CS213/CS213M DS&A
Sharat Chandran

www.cse.iitb.ac.in/~sharat/current/213m

Patterns

- **Algorithmic patterns:**
- Recursion
- Divide-and-conquer
- Amortization
- The greedy method
- Prune-and-search
- Brute force
- Dynamic programming

- **Software design patterns:**
- Iterator
- Adapter
- Position
- Composition
- Template method
- Locator
- Factory method

Agenda

- Recursion Analysis and the master theorem
  - How do we get the solution to the time complexity of algorithms like merge sort
  - And also, towers of Hanoi, the bad Fibonacci algorithm (Generating functions)

Divide and Conquer

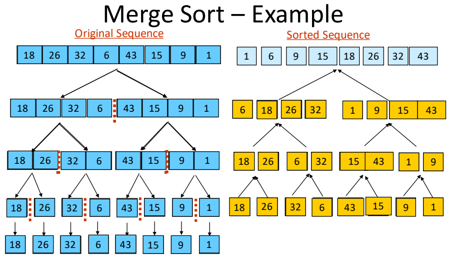
- Recursive in structure
  - *Divide* the problem into sub-problems that are similar to the original but smaller in size
  - *Conquer* the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - *Combine* the solutions to create a solution to the original problem

An Example: Merge Sort

Sorting Problem: Sort a sequence of n elements into non-decreasing order.

- ***Divide***: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

Merge Sort – Example



Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
  if p < r
    then q ← ⌊(p+r)/2⌋
      MergeSort (A, p, q)
      MergeSort (A, q+1, r)
      Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(*A*, 1, *n*)

Procedure Merge

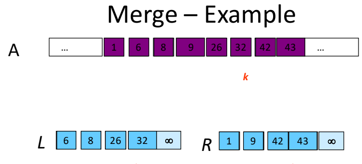
```
Merge(A, p, q, r)
n1 ← q – p + 1
n2 ← r – q
  for i ← 1 to n1
    do L[i] ← A[p + i – 1]
  for j ← 1 to n2
    do R[j] ← A[q + j]
  L[n1+1] ← ∞
  R[n2+1] ← ∞
  i ← 1
  j ← 1
  for k ←p to r
    do if L[i] ≤ R[j]
      then A[k] ← L[i]
           i ← i + 1
      else A[k] ← R[j]
           j ← j + 1
```

Input: Array containing sorted subarrays *A*[*p..q*] and *A*[*q+1..r*].

Output: Merged sorted subarray in *A*[*p..r*].

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge – Example



Analysis of Merge Sort

- Running time $T(n)$ of Merge Sort: finding a closed form solution
  - That is, a solution that has $T(n)$ only on the left-hand side.
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

$\Rightarrow T(n) = \Theta(n \lg n)$

Iterative Method

- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/2^2)) + b(n/2)) + bn \\ &= 2^2T(n/2^2) + 2bn \\ &= 2^3T(n/2^3) + 3bn \\ &= 2^4T(n/2^4) + 4bn \\ &= \ldots \\ &= 2^iT(n/2^i) + ibn \end{aligned}$$

- Note that base, T(n)=b, case occurs when 2^i=n. That is, i = log n.
- So, $$T(n) = bn + bn\log n$$
- Thus, T(n) is O(n log n).

Recurrence Relations

- Equation or an inequality that characterizes a function by its values on smaller inputs.
- Technicality: We ignore floors and ceilings, and boundary conditions
- **Solution Methods**
  - Substitution Method.
  - Recursion-tree Method.
  - Master Method.
- Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms**.
  - **Ex:** Divide and Conquer.

$$T(n) = \Theta(1) \quad \text{if } n \le c$$
$$T(n) = a\,T(n/b) + D(n) + C(n) \quad \text{otherwise}$$

Substitution Method

- **Guess** the form of the solution, then **use mathematical induction** to show it correct.
  - Substitute guessed answer for the function when the inductive hypothesis is applied to smaller values – hence, the name.
- Works well when the solution is easy to guess.
- No general way to guess the correct solution.

