Algorithms and Data Structures

Sorting beyond Value Comparisons

Patrick Schäfer, Ulf Leser

Content of this Lecture

- Radix Exchange Sort
  - Sorting bitstrings in linear time (almost)
- Bucket Sort (aka LSD Radix Sort)

Knowledge

- Until now, we did not use any knowledge on the nature of the values we sort
  - Strings, integers, reals, names, dates, revenues, person's age
  - Only operation we used: "value1 < value2"
  - Exception: Our (refused) suggestion (max-min)/2 for selecting the pivot element in Quicksort (how can we do this for strings?)
- Use knowledge on data type: Positive integers
- First example
  - Assume a list S of n different integers, $\forall i: 1 \leq S[i] \leq n$
  - How can we sort S in O(n) time and with only n extra space?

Sorting Permutations

```
1. S: array_permuted_numbs;
2. B: array_of_size_|S|
3. for i:= 1 to |S| do
4.   B[S[i]] := S[i];
5. end for;
```

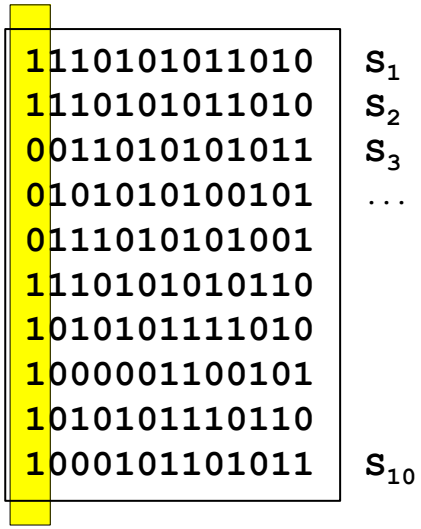
- Very simple
  - If we have all integers [1, n], then the final position of value i must be i
  - Obviously, we need only one scan and only one extra array (B)
- Knowledge we exploited
  - There are n different, unique values
  - The set is „dense" – no value between 1 and n is missing
  - It follows that the position of a value in the sorted list can be derived from the value

Removing Pre-Requisites

- Assume S is not dense (no duplicates)
  - n different integers each between 1 and m with m>n
  - For a given value S[i], we do not know any more its target position
    - How many values are smaller?
    - At most min(S[i], n)
    - At least max(n-(m-S[i]), 0)
  - This is almost the usual sorting problem, and we cannot do much
- Assume S has duplicates
  - S contains n values, where each value is between 1 and m and appears in S with m<n
  - Again: We cannot directly infer the position of S[i] from i alone

Second Example: Sorting Binary Strings

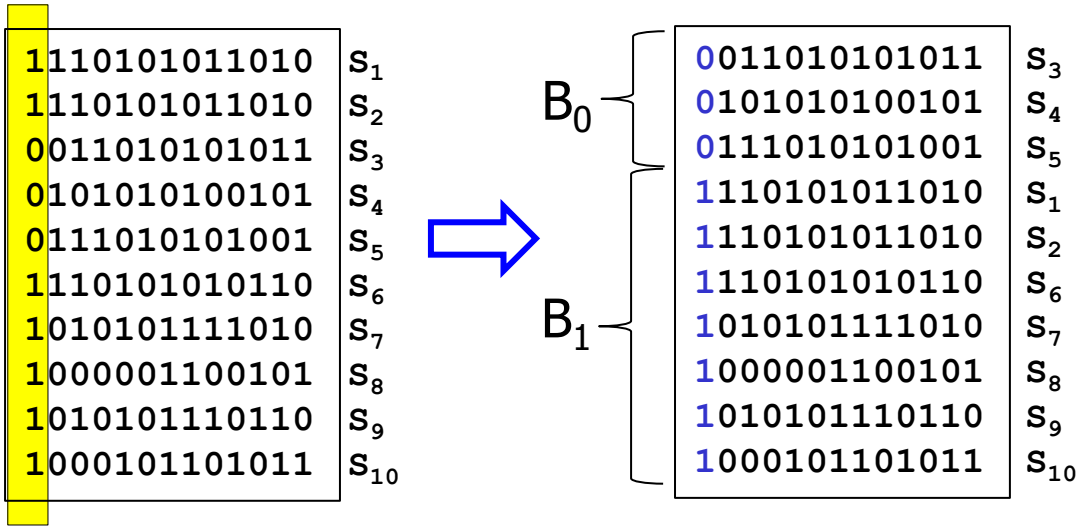
- Assume that all values are binary strings (bistrings) of equal length
  - E.g., integers in machine representation
- The most important position is the left-most bit, and it can have only two different values
  - Alphabet size is 2 in bitstrings



