Chapter 2

Divide and Conquer

Objectives

- Describe the divide-and-conquer approach to solving problems
- Apply the divide-and-conquer approach to solve a problem
- Determine complexity analysis of divide and conquer algorithms

2.1 Binary Search

- We present a **recursive** . The steps are

If x equals the middle item, quit. Otherwise:

1. Divide the array into two subarrays about half .
   a) If x is **smaller** than the **middle** item, choose the **left** subarray.
   b) If x is **larger** than the middle item, choose the **right** subarray.
2. Conquer (solve) the subarray : whether x is in that subarray.
   Unless the subarray is sufficiently small, use recursion to do this.
3. Obtain the solution to the array from the solution to the subarray.

Binary search

Example:

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

Algorithm 2.1

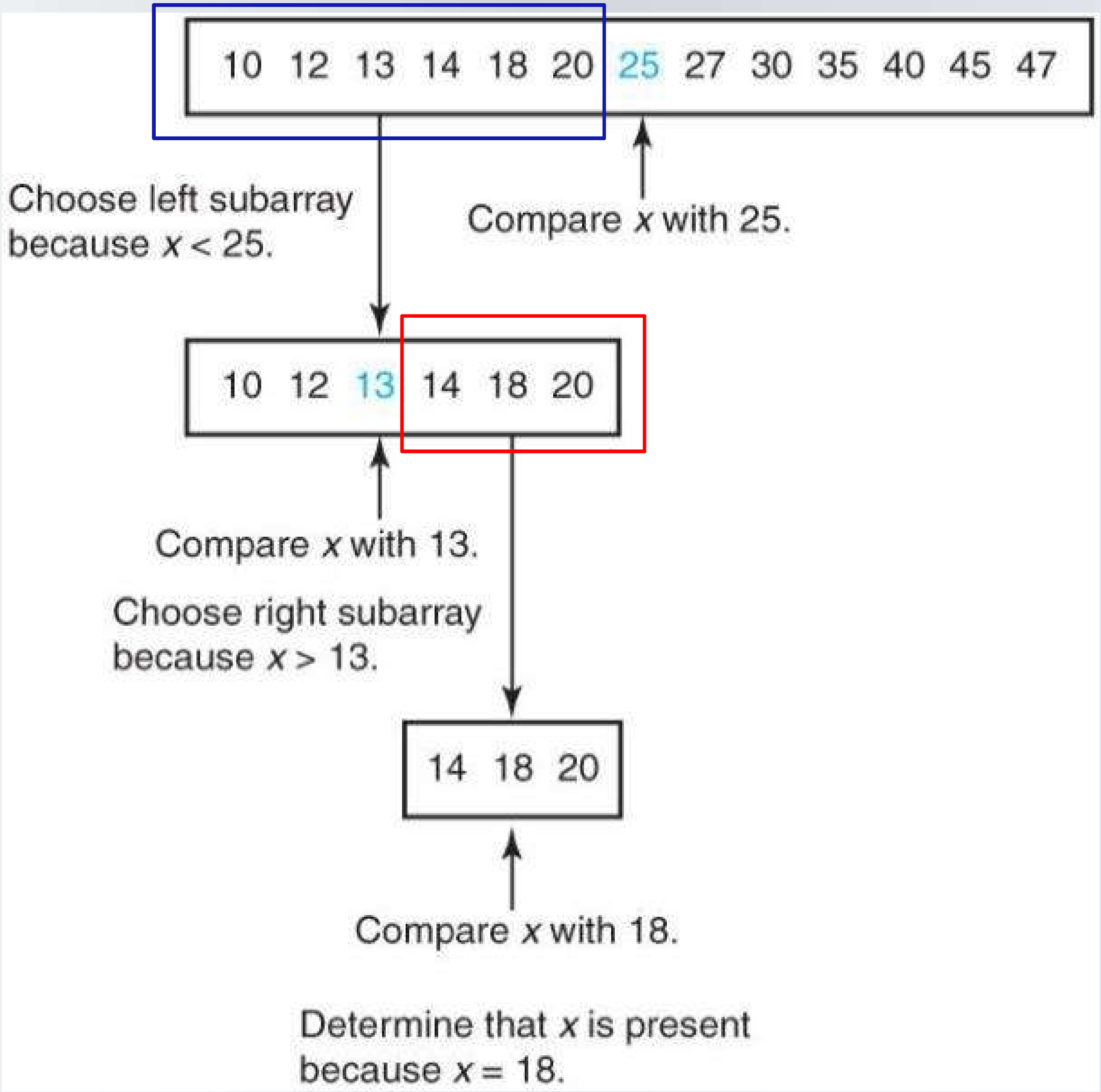
Binary Search (Recursive)

