DATA STRUCTURES AND ALGORITHMS

Fast sorting algorithms

Shellsort, Mergesort, Quicksort

Summary of the previous lecture

- Why sorting is needed ?
  - Examples from everyday life
- What are the basic operations in sorting procedure?
- Simple sorting algorithms
  - Selection sorting
  - Insertion sorting
  - Bubble sorting

Running time of the simple sorting

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

ShellSort

Shellsort is sorting algorithm invented by **D.L. Shell**.
It is also sometimes called the **diminishing increment sort**.

Shellsort also exploits the best-case performance of **Insertion Sort**.

Shellsort's strategy is to make the list "**mostly sorted**" so that a final **Insertion Sort** can finish the job.

When properly implemented, Shellsort will give substantially better performance than $O(n^2)$ in the worst case.

Principle of ShellSort

Shellsort uses a process that forms the basis for many of the quick sort algorithms:

- Break the list into sublists,
- Sort sublists independently,
- Recombine the sublists
- Repeat sorting loop on newly breaked sublists

Shellsort breaks the array of elements into “virtual” sublists. Each sublist is sorted using an **Insertion Sort**. Another group of sublists is then chosen and sorted, and so on.

Algorithm is not stable.

Example

Let us assume that **N** is the number of elements to be sorted, and $\mathbf{N = 2^k}$.

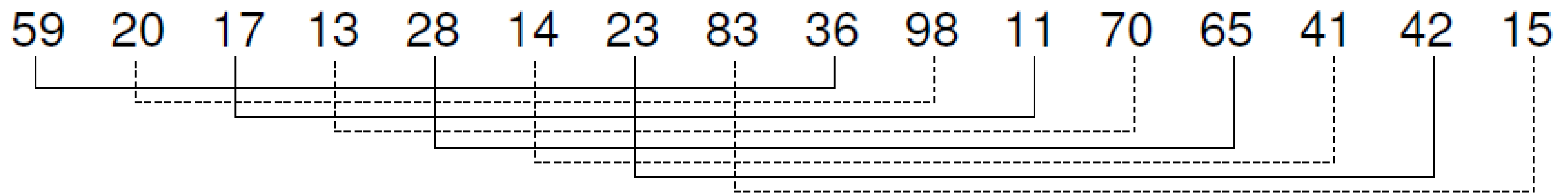
One possible implementation of Shellsort will begin by breaking the list into N/2 sublists of 2 elements each, where the array index of the 2 elements in each sublist differs by N/2.

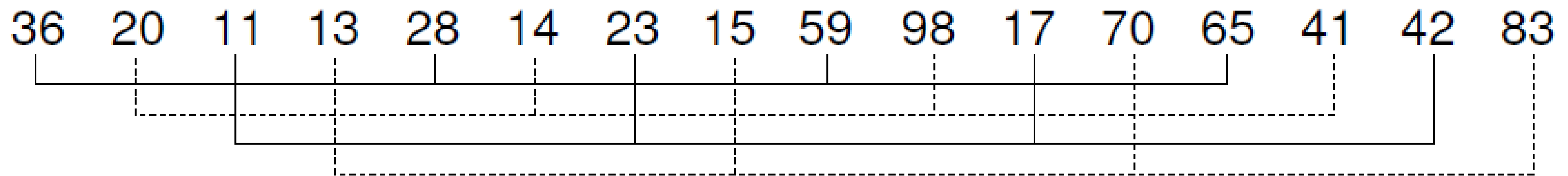
$$\mathbf{j = i + N/2}$$

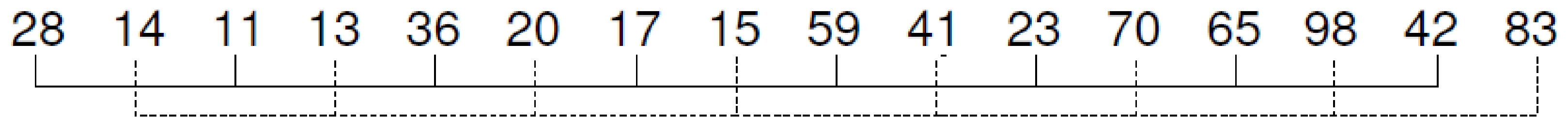
If there are 16 elements in the array indexed from 0 to 15, there would initially be 8 sublists of 2 elements each.
The first sublist would be the elements in positions 0 and 8, the second in positions 1 and 9, and so on.

