CSE 5311

DESIGN AND ANALYSIS OF ALGORITHMS

By:
Arjun Dasgupta
Lokhendra singh

Definitions of Algorithm

- Any well defined **computational procedure** that takes some value or set of values as ***input*** and produces some value or set of values as ***output*** (Introduction to Algorithms)
- A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end (Webster's Dictionary)

UTA

An Example of Algorithm

- Sorting Problem
- ***Input***: A sequence of n numbers $\{a_1, a_2, a_3, \ldots, a_n\}$
- ***Output***: The sequence sorted $\{a'_1, a'_2, a'_3, \ldots, a'_n\}$ such that $a'_1 \leq a'_2 \leq a'_3, \ldots, \leq a'_n$
- We will see several ways to solve the sorting problem. Each way will be expressed as an algorithm
- Two sorting algorithms
  - Insertion_Sort
  - Merge_Sort

Algorithm Efficiency

- Computer A: 10 B instructions/second
- Runs Insertion_Sort of time $2n^2$
- Where, n = 10 million

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)}$$

Efficiency (II)

- Computer B: 10M instructions/second
- Runs Merge_Sort of time $50 n \log_2 n$
- Where, n = 10 million

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)} .$$

- ~17 times faster than Computer A

Analysis of Algorithms Involves evaluating the following parameters

- Memory – Unit generalized as "words"
- Computer time – Unit generalized as "cycles"
- Correctness – Producing the desired output

Sample Algorithm

FINDING LARGEST NUMBER

INPUT: unsorted array ‘A[n]’of n numbers

OUTPUT: largest number

```
large ← A[1]
  for j ← 2 to length[A]
      if large < A[j]
          large ← A[j]
```

<u>Space and Time Analysis</u>

(Largest Number Scan Algorithm)

<u>SPACE</u> S(n): One “word” is required to run the algorithm (step 1…to store variable ‘large’)

<u>TIME</u> T(n): n-1 comparisons are required to find the largest (every comparison takes one cycle)

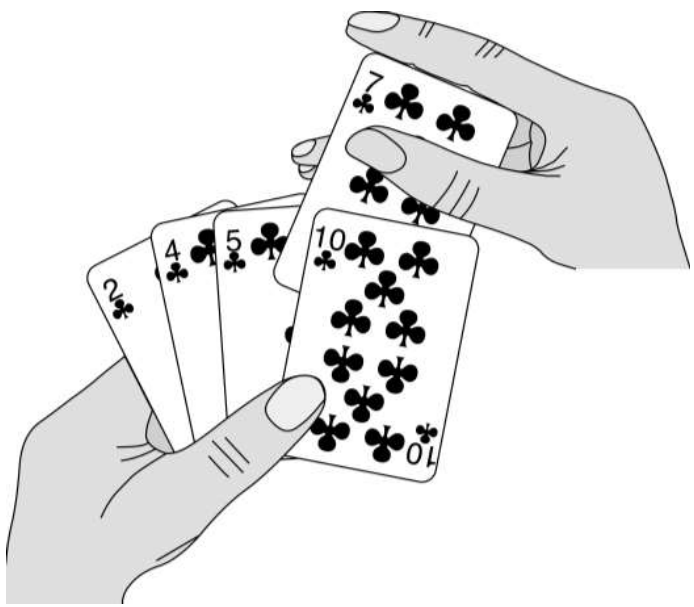
**Aim is to reduce both T(n) and S(n)*

Pseudocode

- Similar to C, C++, Pascal, and Java
- Designed for expressing algorithms
- Software engineering issues of data abstraction, modularity, and error handling are often ignored
- We sometimes embed English statements into pseudocode. Therefore, cannot be compiled