Second Example: Sorting Binary Strings

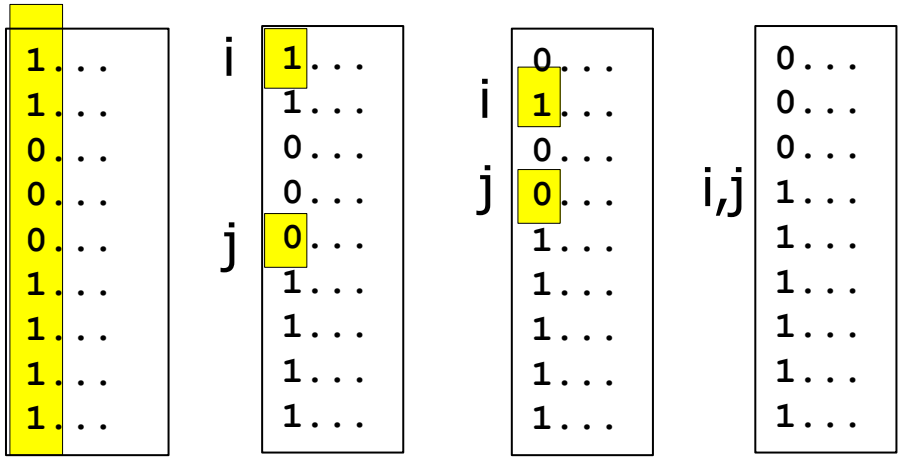
- We can sort all values by first position with a single scan
  - All values with leading 0 => list B0
  - All values with leading 1 => list B1

```
1.  S: array_bitstrings;
2.  B0: array_of_size_|S|
3.  B1: array_of_size_|S|
4.  j0 := 1;
5.  j1 := 1;
6.  for i:= 1 to |S| do
7.    if S[i][1]=0 then
8.      B0[j0] := S[i];
9.      j0 := j0 + 1;
10.   else
11.     B1[j1] := S[i];
12.     j1 := j1 + 1;
13.   end if;
14. end for;
15. return B0[1..j0]+B1[1..j1];
```



Improvement

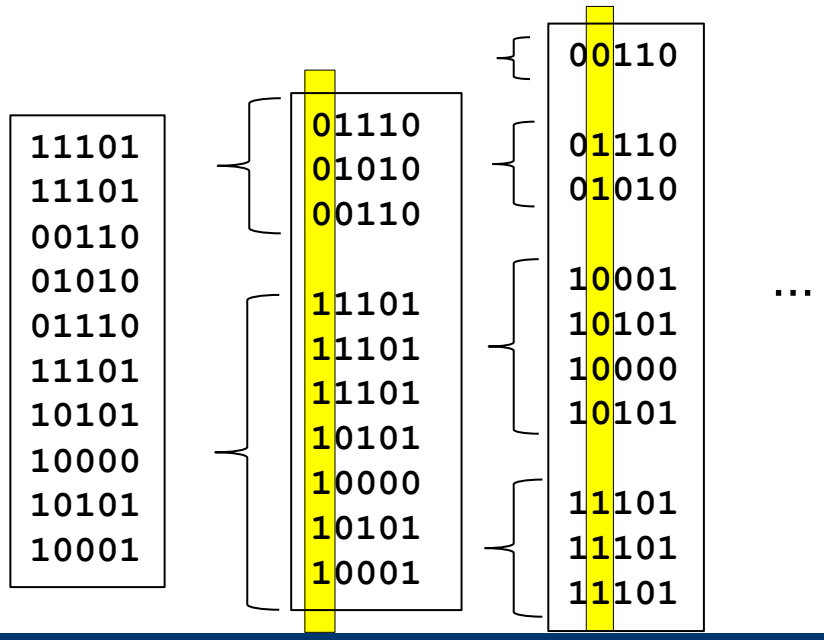
- How can we do this in O(1) additional space?
- Recall QuickSort
  - Call **`divide*(S,1,1,|S|)`**
  - k, l, r, and return value will be used in a minute
  - Note that we return j, the position of the first 1




```
1.  func int divide*(S array;
2.                   k,l,r: int) {
3.    i := l;
4.    j := r;
5.    repeat
6.      while S[i][k]=0 and i<j do
7.        i := i+1;
8.      end while;
9.      while S[j][k]=1 and i<j do
10.       j := j-1;
11.     end while;
12.     swap(S[i], S[j]);
13.   until i=j;
14.   if S[r][k]=0 then //only zeros
15.     j:=j+1;
16.   end if
17.   return j;    // first "1"
18. }
```