- Problem: ... Inputs: ... Outputs:, the location ...

```
void  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 )
       }
}
```

- n, S, and x are not parameters to function location. Because
  - they remain unchanged in each recursive call,
  - a new copy of any variable passed to the routine is made in each recursive call.
  - If a variable's value does not change, the copy is unnecessary.

n =13 elements, search x=**18**



- **Tail-recursion :** no operations are done after the recursive call
- When a routine calls another routine, it is necessary to save the first routine's pending results by pushing them onto the stack of activation records and so on.
- When control is returned to a calling routine, its activation record is popped from the stack .
- The number of **activation records** pushed onto the stack is determined by the **depth** reached in the recursive calls.
- For Binary Search, the stack reaches a depth that in the worst case is about ***lg n + 1.***

Algorithm 1.5

Binary Search

Problem: Determine whether x is in the sorted array S of n keys.

Inputs: positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.

Outputs: *location*, the location of x in S (0 if x is not in S).

```
void binsearch (int n,
                const keytype S[],
                keytype x,
                index& location)
{
   index low, high, mid;

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

Worst Case Complexity Analysis

- x is larger than all array items.
- n is a power of 2 and x > S[n]
- W(n) = the number of comparisons in the recursive call
- W(n) = W(n/2) + 1 for n>1 and n power of 2
- W(1) = 1

- Example B1 in Appendix B:
  - $W(n) = lg\ n + 1$
- n not a power of 2 (exercise)
  - $W(n) = \lfloor lg\ n \rfloor + 1 \in \theta(lg\ n)$

- **Algorithm B.1 -** Factorial
- Problem: Determine $n! = n * (n - 1)!$ *if* $n \geq 1.$

  $0! = 1$
- t_n : the number of multiplications done for a given value of n

  $t_n = t_{n-1} + 1$
- *initial condition :* no multiplications when $n = 0 \rightarrow t_0 = 0$

$t_1 = 1$

$t_2 = t_1 + 1 = 2$

$t_3 = 3$

…

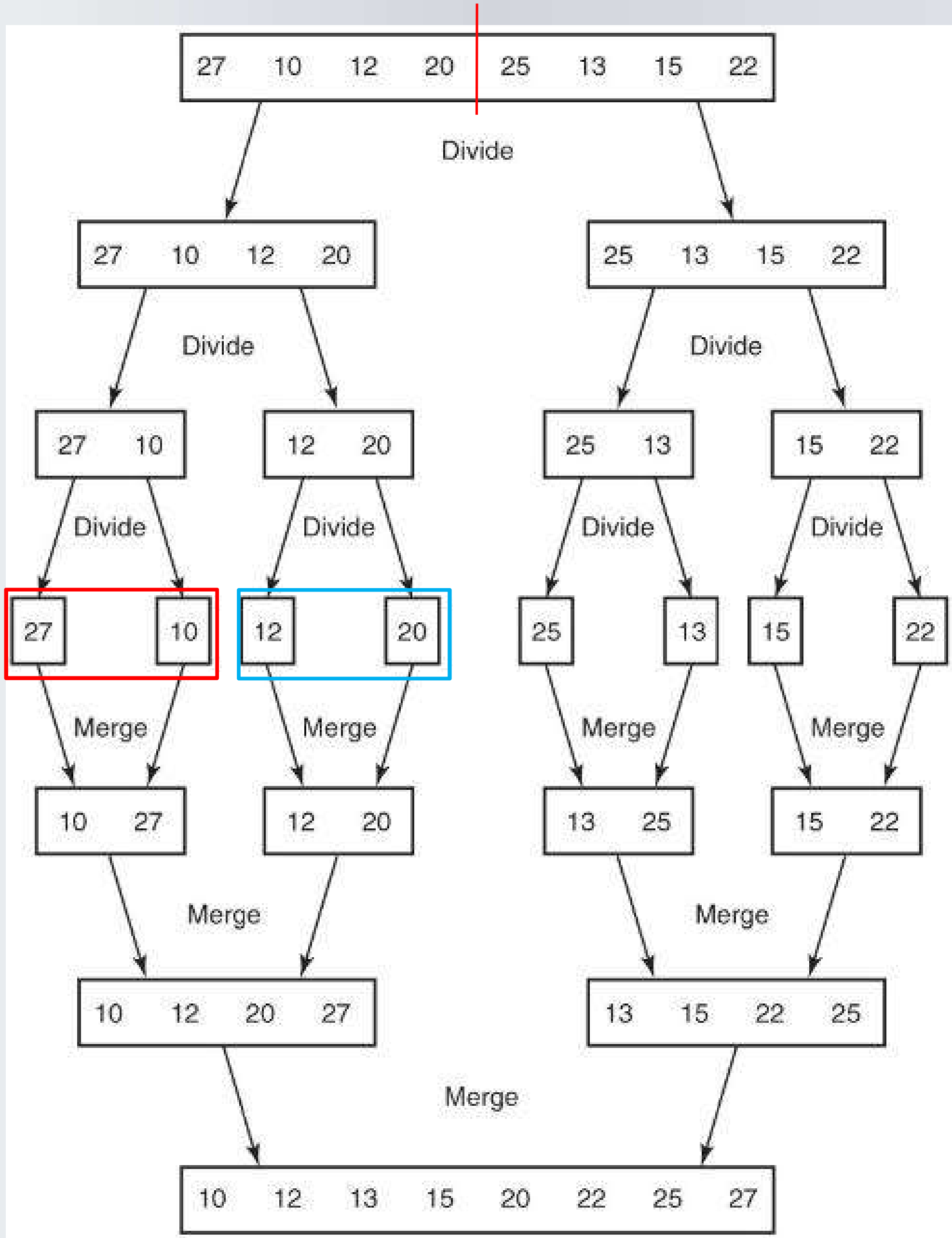
$t_n = n$

Proof by induction

2.2 Mergesort

- Sort an array S of size n (let n be a power of 2)
- *Divide* S into 2 sub-arrays of size n/2
- *Conquer* (solve) recursively sort each sub-array until array is sufficiently small (size 1)
- *Combine* merge the solutions to the sub-arrays into a single sorted array

Figure 2.2



27 10 12 20 25 13 15 22
Divide
27 10 12 20
25 13 15 22
Divide
Divide
27 10
12 20
25 13
15 22
Divide
Divide
Divide
Divide
27
10
12
20
25
13
15
22
Merge
Merge
Merge
Merge
10 27
12 20
13 25
15 22
Merge
Merge
10 12 20 27
13 15 22 25
Merge
10 12 13 15 20 22 25 27

Algorithm 2.2 - Mergesort

Problem: Sort *n* keys in nondecreasing sequence.
Inputs: +ve int *n*, array of keys *S* indexed from *1* to *n.*
Outputs: the sorted array *S* in nondec. order.

```
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);
  }
}
```

Algorithm 2.3 - Merge

- Merges the two arrays **U** and **V** :
- **Input size**
  - h the number of items in U
  - m the number of items in V
- Outputs: the array *S* containing the keys in nondecreasing order.

