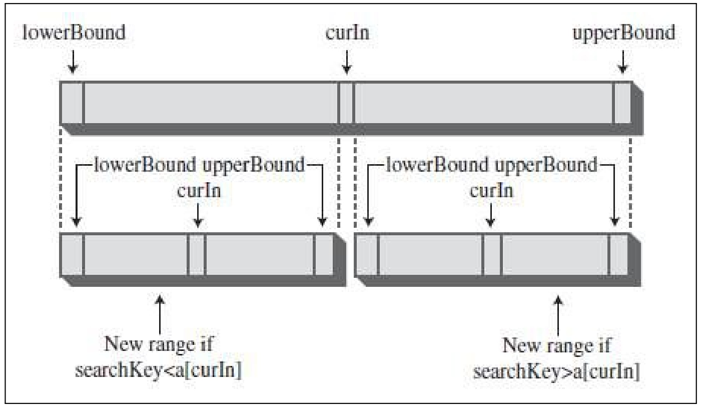
Data Structures - Recursion 2

Dr. TGI Fernando [1] [2]

[1] Email: tgi.fernando@gmail.com
[2] URL: http://tgifernando.wordpress.com/

The Recursive Binary Search

- Remember the binary search we discussed in "Arrays."
- There we wanted to find a given search key in an ordered array using the fewest number of comparisons.
- The solution was to divide the array in half, see which half the desired cell lay in, divide that half in half again, and so on.



The Recursive Binary Search (Contd.)

Here is the recursive binary search algorithm:

Precondition: $s = \{s_0, s_1, ..., s_{n1}\}$ is a sorted sequence of n ordinal values of the same type as x (search key).
Postcondition: Either the index i is returned where $s_i = x$, or -1 is returned.

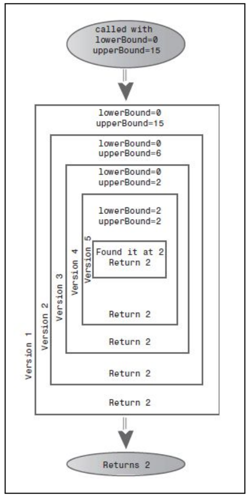
(1) If the sequence is empty, return -1.
(2) Let s_i be the middle element of the sequence.
(3) If $s_i = x$, return its index i.
(4) If $s_i < x$, apply the algorithm on the subsequence that lies above s_i.
(5) Apply the algorithm on the subsequence of s that lies below s_i.

The Recursive Binary Search (Contd.)

```
int search(int[] a, int lowerBound, int upperBound, int x) {
// PRECONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
// POSTCONDITIONS: returns i;
// if i >= 0, then a[i] == x; otherwise i == -1;
  if (lowerBound > upperBound) {
    return -1; // basis
  }
  int i = (lowerBound + upperBound)/2;
  if (a[i] == x) {
    return i; // basis
  }
  else if (a[i] < x) {
    return search(a, i+1, upperBound, x);
  }
  else {
    return search(a, lowerBound, i-1, x);
  }
}
```

The Recursive Binary Search (Contd.)

E.g.
Array items:
9 18 27 36 45 54 63 72 81 90 99 108
117 126 135 144
x = 27,
`lowerBound` = 0, and
`upperBound` = 15



The Recursive Binary Search (Contd.)

The recursive binary search runs in $O(\log n)$ time.

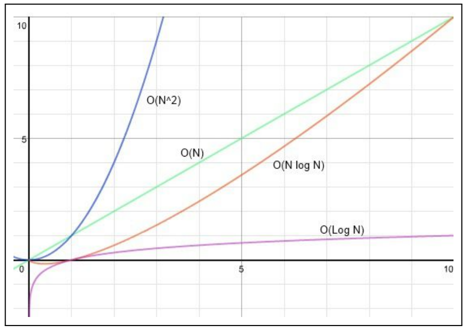
The running time is proportional to the number of recursive calls made. Each call processes a subsequence that is half as long as the previous one. So the number of recursive calls is the same as the number of times that n can be divided in two, namely $\log n$.

Divide-and-Conquer Algorithms

- The recursive binary search is an example of the divide-and-conquer approach.
- You divide the big problem into two smaller problems and solve each one separately.
- The solution to each smaller problem is the same: You divide it into two even smaller problems and solve them.
- The process continues until you get to the base case, which can be solved easily, with no further division into halves.