Sorting Complete Binary Strings

```
1.  func radixESort(S array;
2.                  k,l,r: integer) {
3.    if l≥r or k>m then
4.      return;
5.    end if;
6.    d := divide*(S, k, l, r);
7.    radixESort(S, k+1, l, d-1);
8.    radixESort(S, k+1, d, r);
9.  }
```



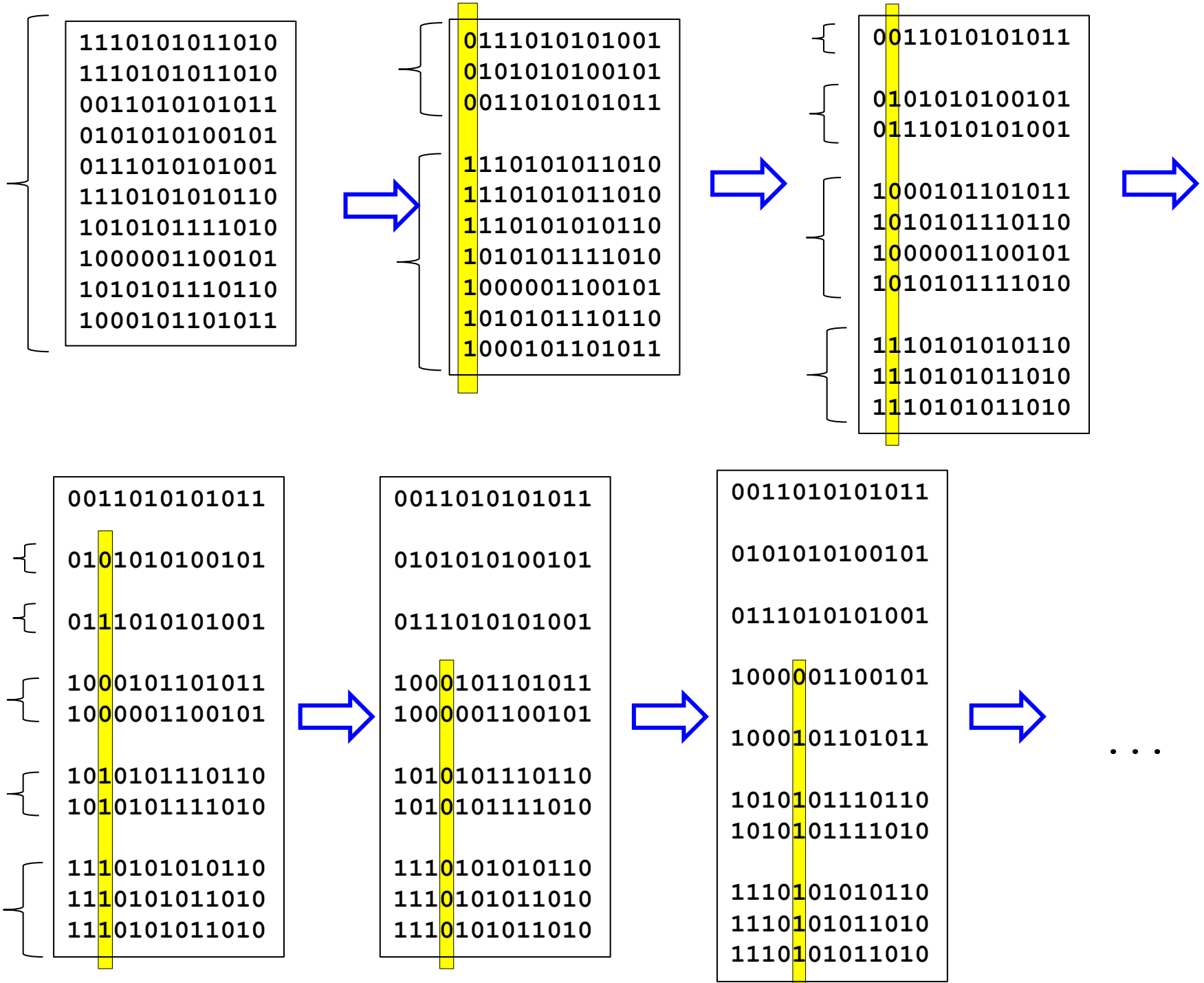
- We can repeat the same procedure on the second, third, … position
- When sorting the k'th position, we need to take care to not sort the entire S again, but only the subarrays with same values in the (k-1) first positions
  - Let m by the length (in bits) of the values in S
  - Call with `radixESort(S,1,1,|S|)`

