Searching and Sorting

Prepared By: Sweta Dargad

Searching Algorithms

- Looking up a phone number, accessing a website and checking a word's definition in a dictionary all involve searching through large amounts of data.
- Searching algorithms all accomplish the same goal—finding an element that matches a given search key, if such an element does, in fact, exist.
- The major difference is the amount of *effort* they require to complete the search.
- One way to describe this *effort* is with Big O notation.
  - For searching and sorting algorithms, this is particularly dependent on the number of data elements.

Goal

- Understand linear search algorithm then discuss the algorithm's efficiency as measured by Big O notation.
- https://www.cs.usfca.edu/~galles/visualization/Search.html

- binary search algorithm, which is much more efficient but more complex to implement.

Sequential Searching

LINEAR_SEARCH(K,N,X):- This algorithm is used to search a particular element having the value X from an array K. The function returns the index of the vector element if the search is successful and return 0 otherwise.

1. [Initialize]

```
i <- 0
```

2. [Search the Vector]

```
Repeat While I < N && K[i] != X
        i <- i + 1
```

3. [Successful Search]

```
if  i < N
        then write ("Successful Search")
        return (i)
else
        write ("Unsuccessful Search")
        return (0)
```

4. [Finish]

```
Exit
```

Linear Search - Example

- Array `num` contains:

17	23	5	11	2	29	3

- Searching for the the value `11`, linear search examines `17, 23, 5,` and `11 and returns 11 is present at index 3.`

- Searching for the the value `7`, linear search examines `17, 23, 5, 11, 2, 29,` and `3 which is not present in list so it will return 0`

Linear Search - Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order
- Disadvantages:
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

Linear Search

`linearSearch` compares each element of an `array` with a *search key*

- Because the array is not in any particular order, it's just as likely that the search key will be found in the first element as the last.
- On average, therefore, the program must compare the search key with *half* of the `array`'s elements.
- To determine that a value is *not* in the `array`, the program must compare the search key to *every* `array` element.
- Linear search works well for *small* or *unsorted* arrays.
- However, for large arrays, linear searching is inefficient.
- If the array is *sorted* (e.g., its elements are in ascending order), you can use the high-speed binary search technique.

Efficiency

Big O: Constant Runtime

- Suppose an algorithm simply tests whether the first element of an `array` is equal to the second element.
- If the `array` has 10 elements, this algorithm requires only one comparison.
- If the `array` has 1000 elements, the algorithm still requires only one comparison.
- In fact, the algorithm is *independent* of the number of `array` elements.
- This algorithm is said to have a constant runtime, which is represented in Big O notation as O(1).
- An algorithm that is O(1) does not necessarily require only one comparison.
- O(1) just means that the number of comparisons is *constant*—it does *not* grow as the size of the `array` increases.
- An algorithm that tests whether the first element of an `array` is equal to any of the next three elements will always require three comparisons, but in Big O notation it's still considered O(1).
- O(1) is often pronounced "on the order of 1" or more simply "order 1."

Efficiency

- ***Big O: Linear Runtime***
- An algorithm that tests whether the first element of an `array` is equal to any of the other elements of the `array` requires at most $n - 1$ comparisons, where n is the number of elements in the `array`.
- If the `array` has 10 elements, the algorithm requires up to nine comparisons.
- If the `array` has 1000 elements, the algorithm requires up to 999 comparisons.
- As n grows larger, the n part of the expression $n - 1$ "dominates," and subtracting one becomes inconsequential.
- Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows.

Efficiency of Linear Search

- An algorithm that requires a total of $n - 1$ comparisons is said to be O(n) and is referred to as having a linear runtime.
- O(n) is often pronounced "on the order of n" or more simply "order n."

Big O: Quadratic Runtime

- Now suppose you have an algorithm that tests whether any element of an `array` is duplicated elsewhere in the `array`.
- The first element must be compared with every other element in the `array`.
- The second element must be compared with every other element except the first (it was already compared to the first).
- The third element then must be compared with every other element except the first two.
- In the end, this algorithm will end up making $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ *comparisons*.
- As n increases, the n^2 term dominates and the n term becomes inconsequential.
- Again, Big O notation highlights the n^2 term, leaving $n^2/2$.

- Big O is concerned with how an algorithm's runtime grows in relation to the *number of items processed*.
- Suppose an algorithm requires n^2 comparisons.
- With four elements, the algorithm will require 16 comparisons; with eight elements, 64 comparisons.
- With this algorithm, *doubling* the number of elements *quadruples* the number of comparisons.
- Consider a similar algorithm requiring $n^2/2$ comparisons.
- With four elements, the algorithm will require eight comparisons; with eight elements, 32 comparisons.
- Again, doubling the number of elements quadruples the number of comparisons.
- Both of these algorithms grow as *the square of n*, so Big O ignores the constant, and both algorithms are considered to be $O(n^2)$, which is referred to as quadratic runtime and pronounced "on the order of n-squared" or more simply "order *n*-squared."

O(n2) Performance

- When n is small, $O(n^2)$ algorithms will not noticeably affect performance.
- As n grows, you'll start to notice the performance degradation.
- An $O(n^2)$ algorithm running on a million-element `array` would require a trillion "operations" (where each could actually require several machine instructions to execute).
  - This could require hours to execute.
- A billion-element `array` would require a quintillion operations, a number so large that the algorithm could take decades! Unfortunately, $O(n^2)$ algorithms tend to be easy to write.

- In this chapter, you'll see algorithms with more favorable Big O measures.
- These efficient algorithms often take a bit more cleverness and effort to create, but their superior performance can be worth the extra effort, especially as *n* gets large and as algorithms are compounded into larger programs.

Efficiency of Linear Search (cont.)

Linear Search's Runtime

- The *linear search* algorithm runs in *O(n)* time.
- The worst case in this algorithm is that *every* element must be checked to determine whether the search key is in the `array`.
- If the `array`'s size doubles, the number of comparisons that the algorithm must perform also *doubles*.
- Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the `array`.
- But we seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the `array`.
- If a program needs to perform many searches on large `array`s, it may be better to implement a different, more efficient algorithm, such as the *binary search* which we present in the next section.

Performance Tip 20.1

Sometimes the simplest algorithms perform poorly. Their virtue is that they're easy to program, test and debug. Sometimes more complex algorithms are required to maximize performance.

Binary Search

Requires array elements to be in order

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

Binary Searching

BINARY_SEARCH(K,N,X):- This algorithm is used to search a particular element having the value X from an array K. The variable LOW, MIDDLE and HIGH denote the lower, middle and upper limits of the search interval, respectively. The function returns the index of the vector element if the search is successful and return 0 otherwise.

1. [Initialize]

LOW <- 0

HIGH <- N-1

2. [Perform Search]

Repeat thru step 4 While LOW <= HIGH

3. [Obtain index of midpoint of interval]

MIDDLE <- (LOW+HIGH)/2

4. [Compare]

if X < K[MIDDLE]

then HIGH <- MIDDLE – 1

else if X > K[MIDDLE]

then LOW <- MIDDLE + 1

else

then write (“Successful Search”)

return (MIDDLE)

Continue….

5. [Unsuccessful Search]

write (“Unsuccessful Search”)

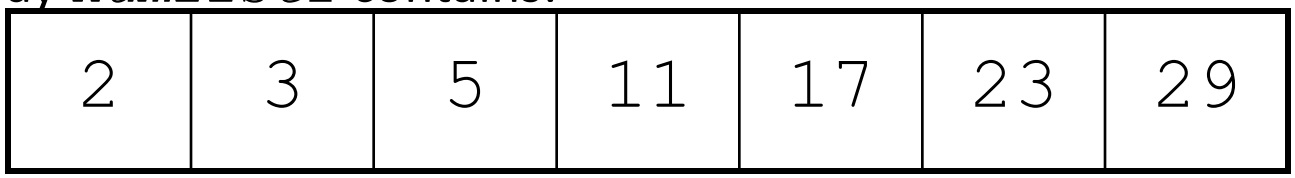
return (0)

6. [Finish]

Exit

Binary Search - Example

- Array `numlist2` contains:



- Index 0 1 2 3 4 5 6

- Searching for the value `11`, binary search finds middle index(0+6/2=3)- examines `11` and stops
- Searching for the the value `11`, linear search examines `2,3,5,11,` and stops

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For array of N elements, performs at most $\boldsymbol{log_2 N}$ comparisons
- Disadvantages:
  - Requires that array elements be sorted

Binary Search

- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the `array` first be sorted.
- This is only worthwhile when the array, once sorted, will be searched a great many times—or when the searching application has *stringent* performance requirements.
- The first iteration of this algorithm tests the *middle* `array` element.
- If this matches the search key, the algorithm ends.

Binary Search (cont.)

- Assuming the `array` is sorted in *ascending* order, then if the search key is *less* than the middle element, the search key cannot match any element in the `array`'s second half so the algorithm continues with only the first *half* (i.e., the first element up to, but *not* including, the middle element).
- If the search key is *greater* than the middle element, the search key cannot match any element in the `array`'s first half so the algorithm continues with only the second *half* (i.e., the element *after* the middle element through the last element).
- Each iteration tests the *middle value* of the `array`'s remaining elements.
- If the element does not match the search key, the algorithm eliminates half of the remaining elements.
- The algorithm ends either by finding an element that matches the search key or by reducing the sub-`array` to zero size.

Binary Search (cont.)

Efficiency of Binary Search

- In the worst-case scenario, searching a sorted `array` of 1023 elements will take only 10 comparisons when using a binary search.
- Repeatedly dividing 1023 by 2 (because, after each comparison, we can eliminate from consideration *half* of the remaining elements) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0.
- The number 1023 ($2^{10} – 1$) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test.

Binary Search (cont.)

- Dividing by 2 is equivalent to one comparison in the binary search algorithm.
- Thus, an `array` of 1,048,575 ($2^{20} - 1$) elements takes a maximum of 20 comparisons to find the key, and an `array` of approximately one billion elements takes a maximum of 30 comparisons to find the key.
- This is a *tremendous* performance improvement over the linear search.
- For a one-billion-element `array`, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search!
- The maximum number of comparisons needed for the binary search of any sorted `array` is the exponent of the first power of 2 greater than the number of elements in the `array`, which is represented as $\log_2 n$.

Binary Search (cont.)

- *All logarithms grow at roughly the same rate*, so in Big O notation the base can be omitted.
- This results in a Big O of O(log *n*) for a binary search, which is also known as logarithmic runtime and pronounced "on the order of log n" or more simply "order log *n*."

http://anim.ide.sk/sorting_algorithms_1.php

http://www.algoanim.ide.sk/

Sorting Algorithms

- Sorting data (i.e., placing the data into some particular order, such as ascending or descending) is one of the most important computing applications.
- Your algorithm choice affects only the algorithm's runtime and memory use.
- In each case, we examine the efficiency of the algorithms using Big O notation.

