Review: Algorithm Efficiency

- A first attempt: *an algorithm is efficient if, when implemented, it runs quickly on real input instances*
- What is missing?
  - What does "quickly" really mean?
  - Run where?
  - Implemented how and in what language?
  - How do you compare to other algorithms, particularly when performance is not consistent across instances?
  - How does the performance scale up?
- Need a more concrete definition, one that is platform and language independent, and has predictive value as the problem scales up

1

Algorithm Efficiency…

- To understand the performance of algorithm *A*, it is not enough to run it on one input.
- Need to understand behavior (memory, running time) over *all* possible input instances.

$$\left.\begin{array}{l}\text{– Minimum}\\ \text{– Maximum}\\ \text{– Average}\end{array}\right\}\ \text{over all inputs of size } n$$

- Complexity is usually expressed as a function (e.g., a polynomial) of the input size *n*

Question. What input distribution should be used when computing the average?

2

Worst, Best, and Average

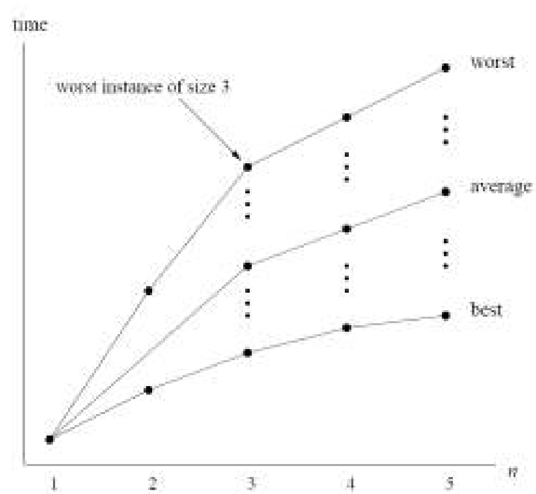
- The *worst case* complexity is the function defined by the *maximum* time taken on any instance of size n.
- The *best case* complexity is the function defined by the *minimum* time taken on any instance of size n.
- The *average-case* complexity is the function defined by an *average* time taken on any instance of size n.

Each of these is a function $N \rightarrow R^{+}$: time vs. size

3

Example: A Sorting Algorithm

- For every instance I, run A and plot point $(|I|, T_A(|I|))$



4

Insertion Sort

Sort(A, n)	**Cost**
for $j \leftarrow 2$ **to** n **do**	$c_1 \cdot n$
$\quad k \leftarrow A[j]$	$c_2 \cdot (n-1)$
$\quad i \leftarrow j-1$	$c_3 \cdot (n-1)$
$\quad$ **while** $i > 0$ **and** $A[i] > k$ **do**	$c_4 \cdot \sum_{j=2}^{n} t_j$
$\qquad A[i+1] \leftarrow A[i]$	$c_5 \cdot \sum_{j=2}^{n} (t_j - 1)$
$\qquad i \leftarrow i-1$	$c_6 \cdot \sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] \leftarrow k$	$c_7 \cdot (n-1)$

$$T(n) = an + b \cdot \sum_{j=2}^{n} t_j + c$$

5

Insertion Sort: Analysis

- Worst, best, and average depend on the values t_j.
  - Best case: $t_j = 1 \ \Rightarrow$
    $T(n) = an + b(n-1) + c = a_1 n + a_0$
  - Worst case: $t_j = j \ \Rightarrow$
    $T(n) = an + b(n+2)(n-1)/2 + c = b_2 n^2 + b_1 n + b_0$
  - Average case: $t_j = j / 2$
    $T(n) = an + b(n+2)(n-1)/4 + c = d_2 n^2 + d_1 n + d_0$

6

Exact Analysis is Difficult

- Best, worst, and average case are difficult to deal with precisely because too many details
- Exact values of constants (a_i, b_i, d_i, ...) depend on:
  – machine
  – compiler
  – implementation of algorithm

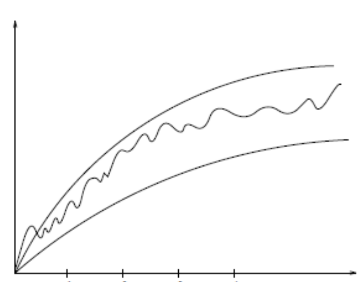
$\Rightarrow$ "Exact" analysis is not very general

