
```

Theorem: MergeSort *correctly sorts any array.*

Proof (induction): Let n be an arbitrary integer.

Let $A[1\,..\,n]$ be an arbitrary array.

Assume that MergeSort sorts any array of size less than n.

Either $n \leq 1$ or $n \geq 2$.

- If $n \leq 1$, then the input array is already sorted, and the algorithm correctly does nothing.
- Suppose $n \geq 2$.

  By the inductive hypothesis, MergeSort$(A[1\,..\,m])$ correctly sorts $A[1\,..\,m]$, since $m < n$.

  By the inductive hypothesis, MergeSort$(A[m+1\,..\,n])$ correctly sorts $A[m+1\,..\,n]$, since $n - m < n$.

  Because Merge correctly merges *any* two sorted lists, the final output is a sorted list containing every element of $A[1\,..\,n]$.

In both cases, we conclude that MergeSort correctly sorts $A[1\,..\,n]$. $\square$

```
MergeSort(A[1 .. n]):
    if n ≥ 2
        m ← ⌈n/2⌉
        MergeSort(A[1 .. m])
        MergeSort(A[m + 1 .. n])
        B[1 .. n] ← Merge(A[1 .. m], A[m + 1 .. n])
        A[1 .. n] ← B[1 .. n]
    return A[1 .. n]
```

Let $T(n)$ be the time to MergeSort an array of length n.

$$\boxed{T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{otherwise} \end{cases}}$$

Theorem: $\boldsymbol{T(n) = \Theta(n \log n)}$

Simplifying assumptions:

1. The input size n is always a power of 2.
2. The $\Theta(n)$ term is really just cn for some constant c.

$$\boxed{T(n) = \begin{cases} 0 & \text{if } n \le 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}}$$

Theorem: $\boldsymbol{T(n) = cn \log_2 n}$ *whenever n is a power of two.*

Proof: Let n be an arbitrary power of two.

Assume that $T(k) = ck \log_2 k$ for any power of two $k < n$.

Either $n = 1$ or $n \ge 2$.

- If $n = 1$, then $T(1) = 0$ and $cn \log_2 n = c \cdot 1 \cdot \log_2 1 = 0$.
- Suppose $n \ge 2$.

$$\begin{aligned} T(n) &= 2T(n/2) + cn && \text{[recurrence]} \\ &= 2 \cdot c(n/2) \log_2(n/2) + cn && \text{[ind. hyp.]} \\ &= cn \log_2(n/2) + cn && \text{[algebra]} \\ &= cn(\log_2 n - 1) + cn && \text{[algebra]} \\ &= cn \log_2 n && \text{[algebra]} \end{aligned}$$

In both cases, we conclude that $T(n) = cn \log_2 n$. $\square$

Recursion Trees

How to solve recurrences of the form

$$\boxed{T(n) = a \cdot T(n/b) + f(n)}$$

Draw a ***rooted tree***, where every node has a ***children***.
The ***root*** stores the value $f(n)$.
Each ***subtree*** is a recursion tree for $T(n/b)$.
Thus, for every d, every node at ***depth*** d stores the value $f(n/b^d)$.

$T(n)$ is the sum of all the values stored in the tree.

The Punchline

$$T(n) = a \cdot T(n/b) + f(n)$$

$$\Downarrow$$

$$\boxed{T(n) = \sum_{d=0}^{\log_b n} a^d \cdot f(n/b^d)}$$

Useful special cases (aka "The Master Theorem"):

- $a \cdot f(n/b) = f(n)$:
  Every term in the sum is equal!

  $$\boxed{T(n) = \Theta(f(n) \log n)}$$

- $a \cdot f(n/b) < c \cdot f(n)$ for some $c < 1$:
  It's a descending geometric series; only the largest term matters!

  $$\boxed{T(n) = \Theta(f(n))}$$

- $a \cdot f(n/b) > c \cdot f(n)$ for some $c > 1$:
  It's an ascending geometric series; only the largest term matters!

  $$\boxed{T(n) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})}$$

- Anything else: You're on your own.

Recursion Trees, Take 2

How to solve recurrences of the form

$$\boxed{T(n) = a \cdot T(n-b) + f(n)}$$

Draw a rooted tree, where every node has a children.
The root stores the value $f(n)$.
Each subtree is a recursion tree for $T(n-b)$.
$T(n)$ is the sum of all the values stored in the tree.

The Other Punchline

$$T(n) = a \cdot T(n-b) + f(n)$$

$$\Downarrow$$

$$\boxed{T(n) = \sum_{d=0}^{n/b} a^d \cdot f(n - db)}$$

Useful special cases (aka "The Slave Theorem"):

- $a \cdot f(n-b) = f(n)$:
  Every term in the sum is equal!

  $$\boxed{T(n) = \Theta(f(n) \cdot n)}$$

- $a \cdot f(n-b) < c \cdot f(n)$ for some $c < 1$:
  It's a descending geometric series; only the largest term matters!

  $$\boxed{T(n) = \Theta(f(n))}$$

- $a \cdot f(n-b) > c \cdot f(n)$ for some $c > 1$:
  It's an ascending geometric series; only the largest term matters!

  $$\boxed{T(n) = \Theta(a^{n/b})}$$

- Anything else: You're on your own.