Recursion

Lecture 10
We will learn

1. How to ask the recursion fairy to solve the problem for us.
We will learn

1. How to ask the recursion fairy to solve the problem for us.
2. How to analyze the running time of a recursive algorithm.
We will learn

1. How to ask the recursion fairy to solve the problem for us.
2. How to analyze the running time of a recursive algorithm.

3. Recursion in action
   1. Tower of Hanoi puzzle
   2. Merge sort
   3. Quick sort
Recursion

How to think about it

Recursion = Induction
Part I

Tower of Hanoi
Tower of Hanoi

The Tower of Hanoi puzzle

Move stack of \( n \) disks from peg 0 to peg 2, one disk at a time.

**Rule:** cannot put a larger disk on a smaller disk.

**Question:** what is a strategy and how many moves does it take?
TOWER OF HANOI VIA RECURSION

The Tower of Hanoi algorithm; ignore everything but the bottom disk.
Recursive Algorithm

\[
\text{Hanoi}(n, \text{src}, \text{dest}, \text{tmp}): \\
\quad \text{Hanoi}(n - 1, \text{src}, \text{tmp}, \text{dest}) \\
\quad \text{Move disk } n \text{ from src to dest} \\
\quad \text{Hanoi}(n - 1, \text{tmp}, \text{dest}, \text{src})
\]
Recursive Algorithm

\[\text{Hanoi}(n, \text{src}, \text{dest}, \text{tmp}):\]

\[
\text{if } (n > 0) \text{ then}
\]

\[\text{Hanoi}(n - 1, \text{src}, \text{tmp}, \text{dest})\]

Move disk \( n \) from src to dest

\[\text{Hanoi}(n - 1, \text{tmp}, \text{dest}, \text{src})\]
Recursive Algorithm

\[
\text{Hanoi}(n, \text{src}, \text{dest}, \text{tmp}) : \\
\text{if } (n > 0) \text{ then} \\
\quad \text{Hanoi}(n - 1, \text{src}, \text{tmp}, \text{dest}) \\
\quad \text{Move disk } n \text{ from src to dest} \\
\quad \text{Hanoi}(n - 1, \text{tmp}, \text{dest}, \text{src})
\]

Proof of correctness.
Recursive Algorithm

\[
\text{Hanoi}(n, \text{src}, \text{dest}, \text{tmp}): \\
\quad \text{if } (n > 0) \text{ then} \\
\quad \quad \text{Hanoi}(n - 1, \text{src}, \text{tmp}, \text{dest}) \\
\quad \quad \text{Move disk } n \text{ from src to dest} \\
\quad \quad \text{Hanoi}(n - 1, \text{tmp}, \text{dest}, \text{src})
\]

Running time analysis.

\[ T(n) \]: time to move \( n \) disks via recursive strategy
Recursive Algorithm

Hanoi($n$, src, dest, tmp):
  if ($n > 0$) then
    Hanoi($n - 1$, src, tmp, dest)
    Move disk $n$ from src to dest
    Hanoi($n - 1$, tmp, dest, src)

Running time analysis.

$T(n)$: time to move $n$ disks via recursive strategy

$$T(n) = 2T(n - 1) + 1 \quad n > 1 \quad \text{and} \quad T(1) = 1$$

$$T(n) = 2^{n-1} (T(1) + 1) = 2^n$$
\begin{align*}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2 - 1) = 2^n - 1
\end{align*}
Part II

Merge Sort
Input  Given an array of \( n \) elements
Goal  Rearrange them in ascending order
Input: Array $A[1 \ldots n]$
Merge Sort [von Neumann]

1. **Input:** Array $A[1 \ldots n]$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m + 1 \ldots n]$, where $m = \lfloor n/2 \rfloor$
Merge Sort [von Neumann]

```plaintext
1. Input: Array $A[1 \ldots n]$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m + 1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

3. Recursively `MergeSort` $A[1 \ldots m]$ and $A[m + 1 \ldots n]$
```
Merge Sort [von Neumann]

**MergeSort**

1. **Input:** Array $A[1 \ldots n]$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m + 1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

