Lecture 4.1 & 4.2

- Dr. Anurag Sharma

Divide-and-Conquer Algorithms

About the topic

- Divide and conquer is an algorithm design paradigm based on multi-branched recursion.
- The divide-and-conquer approach is a **top-down** approach where the algorithm divides an instance of a problem into two or more smaller instances. This process of dividing the instances continues until they are so small that a solution is readily obtainable

Divide-and-Conquer Approach

- This approach involves these steps:
  - Step 1: **Divide** an instance of a problem into one or more smaller instances.
  - Step 2: **Conquer** by solving each of the smaller instances (use recursion until the array is sufficiently small).
  - Step 3: if needed, **combine** the solutions of the smaller instances to obtain the solution of the original instance.

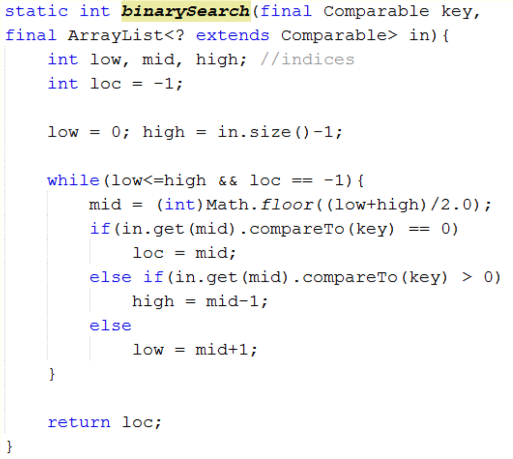
Examples

- Binary Search (how?)
- Merge sort

Binary Search

- In lecture 2.1.13 we discussed iterative version of binary search.
- Now we will discuss how divide-and-conquer technique can be applied to binary search.

Binary Search in Java (from Lec 2.1.13)



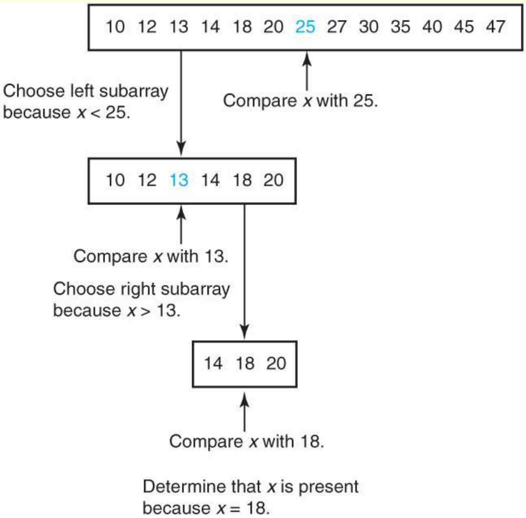
```
static int binarySearch(final Comparable key,
final ArrayList<? extends Comparable> in){
    int low, mid, high; //indices
    int loc = -1;

    low = 0; high = in.size()-1;

    while(low<=high && loc == -1){
        mid = (int)Math.floor((low+high)/2.0);
        if(in.get(mid).compareTo(key) == 0)
            loc = mid;
        else if(in.get(mid).compareTo(key) > 0)
            high = mid-1;
        else
            low = mid+1;
    }

    return loc;
}
```

Binary Search (the story)

- Binary Search locates a key *x* in a non-decreasing sorted array by first comparing x with the middle item of the array. If they are equal, the algorithm is done. If not, the array is divided into two sub-arrays, one containing all the items to the left of the middle item and the other containing all the items to the right. If x is smaller than the middle item, this procedure is then applied to the left sub-array. Otherwise, it is applied to the right sub-array. That is, *x* is compared with the middle item of the appropriate sub-array. If they are equal, the algorithm is done. If not, the sub-array is divided in two. This procedure is repeated until *x* is found or it is determined that *x* is not in the array.

Cont.

- Binary Search is the simplest kind of divide-and-conquer algorithm because the instance is broken down into only one smaller instance, so there is no combination of outputs. The solution of the original instance is the solution to the smaller instance.



