Linear Search
&
Binary Search

What is algorithm and Algorithm design?

- An Algorithm is a Step by Step solution of a specific mathematical or computer related problem.
- Algorithm design is a specific method to create a mathematical process in solving problems.

Sorted Array

- Sorted array is an array where each element is **sorted** in numerical, alphabetical, or some other order, and placed at equally spaced addresses in computer memory.

1	2	3	4
0.2	0.3	1	1.5

Unsorted Array

- Unsorted array is an array where each element is not **sorted** in numerical, alphabetical, or some other order, and placed at equally spaced addresses in computer memory.

1	**2**	**3**	**4**
0.2	**0.3**	**1.5**	**1**

What is searching?

- In computer science, searching is the process of finding an item with specified properties from a collection of items.
- The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or maybe be elements in other search place.

- The **definition** of a **search** is the process of looking f something or someone
- Example : An example of a **search** is a quest to find a missing person

Why do we need searching?

✓Searching is one of the core computer science algorithms.

✓We know that today's computers store a lot of information.

✓To retrieve this information proficiently we need very efficient searching algorithms.

Types of Searching

- Linear search
- Binary search

Linear Search

- The linear search is a sequential search, which uses a loop to step through an array, starting with the first element.
- It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered.
- If the value being searched is not in the array, the algorithm will unsuccessfully search to the end of the array.

Linear Search

- Since the array elements are stored in linear order searching the element in the linear order make it easy and efficient.
- The search may be successful or unsuccessfully. That is, if the required element is found them the search is successful other wise it is unsuccessfully.

Unordered linear/ Sequential search

```
int unorderedlinearsearch (int A[], int n, int data)
{
        for (int i=0; i<n; i++)
        {
                if(A[i] == data)
                        return i;
        }
        return -1;
}
```

Advantages of Linear search

- If the first number in the directory is the number you were searching for ,then lucky you!!.
- Since you have found it on the very first page, now its not important for you that how many pages are there in the directory.
- The linear search is simple - It is very easy to understand and implement
- It does not require the data in the array to be stored in any particular order
- So it does not depends on no. on elements in the directory. Hence constant time .

- Your search time is proportional to number of elements in the directory.
- It has very poor efficiency because it takes lots of comparisons to find a particular record in big files
- The performance of the algorithm scales linearly with the size of the input
- Linear search is slower then other searching algorithms

Analysis of Linear Search

How long will our search take?

In the **best case**, the target value is in the first element of the array.

So the search takes some tiny, and constant, amount of time.

In the **worst case**, the target value is in the last element of the array.

So the search takes an amount of time proportional to the length of the array.

Analysis of Linear Search

In the **average case**, the target value is somewhere in the arra

In fact, since the target value can be anywhere in the array, an element of the array is equally likely.

So on average, the target value will be in the middle of the array.

So the search takes an amount of time proportional to half the length of the array

The worst case complexity is O(n), sometimes known an O(n) search

Time taken to search elements keep increasin as the number of elements are increased.

Binary Search

The general term for a smart search through sorted data is a ***binary search***.

1. The initial search region is the whole array.
2. Look at the data value in the middle of the search region.
3. If you’ve found your target, stop.
4. If your target is less than the middle data value, the new search region is the lower half of the data.
5. If your target is greater than the middle data value, the new search region is the higher half of the data.
6. Continue from Step 2.

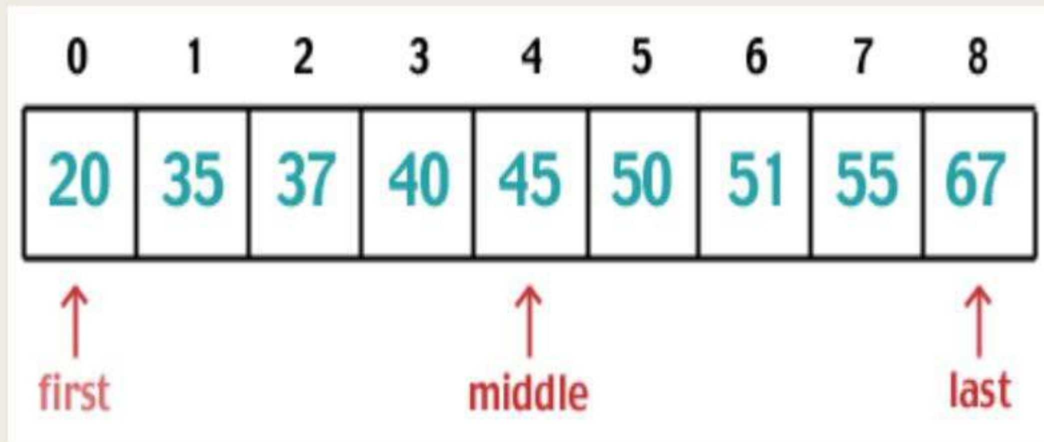
Binary Search

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Binary Search

2. Calculate middle = (low + high) / 2.

= (0 + 8) / 2 = 4.



If 37 == array[middle] return middle

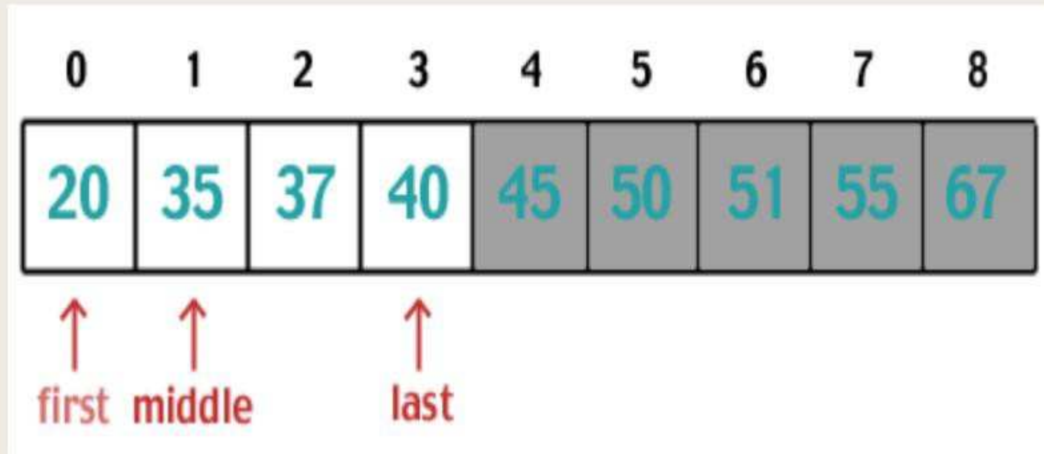
Else if 37 < array[middle] high = middle -1

Else if 37 > array[middle] low = middle +1

Binary Search

Repeat 2. Calculate middle = (low + high) / 2.

= (0 + 3) / 2 = 1.



If 37 == array[middle] ▯ return middle

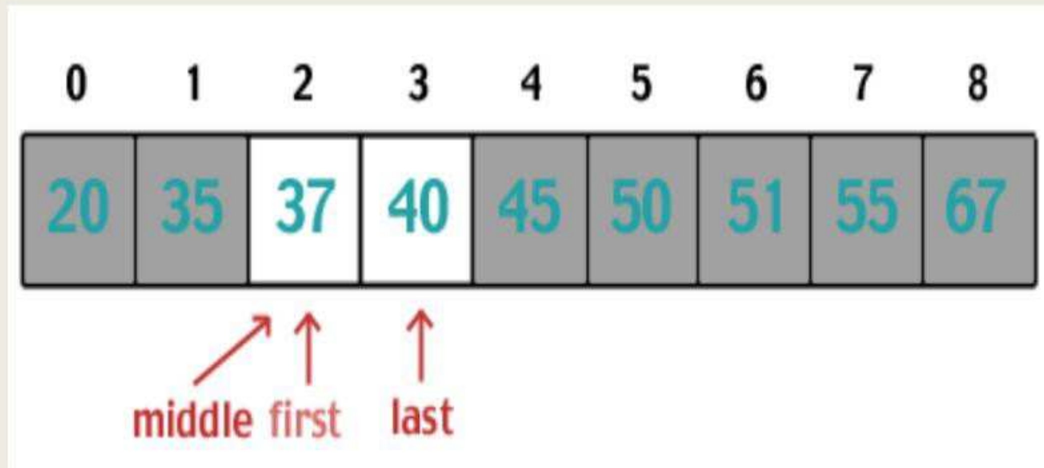
Else if 37 < array[middle] ▯ high = middle -1

Else if 37 > array[middle] ▯ low = middle +1

Binary Search

Repeat 2. Calculate middle = (low + high) / 2.

= (2 + 3) / 2 = 2.



If 37 == array[middle] return middle

Else if 37 < array[middle] high = middle -1

Else if 37 > array[middle] low = middle +1

Binary Search Routine

```
public int binarySearch (int[] number, int searchValue)
{

    int low = 0, high = number.length - 1, mid = (low + high) / 2;

        while (low <= high && number[mid] != searchValue) {
                if (number[mid] < searchValue) {
                        low = mid + 1;
                }
                else
                { //number[mid] > searchValue
                        high = mid - 1;
                }
                mid = (low + high) / 2; //integer
        division will truncate
        }
        if (low > high) {
                mid =
                NOT_FOUND;
        return mid;
        }
        }
```

- Successful Search
  - Best Case – 1 comparison
  - Worst Case – $\log_2 N$ comparisons
- Unsuccessful Search
  - Best Case = Worst Case – log 2 N comparisons
- Since the portion of an array to search is cut into half after every comparison, we compute how many times the array can be divided into halves.
- After K comparisons, there will be $N/2^K$ elements in the list. We solve for K when N/2K = 1, deriving K = $\log_2 N$.