- A second attempt: *an algorithm is efficient if it performs significantly fewer operations, at an analytical level and for large inputs, than a naïve or brute force approach*

7

A Simpler Approach

- It is easier to talk about upper and lower bounds of the function in a manner that avoids machine and implementation details
  – Ignore machine dependent constants
  – Drop lower order terms
  – Look at growth rate as $T(n) \rightarrow \infty$

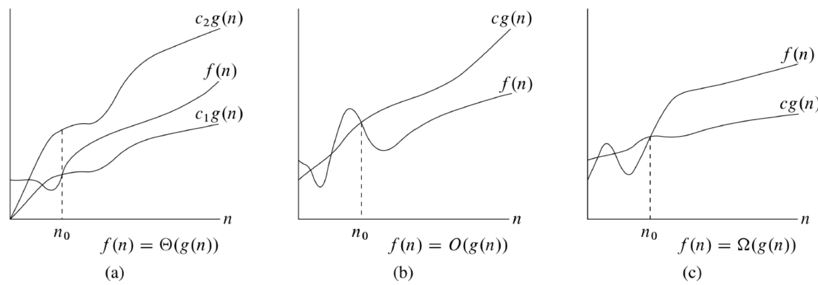


8

Naming the Bounding Functions

- $f(n)=O(g(n))$ means $c \cdot g(n)$ is *upper bound* for $f(n)$
- $f(n)=\Omega(g(n))$ means $c \cdot g(n)$ is *lower bound* for $f(n)$
- $f(n)=\Theta(g(n))$ means $c_1 \cdot g(n)$ is upper bound for $f(n)$ and $c_2 \cdot g(n)$ is lower bound for $f(n)$

c, c_1, and c_2 are constants independent of n, bound holds for "sufficiently large" n



Example: Insertion Sort

$$\text{Time}: \quad T(n) = a_1 n + a_2 \cdot \sum_{j=2}^{n} t_j + a_3$$

$$(a_1 + a_2)n + a_3 - 1 \le T(n) \le a_1 n + a_2 \cdot \frac{(n+2)(n-1)}{2} + a_3$$

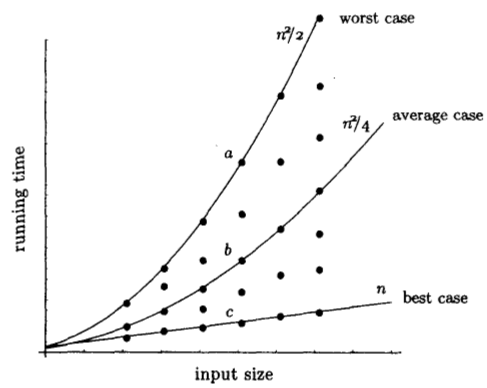
- Worst case grows as $n^2 \Rightarrow T_{\max}(n) = \Theta(n^2)$
- Best case grows as $n \Rightarrow T_{\min}(n) = \Theta(n)$
- Average case grows as $n^2 \Rightarrow T_{\text{ave}}(n) = \Theta(n^2)$

$\Rightarrow$ insertion sort takes "between" $c_1 n$ and $c_2 n^2$

i.e., $T(n) = \Omega(n)$ and $T(n) = O(n^2)$

10

Performance of Insertion Sort

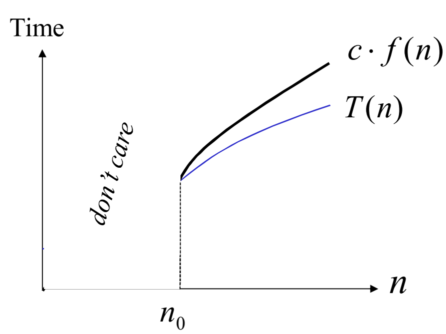


11

Asymptotic Notation: *O*

- Captures the idea of *upper bound* for $T(n)$

$$T(n) = O(f(n)) \Leftrightarrow \exists c, n_0 : T(n) \le c \cdot f(n)\ \forall n \ge n_0$$



12

Examples

- Which of the following is true?