Each list of two elements is sorted using **Insertion Sort**.

Example (cont.)







ShellSort

```
void Shellsort (int v[ ], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for ( i = gap; i < n; i++)
            for ( j = i-gap; j >=0 && v[ j ] > v[ j+gap ];  j -= gap) {

                temp = v[ j ];
                v[ j ] = v[ j+gap ];
                v[ j+gap ] = temp;

            }
}
```

ShellSort properties

Running time of shell sort depends on the increment sequence. For the increments *1; 4; 13; 40; 121; ...*,

$$\mathbf{n_j = 3n_{j-1} + 1}, \quad \text{where } \mathbf{n_0 = 0}$$

which is what is used here, the running time is $O(n^{1.5})$.

For other increments, time complexity is known to be $O(n^{4/3})$ and even $O(n \cdot lg_2(n))$.

Shellsort illustrates how we can sometimes exploit the special properties of an algorithm (Insertion Sort) even if in general that algorithm is unacceptably slow.

MergeSort

A natural approach to problem solving is **divide and conquer**.

In terms of sorting, we might consider:

- breaking the list to be sorted into pieces (*divide step*)
- process sorting on the pieces (*conquer step*)
- put pieces back together somehow (*combine step*)

A simple way to do this would be to split the list in half, sort the halves, and then merge the sorted halves together.

This is the idea behind **Mergesort**.

Mergesort was invented by **John von Neumann** in 1945.

MergeSort in more detail

Since we are dealing with subproblems, we state each subproblem as sorting a subarray ***A*[*p*].**

Initially, $p = 1 \ldots n$, but p values change as we recurse through subproblems. To sort $A[p]$ the following steps must be produced:

1. **Divide Step**

If a given array ***A*** has zero or one element, simply return; it is already sorted. Otherwise, split $A[p]$ into two subarrays $A[p .. q]$ and $A[q + 1 .. n]$, each containing about half of the elements of $A[p .. n]$. That is, q is the halfway point of $A[p]$.