Performance

Array Size	Linear – N	Binary – $log_2 N$
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Important Differences:

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search -O(n) , Binary search has time complexity O(log n).
- Linear search performs equality comparisons and Binary search performs ordering comparisons

BUBBLE SORT ALGORITHM

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where **n** is the number of items.

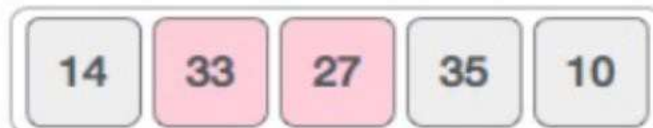
How Bubble Sort Works?

- We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.

- Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

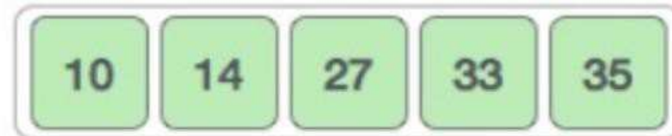
We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Implementation in C

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 10

int list[MAX] = {1,8,4,6,0,3,5,2,7,9};

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0; i < MAX; i++) {
      printf("%d ",list[i]);
   }

   printf("]\n");
}

void bubbleSort() {
   int temp;
   int i,j;

   bool swapped = false;

   // loop through all numbers
   for(i = 0; i < MAX-1; i++) {
      swapped = false;
```

```
      // loop through numbers falling ahead
      for(j = 0; j < MAX-1-i; j++) {
         printf("     Items compared: [ %d, %d ] ", list[j],list[j+1]);

         // check if next number is lesser than current no
         //   swap the numbers.
         //  (Bubble up the highest number)

         if(list[j] > list[j+1]) {
            temp = list[j];
            list[j] = list[j+1];
            list[j+1] = temp;

            swapped = true;
            printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
         }else {
            printf(" => not swapped\n");
         }

      }

      // if no number was swapped that means
      //   array is sorted now, break the loop.
      if(!swapped) {
         break;
      }

      printf("Iteration %d#: ",(i+1));
      display();
   }

}
```

```
main() {
   printf("Input Array: ");
   display();
   printf("\n");

   bubbleSort();
   printf("\nOutput Array: ");
   display();
}
```

If we compile and run the above program, it will produce the following result –

Output

```
Input Array: [1 8 4 6 0 3 5 2 7 9 ]


     Items compared: [ 1, 8 ]  => not swapped
     Items compared: [ 8, 4 ]  => swapped [4, 8]
     Items compared: [ 8, 6 ]  => swapped [6, 8]
     Items compared: [ 8, 0 ]  => swapped [0, 8]
     Items compared: [ 8, 3 ]  => swapped [3, 8]
     Items compared: [ 8, 5 ]  => swapped [5, 8]
     Items compared: [ 8, 2 ]  => swapped [2, 8]
     Items compared: [ 8, 7 ]  => swapped [7, 8]
     Items compared: [ 8, 9 ]  => not swapped
```

```
Iteration 1#: [1 4 6 0 3 5 2 7 8 9 ]
     Items compared: [ 1, 4 ]  => not swapped
     Items compared: [ 4, 6 ]  => not swapped
     Items compared: [ 6, 0 ]  => swapped [0, 6]
     Items compared: [ 6, 3 ]  => swapped [3, 6]
     Items compared: [ 6, 5 ]  => swapped [5, 6]
     Items compared: [ 6, 2 ]  => swapped [2, 6]
     Items compared: [ 6, 7 ]  => not swapped
     Items compared: [ 7, 8 ]  => not swapped
Iteration 2#: [1 4 0 3 5 2 6 7 8 9 ]
     Items compared: [ 1, 4 ]  => not swapped
     Items compared: [ 4, 0 ]  => swapped [0, 4]
     Items compared: [ 4, 3 ]  => swapped [3, 4]
     Items compared: [ 4, 5 ]  => not swapped
     Items compared: [ 5, 2 ]  => swapped [2, 5]
     Items compared: [ 5, 6 ]  => not swapped
     Items compared: [ 6, 7 ]  => not swapped
Iteration 3#: [1 0 3 4 2 5 6 7 8 9 ]
     Items compared: [ 1, 0 ]  => swapped [0, 1]
     Items compared: [ 1, 3 ]  => not swapped
     Items compared: [ 3, 4 ]  => not swapped
     Items compared: [ 4, 2 ]  => swapped [2, 4]
     Items compared: [ 4, 5 ]  => not swapped
     Items compared: [ 5, 6 ]  => not swapped
```

```
Iteration 4#: [0 1 3 2 4 5 6 7 8 9 ]
     Items compared: [ 0, 1 ]  => not swapped
     Items compared: [ 1, 3 ]  => not swapped
     Items compared: [ 3, 2 ]  => swapped [2, 3]
     Items compared: [ 3, 4 ]  => not swapped
     Items compared: [ 4, 5 ]  => not swapped
Iteration 5#: [0 1 2 3 4 5 6 7 8 9 ]
     Items compared: [ 0, 1 ]  => not swapped
     Items compared: [ 1, 2 ]  => not swapped
     Items compared: [ 2, 3 ]  => not swapped
     Items compared: [ 3, 4 ]  => not swapped


Output Array: [0 1 2 3 4 5 6 7 8 9 ]
```

Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

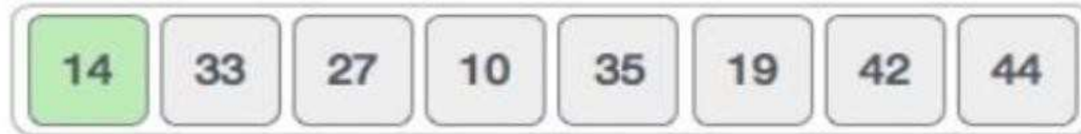
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

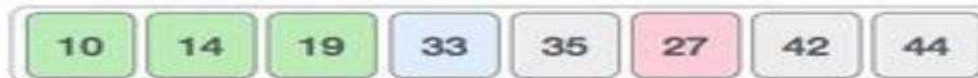
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

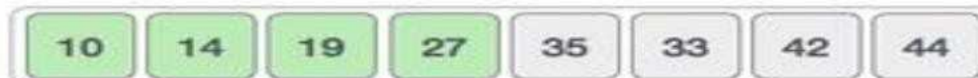
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −





Algorithm

```
Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted
```

Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if

   end for

end procedure
```

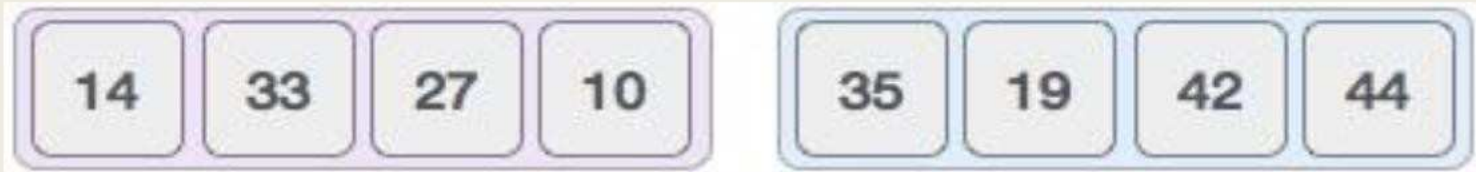
Merge Sort Algorithm

- Merge sort is a sorting technique based on divide and conquer technique. With Average case and worst-case time complexity being O(n log n), it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.

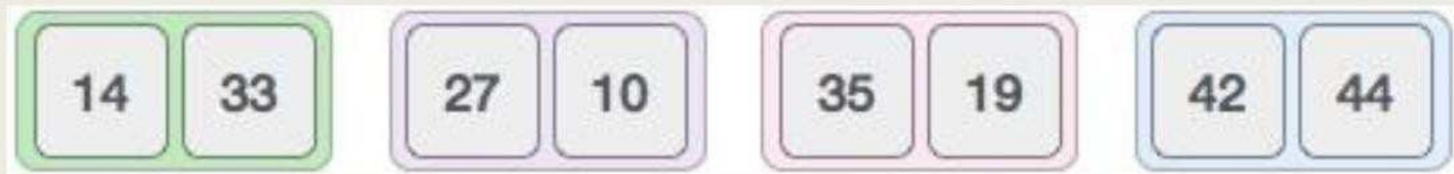
How merge sort works

- To understand merge sort, we take an unsorted array as depicted below −

- We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We se here that an array of 8 items is divided into two arrays of size 4.