10 12 13 14 18 20 25 27 30 35 40 45 47

Choose left subarray because $x < 25$.

Compare x with 25.

10 12 13 14 18 20

Compare x with 13.

Choose right subarray because $x > 13$.

14 18 20

Compare x with 18.

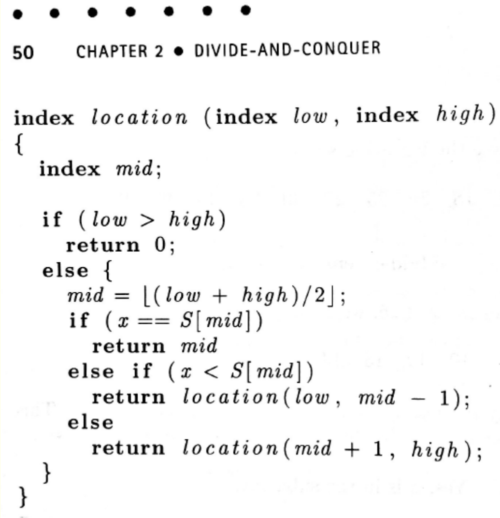
Determine that x is present because $x = 18$.

Figure 2.1 : The steps down by a human when searching with Binary Search. (Note: x = 18)

D&C version of Binary Search

- Divide part: dividing the array (or subarray) into two equal (or almost equal) parts.
- Conquer part: looking for a solution at the mid point. If can't find a solution then go to divide part again.
- Combine: not applicable here.

Code (recursive version)



50 CHAPTER 2 • DIVIDE-AND-CONQUER

```
index location (index low, index high)
{
  index mid;

  if (low > high)
    return 0;
  else {
    mid = ⌊(low + high)/2⌋;
    if (x == S[mid])
      return mid
    else if (x < S[mid])
      return location(low, mid − 1);
    else
      return location(mid + 1, high);
  }
}
```

Is recursion bad as in Fibonacci?

- Let's find out:
  - What is T(n)?
  - What is W(n)?

W(n) for binary search

Calls:	1+2	2+2	3+2	…	Last or second last: $\lfloor\log_2 n\rfloor + 2$	Last: $\lfloor\log_2 n\rfloor + 1 + 2$
↓	↓	↓	↓	↓	↓	↓
$\lVert subarray\rVert$:	$\frac{n}{2} = \frac{n}{2^1}$	$\frac{n}{4} = \frac{n}{2^2}$	$\frac{n}{8} = \frac{n}{2^3}$	…	$\frac{n}{2^{\lfloor\log_2 n\rfloor}}$	1

- $W(n) = \lfloor\log_2 n\rfloor + 1 + 2 * \lfloor\log_2 n\rfloor = 3 * \lfloor\log_2 n\rfloor + 1$
- The worst case big O order is:
- $3 * \lfloor\log_2 n\rfloor + 1 \cong \lfloor\log_2 n\rfloor$
- $\therefore O(\log_2 n)$.

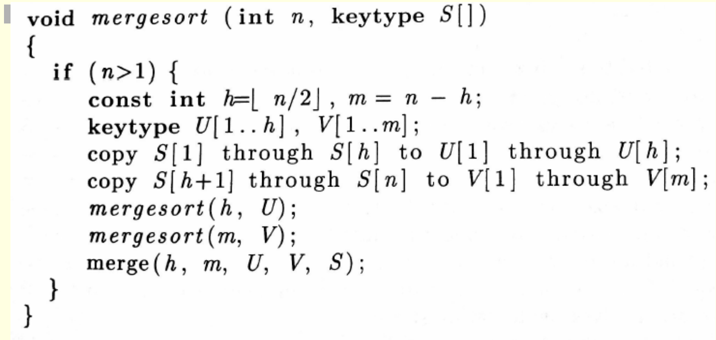
Merge Sort (story)

- Using **two-way merging**, we can combine two sorted arrays into one array. By repeatedly applying the merging procedure, we can sort an array. Eventually the size of the subarrays will become 1, and an array of size 1 is trivially sorted. This procedure is called **mergesort**.