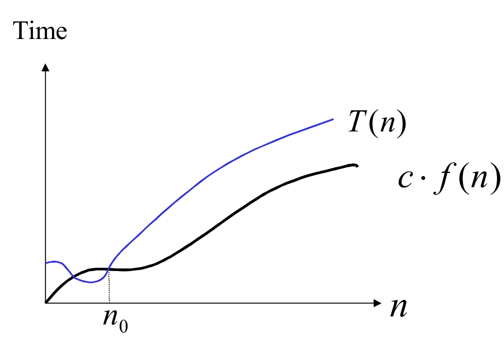
$$100n + 6 = O(n)$$
$$10n^2 + 4n + 2 = O(n^3)$$
$$5n^3 + 4n^2 - 2 = O(n^3)$$
$$\tfrac{1}{100} n^2 = O(n)$$

13

Asymptotic Notation: Ω

- Idea of *lower bound* for $T(n)$

$$T(n) = \Omega(f(n)) \Leftrightarrow \exists c, n_0 : T(n) \geq c \cdot f(n)\ \forall n \geq n_0$$



14

Examples

- Which of the following is true?

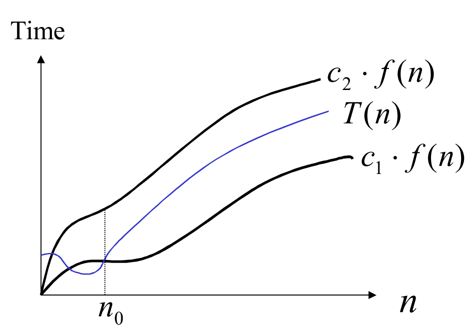
$$0.5n + 6 = \Omega(n)$$
$$2n - 3 = \Omega(n)$$
$$\sqrt{n} = \Omega(\log n)$$
$$n! = \Omega(2^n)$$
$$n^3 \log n - 7n^3 + 2n^2 + 5n\sqrt{n+3} - 8 = \Omega(n^2 \log^5 n)$$

15

Asymptotic Notation: Θ

- Idea of *equivalent bound* for *T*(*n*)

$$T(n) = \Theta(f(n)) \Leftrightarrow$$
$$T(n) = O(f(n)) \;\;\&\;\; T(n) = \Omega(f(n))$$



16

More Examples: O, Ω, Θ

$$3n^2 - 100n + 6 = O(n^2) \text{ because } 3n^2 > 3n^2 - 100n + 6$$
$$3n^2 - 100n + 6 = O(n^3) \text{ because } .01n^3 > 3n^2 - 100n + 6$$
$$3n^2 - 100n + 6 \neq O(n) \text{ because } c \cdot n < 3n^2 \text{ when } n > c$$

$$3n^2 - 100n + 6 = \Omega(n^2) \text{ because } 2.99n^2 < 3n^2 - 100n + 6$$
$$3n^2 - 100n + 6 \neq \Omega(n^3) \text{ because } 3n^2 - 100n + 6 < n^3$$
$$3n^2 - 100n + 6 = \Omega(n) \text{ because } 10^{10^{10}} n < 3n^2 - 100 + 6$$

$$3n^2 - 100n + 6 = \Theta(n^2) \text{ because } O \text{ and } \Omega$$
$$3n^2 - 100n + 6 \neq \Theta(n^3) \text{ because } O \text{ only}$$
$$3n^2 - 100n + 6 \neq \Theta(n) \text{ because } \Omega \text{ only}$$

when n is sufficiently large

- Think of = as meaning *"in the set of functions"*

17

Asymptotic Notation: o

- Idea of *strictly upper bound* for $T(n)$
- $o(f(n))$ refers to the set of functions that grow strictly slower than $f(n)$

$$T(n) = o(f(n)) \Leftrightarrow$$
$$T(n) = O(f(n)) \text{ and } T(n) \neq \Omega(f(n))$$

Examples:

$$7n + 5 = o(n^2)$$
$$n^2 = o(n^2 \log n)$$
$$\log^3 n = o(\sqrt{n})$$

18

Addition and Multiplication of Functions

Suppose $f(n)=O(n^2)$, $f(n)=\Omega(n)$, and $g(n)=\Theta(n^2)$

- What do we know about $f(n) + g(n)$?

  $f(n) + g(n) = O(n^2)$, $f(n) + g(n) = \Omega(n^2)$
- How about $c \cdot f(n)$?

  $c \cdot f(n) = O(n^2)$, $c \cdot f(n)=\Omega(n)$
- How about $f(n) \cdot g(n)$?

