Week 4
Algorithm Analysis & Intro To Sorting

Data Structures and Algorithms

Algorithms Analysis

Adapted from Haluk Bingöl and
David Reed

Simplified Model > Example … Geometric Series Sum …

```java
public class GeometrikSeriesSumPower {

    public static void main(String[] args) {
        System.out.println("1, 4: " + powerA(1, 4));
        System.out.println("1, 4: " + powerB(1, 4));
        System.out.println("2, 4: " + powerA(2, 4));
        System.out.println("2, 4: " + powerB(2, 4));
    }

    ...
1   public static int powerA(int x, int n) {
2       int result = 1;
3       for (int i = 1; i <= n; ++i) {
4           result *= x;
5       }
6       return result;
7   }
8
9   public static int powerB(int x, int n) {
10      if (n == 0) {
11          return 1;
12      } else if (n % 2 == 0) { // n is even
13          return powerB(x * x, n / 2);
14      } else { // n is odd
```

powerB

$$x^n = \begin{cases} 1 & n=0 \\ (x^2)^{\lfloor n/2 \rfloor} & 0<n,\ n \text{ is even} \\ x(x^2)^{\lfloor n/2 \rfloor} & 0<n,\ n \text{ is odd} \end{cases}$$

powerB	n=0	0<n n even	0<n n odd
10	3	3	3
11	2	-	-
12	-	5	5
13	-	10+T(⌊n/2⌋)	-
15	-	-	12+T(⌊n/2⌋)
Total	5	18+T(⌊n/2⌋)	20+T(⌊n/2⌋)

```
1, 4: 1
1, 4: 1
2, 4: 16
2, 4: 16
```

out

powerB

$$x^n = \begin{cases} 5 & n=0 \\ 18+T(\lfloor n/2 \rfloor) & 0<n,\ n \text{ is even} \\ 20+T(\lfloor n/2 \rfloor) & 0<n,\ n \text{ is odd} \end{cases}$$

Simplified Model > Example … Geometric Series Sum …

Let $n = 2^k$ for some $k>0$.
Since n is even, $\lfloor n/2 \rfloor = n/2 = 2^{k-1}$.

For $n = 2^k$, $T(2^k) = 18 + T(2^{k-1})$.

Using repeated substitution

$$
\begin{aligned}
T(2^k) &= 18 + T(2^{k-1}) \\
&= 18 + 18 + T(2^{k-2}) \\
&= 18 + 18 + 18 + T(2^{k-3}) \\
&\dots \\
&= 18j + T(2^{k-j})
\end{aligned}
$$

Substitution stops when $k = j$

$$
\begin{aligned}
T(2^k) &= 18k + T(1) \\
&= 18k + 20 + T(0) \\
&= 18k + 20 + 5 \\
&= 18k + 25.
\end{aligned}
$$

$n = 2^k$ then $k = \log_2 n$

$\mathbf{T(2^k) = 18 \log_2 n + 25}$

powerB

$$
x^n = \begin{cases} 5 & n=0 \\ 18+T(\lfloor n/2 \rfloor) & 0<n,\ n \text{ is even} \\ 20+T(\lfloor n/2 \rfloor) & 0<n,\ n \text{ is odd} \end{cases}
$$

Simplified Model > Example … Geometric Series Sum …

Let $n = 2^k$ for some $k>0$.
Since n is even, $\lfloor n/2 \rfloor = n/2 = 2^{k-1}$.

For $n = 2^k$, $T(2^k) = 18 + T(2^{k-1})$.

Using repeated substitution
$T(2^k) = 18 + T(2^{k-1})$
$\quad = 18 + 18 + T(2^{k-2})$
$\quad = 18 + 18 + 18 + T(2^{k-3})$
$\quad$ …
$\quad = 18j + T(2^{k-j})$

Substitution stops when $k = j$
$T(2^k) = 18k + T(1)$
$\quad = 18k + 20 + T(0)$
$\quad = 18k + 20 + 5$
$\quad = 18k + 25.$

$n = 2^k$ then $k = \log_2 n$

$T(2^k) = 18 \log_2 n + 25$

Suppose $n = 2^k-1$ for some $k>0$.
Since n is odd,
$\lfloor n/2 \rfloor = \lfloor (2^k-1)/2 \rfloor$
$\quad = (2^k-2)/2$
$\quad = 2^{k-1}-1$

$= 2^{k-1}$.

For $n = 2^k-1$,
$T(2^k-1) = 20 + T(2^{k-1}-1)$, $k>1$.

Using repeated substitution
$T(2^k-1) = 20 + T(2^{k-1}-1)$
$\quad = 20 + 20 + T(2^{k-2}-1)$
$\quad = 20 + 20 + 20 + T(2^{k-3}-1)$
$\quad$ …
$\quad = 20j + T(2^{k-j}-1)$

Substitution stops when $k = j$
$T(2^k-1) = 20k + T(2^0-1)$
$\quad = 20k + T(0)$
$\quad = 20k + 5.$

$n = 2^k-1$ then $k = \log(n+1)$

powerB

$$x^n = \begin{cases} 5 & n=0 \\ 18+T(\lfloor n/2 \rfloor) & 0<n,\ n \text{ is even} \end{cases}$$

Simplified Model > Example … Geometric Series Sum …

Let $n = 2^k$ for some $k>0$.
Since n is even, $\lfloor n/2 \rfloor = n/2 = 2^{k-1}$.

For $n = 2^k$, $T(2^k) = 18 + T(2^{k-1})$.

Using repeated substitution

$$
\begin{aligned}
T(2^k) &= 18 + T(2^{k-1}) \\
&= 18 + 18 + T(2^{k-2}) \\
&= 18 + 18 + 18 + T(2^{k-3}) \\
&\dots \\
&= 18j + T(2^{k-j})
\end{aligned}
$$

Substitution stops when $k = j$

$$
\begin{aligned}
T(2^k) &= 18k + T(1) \\
&= 18k + 20 + T(0) \\
&= 18k + 20 + 5 \\
&= 18k + 25.
\end{aligned}
$$

$n = 2^k$ then $k = \log_2 n$

$$\mathbf{T(2^k) = 18 \log_2 n + 25}$$

Suppose $n = 2^k-1$ for some $k>0$.
Since n is odd,

$$
\begin{aligned}
\lfloor n/2 \rfloor &= \lfloor (2^k-1)/2 \rfloor \\
&= (2^k-2)/2 \\
&= 2^{k-1}-1
\end{aligned}
$$

$= 2^{k-1}$.