```
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];
}
```

k	U	V	S(Result)
1	**10** 12 20 27	**13** 15 22 25	10
2	10 **12** 20 27	**13** 15 22 25	10 12
3	10 12 **20** 27	**13** 15 22 25	10 12 13
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25
—	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 ← Final values

Worse-Case Time Complexity (Merge)

- The number of comparisons depends on both *h* and *m.*
- Basic operation: the comparison of ***U* [*i*]** with ***V* [*j*]**.
- Input size: ***h*** and ***m***,
- The worst case : when one of the indices— say, *i*- has reached its exit point *h + 1* whereas the other index *j* has reached *m, 1* less than its exit point.
- at which time the loop is exited because *i equals h + 1. Therefore,*

  W(h, m)= h+m-1

Worst-Case Time Complexity Analysis- Mergesort

- $W(n)$ = time to sort U + time to sort V + time to merge
- $W(n) = W(h) + W(m) + h+m-1$
- First analysis assumes n is a power of 2
  - $h = \lfloor n/2 \rfloor = n/2$
  - $m = n - h = n - n/2 = n/2$
  - $h + m = n/2 + n/2 = n$
- $W(n) = W(n/2) + W(n/2) + n - 1$

  $= 2W(n/2) + n-1$ for $n > 1$ and n a power of 2
- $W(1) = 0$

$$\boxed{\begin{array}{l} W(n) = 2W(n/2) + n-1 \text{ for } n > 1 \text{ and n a power of 2} \\ W(1) = 0 \end{array}}$$

- See Example B19 in Appendix B

Example B19 in Appendix B

$W(n) = (\log_2 n)n - (n - 1) \in \theta(n \lg n)$

- n not a power of 2
- $W(n)=W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n-1$
- From Theorem B4: (W(n) *eventually* nondecreasing, n lg n is smooth)
- $W(n) \in \Theta(n \lg n)$

EXERCISE

Solve

$T(n) = 2T(n/2) + n$, $T(1)=1$

$$T(n) = 2T(n/2) + n = 2[\,2T(n/2^2) + n/2] + n$$
$$= 2^2T(n/2^2) + 2n = 2^2\,[2T(n/2^3) + n/2^2] + 2n$$
$$= 2^3T(n/2^3) + 3n$$
$$\ldots$$
$$= 2^kT(n/2^k) + kn$$

If $n=2^k$ then we have

$$T(n) = nT(1) + (\log_2 n)n = n + n\log_2 n$$
$$= \Theta\,(n \log_2 n)$$

- An **in-place** sort is a sorting algorithm that does not use any **extra space** beyond that needed to store the input.
- Algorithm 2.2 is not an in-place sort (it uses *U and V*)
- At the top level, the sum of the numbers of items in these two arrays is *n.*
- However, it is possible to reduce the amount of extra space to only one array containing *n* items. This is accomplished by doing much of the manipulation on the input array *S.*

Algorithm 2.4 – Mergesort2

Problem: Sort *n* keys in nondecreasing sequence.
Inputs: +ve int *n,* array of keys *S* indexed from *1* to *n.*
Outputs: the sorted array *S* in nondec. order.

```
void mergesort2 (index low, index high)
{
  index mid;

  if (low < high) {
      mid = ⌊(low + high)/2⌋;
      mergesort2(low, mid);
      mergesort2(mid + 1, high);
      merge2(low, mid, high);
  }
}
```

Algorithm 2.5 - Merge 2

- Problem: Merge the two sorted subarrays of *S created in Mergesort 2.*
- Inputs: indices low, mid, and high, and the subarray of S indexed from low to high. The keys in array slots from low to mid are already sorted in nondecreasing order, as are the keys in array slots from mid + 1 to high.
- Outputs: the subarray of *S* indexed from low to high containing the keys in nondecreasing order.

```
void merge2 (index low, index mid, index high)
{
  index i, j, k;
  keytype U[low..high];    // A local array needed for the
                           // merging
  i = low; j = mid + 1; k = low;
  while (i ≤ mid && j ≤ high){
      if (S[i]<S[j]){
          U[k] = S[i]; i++; }

