Algorithms and Data Structures

Sorting:
Merge Sort and Quick Sort

Ulf Leser

Content of this Lecture

- Merge Sort
- Quick Sort

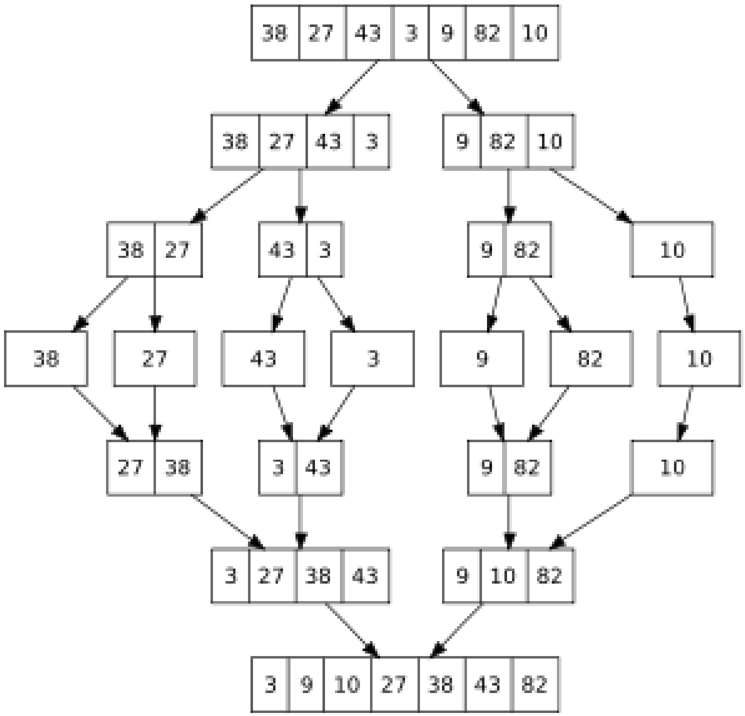
Central Idea for Improvement

- Methods we analyzed so-far did not optimally exploit transitivity of the „greater-or-equal“ relationship
- If $x \leq y$ and $y \leq z$, then $x \leq z$
- If we compared x and y and y and z, there is no need any more to compare x and z
- The clue to lower complexities in sorting is finding ways to exploit such information

Merge Sort

- Given the lower bound, we hope that we can do better
  – Not necessarily: The lower bound does not imply per-se that there is (and that we know) an algorithm which runs in this complexity
- Good news: There are various sort algorithms with O(n*log(n)) comparisons
- (Probably) Simplest one: Merge Sort
  – Divide-and-conquer algorithm
  – Break array in two partitions of equal size
  – Sort each partition recursively, if it has more than 1 elements
  – Merge sorted partitions
- Merge Sort is not in-place: Requires O(n) additional space