For $n = 2^k-1$

$T(2^k-1) = 20$

average $19(\lfloor \log_2(n+1) \rfloor + 1) + 18$

Using repeated substitution

$$
\begin{aligned}
T(2^k-1) &= 20 + T(2^{k-1}-1) \\
&= 20 + 20 + T(2^{k-2}-1) \\
&= 20 + 20 + 20 + T(2^{k-3}-1) \\
&\dots \\
&= 20j + T(2^{k-j}-1)
\end{aligned}
$$

Substitution stops when $k = j$

$$
\begin{aligned}
T(2^k-1) &= 20k + T(2^0-1) \\
&= 20k + T(0) \\
&= 20k + 5.
\end{aligned}
$$

powerB

$$
x^n = \begin{cases} 5 & n=0 \\ 18+T(\lfloor n/2 \rfloor) & 0<n,\ n \text{ is even} \end{cases}
$$

$n = 2^k-1$ then $k = \log(n+1)$

Simplified Model > Example … Geometric Series Sum …

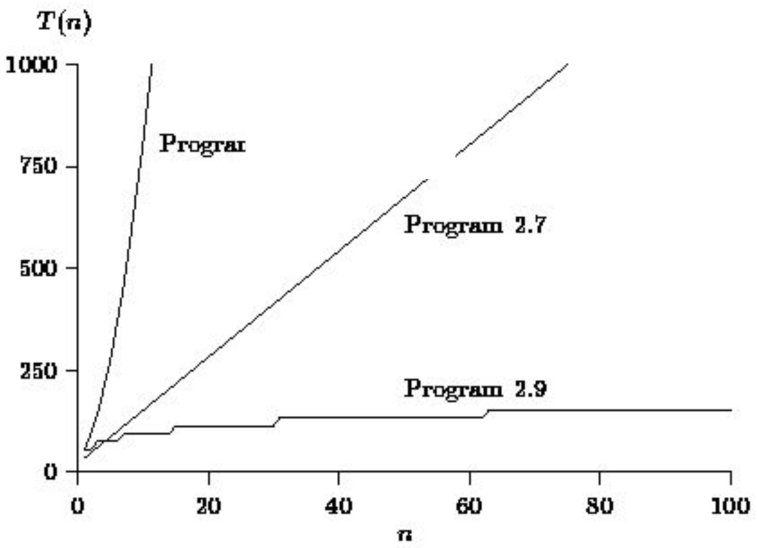
```
public class GeometrikSeriesSumPower {

    public static void main(String[] args) {
        System.out.println("s 2, 4: " + geometrikSeriesSumPower (2,
   4));
    }

    ...
    public static int geometrikSeriesSumPower (int x
        return powerB(x, n + 1) - 1 / (x - 1);
    }

    public static int powerB(int x, int n) {
        if (n == 0) {
            return 1;
        } else if (n % 2 == 0) { // n is even
            return powerB(x * x, n / 2);
        } else { // n is odd
            return x * powerB(x * x, n / 2);
        }
    }
}
```

algorithm	T(n)
Sum	$11/2\ n^2 + 47/2\ n + 27$
Horner	$13n + 22$
Power	$19(\lfloor \log_2(n+1) \rfloor + 1) + 18$

```
s 2, 4: 31
```

out

Comparison



algorithm	T(n)
Sum	$11/2\ n^2 + 47/2\ n + 27$
Horner	$13n + 22$
Power	$19(\lfloor \log_2(n+1) \rfloor + 1) + 18$

Algorithm efficiency

when we want to classify the efficiency of an algorithm, we must first identify the costs to be measured

- memory used? sometimes relevant, but not usually driving force
- execution time? dependent on various factors, including computer specs
- # of steps somewhat generic definition, but most useful

to classify an algorithm's efficiency, first identify the steps that are to be measured

e.g., for searching: # of inspections, …

for sorting: # of inspections, # of swaps, # of inspections + swaps, …

must focus on key steps (that capture the behavior of the algorithm)

- e.g., for searching: there is overhead, but the work done by the algorithm is dominated by the number of inspections

Best vs. average vs. worst case

when measuring efficiency, you need to decide what case you care about

- best case: usually not of much practical use

  the best case scenario may be rare, certainly not guaranteed

- average case: can be useful to know

  on average, how would you expect the algorithm to perform

  can be difficult to analyze – must consider all possible inputs and calculate the average performance across all inputs

- worst case: most commonly used measure of performance

  provides upper-bound on performance, guaranteed to do no wors

sequential search: best? average? worst?

Big-Oh (intuitively)

intuitively: an algorithm is O(f(N)) if the # of *steps* involved in solving a problem of size N has f(N) as the dominant term

O(N): 5N 3N + 2 N/2 – 20
$O(N^2)$: $N^2 N^2 + 100$ $10N^2 – 5N + 100$
…

why aren't the smaller terms important?

- big-Oh is a "long-term" measure
- when N is sufficiently large, the largest term dominates

consider $f_1(N) = 300*N$ (a very steep line) & $f_2(N) = ½*N^2$ (a very gradual quadratic)

in the short run (i.e., for small values of N), $f_1(N) > f_2(N)$

e.g., $f_1(10) = 300*10 = 3{,}000 > 50 = ½*10^2 = f_2(10)$

in the long run (i.e., for large values of N), $f_1(N) < f_2(N)$

e.g., $f_1(1{,}000) = 300*1{,}000 = 300{,}000 < 500{,}000 = ½*1{,}000^2 = f_2(1{,}000)$

Big-Oh and rate-of-growth

big-Oh classifications capture rate of growth

- for an O(N) algorithm, doubling the problem size doubles the amount of work

e.g., suppose Cost(N) = 5N – 3

- Cost(S) = 5S – 3
- Cost(2S) = 5(2S) – 3 = 10S - 3

- for an O(N log N) algorithm, doubling the problem size more than doubles the amount of work

e.g., suppose Cost(N) = 5N log N + N

- Cost(S) = 5S log S + S
- Cost(2S) = 5(2S) log (2S) + 2S = 10S(log(S)+1) + 2S = 10S log S + 12S

- for an O(N^2) algorithm, doubling the problem size quadruples the amount of work

Big-Oh of searching/sorting

sequential search: worst case cost of finding an item in a list of size N

- may have to inspect every item in the list

Cost(N) = N inspections + overhead

□ O(N)

selection sort: cost of sorting a list of N items

- make N-1 passes through the list, comparing all elements and performing one swap