Complexity

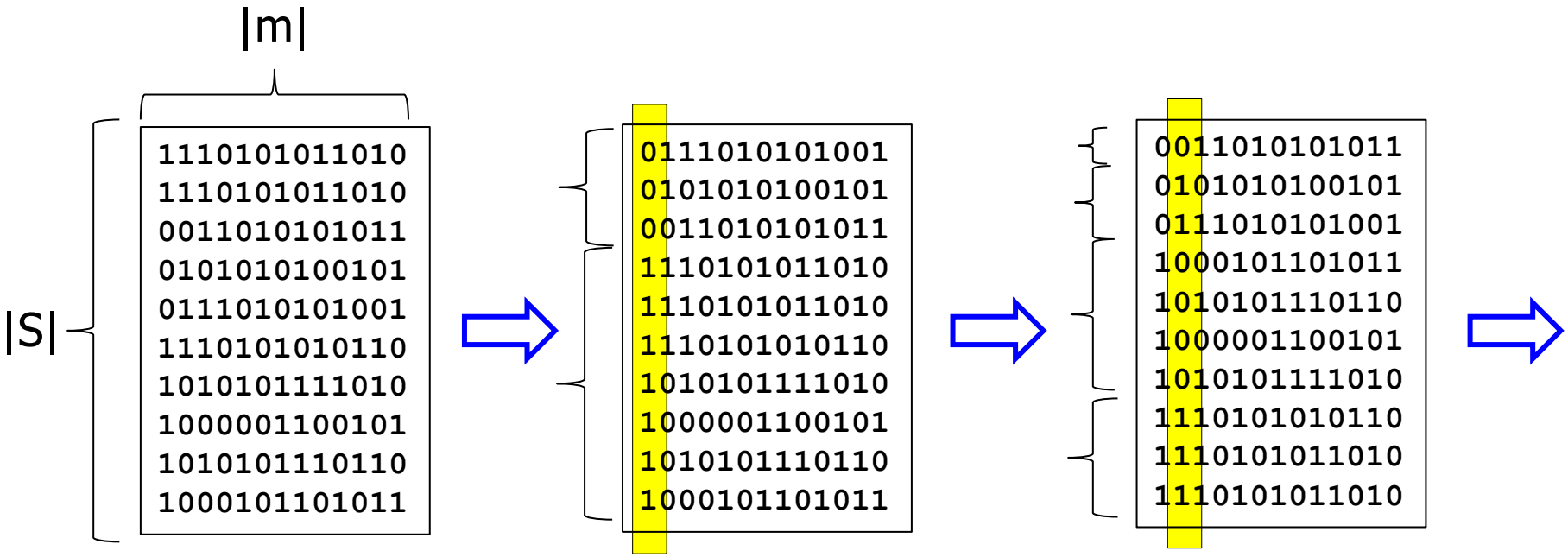
```
1.  func radixESort(S array;
2.                  k,l,r: integer) {
3.    if l≥r or k>m then
4.      return;
5.    end if;
6.    d := divide*(S, k, l, r);
7.    radixESort(S, k+1, l, d-1);
8.    radixESort(S, k+1, d, r);
9.  }
```

```
1.  func int divide*(S array;
2.                   k,l,r: int) {
3.    …
4.    repeat
5.      while S[i][k]=0 and i<j do
6.        i := i+1;
7.      end while;
8.      while S[j][k]=1 and i<j do
9.        j := j-1;
10.     end while;
11.     swap(S[i], S[j]);
12.   until i=j;
13.   …
14.   return j;   // first "1" }
```

- We count the overall number of comparisons for...
  - In divide*, we look at every element S[l…r] exactly once: (r-l)
  - Then we divide S[l…r] in two disjoint halves
    - 1st makes (d-l) comps
    - 2nd makes (r-d) comps
  - The first call to radixESort has O(n) comps, with |S|=n.
- Are we in O(n)?


1110101011010
1110101011010
0011010101011
0101010100101
0111010101001
1110101010110
1010101111010
1000001100101
1010101110110
1000101101011
0111010101001
0101010100101
0011010101011
1110101011010
1110101011010
1110101010110
1010101111010
1000001100101
1010101110110
1000101101011
0011010101011
0101010100101
0111010101001
1000101101011
1010101110110
1000001100101
1010101111010
1110101010110
1110101011010
1110101011010
0011010101011
0101010100101
0111010101001
1000101101011
1000001100101
1010101110110
1010101111010
1110101010110
1110101011010
1110101011010
0011010101011
0101010100101
0111010101001
1000001100101
1000101101011
1010101110110
1010101111010
1110101010110
1110101011010
1110101011010
. . .