Illustration

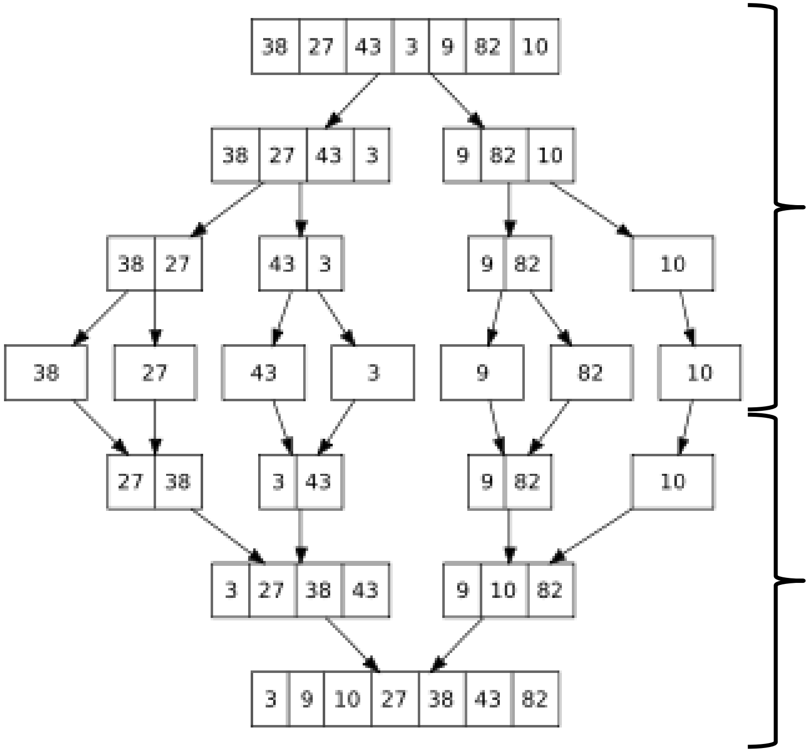


Source: WikiPedia

Divide

Conquer

Illustration



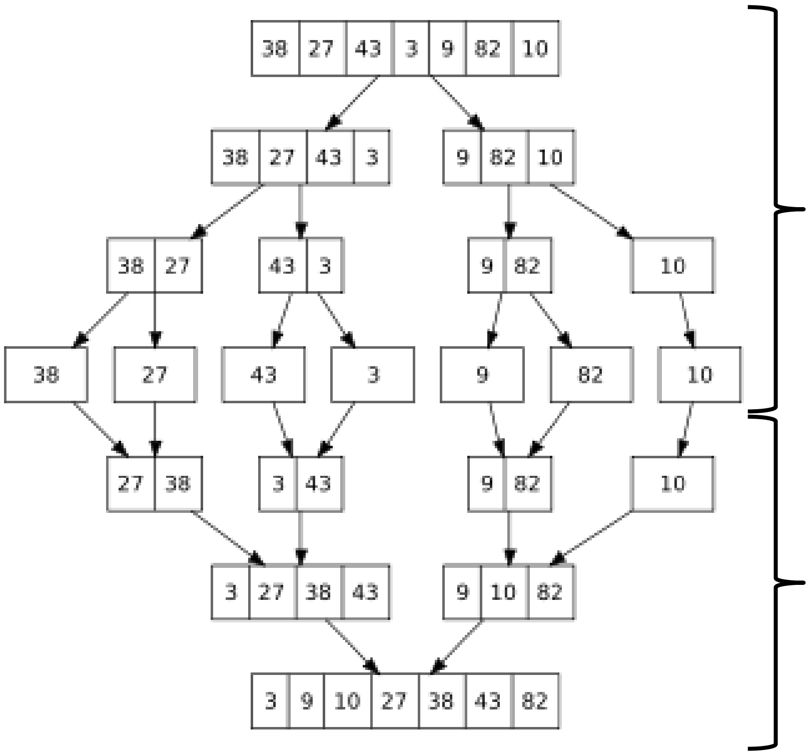
Source: WikiPedia

Divide - Partition

Conquer - Merge

- Here we exploit transitivity
- We save comparisons during merge because both sub-lists are sorted

Algorithm

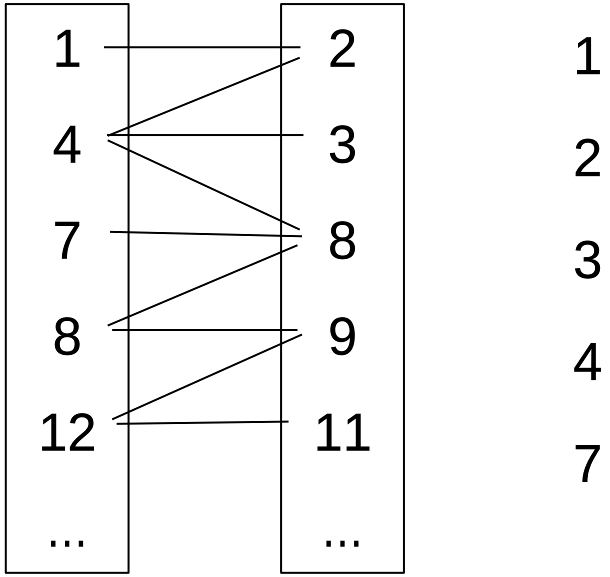


Source: WikiPedia

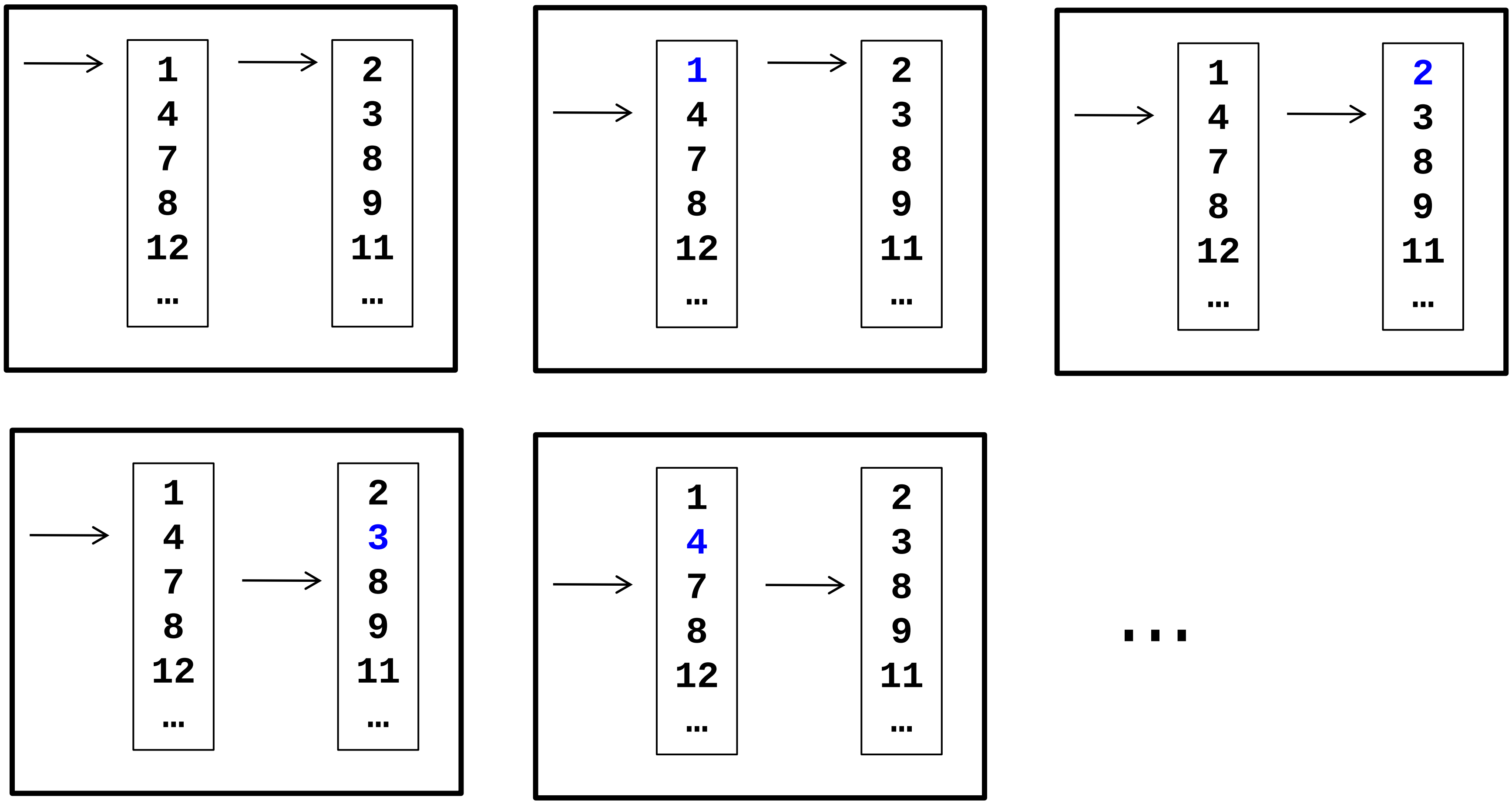
```
function void mergesort(S array;
                        l,r integer) {
  if (l<r) then
    m := (r-l) div 2;
    mergesort( S, l, m);
    mergesort( S, l+m+1, r);
      #merges two sorted lists:
    merge( S, l, l+m ,r);
  else
    # Nothing to do, 1-element list
  end if;
}
```

