Lecture 2
Data Structures (DAT037)

Ramona Enache
(with slides from Nick Smallbone)

Update on complexity issue

```
for (i=0; i<n; i++)
    for(j =0; j<n; j++)
      if (a[i][j]==0) then
         for (i1=0; i1<n; i1++)
           for (j1=0; j1<n; j1++)
                if (a[i1][j1] ==0 && (i1 != i || j1 != j))
                    then return 0;
return 1;
```

Cases:

- no zero: n^2 comparisons (all false)
- one zero: <= n^2 comparisons(all false, except the last – find the zero) + n^2 comparisons (all false)
- at least two zeros < n^2 comparisons (all false, except the last – find 1st zero) + <= n^2 comparisons (all false, except the last – find 2nd zero)

Update on complexity issue

Best case - $O(1)$

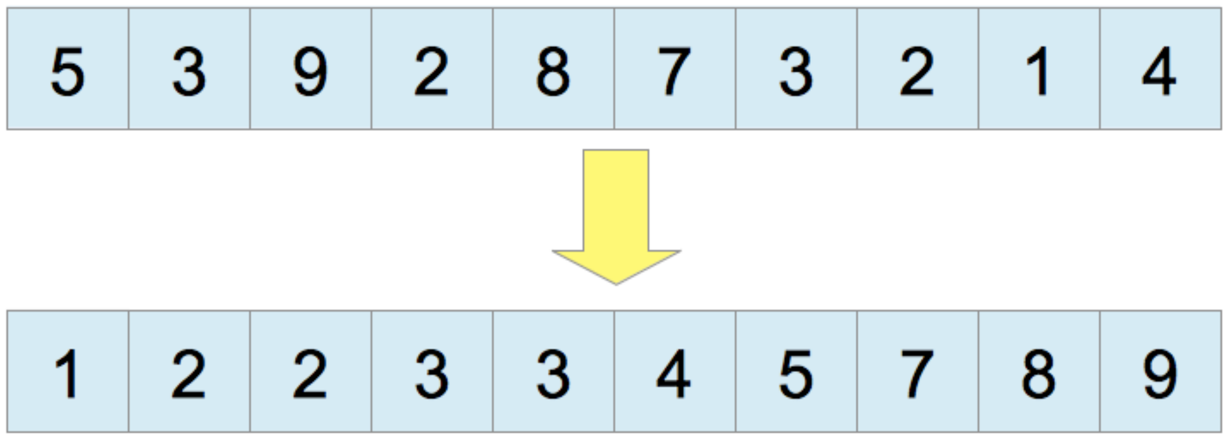
first two elements are 0

Worst case $O(n^2)$

~$2n^2$ steps (last element is 0, rest are not)

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

Sorting



Why Sorting ?

- Easier to perform further operations on a sorted array

! Remember the **min** problem from last time!

O(1) – sorted array

O(N) – unsorted array

Why Sorting ?

- Easier to perform further operations on a sorted array

  + searching: O(N) vs O(log N)

  Prove it!

  + finding duplicates: O(N^2) vs O(N)

Sorting

- Most sorting algorithms are based on comparisons

+ they can be used for any kinds of elements

- more specialized input is easier to optimize -> Later

Insertion Sort

Insertion Sort

- Imagine that someone is dealing you cards
- Whenever you get a new card, you put it into the right place in your hand

Insertion Sort

- Basic algorithm (7.2.2)

```
for (i = 0; i < v.length; i++){
  tmp = v[i];
  for(j=i; j>0 && tmp < a[j-1]; j--)
        a[j] = a[j-1];
  a[j] = tmp ;
}
```

Example

Sorting

5	3	9	2	8

:

Start by “picking up” the 5:

5

Example

5	3	9	2	8

Insert the 3 into the correct place:

3	5	9	2	8

Example

3	5	9	2	8

Insert the 9 into the correct place:

3	5	9	2	8

Example

3	5	9	2	8

Insert the 2 into the correct place:

2	3	5	9	8

Example

2	3	5	9	8

Insert the 8 into the correct place:

2	3	5	8	9

Intuition

- For each iteration with **i**, we assume that v is sorted between 0 and i-1 (not necessarily the final order from the array)
- We add v[i] so that v is sorted between 0 and i, by moving the larger elements 1 position to the right

Complexity

- The inner loop (with j) takes at most i iterations each time

$$T(N) <= 1 + 2 + \ldots (N-1) = N(N-1)/2$$

So, T(N) is O(N^2)

Complexity

- Best case scenario – O(N)

Why ?