Cost(N) = (1 + 2 + 3 + … + N-1) comparisons + N-1 swaps + overhead

= N*(N-1)/2 comparisons + N-1 swaps + overhead

= $\frac{1}{2} N^2 - \frac{1}{2} N$ comparisons + N-1 swaps + overhead

□ $O(N^2)$

General rules for analyzing algorithms

1. **for loops:** the running time of a for loop is at most

   running time of statements in loop × number of loop iterations

```
for (int i = 0; i < N; i++) {
    sum += nums[i];
}
```

2. **nested loops:** the running time of a statement in nested loops is

   running time of statement in loop × product of sizes of the loops

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        nums1[i] += nums2[j] + i;
    }
}
```

General rules for analyzing algorithms

3. consecutive statements: the running time of consecutive statements is

   sum of their individual running times

```
int sum = 0;
for (int i = 0; i < N; i++) {
    sum += nums[i];
}
double avg = (double)sum/N;
```

4. if-else: the running time of an if-else statement is at most

   running time of the test + *maximum running time of the if and else cases*

```
if (isSorted(nums)) {
    index = binarySearch(nums, desired);
}
else {
    index = sequentialSearch(nums, desired);
}
```

Exercises

consider an algorithm whose cost function is

$$Cost(N) = 12N^3 – 5N^2 + N – 300$$

intutitively, we know this is $O(N^3)$

formally, what are values of C and T that meet the definition?

- an algorithm is $O(N^3)$ if there exists a positive constant C & non-negative integer T such that for all N ≥ T, # of steps required ≤ $C*N^3$

consider "merge3-sort"

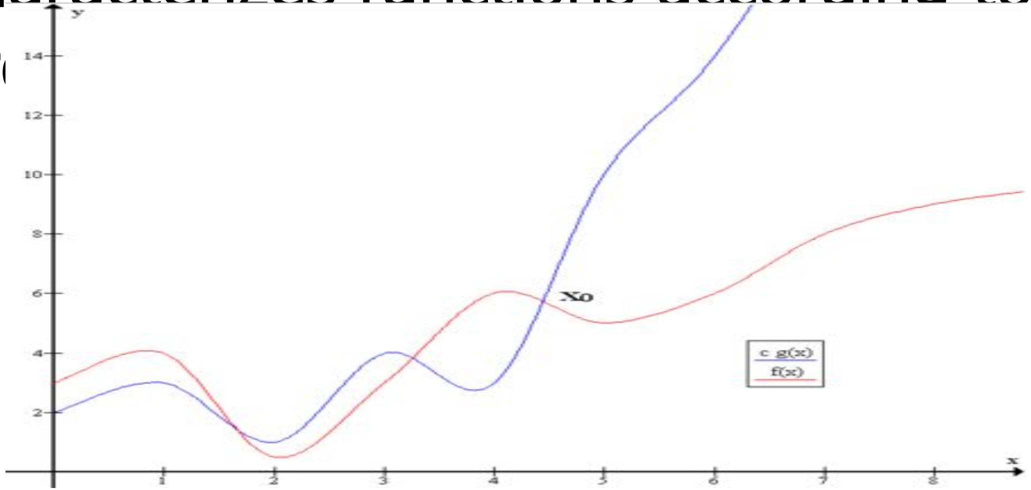
1. If the range to be sorted is size 0 or 1, then DONE.
2. Otherwise, calculate the indices 1/3 and 2/3 of the way through the list.
3. Recursively sort each third of the list.
4. Merge the three sorted sublists together.

what is the recurrence relation that defines the cost of this algorithm?

what is it's big-Oh classification?

Big-Oh

- is used to classify algorithms according to how their running time or space requirements grow as the input size grows
- characterizes functions according to their gr



Big-Oh

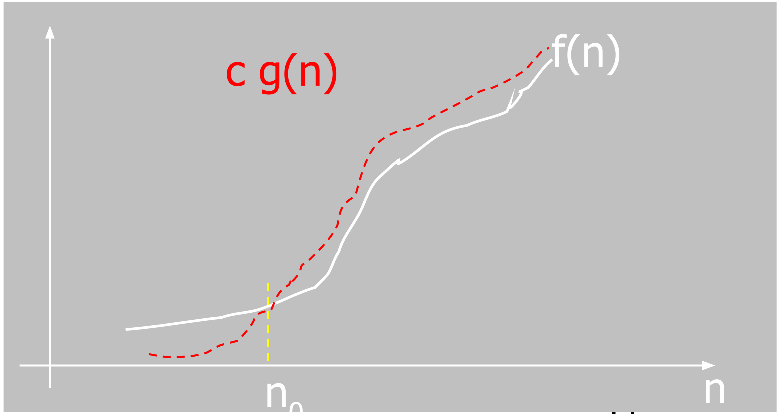
Definition

$f(n) = O(g(n))$ iff

$g(n) > 0, \quad \forall n > 0$

$\exists\, c \in R^+, \quad \exists\, n_0 \in Z^+ \ni$

$f(n) \leq c\, g(n)$ for $n_0 \leq n$



- upper bound

Some Properties

$f(n) = 1^2 + 2^2 + \ldots + n^2$

$= 1/3\ n\ (n+1/2)\ (n+1)$

$= 1/3\ n^3 + 1/2\ n^2 + 1/6\ n$

$= O(n^3)$

$= 1/3\ n^3 + O(n^2)$

Remark

$f(n) = O(1/3\ n^3)$ don't write constant

Mathematical Background>Big-Oh
Some Properties ...

Note that “=“ is not in the mathematical sense

- $\frac{1}{2} n^2 + n = O(n^2)$ √