      else{
          U[k] = S[j]; j++; }

     k++;
  }
  if (i > mid)
     move S[j] through S[high] to U[k] through U[high];
  else
     move S[i] through S[mid] to U[k] through U[high];
  move U[low] through U[high] to S[low] through S[high];
}
```

2.4 Quicksort

- Array recursively divided into two partitions and recursively sorted
- Division based on a **pivot**
- pivot divides the array into two sub-arrays
- All items < pivot placed in sub-array before pivot
- All items >= pivot placed in sub-array after pivot

Example 2.3

- Suppose the array contains these numbers in sequence:

Pivot item

15 22 13 27 12 10 20 25

Pivot item

13 12 10 15 22 27 20 25

Sort the subarrays:

10 12 13 15 20 22 25 27

- **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

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

15 22 **13** 27 12 10 20 25

j i

swap S[2] and S[3]

15 13 **22** 27 12 10 20 25

j=2 I

swap S[3] and S[5]

15 13 **12** 27 22 10 20 25

j=3 i

15 13 **12** 10 22 27 20 25

j=4, pivot=j

Exchange S[low] and S[pivot]

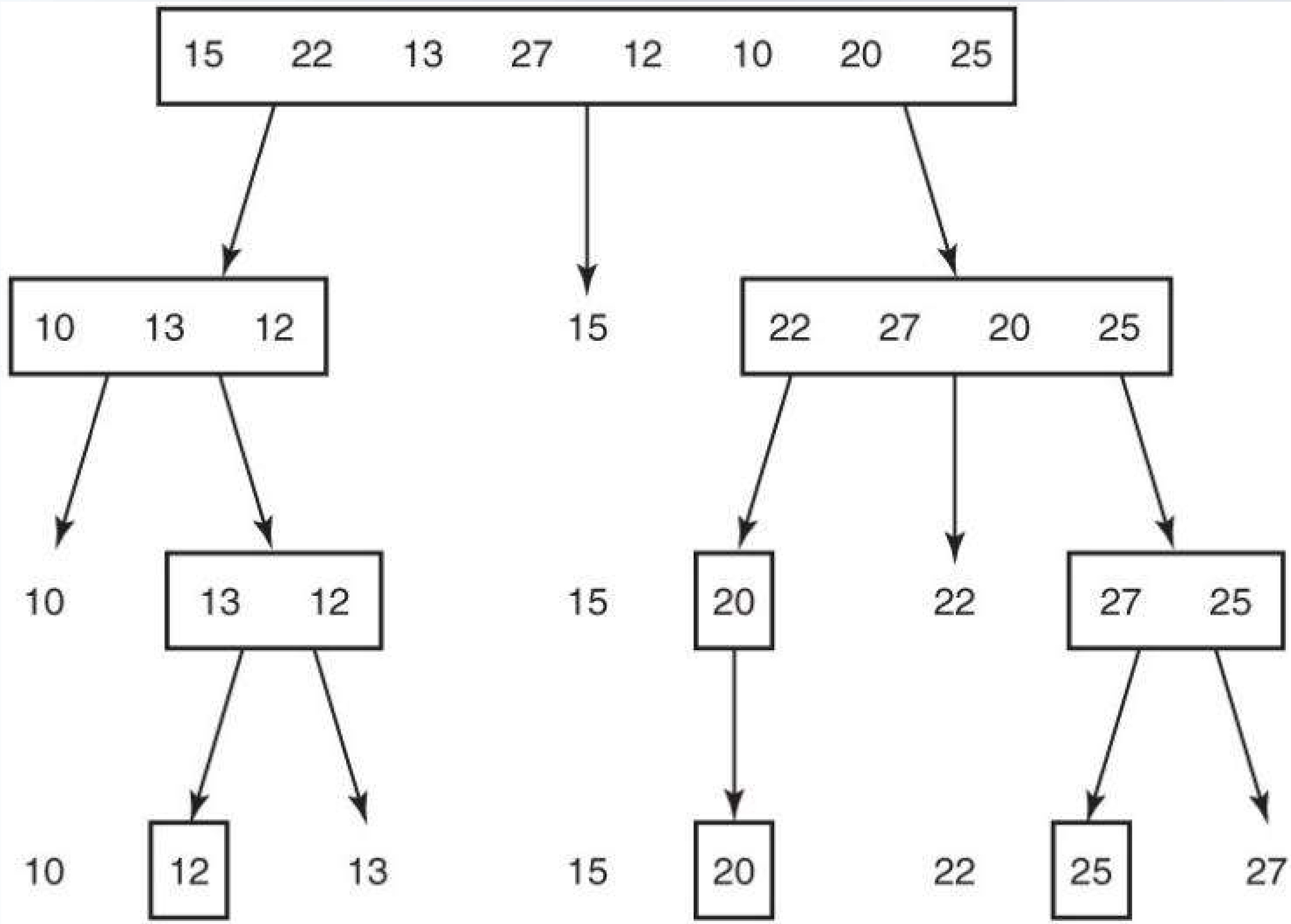
10 13 12 **15** 22 27 20 25