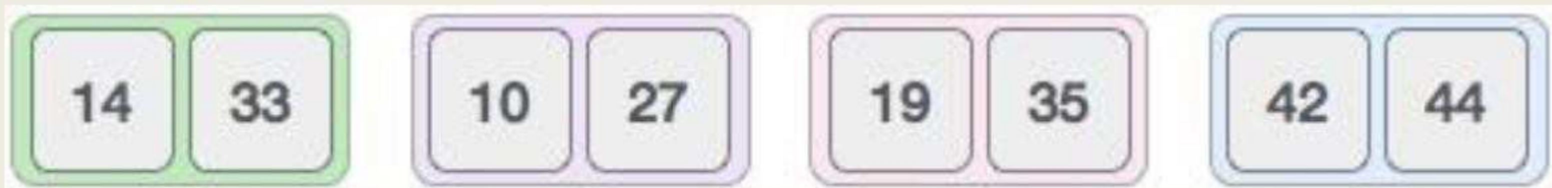


- This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

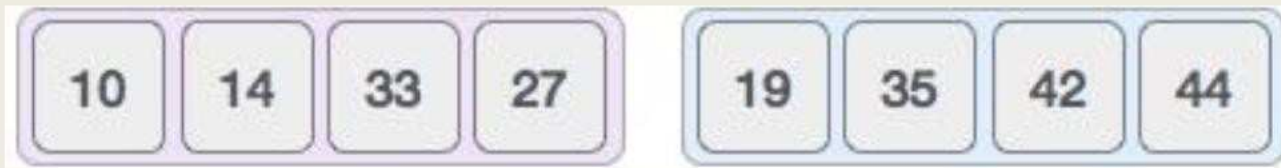


- We further divide these arrays and we achieve atomic value which can no more be divided.

- Now, we combine them in exactly same manner they were broken down.
- We first compare the element for each list and then combine them into another list in sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order 19 and 35. 42 and 44 are placed sequentially.



- In next iteration of combining phase, we compare lists of two data values, and merge them into a list of four data values placing all in sorted order.



- After final merging, the list should look like this −

Algorithm

- Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then merge sort combines smaller sorted lists keeping the new list sorted too.
  - **Step 1** − divide the list recursively into two halves until it can no more be divided.
  - **Step 2** − if it is only one element in the list it is already sorted, return.
  - **Step 3** − merge the smaller lists into new list in sorted order.

Data Structure - Shell Sort

- Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort if smaller value is very far right and have to move to far left.
- This algorithm uses insertion sort on widely spread elements first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as −

- **h = h * 3 + 1**

  where − h is interval with initial value 1

  This algorithm is quite efficient for medium sized data sets as its average and worst case complexity are of O(n^2) where n are no. of items.

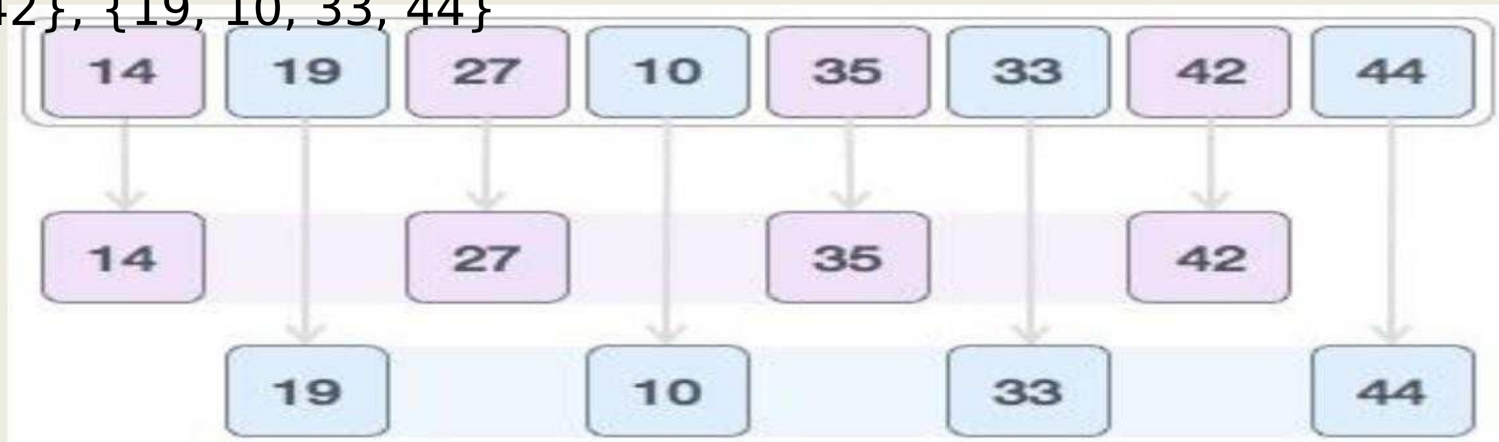
How shell sort works

- We take the below example to have an idea, how shell sort works?

- We take the same array we have used in our previous examples. {35,33,42,10,14,19,27,44}
- For our example and ease of understanding we take the interval of 4.
- And make a virtual sublist of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 14}

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

Then we take interval of 2 and this gap generates two sublists - {14, 27, 35,
42}, {19, 10, 33, 44}



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

And finally, we sort the rest of the array using interval of value 1. Shell
sort uses insertion sort to sort the array. The step by step depiction is shown below −

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

Algorithm

- We shall now see the algorithm for shell sort.
- **Step 1** − Initialize the value of *h*
- **Step 2** − Divide the list into smaller sub-list of equal interval *h*
- **Step 3** − Sort these sub-lists using **insertion sort**
- **Step 4** − Repeat until complete list is sorted

Radix Sort

- Radix Sort is generalization of Bucket Sort
- To sort Decimal Numbers radix/base will be used as 10. so we need 10 buckets.
- Buckets are numbered as 0,1,2,3,…,9
- Sorting is Done in the passes
- Number of Passes required for sorting is number of digits in the largest number in the list.

Ex.

Range	Passes
0 to 99	2 Passes
0 to 999	3 Passes
0 to 9999	4 Passes

- In First Pass number sorted based on Least Significant Digit and number will be kept in same bucket.
- In 2nd Pass, Numbers are sorted on second least significant bit and process continues.
- At the end of every pass, numbers in buckets are merged to produce common list.

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

- Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.
- **Average case and Worst case Complexity - O(n)**

Disadvantages

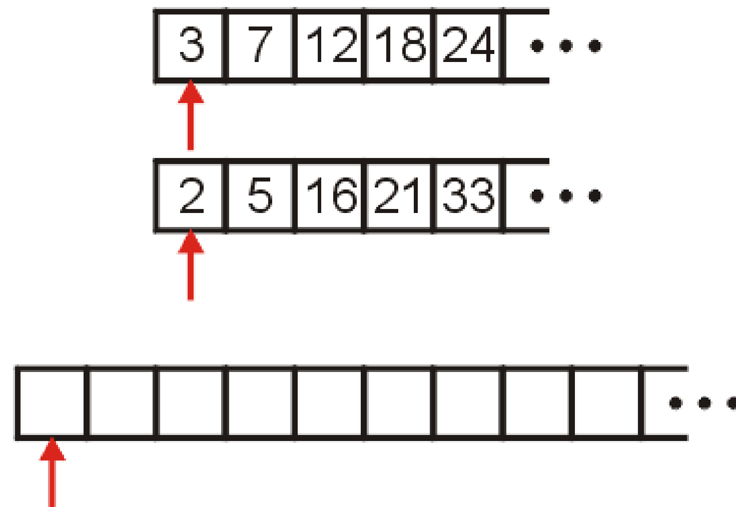
- Still, there are some tradeoffs for Radix Sort that can make it less preferable than other sorts.
- The speed of Radix Sort largely depends on the inner basic operations, and **if** the operations are not efficient enough, **Radix Sort *can* be slower than some other algorithms** such as Quick S and Merge Sort.
- In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix Sort, and it is one of the hardest to make efficient.
- Radix Sort can also take up more **space** than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sublist for **each** of the possible digits or letters.

Merge Sort

- The next sorting algorithm is one which is defined recursively
- Suppose we:
  - divide an unsorted list into two sub-lists,
  - sort each sub list
- How quickly can we recombine the two sub-lists into a single sorted list?

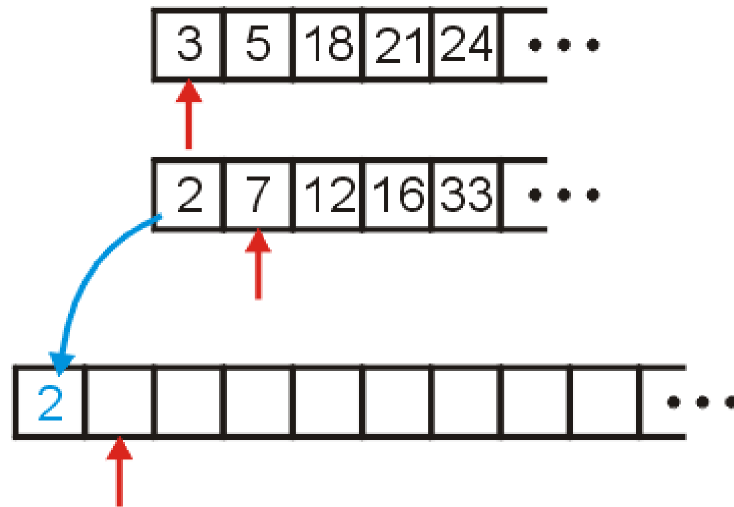
Example

- Consider the two sorted arrays and an empty array
- Define three indices at the start of each array