- $O(n^2) = \frac{1}{2} n^2 + n$ X

Mathematical Background>Big-Oh
Some Properties ...

Theorem

If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$ then

- $f_1(n) + f_2(n) = \max\left(O\left(g_1(n)\right), O\left(g_2(n)\right)\right)$
- $f_1(n) \times f_2(n) = O\left(g_1(n) \times g_2(n)\right)$

meraklisina.com

Mathematical Background>Big-Oh
Some Properties ...

Theorem

Consider polynomial $f(n) = \sum_{i=0}^{m} a_i n^i$ where a_m>0.

Then $f(n) = O(n^m)$.

meraklisina.com

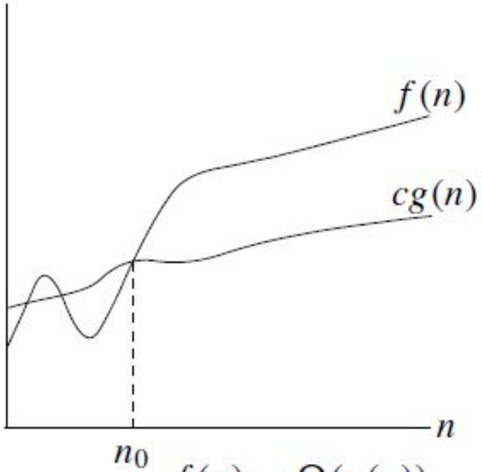
Mathematical Background>Big-Oh
Some Properties ...

- $f(n) = O(f(n))$
- $c\, O(f(n)) = O(f(n))$
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n))) = O(f(n))$
- $O(f(n))\, O(g(n)) = O(f(n)\, g(n))$
- $O(f(n)\, g(n)) = f(n)\, O(g(n))$

Omega

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

asymptotic lower bound



$f(n) = \Omega(g(n))$

Mathematical Background>Omega

Some Properties ...

Theorem

Consider polynomial $f(n) = \sum_{i=0}^{m} a_i n^i$ where $a_m > 0$.

Then $f(n) = \Omega(n^m)$.

Theta

$$\Theta(g(n)) = \{f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .^1$$

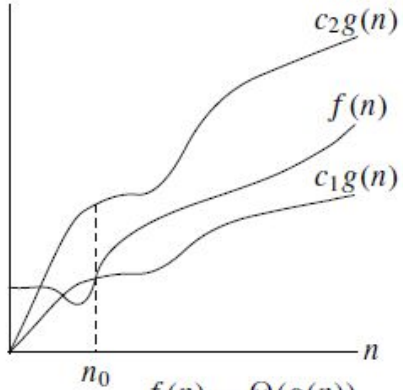
- g(n) is an asymptotically tight bound for f (n).

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

c_1, c_2, and n_0

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2 .$$



$f(n) = \Theta(g(n))$

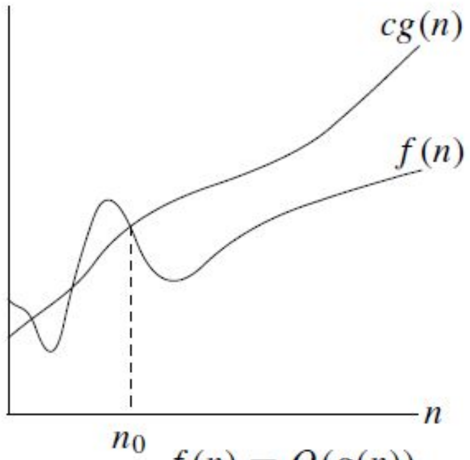
(a)

$c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$

Mathematical Background
Big-Oh

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

- For only an asymptotic upper bound, we use O-notation



$f(n) = O(g(n))$

Mathematical Background> Theta … Example

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

Consider $$f(n) = \sum_{i=0}^{m} a_i n^i$$

$f(n) = O(n^m)$

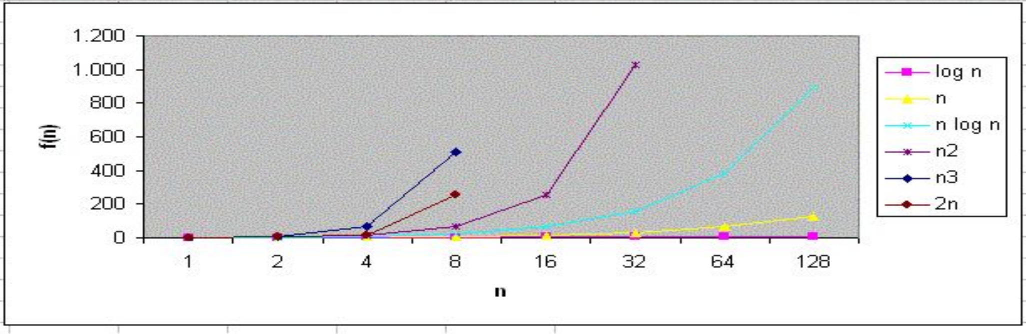
$f(n) = \Omega(n^m)$

So $f(n) = \Theta(n^m)$

Comparison of Orders

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

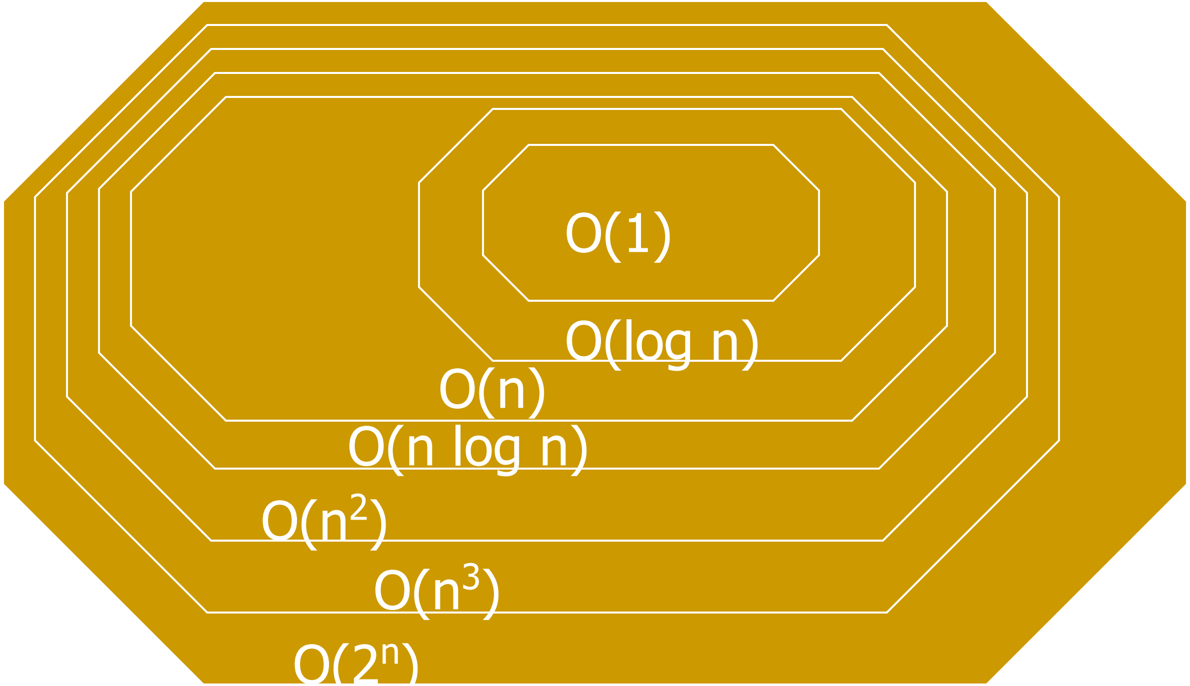
n	$\log_2(n)$	n	$n \log_2(n)$	n^2	n^3	2^n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4.096	65.536
32	5	32	160	1.024	32.768	4.294.967.296
64	6	64	384	4.096	262.144	18.446.744.073.709.600.000
128	7	128	896	16.384	2.097.152	340.282.366.920.938.000.000.000.000.000.000.000.000



meraklisina.com

Comparison of Orders

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



meraklisina.com

O() Analysis of Running Time

Sequential Composition

Worst-case running time of statements

S_1;
S_2;
…
S_m;

is $O(\max(T_1(n), T_2(n), \ldots, T_m(n))$

where

$O(T_i(n))$ is the running time of statement S_i

O() Analysis of Running Time
Iteration

Worst-case running time of statements