Illustration



- For every k, we look at every S[i][k] once to see whether it is 0 or 1 – together, we have at most m*|S| comparisons
  - Of course, we can stop at every interval with (r-l)=1
  - m*|S| is the worst case

Complexity (Correct)

```
1.  func radixESort(S array;
2.                  k,l,r: integer) {
3.    if l≥r or k>m then
4.      return;
5.    end if;
6.    d := divide*(S, k, l, r);
7.    radixESort(S, k+1, l, d-1);
8.    radixESort(S, k+1, d, r);
9.  }
```

```
1.  func int divide*(S array;
2.                   k,l,r: int) {
3.    …
4.    repeat
5.      while S[i][k]=0 and i<j do
6.        i := i+1;
7.      end while;
8.      while S[j][k]=1 and i<j do
9.        j := j-1;
10.     end while;
11.     swap(S[i], S[j]);
12.   until i=j;
13.   …
14.   return j;   // first "1" }
```

- We count …
  - Every call to radixESort first performs (r-l) comps and then divides S[l…r] in two disjoint halves
    - 1st makes (d-l) comps
    - 2nd makes (r-d) comps
- First call to radixESort has O(n) comps, with |S|=n
- Recursion depth is fixed to m
- Thus: O(m*|S|) comps

RadixESort - Bonuses

- It may not have to examine all the bits/positions

```
00110101010111   = 2 / 13
01010101001011   = 3 / 13
01110101010011   = 3 / 13
10000011001011   = 5 / 13
10001011010111   = 5 / 13
10101011101110   = 11 / 13
10101011110110   = 11 / 13
11101010101110   = 10 / 13
11101010110110   = 13 / 13
11101010110110   = 13 / 13
```

~58% of bits examined

- It works for variable length bitstrings (equal bits have to be at equal positions or padded with 0)

```
00110101
0101010100101
01110
10000011
10001011
10101011101
10101011110
11101010101
```

RadixESort or QuickSort?

- Assume we have data that can be represented as bitstrings such that more important bits are left (or right – but consistent)
  - Integers, strings, bitstrings, …
  - Equal length is not necessary, but „the same" bits must be at the same position in the bitstring (otherwise, one may pad with 0)
- Decisive: m <? >? log(n)
  - If S is large / maximal bitstring length is small: RadixESort
  - If S is small / maximal bitstring length is large: QuickSort

Content of this Lecture

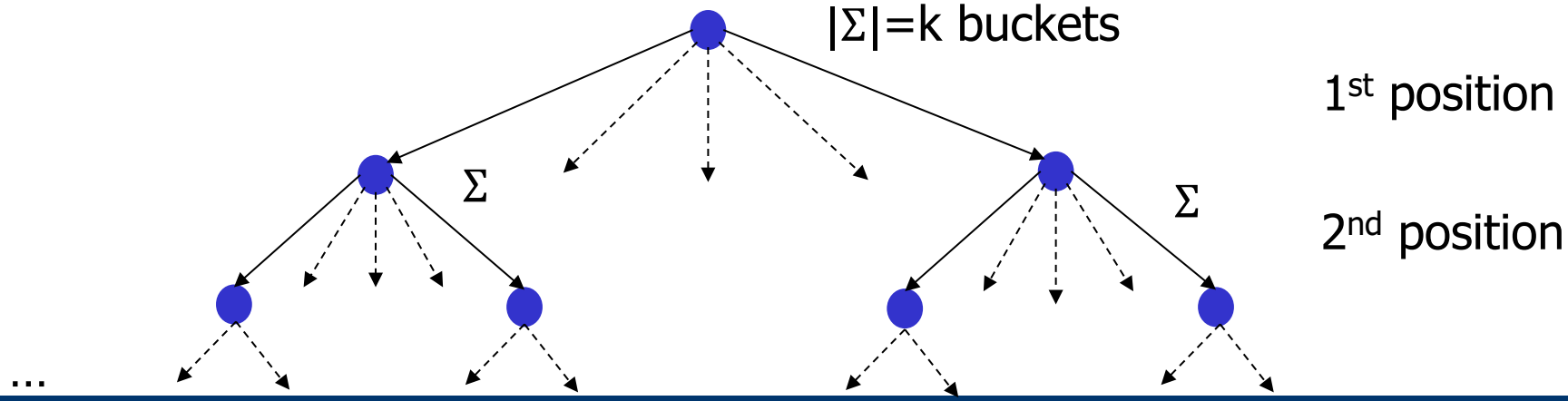
- Radix Exchange Sort
- Bucket Sort
  - Generalizing the Idea of Radix Exchange Sort to arbitrary alphabets