  $f(n) \cdot g(n) = O(n^4)$, $f(n) \cdot g(n) = \Omega(n^3)$

19

Algorithm Design Paradigms

Incremental approach

1. Solve $\langle a_1 \rangle$
2. for $i = 2, 3, \ldots, n$

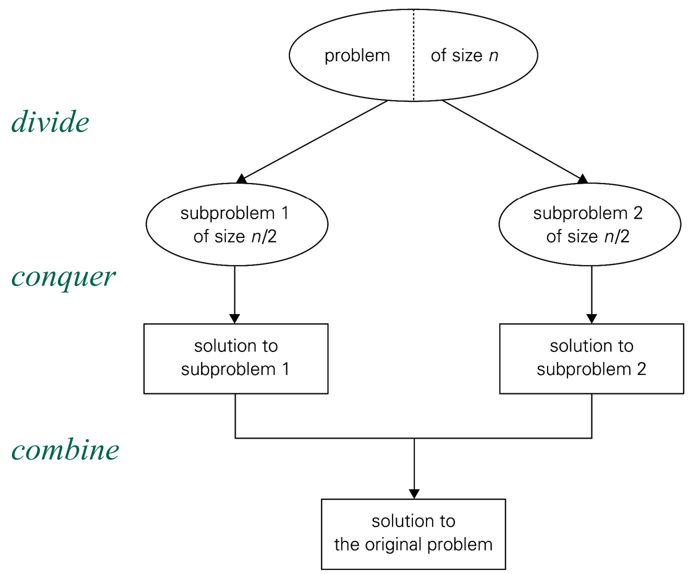
   Solve $\langle a_1,, \ldots, a_{i-1}, a_i \rangle$ using solution of $\langle a_1, , \ldots, a_{i-1} \rangle$

Divide-and-Conquer approach

1. *Divide* by splitting problem into 2 or more smaller sub-problems, e.g., $\langle a_1, \ldots, a_{n/2} \rangle$ and $\langle a_{n/2+1}, \ldots, a_n \rangle$
2. *Conquer* by solving sub-problems independently
3. *Combine* partial results to solve original instance

20

DAC With 2 Sub-problems



21

What paradigm does the following algorithm use?

Sort (A, n)

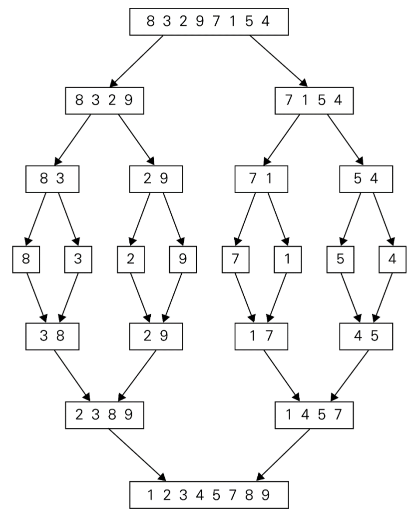
```
for j ← 2 to n do
    k ← A[j]
    i ← j − 1
    while i > 0 and A[i] > k do
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← k
```

- Insertion Sort uses the *incremental approach*

22

A DAC Sort

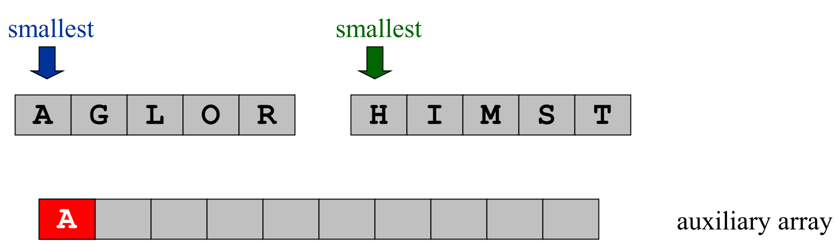
MergeSort(A, p, r)
  if $p = r$ **then return**
  else
    $q \leftarrow (p+r)/2$
    MergeSort(A, p, q)
    MergeSort(A, q+1, r)
    Merge(A,p,q,r)
end