Question

- Which of the arrays fall in the best-case scenario ?
    a) [1,2,3,4]
    b) [2,4,1,3]
    c) [4,3,2,1]
    d) [1,3,2,4]

govote.at
code 192509

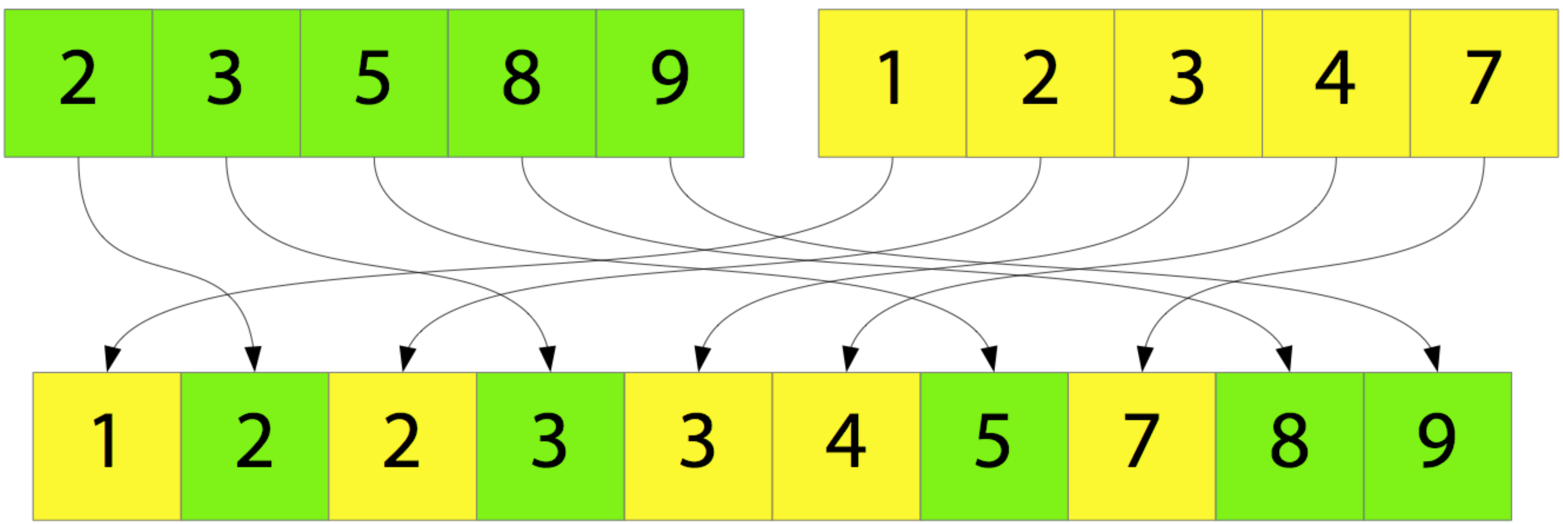
Question

- Which of the arrays fall in the worst-case scenario ?
    a) [1,2,3,4]
    b) [2,4,1,3]
    c) [4,3,2,1]
    d) [1,3,2,4]

govote.at
Code 789176

Merge sort

We can *merge* two sorted lists into one in linear time:

