Chapter 3:Sorting Algorithms

Comparison Based Sorting

1. Smallest Based Sort
2. Insertion Sort
3. MergeSort
4. QuickSort
5. HeapSort
6. Shell Sort

Non-Comparision Based Sorting Algorithms

1. Bucket Sorting Algorithms
2. Radix Sort

Smallest Based Sort: Introduction

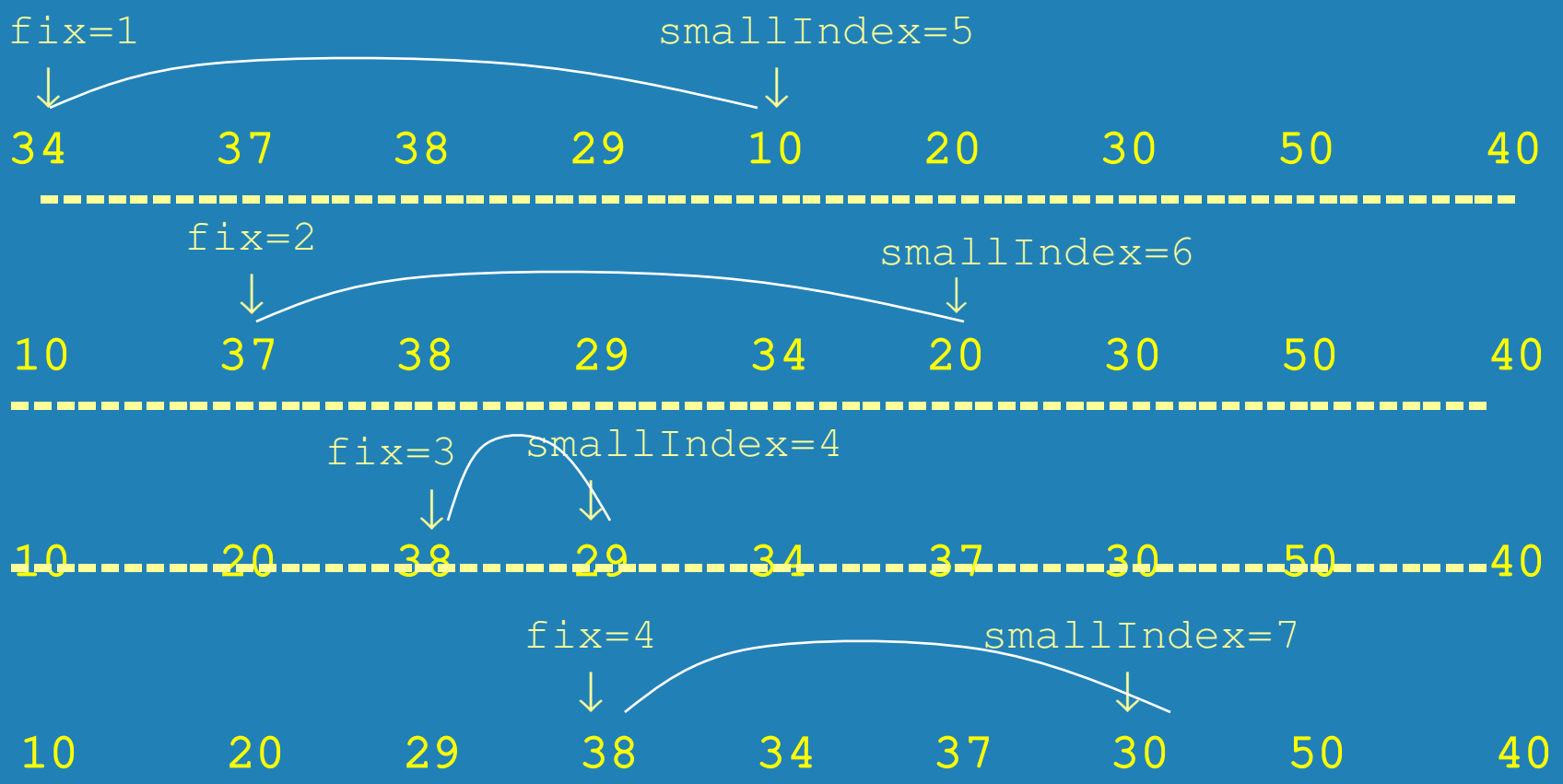
The general idea is that smallest element is swapped with the element at the first position, the next smallest is swapped with the element at the second position etc. This is a *brute force* technique

Initially, both *fix* and *smallIndex* point at the first number. *smallestBasedSort* then checks which numbers from A[1] to the end is the smallest and returns the *smallIndex*. Then, the numbers in *fix* and *smallIndex* are swapped.

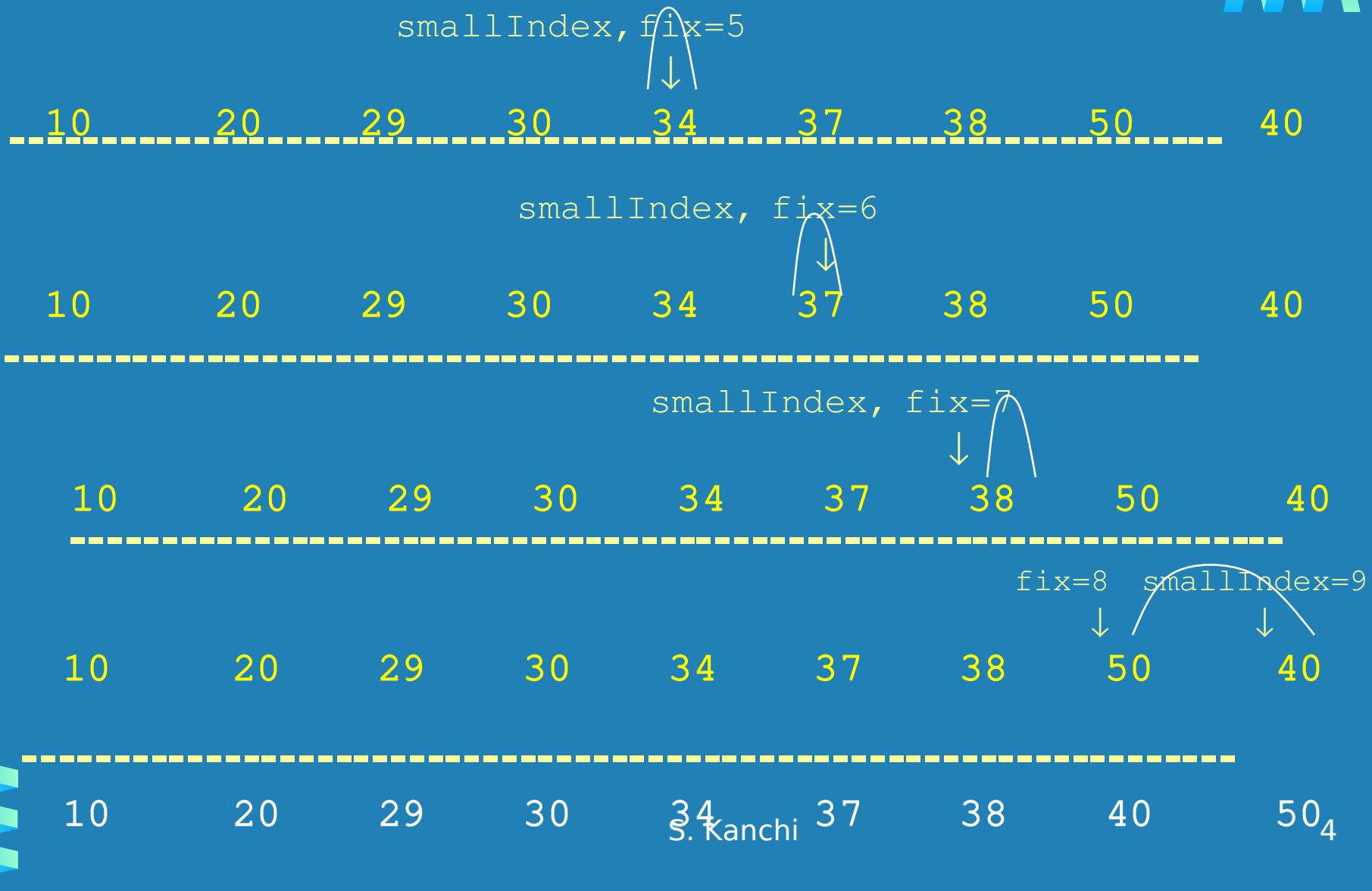
After that *fix* points to the next number, A[2], and then smallestBasedSort looks for the smallest number from A[2] to the end. Then *smallestBasedSort* swaps *smallIndex* and *fix*.

The whole search and swap process continues until *fix* reaches $(n-1)^{st}$ position.

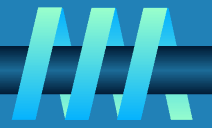
Smallest Based Sort:Demonstration-1



Smallest Based Sort:Demonstration-2



Notation

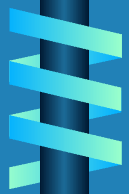


We will assume that

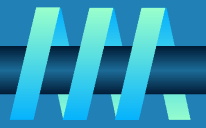
Index = (0,1, 2, …n)

Data is the type of data to be sorted

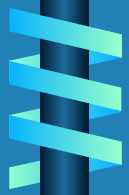
ArrayOfData is an array of Data .



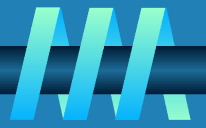
Smallest Based Sort: Algorithm



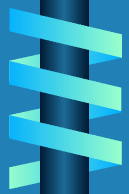
```
procedure smallestBasedSort(var A: ArrayOfData,
n:Index)
var fix, smallIndex:Index;
begin
for fix  := 1 to n-1 do
      smallIndex := findSmallestIndex(A,fix,n);
      swap(A, smallIndex, fix);
   endfor
end;
```



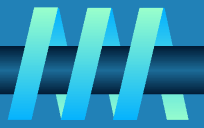
Smallest Based Sort: Algorithm



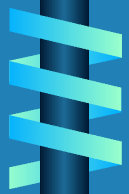
```
procedure findSmallestIndex(A:arrayOfData,low:
integer, high: integer):Index
var i, smallIndex:Index;
begin

      smallIndex := low;
      for i := low + 1 to high do
          if (A[i] < A[smallIndex]) then
             smallIndex := i;
          endif;
      endfor
      findSmallestIndex := smallIndex;
end;
```



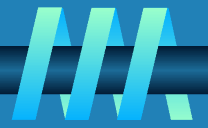
Smallest Based Sort: Algorithm



```
procedure swap(var A:ArrayOfData,i:Index,j:Index)
var temp:Data;
begin
temp := A[i];
        A[i] := A[j];
        A[j] := temp;
end;
```



Smallest Based Sort: Analysis



Time Complexity of Swap: Time swap is O(1) because it takes only three assignment statements to swap two elements.

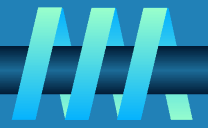
Time Complexity of findSmallestIndex: The actual time is O(high-low+1)

Time Complexity of smallestBasedSort: It has loop that executes n-1 times and each time a swap (1 time) and a findSmallestIndex(n-i+1) is done. The time is therefore

$$\sum_{i=1}^{n-1} n-i+2 = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 2$$

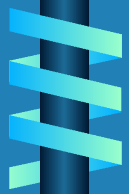
$$n(n-1) - \frac{n(n-1)}{2} + 2n - 2 = \frac{n(n-1)}{2} + 2n - 2 = O(n^2)$$

Smallest Based Sort: Analysis

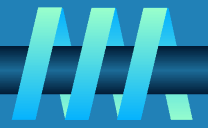


Since the smallestBasedSort loops n-1 time irrespective of the input, the best case, worst case and average case are all same.

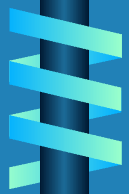
Also, note that nowhere in the process we are using any memory other than some fixed number of additional variables. Therefore memory complexity is O(1).



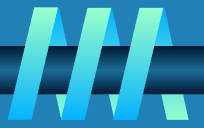
Time Complexity Table



	Memory Complexity	Worst Case Time Complexity	Average Case Time Complexity	Best Case Time Complexity
Smallest Based Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n^2)$



Insertion Sort: Introduction

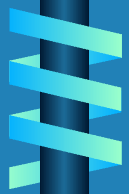


Insertion sort uses a *decrease-and-conquer* technique. Insertion sort works by inserting every element by comparing it with previous elements. First, second number is compared with first and if needed it is swapped with the first. So, the first number and second number are in the right place relative to each other. Now, the third number compared to second number and then the first number and inserted in position one, two or three. This process continues until all numbers are put in the right place.

12 29 34 38 13 20 30 50 40

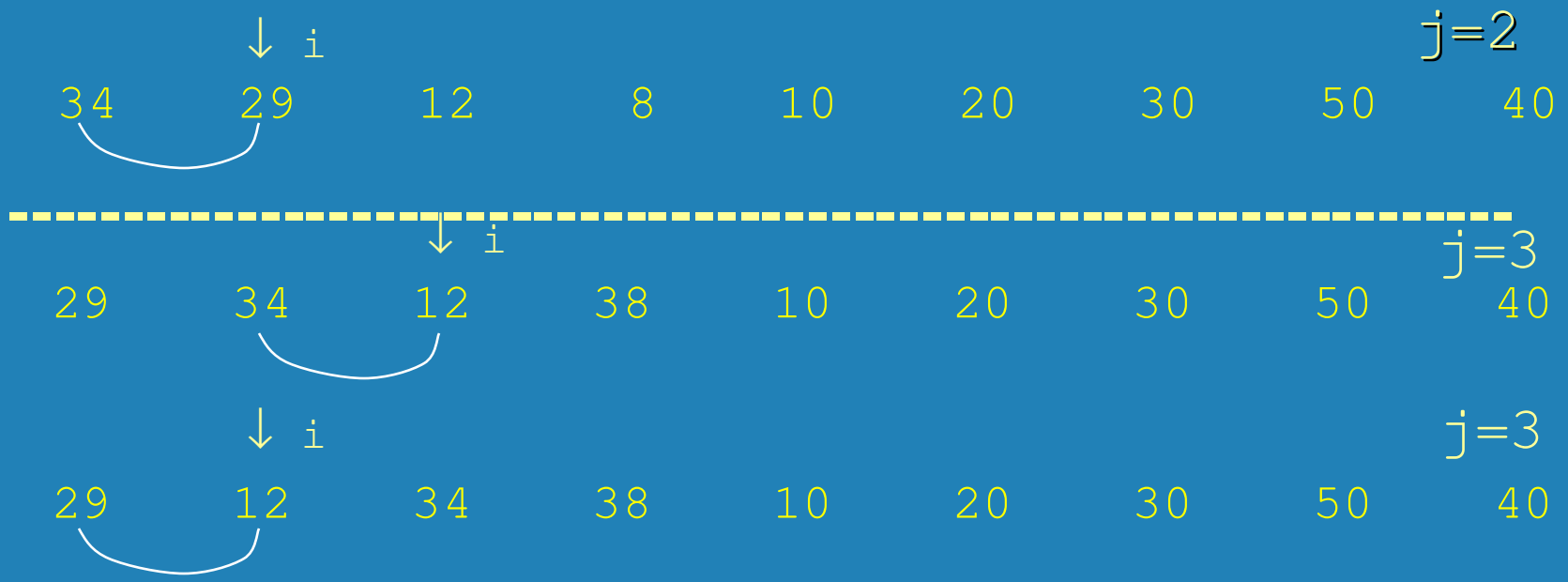
13 is put in position number 2, by moving 38, 34 and 29 to the right

12 13 29 34 38 20 30 50 40

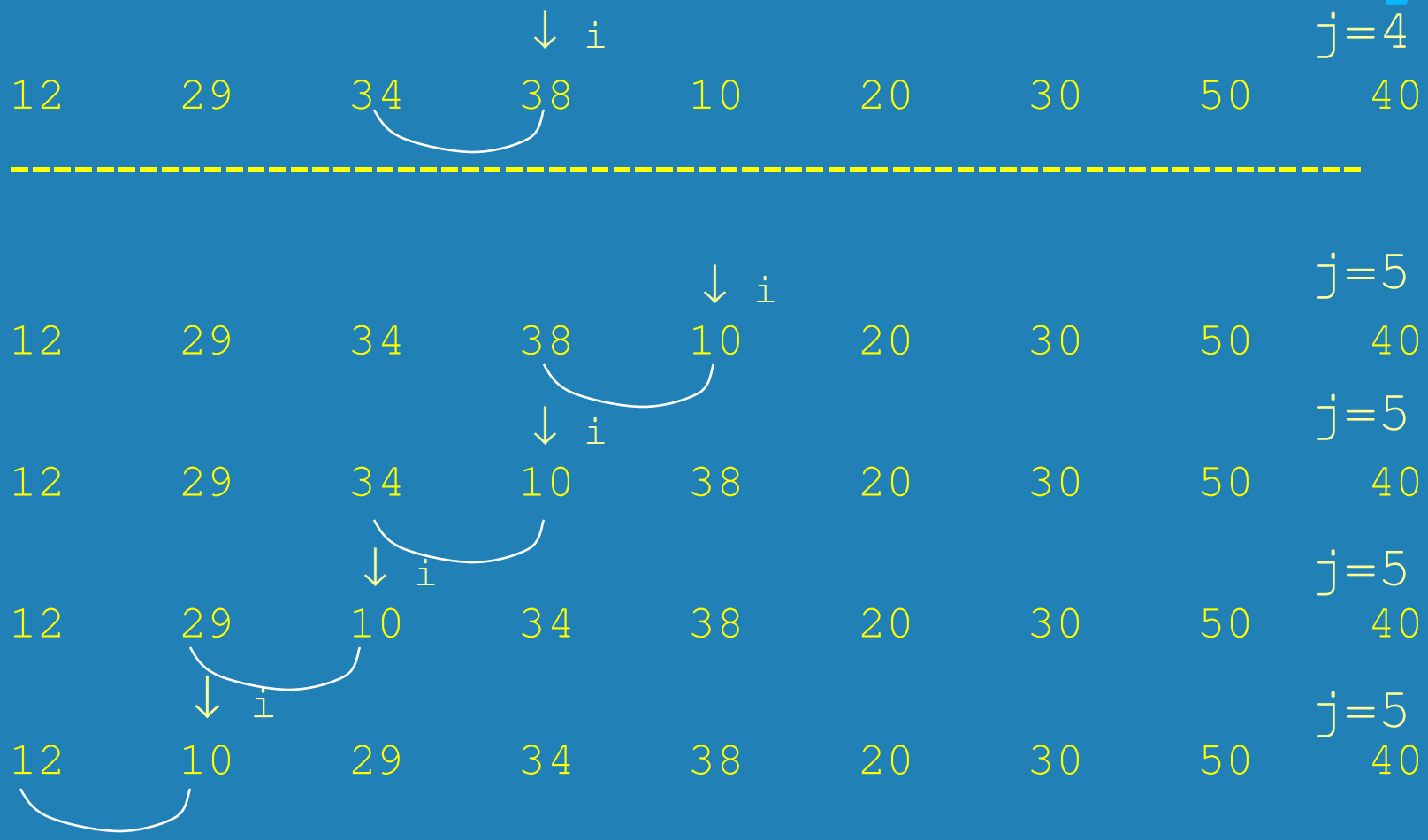


Insertion Sort :Demonstration-1

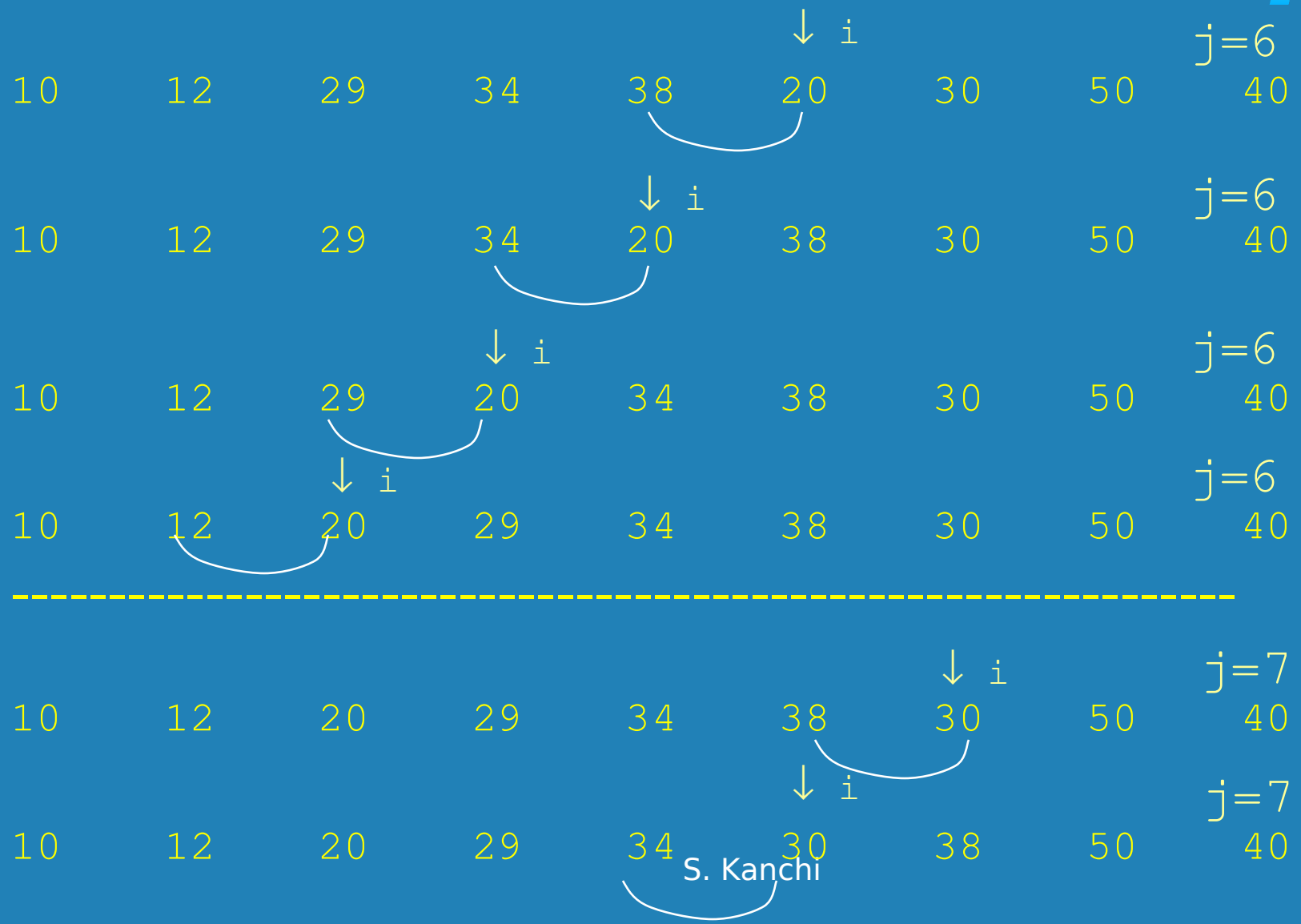
The idea of insertion sort is to insert elements if they are in the wrong place.