Bucket Sort

- Representing "normal" Strings as bitstrings is a bad idea
  - One byte per character -> 8*length bits (large m for RadixESort)
  - But: There are only ~26 different values (no case)
- One could find shorter encodings – we go a different way

Bucket Sort generalizes RadixESort

- Assume |S|=n, m being the length of the largest value, alphabet Σ with |Σ|=k and a lexicographical order (e.g., "A" < "AA")
- We first sort S on first position into k buckets (with a single scan)
- Then sort every bucket again for second position, etc.
- After at most m iterations, we are done
- Time complexity (ignoring space issues): O(m*(|S|+…))
- But space is an issue



Space in Bucket Sort

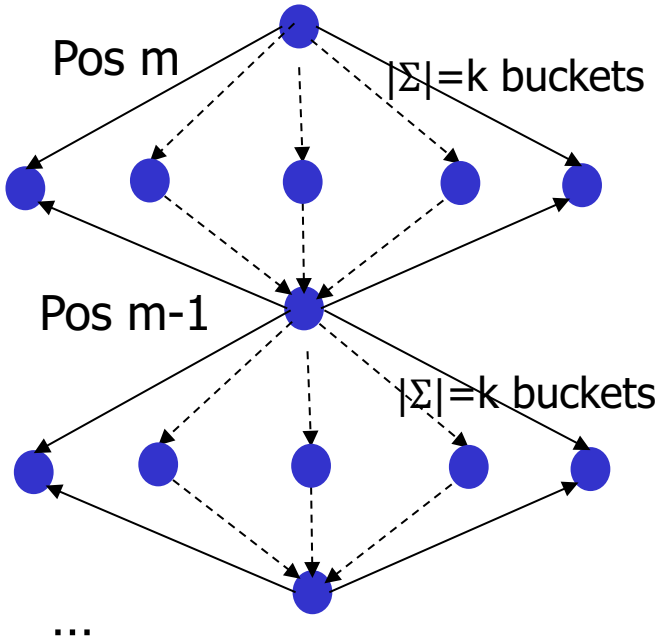
- A naïve array-based implementation allocates $k * |S|$ values for every phase of sorting into each bucket B
  - We do not know how many values start with a given character
  - Can be anything between 0 and |S|
- This would need to allocate a total of $O(k^m * |S|)$ space for m iterations – too much!

- We reduce this to $O(k + |S|)$
  - Requires a stable sorting algorithm for single characters
  - 1-phase Bucket Sort is stable (if implemented that way)

Bucket Sort - Idea

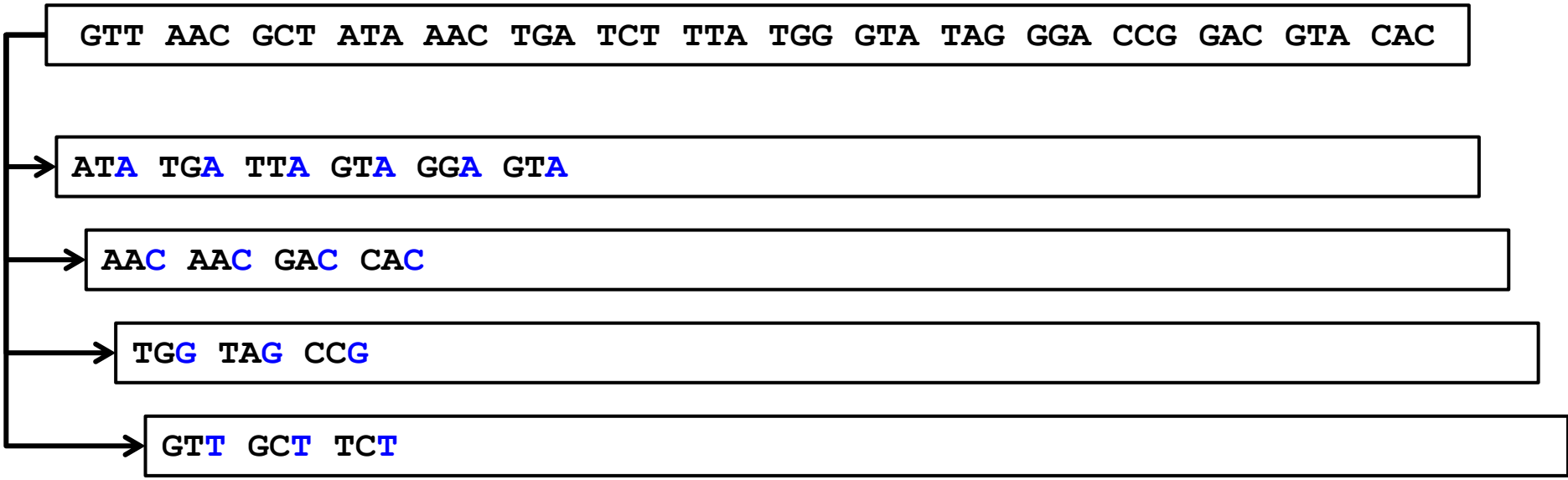
- If we sort from back-to-front and keep the order of once sorted suffixes, we can (re-)use the additional space
  - Order was not preserved in RadixESort, but there we could sort in-place – other problems

```
1.  func bucketSort(S array, m,k: integer){
2.    B:= Array of Queues with |B|=k
3.    for i := m down to 1 do
4.       use stable sorting algorithm
         to sort S on position I
5.       merge all buckets
6.    end for
7.  }
```