23

Merging

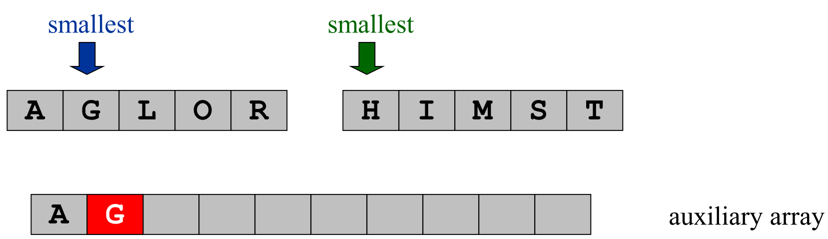
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



24

Merging

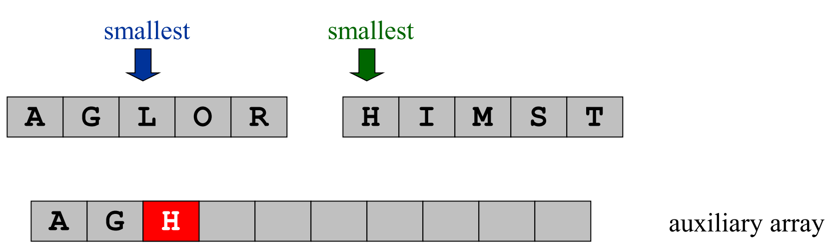
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



25

Merging

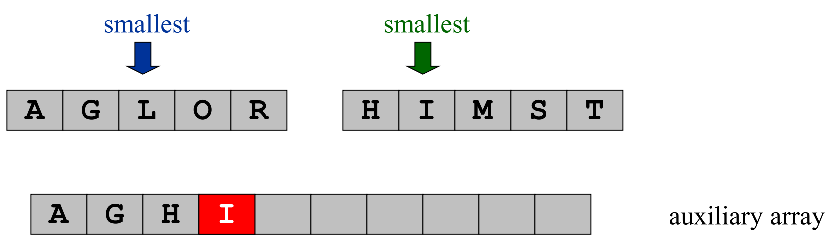
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



26

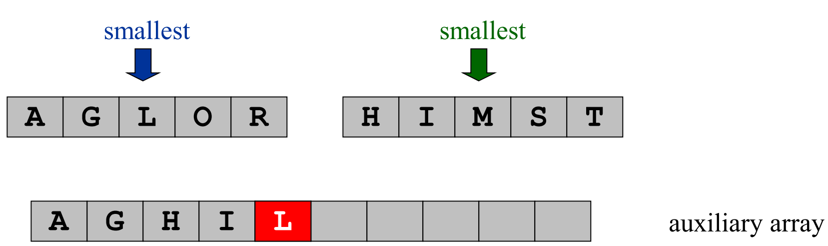
Merging

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



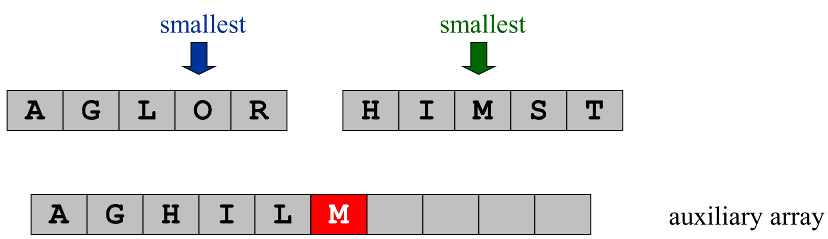
Merging

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

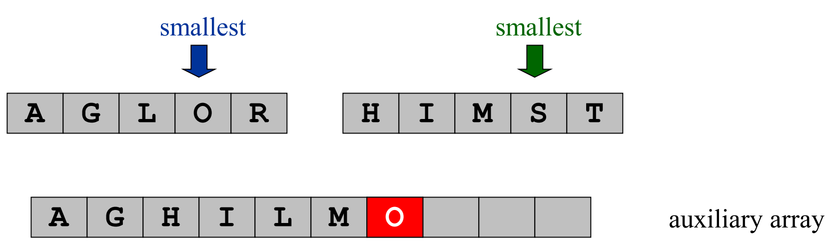
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



29

Merging

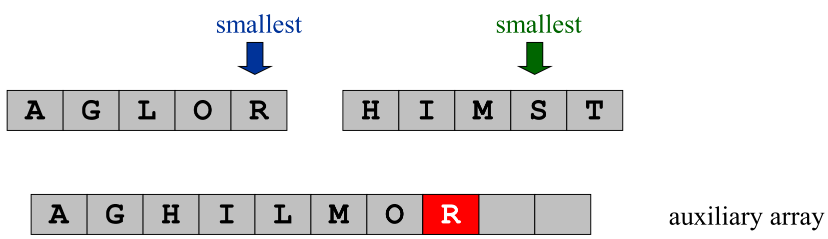
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