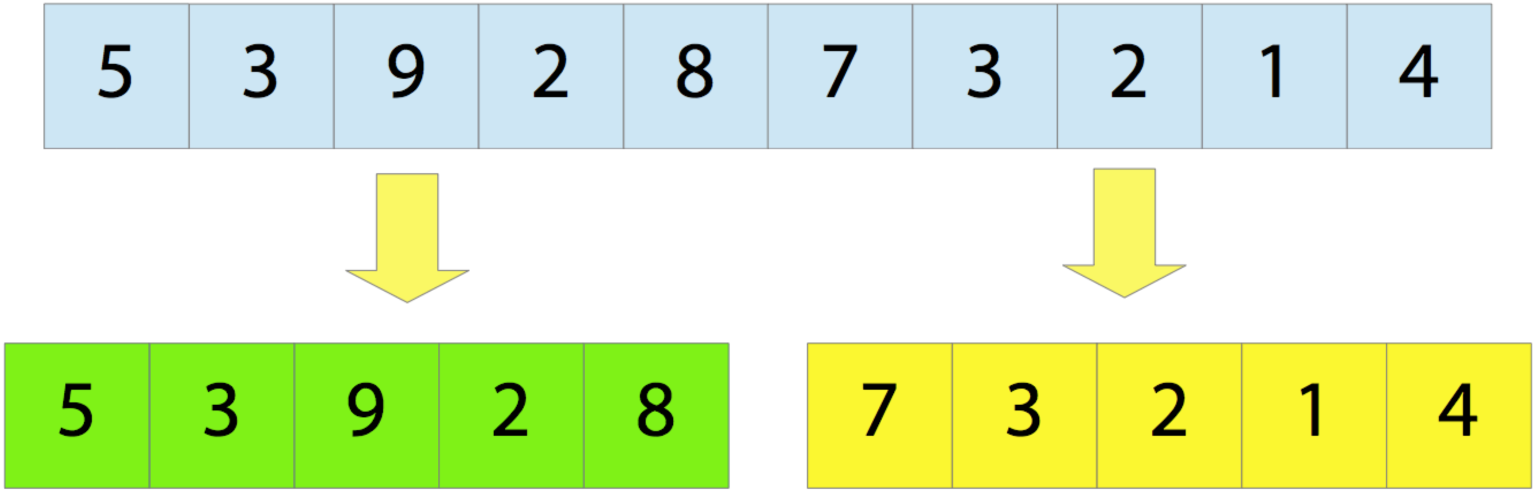


Merge sort

- Split the list into 2 equal parts
- Recursively merge sort the 2 parts
- Merge sorted lists together