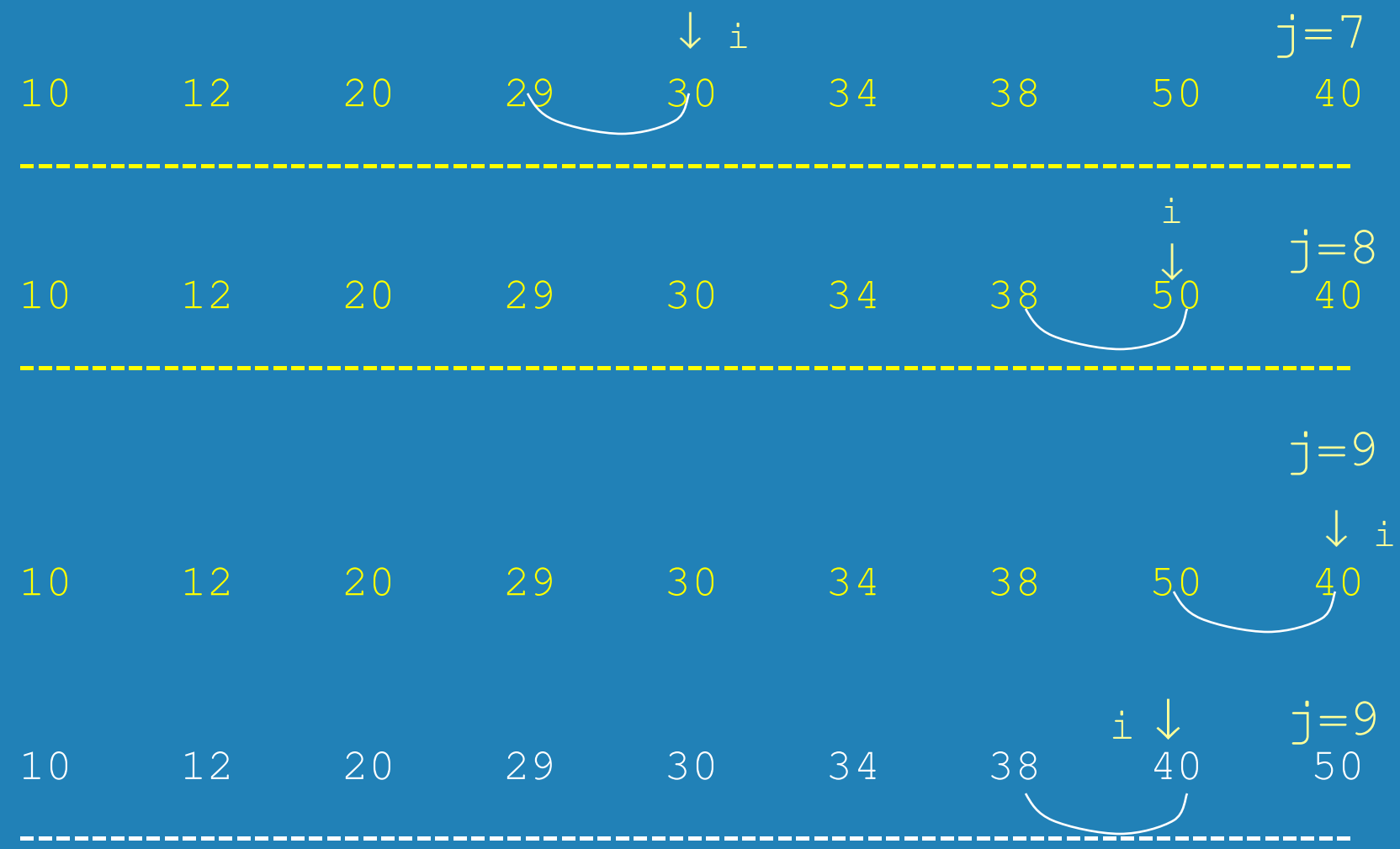
Insertion Sort :Demonstration-2



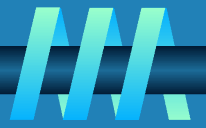
Insertion Sort :Demonstration-3



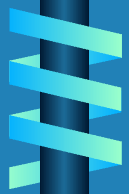
Insertion Sort :Demonstration-4



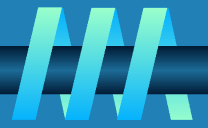
Insertion Sort: Algorithm



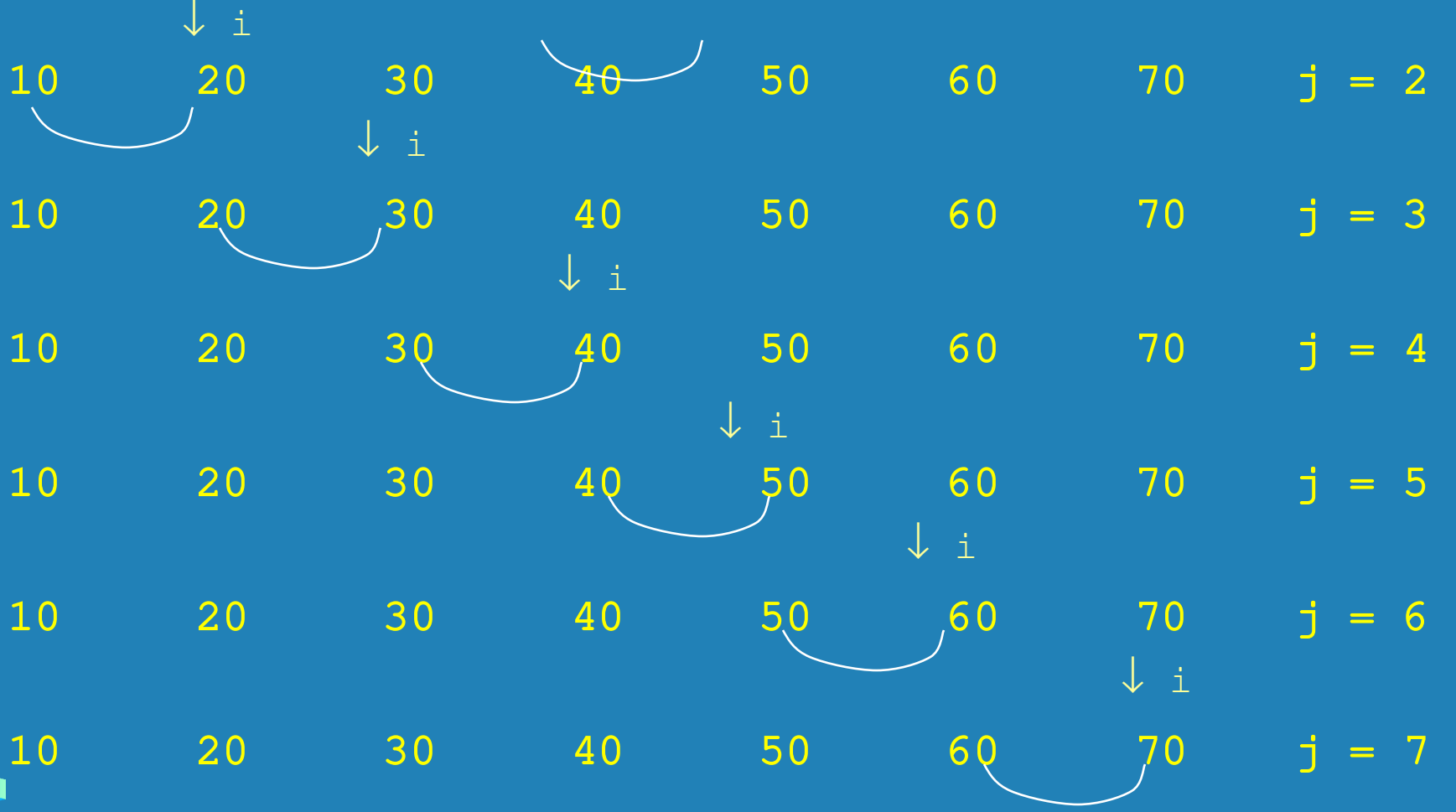
```
// Assume array A[1..n] is global

procedure insertionSort()
begin
     var i:Index;
     for j := 2 to n do begin
       i := j;
       while (i > 1 and (A[i] < A[i-1]) ) do
         swap (A, i, i-1);
         i := i-1;
       endwhile
     endfor
end;
```



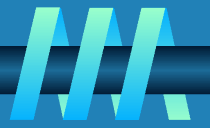
Insertion Sort: Best Case Analysis



Best Case Scenario: Array is already sorted

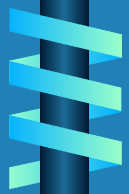


Insertion Sort: Best Case Analysis

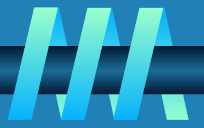


In the best case, the while loop never executes. Only the for loop executes and it executes n-1 times.

Therefore the best case time complexity is $\Omega(n)$



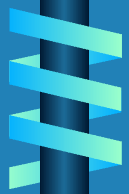
Insertion Sort: Worst Case Analysis



Worst Case: Array is reverse sorted. So in each iteration to fix the j^{th} position there are j comparisons and swaps.

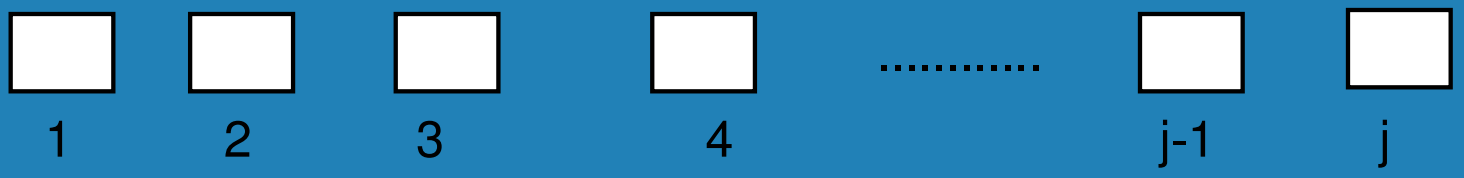
$$\sum_{j=2}^{j=n} j = \sum_{j=1}^{j=n} j - 1 = \frac{n(n+1)}{2} - 1 = O(n^2)$$

Therefore average case analysis is needed.



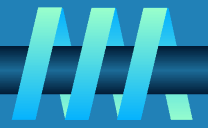
Insertion Sort: Average Case Analysis

The item in position j can go into any one of the positions j, j-1, j-2, …. 1 all events being equally likely.



Prob	1/j	1/j	1/j	1/j	.....	1/j	1/j
Time	j-1	j-1	j-2	j-3	.....	2	1

Insertion Sort Average Analysis

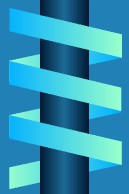


$$Ave(T) = \sum_{j=2}^{n} \frac{1}{j}1 + \frac{1}{j}2 + \frac{1}{j}3 \ldots + \frac{1}{j}(j-1) + \frac{1}{j}(j-1)$$

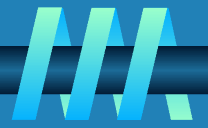
$$= \sum_{j=2}^{n} \sum_{i=1}^{j-1} \frac{1}{j} i + \sum_{j=2}^{n} \frac{1}{j}(j-1)$$

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

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



Insertion Sort: Average Analysis



$$= \frac{1}{2}\left[\frac{n(n+1)}{2} - 1\right] - \frac{1}{2}(n-1) + (n-1) - \sum_{j=2}^{n} \frac{1}{j}$$

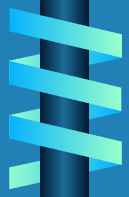
$$= \frac{n^2}{4} + \frac{n}{4} - \frac{1}{2} - \frac{1}{2}n + \frac{1}{2} + n - 1 - \sum_{j=2}^{n} \frac{1}{j}$$

$$= \frac{n^2}{4} + \frac{3}{4}n - 1 - \sum_{j=2}^{n} \frac{1}{j}$$

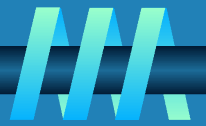
But, $\sum_{j=2}^{n} \frac{1}{j} = \int_{2}^{n} \frac{1}{x}\,dx = \ln \mathbf{x}\Big|_{2}^{n} = \ln n - \ln 2$

$$Ave(T) = \frac{n^2}{4} + \frac{3}{4}n - 1 - \ln n + \ln 2$$

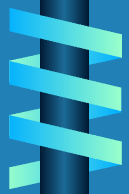
$$Ave(T) = \theta(n^2)$$



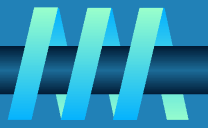
Time Complexity Table



	Memory	Worst	Ave	Best
Smallest Based Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n^2)$
Insertion Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n)$



Merge Sort: Introduction



In merge sort, an array is divided equally into two parts. Each part is sorted and then the two sorted arrays are merged. This is a *divide-and-conquer* technique

For example,

34 43 12 23 67 22

Is divided into two arrays

34 43 12 and 23 67 22

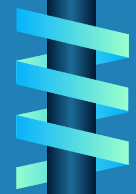
Each is sorted. We get

12 34 43 and 22 23 67.

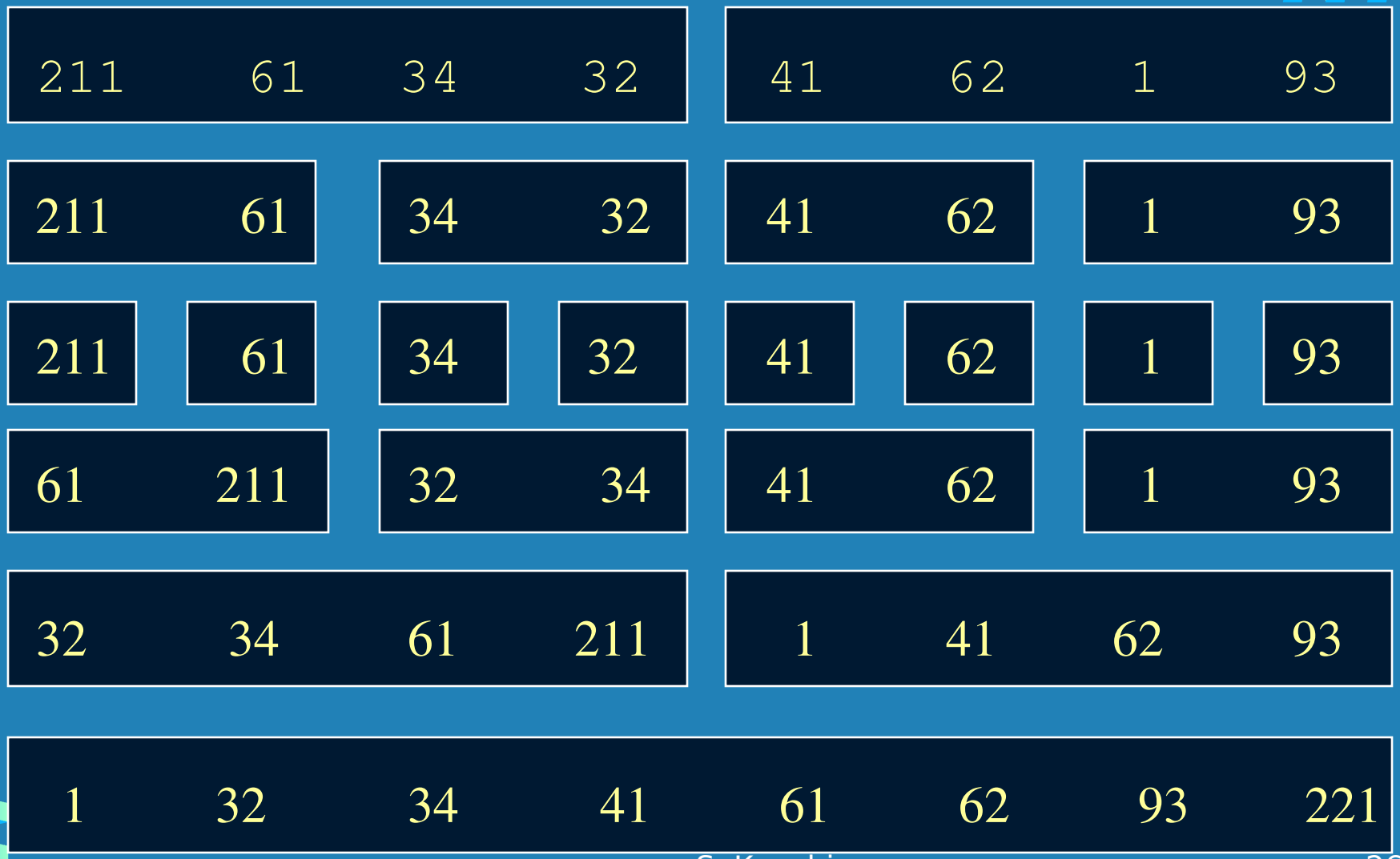
Now the two arrays are merged into

12 22 23 34 43 67.

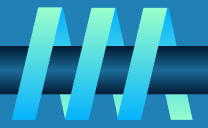
However, each array is sorted using recursive merge sort again



Merge Sort :Demonstration



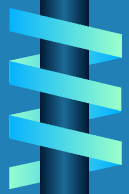
Merge Sort:Algorithm



Assume that array is global.

```
procedure mergeSort (var low: integer,high: integer)
begin
  if (low < high) then
   midPoint := (low + high)/2 ;
   a := mergeSort (low, midPoint);
   b := mergeSort (midPoint + 1, high);
   c := merge (a,b);
  endif
end;
```

Every time there is a merge, an array of size N is created.



Merge Sort: Analysis

The average case, worst case and best case are all the same and is given by the following recurrence relation.

$$T(n)=n+2\mathrm{T}\left(\frac{n}{2}\right), T(1)=1$$

Solving,

$$T(n)=n+2\mathrm{T}\left(\frac{n}{2}\right)$$

$$= \mathrm{n}+2\left[\frac{n}{2}+2\mathrm{T}\left(\frac{n}{4}\right)\right]$$

$$=n+n+2^2T\left(\frac{n}{2^2}\right)$$

$$=2\mathrm{n}+2^2T\left(\frac{n}{2^2}\right)$$

$$=3\mathrm{n}+2^3T\left(\frac{n}{2^3}\right)$$

$$=\ldots\ldots\ldots$$

$$=(\log_2 n)n+2^{\log_2 n}T\left(\frac{n}{2^{\log_2 n}}\right)$$

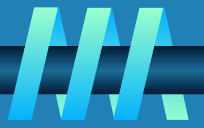
$$=\mathrm{n}\log_2 n+\mathrm{n}\mathrm{T}\left(\frac{n}{n}\right), \text{ since } 2^{\log_2 n}=n$$

$$=\mathrm{n}\log_2 n+\mathrm{n}\mathrm{T}(1)$$

$$=\mathrm{n}\log_2 n+n.1$$

$$T(n)=O(\mathrm{n}\log_2 n)$$

Merge Sort: Analysis

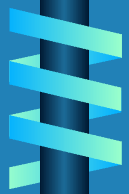


It looks we need a one additional array for each merge step, according to the demonstration.

There are logn merge steps. Therefore memory complexity is O(nlogn).

However, the memory complexity can be reduced to O(n) simply by using 2 arrays in the merge step. [There is an exercise question relating to this.]

Therefore the memory complexity of mergesort is O(n).



Time Complexity Table

Name of the sorting Algorithm	Memory Complexity	Worst Case Time Complexity	Ave	Best
Smallest Based Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n^2)$
Insertion Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n)$
Merge Sort	O(n)	O (n logn)	θ (n logn)	Ω (n logn)

Quick Sort: Introduction

Quick sort uses *Divide and Conquer* technique.

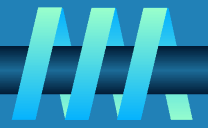
Pick a split point and rearrange the array so that every number to the left side elements are less than the number at split point every number to the right of split point is greater than the number at split point. The numbers on each side need not be in any particular order. For example if 62 is chosen as split point in the array below,

23 411 62 41 1 4

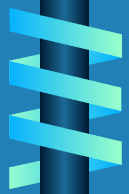
Then the array after the rearrangement becomes,

Now sort the left and the right side using quicksort

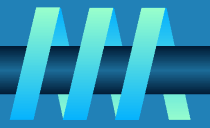
Quick Sort: Algorithm



```
QuickSort (A,low,high)
begin
 if (low<high)then
       splitPoint:=split(A)
        QuickSort(A,low,splitPoint-1);
        QuickSort(A,splitPoint+1,high);
 endif
end;
```



Quick Sort : Preliminary Analysis



Let us analyze the time complexity of Quick sort.

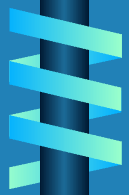
If the split always splits in the middle is done in linear time, the time complexity T(n) of Quicksort is given by:

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

$T(1) = 1$

The solution for the above as we saw in Merge sort analysis is nlogn.

Let us try to develop a split algorithm that takes linear time, whether or not it splits in the “middle”.

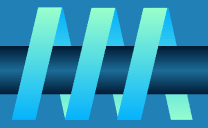


Quick Sort: The split algorithm

Let low be the first index and *high* is the last index. Let us assume that we always want to use A[low], denoted by *x*, to be the number we want to split around. Let *s* be the current split point. Initially s is set to *low+1*. *y* runs from low+1 to high. For each y, if A[y] < x, the elements at s and y are swapped and s is set to s+1. At the end, s is set to s-1 and elements at low and s are swapped.