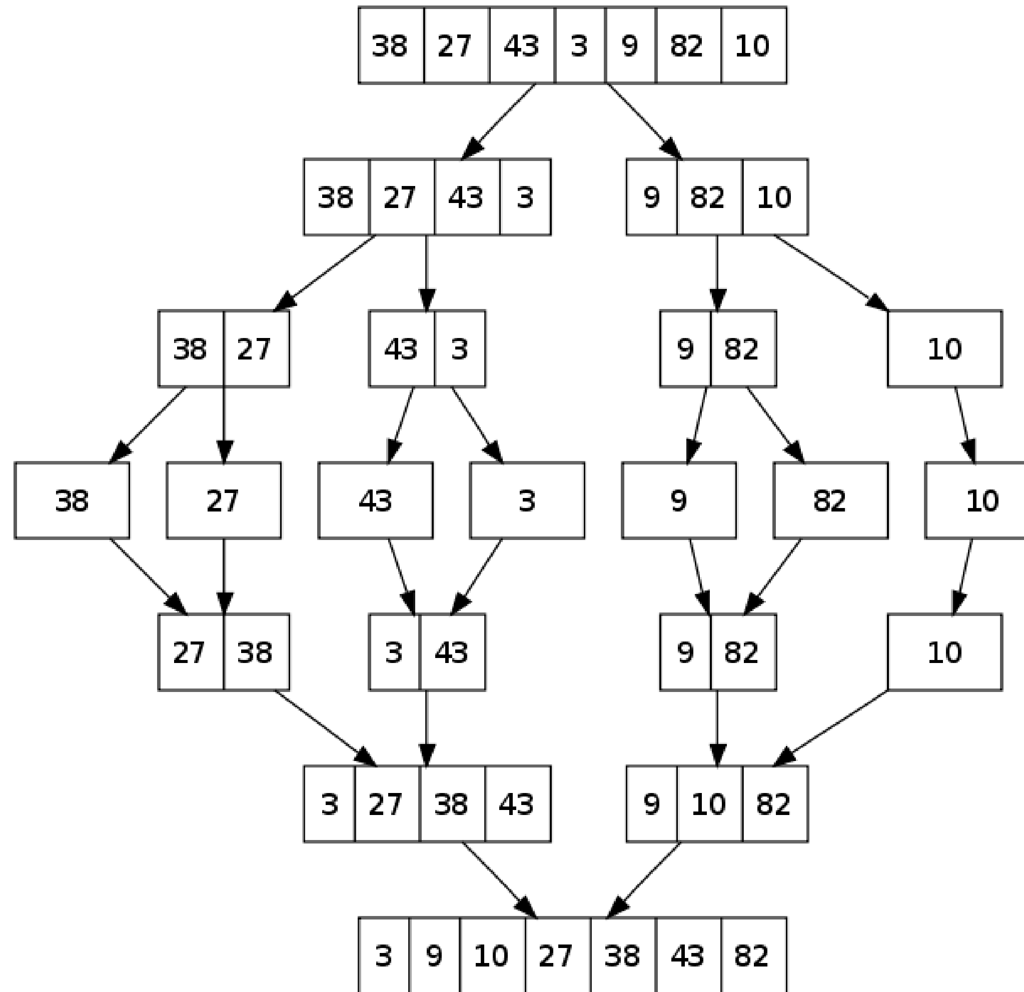
2. **Conquer Step**

Conquer by recursively sorting the two subarrays $A[p ... q]$ and $A[q + 1 .. n]$.

3. **Combine Step**

Combine the elements back in $A[p]$ by merging the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. n]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

MergeSort



Nice example:

http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/mergeSort/mergeSort.html

MergeSort running time

Assume that n is a power of 2 so that each divide step yields two subproblems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

Divide: Just compute q as the average of p and n, which takes constant time i.e. O(1).
Conquer: Recursively solve 2 subproblems, each of size $n/2$, which is $2T(n/2)$.
Combine: MERGE on an n-element subarray takes $O(n)$ time.

MergeSort running time

Summed together they give a function that is linear in *n*, which is O(*n*).

Therefore, the recurrence for merge sort running time is:

$$T(n) = O(n), \quad \text{if } n=1$$
$$T(n) = 2T(n/2) + O(n), \quad \text{if } n > 1$$
$$T(n) = O(n \log_2 n), \quad \text{if (recursive)}$$

MergeSort code

```
#include <stdio.h>

void mergeSort (int numbers[ ], int temp[ ], int array_size);
void m_sort       (int numbers[ ], int temp[ ], int left, int right);
void merge       (int numbers[], int temp[], int left, int mid, int right);

int main()
{
            int array1 [5] = {65, 72, 105, 55, 2};
            int temp_array [5];

            mergeSort (array1, temp_array, 5);
            printf(“Sorted array\n\n”);
                for (int i = 0; i < 5; i++)
                {
                        printf(“%d”, array1[i] );
                }
            return 0;
}
```

MergeSort code (cont.)

```
void mergeSort ( int numbers[ ], int temp[ ], int array_size)
{
    m_sort (numbers[ ], temp[], 0, array_size - 1);
}
```

```
void m_sort ( int numbers[ ], int temp[ ], int left, int right )
{
  int mid;

  if (right > left)
  {
    mid = (right + left) / 2;
    m_sort (numbers, temp, left, mid);
    m_sort (numbers, temp, (mid+1), right);

    merge(numbers, temp, left, (mid+1), right);
  }
}
```