3. Recursively MergeSort $A[1 \ldots m]$ and $A[m + 1 \ldots n]$

4. Merge the sorted arrays

$(UIUC)$ CS/ECE 374 13
March 1, 2021 13 / 24
Merging Sorted Arrays

1. Use a new array $B$ to store the merged array
2. Scan $A[1 \ldots m]$ and $A[m + 1 \ldots n]$ from left-to-right, storing elements in $B$ in order

\[
A \ G \ L \ O \ R \quad H \ I \ M \ S \ T
\]

\[
A
\]
Merging Sorted Arrays

1. Use a new array $B$ to store the merged array.

2. Scan $A[1 \ldots m]$ and $A[m + 1 \ldots n]$ from left-to-right, storing elements in $B$ in order.

$$A \ G \ L \ O \ R \quad H \ I \ M \ S \ T \quad A \ G$$
Merging Sorted Arrays

1. Use a new array $B$ to store the merged array

2. Scan $A[1 \ldots m]$ and $A[m + 1 \ldots n]$ from left-to-right, storing elements in $B$ in order

\[
\begin{array}{cccccccc}
A & G & L & O & R & H & I & M & S & T \\
A & G & H
\end{array}
\]
Merging Sorted Arrays

1. Use a new array $B$ to store the merged array

2. Scan $A[1 \ldots m]$ and $A[m + 1 \ldots n]$ from left-to-right, storing elements in $B$ in order

$$
\begin{align*}
A & \quad G \quad L \quad O \quad R \\
H & \quad I \quad M \quad S \quad T \\
A & \quad G \quad H \quad I 
\end{align*}
$$
Merging Sorted Arrays

1. Use a new array $B$ to store the merged array

2. Scan $A[1 \ldots m]$ and $A[m + 1 \ldots n]$ from left-to-right, storing elements in $B$ in order

$A G L O R \quad H I M S T$

$A G H I L M O R S T$
Formal Code

\begin{aligned}
\textbf{MERGE}(A[1..n], m): \\
i \leftarrow 1; \ j \leftarrow m + 1 \\
\text{for } k \leftarrow 1 \text{ to } n \\
\quad \text{if } j > n \\
\quad \quad B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \\
\quad \text{else if } i > m \\
\quad \quad B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \\
\quad \text{else if } A[i] < A[j] \\
\quad \quad B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \\
\quad \text{else} \\
\quad \quad B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \\
\text{for } k \leftarrow 1 \text{ to } n \\
A[k] \leftarrow B[k]
\end{aligned}

\textbf{MERGE}(A[1..n]): \\
\text{if } n > 1 \\
\quad m \leftarrow \lfloor n/2 \rfloor \\
\quad \text{MERGE}(A[1..m]) \\
\quad \text{MERGE}(A[m+1..n]) \\
\quad \text{MERGE}(A[1..n], m)

\textbf{MERGE}(A[1..m], m): \\
i \leftarrow 1; \ j \leftarrow m + 1 \\
\text{for } k \leftarrow 1 \text{ to } n \\
\quad \text{if } j > n \\
\quad \quad B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \\
\quad \text{else if } i > m \\
\quad \quad B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \\
\quad \text{else if } A[i] < A[j] \\
\quad \quad B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \\
\quad \text{else} \\
\quad \quad B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \\
\text{for } k \leftarrow 1 \text{ to } n \\
A[k] \leftarrow B[k]
Proving Correctness

Obvious way to prove correctness of recursive algorithm:

By induction on \( n \) that MergeSort is correct if you assume Merge is correct.

How do we prove that Merge is correct?

Also by induction!

One way is to rewrite Merge into a recursive version.

For algorithms with loops one comes up with a natural loop invariant that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration \( k \) the following hold:

- \( B[1..k] \) contains the smallest \( k \) elements of \( A \) correctly sorted.
- \( B[1..k] \) contains the elements of \( A[1..(i - 1)] \) and \( A[(m + 1)..(j - 1)] \).
- No element of \( A \) is modified.
Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct?
Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on $n$ that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural loop invariant that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.
Proving Correctness

Obvious way to prove correctness of recursive algorithm: induction!