Mergesort

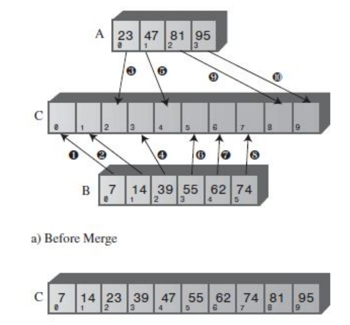
- Much more efficient sorting technique than those we saw under "Simple Sorting."
- The merge sort is $O(N * \log N)$.



- Fairly easy to implement (compared to quicksort and shellsort).
- Requires an additional array in memory, equal in size to the one being sorted.

Merging two sorted arrays

- The heart of the mergesort algorithm.
- Merging two sorted arrays A and B creates a third array, C, that contains all the elements of A and B, also arranged in sorted order.
- Let's see the merging process first.
- The circled numbers indicate the order in which elements are transferred from A and B to C.



a) Before Merge

b) After Merge

Merging operations

Step	Comparison (If Any)	Copy
1	Compare 23 and 7	Copy 7 from B to C
2	Compare 23 and 14	Copy 14 from B to C
3	Compare 23 and 39	Copy 23 from A to C
4	Compare 39 and 47	Copy 39 from B to C
5	Compare 55 and 47	Copy 47 from A to C
6	Compare 55 and 81	Copy 55 from B to C
7	Compare 62 and 81	Copy 62 from B to C
8	Compare 74 and 81	Copy 74 from B to C
9		Copy 81 from A to C
10		Copy 95 from A to C

Code - Merge operation

```
// merge A and B into C
public static void merge( int[] arrayA, int sizeA, int[]
 arrayB, int sizeB, int[] arrayC ) {
  int aDex=0, bDex=0, cDex=0;
  while(aDex < sizeA && bDex < sizeB) // neither array empty
    if( arrayA[aDex] < arrayB[bDex] )
      arrayC[cDex++] = arrayA[aDex++];
    else
      arrayC[cDex++] = arrayB[bDex++];
  while(aDex < sizeA) // arrayB is empty,
    arrayC[cDex++] = arrayA[aDex++]; // but arrayA isn t
  while(bDex < sizeB) // arrayA is empty,
    arrayC[cDex++] = arrayB[bDex++]; // but arrayB isn t
}
```

Note: This is not a recursive method.

Sorting by merging

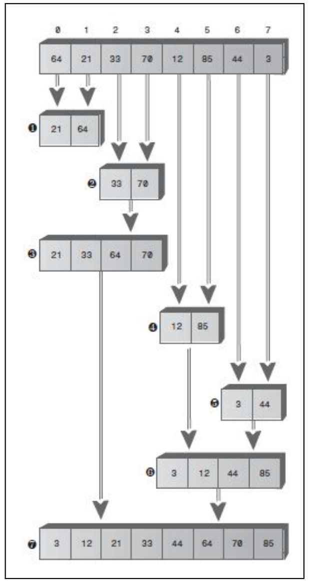
- Divide an array in half, sort each half, and then use the `merge()` method to merge the two halves into a single sorted array.
- How do you sort each half? (Recursion!!)
- You divide the half into two quarters, sort each of the quarters, and merge them to make a sorted half.
- Similarly, each pair of 8ths is merged to make a sorted quarter, each pair of 16ths is merged to make a sorted 8th, and so on.
- You divide the array again and again until you reach a subarray with only one element.
- This is the **base case**; its assumed an array with one element is already sorted.

Sorting by merging (Contd.)

- As `mergeSort()` returns from finding two arrays of one element each, it merges them into a sorted array of two elements.
- Each pair of resulting 2-element arrays is then merged into a 4-element array.
- This process continues with larger and larger arrays until the entire array is sorted.
- This is easiest to see when the original array size is a power of 2 (see the next slide).

Note: We don't merge two separate arrays into a third one, as we demonstrated in the `merge` method (see slide 11). Instead, we merge parts of a single array into itself.