A[y]<x,swap(s,y),s:=s+1

Split Algorithm: Demonstration-1

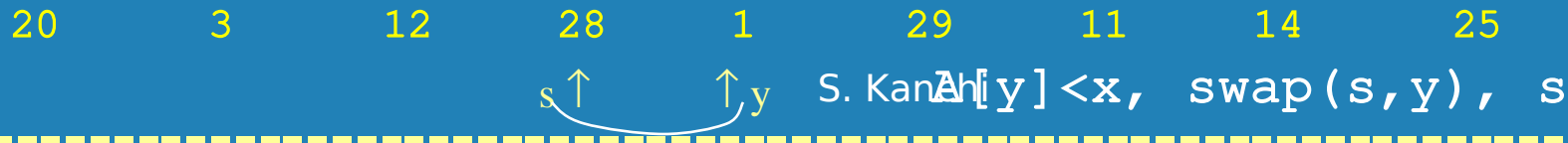


x=20

A[y]<x,swap(s,y),s:=s+1

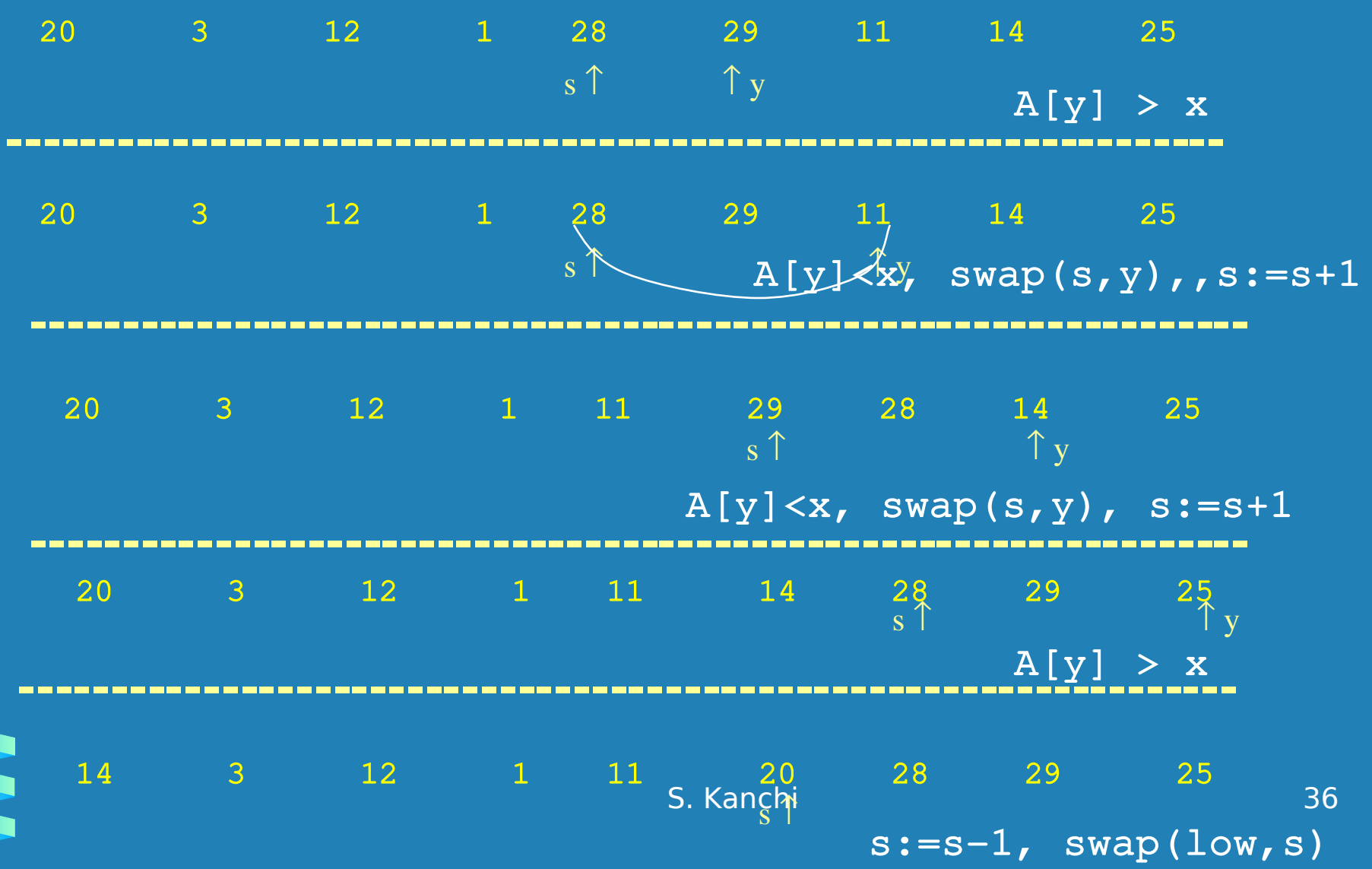
A[y] > x

A[y]<x, swap(s,y), s:=s+1

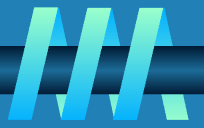


A[y]<x, swap(s,y), s:=s+1

Split Algorithm: Demonstration-2

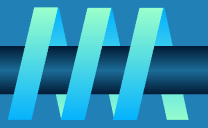


Quick Sort: Split Procedure



```
procedure split (var A:Array, var low:Index, var
   high:Index):Index;
var s,x :integer;
s:=low + 1;
x:= A[low];
for y:=low+1 to high do
   if (A[y]<x) then
       swap (y, s);
       s:=s+1;
   endif
endfor
s:=s-1;
swap (low, s);
split:=s;
end;
```

Analysis of the Split algorithm



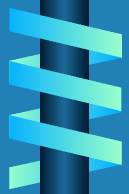
The time complexity of split procedure is O(n). It is an in-place algorithm.

Does it always split in the middle of the array? If it splits in the middle, then we have the time complexity of quick sort as given below.

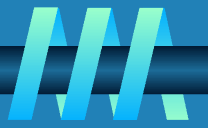
$T(n)=2T(n/2) + n$

$T(1) = 1$

But as saw in the demonstration, the split may not always split in the middle.

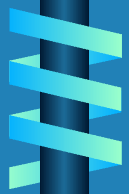


Quick Sort Algorithm Revisited



Note that even though the split algorithm may not split the array in the middle, the quicksort algorithm still sorts the array of input elements.

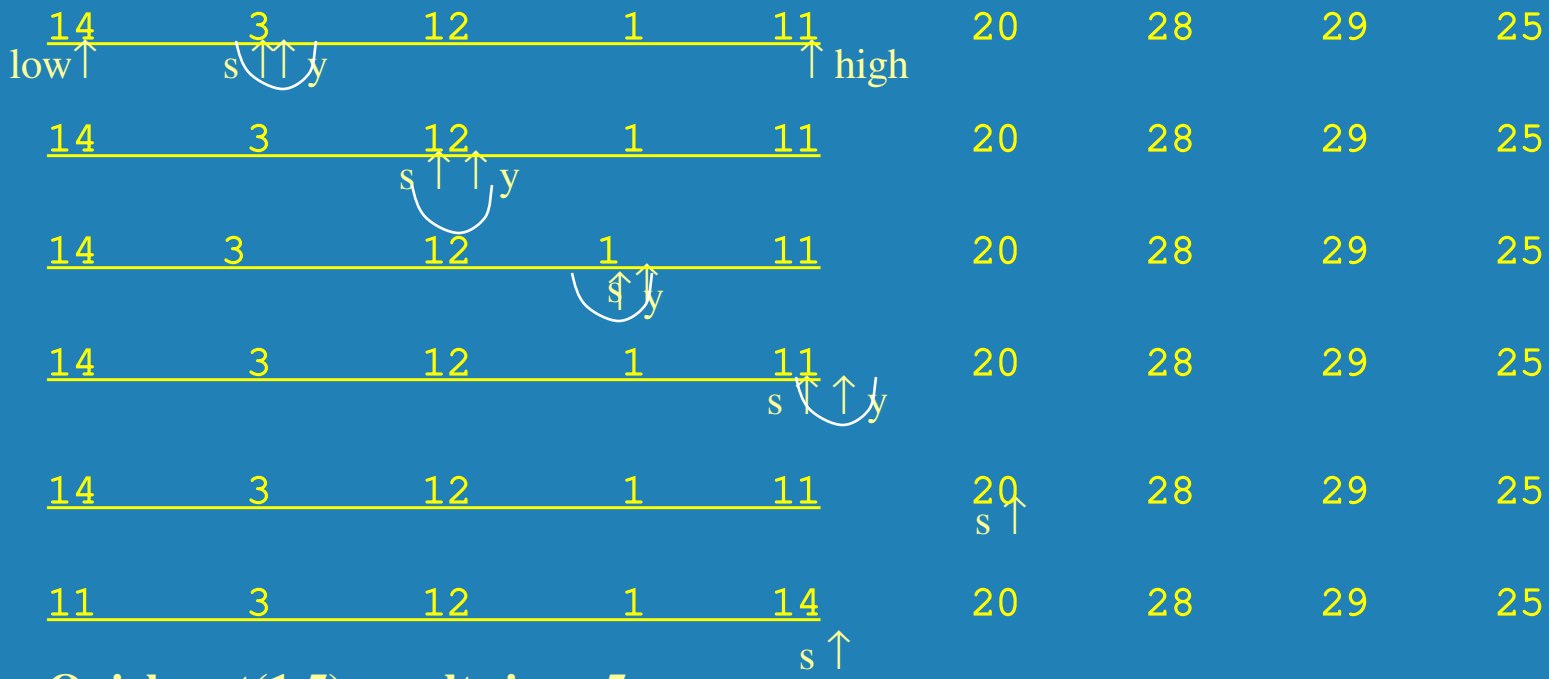
```
QuickSort (A,low,high)
begin
 if (low<high)then
       splitPoint:=split(A)
        QuickSort(A,low,splitPoint-1);
        QuickSort(A,splitPoint+1,high);
 endif
end;
```



Quick Sort: Demonstration -1

As shown in the previous slides, Quicksort(1,9) results in s=6

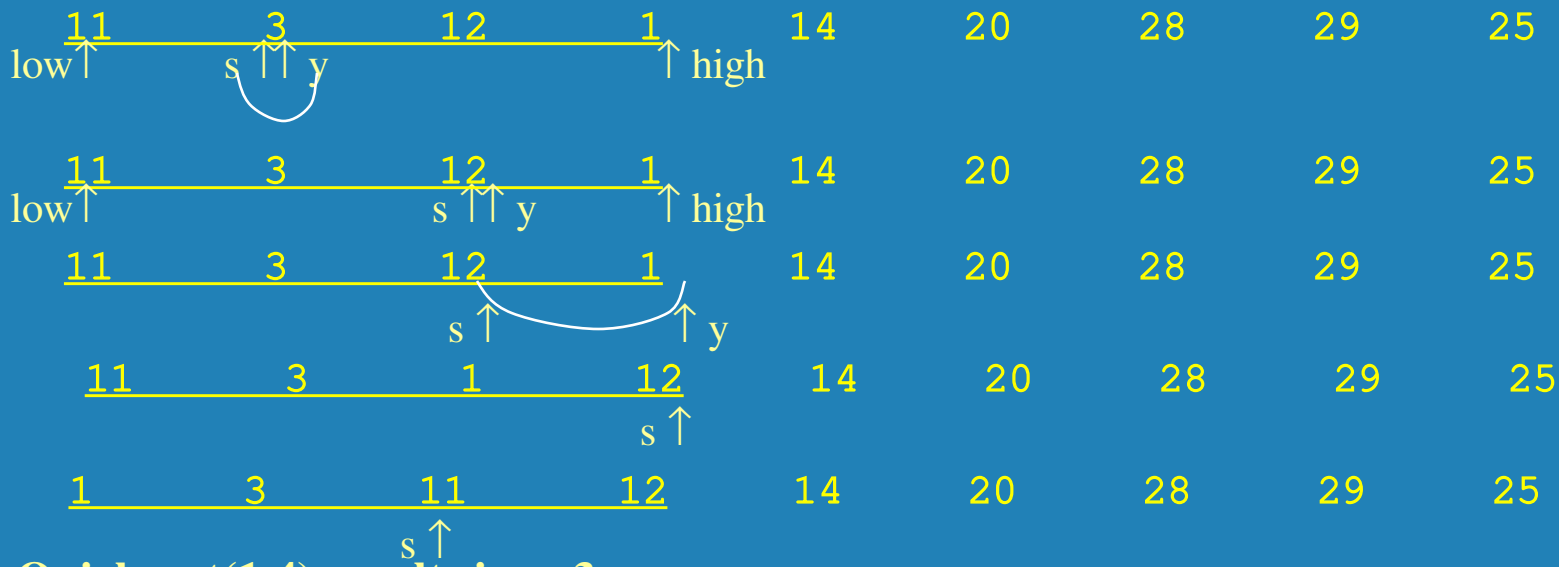
Quicksort(1,5) {s=6}



Quicksort(1,5) results in s=5

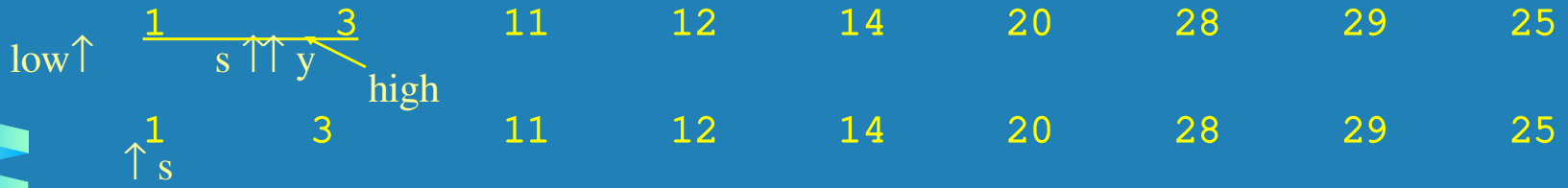
Quick Sort: Demonstration-2

Quicksort(1,4) {s =5}



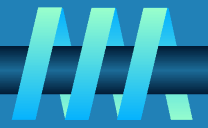
Quicksort(1,4) results in s=3

Quicksort(1,2){s =3}



Quicksort(1,2) results in s=1

Quick Sort: Demonstration-3



Calls Quicksort (1,0){s=1}, Quicksort(2,2){s=1} and Quicksort(4,4){s=3} Quicksort(6,5) {s =5} have no effect

---

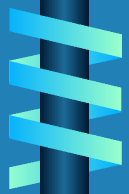
Quicksort(7,9){s =6}

1	3	11	12	14	20	28	29	25
						low↑	s⇑y	↑ high
1	3	11	12	14	20	28	25	29
							s ⇑ y	
1	3	11	12	14	20	25	28	29
							s ↑	↑ y

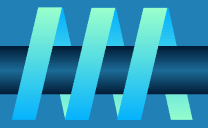
Quicksort(7,9) results in s=8

Calls Quicksort (7,7){s=8} and Quicksort(9,9) {s =8} have no effect.

Array is sorted !!!!



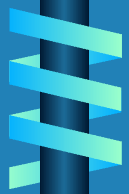
Quick Sort: Worst Case Time



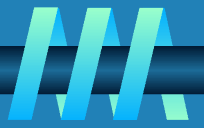
Consider the Quicksort algorithm.

```
QuickSort (A,low,high)
begin
 if (low<high)then
       splitPoint:=split(A)
        QuickSort(A,low,splitPoint-1);
        QuickSort(A,splitPoint+1,high);
 endif
end;
```

Note that the time of split algorithm is fixed. It is linear in the size of the array that is being split. The worst case occurs when the split point always turns out to be one end of the array.



Quick Sort: Worst Case Time



Assuming that split always splits the array so that the two 'halfs' are of size 0 and n-1, the recursive time complexity is:

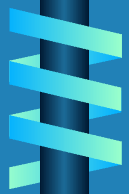
Solution:

$$T(n) = n + T(n-1), \mathrm{T}(1) = 1.$$

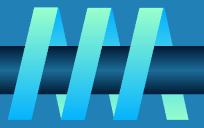
$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + n - 1 + T(n-2) \\ &= n + (n-1) + T(n-2) \\ &= n + n - 1 + (n-2) + T(n-3) \\ &= n + n - 1 + n - 2 + \ldots + T(n-(n-1)) \\ &= n + n - 1 + n - 2 + \ldots + 2 + T(1) \end{aligned}$$

$$T(n) = n + (n-1) + (n-2) + 3 + 2 + 1$$

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$$

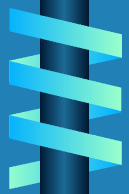


Quick Sort: Average Case

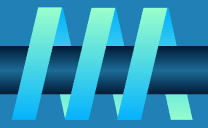


Since time of split is always linear,i.e O(n), the performance of Quicksort really depends of "where" the split occurs, if the split occurs in on extreme end of the array, we obtain a worst case performance of $O(n^2)$

The average case time A(n), depends on where exactly the split occurs. Let us assume that it occurs at position i, $(1 \leq i \leq n)$ and it is equally likely for all positions 1 to n.Therefore the probability that split occurs at position i is therefore 1/n. If the split occurs at position i, then the next two calls of Quicksort are on array of sizes i-1 and n-(i+1)+1=n-i. The time for the two Quicksort calls are A(i-1) and A(n-i).



Quick Sort: Average Case



Therefore,

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

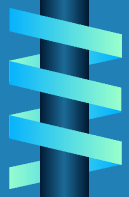
Simplify A(n)

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

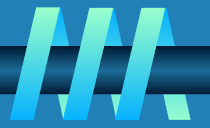
$$= n - 1 + \frac{1}{n} [A(0) + A(1) + A(2) \ldots A(n-1)$$

$$[A(0) + \ldots\ldots + A(2) \quad A(n-1)]$$

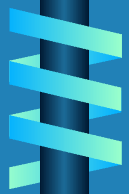
$$A(n) = n - 1 + \frac{1}{n} 2 \sum_{i=0}^{n-1} A(i)$$



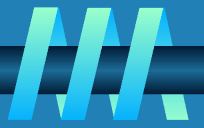
Quick Sort: Average Case



Use A(n) = .. A(n-1) = … multiply A(n) by n-1 and A(n-1) by n andsubtract. Integrate to get approximate value



Quick Sort: Best Case Analysis



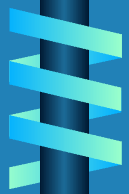
In the best case, split occurs in the middle on each call to Split. Construct the best case scenario for array of size 8. The best case time is given by

$$T(n) = n + 2T\left[\frac{n}{2}\right], \; T(1) = 1$$

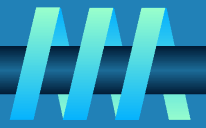
This was solved in the context of merge sort to be:

$$T(n) = n \log n$$

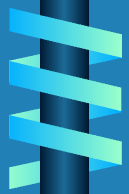
Question: What is the worst case memory complexity of Quicksort?



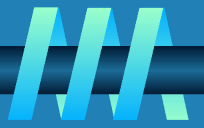
Sort Time Complexity



	Memory	Worst	Ave	Best
Smallest Based Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n^2)$
Insertion Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n)$
Merge sort	O(n)	O (n logn)	θ(n logn)	Ω(n logn)
Quick Sort	O(n)	$O(n^2)$	θ(n logn)	Ω(n logn)



Introduction to Heap Sort



Heapsort is a *transform-and-conquer* technique

Binary Tree: A tree in which each node has 0, 1 or 2 children.

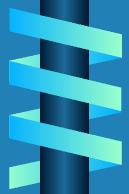
Complete Binary Tree: A binary tree in which the distance between root and every leaf is same and every node has 0 or 2 children.

Relationship between number of nodes and height in a complete binary tree:

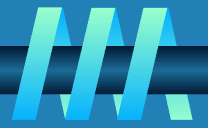
If the depth of the complete binary tree is h, then the number of nodes is given by

$n = 2^0 + 2^1 + 2^2 + \ldots 2^{h-1} = (2^h - 1)/(2-1) = 2^h - 1$ or $O(2^h)$

If there are n nodes in a complete binary tree, the height of the tree is log(n+1) or O(logn)



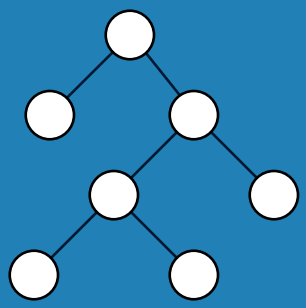
Introduction to Heap Sort

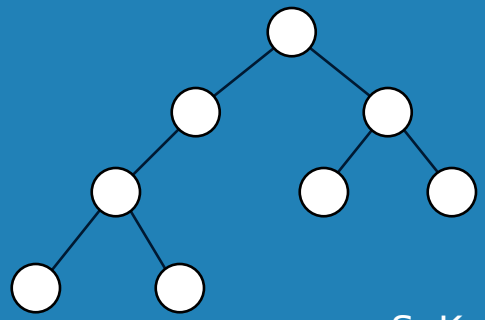


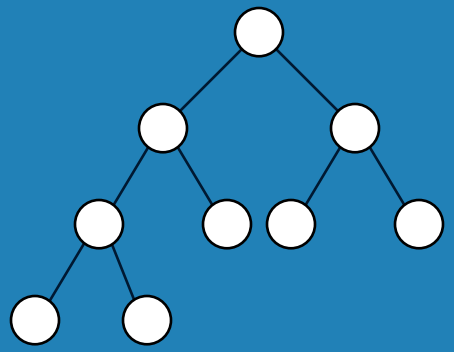
Structural Requirement of a heap: A complete binary tree with some right most leaves removed. Note that the relationship between n and h for heap is same as that of complete binary tree.

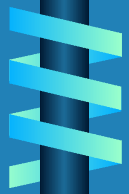
Data Ordering Requirement of a heap: Data at any node is greater than the data of the children.

Which of the following the heap structure requirement?

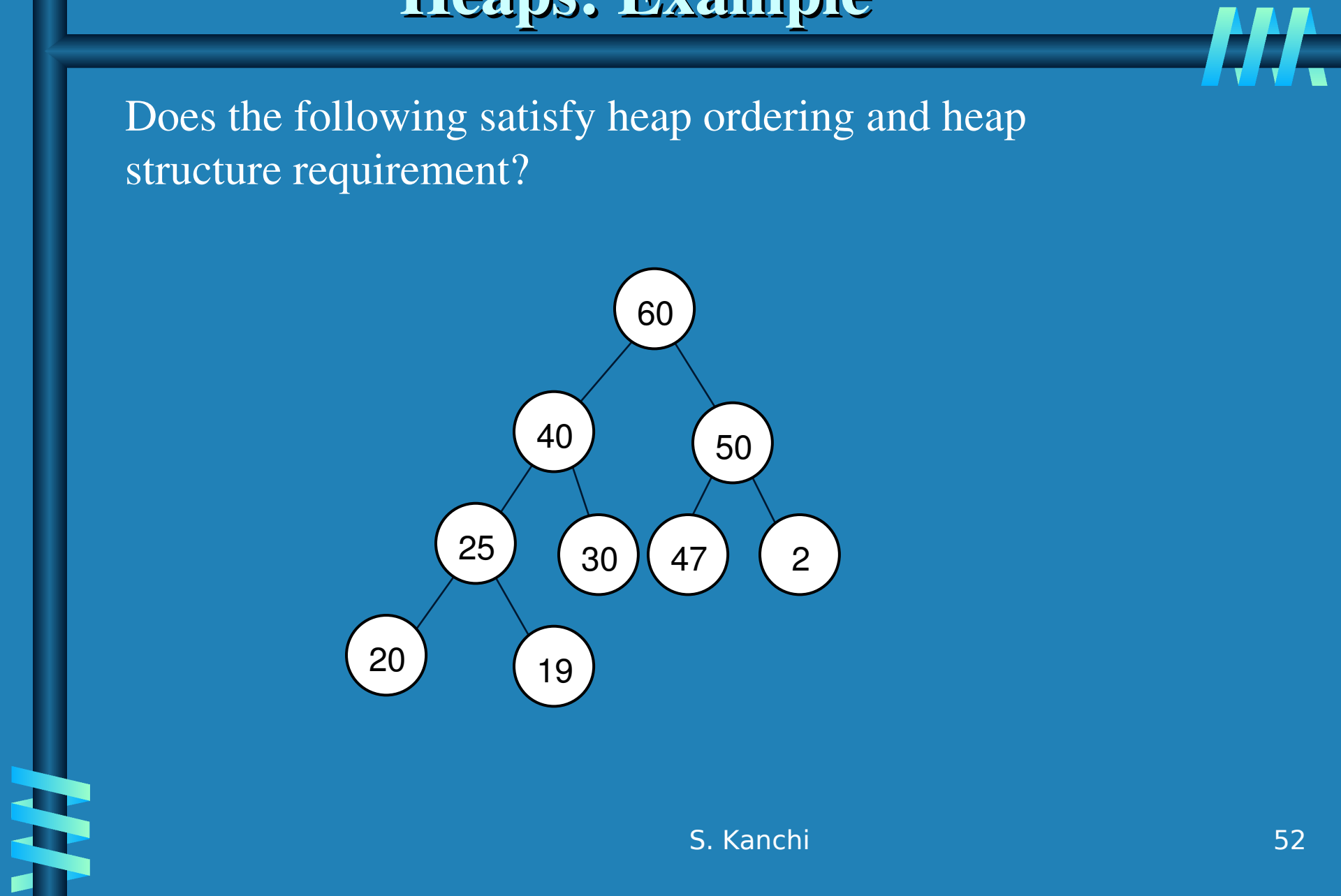






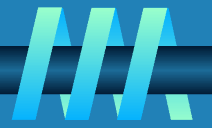


Heaps: Example



Does the following satisfy heap ordering and heap structure requirement?