Merging Two Sorted Lists

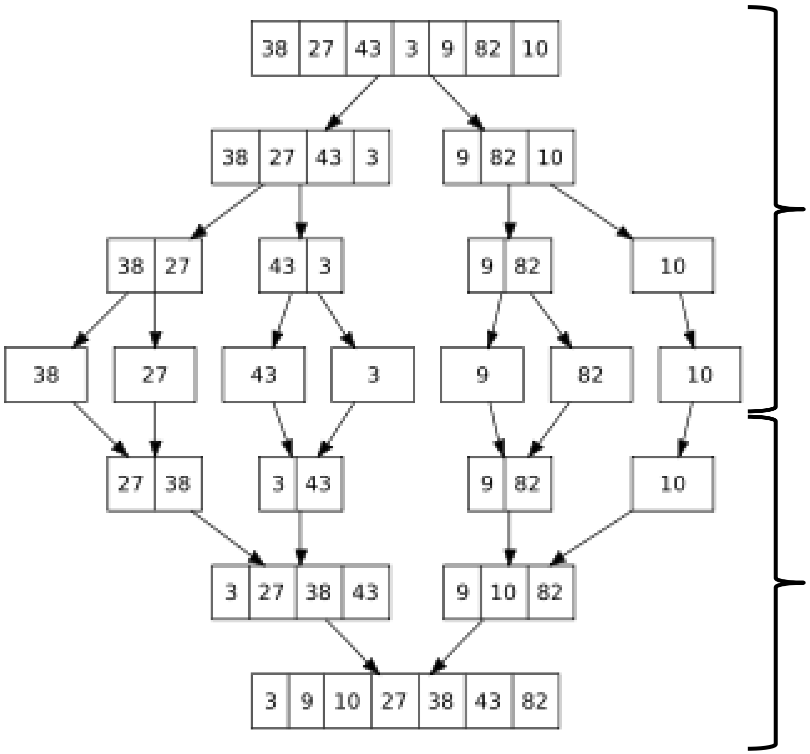
- We briefly looked at this problem before: Intersection of two sorted doc-lists in Information Retrieval
- Idea
  - Move one pointer through each list
  - Whatever element is smaller, copy to a new list and increment this pointer
    - “New list” requires additional space
  - Repeat until one list is exhausted
  - Copy rest of other list to new list



Example



Merge



Source: WikiPedia

```
function void merge(S array;
                   l,m,r integer) {
  B: array;
  i := l;         # Start 1st list
  j := m+1;       # Start 2nd list
  k := l;         # Target list
  while (i<=m) and (j<=r) do
    if S[i]<=S[j] then
      B[k] := S[i]; # From 1st list
      i := i+1;
    else
      B[k] := S[j]; # From 2nd list
      j := j+1;
    end if;
    k := k+1;       # Next target
  end while;
  if i>m then      # What remained?
    copy S[j..r] to B[k..k+r-j+1];
  else
    copy S[i..m] to B[k..k+m-i+1];
  end if;
    # Back to place
  copy B[1..k] to S[l..r]
}
```

Complexity Analysis

- Theorem
  Merge Sort requires $\Omega(n*\log(n))$ and $O(n*\log(n))$ comparisons
- Proof
  - Merging two sorted lists of size n requires O(n) comparisons
    - After every comp, 1 element is moved; there are only 2*n elements
  - Merge Sort calls Merge Sort twice with half of the array
    - Let T(n) be the number of comparisons
    - Thus: $T(n) = T(n/2) + T(n/2) + O(n) \sim 2*T(n/2) + n$
  - This is O(n*log(n))
    - See recursive solution of max subarray