MergeSort code (cont.)

```
void merge ( int numbers[ ], int temp[ ], int left, int mid, int right )
{
  int i, left_end, num_elements, tmp_pos;
  left_end = (mid - 1);
  tmp_pos = left;
  num_elements = (right - left + 1);

  while ( (left <= left_end) && (mid <= right) )
  {
    if (numbers [left] <= numbers [mid])
    {
      temp [tmp_pos] = numbers [left];
      tmp_pos += 1;
      left += 1;
    }
    else
    {   temp [tmp_pos] = numbers [mid];
        tmp_pos += 1;
         mid += 1;
    }  }
```

MergeSort code (cont.)

```
while (left <= left_end)
 {
   temp [tmp_pos] = numbers[left];
   left += 1;
   tmp_pos += 1;
 }
 while (mid <= right)
 {
   temp [tmp_pos] = numbers[mid];
   mid += 1;
   tmp_pos += 1;
 }
 // modified
 for (i=0; i < num_elements; i++)
 {
   numbers [right] = temp [right];
   right -= 1;
 } }
```

Quicksort

Quicksort is named so because, when properly implemented, it is the **fastest known general-purpose in-memory sorting algorithm** in the average case. It does not require the extra array needed by Mergesort, so it is **space efficient** as well.

Quicksort uses the the same **divide and conquer** principle.

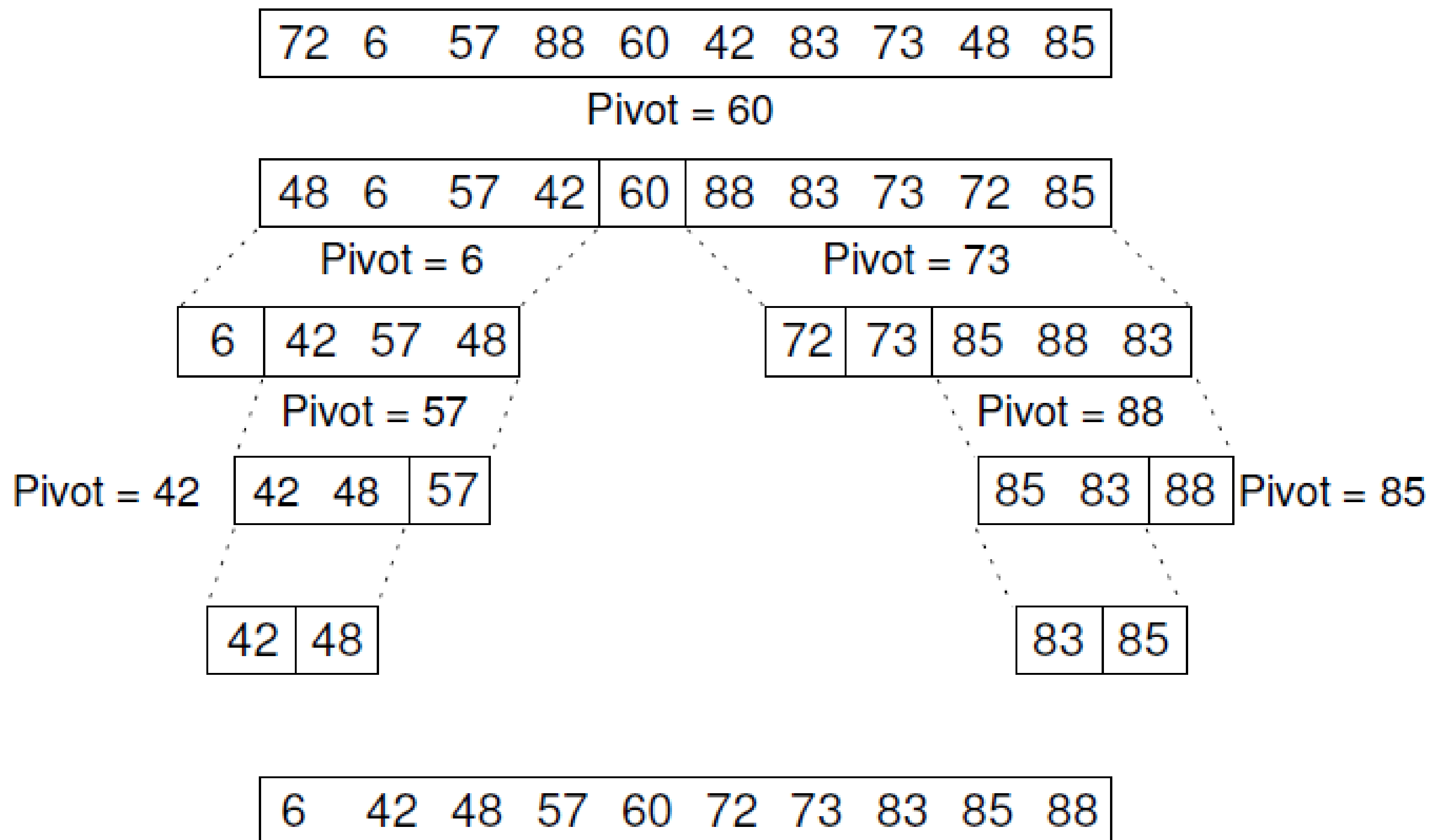
Quicksort algorithm was developed by Tony Hoare (in USSR) in 1960.