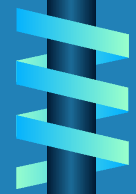
Heap as a data structure



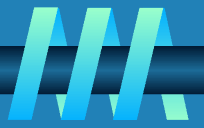
Let us assume that the following operations are available for a heap and time for each is O(1).

Assume that heap is stored as a binary tree data structure.

```
rightChild(node)
leftChild(node)
value(node)
setValue(node)
isRoot(node)
isLeaf(node).
```



Fix Heap

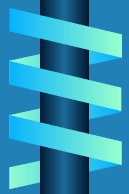


Let us say that we have a heap. We want to add an element to the heap so that the result is also a heap. We call this algorithm the fixheap algorithm. We will assume that data to insert is given and there is a vacant node.

There are two versions of the fixheap algorithm. One in which the root is vacant. This is called heapDown algorithm.

One in which a leaf is vacant. This is called the heapUp algorithm.

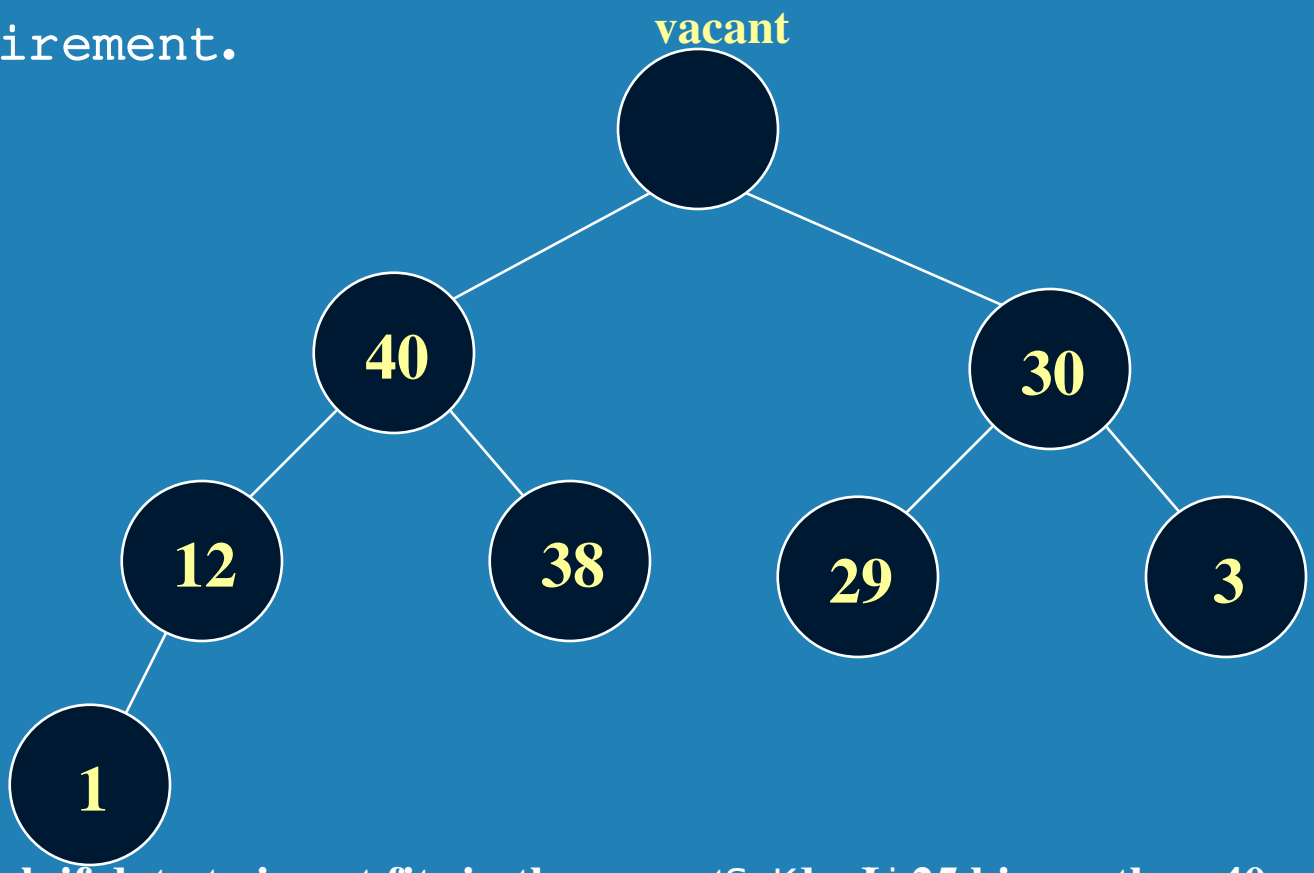
We will be using both as needed.



Fix Heap Demonstration-1

Assume that the root is vacant and we want to insert 25. Note that the rest of tree follows the heap structure and data ordering requirement.

25

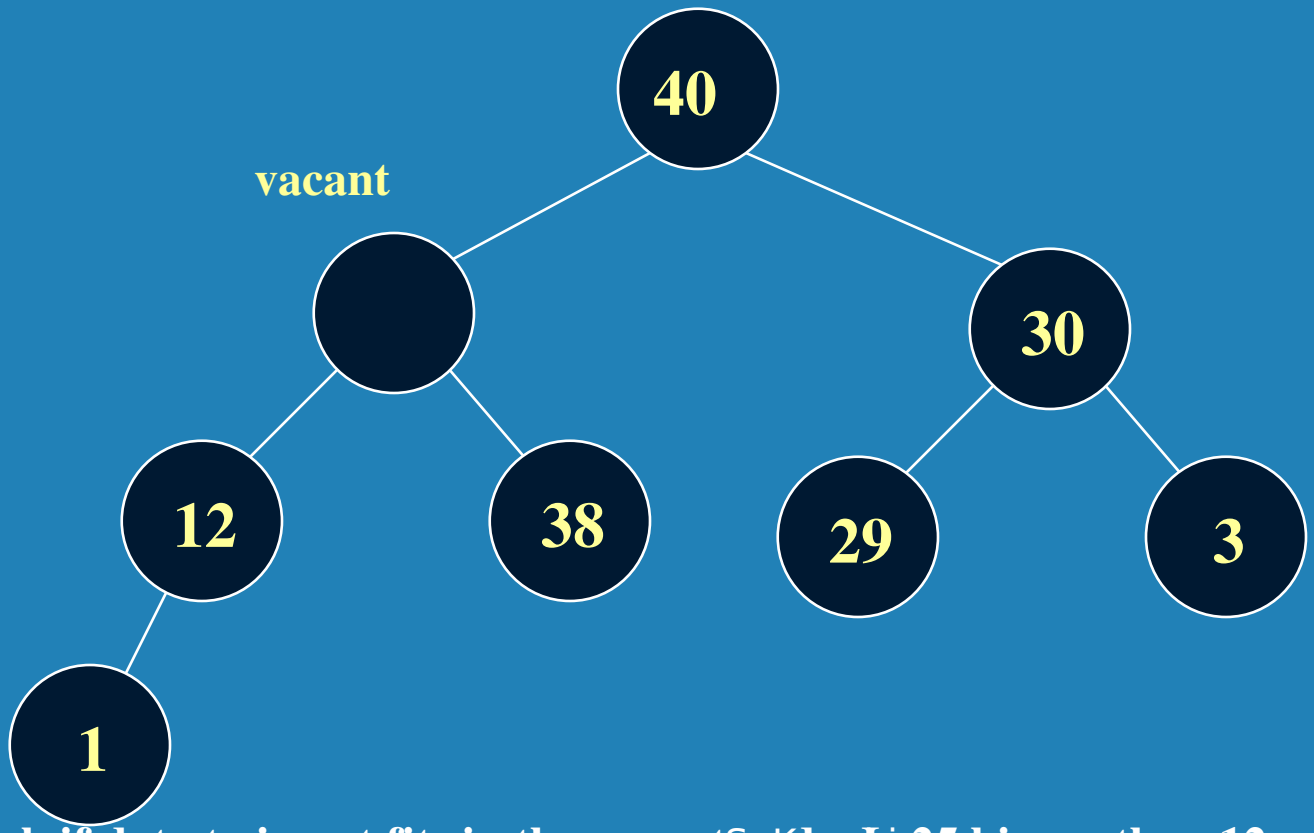


Check if data to insert fits in the vacant node. Is 25 bigger than 40 and 30? No.

Fix Heap Demonstration-2

Move the larger data among the children of the root up to the root, and make the child vacant

25

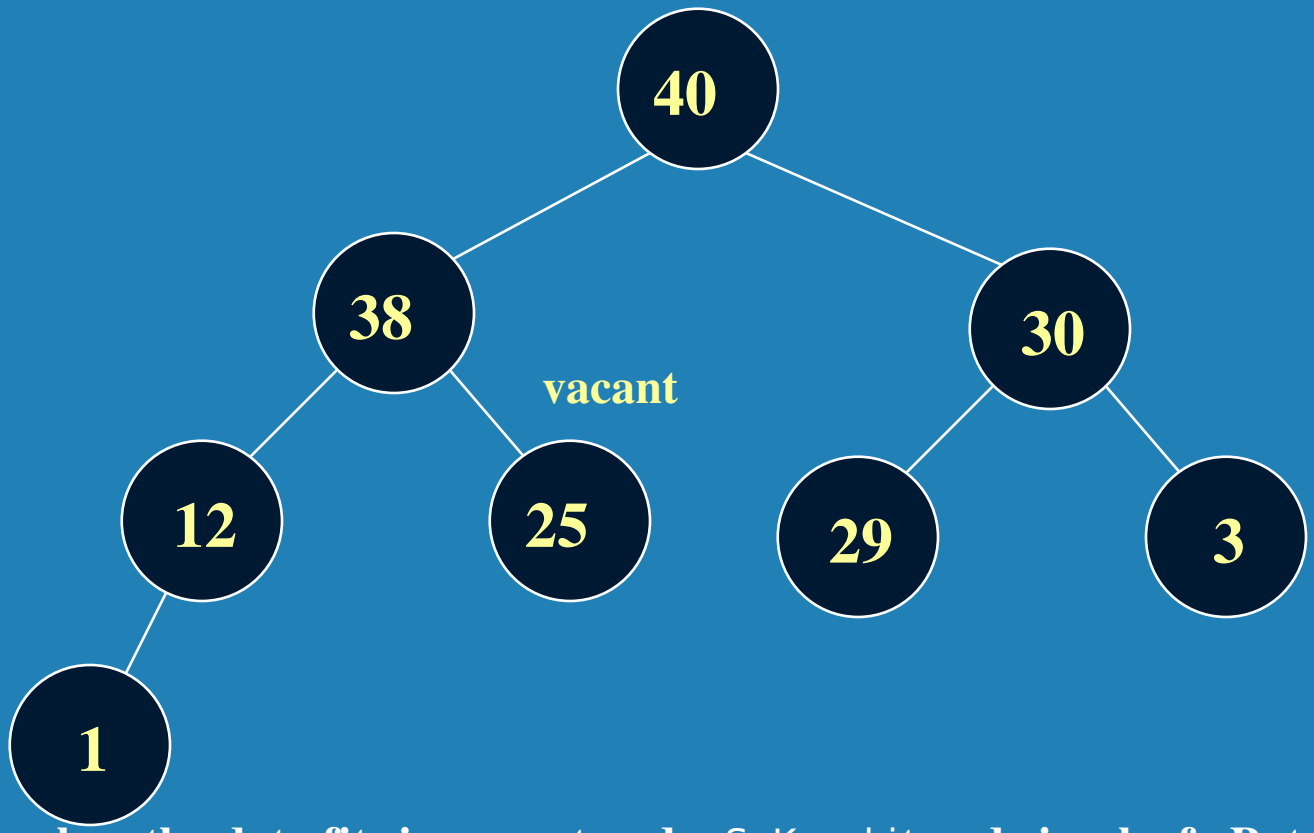


Check if data to insert fits in the vacant node. Is 25 bigger than 12 and 38? No.

Fix Heap Demonstration-3

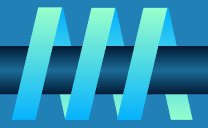
Move the larger data among the children of the root up to the root, and make the child vacant

25



Stop when the data fits in vacant node, or vacant node is a leaf. Put the 25 in vacant node.

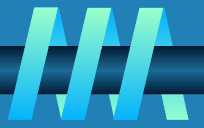
Fix Heap:Algorithm



```
procedure fixheap (root, key) begin
     vacant := root;
     while (vacant is not leaf and not done)begin
          if (key > value (leftchild (vacant) and
              ( key > value (rightchild(vacant))then
                    setValue(vacant,key);
                     done:=true;
          else
             move the larger value of children
            of vacant to vacant
             let vacant := node containing larger of
              values at left child and right child of
              vacant
         endif
     endwhile
    if (vacant is a leaf)
      then setValue(vacant, key);
end;
```

The time complexity of fixheap is O(h).

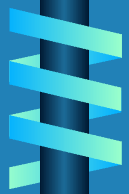
Sorting Using Heaps



Suppose that a heap of n elements is given. How do we “sort” using the heap?

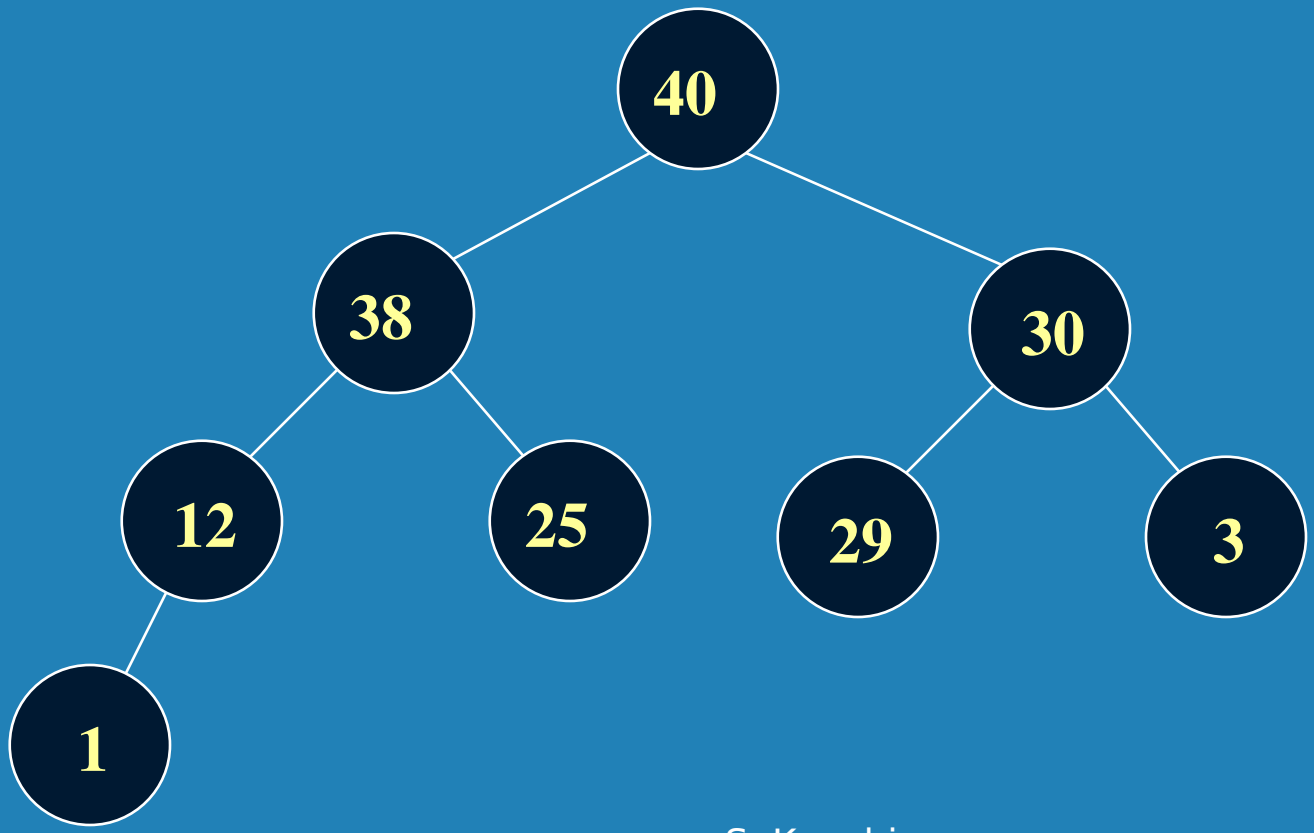
Consider the following idea. Suppose we remove the largest element that is at the root and set it aside. Now the root is vacant.

Since finding the second largest, third largest etc gets more complex, we simply remove the data at the leaf and run a fix heap algorithm with the root as vacant! After the fixheap the second largest is at the root and the heap has one less leaves.We repeat this process until all nodes in the heap are removed and values are set aside in the descending order!



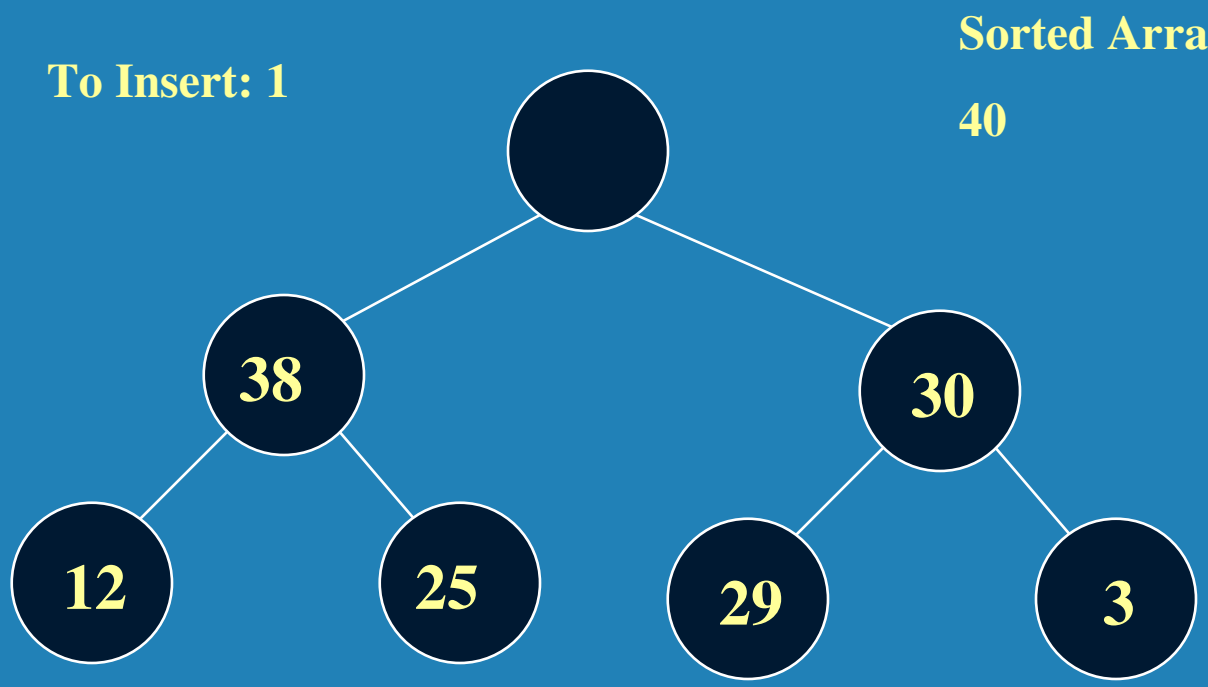
Demonstration of HeapSort-0

Let us sort the heap below. General idea is that we will set aside the value at root and remove the rightmost leaf and reinsert the data at rightmost leaf.



Demonstration of HeapSort-1

We will set aside 40. Run the fix heap with 1 to insert.