```
for(S_1; S_2;  S_3)
  S_4;
```

is $O(max(T_1(n), T_2(n)x(I(n)+1), T_3(n)xI(n), T_4(n)xI(n))$

where

$O(T_i(n))$ is the running time of statement S_i

$I(n)$ is the number of iterations executed in the worst case

Conditional Execution

Worst-case running time of statements

```
if (S1)
    S2;
else
    S3;
```

is $O(\max(T_1(n), T_2(n), T_3(n))$

where

$O(T_i(n))$ is the running time of statement S_i

O() Analysis of Running Time Example

```
1int findMaximum(int[] a) {
2     int result = a[0];
3     for (int i = 1; i < a.length; ++i) {
4          if (result < a[i]) {
5               result = a[i];
6          }
7     }
8     return result;
9}
```

Worst-case running time
    if statement 5 executes all the time

When ?

Best-case running time
    if statement 5 never executes

When ?

On-the-average running time
    if statement 5 executes half of

Analysis of insertion sort

Best case?

Worst case?

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
```

Analyzing recursive algorithms

recursive algorithms can be analyzed by defining a recurrence relation:

cost of searching N items using binary search =
cost of comparing middle element + cost of searching correct half (N/2 items)

more succinctly: Cost(N) = Cost(N/2) + C

Cost(N)	= Cost(N/2) + C	can unwind Cost(N/2)
	= (Cost(N/4) + C) + C	
	= Cost(N/4) + 2C	can unwind Cost(N/4)
	= (Cost(N/8) + C) + 2C	
	= Cost(N/8) + 3C	can continue unwinding
	= …	
	= Cost(1) + ($\log_2 N$)*C	
	= C $\log_2$ N + C' where C' = Cost(1)	
	□ O(log N)	

Sorting Algorithms

Slides adapted from Tolga Can

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
- An ***internal sort*** requires that the collection of data fit entirely in the computer's main memory.
- We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
- We will analyze only internal sorting algorithms.
- Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases,

Sorting Algorithms

- There are many sorting algorithms, such as:
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort

- The first three are the foundations for faster

Selection Sort

- The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.

Sorted **Unsorted**

23	78	45	8	32	56	Original List
8	78	45	23	32	56	After pass 1
8	23	45	78	32	56	After pass 2
8	23	32	78	45	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

Selection Sort (cont.)

```
void selectionSort(double[] a, int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
       if (a[j] < a[min]) min = j;
    swap(a, i, min);
  }
}


void swap(double[] a, int lhs, int rhs )
{
```

Selection Sort -- Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

  So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.

    - Ignoring other operations does not affect our final result.

- In selectionSort function, the outer for loop executes n-1 times.
- We invoke swap function once at each iteration.

  Total Swaps: n-1

  Total Moves: 3*(n-1)   (Each swap has three moves)

Selection Sort – Analysis (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.
  - # of key comparisons = 1+2+...+n-1 = n*(n-1)/2
  - **So, Selection sort is $O(n^2)$**
- The best case, the worst case, and the average case of the selection sort algorithm are same. all of them are **$O(n^2)$**
  - This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
  - Since $O(n^2)$ grows so rapidly, the selection sort algorithm is appropriate only for small n.
  - Although the selection sort algorithm requires $O(n^2)$ key comparisons, it only

Comparison of *N*, *logN* and *N^2*

N	O(LogN)	O(N^2)
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

Sorted **Unsorted**

23	78	45	8	32	56	Original List
23	78	45	8	32	56	After pass 1
23	45	78	8	32	56	After pass 2
8	23	45	78	32	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

Insertion Sort Algorithm

```
void insertionSort(double[] a, int n)
{
   for (int i = 1; i < n; i++)
   {
      double tmp = a[i];

      for (int j=i; j>0 && tmp < a[j-1]; j--)
         a[j] = a[j-1];
      a[j] = tmp;
   }
}
```

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** **□ O(n)**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: 2*(n-1) □ O(n)
  - The number of key comparisons: (n-1) □ O(n)
- ***Worst-case:*** **□ $O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 2,3, …, n
  - The number of moves: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 □ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 □ $O(n^2)$
- Average-case: **□ $O(n^2)$**

Analysis of insertion sort

- Which running time will be used to characterize this algorithm?
  - Best, worst or average?
- Worst:
  - Longest running time (this is the upper limit for the algorithm)
  - It is guaranteed that the algorithm will not be worse than this.
- Sometimes we are interested in average case. But there are some problems with the average case.
  - It is difficult to figure out the average case. i.e. what is average input?
  - Are we going to assume all possible inputs are equally likely?

Bubble Sort

- The list is divided into two sublists: sorted and unsorted.
- The smallest element is bubbled from the unsorted list and moved to the sorted sublist.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up

Bubble Sort

23	78	45	8	32	56	Original List
8	23	78	45	32	56	After pass 1
8	23	32	78	45	56	After pass 2
8	23	32	45	78	56	After pass 3
8	23	32	45	56	78	After pass 4

Bubble Sort Algorithm

```
void bubleSort(double[] a, int n)
{
   bool sorted = false;
   int last = n-1;

   for (int i = 0; (i < last) && !sorted; i++){
      sorted = true;
      for (int j=last; j > i; j--)
         if (a[j-1] > a[j]{
            swap(a, j, j-]);
            sorted = false; // signal exchange
         }
   }
```

Bubble Sort – Analysis

- ***Best-case:*** **▯ O(n)**
  - Array is already sorted in ascending order.
  - The number of moves: 0 ▯ O(1)
  - The number of key comparisons: (n-1) ▯ O(n)
- ***Worst-case:*** **▯ $O(n^2)$**
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 ▯ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ▯ $O(n^2)$
- Average-case: **▯ $O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is $O(n^2)$**