Example

http://www.algoanim.ide.sk/index.php?page=showanim&id=78

Bubble Sort

BUBBLE_SORT(K,N):- This algorithm is used to sort all N elements in ascending order. N is integer type variable indicate there are N elements in vector K. The variable PASS and LAST denote the pass counter and position of the last element, respectively. EXCHS is used to count the number of exchange made on any pass. I denoted index elements.

1. [Initialize]

LAST <- N-1

2. [loop on pass index]

Repeat thru step 5 for pass = 1,2,…,N-1

3. [Initialize exchanges counter for this pass]

EXCHS <- 0

4. [Perform pair wise comparisons on unsorted elements]

Repeat for I = 0,1, 2,….., LAST - 1

if K[I] > K[I + 1]

then K[I] < ----> K[I + 1] // swap K[i] with K[i+1]

EXCHS <- EXCHS + 1 // for every swap increment EXCHS

5. [Were any exchanges made on this pass ?]

if EXCHS = 0

then Return

else

LAST < - LAST - 1

6**.[finish]**

Exit

Bubble Sort

Array **a** contains **n** elements to be sorted.

```
for (i=0; i<n-1; i++) {
  for (j=0; j<n-1-i; j++)
      if (a[j+1] < a[j]) {  /* compare neighbors */
            tmp = a[j]; /* swap a[j] and a[j+1] */
            a[j] = a[j+1];
            a[j+1] = tmp;
  }
}
```

20.3.1 Insertion Sort

Insertion Sort Algorithm

- The algorithm's first iteration takes the `array`'s second element and, if it's less than the first element, swaps it with the first element (i.e., the algorithm *inserts* the second element in front of the first element).
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
- At the *i*th iteration of this algorithm, the first *i* elements in the original `array` will be sorted.

Example

- http://www.algoanim.ide.sk/index.php?page=showanim&id=79

Insertion Sort

First Iteration

- initialize the `array` named `data` with the following values:

```
34   56   4   10   77   51   93   30   5   52
```

- The function first looks at `items[0]` and `items[1]`, whose values are `34` and `56`, respectively.
- These two elements are already in order, so the algorithm continues—if they were out of order, the algorithm would swap them.

Insertion Sort

Second Iteration

- In the second iteration, the algorithm looks at the value of `items[2]` (that is, `4`).
- This value is less than `56`, so the algorithm stores `4` in a temporary variable and moves `56` one element to the right.
- The algorithm then determines that `4` is less than `34`, so it moves `34` one element to the right.
- At this point, the algorithm has reached the beginning of the `array`, so it places `4` in `items[0]`.
- The array now is

```
4   34   56   10   77   51   93   30   5   52
```

20.3.1 Insertion Sort

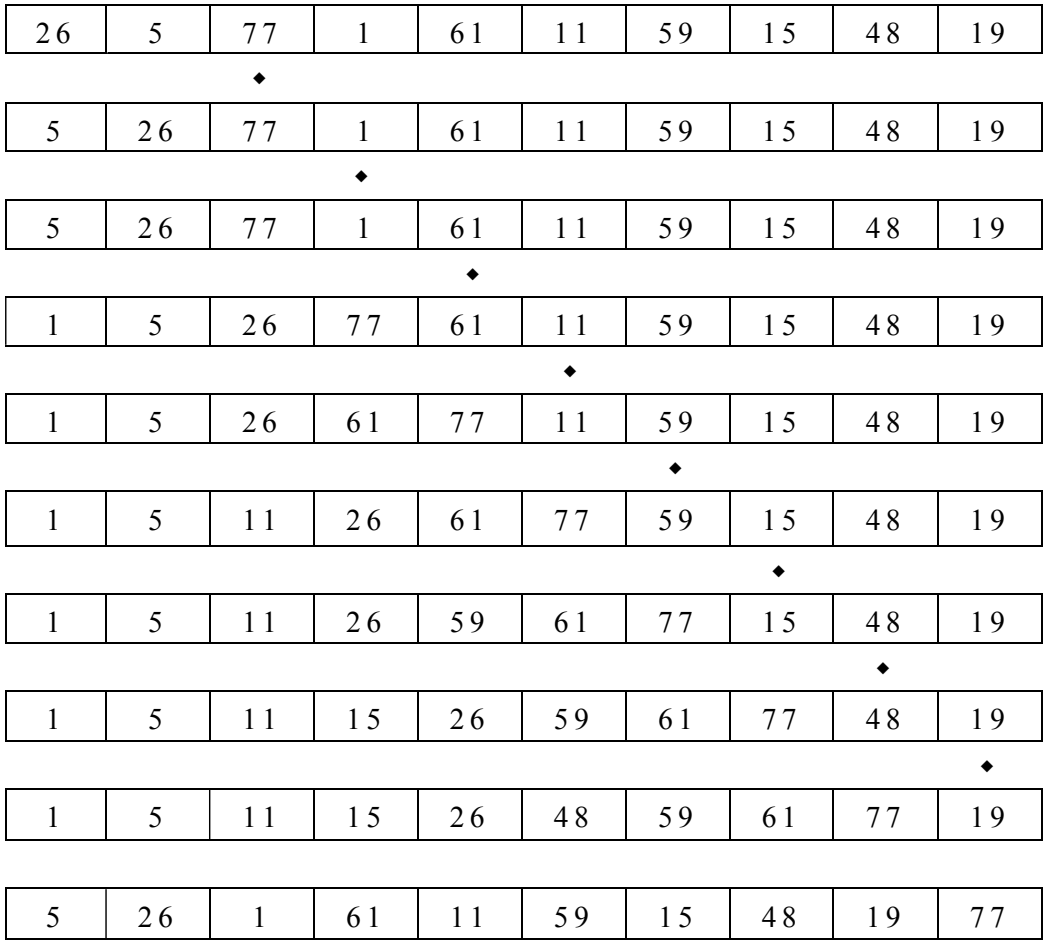
Third Iteration and Beyond

- In the third iteration, the algorithm places the value of `items[3]` (that is, `10`) in the correct location with respect to the first four array elements.
- The algorithm compares `10` to `56` and moves `56` one element to the right because it's larger than `10`.
- Next, the algorithm compares `10` to `34`, moving `34` right one element.
- When the algorithm compares `10` to `4`, it observes that `10` is larger than `4` and places `10` in `items[1]`.
- The array now is

```
4    10    34    56    77    51    93    30    5    52
```

- Using this algorithm, after the *i*th iteration, the first *i* + 1 array elements are sorted.
- They may not be in their final locations, however, because the algorithm might encounter smaller values later in the `array`.

Insertion Sort



Insertion Sort

INSERTION_SORT(K,N):- This algorithm is used to sort all N elements in ascending order. N is integer type variable indicate there are N elements in vector K. The variable TEMP denotes the temporary variable. I and J denotes index elements.

1. [loop on index]

Repeat thru step 4 for I = 1,…,N-1

2. [Initialize variable]

TEMP <- K[I]

J <- I -1

3. [Perform comparisons]

Repeat while J>=0 && TEMP < K[J]

K[J+1] <- K[J]

J <- J – 1

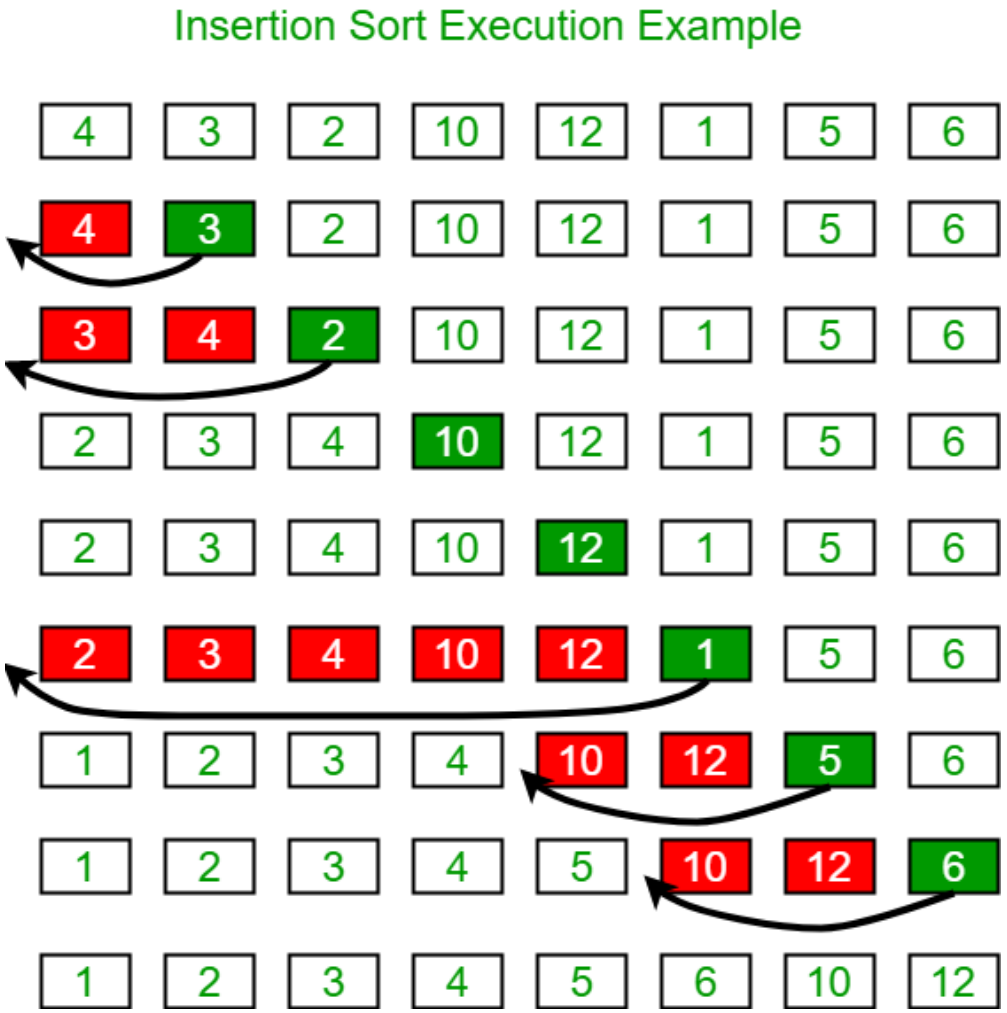
4. [Place value in proper position]

K[J+1] <- TEMP

5. [Finish]

Exit

Example


Insertion Sort Execution Example
4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12

Insertion Sort

```
void insertionSort(int K[ ], int n)
{
    int i, temp, j;
    for (i = 1; i < n; i++) {
     temp = K[i];
        j = i - 1;
              /* Move elements of K[0..i-1], that are greater than key,
to one position ahead of their current position */
  while (j >= 0 && K[j] > temp )
 {
            K[j + 1] = K[j];
            j = j - 1;
        }
        K[j + 1] = temp;
    }
}
```

Insertion Sort: Example

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Insertion Sort - Analysis

- Pass *p* involves at most *p* comparisons
- Total comparisons = $\sum i$; i = [1, n-1]
- = $O(n^2)$

Insertion Sort - Analysis

- Worst Case ?
  - Reverse sorted list
  - Max possible number of comparisons
  - $O(n^2)$
- Best Case ?
  - Sorted input
  - 1 comparison in each pass
  - O(n)

20.3.2 Selection Sort (cont.)

Selection Sort Algorithm

- The algorithm's first iteration of the algorithm selects the smallest element value and swaps it with the first element's value.
  - The second iteration selects the second-smallest element value (which is the smallest of the remaining elements) and swaps it with the second element's value.
  - The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element's value, leaving the largest value in the last element.
  - After the i^{th} iteration, the smallest i values will be sorted into increasing order in the first `array` elements.