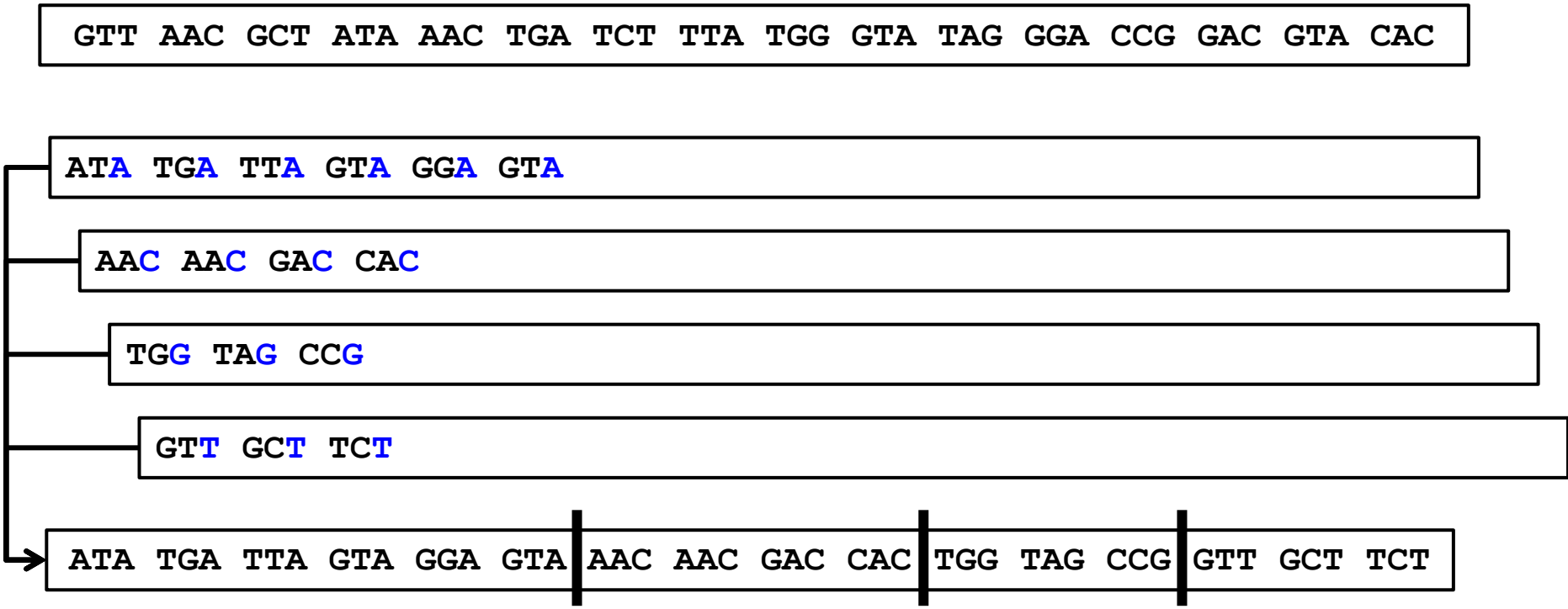
Bucket Sort

- If we sort from back-to-front and keep the order of once sorted suffixes, we can (re-)use the additional space