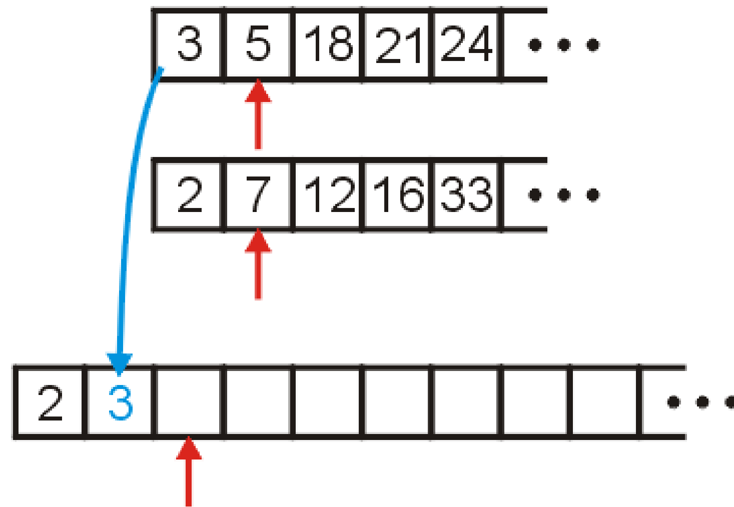
Example

- We compare 2 and 3:   2 < 3
- Copy 2 down
- Increment the corresponding indices



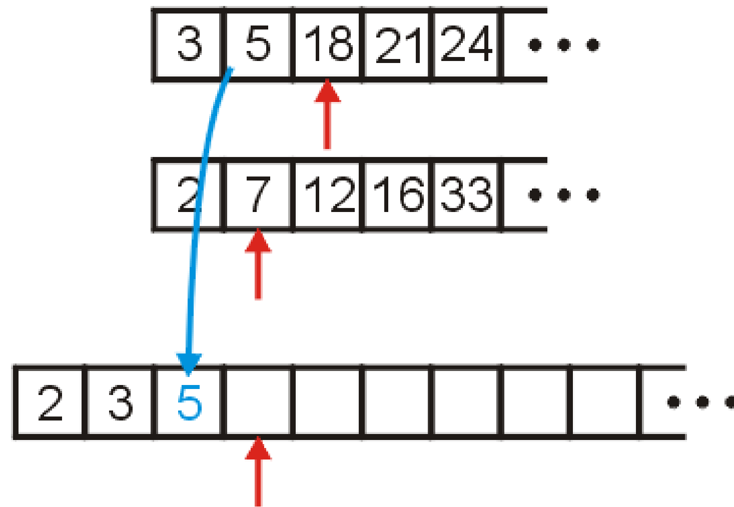
Example

- We compare 3 and 7
- Copy 3 down
- Increment the corresponding indices



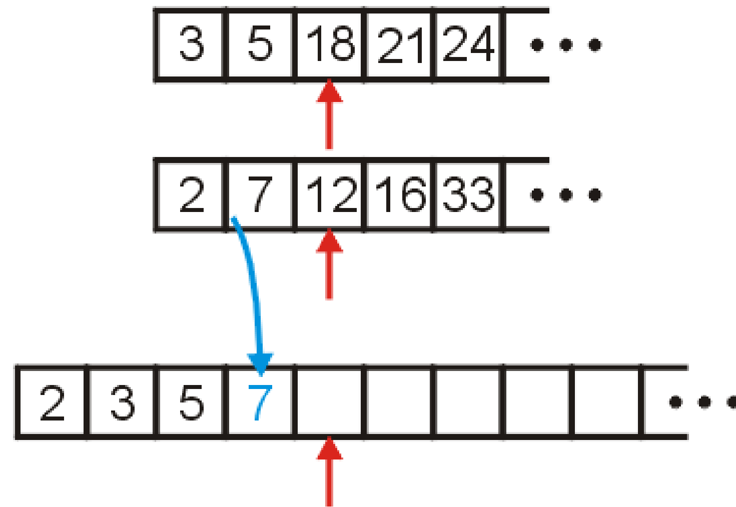
Example

- We compare 5 and 7
- Copy 5 down
- Increment the appropriate indices



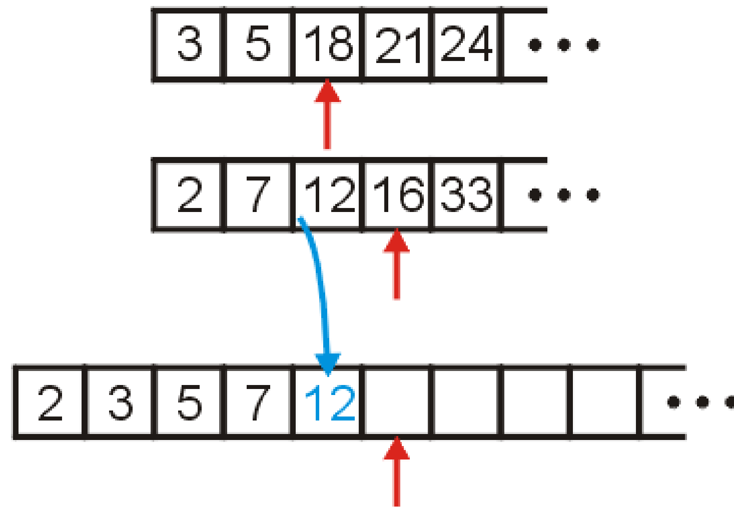
Example

- We compare 18 and 7
- Copy 7 down
- Increment...



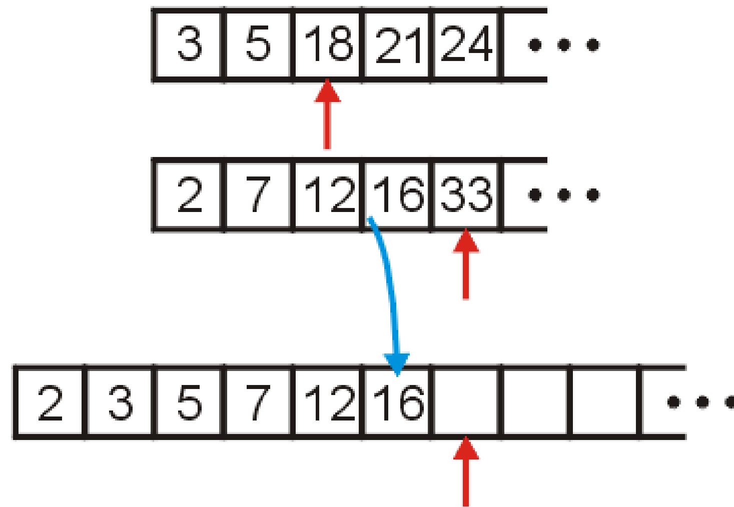
Example

- We compare 18 and 12
- Copy 12 down
- Increment...



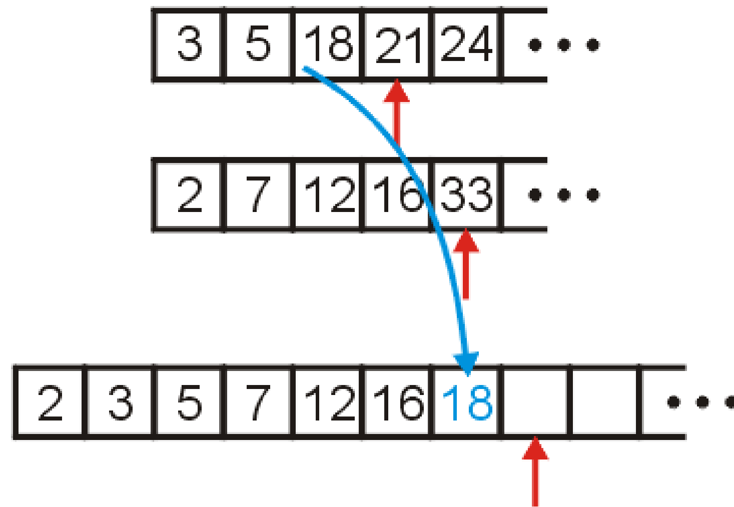
Example

- We compare 18 and 16
- Copy 16 down
- Increment...



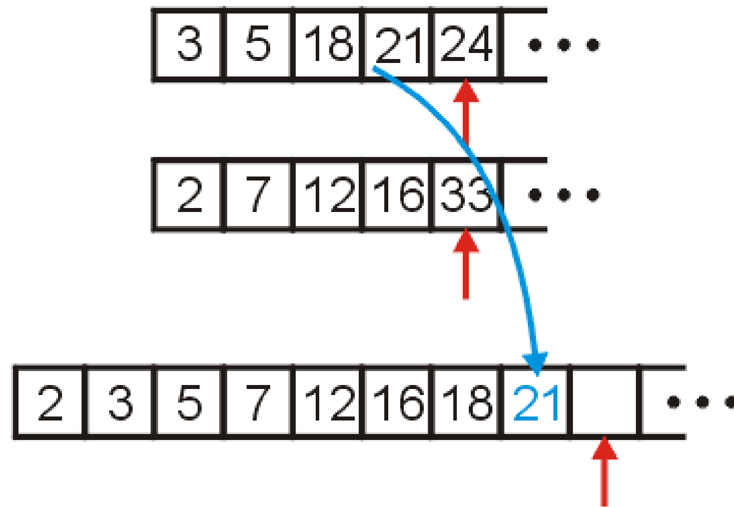
Example

- We compare 18 and 33
- Copy 18 down
- Increment...