Insertion_Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

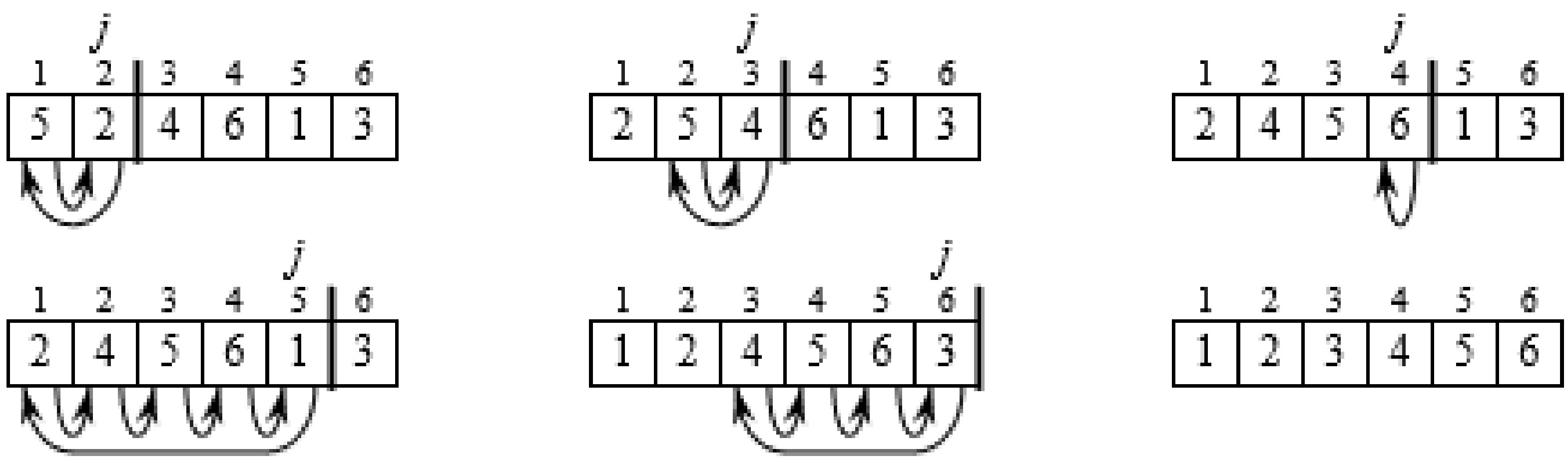


Let t_j be the number of times that the while loop test is executed

Insertion_Sort

INSERTION-SORT(A, n)

	cost	times
for $j = 2$ **to** n	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\quad\quad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$



Insertion_Sort Analysis

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$\quad i = j-1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$\qquad i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

Let $T(n)$ = running time of INSERTION-SORT.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1) .$$

Insertion_Sort, Analysis (II)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Best case: Array is already sorted

- Thus $t_j = 1$ for $j = 2, 3, \ldots, n$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

So $T(n)$ can be expressed as $a(n) + b$

In other words, $T(n)$ can be expresses as a linear function of n

Insertion_Sort, Analysis (III)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Worst case: Array is in reverse sorted order

- Thus $t_j = j$ for $j = 2, 3, \ldots, n$ (while loop runs f times)

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j \text{ and } \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1).$$

- $\sum_{j=1}^{n} j$ is an arithmetic series and equals $\frac{n(n+1)}{2}$.
- Letting $k = j - 1$, we see that $\sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.

Insertion_Sort, Analysis (IV)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Worst case: Array is in reverse sorted order

- Running time thus becomes

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- So $T(n)$ can be expressed as $an^2 + bn + c$
- In other words, $T(n)$ can be expressed as a quadratic function of n

Order of Growth

- An abstraction to ease analysis and focus on the important features
- Look only at the leading term of the formula for running time
  - Drop lower-order terms
  - Ignore the constant coefficient in the leading term

Order of Growth (Example)

- For insertion sort, the worst-case running time is $an^2 + bn + c$.
  - Drop lower-order terms → an^2
  - Ignore constant coefficient → n^2
- We cannot say that the worst-case running time equals n^2
- But, it grows like n^2
- We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2