Example

- http://www.algoanim.ide.sk/index.php?page=showanim&id=80

20.3.2 Selection Sort (cont.)

First Iteration

declare and initialize the array named data with the following values:

```
34   56   4   10   77   51   93   30   5   52
```

- The selection sort first determines the smallest value (4) in the array, which is in index 2.
- The algorithm swaps 4 with the value in element 0 (34), resulting in

```
4   56   34   10   77   51   93   30   5   52
```

20.3.2 Selection Sort (cont.)

Second Iteration

- The algorithm then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in element 8.
- The program swaps the 5 with the 56 in element 1, resulting in

```
4    5    34    10    77    51    93    30    56    52
```

20.3.2 Selection Sort (cont.)

Third Iteration

- On the third iteration, the program determines the next smallest value, 10, and swaps it with the value in element 2 (34).

```
4    5    10    34    77    51    93    30    56    52
```

- The process continues until the array is fully sorted.

```
4    5    10    30    34    51    52    56    77    93
```

- After the first iteration, the *smallest* element is in the *first* position; after the second iteration, the *two smallest* elements are in order in the *first two* positions and so on.

20.3.2 Selection Sort (cont.)

- The selection sort algorithm iterates $n - 1$ times, each time swapping the smallest remaining element into its sorted position.
- Locating the smallest remaining element requires $n - 1$ comparisons during the first iteration, $n - 2$ during the second iteration, then $n - 3$, … , 3, 2, 1.
- This results in a total of $n(n - 1)/2$ or $(n^2 - n)/2$ comparisons.
- In Big O notation, smaller terms drop out and constants are ignored, leaving a Big O of $O(n^2)$.

Example

- https://www.gatevidyalay.com/tag/selection-sort-algorithm-pdf/

Selection Sort

SELECTION_SORT(K,N):- This algorithm is used to sort all N elements in ascending order. N is integer type variable indicate there are N elements in vector K. The variable PASS denoted the pass index and the position of the first elements in the vector. MIN_INDEX denoted the position of the smallest element. I denoted index elements.

1. [loop on pass index]

Repeat thru step 4 for pass = 1,2,…,N-1

2. [Initialize minimum index]

MIN_INDEX <- PASS

3. [Make a pass and obtain element with smallest value]

Repeat for I = PASS+1, PASS+2,….., N

if K[I] < K[MIN_INDEX]

then MIN_INDEX <- I

4. [Exchange elements]

if MIN_INDEX ≠ PASS

then K[PASS] <——> K[MIN_INDEX]

5.[finish] Exit

Merge Sort (A Recursive Implementation)

- Merge sort is an efficient sorting algorithm but is conceptually *more complex* than insertion sort and selection sort.
- The merge sort algorithm sorts an `array` by splitting it into two equal-sized sub-`array`s, sorting each sub-`array` then *merging* them into one larger `array`.
- Merge sort performs the merge by looking at each sub-`array`s first element, which is also the smallest element in that sub-`array`.
- Merge sort takes the smallest of these and places it in the first element of merged, sorted `array`.
- If there are still elements in the sub-`array`s, merge sort looks at the second element in that sub-`array` (which is now the smallest element remaining) and compares it to the first element in the other sub-`array`.

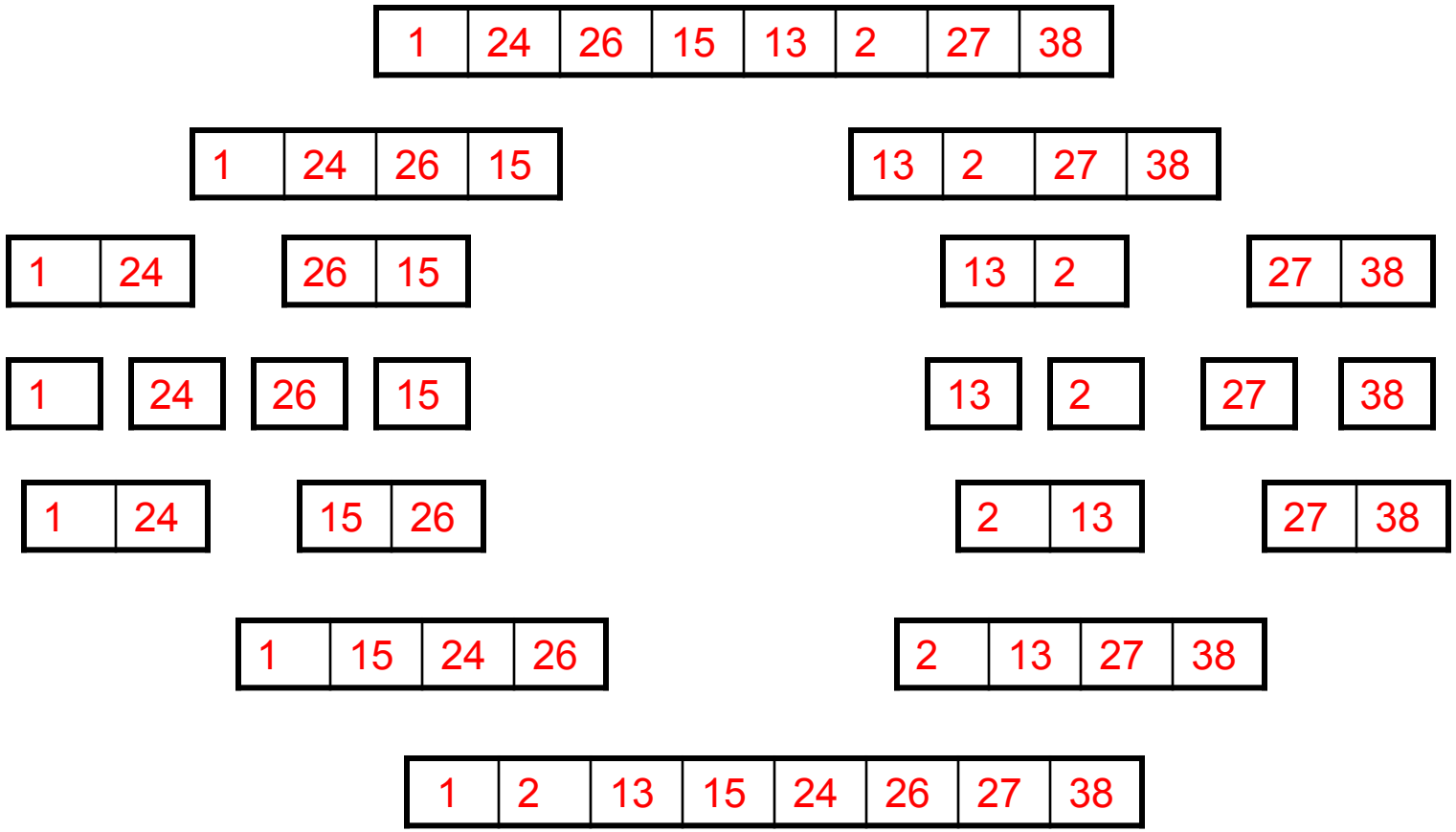
Merge Sort (A Recursive Implementation) (cont.)

- Merge sort continues this process until the merged `array` is filled.
- Once a sub-`array` has no more elements, the merge copies the other array's remaining elements into the merged array.

Example:

http://www.algoanim.ide.sk/index.php?page=showanim&id=75

Mergesort Example



http://www.algoanim.ide.sk/index.php?page=showanim&id=118

Merge 2 arrays

SIMPLE_MERGE(K,FIRST, SECOND, THIRD):- In this algorithm given two ordered subtables stored in a vector K. FIRST through SECOND – 1 elements and SECOND through THIRD elements represent the first and second tables respectively. TEMP is a temporary array. I and J denote the cursor associated with the first and second subtables, respectively. L is index variable.