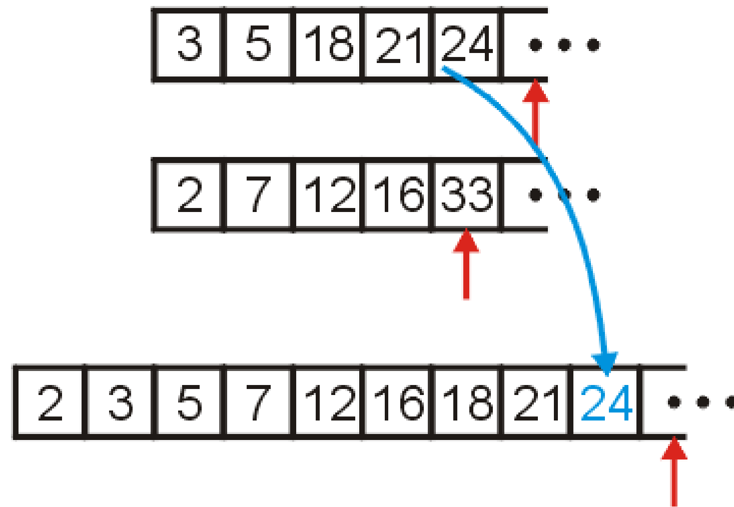
Example

- We compare 21 and 33
- Copy 21 down
- Increment...



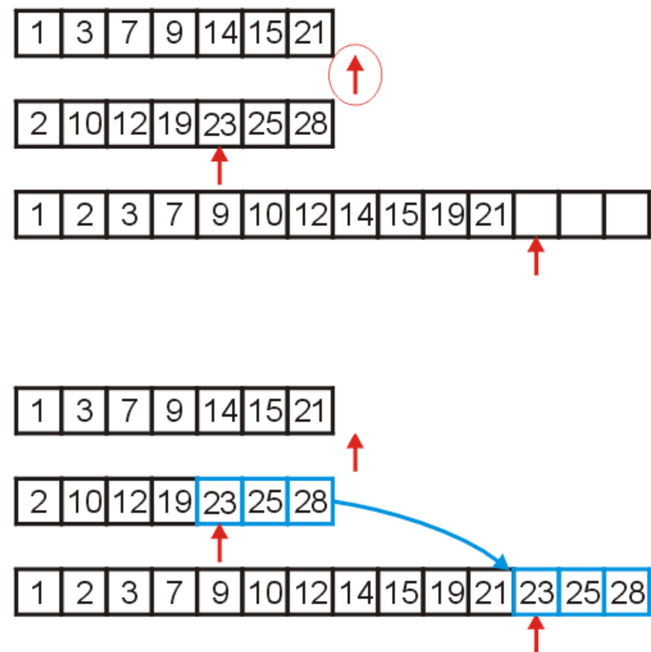
Example

- We compare 24 and 33
- Copy 24 down
- Increment...



Example

- We would continue until we have passed beyond the limit of one of the two arrays
- After this, we simply copy over all remaining entries in the non-empty array



Merging Two Lists

- Programming a merge is straight-forward:
  - the sorted arrays, `array1` and `array2`, are of size `n1` and `n2`, respectively, and
  - we have an empty array, `arrayout`, of size `n1 + n2`
- Define three variables

```
int in1 = 0, in2 = 0, out = 0;
```

which index into these three arrays

Merging Two Lists

- We can then run the following loop:

```
#include <cassert>
//...
int in1 = 0, in2 = 0, out = 0;

while ( in1 < n1 && in2 < n2 ) {
    if ( array1[in1] < array2[in2] ) {
        arrayout[out] = array1[in1];
        ++in1;
    } else {
        assert( array1[in1] >= array2[in2] );
        arrayout[out] = array2[in2];
        ++in2;
    }
    ++out;
}
```

Merging Two Lists

- We’re not finished yet, we have to empty out the remaining array

```
for ( /* empty */ ; in1 < n1; ++in1, ++out ) {
    arrayout[out] = array1[in1];
}

for ( /* empty */ ; in2 < n2; ++in2, ++out ) {
    arrayout[out] = array2[in2];
}
```

Run-time Analysis of Merging

- Assume that the sum of the length of both lists being merged is n
- The statement `++out` will only be run at most n times
- Therefore, the body of the loops run exactly n times
- Hence, merging may be performed in (n) time

Merge Sort

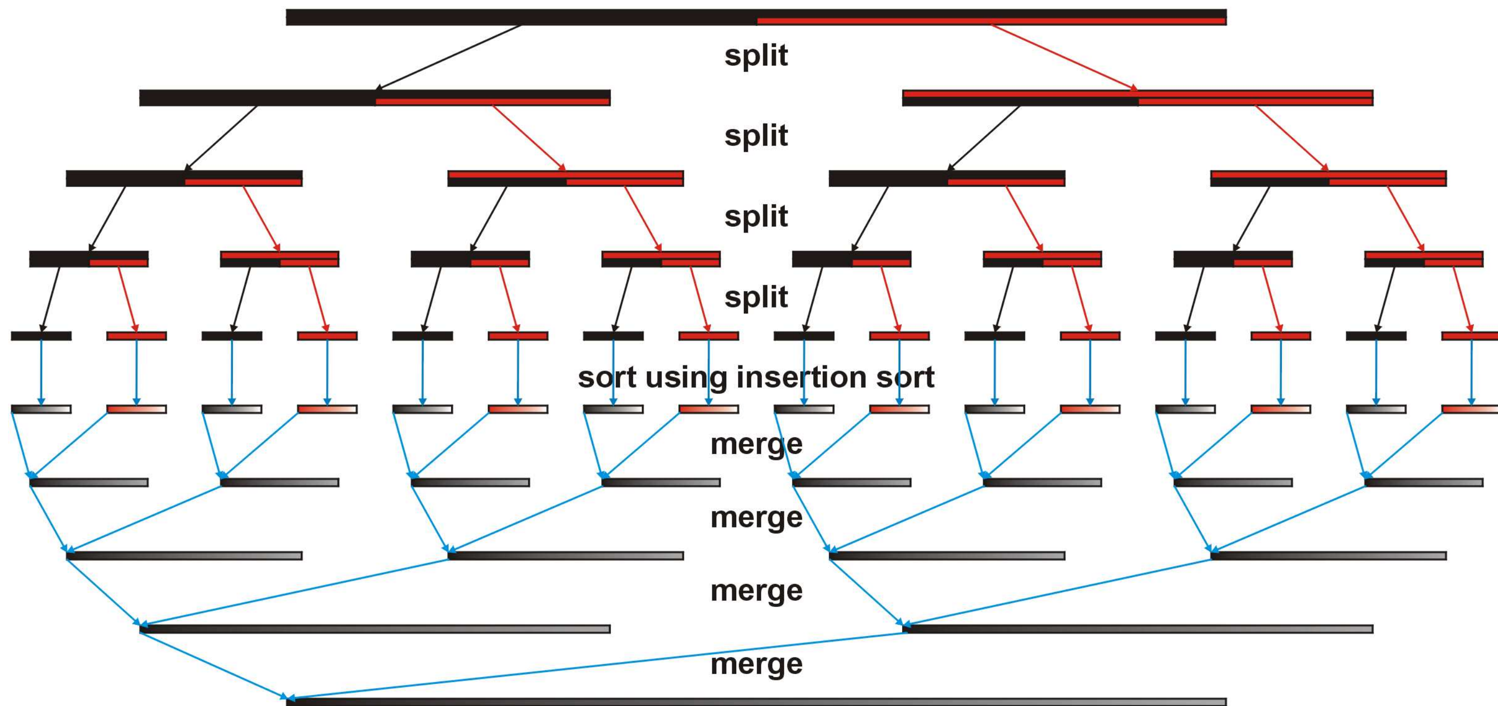
- Question:
  - we split the list into two sub-lists and sorted them
  - how should we sort those lists?
- Answer (theoretical):
  - if the size of these sub-lists is > 1, use merge sort again
  - if the sub-lists are of length 1, do nothing: a list of length one is sorted

Merge Sort

- However, just because an algorithm has excellent asymptotic properties, this does not mean that it is practical at all levels
- Answer (practical):
  - If the sub-lists are less than some threshold length, use an algorithm like insertion sort to sort the lists
  - Otherwise, use merge sort, again