D&C version of Mergesort

- Mergesort has three steps.
  - Step 1: Divide the array into 2 sub-arrays each of n/2.
  - Step 2: Solve each sub-array by sorting it (use recursion till array is sufficiently small).
  - Step 3: Combine solutions to the sub-arrays by merging them into a single sorted array.

Pseudocode for mergesort



```
void mergesort (int n, keytype S[])
{
  if (n>1) {
     const int h=⌊ n/2⌋, m = n − h;
     keytype U[1..h], V[1..m];
     copy S[1] through S[h] to U[1] through U[h];
     copy S[h+1] through S[n] to V[1] through V[m];
     mergesort(h, U);
     mergesort(m, V);
     merge(h, m, U, V, S);
  }
}
```

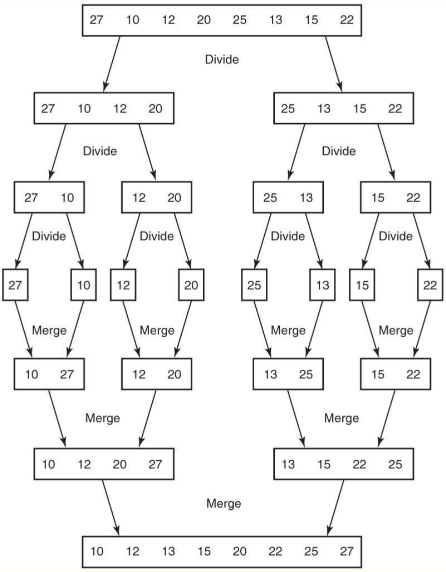
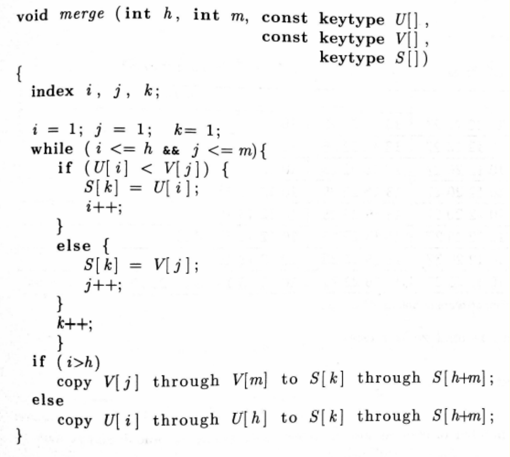


Figure 2.2: The steps done by a human when sorting with Mergesort.

Merge subroutine



```
void merge (int h, int m, const keytype U[] ,
                          const keytype V[] ,
                                keytype S[])
{
  index i , j , k;

  i = 1; j = 1;  k= 1;
  while ( i <= h && j <= m){
     if (U[ i] < V[ j]) {
        S[ k] = U[ i ];
        i++;
     }
     else {
        S[ k] = V[ j ];
        j++;
     }
     k++;
     }
  if ( i>h)
     copy V[ j ] through V[m] to S[ k] through S[ h+m];
  else
     copy U[ i ] through U[ h ] to S[ k] through S[ h+m];
}
```

Time complexity for merge sort

- What is $W(n)$ for mergesort?
- $W(n) = W(divide) + W(conquer)$
- $W(divide) = 1 + 2 + 2^2 + 2^{\lfloor \log_2 N \rfloor} + (N - 2^{\lfloor \log_2 N \rfloor}) = 2^{\lfloor \log_2 N \rfloor + 1} + N - 2^{\lfloor \log_2 N \rfloor} = 2^{\lfloor \log_2 N \rfloor} + N$
- $W(conquer) = N + N + \cdots + N = \sum_{i=1}^{\lfloor \log_2 N \rfloor} N$
- $= \mathrm{N} \lfloor \log_2 N \rfloor$
- $W(n) = 2^{\lfloor \log_2 N \rfloor} + N + \mathrm{N} \lfloor \log_2 N \rfloor$
- $W(n) \cong 2N + N \log_2 N$
- Big O would be $\mathrm{O}(N \log_2 N)$