Remarks

- Merge Sort is worst-case optimal: Even in the worst of all cases, it does not need more than (in the order of) the minimal number of comparisons
  - Given our lower bound for sorting
- But there are also disadvantages
  - O(n) additional space
  - Requires $\Omega(n*\log(n))$ moves
    - Sorted sub-arrays get copied to new array in any case
    - See Ottmann/Widmayer for proof
- Note: Basis for sort algorithms on external memory

Summary

	Comparisons worst case	Comparisons best case	Additional space	Moves worst/best
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(1)$	$O(n^2)$ / $O(n)$
Bubble Sort	$O(n^2)$	$O(n)$	$O(1)$	$O(n^2)$ / $O(1)$
Merge Sort	$O(n*\log(n))$	$O(n*\log(n))$	$O(n)$	$O(n*\log(n))$

Content of this Lecture

- Merge Sort
- Quick Sort
  - Algorithm
  - Average Case Analysis
  - Improving Space Complexity

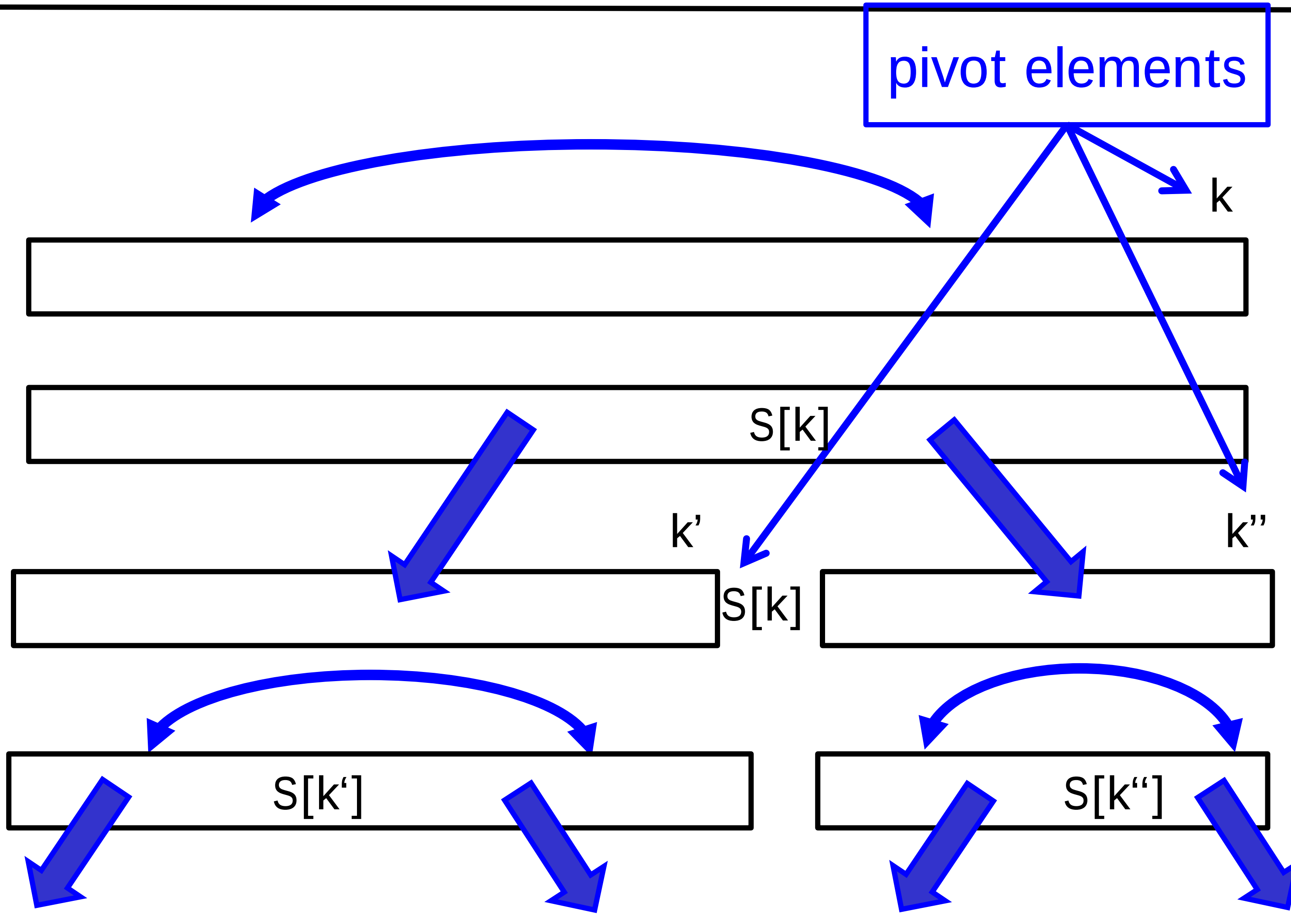
Comparison Merge Sort and Quick Sort

- What can we do better than Merge Sort?
  - The O(n) additional space is a problem
  - We need this space because the growing sorted runs have fixed sizes of up to 50% of |S| (2, 4, 8, ..., ceil(n/2))
  - We cannot easily merge two sorted lists in-place, because we have no clue how the numbers are distributed in the two lists
- Quick-sort uses a similar yet different way
  - We also recursively generate sort-of sorted runs
  - Whenever we create two such runs, we make sure that one contains only small and one contains only large values
    - Relative to a value that needs to be determined
  - This allows us to do a kind-of "merge" in-place