Merge Sort

- Thus, a graphical interpretation of merge sort would be

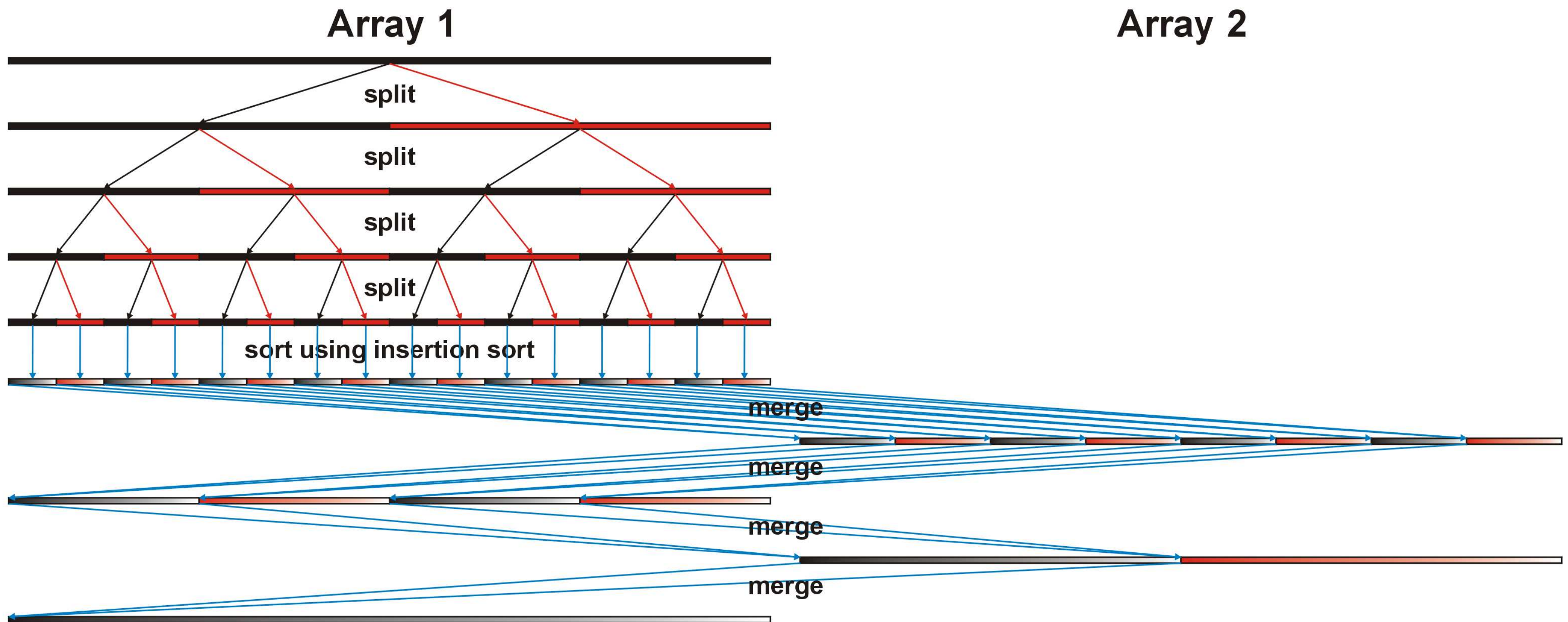


Merge Sort

- Some details:
  - if the list size is odd, just split the array into two almost equally sized list – one even, one odd
  - each merging requires an additional array
    - we can minimize the amount of memory required by using two arrays, splitting and sorting in one, then merging the results between the two arrays

Merge Sort

- Merge sort using two arrays



Example

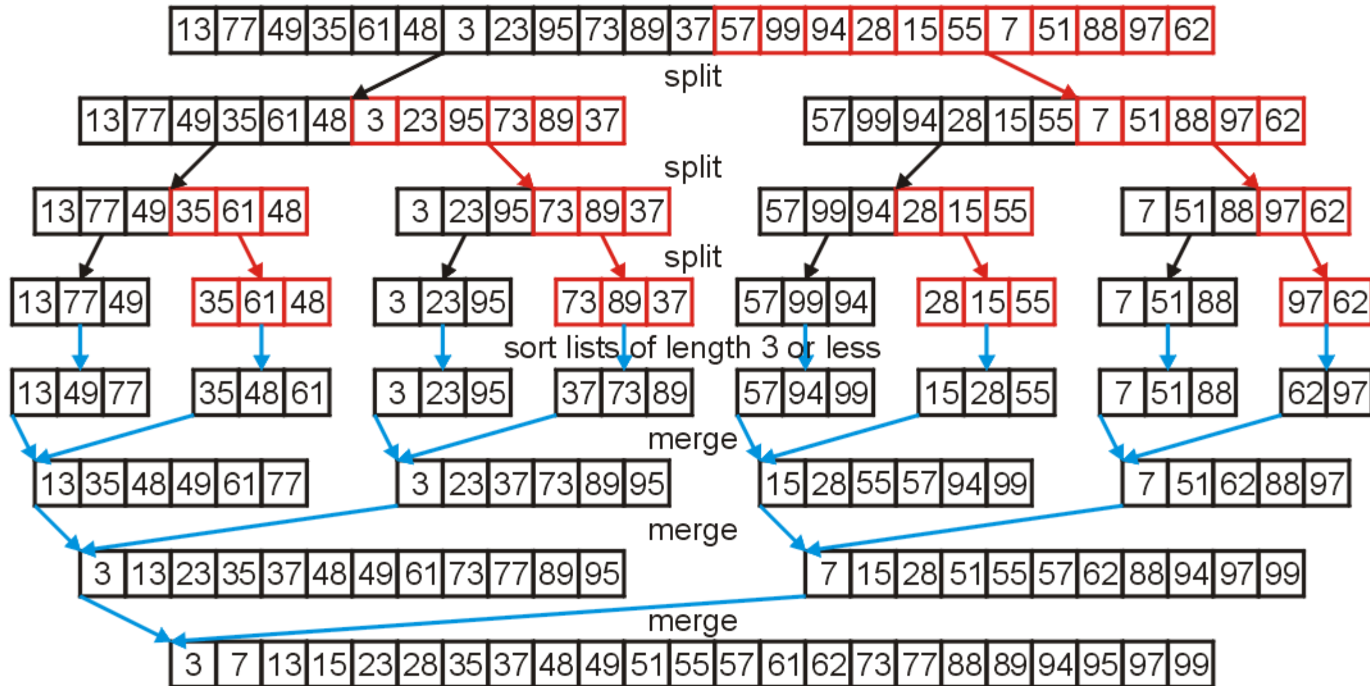
- Consider the following is of unsorted numbers

13 77 49 35 61 48 3 23 95 73 89 37 57 99 94 28 15 55 7 51 88 97 62

- Sorting these using merge sort is relatively straight-forward, as the next slide shows

Example

- Applying the merge sort algorithm:



Run-time Analysis of Merge Sort

- Thus, the time required to sort an array of size $n > 1$ is:
  - the time required to sort the first half,
  - the time required to sort the second half, and
  - the time required to merge the two lists
- That is:

$$\mathrm{T}(n) \quad \begin{array}{lll} \boldsymbol{\Theta}(1) & n & 1 \\ 2\,\mathrm{T}\left(\frac{n}{2}\right) \quad \boldsymbol{\Theta}(n) & n & 1 \end{array}$$

Run-time Analysis of Merge Sort

- We have solved this type of problem before: assume $n = 2^k$ for some $k > 0$
- Therefore, we have:

$$\begin{aligned} T(n) &= T(2^k) \\ &= 2\,T\left(\frac{2^k}{2}\right) + 2^k \\ &= 2\,T\left(2^{k-1}\right) + 2^k \end{aligned}$$

Run-time Analysis of Merge Sort

- Repeating this, we have:

$$
\begin{aligned}
\mathrm{T}(n) &= 2\,\mathrm{T}\left(2^{k-1}\right) + 2^k \\
&= 2\left(2\,\mathrm{T}\left(\frac{2^{k-1}}{2}\right) + 2^{k-1}\right) + 2^k \\
&= 2^2\,\mathrm{T}\left(2^{k-2}\right) + 2^k + 2^k \\
&= 2^2\,\mathrm{T}\left(2^{k-2}\right) + 2 \cdot 2^k
\end{aligned}
$$

Run-time Analysis of Merge Sort

- A third time, we get:

$$
\begin{aligned}
T(n) &= 2^2 T(2^{k-1}) + 2 \cdot 2^k \\
&= 2^2\left(2T\left(\frac{2^{k-2}}{2}\right) + 2^{k-1}\right) + 2 \cdot 2^k \\
&= 2^3 T(2^{k-3}) + 2^k + 2 \cdot 2^k \\
&= 2^3 T(2^{k-3}) + 3 \cdot 2^k
\end{aligned}
$$

- Thus we note a pattern...

Run-time Analysis of Merge Sort

- Noting the pattern, we assume that if we repeat this process k times, we get:

$$
\begin{aligned}
\mathrm{T}(n) \quad & 2^k\ \mathrm{T}\ 2^{\tilde{k}}\ ^{k} \quad k\ \ 2^k \\
& 2^k\ \mathrm{T}\ 2^0 \quad k\ \ 2^k \\
& 2^k\ \mathrm{T}\ 1 \quad k\ \ 2^k \\
& 2^k\ \ 1 \quad k\ \ 2^k
\end{aligned}
$$

Run-time Analysis of Merge Sort

- Recall that by assumption, $n = 2^k$, and therefore $k = \log_2(n)$
- Therefore

$$
\begin{aligned}
\mathrm{T}(n) \quad & \mathbf{\Theta}\ 2^k \quad k\ \ 2^k \\
& \mathbf{\Theta}(n \quad \log_2(n)n) \\
& \mathbf{\Theta}(n \ln(n))
\end{aligned}
$$

Summary

- Thus, merge sort:
  - divides an unsorted list into two equal or nearly equal sub lists,
  - sorts each of the sub lists by calling itself recursively, and then
  - merges the two sub lists together to form a sorted list

Run-time Summary

- The following table summarizes the run-times of merge sort

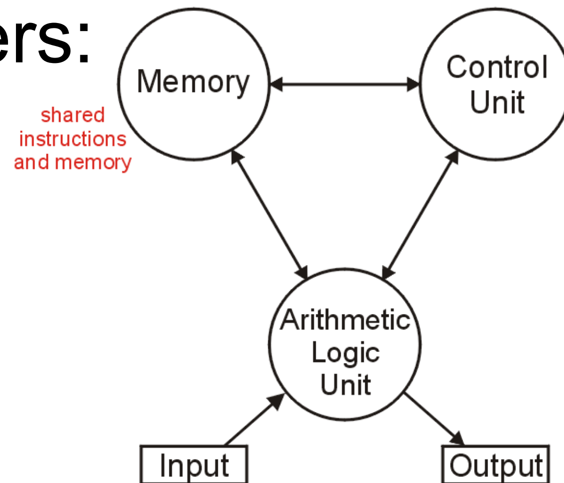
Case	Run Time	Comments
Worst	$(n \ln(n))$	No worst case
Average	$(n \ln(n))$	
Best	$(n \ln(n))$	No best case

Comments

- In practice, merge sort is faster than heap sort, though they both have the same asymptotic run times
- Merge sort requires an additional array, something which heap sort does not require
- Quick sort falls in between w.r.t. time but does not require $\mathbf{O}(n)$ additional memory

Merge Sort

- The (likely) first implementation of merge sort was on the ENIAC in 1945 by John von Neumann
  - the creator of the *von Neumann architecture* used by all modern computers:



http://en.wikipedia.org/wiki/Von_Neumann

Usage Notes

- These slides are made publicly available on the web for anyone to use
- If you choose to use them, or a part thereof, for a course at another institution, I ask only three things:
  - that you inform me that you are using the slides,
  - that you acknowledge my work, and
  - that you alert me of any mistakes which I made or changes which you make, and allow me the option of incorporating such changes (with an acknowledgment) in my set of slides

Sincerely,

Douglas Wilhelm Harder, MMath

dwharder@alumni.uwaterloo.ca

Quick Sort

Divide: Partition the array into two sub-arrays

A[p . . q-1] and A[q+1 . . r] such that each element of

A[p . . q-1] is less than or equal to A[q], which in turn

less than or equal to each element of A[q+1 . . r]

Quick Sort

Conquer: Sort the two sub-arrays A[p . . q-1] and

A[q+1 . . r] by recursive calls to quick sort.