Example

Recurrence: $T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + n$ if $n > 1$

- Guess: $T(n) = n \lg n + n$.
- Induction:
  - **Basis:** $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$.
  - **Hypothesis:** $T(k) = k \lg k + k$ for all $k < n$.
  - **Inductive Step:**

$$\begin{aligned} T(n) &= 2\,T(n/2) + n \\ &= 2\,((n/2)\lg(n/2) + (n/2)) + n \\ &= n\,(\lg(n/2)) + 2n \\ &= n \lg n - n + 2n \\ &= n \lg n + n \end{aligned}$$

Example: Substitution Method

- Also called as the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess: T(n) < cn log n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

- Wrong: we cannot make this last line be less than cn log n

Substitution Method

- Consider again the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess #2: $T(n) < cn \log^2 n$.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n \end{aligned}$$

  - if $c > b$.
- So, $T(n)$ is $O(n \log^2 n)$.
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Recursion-tree Method

- Making a good guess is sometimes difficult with the substitution method.
- Use **recursion trees** to devise good guesses.
- Recursion Trees
  - Show successive expansions of recurrences using trees.
  - Keep track of the time spent on the subproblems of a divide and conquer algorithm.
  - Help organize the algebraic bookkeeping necessary to solve a recurrence.

Recursion Tree – Example

- Running time of Merge Sort:

$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

- Rewrite the recurrence as

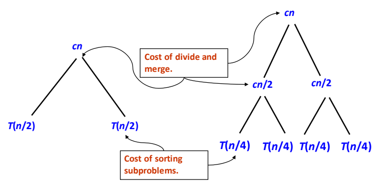
$$T(n) = \boldsymbol{c} \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \boldsymbol{cn} \quad \text{if } n > 1$$

$\boldsymbol{c > 0}$: Running time for the base case and time per array element for the divide and combine steps.

Recursion Tree for Merge Sort

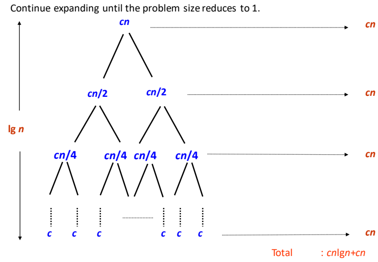
For the original problem, we have a cost of *cn*, plus two subproblems each of size (n/2) and running time $T(n/2)$.

Each of the size n/2 problems has a cost of cn/2 plus two subproblems, each costing $T(n/4)$.



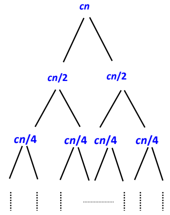
Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



- Each level has total cost ***cn***.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves $\Rightarrow$ *cost per level remains the same*.
- There are lg n + 1 levels, height is lg n. (Assuming n is a power of 2.)
    - Can be proved by induction.
- Total cost = sum of costs at each level = (lg n + 1)cn = cnlg$n$ + cn = $\Theta(n \lg n)$.

Recursion Trees – Caution Note

- Recursion trees only generate guesses.
  - Verify guesses using substitution method.
- A small amount of "sloppiness" can be tolerated.
- If careful when drawing out a recursion tree and summing the costs, can be used as direct proof.

The Master Method

- Based on the Master theorem.
- "Cookbook" approach for solving recurrences of the form

  $T(n) = aT(n/b) + f(n)$
  - $a \geq 1$, $b > 1$ are constants.
  - $f(n)$ is asymptotically positive.
  - n/b may not be an integer, but we ignore floors and ceilings.
- Requires memorization of three cases.

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and
Let $T(n)$ be defined on nonnegative integers by the recurrence
$T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.
$T(n)$ can be bounded asymptotically in three cases:

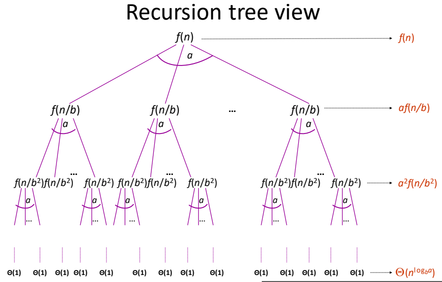
1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if, for some constant $c < 1$ and all sufficiently large n, we have $a \cdot f(n/b) \leq c\, f(n)$, then $T(n) = \Theta(f(n))$.