Main Idea

- Let k be an arbitrary index of S, $1 \leq k \leq |S|$
- Look at element S[k] (call it the pivot element)
- Modify S such that $\exists i: \forall j \leq i: S[j] \leq S[k]$ and $\forall l > i: S[k] \leq S[l]$
  - How? Wait a minute
  - S is broken in two subarrays S' and S''
    - S' with values smaller-or-equal than S[k]
    - S'' with values larger-or-equal than S[k]
  - Note that afterwards S[k] is at its final position in the array
  - S' and S'' are smaller than S
    - But we don't know how much smaller – depends on choice of k
- Treat S' and S'' using the same method recursively
  - How often? Not clear – depends on choice of k (again)

Illustration

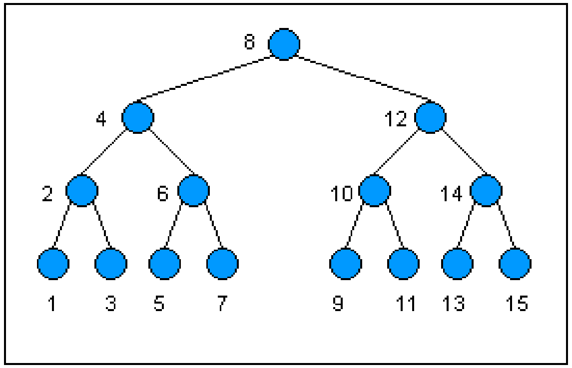


Quick Sort Framework

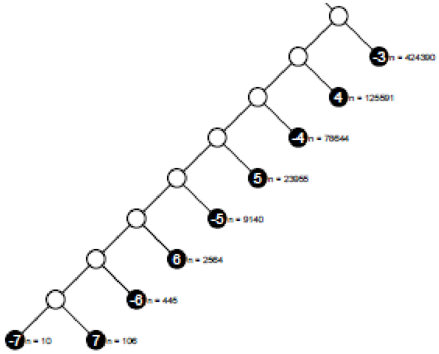
```
1. func void qsort(S array;
2.                  l,r integer) {
3.   if r≤l then
4.     return;
5.   end if;
6.   pos := divide( S, l, r);
7.   qsort( S, l, pos-1);
8.   qsort( S, pos+1, r);
9. }
```

- Start with qsort(S, 1, |S|)
- 6: “Sort” S around the pivot element (**`divide`**)
  - Problem 1: Choose k
  - Problem 2: Do this in-place
- 7: Sort all values smaller-or equal than pivot element
- 8: Sort all values larger-or-equal than pivot element
- Problem 3: How often do we need to do this?

Addressing P1 – approaching P3



- P1: We need to choose k (S[k])
- S[k] determines the sizes of S’ and S’’
- S[k] in the middle of the values of S
  - S’ and S’’ are of equal size (~|S|/2)
  - Creates a “nice” search tree
- S[k] at the border of the values of S
  - |S’|~0 and |S’’|~|S|-1 or vice versa
  - Creates a “bad” search tree
- Hint to P3: Somewhere in [log(n), n] times
  - Depending on choice of S[k]



Intermezzo: Mean and Median

- In statistics, one often tries to capture the essence of a (potentially large) set of values
- One essence: Mean
  - Average temperature per month, average income per year, average height of males at age of 18, average duration of study, ...
- Less sensible to outliers: Median
  - The middle value
  - Assume temps in June 25 24 24 23 25 25 24 4 -1 9 18 24
  - Which temperature do you expect for an average day in June?
    - Mean: 18.6
    - Median: 24 – more realistic
  - How long will you need for your Bachelor? 6,35 semesters?

P1: Choosing k

- In the best case, S[k] is the median of S
- Approximations
  - If S is an array of people's income in Germany, we call the "Statistische Bundesamt" to ask for the mean of all incomes in Germany, and could scan the array until we find a value that is 10% or less different, and use this value as pivot
    - If S is large and randomly drawn from a set of incomes, this scan will be very short
  - If S is an array of family names in Berlin, we take the Berlin telephone book, open it roughly in the middle, and could scan the array until we find a value that is 10% or less different
- There is no exact and simple way to find the median of a large list of values (without sorting them)

P1: Choosing k - Again

- Option 1: Scan S to find min/max; search S[k]~(max-min)/2
  - Why should the values in S be equally distributed in this range?
  - For instance: Incomes are not equally distributed in their range
- Option 2: Choose a set of values X from S at random and determine S[k]~median(X)
  - X follows the same distribution (same median) as S, but |X|<<|S|
  - Since this procedure would have to be performed for each qSort, only (too) small X do not influence runtime a lot
- More popular option 3: Choose k at random
  - For instance, simply use the last value in the array
  - Also relieves from searching an appropriate S[k]
  - We'll see that this already produces quite good result on average

Quick Sort Framework

```
1. func void qsort(S array;
2.                  l,r integer) {
3.   if r≤l then
4.     return;
5.   end if;
6.   pos := divide( S, l, r);
7.   qsort( S, l, pos-1);
8.   qsort( S, pos+1, r);
9. }
```