Quick Sort

Combine: Since the sub-arrays are sorted in place, no

work is needed to combine them.

Quick Sort

```
QUICKSORT(A, p, r)

if p< r

   then q ← PARTITION(A, p, r)

            QUICKSORT(A, p, q-1)

            QUICKSORT(A, q+1, r)
```

Quick Sort

PARTITION(A, p, r)

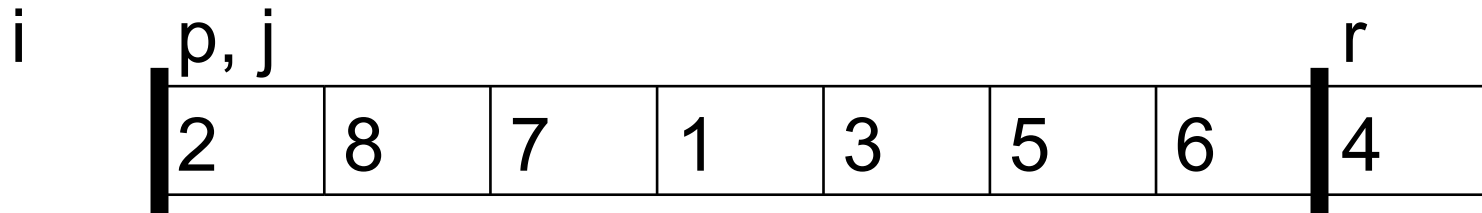
x ← A[r]

i ← p-1

Quick Sort

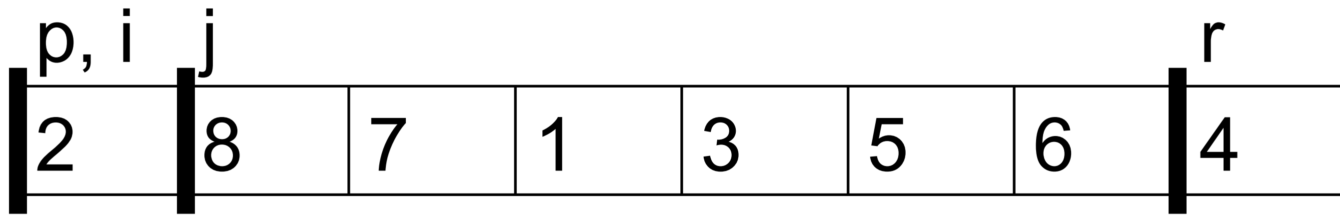
```
for j ← p to r-1
    do if A[j] <= x
        then i ←i+1
            exchange A[i] ← → A[j]
exchange A[i+1] ← → A[r]
return i+1
```

Quick Sort



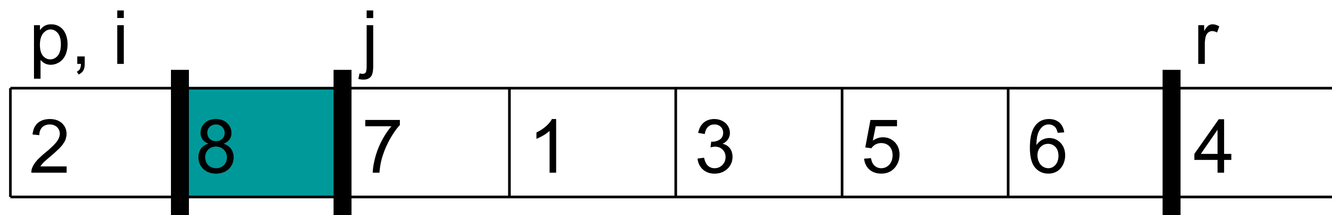
(a)

Quick Sort



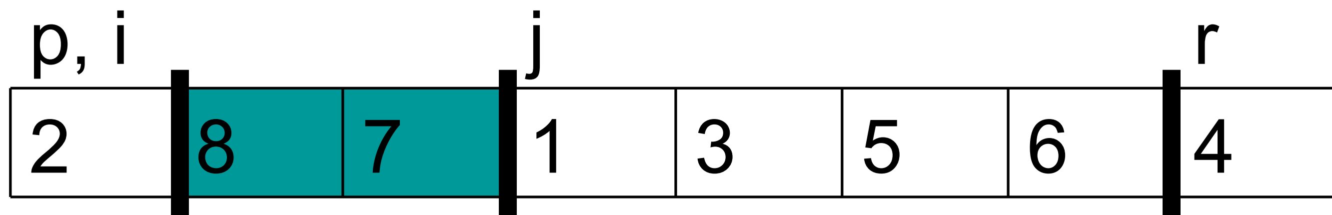
(b)

Quick Sort



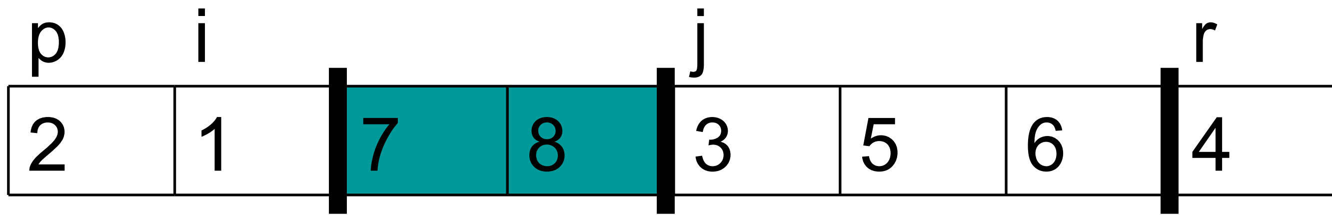
(c)

Quick Sort



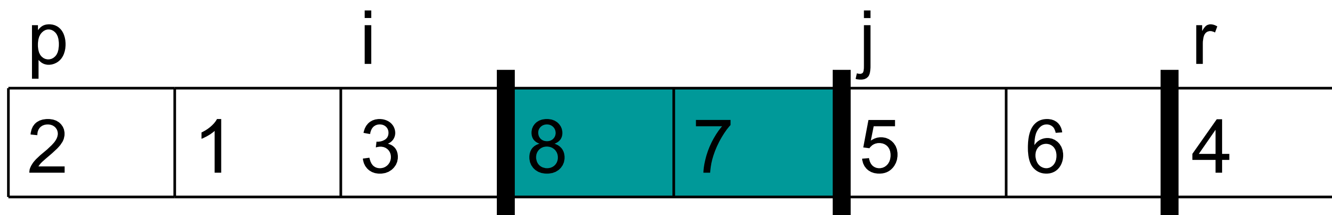
(d)

Quick Sort



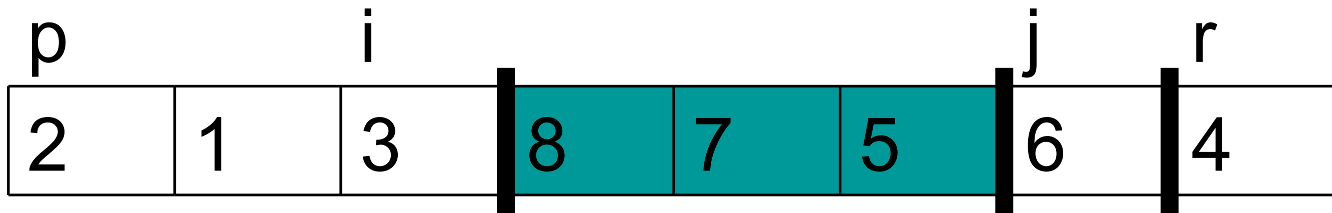
(e)

Quick Sort



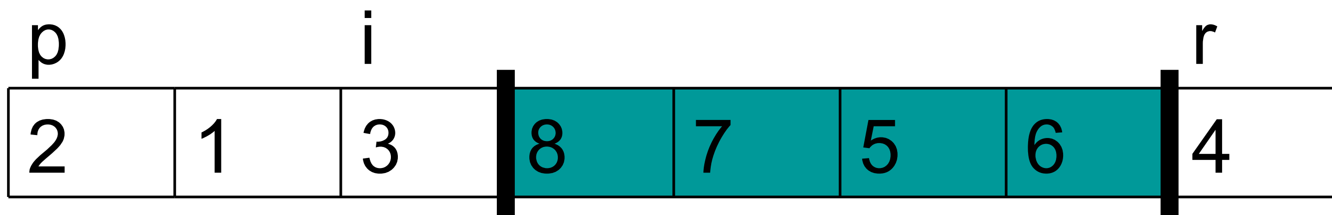
(f)

Quick Sort



(g)

Quick Sort



(h)

Quick Sort

(i)

Quick Sort

Worst-case partitioning:

The partitioning routine produces one sub-problem

with n-1 elements and another sub-problem with 0

elements. So the partitioning costs $\theta(n)$ time.