Asymptotic notation, *tight bound*

ASYMPTOTES

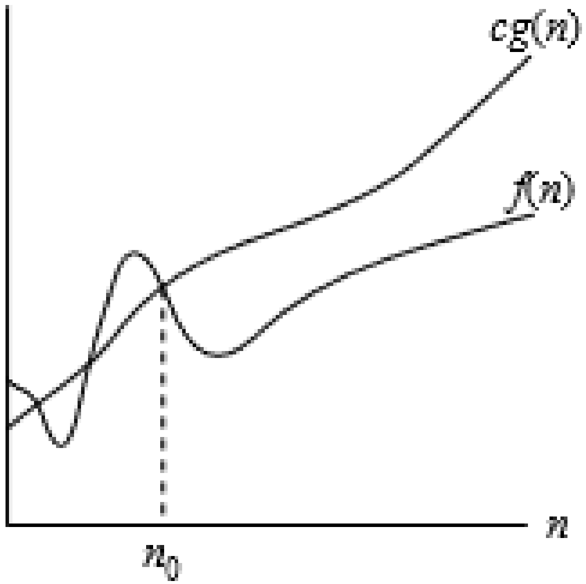
Used to formalize that an algorithm has running time or storage requirements that are ``never more than,'' ``always greater than,'' or ``exactly'' some amount

ASYMPTOTIC NOTATIONS

O-notation (Big Oh)

- *Asymptotic Upper Bound*

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .$$



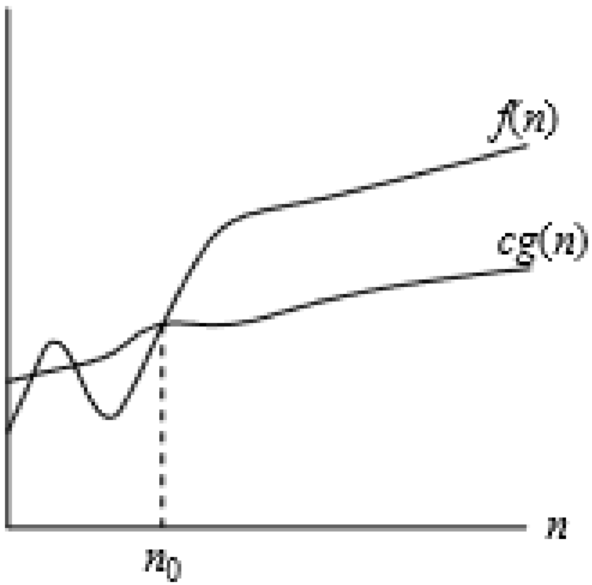
$g(n)$ is an *asymptotic upper bound* for $f(n)$.

ASYMPTOTIC NOTATIONS

Ω-notation (Big omega)

- *Asymptotic lower Bound*

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



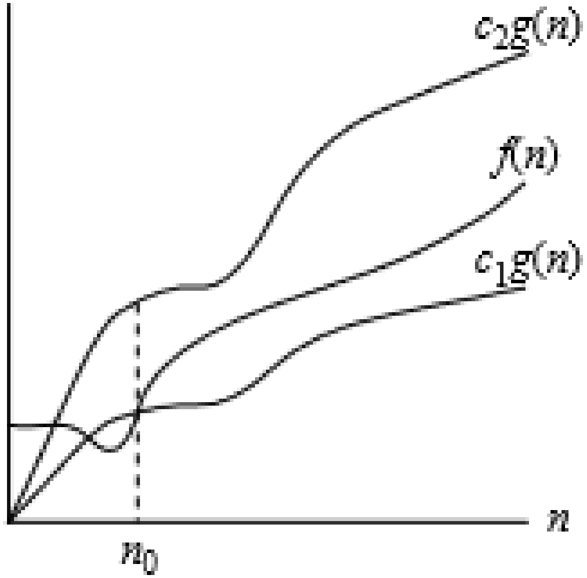
$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

ASYMPTOTIC NOTATIONS

Θ-notation (Theta)