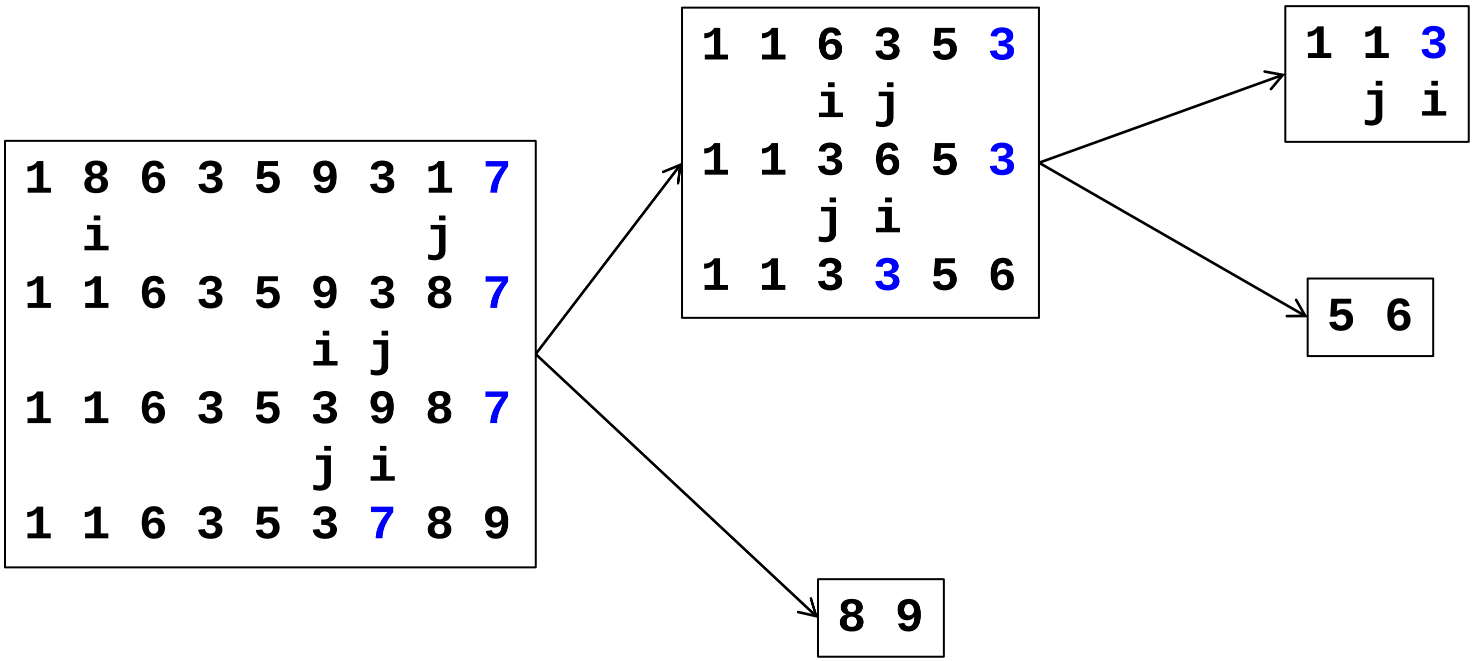
- Start with qsort(S, 1, |S|)
- 6: “Sort” S around the pivot element (**`divide`**)
  - Problem 1: Choose k
  - Problem 2: Do this in-place
- 7: Sort all values smaller-or equal than pivot element
- 8: Sort all values larger-or-equal than pivot element
- Problem 3: How often do we need to do this?

P2: Do this in-place

- We use k=r
- Simple idea
  - Search from l towards r until a value greater-or-equal S[r]
  - Start from r towards l until a values smaller-or-equal S[r]
  - Swap values
  - Start again, if i has not yet reached j
  - When we have stopped, all values left from i are smaller than S[r], and all values right from j are larger than S[r] – move S[r] right in the middle

```
func integer divide(S array;
                    l,r integer) {
  val := S[r];
  i := l-1;
  j := r;
  while true
    repeat
      i := i+1;
    until S[i]>=val;
    repeat
      j := j-1;
    until S[j]<=val or j<i;
    if i>j then
      break while;
    end if;
    swap( S[i], S[j]);
  end while;
  swap( S[i], S[r]);
  return i;
}
```

Example


1 8 6 3 5 9 3 1 7
i j
1 1 6 3 5 9 3 8 7
i j
1 1 6 3 5 3 9 8 7
j i
1 1 6 3 5 3 7 8 9
1 1 6 3 5 3
i j
1 1 3 6 5 3
j i
1 1 3 3 5 6
1 1 3
j i
5 6
8 9

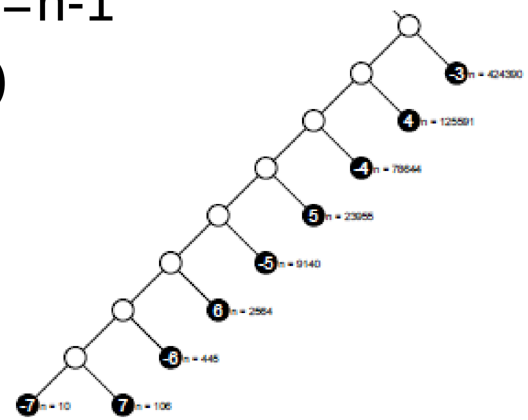
P2: Complexity

- # of comparisons: O(r-l)
  - Whenever we perform a comparison, either i or j are incremented / decremented
  - i starts from l, j starts from r, and the algorithm stops once they meet
  - This is worst, average and best case
- # of swaps: O(r-l) in worst case
  - Example: 8,7,8,6,1,3,2,3,5
  - Gives ~(r-l)/2 swaps

```
func integer divide(S array;
                    l,r integer) {
  val := S[r];
  i := l-1;
  j := r;
  while true
    repeat
      i := i+1;
    until S[i]>=val;
    repeat
      j := j-1;
    until S[j]<=val or j<i;
    if i>j then
      break while;
    end if;
    swap( S[i], S[j]);
  end while;
  swap( S[i], S[r]);
  return i;
}
```