1. [Initialize]

```
I <- FIRST
J <- SECOND
L <- 0
```

2. [Compare corresponding elements and output the smallest]

```
Repeat while I < SECOND and J <= THIRD
    if K[I] <= K[J]
    then L <- L + 1
         TEMP[L] <- K[I]
         I <- I + 1
    else
         L <- L + 1
         TEMP [L] <- K[J]
         J <- J + 1
```

3. [Copy the remaining unprocessed elements in output area]

```
if   I  >= SECOND
then   repeat while J <= THIRD
                    L <- L +1
                    TEMP [L] <- K[J]
                    J <- J +1
else   repeat while  I < SECOND
                    L <- L +1
                    TEMP[L] <- K[I]
                    I <- I +1
```

4. [Copy elements in temporary vector into original area]

```
Repeat for I = 1, 2,….., L
  K[FIRST -1 +I] <- TEMP[I]
```

5. [Finished]

exit

MergeSort(arr[], low, high)
If high > low

1. Find the middle point to divide the array into two halves:

mid = (low+high)/2

2. Call mergeSort for first half:

Call mergeSort(arr, low, mid)

3. Call mergeSort for second half:

Call mergeSort(arr, mid+1, high)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, low, mid+1, high)

mergeSort

```
void merge_sort(int k[], int low, int high)
{
        int size, mid;
        size = high-low +1;
        if(size<=1)
                return;
        mid = (low+high)/2;
        merge_sort(k,low,mid);
        merge_sort(k,mid+1,high);
        simple_merge(k,low,mid+1,high);
}
```

```
void simple_merge(int k[], int first, int sec, int third)
{
        int temp[80], i = first, j = sec,l= first;

        while((i<sec) && (j<=third))
        {
            if(k[i]<=k[j])
                temp[l] = k[i++] ;
            else
                temp[l] = k[j++] ;
            l++;
        }
```

```
if(i<sec)
        for(;i<sec;i++)
            temp[l++] = k[i];
    else
        for(;j<=third;j++)
            temp[l++]=k[j];

    for(i=first;i<=third;i++)
            k[i]=temp[i];
}
```

```
Unsorted array:
 30 47 22 67 79 18 60 78 26 54

split:    30 47 22 67 79 18 60 78 26 54
          30 47 22 67 79
                         18 60 78 26 54

split:    30 47 22 67 79
          30 47 22
                   67 79

split:    30 47 22
          30 47
                22

split:    30 47
          30
             47

merge:    30
             47
          30 47

merge:    30 47
                22
          22 30 47
```

Fig. 20.6 | Sorting an array into ascending order with merge sort. (Part 8 of 10.)

```
split:              67 79
                    67
                       79


merge:              67
                       79
                    67 79

merge:    22 30 47
                    67 79
          22 30 47 67 79

split:                     18 60 78 26 54
                           18 60 78
                                    26 54

split:                     18 60 78
                           18 60
                                 78

split:                     18 60
                           18
                              60
```

Fig. 20.6 | Sorting an array into ascending order with merge sort. (Part 9 of 10.)

```
merge:                    18
                             60
                          18 60

merge:                    18 60
                                78
                          18 60 78

split:                                26 54
                                      26
                                         54

merge:                                26
                                         54
                                      26 54

merge:                    18 60 78
                                      26 54
                          18 26 54 60 78

merge:    22 30 47 67 79
                          18 26 54 60 78
          18 22 26 30 47 54 60 67 78 79

Sorted array:
 18 22 26 30 47 54 60 67 78 79
```

Fig. 20.6 | Sorting an array into ascending order with merge sort. (Part 10 of

Efficiency of Merge Sort

- Merge sort is a far more efficient algorithm than either insertion sort or selection sort—although that may be difficult to believe when looking at the busy output
- Consider the first (nonrecursive) call to function `mergeSort`
- This results in two recursive calls to function `mergeSort` with `sub-array`s that are each approximately half the original `array`'s size, and a single call to function `merge`.
- The call to `merge` requires, at worst, $n - 1$ comparisons to fill the original `array`, which is $O(n)$.
- The two calls to function `mergeSort` result in four more recursive calls to function `mergeSort`—each with a sub-`array` approximately one-quarter the size of the original `array`—and two calls to function `merge`.
- These two calls to function `merge` each require, at worst, $n/2 - 1$ comparisons, for a total number of comparisons of $O(n)$.

Efficiency of Merge Sort

- This process continues, each call to `mergeSort` generating two additional calls to `mergeSort` and a call to `merge`, until the algorithm has split the array into one-element sub-`array`s.
- At each level, $O(n)$ comparisons are required to merge the sub-`array`s.
- Each level splits the size of the `array`s in half, so doubling the size of the array requires one more level.
- Quadrupling the size of the array requires two more levels.
- This pattern is logarithmic and results in $\log_2 n$ levels.
- This results in a total efficiency of $O(n \log n)$.

Quick Sort

- As the name implies, it is quick, and it is the algorithm generally preferred for sorting
- Another divide-and-conquer algorithm

- **Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.**
- **A large array is partitioned into two arrays one of which holds values smaller than specified value say pivot based on which the partition is made and another array holds values greater than pivot value. The quick sort partitions an array and then calls itself recursively twice to sort the resulting two subarray.**
- **This algorithm is quite efficient for large sized data sets as its average and worst case complexity are of Onlogn where n are no. of items.**

Versions of Quick Sort

- There are many different versions of quickSort that pick pivot in different ways.
- Always pick first element as pivot.
- Always pick last element as pivot
- Pick a random element as pivot.
- Pick median as pivot.
- The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Example:

- http://www.algoanim.ide.sk/index.php?page=showanim&id=117

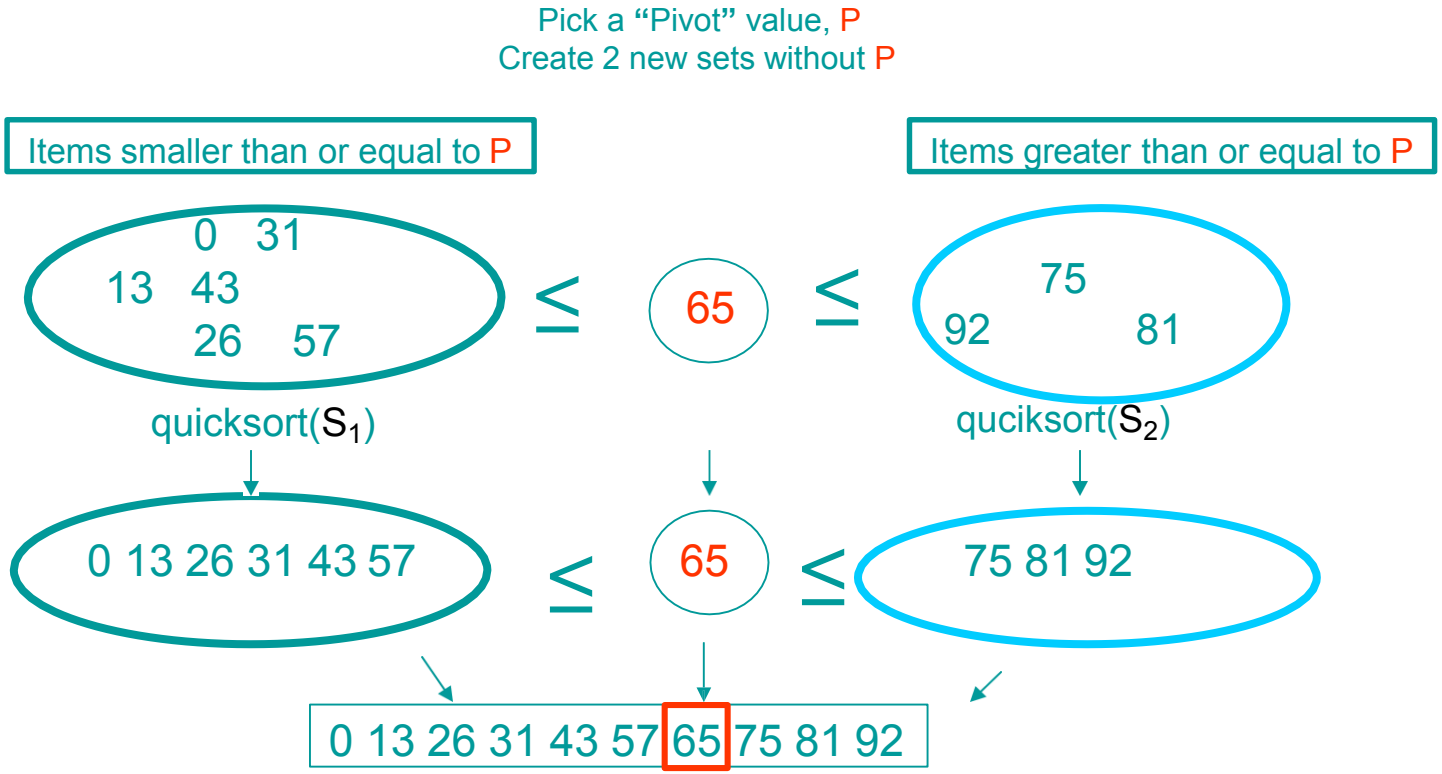
- Solved Example
- file:///E:/Sweta/Data%20Structure%202020-2021%20odd%20sem/Docs/quick-sort-complete-example.pdf

Basic Ideas

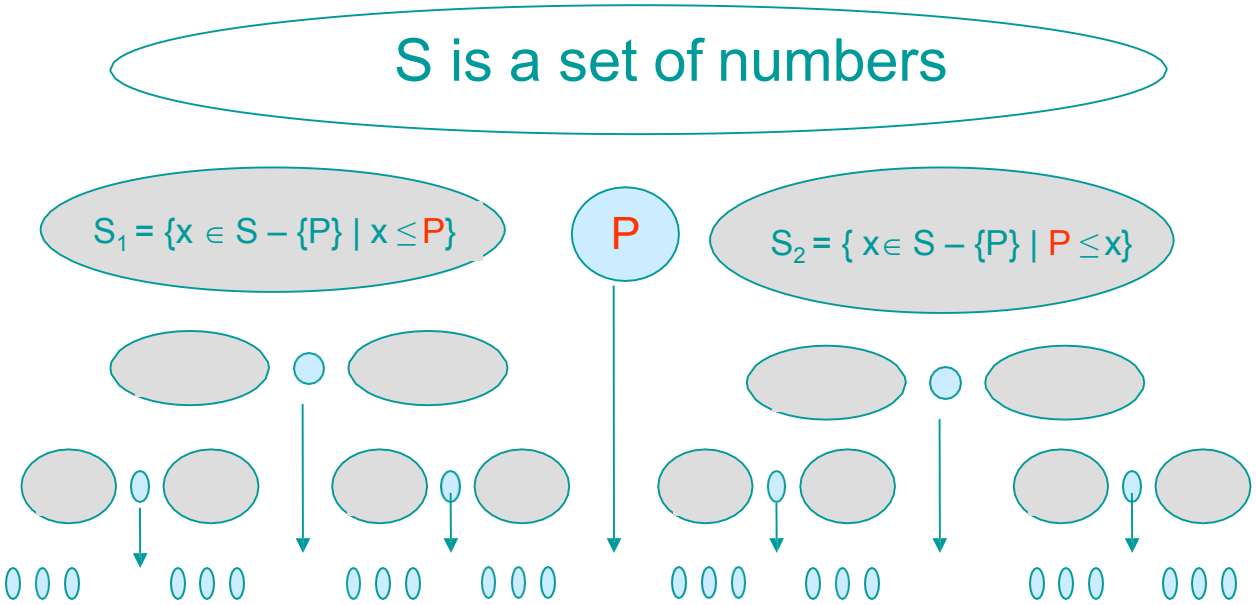
(Another divide-and-conquer algorithm)

- Pick an element, say ***P*** (the pivot)
- Re-arrange the elements into 3 sub-blocks,
  1. those less than or equal to (≤) ***P*** (the **left-**block S_1)
  2. ***P*** (the only element in the **middle**-block)
  3. those greater than or equal to (≥) ***P*** (the **right-**block S_2)
- Repeat the process **recursively** for the **left-** and **right-** sub-blocks. Return {quicksort(S_1), ***P***, quicksort(S_2)}. (That is the results of quicksort(S_1), followed by ***P***, followed by the results of quicksort(S_2))

Basic Ideas


Pick a "Pivot" value, P
Create 2 new sets without P
Items smaller than or equal to P
Items greater than or equal to P
0 31
13 43
26 57
≤
65
≤
75
92 81
quicksort(S_1)
quciksort(S_2)
0 13 26 31 43 57
≤
65
≤
75 81 92
0 13 26 31 43 57 65 75 81 92

Basic Ideas



Basic Ideas

Note:

- The main idea is to find the "right" position for the pivot element ***P***.
- After each **"pass"**, the pivot element, ***P***, should be **"in place".**
- Eventually, the elements are sorted since each pass puts at least one element (i.e., ***P***) into its final position.