30

Merging

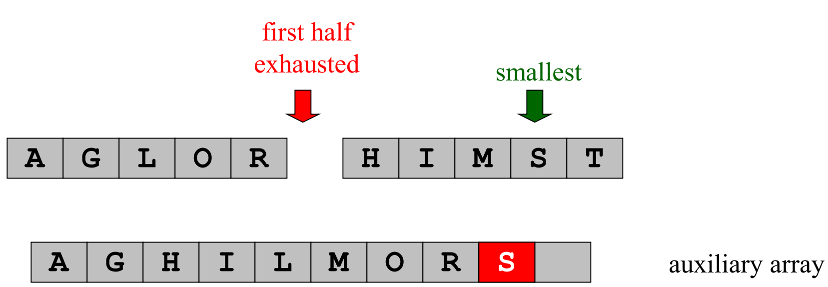
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



31

Merging

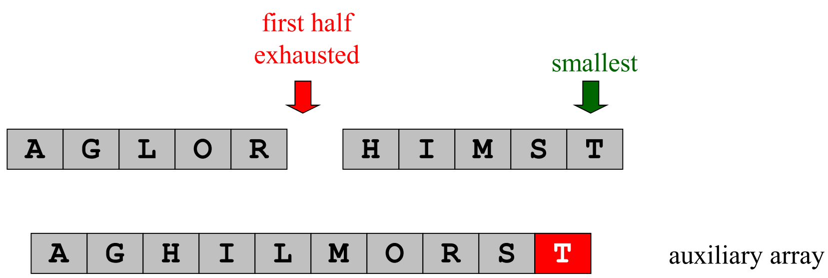
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



32

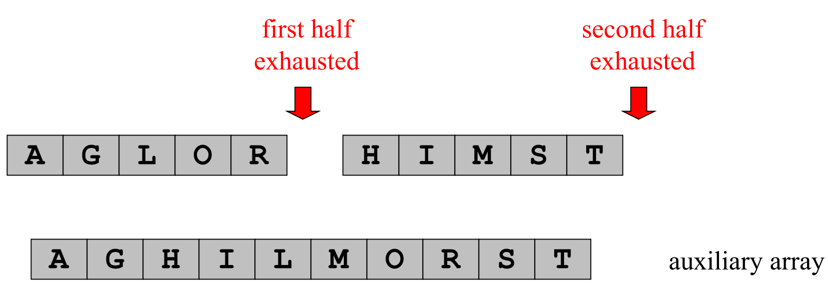
Merging

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Pseudocode

```
MERGE(A, p, q, r)
n₁ ← q − p + 1
n₂ ← r − q
create arrays L[1 .. n₁ + 1] and R[1 .. n₂ + 1]
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
```

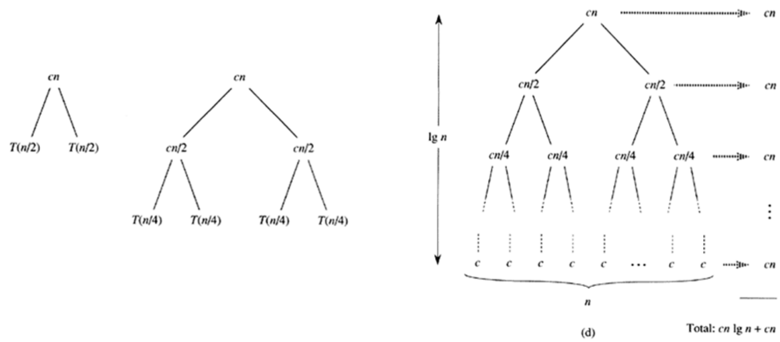

Running Time of Merge Sort

- How long does the basic step (merge) take?
- Time taken can be described by a recurrence:

$$T(n) = 2T(n/2) + cn$$
$$(\text{for } n \geq 2)$$

- What does this resolve to?