Merge sort

1. *Split* the list into two equal parts

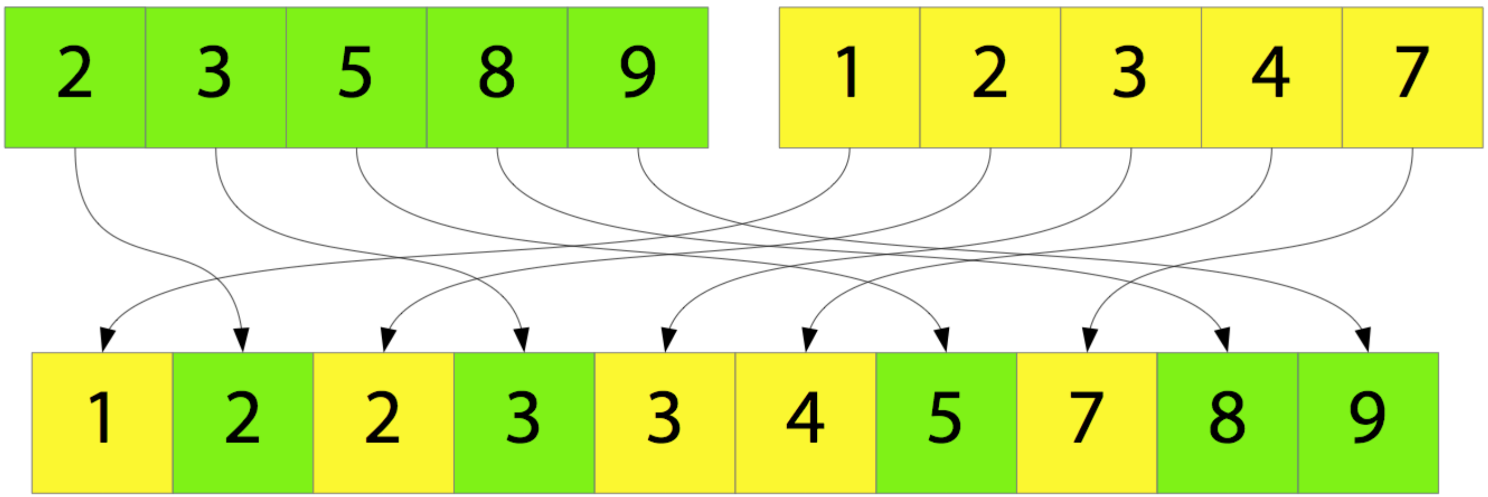


Merge sort

2. *Recursively* mergesort the two parts

Merge sort

3. *Merge* the two sorted lists together



Merge sort

```
mergeSort (v[], tmp[], left, right){
 if (left < right){
   center = (left + right)/2;
   mergeSort(v, tmp, left, center);
   mergeSort(v, tmp, center+1, right);
   merge(v, tmp, left, center+1, right);
 }
}
```

Merge sort

```
merge (v[], tmp[], leftPos, rightPos, rightEnd){
  leftEnd = rightPos - 1;
  tmpPos = leftPos;
  numElems = rightEnd - leftPos + 1

 while (leftPos <= leftEnd && rightPos <= rightEnd)
    if (v[leftPos] < v[rightPos]) tmp[tmpPos++]= v[leftPos]++;
      else tmp[tmpPos++] = v[rightPos++];

  while (leftPos <= leftEnd)
      tmp[tmpPos++] = v[leftPos++];
   while (rightPos <= rightEnd)
      tmp[tmpPos++] = v[rightPos++];

    for (i=0; i<numElems; i++, rightEnd--)
       v[rightEnd] = tmp[rightEnd]
}
```

Merge loop

Add remaining elements

Copy back to original array

Complexity of Merge sort

merge is O(N)

mergeSort is O(?)

Complexity of Merge sort

Let $T(N)$ be the complexity of merge sort

$T(1) = 1$

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

recursive call of mergeSort

merge

Complexity of Merge sort

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

$= N + 2(N/2 + 2(T/4)) = 2N + 4T(N/4)$

$T(N) = kN + 2^kT(N/2^k)$

for $k >= 1$, $k <= \log N$

Prove it
by induction!

Complexity of Merge sort

for k = log N

T(N/2^k) = T(1) = 1

So

T(N) = N log N + 1 => T(N) is O(N log N)

For N != 2^N, k = `ceil (log N)`

Complexity of Merge sort

Best time complexity

- best and worst case complexity – O(N log N)
- not in place – O(N) extra space (tmp)
- only sequential access to the list – good for functional programming

Dynamic Arrays

- A **dynamic array** is an array which can be resized.
- It contains a variable for storing the size of the used part of the array
- add copies the array when it gets full but doubles the size of the array each time

Dynamic Arrays

Dynamic arrays provide

- indexing – O(1)
- insert/delete at first – O(N)
- **insert/delete after – O(1)**

Amortized complexity

Dynamic Arrays

Dynamic arrays in Java

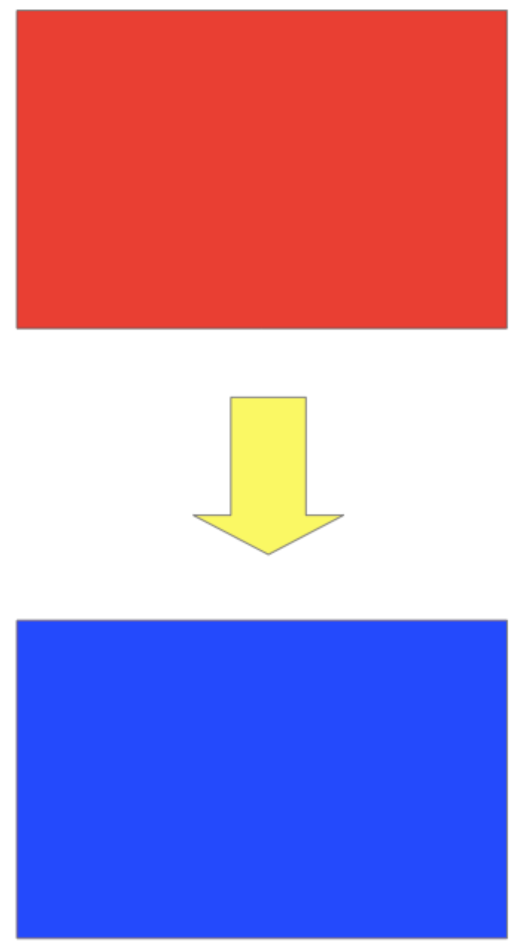
- `ArrayList`
  - reading elements sequentially from a file
- `StringBuilder`
  - appending strings efficiently