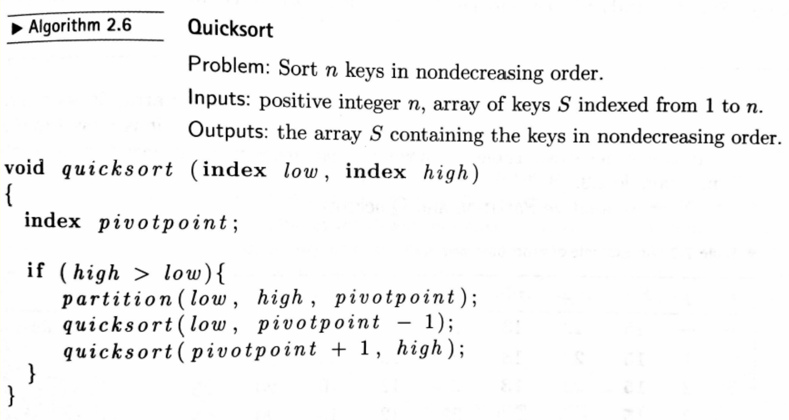
Time complexity for merge sort (simplified)

- What is $W(n)$ for mergesort?
- $W(n) = W(divide) + W(conquer)$
- $W(divide) = 1 + 2 + 2^2 + 2^{\log_2 N} = 2^{\log_2 N + 1} \cong N$
- $W(conquer) = N + N + \cdots + N = \sum_{i=1}^{\log_2 N} N$
- $= \mathrm{N} \log_2 N$
- $W(n) = N + \mathrm{N} \log_2 N \cong \mathrm{N} \log_2 N$
- Big O would be $\mathrm{O}(N \log_2 N)$

Quick Sort

- Partition exchange sort or quicksort is similar to mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively.
- In quicksort, the array is partitioned by placing all items smaller than some small **pivot item.** The pivot item can be any item.
- Quicksort then recursively continues to partition until an array with one item is left. This array is trivially sorted.

Pseudo code for quicksort



▶ Algorithm 2.6 **Quicksort**

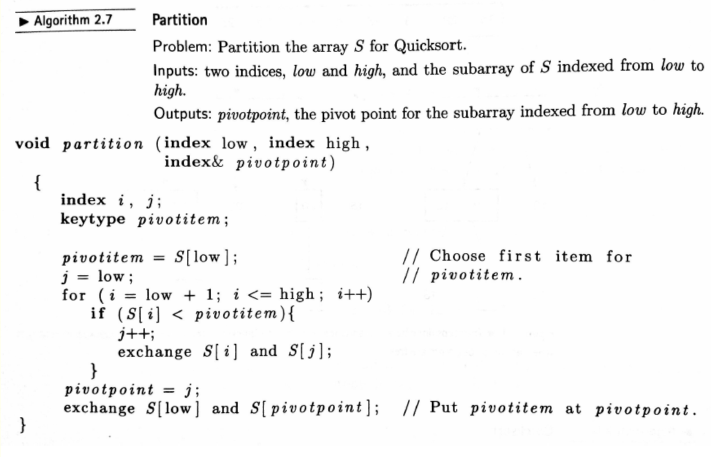
Problem: Sort n keys in nondecreasing order.

Inputs: positive integer n, array of keys S indexed from 1 to n.

Outputs: the array S containing the keys in nondecreasing order.

```
void quicksort (index low, index high)
{
  index pivotpoint;

  if (high > low){
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint - 1);
     quicksort(pivotpoint + 1, high);
  }
}
```

Cont.



▶ Algorithm 2.7 **Partition**

Problem: Partition the array S for Quicksort.

Inputs: two indices, *low* and *high*, and the subarray of S indexed from *low* to *high*.