Quicksort

The steps of Quicksort algorithm are:

- Pick an element, called a **pivot**, from the list.
- Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
- Recursively sort the sublist of lesser elements and the sublist of greater elements.

Quicksort example



Quicksort

Quick sort works by partitioning a given **array *A*[*p*], p=0, .. r** into two non-empty subarray $A[p\ ..\ q]$ and $A[q+1\ ..\ r]$ such that every key in $A[p\ ..\ q]$ is less than or equal to every key in $A[q+1\ ..\ r]$.

The exact position of the partition depends on the given array and index **q** is computed as a part of the partitioning procedure.

QuickSort

If $p < r$ then

q = Partition (A, p, r)

Recursive call to Quick Sort (A, p, q)

Recursive call to Quick Sort (A, $q + r$, r)

Partitioning procedure

1 | 12 | 5 | 26 | 7 | 14 | 3 | 7 | 2 — unsorted

1 | 12 | 5 | 26 | 7 | 14 | 3 | 7 | 2 — pivot value = 7

i (at 1), pivot value (at 7), j (at 2)

1 | 12 | 5 | 26 | 7 | 14 | 3 | 7 | 2 — $12 \geq 7 \geq 2$, swap 12 and 2

i (at 12), j (at 2)

1 | 2 | 5 | 26 | 7 | 14 | 3 | 7 | 12 — $26 \geq 7 \geq 7$, swap 26 and 7

i (at 26), j (at 7)

1 | 2 | 5 | 7 | 7 | 14 | 3 | 26 | 12 — $7 \geq 7 \geq 3$, swap 7 and 3

i (at 7), j (at 3)

1 | 2 | 5 | 7 | 3 | 14 | 7 | 26 | 12 — i > j, stop partition

j (at 3), i (at 14)

1 | 2 | 5 | 7 | 3   14 | 7 | 26 | 12 — run quick sort recursively

...

1 | 2 | 3 | 5 | 7 | 7 | 12 | 14 | 26 — sorted

Choice of Pivot

Possibilities of pivot choice:

- the leftmost element (worst-case behavior on already sorted arrays)
- the rightmost element (worst-case behavior on already sorted arrays)
- random index of the array
- middle index of the array
- median of the first, middle and last element of the partition for the pivot (especially for longer partitions).

Quicksort running time

Worst case performance

$T(n) = O(n^2)$

Best case performance

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

Average case performance

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

Quicksort code

```
void quickSort(int arr[ ], int left, int right) {
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

        while (i <= j) {                    // partition
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    }
```

Quicksort code (cont.)

```
/* recursion */

    if (left < j)
        quickSort(arr, left, j);

    if (i < right)
        quickSort(arr, i, right);

}
```