Lectures 5-7
Recursion, Divide and Conquer:
Fast exponentiation,
MergeSort

Adam Smith
Recursive algorithms

- Powerful language for expressing algorithms
  - But delicate to reason about

- Usually: we design algorithms so they are easy to understand
  - Recursive calls are to smaller instances

- Example:
  - Insertion Sort
  - Fibonacci sequence (?)
Exercise

• How can we write InsertionSort recursively?

**INSERTION-SORT** \( (A, n) \overset{\text{▷}}{\rightarrow} A[1 \ldots n] \)

What recursive call goes here?

\[
\begin{align*}
key & \leftarrow A[n] \\
i & \leftarrow n - 1 \\
\text{while } i > 0 \text{ and } A[i] > key & \\
\hspace{1em} \text{do } A[i+1] & \leftarrow A[i] \\
\hspace{2em} i & \leftarrow i - 1 \\
A[i+1] & = key
\end{align*}
\]
Another exercise

- Foo \((A, i, j) \rightarrow A[1 \ldots n]\)
  - \(k \leftarrow \text{argmin}(A, i, j)\)
  - Swap \(A[i]\) and \(A[k]\)
  - Foo\((A, i+1, j)\)

- What is the complexity of this algorithm?
  - *Our first recurrence* :
    \[
    T(n) = c \ n + T(n-1)
    \]
  - Solution: \(T(n) = \Theta(n^2)\)

- What does this algorithm do?
Exercise 3: What does this function do?

• Foo(a,n)▷ a is a real number, n is an integer
  1. If n=1 return a
  2. Else
     • If n is even then
       – X = Foo(a,n/2)
       – Return X*X
     • Else
       – X = Foo(a , (n-1)/2)
       – Return a*X*X
The divide-and-conquer design paradigm

1. **Divide** the problem (instance) into subproblems.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** subproblem solutions.

- We’ll see lots of examples
A faster sort: Merge Sort

**Merge-Sort** $A[1 \ldots n]$

1. If $n = 1$, done.
2. Recursively sort $A[1 \ldots \lfloor n/2 \rfloor]$ and $A[\lceil n/2 \rceil+1 \ldots n]$.
3. “Merge” the 2 sorted lists.

*Key subroutine:* **Merge**
A faster sort: Merge Sort

input $A[1..n]$ 

$A[1..dn/2e]$  
$A[dn/2e+1..n]$  

MERGE SORT  
MERGE SORT  

Sorted $A[1..dn/2e]$  
Sorted $A[dn/2e+1..n]$  

MERGE  

output 

S. Raskhodnikova and A. Smith. Based on notes by E. Demaine and C. Leiserson
Merging two sorted arrays

Time = one pass through each array
= $\Theta(n)$ to merge a total of $n$ elements
(linear time).

S. Raskhodnikova and A. Smith. Based on notes by E. Demaine and C. Leiserson
Pseudocode

Initial call: **MERGE-SORT**(\(A, 1, A.length\))

**MERGE-SORT**(\(A, p, r\))

1. If \(p < r\), \(\Rightarrow \) if \(p \geq r\), do nothing
2. \(q := \text{floor}(p+r)/2)\)
3. **Merge-Sort**(\(A, p, q\))
4. **Merge-Sort**(\(A, q+1, r\))
5. **Merge**(\(A,p,q,r\))
6. return
Exercise

• Write pseudocode for Merge
• Prove correctness
• Check that your pseudocode runs in $O(n)$ time
Correctness of Merge-Sort

Input size $n = r - p + 1$ (length of subarray)

Claims:

1. (Base case) When $n=1$, Merge-Sort works correctly

2. (Induction Step) For every $n > 2$, if Merge-Sort correct for inputs of length $< n$, then Merge-Sort is correct for inputs of length $n$

Together these claims imply that Merge-Sort is correct.
Proof of correctness

• Proof of Claim 1:
  – When \( n = 1 \), we have \( p = r \), so Merge-Sort does nothing, which is the right thing for a list of size 1.

• Proof of Claim 2:
  – Assume that the Merge subroutine is correct.
  – Sufficient to prove that each of the recursive calls to Merge-Sort works.
  – By induction hypothesis, sufficient to show that on input length \( n \), the two sub-arrays have length strictly less than \( n \).
Proof of correctness cont’d

• Note that $p \leq q < r$
  – (Why? $p < r \Rightarrow (p+r)/2 > p \Rightarrow q \geq p$.
    also $p < r \Rightarrow (p+r)/2 < r \Rightarrow q < r$ since rounding down will mean that $q \leq r - 1$).

• The lengths of the subarrays are
  – $q-p+1 \leq (r-1)-p+1 = n-1 < n$
  – $r - (q+1) +1 = r-q \leq r-p = n-1 < n$

• So the two subarrays have length strictly less than $n$, and inductive hypothesis applies. QED.
Analyzing Merge Sort

\[ T(n) \]
\[ \Theta(1) \]
\[ 2T(n/2) \]

**Abuse**

**Sloppiness:** Should be \( T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) \), but it turns out not to matter asymptotically.

**MERGE-SORT** \( A[1 \ldots n] \)

1. If \( n = 1 \), done.

2. Recursively sort \( A[1 \ldots \lfloor n/2 \rfloor] \) and \( A[\lceil n/2 \rceil + 1 \ldots n] \).

3. “Merge” the 2 sorted lists
Recurrence for Merge Sort

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1; \\
2T(n/2) + \Theta(n) & \text{if } n > 1. 
\end{cases} \]

- We usually omit stating the base case because our algorithms always run in time \( \Theta(1) \) when \( n \) is a small constant.
- CLRS and next lectures provide several ways to find a good upper bound on \( T(n) \).
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.
Recursion tree

Solve \( T(n) = 2T(n/2) + cn \), where \( c > 0 \) is constant.
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$h = \lg n$

$\Theta(1)$ \hspace{1cm} \#leaves = n \hspace{1cm} \Theta(n)$

Total = $\Theta(n \lg n)$
Merge Sort vs Insertion Sort

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, Merge Sort asymptotically beats Insertion Sort in the worst case.
  - In practice, Merge Sort beats Insertion Sort for $n > 30$ or so…
  - But how does Merge Sort do on nearly sorted inputs?
- Go test it out for yourself!