Worst-Case Complexity of Quick Sort

- Worst case for number of comparisons:
  A reverse-sorted list
  – S[r] always is the smallest element
  – Requires r-l comparisons in every call of **`divide()`**
  – Every pair of qSort's has |S'|=0 and |S''|=n-1
  – This gives (n-1)+((n-1)-1)+...+1 = $O(n^2)$



Content of this Lecture

- Merge Sort
- Quick Sort
  - Algorithm
  - Average Case Analysis
  - Improving Space Complexity

Intermediate Summary

- Great disappointment
- We are in O(1) additional space, but as slow as our basic sorting algorithms
- But – only in worst case
- Let‘s look at the average case

Average Case

- Without loss of generality, we assume that S contains all values 1...|S| in arbitrary order
  - If S had duplicates, we would at best save swaps (see code)
  - Sorting n different values is the same problem as sorting the values 1...n – replace each value by its rank
- For k, we choose any value in S with equal probability 1/n
- This choice divides S such that |S'|=k-1 and |S''|=n-k
- Let T(n) be the average # of comparisons. Then:

$$T(n) = \frac{1}{n}\sum_{k=1}^{n}\left(T(k-1)+T(n-k)\right)+bn = \frac{2}{n}\sum_{k=1}^{n-1}T(k)+bn$$

  - Where bn is the time to divide the array and T(0)=0

Induction

- We need to show that, for some c independent of n:

$$T(n) \le c * n * \log(n)$$

- We proof by induction (for n≥2)
  - Clearly, T(1)=0 ≤ 1*log(1)
  - We assume that the assumption holds for all i<n
  - Then

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + bn$$

$$\le \frac{2c}{n}\sum_{k=1}^{n-1} k * \log(k) + bn$$

$$= \frac{2c}{n}\left[\sum_{k=1}^{n/2} k * \log(k) + \sum_{k=1}^{n/2-1}\left(\frac{n}{2}+k\right) * \log\left(\frac{n}{2}+k\right)\right] + bn$$

Continued

log(k)≤log(n)

$$T(n) \le \frac{2c}{n}\left[\sum_{k=1}^{n/2} k * \log(k) + \sum_{k=1}^{n/2-1}\left(\frac{n}{2}+k\right) * \log\left(\frac{n}{2}+k\right)\right] + bn$$

$$\le \frac{2c}{n}\left[\sum_{k=1}^{n/2} k * \log(n) + \sum_{k=1}^{n/2-1}\left(\frac{n}{2}+k\right) * \log(n)\right] + bn$$

$$= \frac{2c}{n}\left[\left(\frac{n^2}{2} - \frac{n}{2}\right) * \log(n) - \frac{n^2}{8} - \frac{n}{4}\right] + bn$$

$$= c * n * \log(n) - c * \log(n) - \frac{cn}{4} - \frac{c}{2} + bn$$

$$\le c * n * \log(n) - cn/4 - c/2 + bn$$

$$\le c * n * \log(n)$$

Set c≥4b

Conclusion

- Although there are cases where we need $O(n^2)$ comparisons, these are so rare in the set of all possible permutations that we do not need more than $O(n*\log(n))$ comparisons on average
- In other words: If we sum the runtimes of Quick Sort over many (all) different orders of n values (for different n), then this sum will grow with $n*\log(n)$, not with n^2
- One can show the same for the # of swaps
- Quick Sort is a fast general-purpose sorting algorithm

Content of this Lecture

- Merge Sort
- Quick Sort
  - Algorithm
  - Average Case Analysis
  - Improving Space Complexity

Looking at Space Again

- We were quite sloppy
- Quick Sort does need extra space – every recursive call puts some data on the stack
  - Array can be passed by reference or declared as a global variable
  - But we need to pass l and r
- Our current version has worst-case space complexity O(n)
  - Consider the worst-case of the time complexity
    - Reverse-sorted array
  - Creates 2*n recursive calls
  - This requires n times 2 integers on the stack

Improving Space Complexity

- In the recursive decent, always treat the smaller of the two sub-arrays first (S’ or S’’, whatever is smaller)
- This branch of the search tree can generate at most O(log(n)) calls, as the smaller array always is smaller than |S|/2 (or it would not be the smaller one)
- Use iteration (no stack) to sort the bigger array afterwards
- Space complexity: O(log(n))

Implementation

```
func integer qSort(S array;
                   l,r int) {
  if r≤l then
    return;
  end if;
  val := S[r];
  i := l-1;
  j := r;
  while true
    repeat
      i := i+1;
    until S[i]>val;
    repeat
      j := j-1;
    until S[j]<val or j<i;
    if i>j then
      break while;
    end if;
    swap( S[i], S[j]);
  end while;
  swap( S[i], S[r]);
  qsort(S, l, i-1);
  qSort(S, i+1, r);
}
```