Quick Sort

Worst-case partitioning:

The recurrence for the running time

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

$=T(n-1) + \theta(n)$

$=$------------------ $\theta(n^2)$

Quick Sort

Worst-case partitioning:

The $\theta(n^2)$ running time occurs when the input array is already completely sorted – a common situation in which insertion sort runs in $O(n)$ time

Quick Sort

Best-case partitioning:

The partitioning procedure produces two

sub-problems, each of size not more than n/

2.

Quick Sort

Best-case partitioning:

The recurrence for the running time

$$T(n) <= 2T(n/2) + \theta(n)$$

$$= \text{-----} \ O(n \lg n)$$

Quick Sort

Best-case partitioning:

The equal balancing of the two sides of the partition at every level of the recursion produces faster algorithm.

Quick Sort

Balanced partitioning:

Suppose, the partitioning algorithm always

produces 9-to-1 proportional split, which

seems quite unbalanced.

Quick Sort

Balanced partitioning:

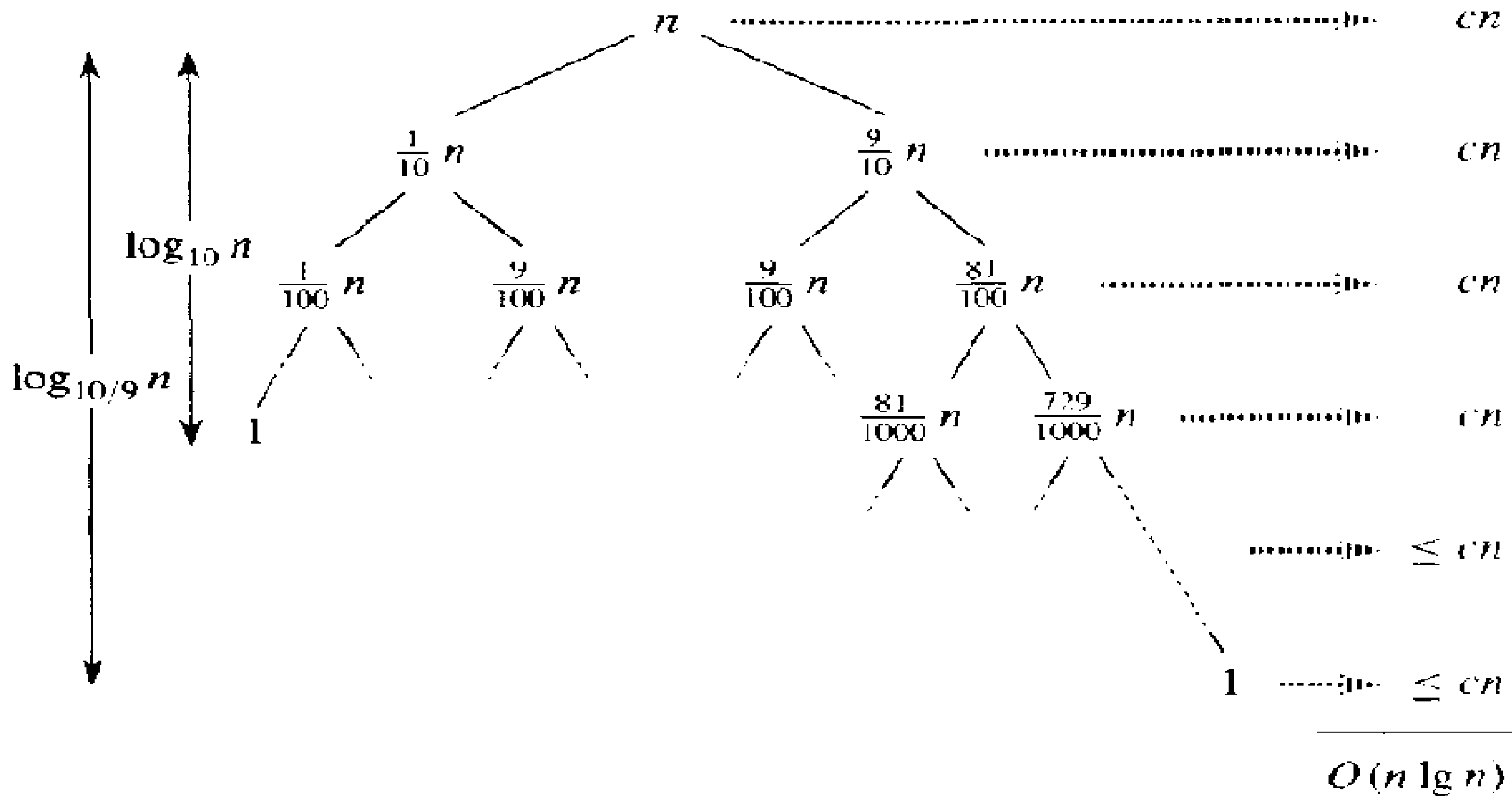
The recurrence for the running time

$T(n) <= T(9n/10) + T(n/10) + cn$

= ------------- $O(n \lg n)$

Quick Sort

Balanced partitioning: The recursion tree



Quick Sort

Balanced partitioning:

In fact, a 99-to-1 split yields an $O(n \lg n)$ running time. Any split of constant proportionality yields a recursion tree of depth $\theta(\lg n)$

Quick Sort

Intuition for the average case:

It is unlikely that the partitioning always happens

in the same way at every level.

Quick Sort

Intuition for the average case:

In the average case, PARTION produces a mix of

“good” and “bad” splits.

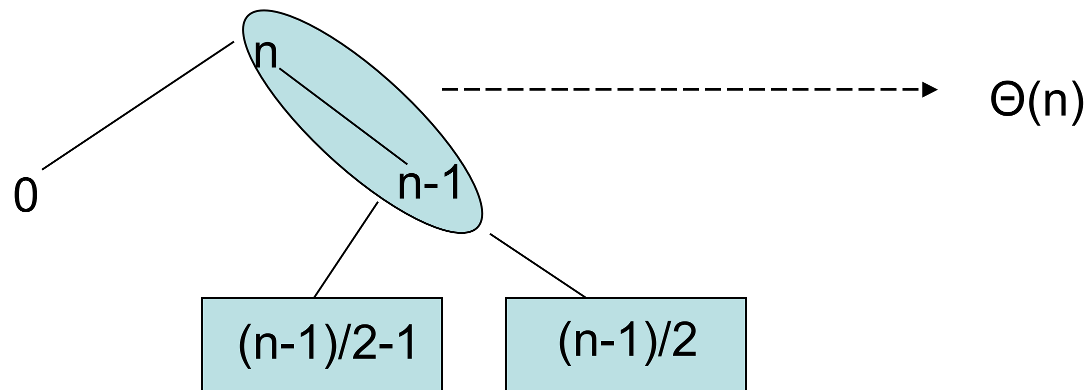
Quick Sort

Intuition for the average case:

The combination of the bad split followed by the good split

produces three arrays of sizes 0, (n-1)/2-1, and (n-1)/2 at a

combined partitioning cost of $\theta(n) + \theta(n-1) = \theta(n)$

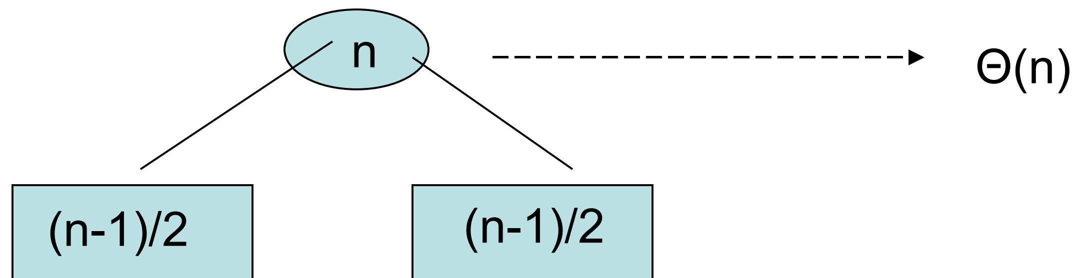


Quick Sort

Intuition for the average case:

A single level of partitioning produces two sub-arrays of size

(n-1)/2 at a cost of θ(n).



A Randomized Version of Quick Sort

Instead of always using A[r] as the pivot, we will use a randomly chosen element from the sub-array A[p..r].

A Randomized Version of Quick Sort

Because the pivot element is randomly chosen,

we expect the split of the input array to be

reasonably well balanced on average.

A Randomized Version of Quick Sort

RANDOMIZED-PARTITION(A, p, r)

i ← RANDOM(p, r)

exchange A[r] ← → A[i]

return PARTITION(A, p, r)

A Randomized Version of Quick Sort

```
RANDOMIZED-QUICKSORT(A, p, r)

if p<r then

        q ← RANDOMIZED-PARTITION(A, p, r)

        RANDOMIZED-QUICKSORT(A, p, q-1)

        RANDOMIZED-QUICKSORT(A, q+1, r)
```