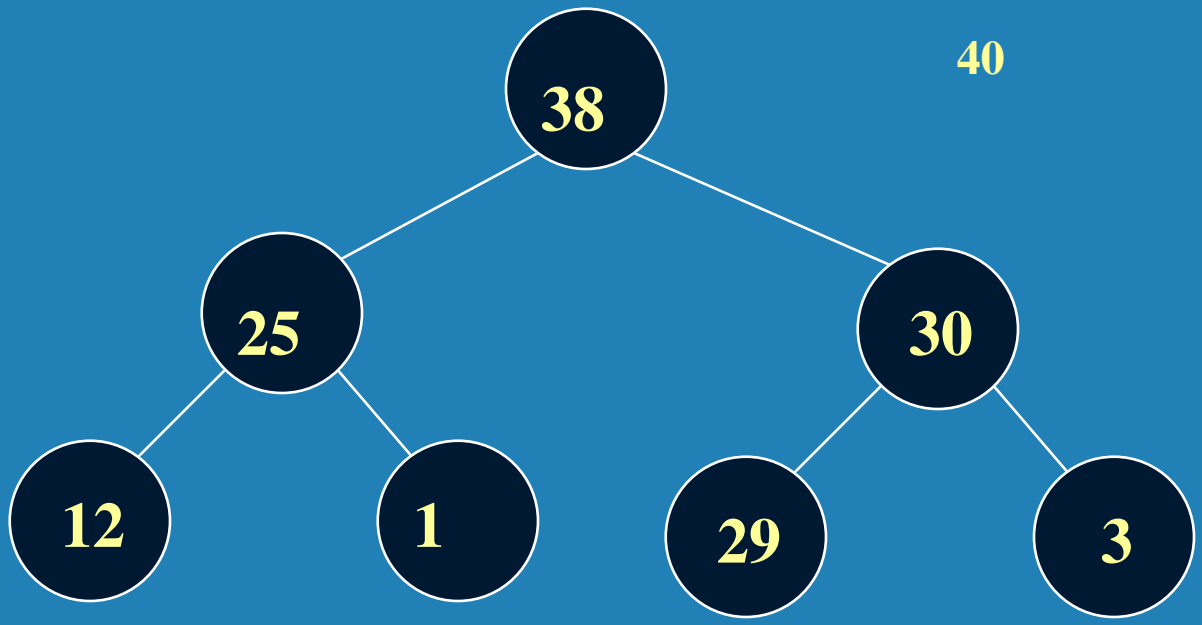


Demonstration of HeapSort-2

After the fix heap algorithm, we get:

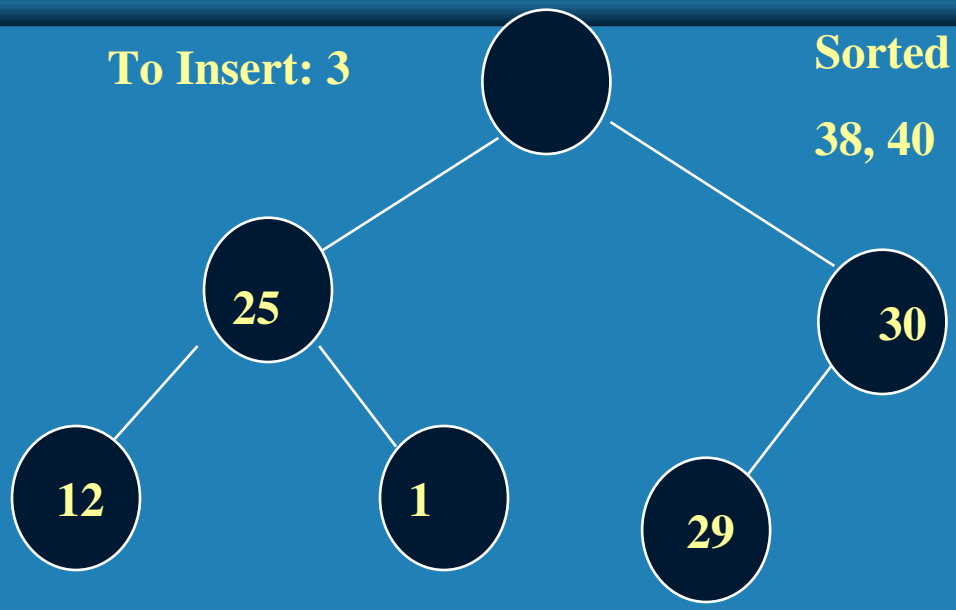
Sorted Array:

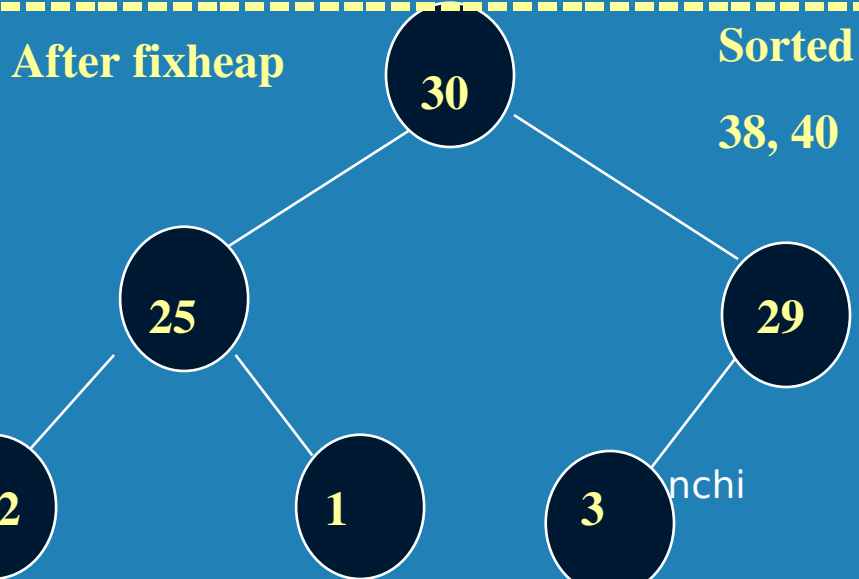
40



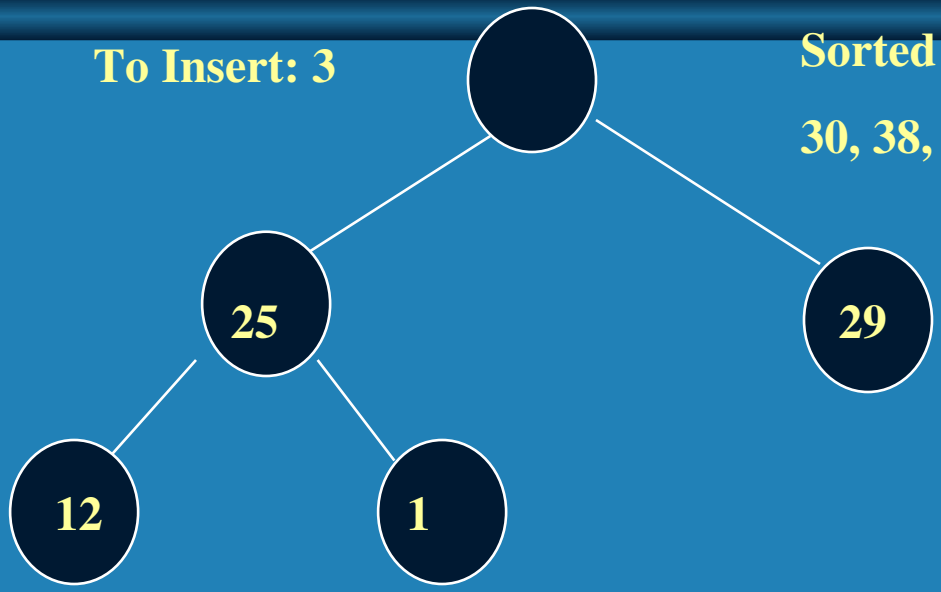
Repeat the process by setting aside 38, and removing leaf with 3 and reinserting 3 into the tree.

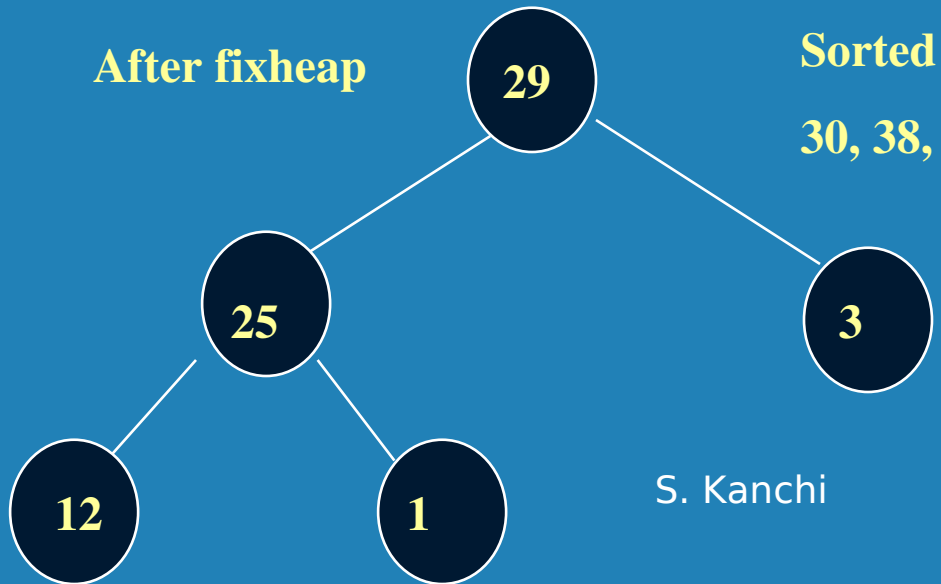
Demonstration of HeapSort-3



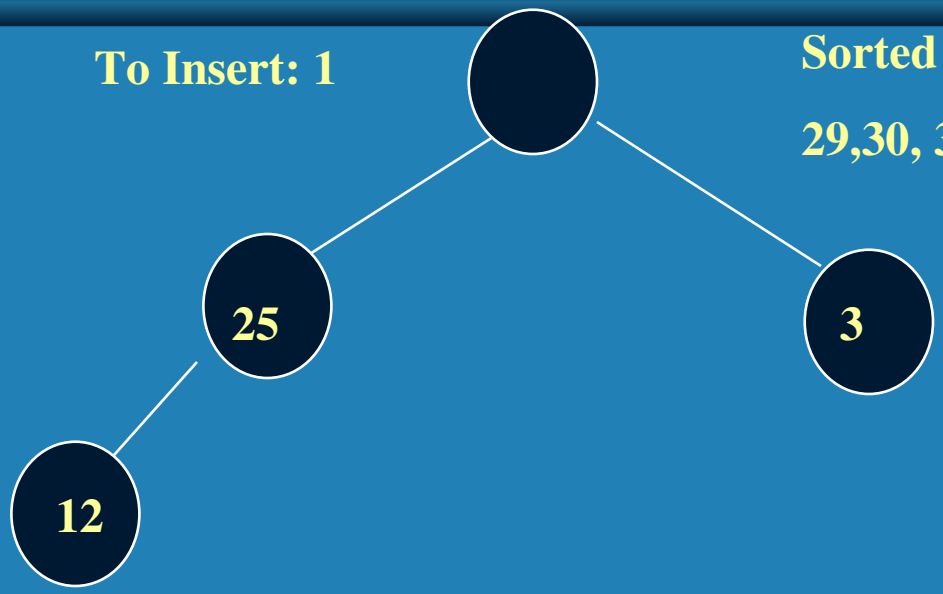


Demonstration of HeapSort-4



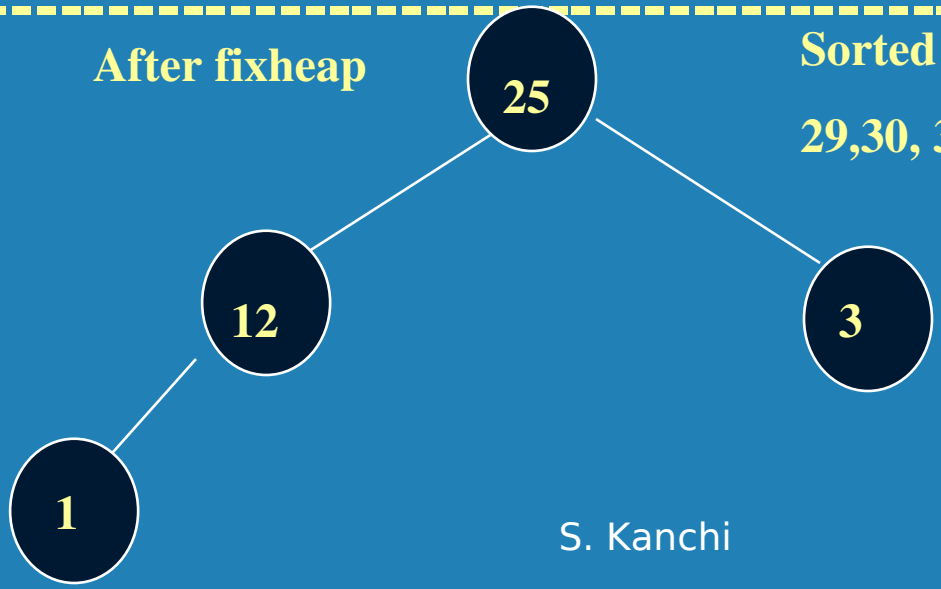


Demonstration of HeapSort-5



Sorted Array:

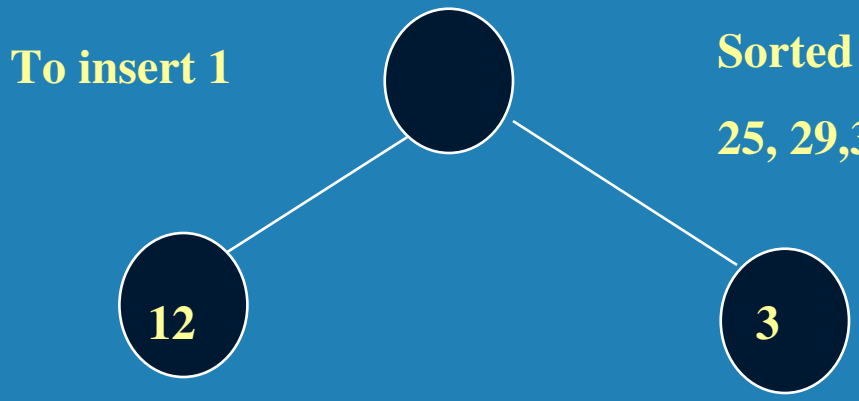
29,30, 38, 40

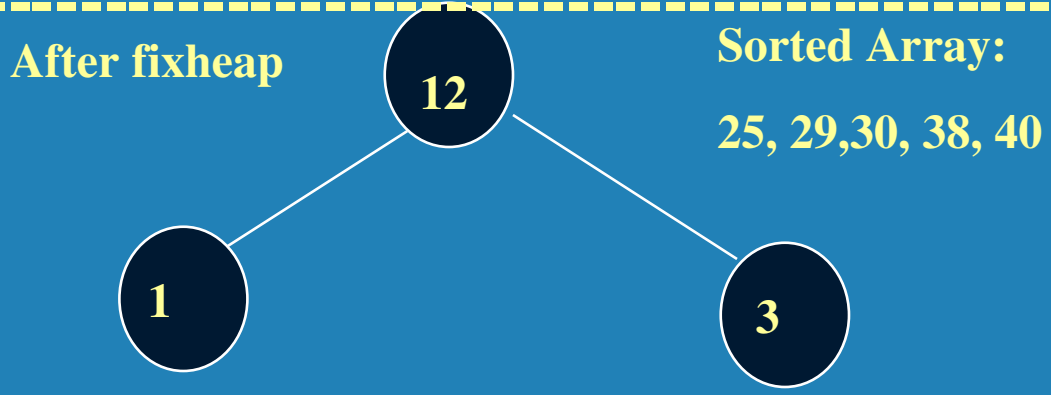


Sorted Array:

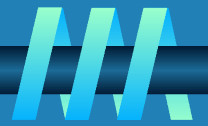
29,30, 38, 40

Demonstration of HeapSort-6

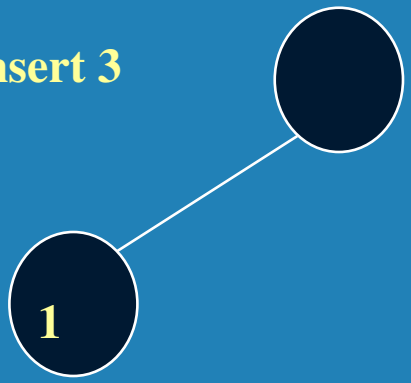




Demonstration of HeapSort-7



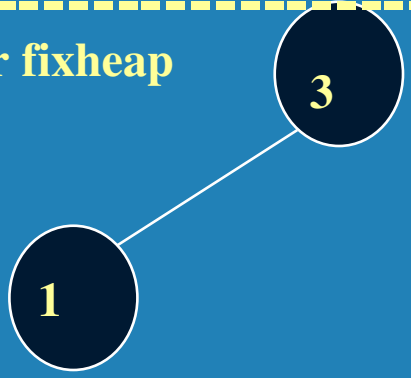
To insert 3



Sorted Array:

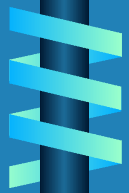
12, 25, 29,30, 38, 40

After fixheap



Sorted Array:

12, 25, 29,30, 38, 40



Demonstration of HeapSort-8

To insert 1

Sorted Array:

3, 12, 25, 29,30, 38, 40

After fixheap

Sorted Array:

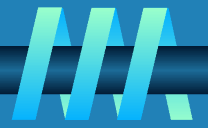
3, 12, 25, 29,30, 38, 40

In the last step, simply add the remaining data at root to the array and remove the root.

Sorted Array

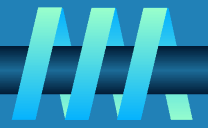
1, 3, 12, 25, 29,30, 38, 40

HeapSort Code



```
procedure heapSort (root, n);
var count:integer;
var data: integer;
begin
  // Construct heap out of the elements in the tree
      rooted at 'root'.
  count := 0;
  while (count  ≤  n)
   count ++;
   save the value at root in an array and make root
   vacant;
   rLeaf := right most leaf;
   data := value(rLeaf);
   remove the node at rLeaf from the heap.
   fixheap (root, data);
  endwhile;
end;
```

Heap Sort: Preliminary Analysis



We will postpone the problem of starting with a heap structure with randomly ordered elements and and turning it into a heap. This would later be called the constructHeap algorithm.

The loop within the heapSort code executes O(n) times.

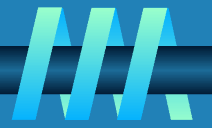
The fixheap in the loop takes O(h) time. Note that the height of the tree is reducing as the iterations progress. However, the worst case for the height is O(logn). So, in each iteration, the time of fixheap is bounded by O(logn).

The only other difficult part of the loop is getting the right most leaf. This can be done in O(h) time if size of heap is known also. [Write an O(h) time algorithm] Therefore the worst case time for getting the right most leaf is also O(logn).

Assuming we start with a heap, time complexity of heapSort is O(nlogn).

If we can construct heap out of a heap structure in O(nlogn) time we can actually get a O(nlogn) algorithm for heapSort.

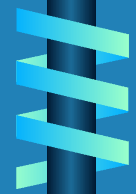
Construct heap: Introduction



Suppose we are given a heap structure, but the data does not follow the data ordering requirement. We would like to construct heap out of the structure.

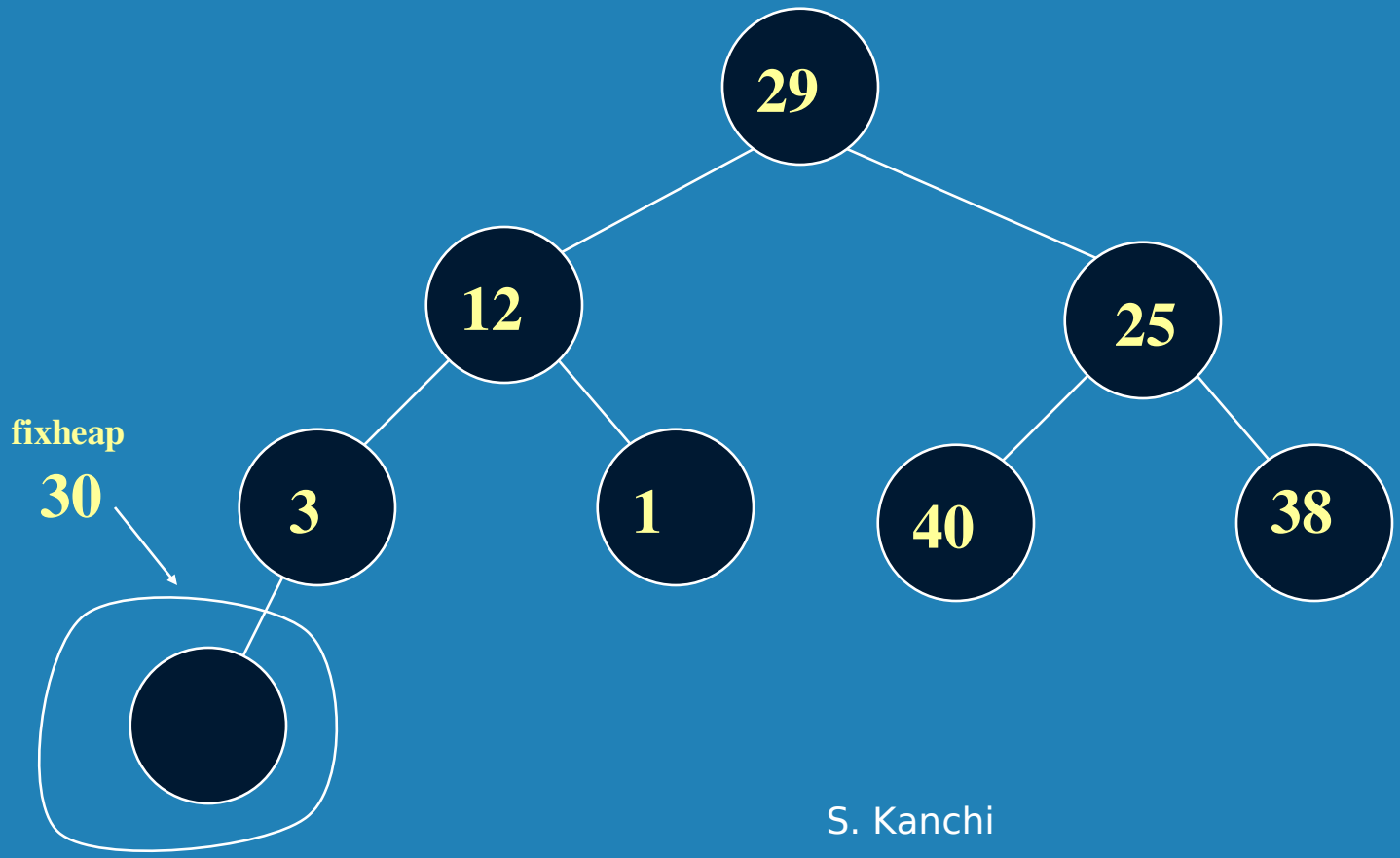
We will execute fixheap on subtrees rooted at each node in a level, starting lowest level working up to the root.

Each time we execute fixheap at a level, the data to insert would be the data at the root of the subtree, and vacant node would be root of the subtree.



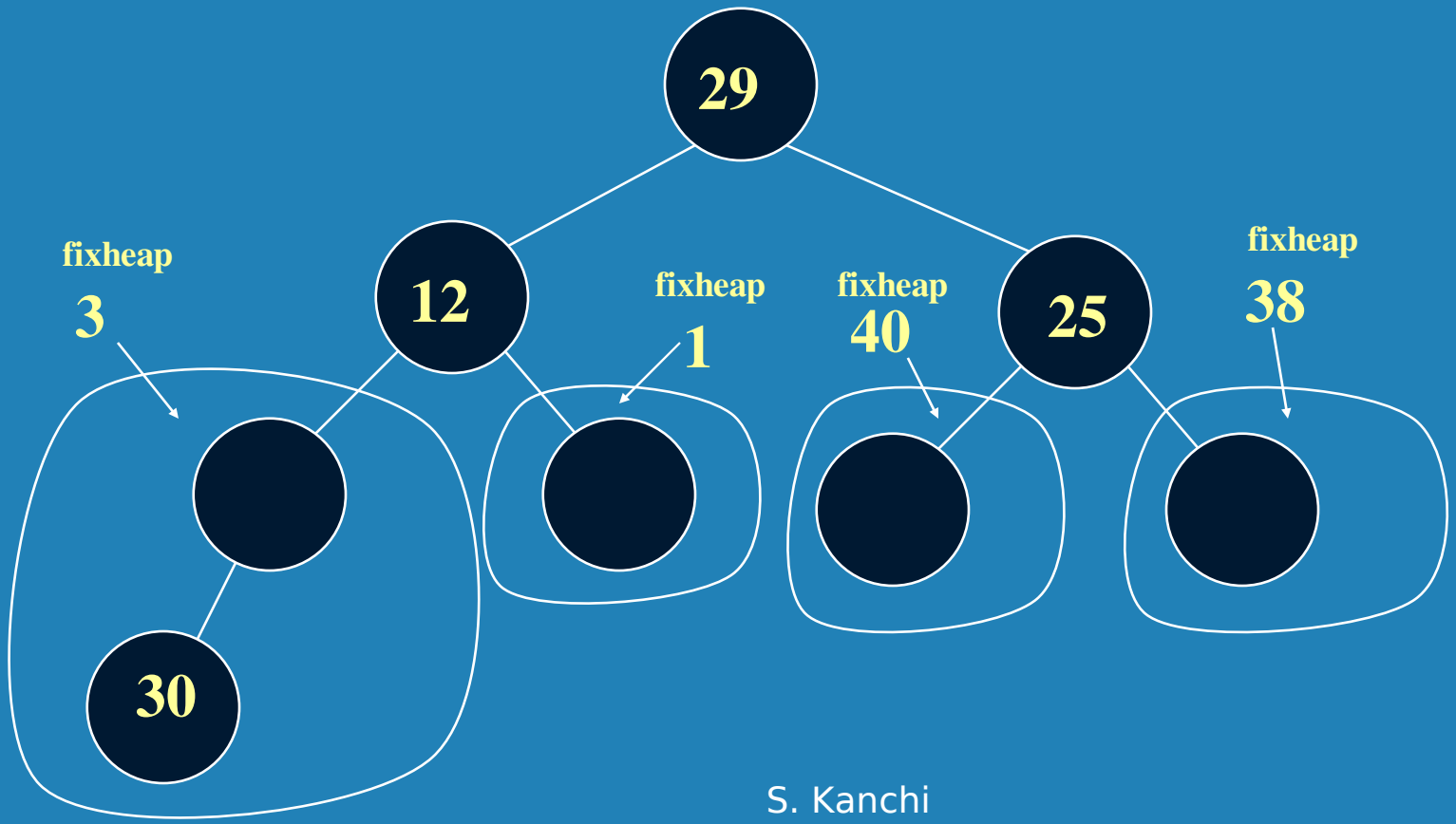
Construct heap: Demonstration-1

First fixheap each node at the last level (level =4) in the tree. The data to insert is the value at the node.