```
void partition (index low, index
high, index& pivot)
{
 index i,j; keytype  pivotitem;
   pivotitem = S[low];
   j=low;
   for (i=low+1; i<=high; i++)
         if (S[i] < pivotitem){
            j++;
            swap S[i] and S[j];
         }
   pivot= j;
   swap S[low] and S[pivot];
}
```

- **Algorithm 2.7 - Partition**
- Problem: Partition the array *S for Quicksort.*
- Inputs: two indices, *low and high,* and the subarray of S [low ,high].
- Outputs: *pivotpoint,* the pivot point for the subarray S [low ,high].

```
void partition (index low, index high, index& pivot)
{
 index i,j; keytype  pivotitem;
   pivotitem = S[low];
   j=low;
   for (i=low+1; i<=high; i++)
         if (S[i] < pivotitem){
            j++;
            swap S[i] and S[j];
         }
   pivot= j;
   swap S[low] and S[pivot];
}
```

Figure 2.3



15 22 13 27 12 10 20 25
10 13 12
15
22 27 20 25
10
13 12
15
20
22
27 25
10
12
13
15
20
22
25
27

Basic operation

- Comparison of S[i] with pivot item
- Input size n

Every-Case Complexity Analysis of Partition

- Input size: $n = high - low + 1$, the number of items in the subarray.
- Because every item except the first is compared,
- $T(n) = n - 1$

Worst-Case Complexity Analysis of Quicksort

- Array is repeatedly sorted into an empty sub-array which is less than the pivot and a sub-array of n-1 containing items greater than pivot
- If there are k keys in the current sub-array, k-1 key comparisons are executed

Worst-Case Complexity Analysis of Quicksort

- the worst case occurs if the array is already sorted
- T(n) is specified because analysis is for the every-case complexity for the class of instances already sorted in non-decreasing order

$T(n)$ = time to sort left sub-array + time to sort right sub-array + time to partition

$T(n) = T(0) + T(n-1) + n - 1$

$T(n) = T(n - 1) + n - 1$ for $n > 0$

$T(0) = 0$

From B16

$T(n) = n(n-1)/2$

Worst Case

- Use induction to show it is the worst case

  $W(n) <= n(n-1)/2$

Average-Case Time Complexity of Quicksort

- Value of pivotpoint is equally likely to be any of the numbers from 1 to n
- Average obtained is the average sorting time when every possible ordering is sorted the same number of times
- Let *p* be *the* value of *pivotpoint* returned by partition

$$A(n) = \sum_{p=1}^{n} \frac{1}{n} [A(p-1) + A(n-p)] + n - 1$$

$$A(n) = \frac{1}{n} \sum_{p=1}^{n} [A(p-1) + A(n-p)] + n - 1$$

$$A(n) = \frac{2}{n} \sum_{p=1}^{n} A(p-1) + n - 1$$

Multiplying by n we have

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1). \qquad (2.2)$$

Applying Equality 2.2 to $n-1$ gives

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2). \qquad (2.3)$$

Subtracting Equality 2.3 from Equality 2.2 yields

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1),$$

which simplifies to

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

If we let

$$a_n = \frac{A(n)}{n+1},$$

we have the recurrence

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad \text{for } n > 0$$
$$a_0 = 0.$$

Like the recurrence in Example B.22 in Appendix B, given by the approximate solution to this recurrence is

$$a_n \approx 2 \ln n,$$

which implies that

$$A(n) \approx (n+1)\, 2 \ln n = (n+1)\, 2 (\ln 2)(\lg n)$$
$$\approx 1.38 (n+1) \lg n \in \Theta(n \lg n).$$