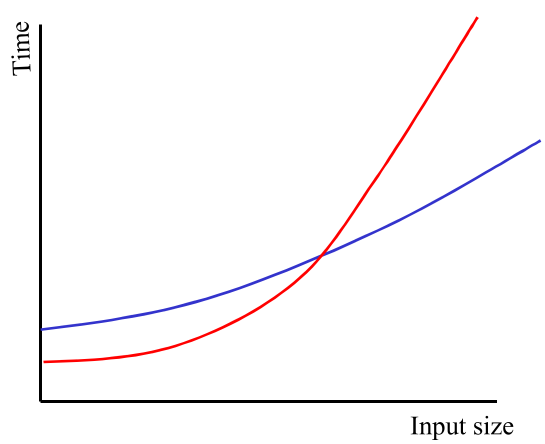
Recursion Tree for $T(n) = 2T(n/2)+cn$



$T(n) = 2T(n/2) + cn = \Theta(n \log n)$

37

Insertion vs. Merge Sort



38

Running time comparison

$T_1(n) = 0.1\, n^2$ $T_2(n) = 10\, n \log n$

Input size, n	$\Theta(n^2)$	$\Theta(n \log n)$
10	10 µsec	332 µsec
100	1 msec	6.64 msec
1,000	100 msec	100 msec
10,000	10 sec	1.3 sec
100,000	17 min	16 sec
1,000,000	28 hours	3 min
10,000,000	116 days	39 min

More comparisons

Input size	n	$n \log n$	n^2	n^3	1.5^n	2^n	$n!$
10	<1sec	<1sec	<1sec	<1sec	<1sec	<1sec	4sec
30	<1sec	<1sec	<1sec	<1sec	<1sec	18min	10^{25} years
50	<1sec	<1sec	<1sec	<1sec	11 min	36 years	too long
100	<1sec	<1sec	<1sec	1sec	12892 years	10^{17} years	too long
1,000	<1sec	<1sec	1sec	18 min	too long	too long	too long
10,000	<1sec	<1sec	2 min	12 days	too long	too long	too long
100,000	<1sec	2 sec	3 hours	32 years	too long	too long	too long
1,000,000	1sec	20sec	12 days	31710 years	too long	too long	too long

Running times on a processor than can execute one million steps per second. 'Too long' means $> 10^{25}$ years

Correctness

- Recall that to argue correctness it is not enough to try your algorithm on a few test cases
- Both *incremental* and *divide-and-conquer* algorithms are usually shown to be correct by means of *mathematical induction*
  - For incremental use “weak induction”
  - For divide-and-conquer use “strong induction”
- In general, analyze loops using a *loop invariant*, a property that holds at the start of each iteration which, upon exiting, can be used to show that a section of code is correct

41

Mathematical Induction

- Let $P(k)$ denote some property of natural number k

<u>Examples</u>: $P(k)$ means: $1+3+5+\ldots+2k-1 = k^2$

$P(k)$ means: “my algorithm correctly sorts any array of size k”

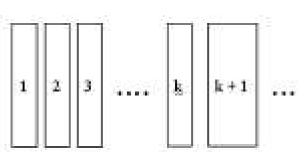
$P(k)$ means: “my algorithm takes time at most ck for some $c>0$”

1. Establish one or more base cases, e.g., $P(0)$, $P(1)$

2a. Show $P(k) \Rightarrow P(k+1)$ (weak/standard induction)

or

2b. Show $\{P(1),\ldots, P(k)\} \Rightarrow P(k+1)$ (strong induction)



42

Insertion Sort

$P(k)$: At the start of the iteration with $j = k \geq 2$, the subarray $A[1: k-1]$ contains the elements originally in $A[1: k-1]$ in ascending order

We want to show that $P(k)$ holds for all $2 \leq k \leq n+1$

```
Sort(A, n)
for j ← 2 to n do
    k ← A[j]
    i ← j − 1
    while i > 0 and A[i] > k do
        A[i + 1] ← A[i]
        i ← i + 1
    A[i + 1] ← k
```

43

Merge Sort

$P(k)$: If $r - p + 1 = k$, then Merge Sort sorts correctly $A[p:r]$

We want to show that $P(k)$ holds for all $k \geq 1$

```
MergeSort(A, p, r)
  if p = r then return
  else
      q ← (p+r)/2
      MergeSort(A, p, q)
      MergeSort(A, q+1, r)
      Merge(A,p,q,r)
end
```

Base case: $k = 1$

Strong induction: Assume $P(1), P(2), \ldots, P(k-1)$ and show $P(k)$

44