Issues:

- How to choose the pivot ***P*** ?
- How to partition the block into sub-blocks?

Example

Input:	65	70	75	80	85	60	55	50	45	
P: 65	*i*									
Pass 1	**65**	70	75	80	85	60	55	50	45	
(i)		*i*							*j*	← swap (A[i], A[j])
	65	**45**	75	80	85	60	55	50	**70**	
(ii)			*i*					*j*		← swap (A[i], A[j])
	65	**45**	**50**	80	85	60	55	**75**	**70**	
(iii)				*i*			*j*			← swap (A[i], A[j])
	65	**45**	**50**	**55**	85	60	**80**	**75**	**70**	
(iv)					*i*	*j*				← swap (A[i], A[j])
	65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**	
(v)					*j*	*i*				if (i>=j) break
	60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**	swap (A[left], A[j])

Items smaller than or equal to 65

Items greater than or equal to 65

Quick Sort

Example

Result of Pass 1: 3 sub-blocks:

60 45 50 55 **65** 85 80 75 70

Pass 2a (left sub-block): 60 45 50 55 (P = 60)

i *j*

60 45 50 55

j *i* if `(i>=j) break`

55 45 50 60 swap (A[left], A[j])

Pass 2b (right sub-block): 85 80 75 70 (P = 85)

i *j*

85 80 75 70

j *i* if `(i>=j) break`

70 80 75 85 swap (A[left], A[j])

Running time analysis

- The advantage of this quicksort is that we can sort "in-place", i.e., *without* the need for a temporary buffer depending on the size of the inputs. (cf. mergesort)

Partitioning Step: Time Complexity is θ**(n).**

Recall that quicksort involves *partitioning, and 2 recursive calls.*
Thus, giving the basic quicksort relation:

T(n) **=** θ**(n) +** ***T(i) + T(n-i-1) = cn+ T(i) + T(n-i-1)***

where ***i*** is the size of the first sub-block after partitioning. We shall take ***T(0) = T(1) = 1*** as the initial conditions.

To find the solution for this relation, we'll consider three cases:

1. The Worst-case (?)
2. The Best-case (?)
3. The Average-case (?)

All depends on the value of the pivot!!

Running time analysis

Worst-Case (Data is sorted already)

- When the pivot is the smallest (or largest) element at partitioning on a block of size ***n***, the result
  - yields one empty sub-block, one element (pivot) in the "correct" place and one sub-block of size ***(n-1)***
  - takes θ**(n)** times.
- Recurrence Equation:

$$\left.\begin{array}{l} T(1) = 1 \\ T(n) = T(n-1) + cn \end{array}\right\}$$

Solution: $\theta(n^2)$

Worse than Mergesort!!!

Running time analysis

Best case:

- The pivot is in the middle (median) (at each partition step), i.e. after each partitioning, on a block of size ***n,*** the result
  - yields two sub-blocks of approximately equal size and the pivot element in the "middle" position
  - takes ***n*** data comparisons.
- Recurrence Equation becomes

T(1) = 1

T(n) = 2T(n/2) + cn

Solution: $\theta(n \log n)$

Comparable to Mergesort!!

Running time analysis

Average case:

It turns out the average case running time also is θ***(n logn).***

So the trick is to select a good pivot

Different ways to select a good pivot.

- First element
- Last element
- Median-of-three elements
  - Pick three elements, and find the median *x of these elements*. Use that median as the pivot.
- Random element
  - Randomly pick a element as a pivot.

Different sorting algorithms

Sorting Algorithm	Worst-case time	Average-case time	Space overhead
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(1)$

Quick Sort

QUICK_SORT(K, LB, UB):- K is a vector. LB and UB denote the lower and upper bounds of the current subtable. I and J index varible. KEY contains the key value. FLAG is a logical varible.

1. [Initialize]

```
FLAG <-  true
```

2. [Perform sort]

```
if  LB < UB
then I <- LB
     J <- UB + 1
     KEY <- K[LB]
     repeat while FLAG
        I <- I + 1
        repeat while K[I] < KEY && I < UB
                I<- I + 1
        J <- J -1
        repeat while K[J] > KEY
                J <- J -1
```

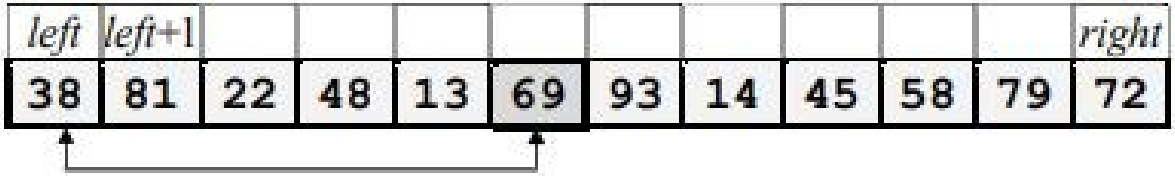
Quick Sort (Continue....)

```
if I < J
then K[I] <———> k[J]
else
        FLAG <- false
K[LB] < ———> K[J]
call QUICK_SORT(K, LB, J-1)
call QUICK_SORT(K, J + 1, UB)
```

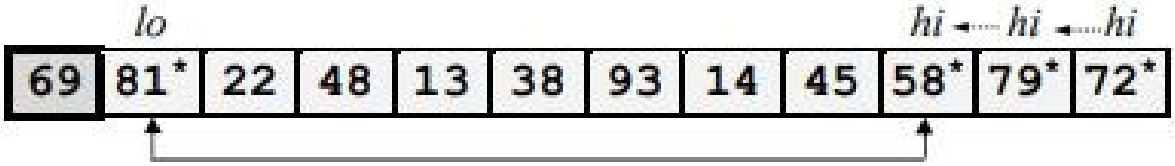
7. [Finish]

Exit

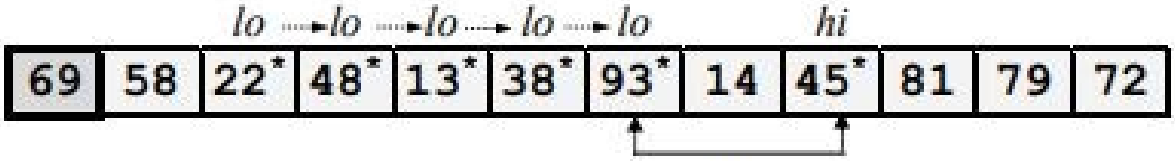
Swap pivot element with leftmost element. *lo*=*left*+1; *hi*=*right*;



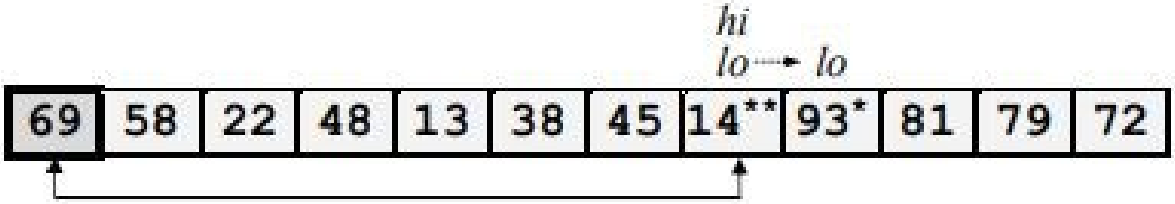
Move *hi* left and *lo* right as far as we can; then swap *A*[*lo*] and *A*[*hi*], and move *hi* and *lo* one more position.



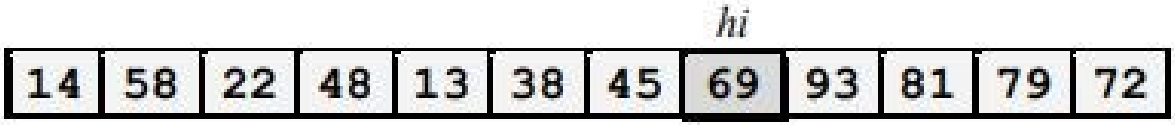
Repeat above



Repeat above until *hi* and *lo* cross; then *hi* is the final position of the pivot element, so swap *A*[*hi*] and *A*[*left*].