Complexity of recursive programs

Divide and Conquer – split the problem into smaller instances of the same problem, solve them and combine the result

Complexity of recursive programs

To solve this…

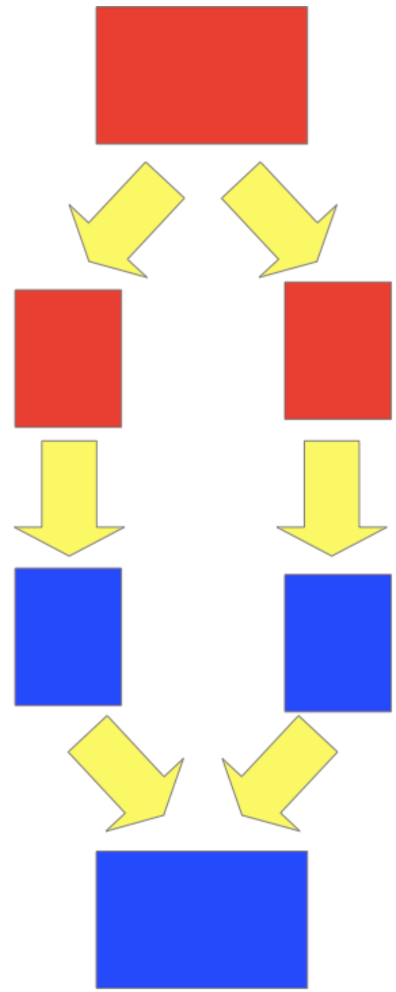


Complexity of recursive programs

1. *Split* the problem into subproblems

2. *Recursively* solve the subproblems

3. *Combine* the solutions



Complexity of recursive programs

Example:

Merge sort

Quick sort

Later

Complexity of recursive programs

Computing complexity T(N)

- Recursive expression
- No general formula
- It helps to see a pattern that expresses T(N) as a formula which only depends on N and the base case (T(1)/T(0))

Example

$T(1) = 1$

$T(N) = N + T(N-1)$

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

...

$= N + (N-1) + ... + 2 + 1 = N(N+1)/2$

So T(N) is O(N^2)

Example

$T(1) = 1$

$T(N) = N + T(N-1)$

$= N + (N/2 + T(N/4))$

…

$= N + N/2 + \ldots + N/2^{(\log N)} + T(N/2^{\log N})$

$= N(1 + \frac{1}{2} + \ldots + \frac{1}{2}^{(\log N-1)}) + T(N/2^{\log N})$

$= N(1 - 1/2^{(\log N)})/(1 - 1/2) + T(1)$

$= 2N + 1$

Asymptotic limit of geometric progression

So $T(N)$ is $O(N)$

Question

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

T(N) is :
a) O(log N)
b) O(N)
c) O(N log N)
d) O(N^2)

govote.at
Code 308412

To Do

Read from the book
- + 7.2 (insertion sort)
- + 7.6 (merge sort)
- + 3.4 (array list) – if you're curious
  - better: http://en.wikipedia.org/wiki/Dynamic_array

Videos:
- + Insertion sort: https://www.youtube.com/watch?v=ROalU379l3U
- + Merge sort: https://www.youtube.com/watch?v=XaqR3G_NVoo