Mergesort

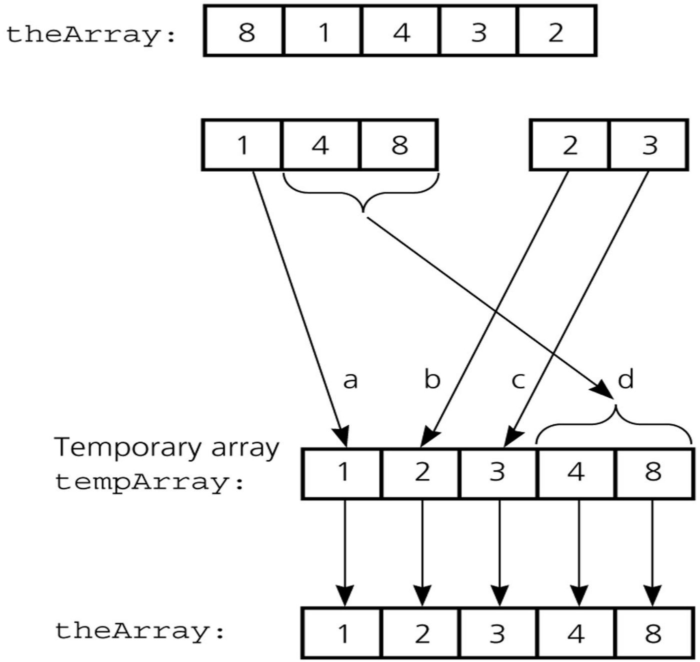
- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

```java
class MergeSort
{   void sort(int arr[], int l, int r)    {
        if (l < r)          {
            // Find the middle point
            int m = (l+r)/2;
            // Sort first and second halves
            sort(arr, l, m);
            sort(arr , m+1, r);
            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }
  // Driver method
    public static void main(String args[])
{
        int arr[] = {12, 11, 13, 5, 6, 7};
        System.out.println("Given Array");
        printArray(arr);
```

```java
static void printArray(int arr[])    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
          System.out.print(arr[i] + " ");
         System.out.println();
}
```

```java
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
{
        // Find sizes of two subarrays to be
merged
        int n1 = m - l + 1;
        int n2 = r - m;
        /* Create temp arrays */
        int L[] = new int [n1];
        int R[] = new int [n2];
        /*Copy data to temp arrays*/
        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];
        int i = 0, j = 0;
// Initial index of merged subarry array
        int k = l;
        while (i < n1 && j < n2)        {
            if (L[i] <= R[j])           {
                arr[k] = L[i];
                i++;
            }
            else            {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        /* Copy remaining elements of L[] if any */
        while (i < n1)        {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2)
        {
```

Mergesort - Example



Divide the array in half

Sort the halves

Merge the halves:

a. $1 < 2$, so move 1 from left half to `tempArray`
b. $4 > 2$, so move 2 from right half to `tempArray`
c. $4 > 3$, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

Merge

```
int MAX_SIZE = maximum-number-of-items-in-array;
void merge(double[] theArray, int first, int mid, int last) {
   double[] tempArray = new double[MAX_SIZE];// temporary array
   int first1 = first;    // beginning of first subarray
   int last1 = mid;       // end of first subarray
   int first2 = mid + 1;  // beginning of second subarray
   int last2 = last;      // end of second subarray
   int index = first1; // next available location in tempArray
   for ( ; (first1 <= last1) && (first2 <= last2); ++index) {
      if (theArray[first1] < theArray[first2]) {
         tempArray[index] = theArray[first1];
         ++first1;
      }
      else {
```

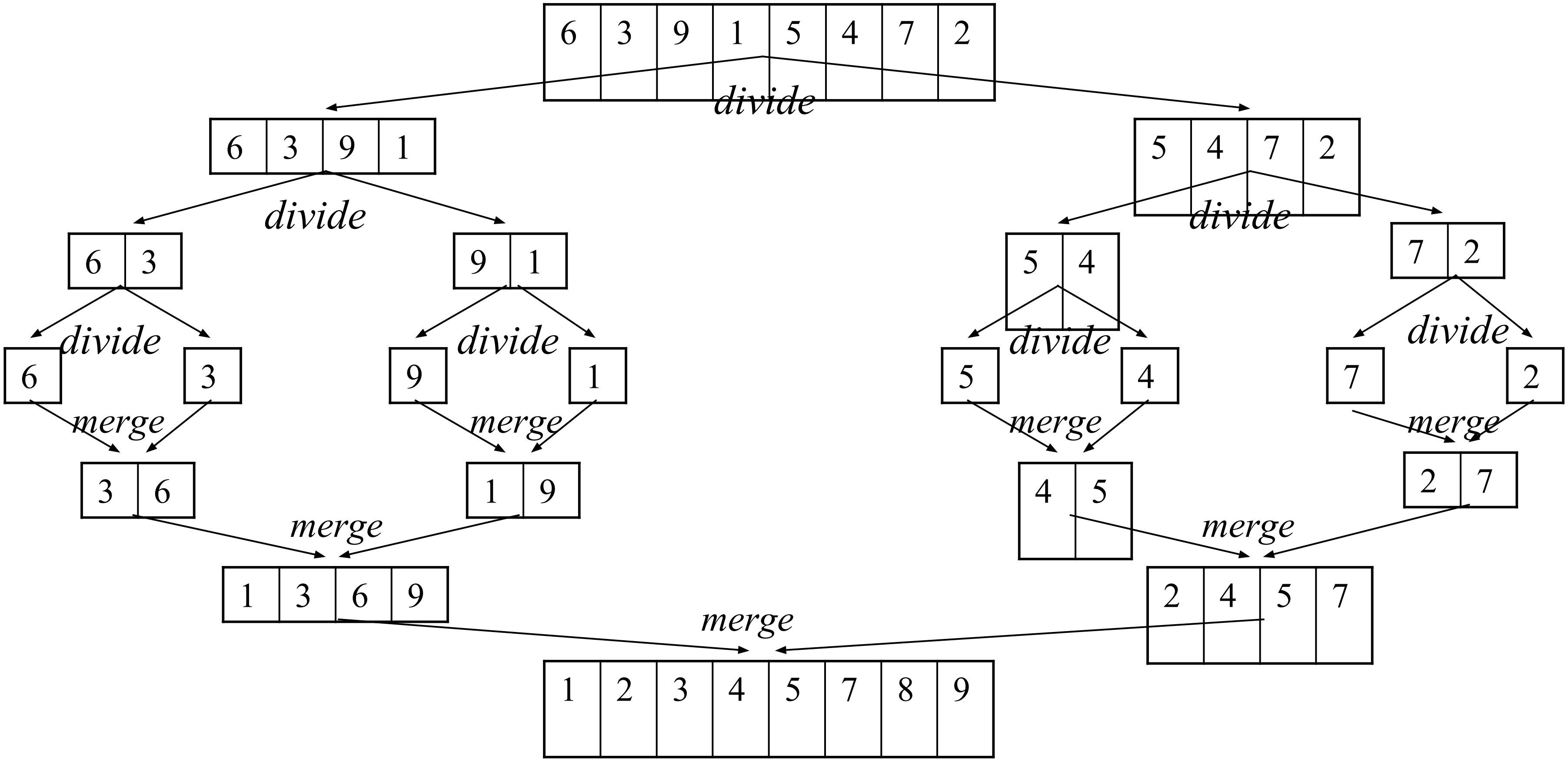
Merge (cont.)

```
 // finish off the first subarray, if necessary
   for (; first1 <= last1; ++first1, ++index)
      tempArray[index] = theArray[first1];

   // finish off the second subarray, if necessary
   for (; first2 <= last2; ++first2, ++index)
      tempArray[index] = theArray[first2];

   // copy the result back into the original array
   for (index = first; index <= last; ++index)
      theArray[index] = tempArray[index];
}  // end merge
```