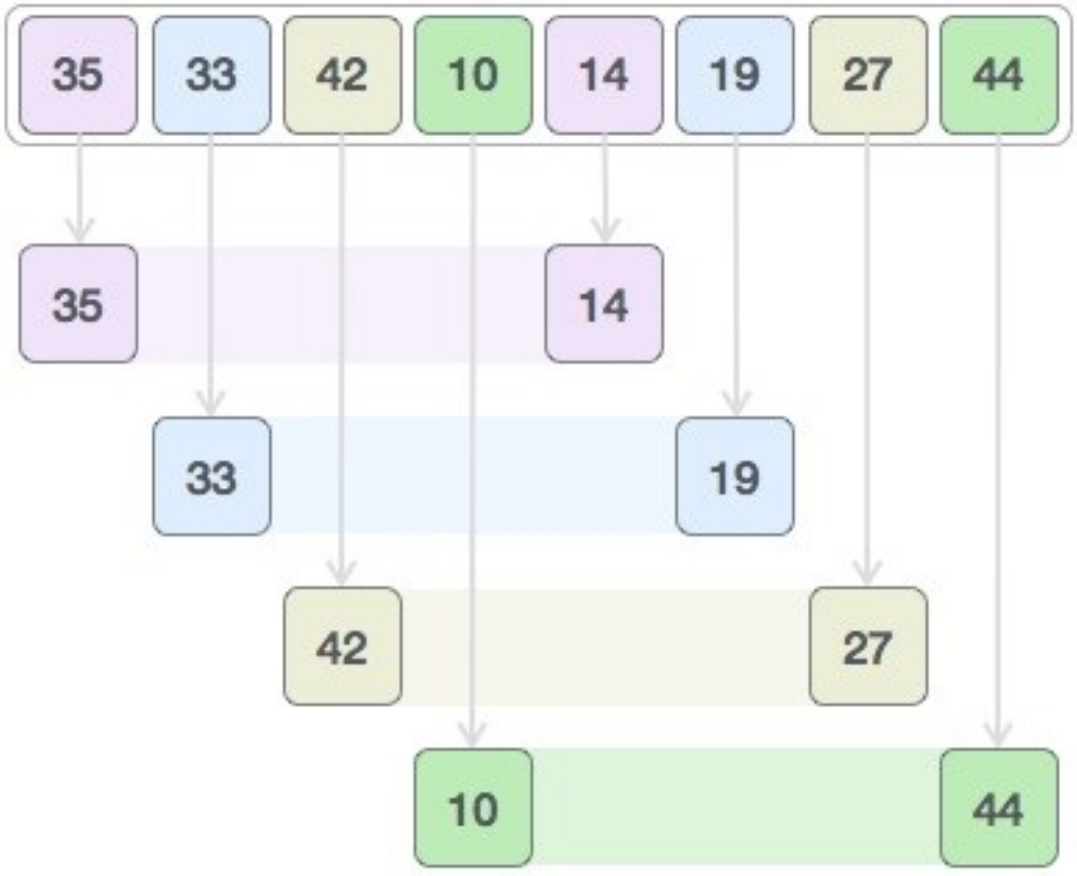
Partitioning complete; return value of *hi*.



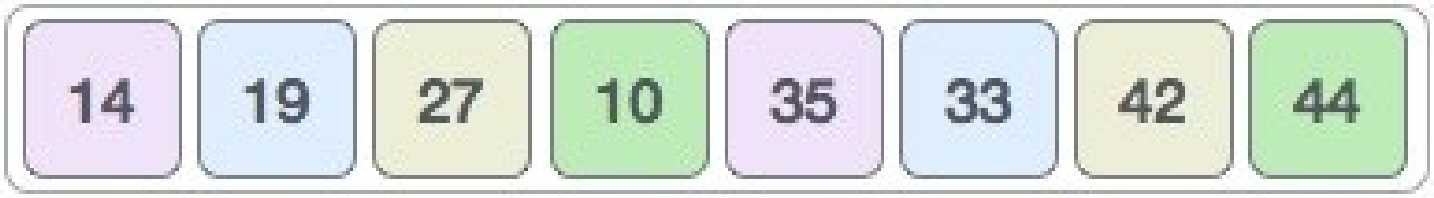
Shell Sort

- ShellSort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead.
- When an element has to be moved far ahead, many movements are involved.
- The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of h.
- We keep reducing the value of gap until it becomes 1.
- An array is said to be h-sorted if all sublists of every h'th element is sorted.
- This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items.

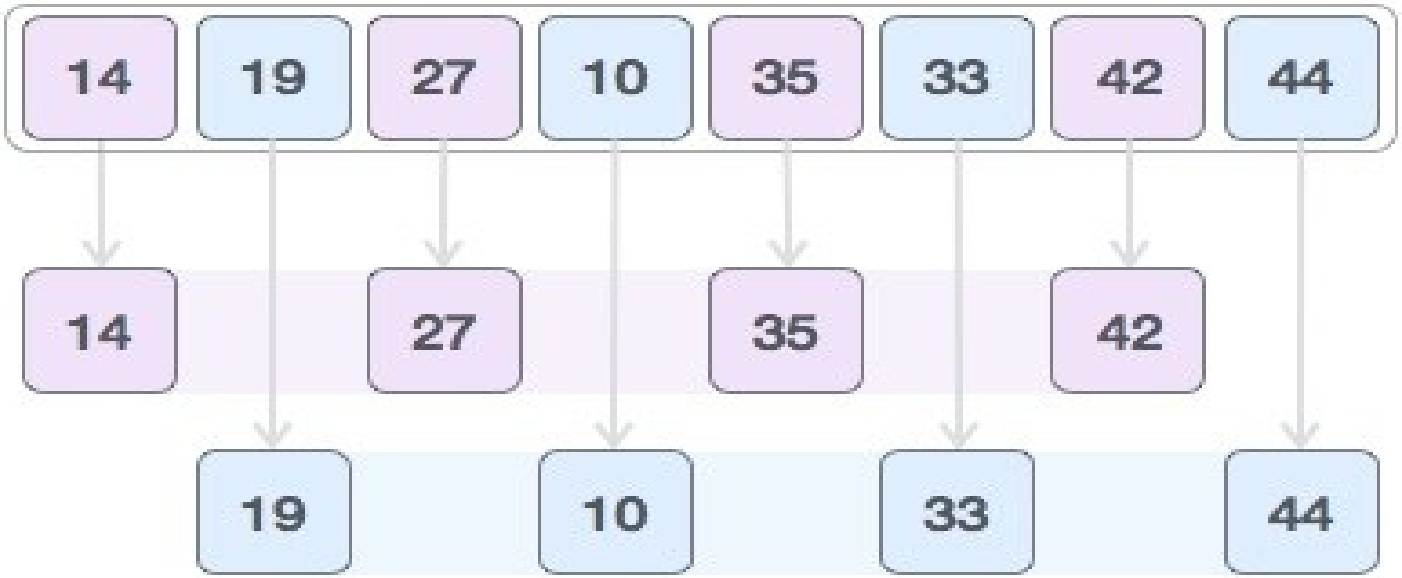
Let us consider the following example to have an idea of how shell sort works. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



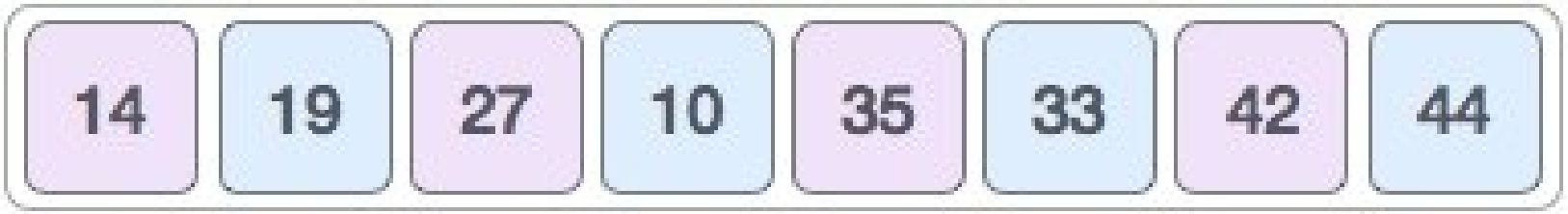
- We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this
-



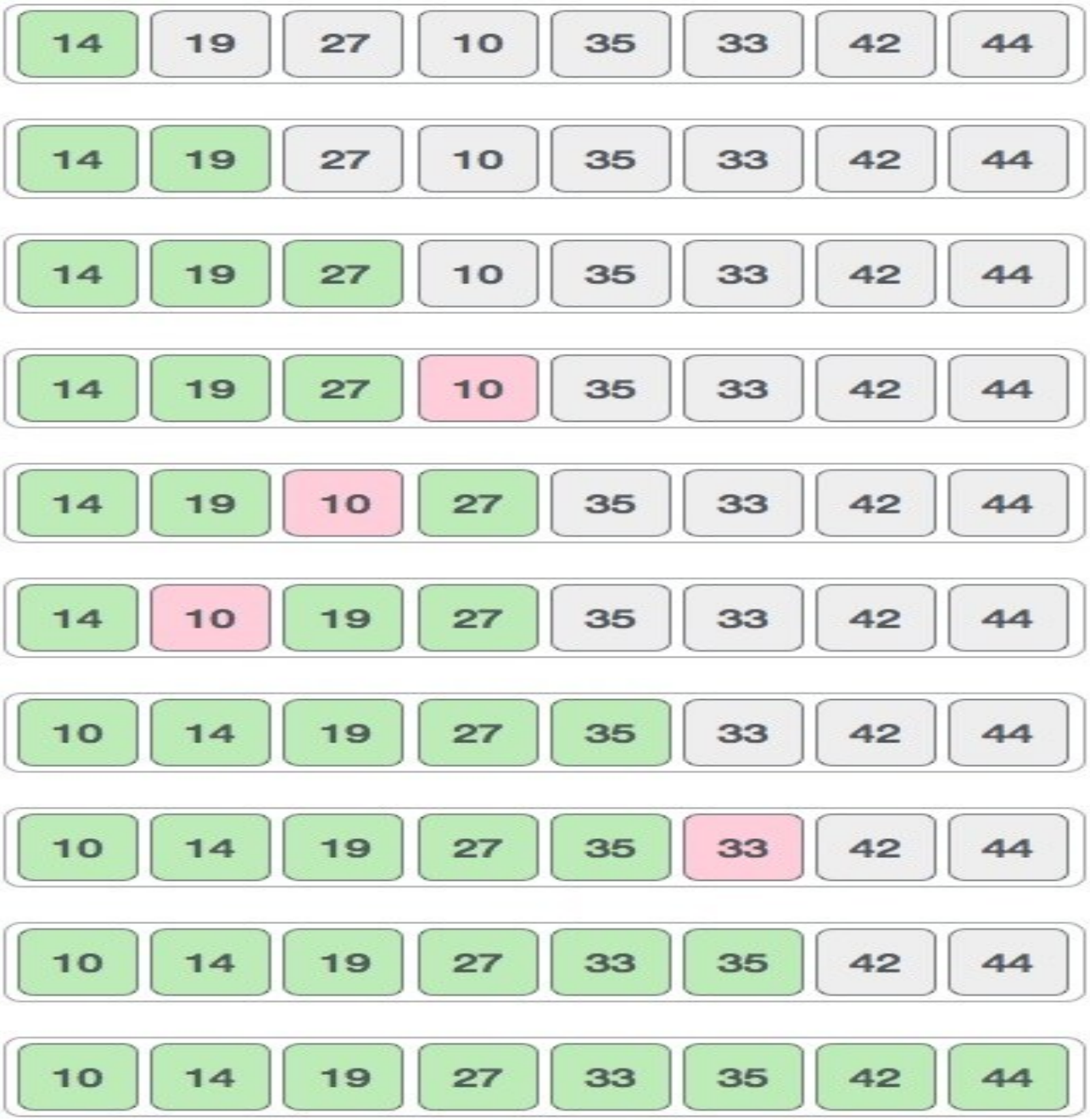
- Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



- We compare and swap the values, if required, in the original array. After this step, the array should look like this −



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array


14 19 27 10 35 33 42 44
14 19 27 10 35 33 42 44
14 19 27 10 35 33 42 44
14 19 27 10 35 33 42 44
14 19 10 27 35 33 42 44
14 10 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 33 35 42 44
10 14 19 27 33 35 42 44

Shell Sort

SHELL_SORT(K,N):- This algorithm is used to sort all N elements in ascending order. N is integer type variable indicate there are N elements in vector K. The variable GAP is used to find gap between two elements. I and J denotes index elements.