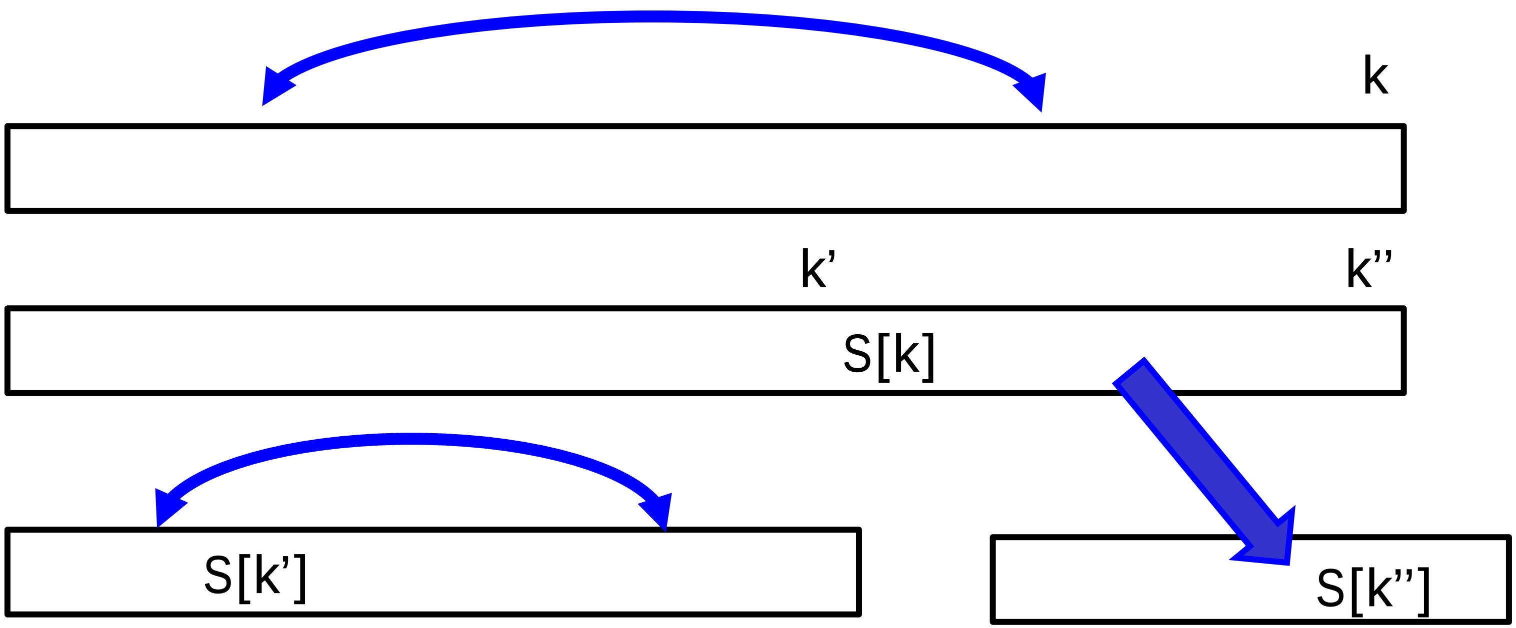
```
func integer qSort++(S array;
                     l,r int) {
  if r≤l then
    return;
  end if;
  while r > l do
    val := S[r];
    i := l-1;
    j := r;
    while true
      …        # as before
    end while;
    swap( S[i], S[r]);
    if (i-1-l) < (r-i-1) then
      qsort(S, l, i-1);
      l := i+1;
    else
      qSort(S, i+1, r);
      r := i-1;
    end if;
  end while;
}
```

Implementation

- 14-20: Choose the smaller and sort it recursively
  - Note: Only one call is made for each division
- We adjust l/r and continue to sort the larger sub-array
  - New loop (6-21) applies the same procedure performing the next sort
- We turned a linear tail recursion into an iteration without stack

```
1.  func integer qSort++(S array;
2.                           l,r int) {
3.    if r≤l then
4.      return;
5.    end if;
6.    while r > l do
7.      val := S[r];
8.      i := l-1;
9.      j := r;
10.     while true
11.       …       # as before
12.     end while;
13.     swap( S[i], S[r]);
14.     if (i-1-l) < (r-i-1) then
15.       qsort(S, l, i-1);
16.       l := i+1;
17.     else
18.       qSort(S, i+1, r);
19.       r := i-1;
20.     end if;
21.   end while;
22.}
```

Illustration



Improving Space Complexity Further

- Even O(1) space is possible
  - Do not store l/r, but search them at runtime within the array
  - Requires extra work in terms of runtime, but within the same complexity
  - See Ottmann/Widmayer for details
  - Is it worth it in practice?
    - Log(n) usually is not a lot of space

Summary

	Comps worst case	avg. case	best case	Additional space	Moves (wc / ac)
Selection Sort	$O(n^2)$		$O(n^2)$	$O(1)$	$O(n)$
Insertion Sort	$O(n^2)$		$O(n)$	$O(1)$	$O(n^2)$
Bubble Sort	$O(n^2)$		$O(n)$	$O(1)$	$O(n^2)$
Merge Sort	$O(n*log(n))$		$O(n*log(n))$	$O(n)$	$O(n*log(n))$
QuickSort	$O(n^2)$	$O(n*log(n)$	$O(n*log(n)$	$O(log(n))$	$O(n^2)$ / $O(n*log(n))$