- Easy to show by induction on \( n \) that MergeSort is correct if you assume Merge is correct.
- How do we prove that Merge is correct? Also by induction!
- One way is to rewrite Merge into a recursive version.
- For algorithms with loops one comes up with a natural *loop invariant* that captures all the essential properties and then we prove the loop invariant by induction on the index of the loop.

At the start of iteration \( k \) the following hold:

- \( B[1..k] \) contains the smallest \( k \) elements of \( A \) correctly sorted.
- \( B[1..k] \) contains the elements of \( A[1..(i - 1)] \) and \( A[(m + 1)..(j - 1)] \).
- No element of \( A \) is modified.
$T(n)$: time for merge sort to sort an $n$ element array
Running Time

\( T(n) \): time for merge sort to sort an \( n \) element array

\[
T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn
\]
**Running Time**

\[ T(n) : \text{time for merge sort to sort an } n \text{ element array} \]

\[ T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \]

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know \( f(n) \) such that \( T(n) = \Theta(f(n)) \).

1. \( T(n) = O(f(n)) \) - upper bound
2. \( T(n) = \Omega(f(n)) \) - lower bound
Solving Recurrences: Some Techniques

1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus
2. Expand the recurrence and spot a pattern and use simple math
3. Recursion tree method — imagine the computation as a tree
4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds
Recursion Trees

\[
\begin{align*}
\text{size} & \quad \text{work} \\
\sqrt{\frac{n}{4}} & \quad \sqrt{\frac{n}{4}} \\
\sqrt{\frac{n}{4}} & \quad \sqrt{\frac{n}{4}} \\
\sqrt{\frac{n}{4}} & \quad \sqrt{\frac{n}{4}} \\
1 & \quad \sqrt{n} \\
2^k &= n \\
k &= \log n \\
O(n \log n) &
\end{align*}
\]
Part III

Quick Sort
Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array.
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.
Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.
Quick Sort [Hoare]

1. Pick a pivot element from array.
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$.
3. Recursively sort the subarrays, and concatenate them.
Quick Sort [Hoare]

1. Pick a pivot element from array

2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$

3. Recursively sort the subarrays, and concatenate them.
Quick Sort: Example

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: \(16\)

\[
\begin{align*}
\text{size} & \quad n \\
\text{work} & \quad n-1 \\
\text{else} & \quad n-2 \\
\end{align*}
\]

\[
\frac{n(n+1)}{2}
\]

\(O(n^2)\)
Let \( k \) be the rank of the chosen pivot. Then,

\[
T(n) = T(k - 1) + T(n - k) + O(n)
\]
1. Let $k$ be the rank of the chosen pivot. Then,
$$T(n) = T(k - 1) + T(n - k) + O(n)$$

2. If $k = \lfloor n/2 \rfloor$ then
$$T(n) = T(\lfloor n/2 \rfloor - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n).$$
Then, $T(n) = O(n \log n)$. 
Let \( k \) be the rank of the chosen pivot. Then,
\[
T(n) = T(k - 1) + T(n - k) + O(n)
\]

If \( k = \lceil n/2 \rceil \) then
\[
T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n).
\]
Then, \( T(n) = O(n \log n) \).

Theoretically, median can be found in linear time.
Let \( k \) be the rank of the chosen pivot. Then,
\[
T(n) = T(k - 1) + T(n - k) + O(n)
\]

If \( k = \left\lfloor n/2 \right\rfloor \) then
\[
T(n) = T(\left\lfloor n/2 \right\rfloor - 1) + T(\left\lfloor n/2 \right\rfloor) + O(n) \leq 2T(n/2) + O(n)
\]
Then,
\[
T(n) = O(n \log n).
\]

Theoretically, median can be found in linear time.

Typically, pivot is the first or last element of array. Then,
\[
T(n) = \max_{1 \leq k \leq n} (T(k - 1) + T(n - k) + O(n))
\]

In the worst case, \( T(n) = T(n - 1) + O(n) \), which means \( T(n) = O(n^2) \). Happens if array is already sorted and pivot is always first element.