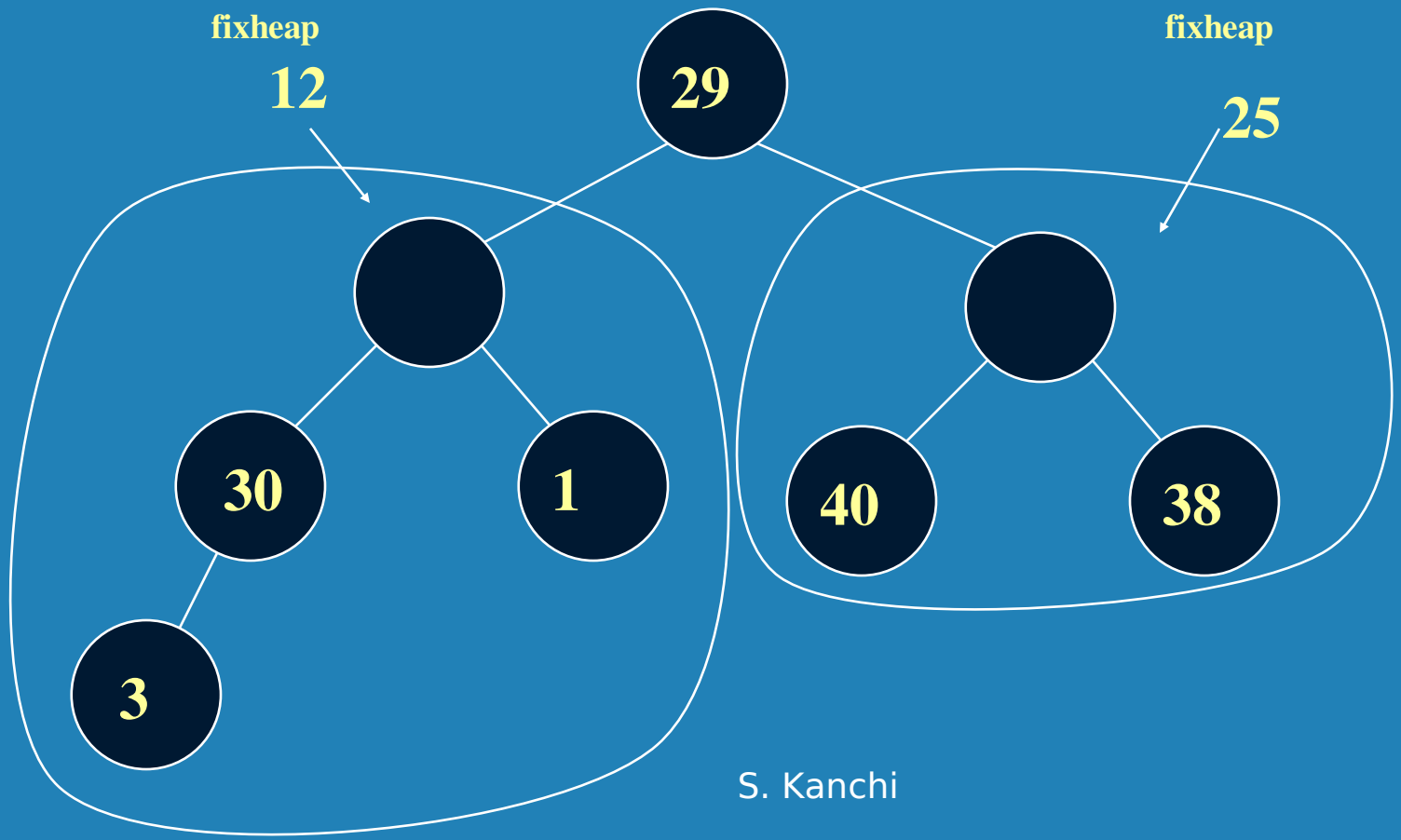
Construct heap: Demonstration-2

Next, fixheap each node at the next higher level (level =3) in the tree. The data to insert is the value at the node.



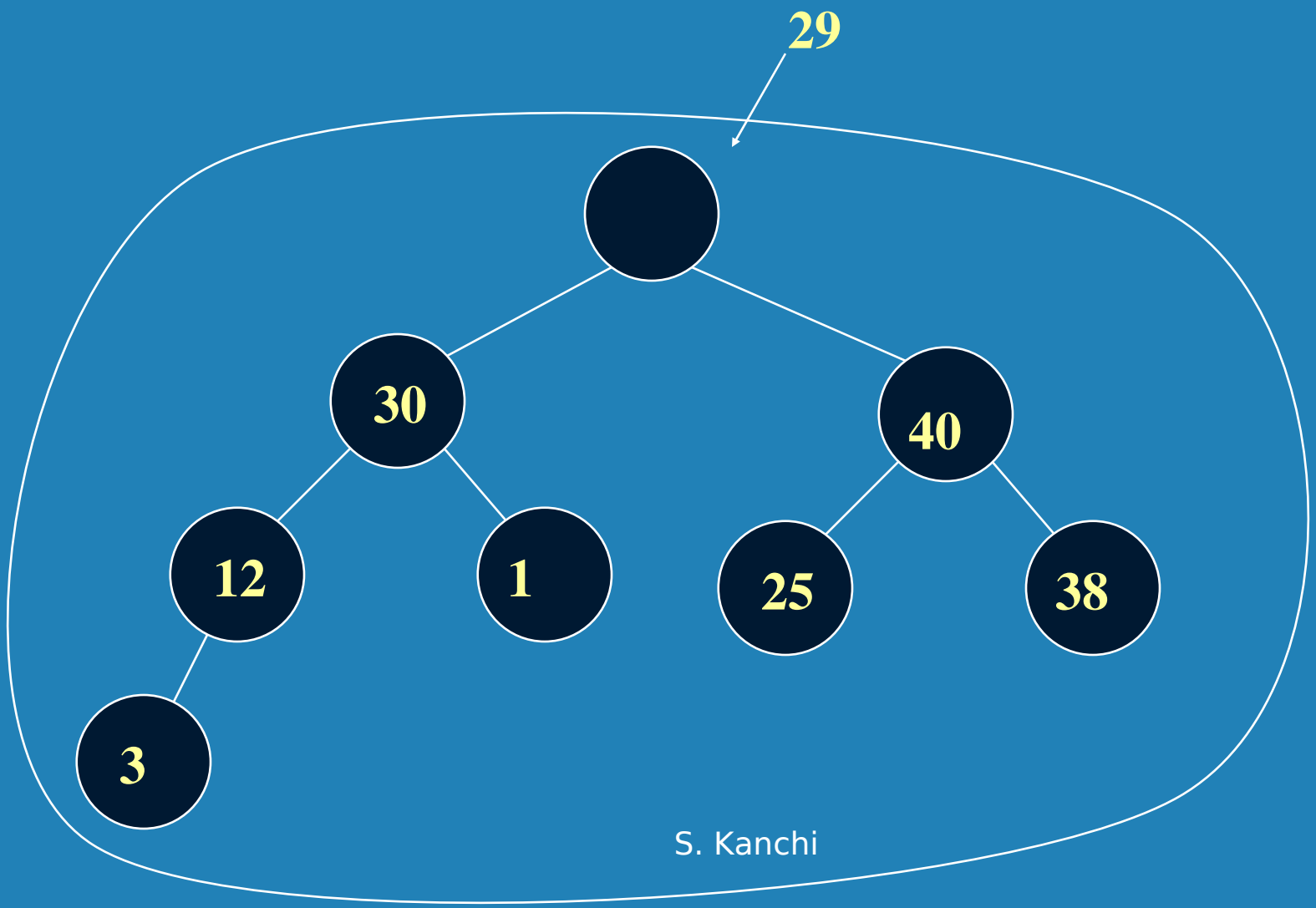
Construct heap: Demonstration-3

Next, fixheap each node at level=2



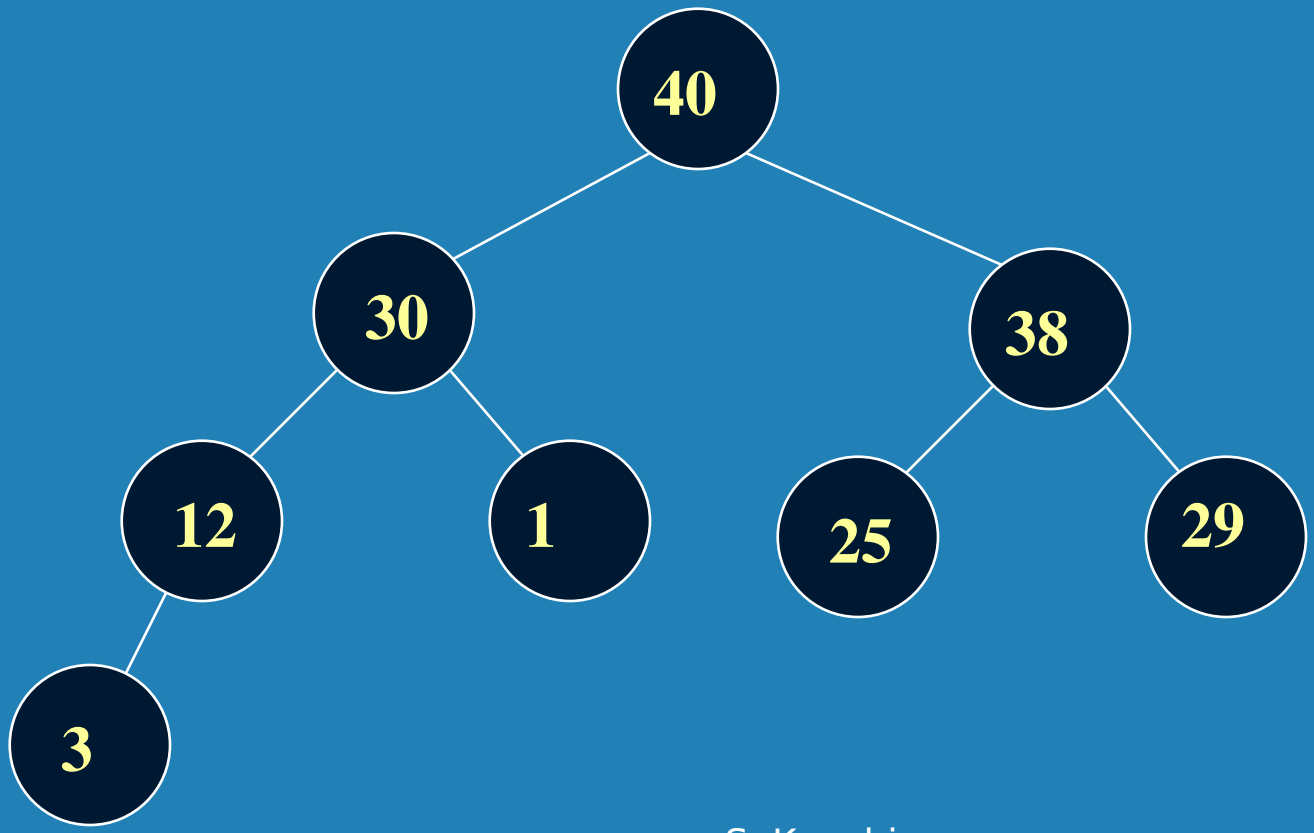
Construct heap: Demonstration-4

Next, fixheap the node at level=1



Construct heap: Demonstration-5

The construction of heap is now complete. Once the construction of heap is complete, we should be able to run the loop in the heapSort algorithm as described above.



Construct heap:Algorithm

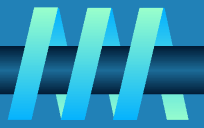
We are looking for an algorithm to construct heap that takes O(nlogn) time as mentioned above. There are two ways to construct heap.

One way is to construct starting at the lowest level as described above. The code for that construct heap is

```
for level := h down to 1 do
    for each node n in level 'level',
            fixheap(n, value(n));
   endfor
endfor;
```

The problem with the code is that it has time complexity that is very high, since we have to find nodes at each level given the root. So we rewrite the algorithm recursively as follows:

Construct heap: Recursive Algorithm

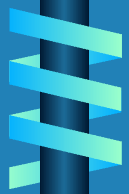


We simply perform a post-order traversal of the tree, fixing as we travel the tree.

```
procedure constructHeap (node)
begin
       if (leftchild (node) exists)
       then constructHeap (leftchild (node));

      if (rightchild(node)  exists)
       then constructHeap ((rightchild(node));

       fixheap (node, value(node));
 end;
```

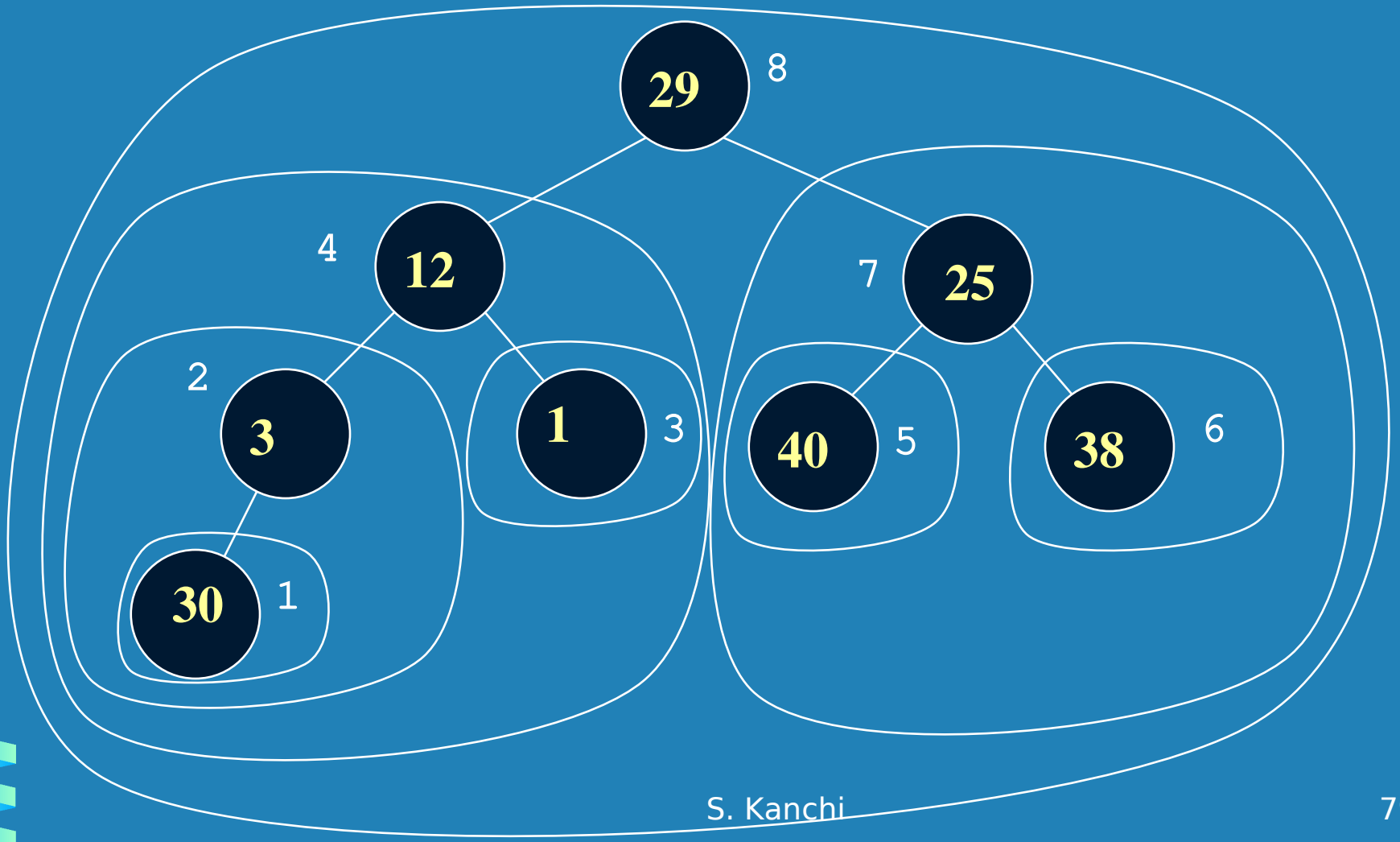
To make a random tree with heap structure into a heap we call

```
constructHeap(root);
```

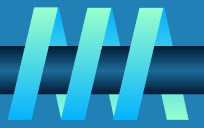


Construct Heap: Recursive Algorithm Demonstration

The following shows the order in which the fixheaps will occur in the recursive version of construct heap.



Construct Heap: Time Complexity



To construct heap a tree of height h, we have to construct heap two trees of height h-1 and then fixheap a tree of height h. The time for fixheap is O(h) if the heap has height h.

$T(h) = 2T(h-1) + h.$

$T(0) = 0.$

Solving the recursion,

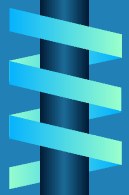
$T(h) = h + 2T(h-1)$

$= h + 2[(h-1) + 2T(h-2)]$

$= h + 2(h-1) + 2^2T(h-2)$

$= h + 2(h-1) + 2^2(h-2) + 2^3T(h-3)$

$= h + 2(h-1) + 2^2(h-2) + \ldots + 2^hT(h-h)$



Construct Heap: Time Complexity

$$T(h) = \sum_{i=0}^{h}(h - i)2^i = \sum_{i=0}^{h} h2^i - \sum_{i=0}^{h} i2^i$$

$$= h\sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} i2^i$$

$$= h\left(\frac{2^{h+1} - 1}{2 - 1}\right) - \sum_{i=1}^{h} i2^i$$

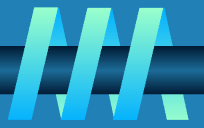
$$= h2^{h+1} - h - \sum_{i=1}^{h} i2^i = h2^{h+1} - h - [\quad]$$

$$= h2^{h+1} - h - \left[\frac{h2^h}{\ln 2} - \frac{2}{\ln 2} - \frac{2^h}{(\ln 2)^2} + \frac{2}{(\ln 2)^2}\right]$$

$= \mathbf{O(h2^h)}$

Therefore constructHeap on a tree of height h takes $O(h2^h)$ time. Since the heap is of height logn, the time for constructHeap of a heap of n elements is O(nlogn).

Heap Sort: Time Complexity

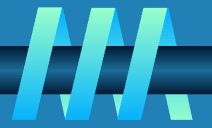


Therefore the time for heapsort is

```
procedure heapSort (root, n);
var count:integer;
var data: integer;
begin
  constructHeap(root,n);
  count := 0;
  while (count  ≤  n)
   count ++;
   save the value at root in an array and make root
   vacant;
   rLeaf := right most leaf;
   data := value(rLeaf);
   remove the node at rLeaf from the heap.
   fixheap (root, data);
  endwhile;
end;
```

Time for constructHeap is O(nlogn). As described earlier, the time for the while loop is also O(nlogn). The time for heapsort is therefore, O(nlogn).

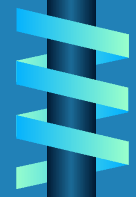
Heap Sort: Memory Complexity



The problem is that even though we got the time complexity we wanted, it is not easy to get the memory complexity to be O(1). Currently, we are using a lot of memory. First of all, we need an extra array to store the elements that we removed from the root in each iteration. We are also using a recursive algorithm for constructHeap, which takes memory on the stack.

The memory complexity in the current situation is not O(1). [Analyze the memory complexity]

We can however, implement the entire heapsort on a array of n randomly ordered elements even without a tree structure. In this implementation the time complexity will remain the same, but the memory complexity will be O(1)!