1. [Find gap]

GAP <- N/2 + 1

2. [Loop on pass index]

Repeat thru step 3 for I = GAP, GAP – 1, ….. , 1

3. [Perform comparisons]

Repeat for J = 0,1,2, ….. N - GAP

if K[J] > K[J + I]

then K[J] <- > K[J + I]

4. [Finish]

Exit

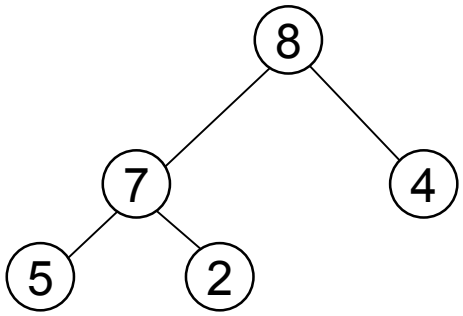
Heap Sort

- http://www.algoanim.ide.sk/index.php?page=showanim&id=52

The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

    Parent(x) ≥ x



Heap

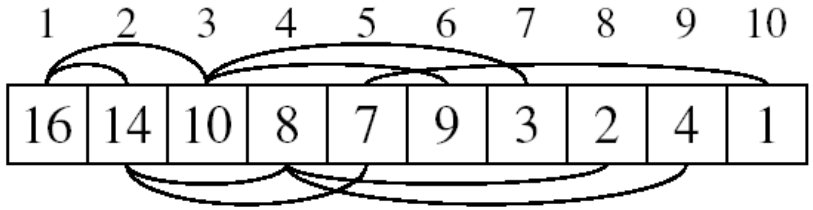
From the heap property, it follows that:

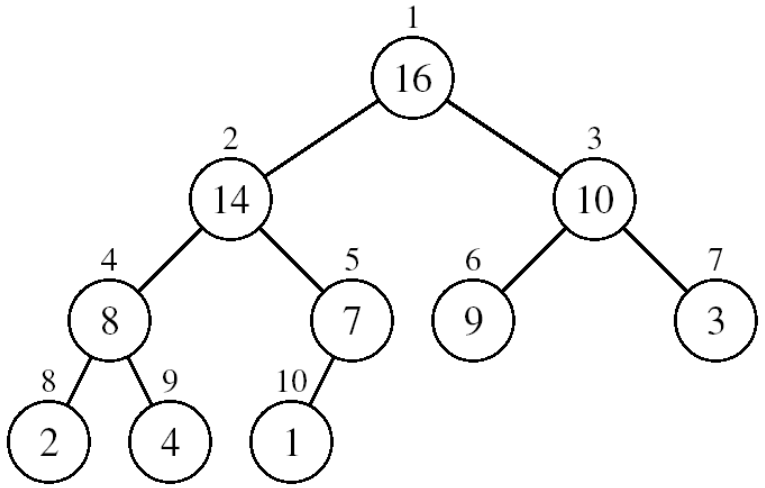
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array A.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves





Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

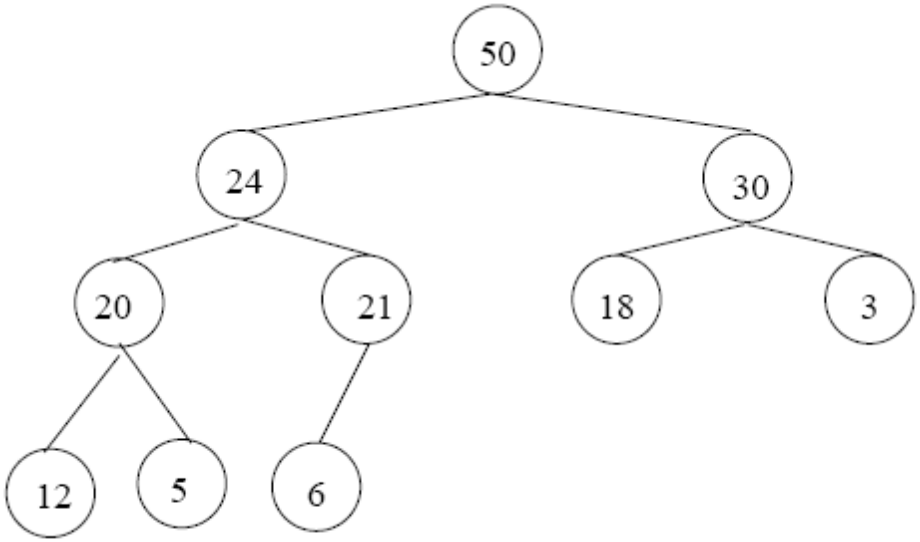
$$A[\text{PARENT}(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

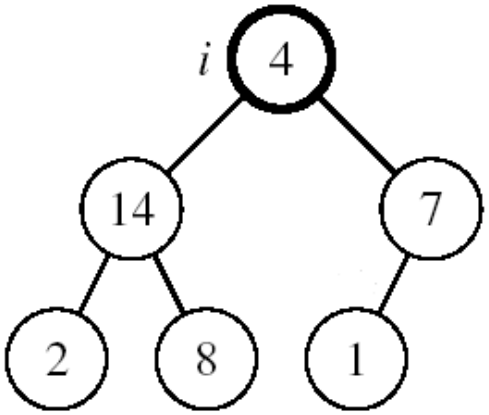


Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT

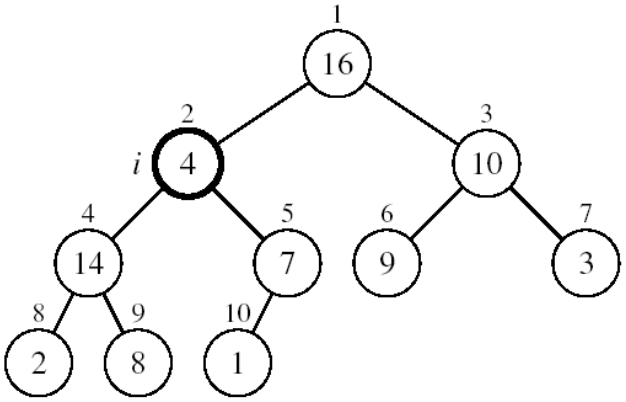
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



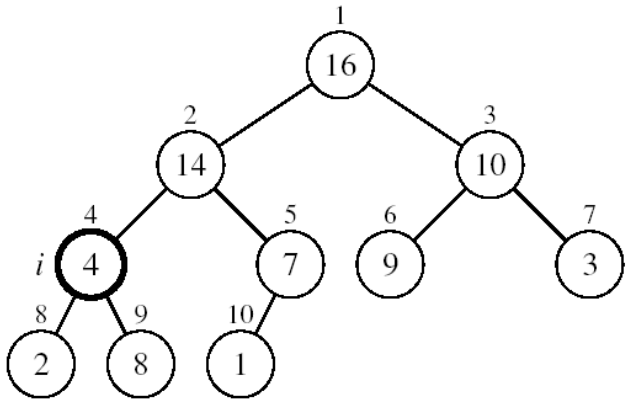
Example

MAX-HEAPIFY(A, 2, 10)



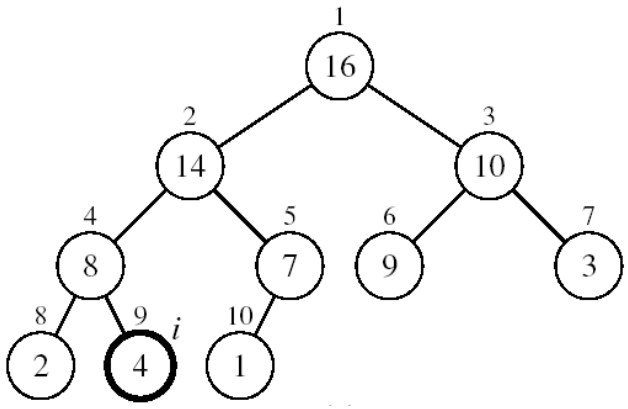
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

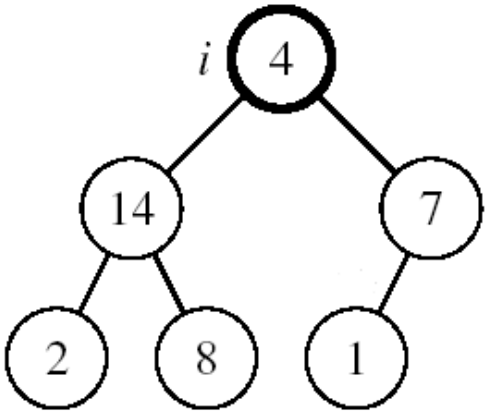
A[4] ↔ A[9]



Heap property restored

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - $A[i]$ may be smaller than its children



Alg: MAX-HEAPIFY(A, i, n)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and A[l] > A[i]
4. **then** largest ← l
5. **else** largest ← i
6. **if** r ≤ n and A[r] > A[largest]
7. **then** largest ← r
8. **if** largest ≠ i
9. **then** exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest, n)

MAX-HEAPIFY Running Time

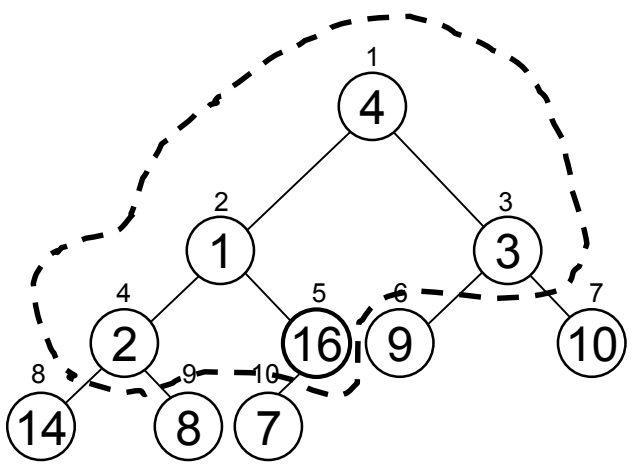
- Intuitively:
  - It traces a path from the root to a leaf (longest path length: d)
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons is $2d$
  - Running time is $O(d)$ or $O(lgn)$
- Running time of MAX-HEAPIFY is $O(\log n)$
- Can be written in terms of the height of the heap, as being $O(h)$
  - Since the height of the heap is $\lfloor \lg n \rfloor$

Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

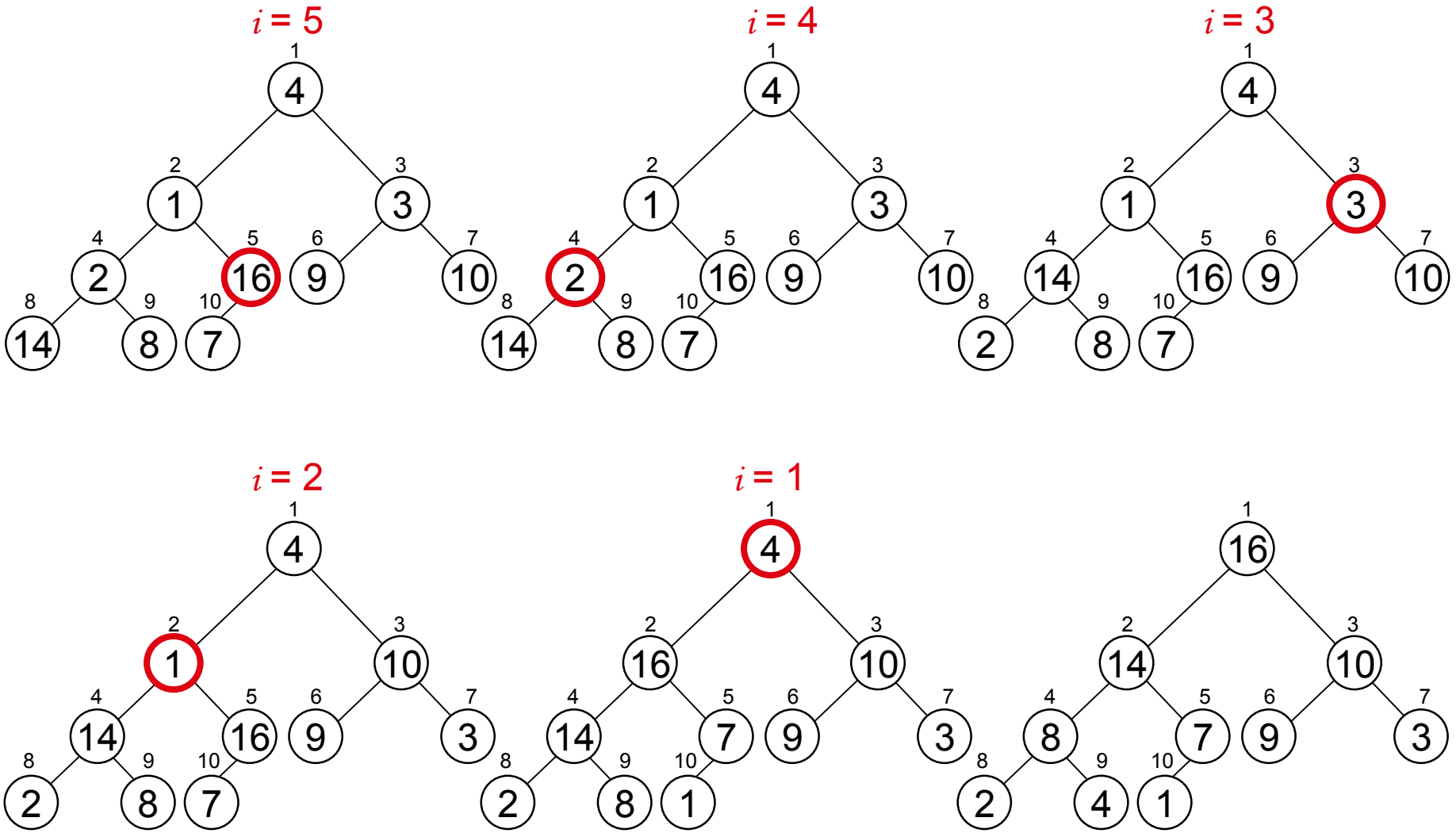


A:

4	1	3	2	16	9	10	14	8	7

Example: A

4	1	3	2	16	9	10	14	8	7



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)

$O(\lg n)$

$O(n)$

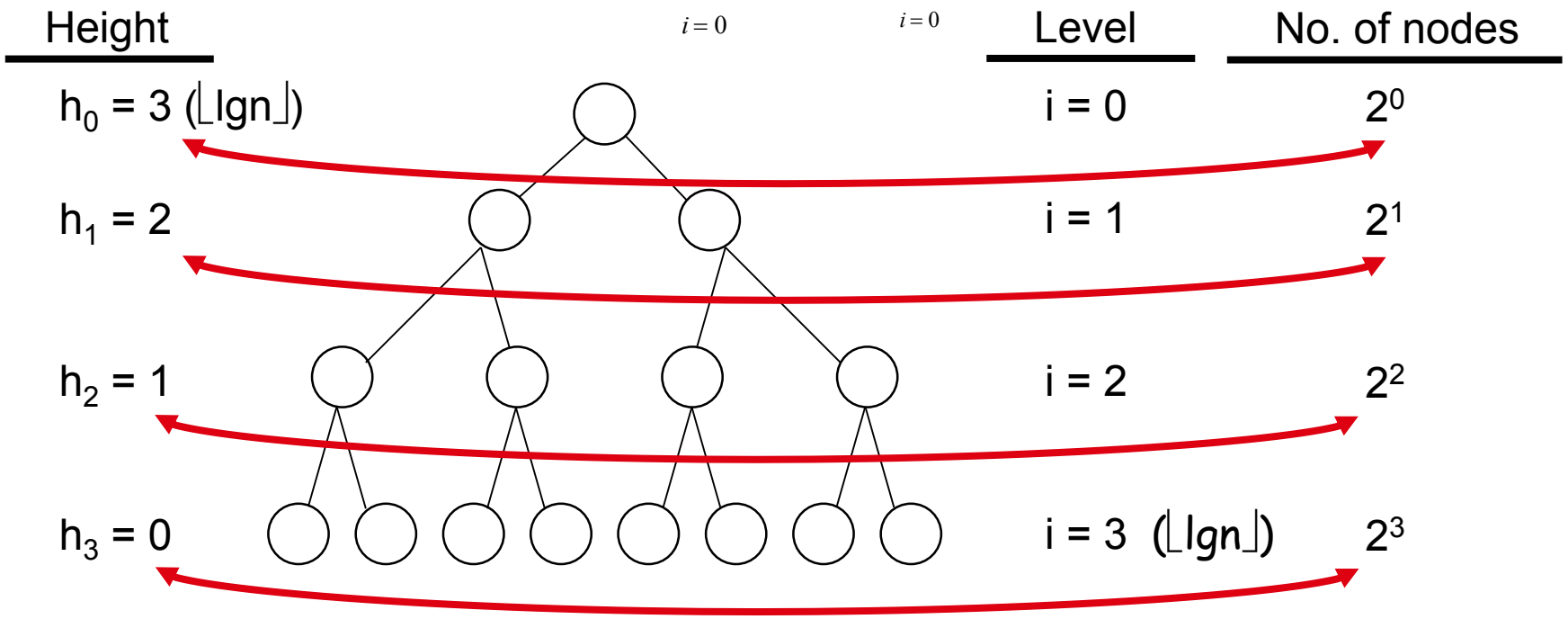
⇒ Running time: $O(n \lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes $O(h) \Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h - i) = O(n)$$



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

Create Heap

CREATE_HEAP(K,N):- This algorithm is used to create a heap tree. Vector K containing the N records of a table. The index variable Q controls the number of insertions. The integer variable J denotes the index of the parent of key. KEY contains the key value. I denotes index elements.

1. [Build Heap]

Repeat thru step 7 for Q = 2,3,4,….. N

2. [Initialize construction phase]

I <- Q

KEY <- K[Q]

3. [Obtain parent of new record]

J <- TRUNC (I / 2)

4. [Place new record in existing heap]

Repeat thru step 6 while I > 1 and KEY > K[J]

5. [Interchange record]

K[I] <- K [J]

6. [Obtain next parent]

I <- J

J <- TRUNC (I / 2)

IF J < 1

then J<- 1

7. [Copy new record into its proper place]

K[I] <- KEY

8. [Finish]

exit

Heap Sort

HEAP_SORT(K,N):- This algorithm is used to sort all N elements in ascending order. N is integer type variable indicate there are N elements in vector K. The variable Q represents the pass index. KEY contains the key value. I and J denotes index elements.

1. [Create the initial Heap]

Call CREATE_HEAP(K,N)

2. [Perform sort]

Repeat thru step 10 for Q = N, N-1, N-2, ….. 2

3. [Exchange record]

K[1] <---> K[Q]

4. [Initialize pass]

I <- 1

KEY <- K[1]

J <- 2

5. [Obtain index of largest son of new record]

if J + 1 < Q

then if K[J + 1] > K[J]

then J <- J + 1

6. [Reconstruct the new heap]

Repeat thru step 10 while J <= Q -1 and K[J] > KEY

7. [Interchange record]

K[I] <- K [J]

8. [Obtain next left son]

I <- J

J <- 2 * I

9. [Obtain index of next largest son]

if J + 1 < Q

then if K[J + 1] > K[J]

then J <- J + 1

else if J > N

then J<- N

10. [Copy record into its proper place]

K[I] <- KEY

11. [Finish]

exit

Algorithm	Location	Big O
Searching Algorithms		
Linear search	Section 20.2.1	$O(n)$
Binary search	Section 20.2.2	$O(\log n)$
Recursive linear search	Exercise 20.8	$O(n)$
Recursive binary search	Exercise 20.9	$O(\log n)$
Sorting Algorithms		
Insertion sort	Section 20.3.1	$O(n^2)$
Selection sort	Section 20.3.2	$O(n^2)$
Merge sort	Section 20.3.3	$O(n \log n)$
Bubble sort	Exercises 20.5–20.6	$O(n^2)$
Quicksort	Exercise 20.10	Worst case: $O(n^2)$ Average case: $O(n \log n)$

Fig. 20.7

n	Approximate decimal value	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	$2^{10} \cdot 10$	2^{20}
2^{20}	1,000,000	20	2^{20}	$2^{20} \cdot 20$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$2^{30} \cdot 30$	2^{60}

Fig. 20.8 | Approximate number of comparisons for common Big O notations.