- *Asymptotic tight Bound*

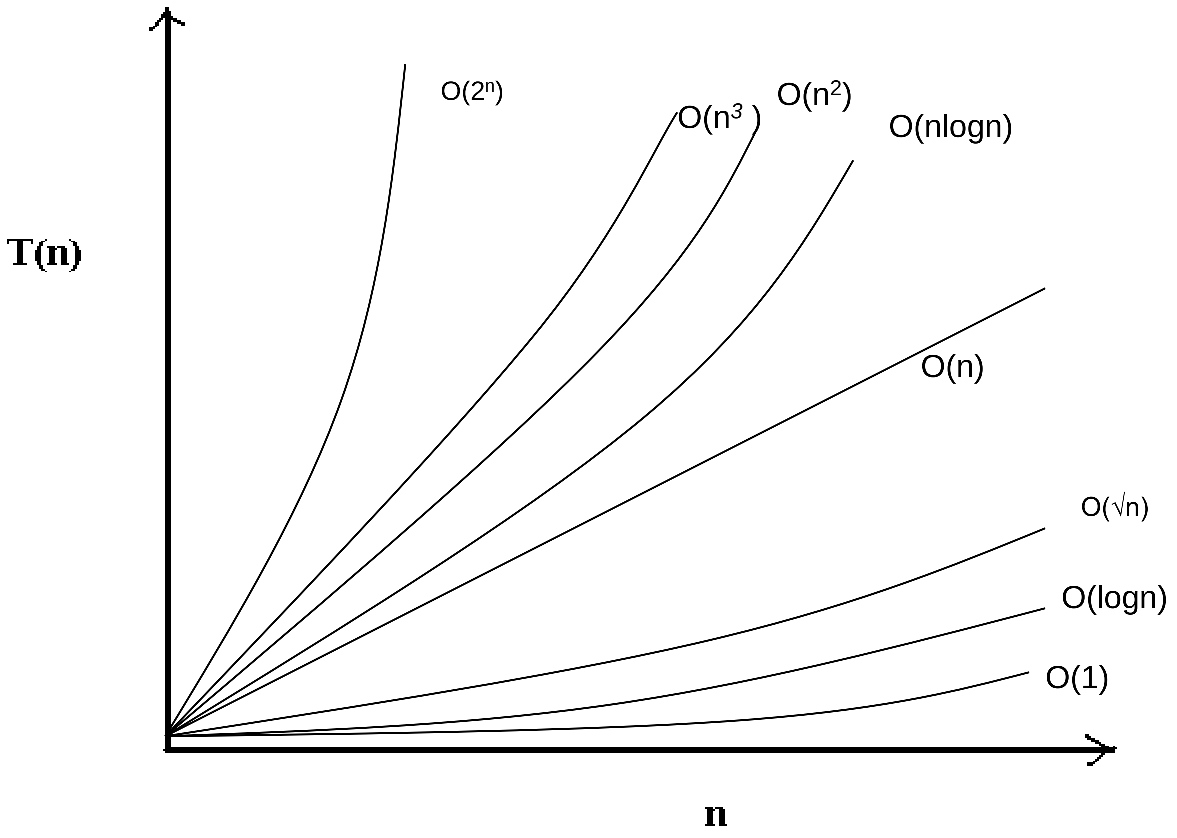
$$\Theta(g(n)) = \{ f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

- O**-notation ----------------** Less than equal to (**“**≤**”**)
- Θ**-notation ----------------** Equal to (“=“)
- Ω**-notation ----------------** Greater than equal to (**“**≥**”**)

Common plots of *O*( )



Examples of algorithms for sorting techniques and their complexities

- Insertion sort : $O(n^2)$
- Selection sort : $O(n^2)$
- Bubble sort: $O(n^2)$
- Quick sort : $O(n \lg n)$
- Merge sort : $O(n \lg n)$