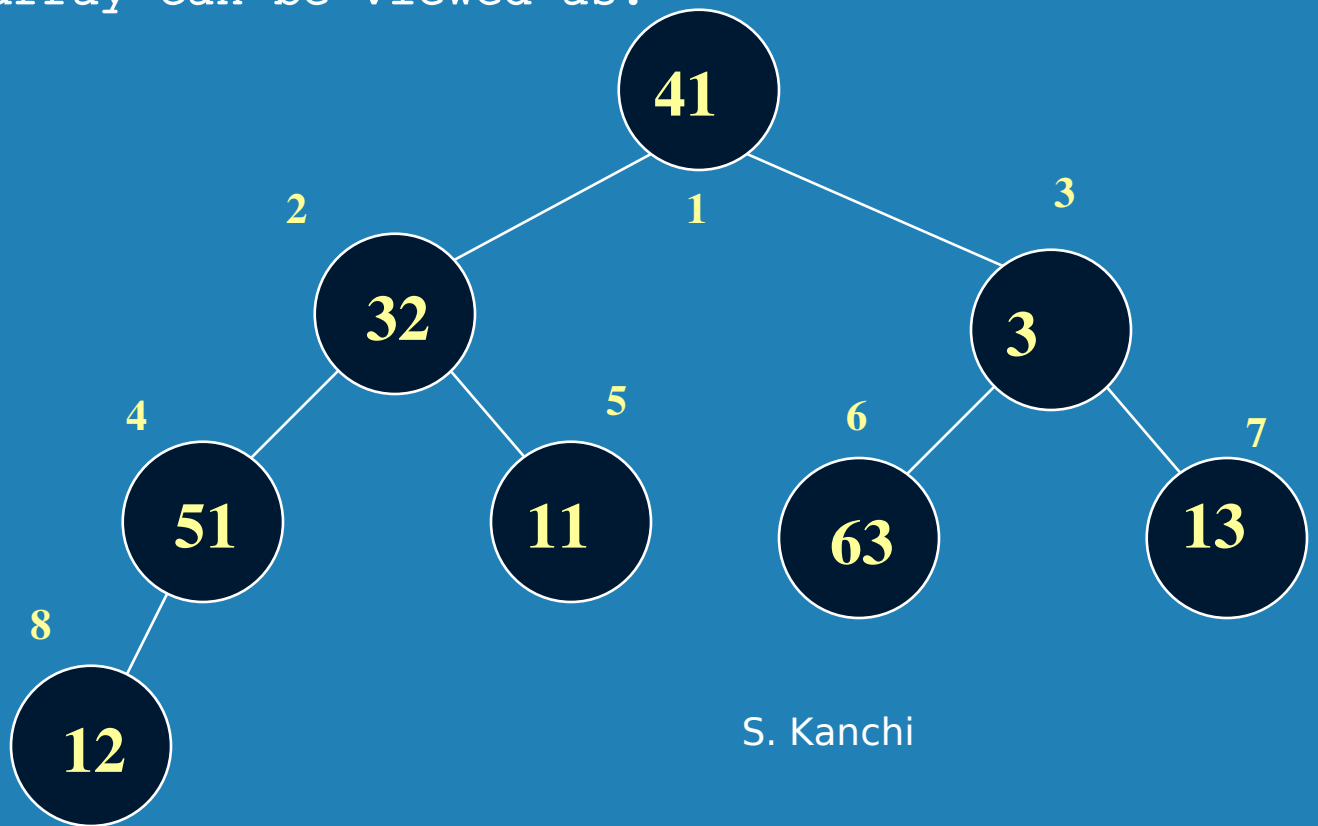


Heap Sort: Array Implementation

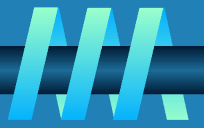
Every array is actually a heap! Consider the array:

41	32	3	51	11	63	13	12
1	2	3	4	5	6	7	8

The array can be viewed as:



Heap Sort: Array Implementation



Each node is simply the index in the array A. Let n be size of the array. heapSize = n;

The root:=0,

leftchild(node) := 2*node, if 2*node <= heapSize or nil otherwise

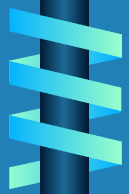
rightChild(node):= 2*node +1, if 2*node +1 <= heapSize or nil otherwise

value(node) := A[node],

setValue(node,key) sets A[node]:=key

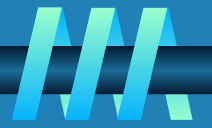
parent(node) := node/2,

isRoot(node) and isLeaf(node) are O(1) operation.



Each can be implemented as an O(1) operation on the array A.

FixHeap: Array Implementation

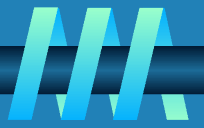


Therefore the fixheap for an array can be implemented exactly as is with the functions replaced by array based functions w/o changing the time complexity.

```
procedure fixHeap (root, key) begin
        vacant := root;
        while (vacant is not leaf and not done)begin
                if (key > value (leftchild (vacant) and
                    ( key > value (rightchild(vacant))then
                     setValue(vacant,key);
                      done:=true;
                else
                   move the larger value of children
                  of vacant to vacant
                   let vacant := node containing larger of
                    values at left child and right child of vacant
               endif
      endwhile
   if (vacant is a leaf)
     then setValue(vacant,key);
end;
```

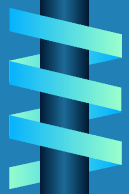
The memory complexity of the code above is O(1)!

Heap Sort: Array Implementation

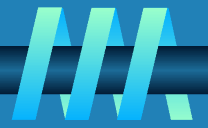


Now that fixheap on an array is done, let us consider how to implement heapSort itself w/o changing the time complexity but changing the memory complexity to O(1).

HeapSort has a loop in which we are setting aside the data at root into another array and removing the right most leaf. Then we are executing fixheap using the data at the right most leaf. In the array implementation, the right most leaf is simply A[k], where k is the last index of the array that represents the heap. Since we do not need the right most leaf, we store the data at root at the last index, and reduce the heap size by 1, so the new heap goes from 1 to k-1!



Heap Sort: Array Implementation

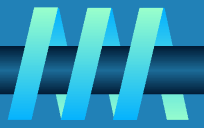


The code with the array syntax is shown below:

```
procedure heapSort (root, n);
var count:integer;
var toInsert: integer;
begin
  // Construct heap out of the elements in the tree
      rooted at 'root'.
  count := 1;
  heapSize:= n;
  while (count  ≤ n)
   toInsert:= A[n-count+1];
   A[n-count+1] := A[1];
   heapSize := heapSize-1;
   fixheap (1, toInsert);
   count:=count+1;

  endwhile;
end;
```

Since the time for fixheap is still O(logn), ignoring the time of constructHeap, the time complexity of the code above is O(nlogn). However, now there is no additional array used! Memory complexity of the code above is O(1).

Construct Heap: Array Implementation

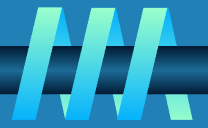


Now the question is how can we modify our recursive construct heap to be non-recursive since the recursive version would take stack space and we are looking for O(1) memory complexity. Recall that construct heap simply runs fixheap for each node starting at nodes in the lowest level and working up to the root. Since nodes starting at lowest level up to the root in an array representation is simply rightmost index to leftmost index, we rewrite our construct heap as:

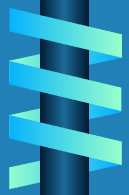
```
for i = n down to 1 do
        fixheap(i, value(i));
Endfor
```

Since fixheap takes O(1) memory, the memory complexity of the code above is O(1), while the time complexity is O(nlogn), since we run n fixheaps.

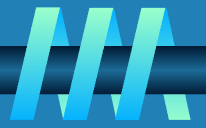
Sort Time Complexity



	Memory	Worst	Ave	Best
Smallest Based Sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n^2)$
Insertion sort	In Place	$O(n^2)$	$\theta(n^2)$	$\Omega(n)$
Mergesort	O(n logn)	O (n logn)	θ(n logn)	Ω(n logn)
Quick Sort	O(n)	$O(n^2)$	θ(n logn)	Ω(n logn)
Heap Sort	In Place	O(n logn)	θ(n logn)	Ω(n logn)



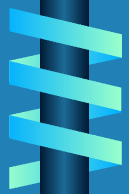
Shell Sorting- Introduction



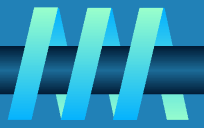
In this sorting technique, the original list is divided into sub lists containing every k^{th} element. The sub lists are sorted independently. This idea is repeated for decreasing sequence of k's called the h-sequence. The h-sequence always ends at 1.

For example, if h-sequence = (5,3,2,1). Then every 5^{th} element is sorted, then every 3^{rd}, then every 2^{nd}, and finally every element is sorted.

The number of elements in the h-sequence is denoted by t.



Shell Sorting- Demonstration-1



Let us assume that the h-sequence with t=4 is $(h_4, h_3, h_2, h_1) = (5, 3, 2, 1)$

7	19	24	13	31	8	82	18	44	63	5	29

h_4= 5; Sort every 5th element

7	19	24	13	31	8	82	18	44	63	5	29
1	2	3	4	5	1	2	3	4	5	1	2

[7,8,5] becomes [5,7,8]

[19,82,29] becomes [19,29,82]

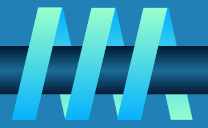
[24,18] becomes [18,24]

[13,44] stays [13,44]

[31,63] stays [31,63]

5	19	18	13	31	7	29	24	44	63	8	82

Shell Sorting- Demonstration-2



h_3= 3; Sort every 3rd element

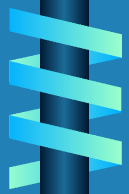
5	19	18	13	31	7	29	24	44	63	8	82
5	19	18	13	31	7	29	24	44	63	8	82
1	2	3	1	2	3	1	2	3	1	2	3

[5, 13, 29,63] stays [5,13,29,63]

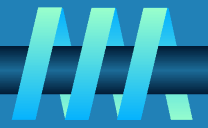
[19,31,24,8] becomes [8,19,24,31]

[18, 7,44,82] becomes [7, 18,44,82]

5	8	7	13	19	18	29	24	44	63	31	82



Shell Sorting- Demonstration-3



h_2= 2; Sort every 2nd element

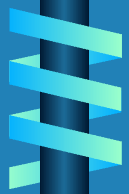
5	8	7	13	19	18	29	24	44	63	31	82

5	8	7	13	19	18	29	24	44	63	31	82
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
1	2	1	2	1	2	1	2	1	2	1	2

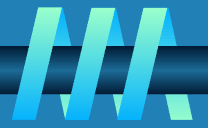
[5, 7,19,29,44,31] becomes [5,7,19,29,31,44]

[8,13,18,24,63,82] stays [8,13,18,24,63,82]

5	8	7	13	19	18	29	24	31	63	44	82



Shell Sorting- Demonstration-4



h_1= 1; Sort every element

5	8	7	13	19	18	29	24	44	63	31	82
5	8	7	13	19	18	29	24	44	63	31	82
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
1	1	1	1	1	1	1	1	1	1	1	1

[5, 8, 7, 13, 19, 18, 29,24,31,63,44, 82] becomes [5,7,8,13,18,19,24,29,31,44,63,82]

5	7	8	13	18	19	24	29	31	44	63	82

Array is now sorted!

Question: If we are going to sort every element in the end, why bother sorting every 5^{th}, 3^{rd}, 2^{nd} elements before sorting every element??

Shell Sorting-Implementation

Note that the number of swaps needed to sort the sub lists keeps decreasing as the iterations progress.

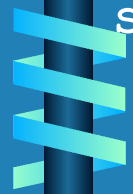
QUESTIONS:

1. Knowing the fact above,what algorithm should be used for sorting the sub lists?

2. What kind of data structure is needed to store the sub lists?

3. What is the right choice for the number of items in h-sequence and what should be the values?

4. What is the time complexity given a h-sequence of size k and an array of size n?



Shell Sorting-Analysis

Use insertion sort to sort each sub list, since it is in-place algorithm. Note that insertion can be modified to sort every k^{th} element in an array. Using mergesort, quicksort will cost memory complexity. Using heapsort will add complexity to the the algorithm.

Memory complexity is O(1) if we used insertion sort to sort the sub list.

There are no results on the time complexity of Shell Sort. Known results on shell sort complexity are:

If t=2, and if $(h_2,h_1)= (1.732n^{1/3}, 1)$ then time complexity of shell sort is $O(n^{5/3})$ and is less than $O(n^2)$.

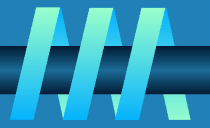
If t = logn, and $h_k = 2^k - 1$ for $1 \leq k \leq \log n$, the

Non-Comparison Based Sorting Algorithms

Bucket Sort

Radix Sorting

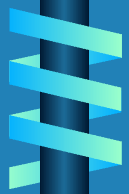
Bucket Sorting



Bucket sorting algorithms are based on the idea that data can be distributed into buckets and sorted. The basis of distribution may depend on the assumed type of data.

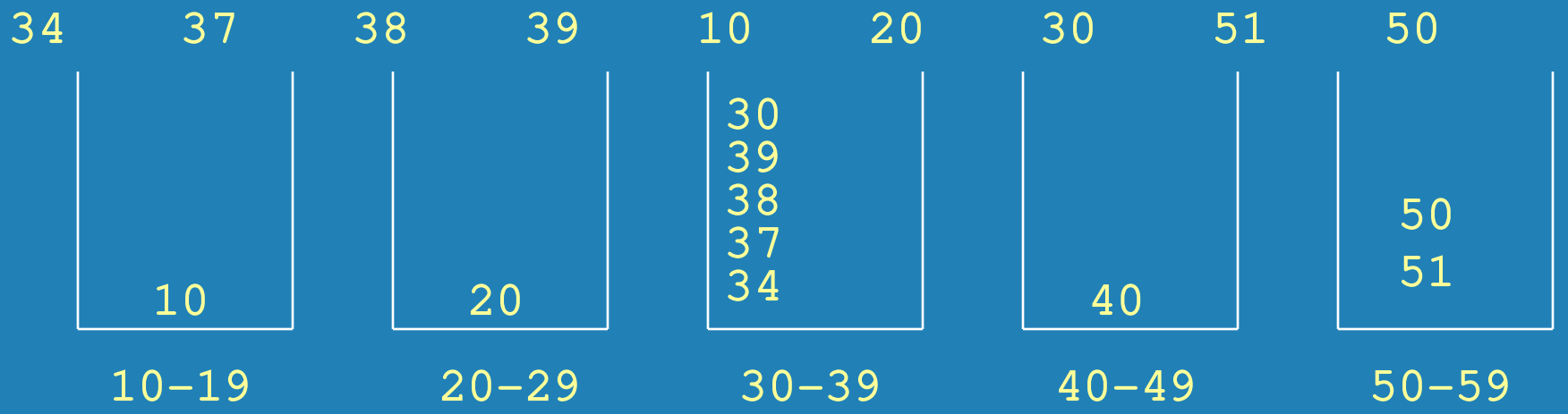
GENERAL ALGORITHM:

1. Distribute the n elements into k buckets
2. Sort each bucket
3. Merge the buckets



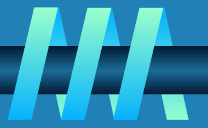
Bucket Sorting Demonstration

Let us assume that the numbers that are sorting are between 10 and 99 then using 10 buckets as shown below, we get the following distribution.



After sorting each bucket,

Bucket Sorting

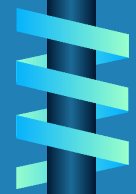


Assuming that on the average, the elements will be equally likely to go into k buckets and that the distribution per element takes only O(1) time, the time complexity can be analyzed as:

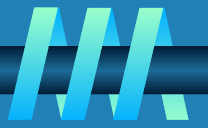
1. Distribution into k buckets -- n time
2. Sort each bucket --- k.n/k log(n/k) time
   using the best sorting algorithm,
3. Merge the buckets -- n.

Time complexity --- nlog(n/k)

Space complexity : We should such a data structure that distribution and merge take only n time. Using k size array of pointers to linked lists is appropriate.Since there are total of n elements in all the linked lists the space complexity is k+n.



Bucket Sorting

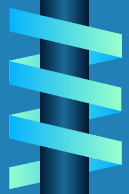


Disadvantages:

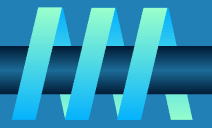
1. The buckets may not be equally distributed in the worst case every element can go in the same bucket.
2. Space is wasted since each bucket should have at least one space in the array
3. The bucket sorting works only if the data type is an integer.

What is the worst case time and space complexities?

What is the best case time and space complexities?



Radix Sorting



Radix sorting is a form of bucket sorting. Since we are looking for time complexity that is less than O(nlogn) and we know the type of data to be sorted, let us assume that we know the number of digits in the decimal data.

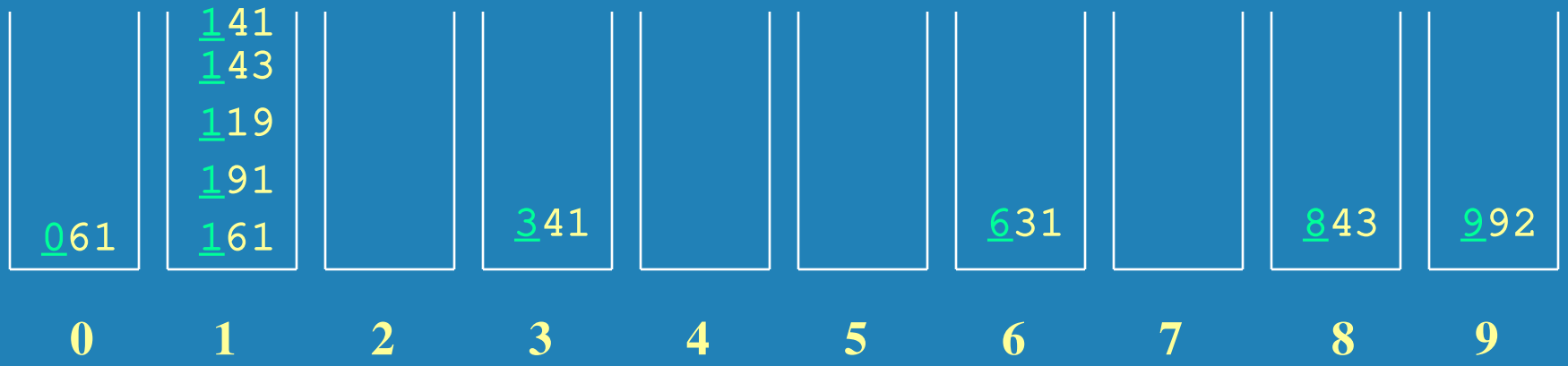
We will be using 10 buckets in the radix sort, since we know that are 10 possibilities for each digit in a decimal number.

Most significant radix sort, and least significant radix sort are two variations of radix sort.

Most Significant Radix Sort Demonstration-1

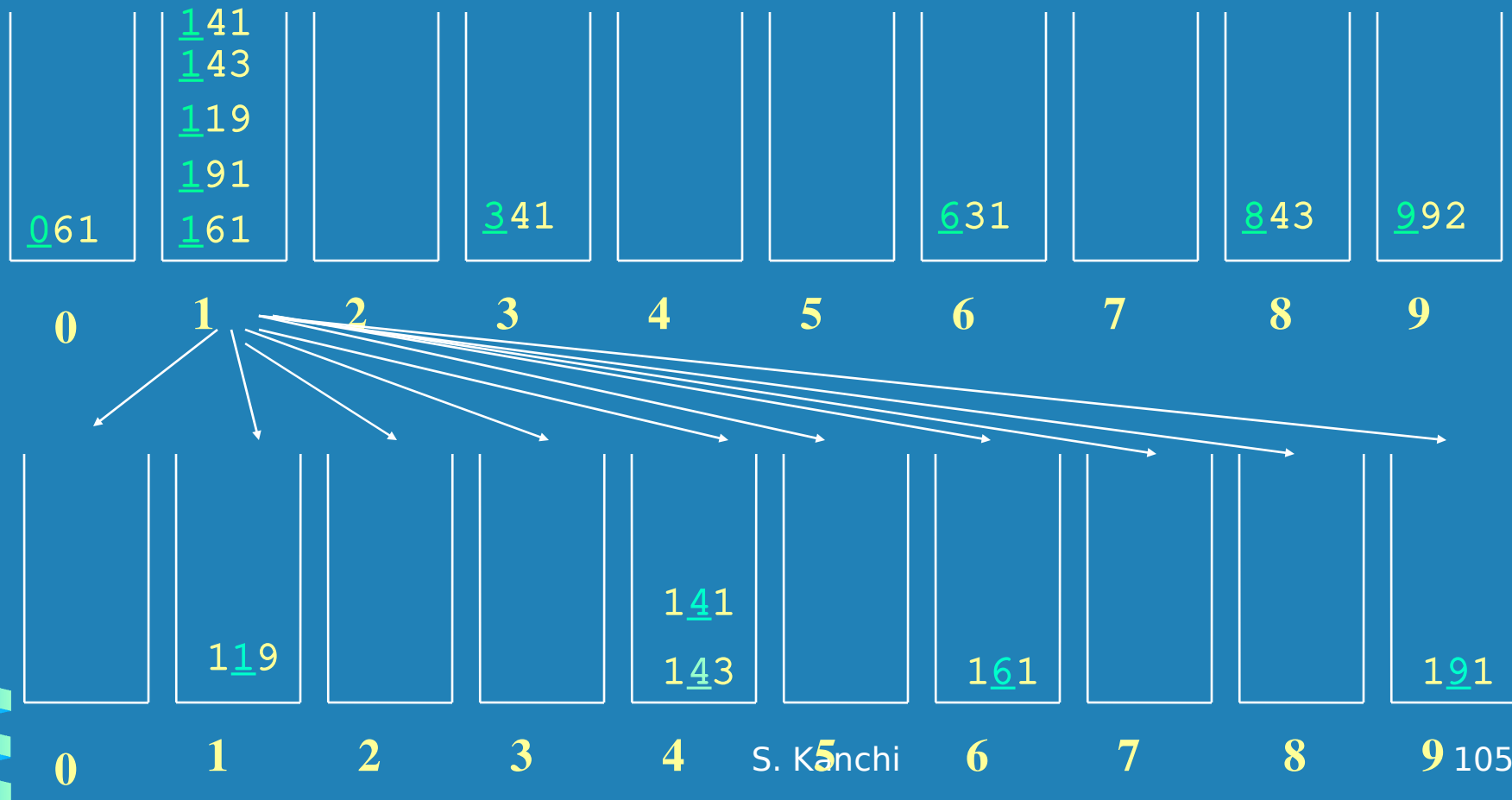
061 161 191 341 631 992 843 119 143 141

Distribute the numbers from left to right on the basis of the most significant that is 1st digit.



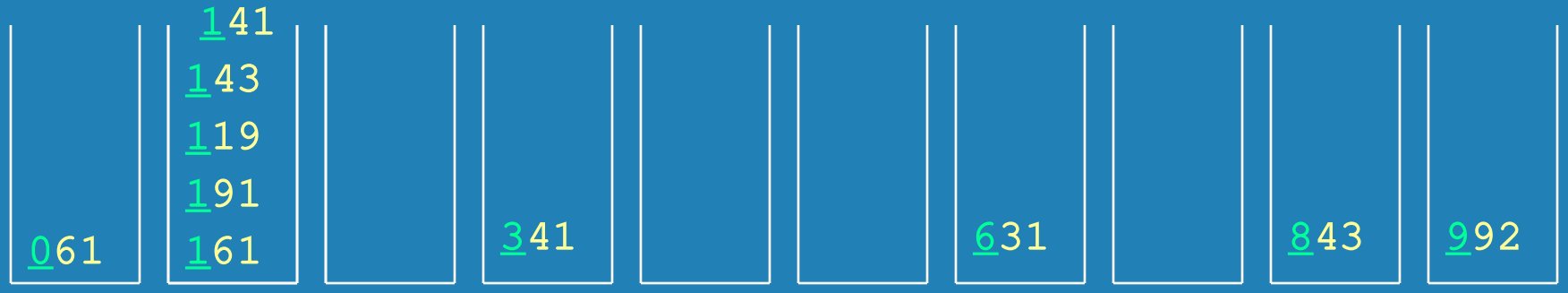
Most Significant Radix Sort Demonstration-2

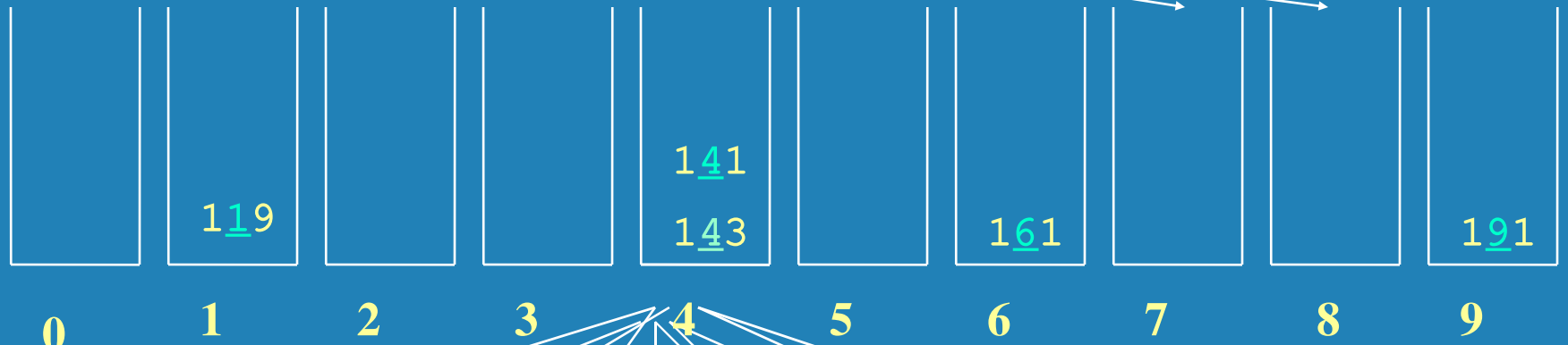
For each bin that has more than one element, make 10 additional bins and distribute the numbers using the next most significant digit, that is the 2nd digit.