Outputs: *pivotpoint*, the pivot point for the subarray indexed from *low* to *high*.

```
void partition (index low, index high,
                index& pivotpoint)
  {
     index i, j;
     keytype pivotitem;

     pivotitem = S[low];                  // Choose first item for
     j = low;                             // pivotitem.
     for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem){
           j++;
           exchange S[i] and S[j];
        }
     pivotpoint = j;
     exchange S[low] and S[pivotpoint];   // Put pivotitem at pivotpoint.
  }
```

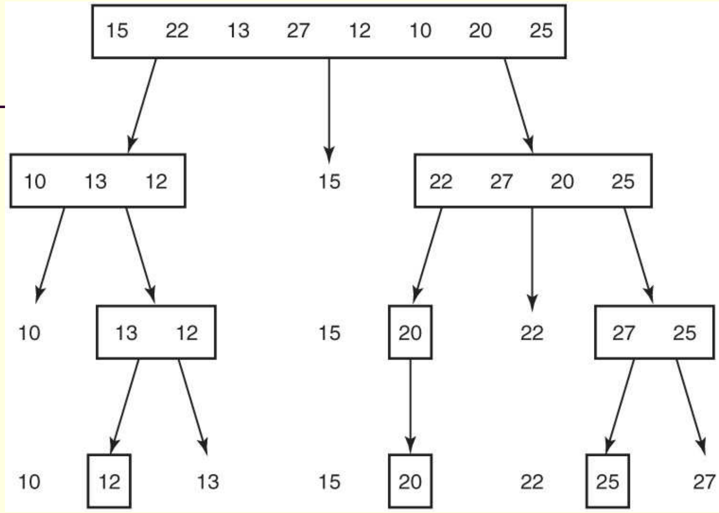


Figure 2.3: The steps done by a human when sorting with Quicksort. The subarrays are enclosed in rectangles whereas the pivot points are free.

Time complexity for quicksort

- What is $W(n)$ for quicksort?
- $W(n) \ = \ W(divide + conquer)$
- if the pivot always reaches the far right then total passes?
- $W(n) = (n-1) + (n-2) + \ ... + 1$
- $W(n) = \frac{n(n-1)}{2}$
- Big O would be $O(N^2)$

Cont.

- What is $B(n)$ for quicksort?
- $B(n) \ = \ W(divide + conquer)$
- if the pivot is always in the middle of the array
- $B(n) = (n-2^0) + (n-2^0-2^1) + \ ... + \left(n-2^0-2^1-\cdots-2^{\log_2 n}\right) //simple\ case$
- $B(n) = n - \left(2^0 + \mathbf{2^1} + \cdots + 2^{\log_2 n} - \mathbf{2^1}\right) + (n-1)\log_2 n$
- $= n - \left(2^{\log_2 n+1} - 1 - 2\right) + (n-1)\log_2 n$
- $= (n-1)\log_2 n - 2$
- Big O would be $O(n \log_2 n)$

Cont.

- What is $A(n)$ for quicksort?
- if the pivot is equally likely to exist anywhere in the array then expected value $E(n)$ for the first divide:
- $E^1(n) = \frac{1}{n-1} * (n-1) + \frac{1}{n-1} * (n-1) + \cdots = n - 1//same\ iterations$
- Following divides have additional 2^k pivots until $E^k(n) = 1$
- Maximum value of k is $log_2 n$ except for the worst case.
- Overall $A(n) = E^1(n) + E^2(n) + \cdots + E^k(n)$
- Probability of pivot being placed in a position to lead worse conditions is low and for simplicity it can be ignored.*

*see lecture capture for details.

Cont.

- Overall $A(n) = (n-2^0) + (n-2^0-2^1) + \ ... + (n-2^0-2^1-\cdots-2^{\log_2 n})$

$//iterate\ through\ all\ elements\ except\ pivot\ points$

- $A(n) = nlog_2 n - 2^{\log_2 n+1} = nlog_2 n - n - 1$
- Big O order is $O(nlog_2 n)$