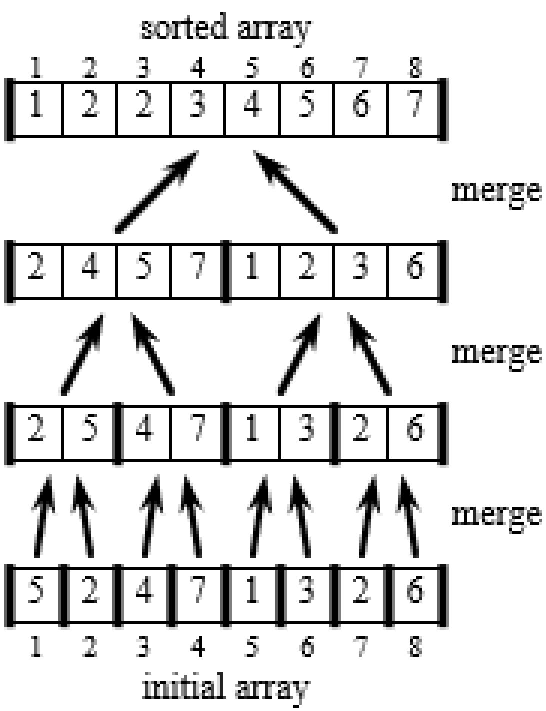
Now, Merge Sort

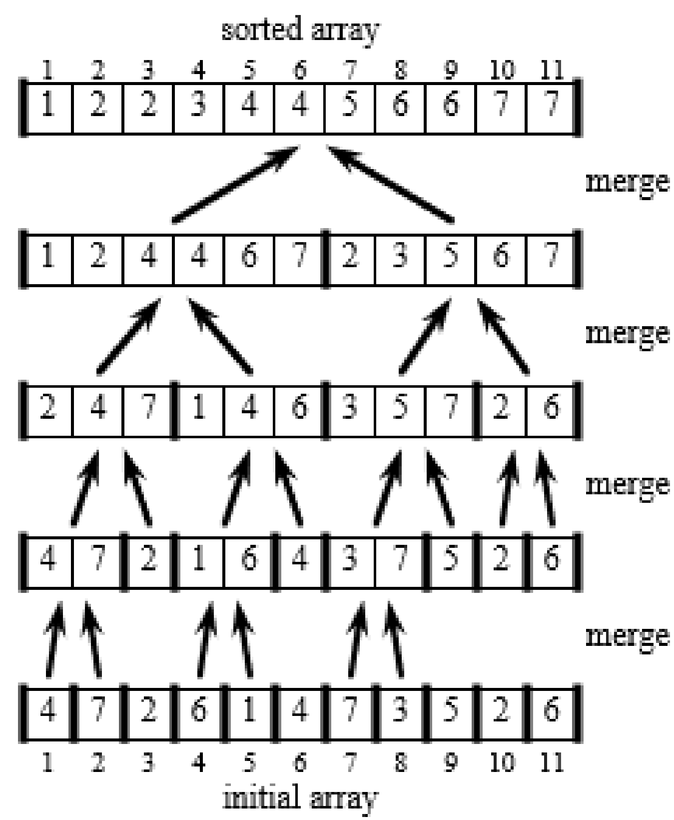
- Insertion Sort is an incremental algorithm
- Divide and conquer is another common approach, which is used for Merge Sort
- **Divide** by splitting into two subarray $A[p \ldots q]$•and $A[q\text{-}1 \ldots r]$• where q is the halfway point of $A[p \ldots r]$•
- **Conquer** by recursively sorting the two subarrays
- **Combine** by merging the two sorted subarrays

Merge Sort → Divide & Conquer

MERGE-SORT(A, p, r)

if $p < r$	// check for base case
$q = \lfloor (p + r)/2 \rfloor$	// divide
MERGE-SORT(A, p, q)	// conquer
MERGE-SORT$(A, q + 1, r)$	// conquer
MERGE(A, p, q, r)	// combine





Pseudocode for Merge

```
MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
        L[i] = A[p + i - 1]
  for j = 1 to n2
        R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
        if L[i] ≤ R[j]
              A[k] = L[i]
              i = i + 1
        else A[k] = R[j]
              j = j + 1
```

Analyzing Divide & Conquer

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- A recurrence equation (more commonly, a recurrence) is used to describe the running time of a divide-and-conquer algorithm
- It describes a function in terms of its sub-problems

Analyzing Divide & Conquer

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- If the problem is small enough (say, $n \le c$ a constant), we have a base case of constant time: $\Theta(1)$
- Otherwise, suppose that we divide into a sub-problems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$)
- Time to divide a size- n problem is $D(n)$
- Have a sub-problems to solve, each of size $a = n$; each sub-problem takes $T(n/b)$ time
- $C(n)$ is the time to combine solutions be

Analyzing Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1\,. \end{cases}$$

- Solution to the above recurrence is: $cn \lg n + cn$
- This can be proven via induction
- Master Theorem (Ch 4) can be used to get to the solution
- Or, we can work it in the class

Home Work

Exercises:

- 1.1-1, 1.1-2
- 1.2-1, 1.2-2
- 2.1-2, 2.1-4
- 2.2-1
- 2.3-3, 2.3-4