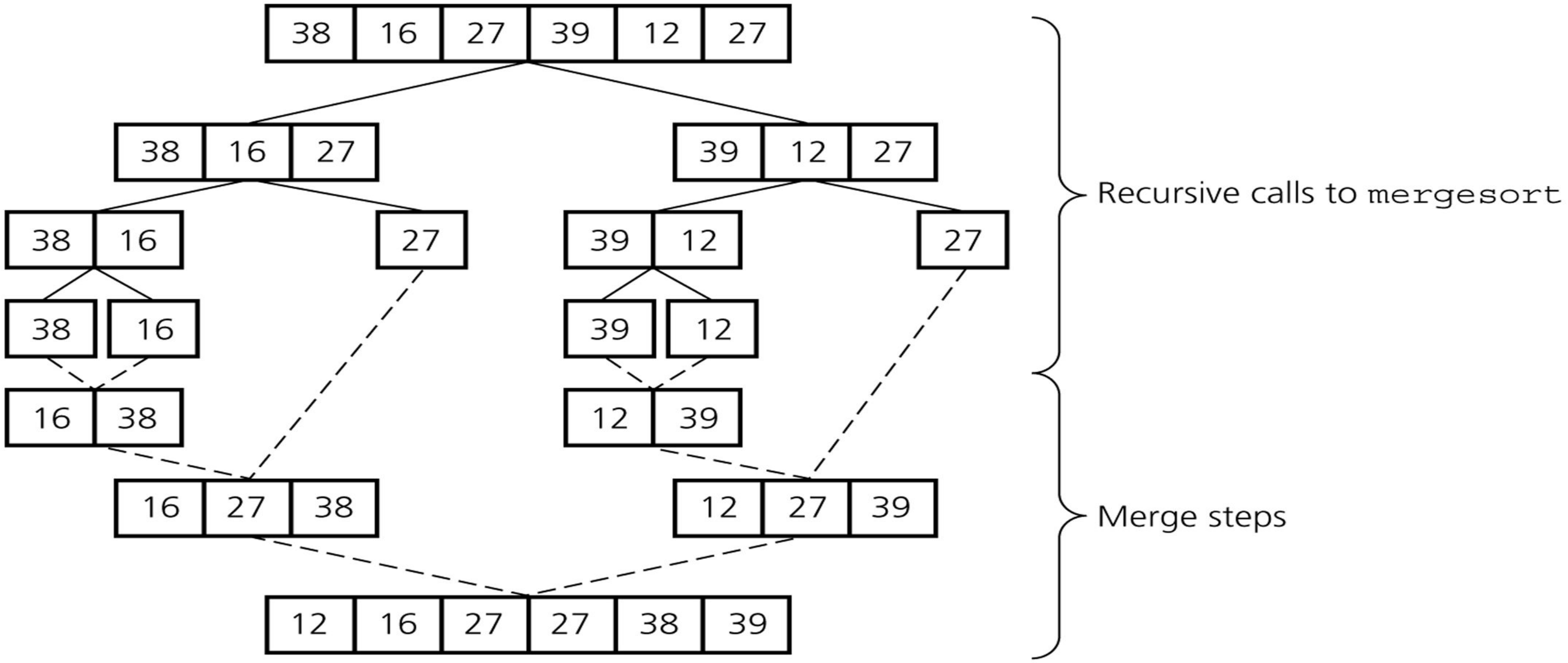
Mergesort

```
void mergesort(double[] theArray, int first, int last) {
   if (first < last) {
      int mid = (first + last)/2;    // index of midpoint
      mergesort(theArray, first, mid);
      mergesort(theArray, mid+1, last);

      // merge the two halves
      merge(theArray, first, mid, last);
   }
}  // end mergesort
```

Mergesort - Example

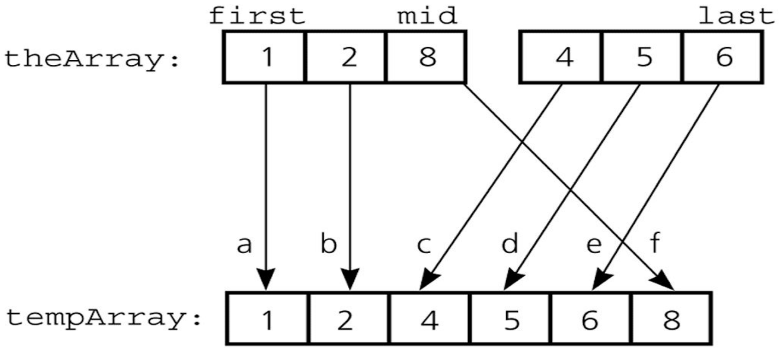


Mergesort – Example2



Mergesort – Analysis of Merge

A worst-case instance of the merge step in *mergesort*

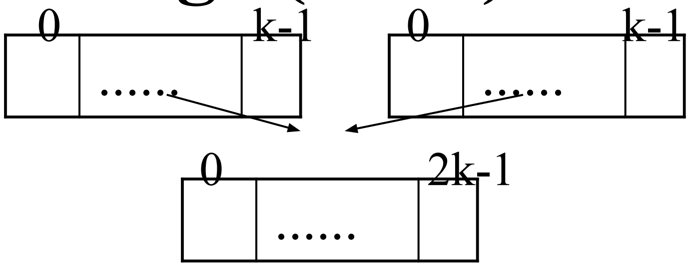


Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

Mergesort – Analysis of Merge (cont.)