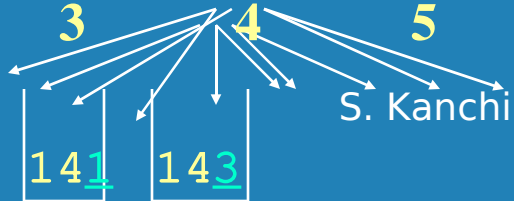


Most Significant Radix Sort Demonstration-3

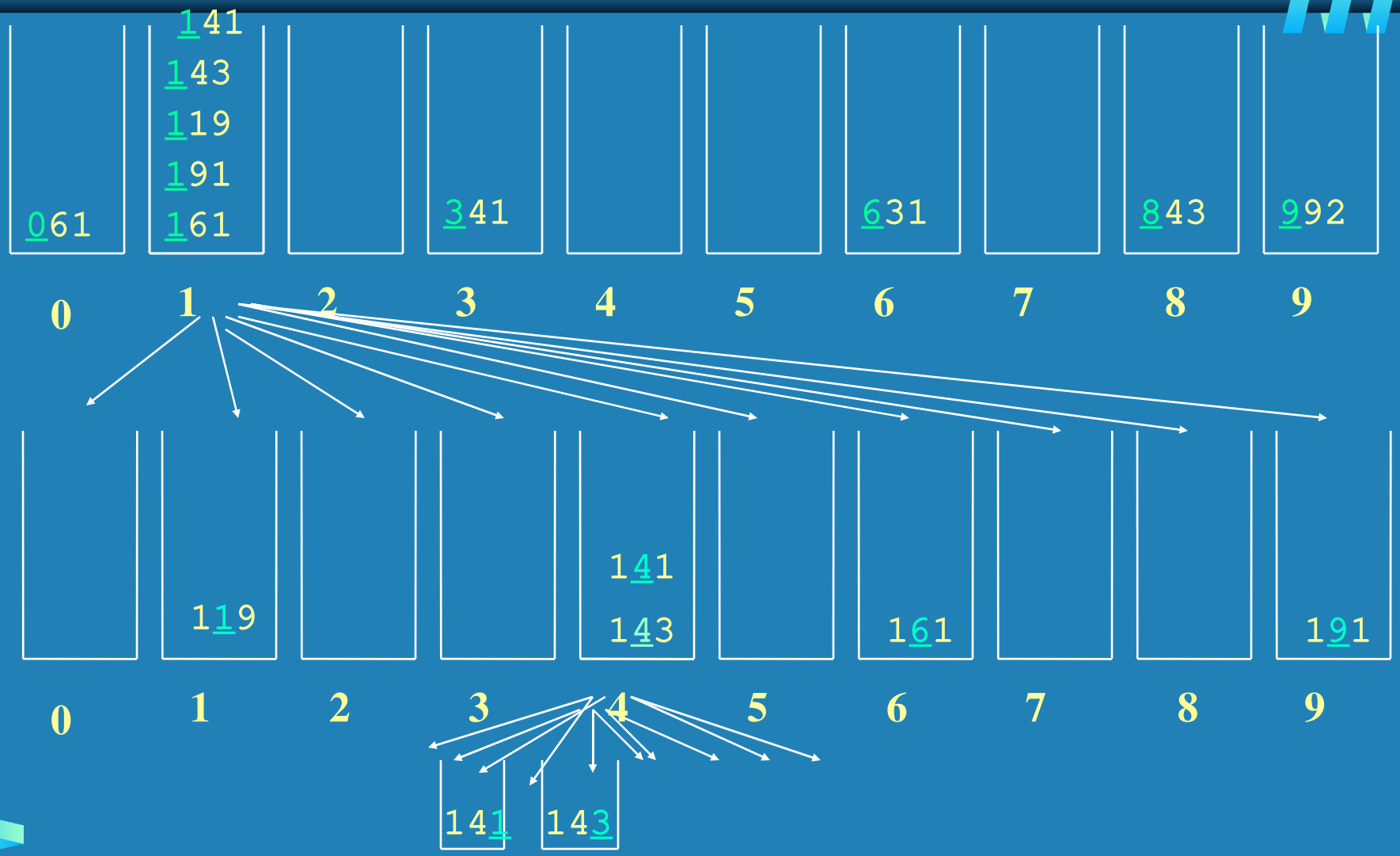
Since bin #4 has two elements we create 10 more bins and distribute using 3rd digit.







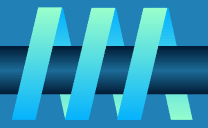
Most Significant Radix Sort Demonstration-4



Now simply read the bins from left to right to get sorted elements:

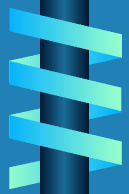
061 119 141 143 161 191 341 631 843 992

Most significant Radix sort



There is too much book-keeping for the pointers.

Time-complexity = n * r, if there r radixes.

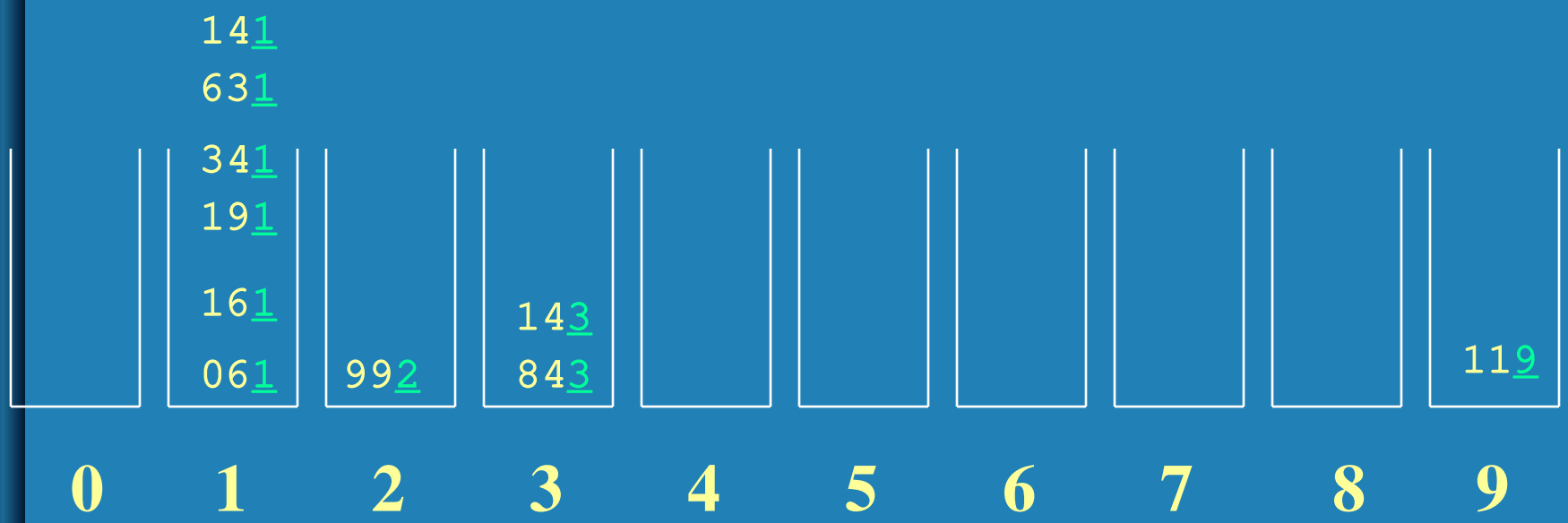


Least Significant Radix sort Demonstration-1

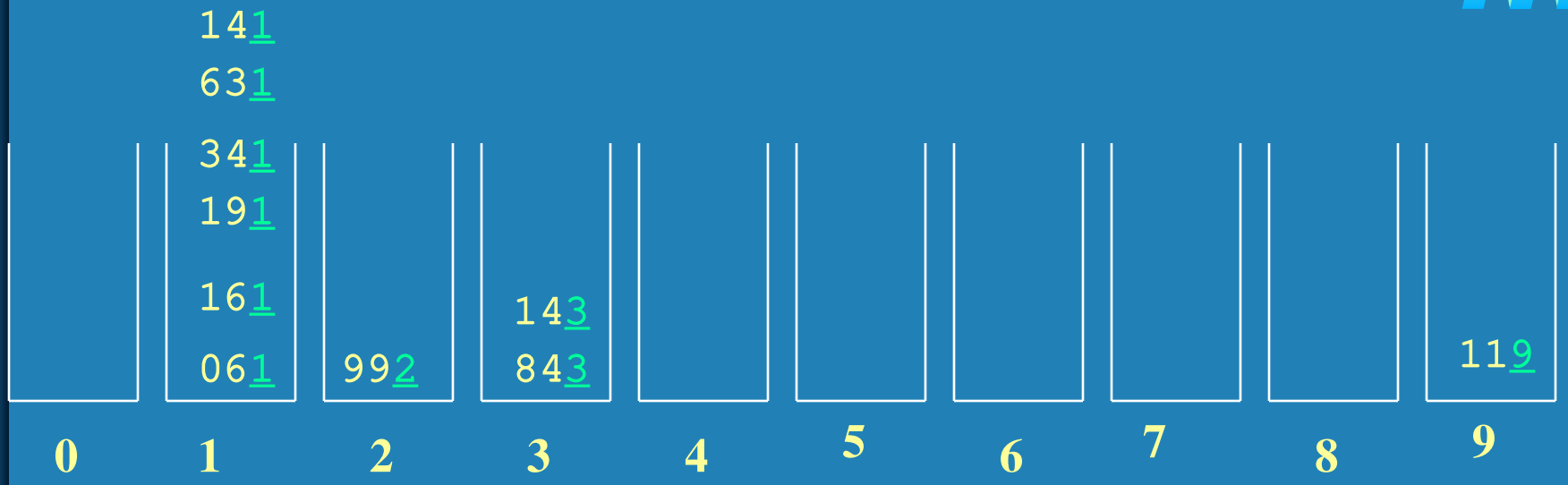
The idea in least significant sort is simply to start with the least significant digit for radix sorting.

061 161 191 341 631 992 843 119 143 141

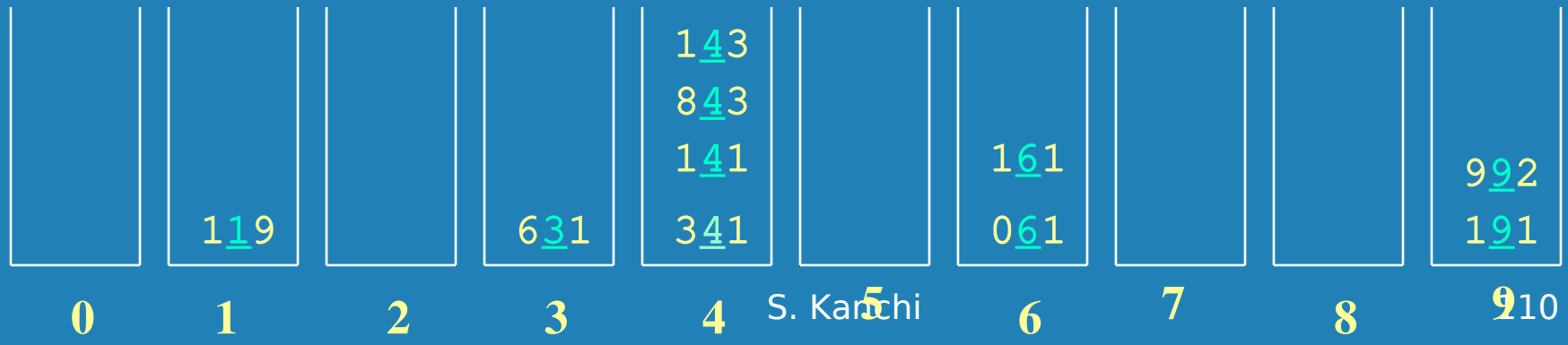
Distribute the numbers from left to right on the basis of the least significant that is 3rd digit.



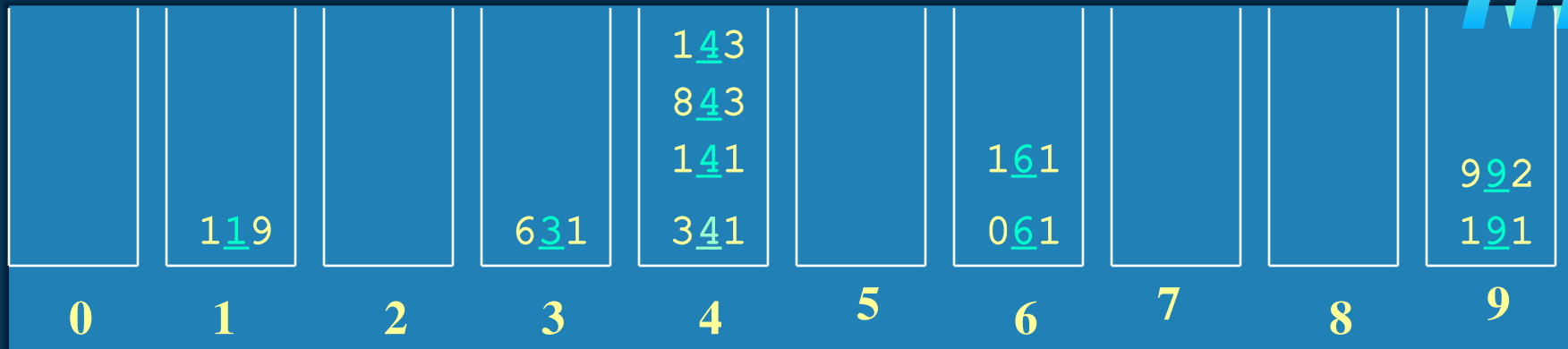
Least Significant Radix sort Demonstration-2



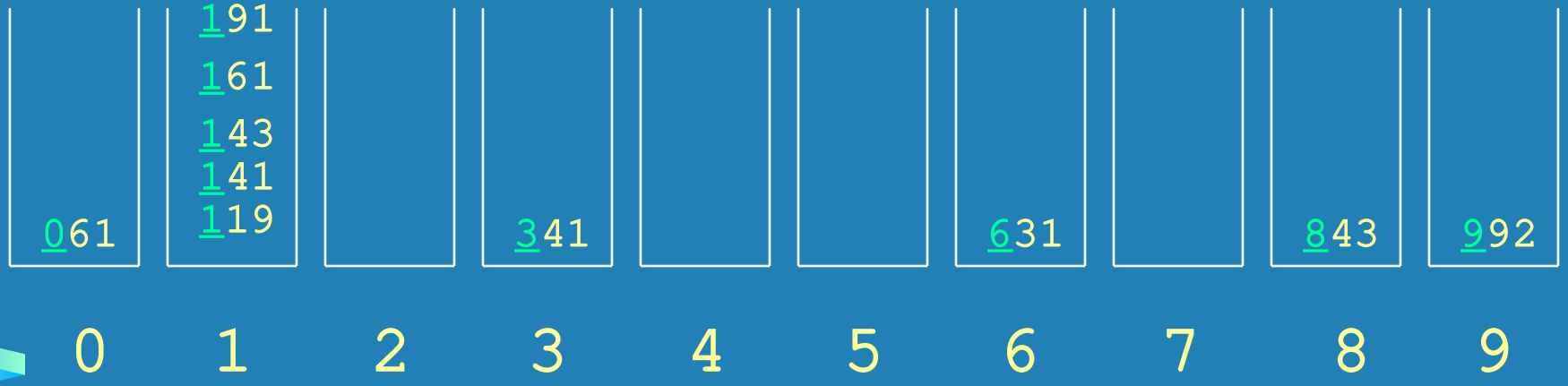
Distribute the numbers from left to right on the basis of the next least significant that is 2nd digit.



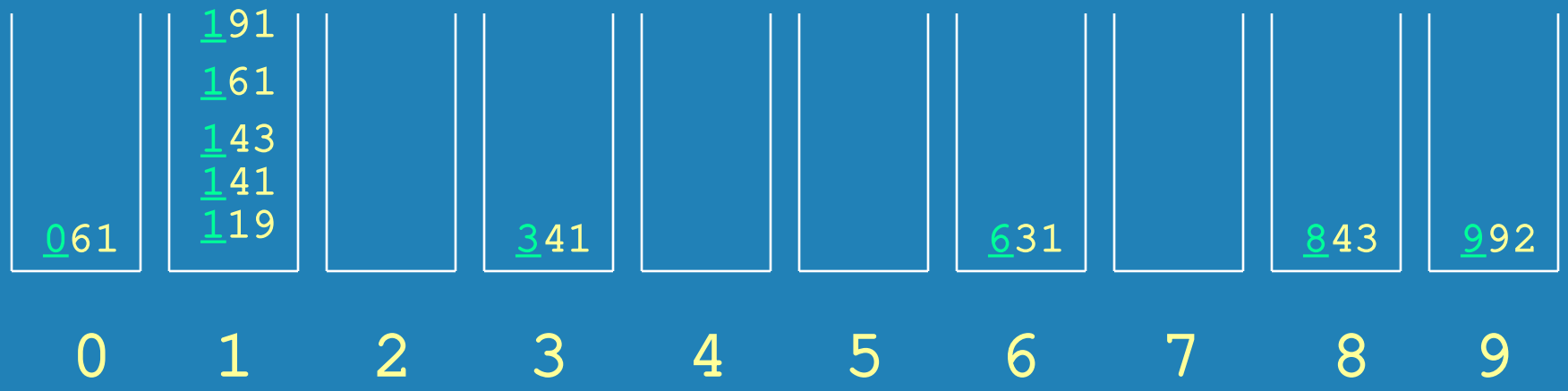
Least Significant Radix sort Demonstration-3



Distribute the numbers from left to right on the basis of the next least significant that is 1st digit.



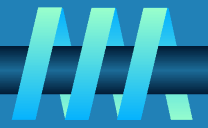
Least Significant Radix sort Demonstration-4



Now read the numbers from left to right to get a sorted array:

061 119 141 143 161 191 341 631 843 992

Least Significant Radix Sort- Time complexity

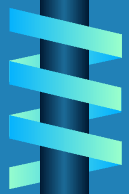


First of all the memory complexity is very less, since the same bins can be reused in alternate iterations.

The number of iterations is the number of radixes. Let us say r. Each iteration involves looking at the n numbers. Therefore the time complexity is O(nr).

If r is considered a constant, then we have a O(n) sorting algorithm!.

But is r really a constant?



Ch. 3 Exercises

1. Find the time complexity of quick sort when split takes O ($n^{.9}$) time and split in the middle.

2. What is the worst case space complexity of Quicksort. Explain your answer.

3. What is the best case space complexity of Quicksort. Explain your answer.

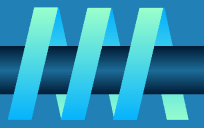
4. Write the merge code for mergesort. Analyze the time complexity of merge algorithm.

5. Convert each sorting algorithm to use linked list instead of an array. Analyze the new time and memory complexity.

6. Run the smallest based sort on the following array. Show the pointers small, smallestIndex and fix.

56	45	67	78	90	34	45	8

Ch. 3 Exercises



7. Run the insertion sort on the following array. Show the i and j .

56 45 67 78 90 34 45

8. Show the demonstration of mergesort on the following array.

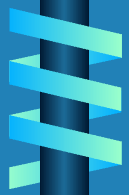
56 45 67 78 90 34 45 8

9. Demonstrate the split algorithm on the following array. Show the pointers s, y low and high.

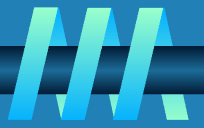
56 45 67 78 90 34 45 8

10. Demonstrate the quicksort algorithm on the following array. Show all the split algorithms and the low and high for each call.

56 45 67 78 90 34 45



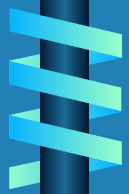
Ch. 3 Exercises



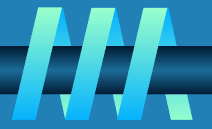
11. Show that it is impossible to have a split algorithm for quicksort that takes $O(n^{.9})$ time and always splits in the center of the array.

12. Describe the worst case and best case time complexity scenarios of quicksort and mergesort and derive the best case and worst case time complexities.

13. Although we have shown that O(n) is the worst case space complexity of Quicksort, it is possible to reduce this to O(logn). Find the technique that achieves the O(logn) memory complexity.

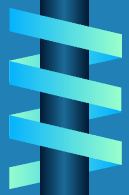


Ch. 3 Exercises



14. A sorting method is "stable" if equal keys remain in the same relative order in the sorted list as they were in the original list. That is, let us say that L[i] = L[j] and i < j in the original list and after the sorting,
let us say L[i] was moved to L[k] and L[j] was moved to L[m], then sort is stable only if k < m. Which of the following sorting algorithms are stable? For the ones that are not stable give an example to indicate why it is not stable.

a. Insertion Sort
b. Smallest Based sort
c. Bubblesort
d. Quicksort
e. Heapsort
f. Shellsort
g. Radix sort
h. Mergesort



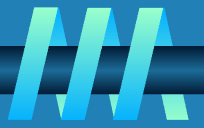
Ch. 3 Exercises

15. The problem is to sort 500 exam papers alphabetically by students last name. The sorting will be done in the office with two desktops temporarily cleared of all other papers, books and coffee cups. It is 1AM and the person wants to go home as soon as possible. Which sorting algorithm would you recommend.

16. Give the details of quicksort when used on a singly linked list of n elements. Rewrite the quicksort and split algorithms in the linked list notation to answer this question.

17. Show that execution of quicksort and heapsort on the following array.

Ch.3 Exercises



18. Show that 2n memory is enough for mergesort. Rewrite the algorithm to switch between two arrays.

19. If there are repeated elements in the array, which algorithm needs a change among the smallest based sort, insertion sort, quicksort, mergesort and heapsort? Is there any change in time complexities.