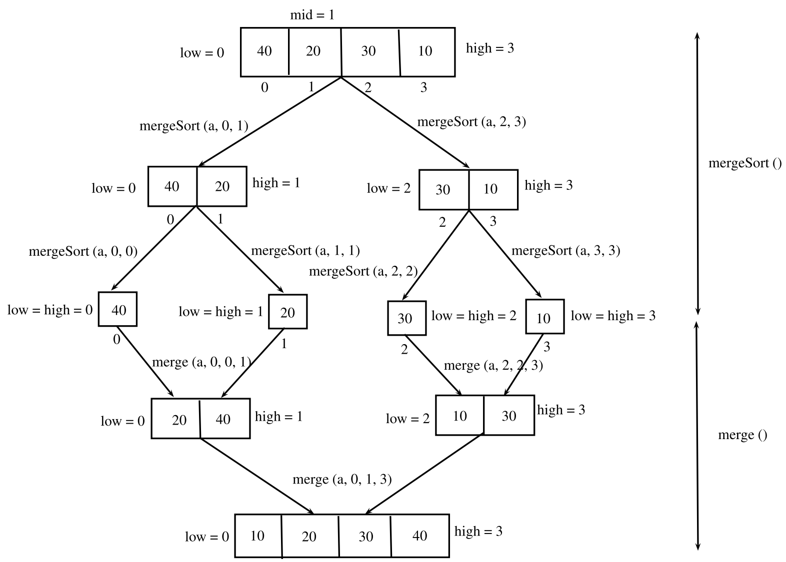
Merging larger and larger arrays



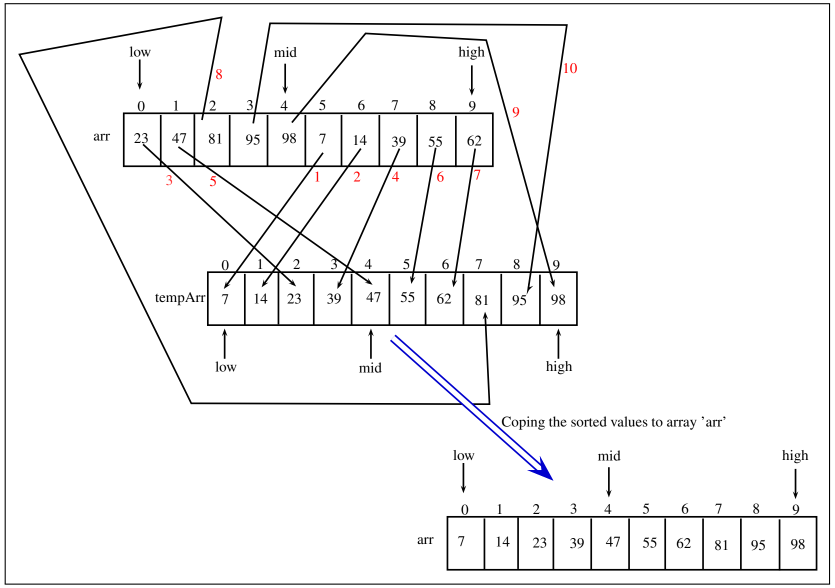
The `mergeSort()` method

```
public static void mergeSort (long[] arr, int low, int high)
 {
  if (low == high) // if range is 1,
    return; // no use sorting
  else {
    int mid = (low+high)/2; // find midpoint
    mergeSort (arr, low, mid); // sort low half
    mergeSort (arr, mid+1, high); // sort high half
    merge (arr, low, mid, high); // merge them
  }
}
```

The `mergeSort()` method - Example



The `merge()` method



Raising a number to a power

- How would you write a function to obtain x^n?
- Iterative approach - need to multiply x by itself n times. **E.g.** $2^8 = 2*2*2*2*2*2*2*2$
- Need *n-1* multiplications ($O(n)$)).
- This method is tedious for large values of n,
- What about a recursive method?

$$x^n = \begin{cases} 1 & if\ n = 0\ or\ n = 1 \\ x * x^{n-1} & if\ n > 1 \end{cases}$$

- This method also needs $n-1$ multiplications.

Raising a number to a power (Contd.)

One solution is to rearrange the problem so you multiply by multiples of 2 whenever possible, instead of by 2.

E.g.

$$\begin{aligned} 2^8 &= (2*2)*(2*2)*(2*2)*(2*2) \\ &= 4*4*4*4 \end{aligned}$$

So we've found the answer to 2^8 with only three multiplications instead of seven. <u>That's $O(\log n)$ time instead $O(n)$</u>.

How do we transform this into a recursive equation?

$$\begin{aligned} 2^8 &= 4*4*4*4 \\ &= 4^4 \\ &= (2^2)^{8/2} \end{aligned}$$

Raising a number to a power (Contd.)

Thus, the general form is

$$x^n = (x^2)^{n/2}$$

But, this form fails when n is odd because $n/2$ is not a whole number.

In this case, we can write the general form as

$$x^n = x * (x^2)^{\lfloor n/2 \rfloor}$$

Combining these two cases and base cases, we have the following recursive formula

$$x^n = \begin{cases} 1 & if\ n = 0 \\ x & if\ n = 1 \\ (x^2)^{n/2} & if\ n\ is\ even\ and\ n > 1 \\ x * (x^2)^{\lfloor n/2 \rfloor} & if\ n\ is\ odd\ and\ n > 1 \end{cases}$$