Master Method – Examples

- $T(n) = 16T(n/4)+n$
  - $a = 16$, $b = 4$, $n^{\log_b a} = n^{\log_4 16} = n^2$.
  - $f(n) = n = O(n^{\log_b a-\varepsilon}) = O(n^{2-\varepsilon})$, where $\varepsilon = 1$ $\Rightarrow$ **Case 1.**
  - Hence, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

- $T(n) = T(3n/7) + 1$
  - $a = 1$, b=7/3, and $n^{\log_b a} = n^{\log_{7/3} 1} = n^0 = 1$
  - $f(n) = 1 = \Theta(n^{\log_b a})$ $\Rightarrow$ **Case 2**.
  - Therefore, $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$

Recursion tree view



$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and
Let $T(n)$ be defined on nonnegative integers by the recurrence
$T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.
$T(n)$ can be bounded asymptotically in three cases:

1. If $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$,
   and if, for some constant $c < 1$ and all sufficiently large n,
   we have $a \cdot f(n/b) \leq c\, f(n)$, then $T(n) = \Theta(f(n))$.

Master Method – Examples

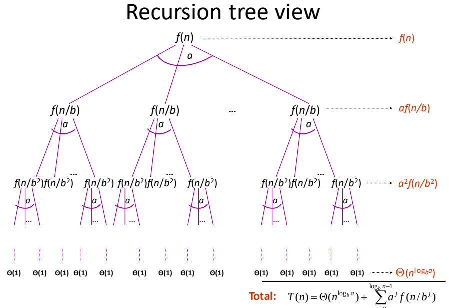
- $T(n) = 3T(n/4) + n \lg n$
  - $a = 3$, $b=4$, thus $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$
  - $f(n) = n \lg n = \Omega(n^{\log_4 3 + \varepsilon})$ where $\varepsilon \approx 0.2 \Rightarrow$ **Case 3**.
  - Therefore, $T(n) = \Theta(f(n)) = \Theta(n \lg n)$.

- $T(n) = 2T(n/2) + n \lg n$
  - $a = 2$, $b=2$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_2 2} = n$
  - ***f(n)*** is asymptotically larger than $n^{\log_b a}$, but not polynomially larger. The ratio $\lg n$ is asymptotically less than n^{ε} for any positive ε. Thus, the Master Theorem ***doesn't*** apply here.

Master Theorem – What it means?

- **Case 1:** If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
  - $n^{\log_b a} = a^{\log_b n}$: Number of leaves in the recursion tree.
  - $f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow$ Sum of the cost of the nodes at each internal level asymptotically smaller than the cost of leaves by a *polynomial* factor.
  - Cost of the problem dominated by leaves, hence cost is $\Theta(n^{\log_b a})$.

Recursion tree view



Master Theorem – What it means?

- **Case 2:** If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
  - $n^{\log_b a} = a^{\log_b n}$: Number of leaves in the recursion tree.
  - $f(n) = \Theta(n^{\log_b a}) \Rightarrow$ Sum of the cost of the nodes at each level asymptotically the same as the cost of leaves.
  - There are $\Theta(\lg n)$ levels.
  - Hence, total cost is $\Theta(n^{\log_b a} \lg n)$.

Master Theorem – What it means?

- **Case 3:** If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$, and if, for some constant $c < 1$ and all sufficiently large n, we have $a \cdot f(n/b) \leq c\, f(n)$, then $T(n) = \Theta(f(n))$.
  - $n^{\log_b a} = a^{\log_b n}$: Number of leaves in the recursion tree.
  - $f(n) = \Omega(n^{\log_b a+\varepsilon}) \Rightarrow$ Cost is dominated by the root. Cost of the root is asymptotically larger than the sum of the cost of the leaves by a polynomial factor.
  - Hence, cost is $\Theta(f(n))$.