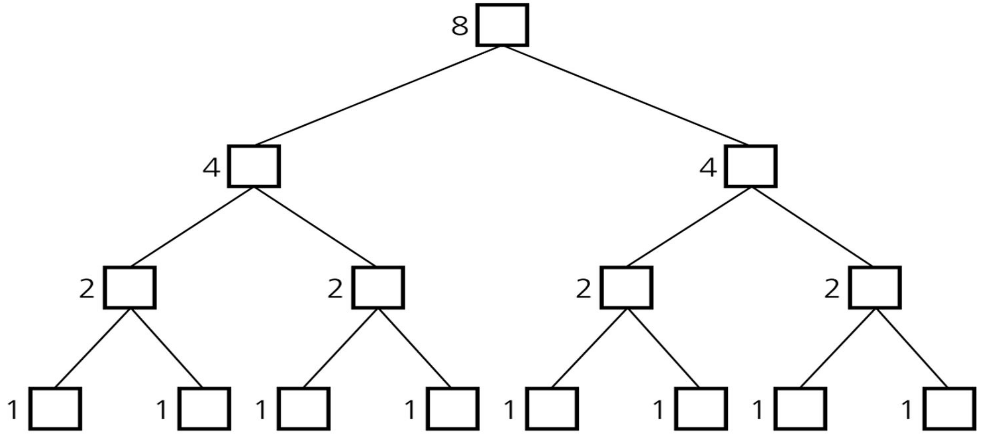
Merging two sorted arrays of size ***k***



- ***Best-case:***
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves: 2k + 2k
  - The number of key comparisons: k
- ***Worst-case:***
  - The number of moves: 2k + 2k
  - The number of key comparisons: 2k-1

Mergesort - Analysis

Levels of recursive calls to *`mergesort`*, given an array of eight items



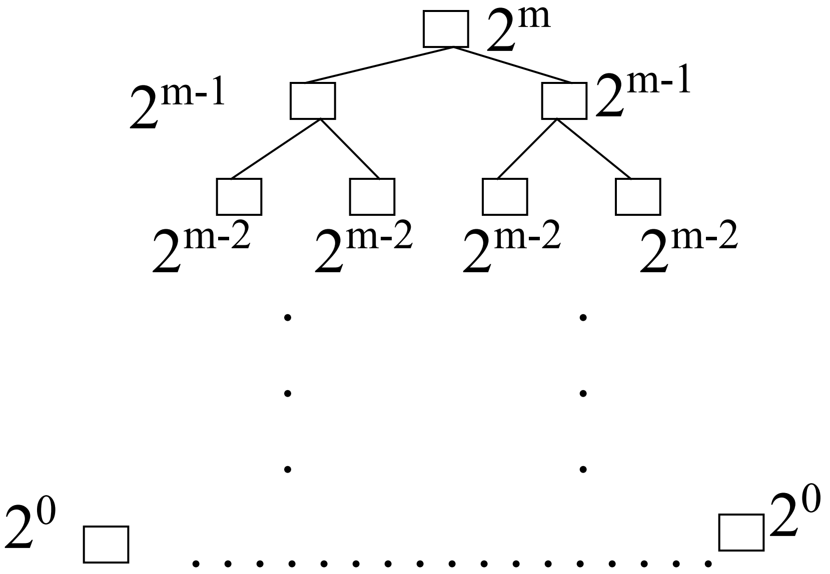
Level 0: `mergesort` 8 items

Level 1: 2 calls to `mergesort` with 4 items each

Level 2: 4 calls to `mergesort` with 2 items each

Level 3: 8 calls to `mergesort` with 1 item each

Mergesort - Analysis



level 0 : 1 merge (size 2^{m-1})

level 1 : 2 merges (size 2^{m-2})

level 2 : 4 merges (size 2^{m-3})

level m-1 : 2^{m-1} merges (size 2^0)

level m

Mergesort - Analysis

- ***Worst-case –***

The number of key comparisons:

$= 2^0*(2*2^{m-1}-1) + 2^1*(2*2^{m-2}-1) + ... + 2^{m-1}*(2*2^0-1)$

$= (2^m - 1) + (2^m - 2) + ... + (2^m – 2^{m-1})$ (m terms)

$$\sum_{i=0}^{m-1} 2^i$$

$= m*2^m –$

$= m*2^m – 2^m – 1$

Using m = log n

$\mathbf{= n * \log_2 n – n – 1}$

Mergesort – Analysis

- Mergesort is extremely efficient algorithm with respect to time.
  - Both worst case and average cases are **O (n * $\log_2$n)**

- But, mergesort requires an extra array whose size equals to the size of the original array.

- If we use a linked list, we do not need an extra array
  - But, we need space for the links
  - And, it will be difficult to divide the list into half (O(n))

Analyzing recursive algorithms

recursive algorithms can be analyzed by defining a recurrence relation:

cost of searching N items using binary search =
cost of comparing middle element + cost of searching correct half (N/2 items)

more succinctly: $\text{Cost}(N) = \text{Cost}(N/2) + C$

$$
\begin{aligned}
\text{Cost}(N) &= \text{Cost}(N/2) + C && \text{can unwind Cost}(N/2) \\
&= (\text{Cost}(N/4) + C) + C \\
&= \text{Cost}(N/4) + 2C && \text{can unwind Cost}(N/4) \\
&= (\text{Cost}(N/8) + C) + 2C \\
&= \text{Cost}(N/8) + 3C && \text{can continue unwinding} \\
&= \ldots \\
&= \text{Cost}(1) + (\log_2 N)*C \\
&= C \log_2 N + C' \text{ where } C' = \text{Cost}(1)
\end{aligned}
$$

 O(log N)

Analyzing merge sort

cost of sorting N items using merge sort =

cost of sorting left half (N/2 items) + cost of sorting right half (N/2 items) + cost of merging (N items)

more succinctly: $Cost(N) = 2Cost(N/2) + C_1 N + C_2$

$Cost(N) = 2Cost(N/2) + C_1{*}N + C_2$ can unwind Cost(N/2)

$= 2(\ 2Cost(N/4) + C_1 N/2 + C_2) + C_1 N + C_2$

$= 4Cost(N/4) + 2C_1 N + 3C_2$ can unwind Cost(N/4)

$= 4(\ 2Cost(N/8) + C_1 N/4 + C_2) + 2C_1 N + 3C_2$

$= 8Cost(N/8) + 3C_1 N + 7C_2$ can continue unwinding

$= \ldots$

$= NCost(1) + (\log_2 N) C_1 N + (N-1)\, C_2$

$= C_1 N \log_2 N + (C'+C_2)N - C_2$ where $C' = Cost(1)$

 O(N log N)