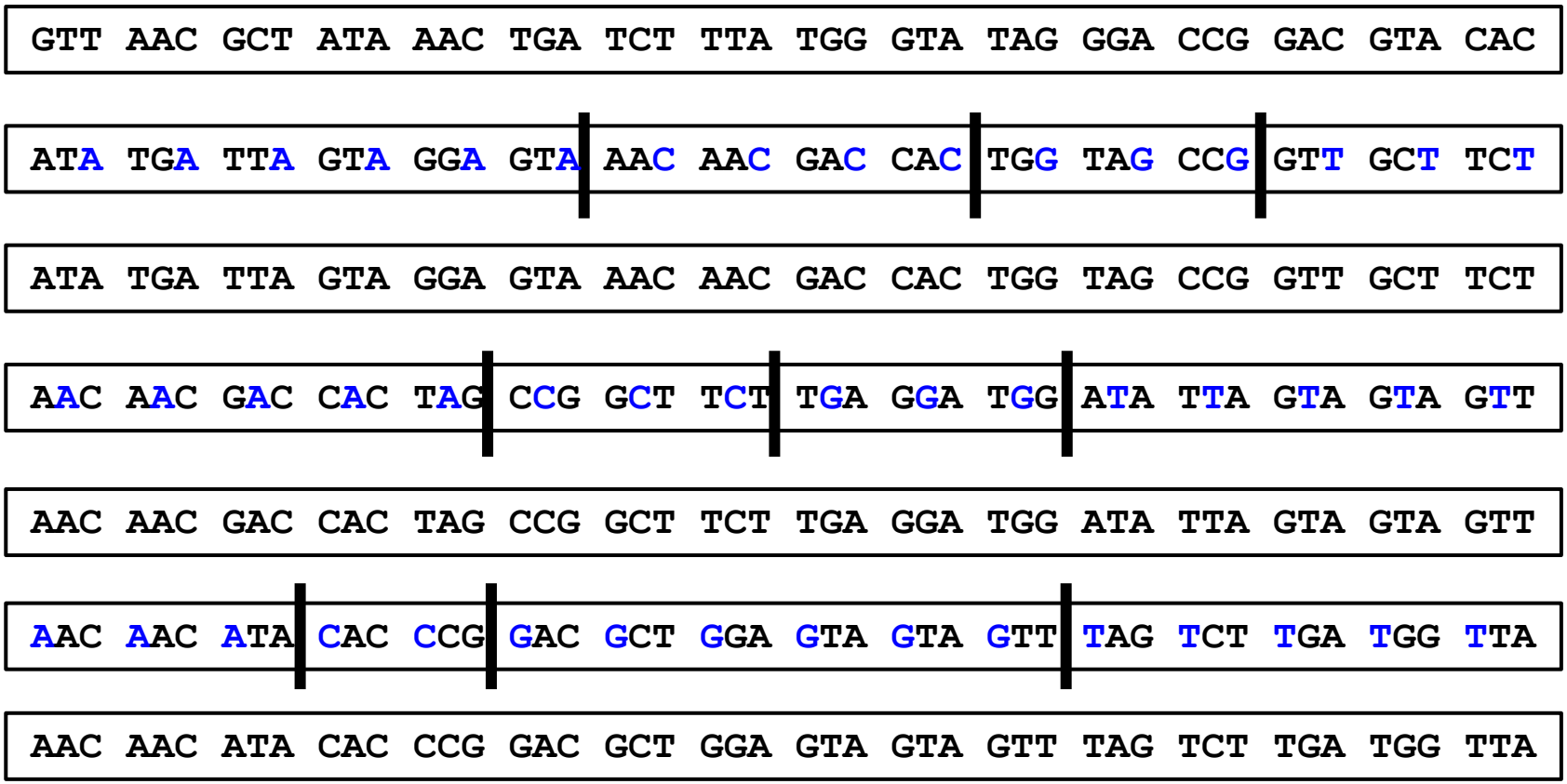
Bucket Sort

- If we sort from back-to-front and keep the order of once sorted suffixes, we can (re-)use the additional space



Bucket Sort

- If we sort from back-to-front and keep the order of once sorted suffixes, we can (re-)use the additional space



Bucket Sort – Pseudocode (aka LSD Radix Sort)

- Sort S from back-to-front
  - (Re-)use k queues, one for each bucket.
  - *findBucket* translates the i-th char of S[j] into a bucket.
  - Stable: the order of once sorted suffixes is kept (append to end of queue).
  - Finally, merge buckets and repeat with next position.

- Different mappings can be used, such as ‚A-Z' mapped to 1-26.
  - For linear time sorting, the number of buckets must be equal to k.

```
1.  func bucketSort(S array,
                    m, k: integer){
2.    B:= Array of Queues with |B|=k
3.    for i := m down to 1 do
4.      for j := 1 to |S| do
5.        k := findBucket(S[j][i]);
6.        B[k].enqueue(S[j]);
7.      end for
8.      j := 1;
9.      for k := 1 to |B| do
10.       while not B[k].isEmpty() do
11.         S[j] := B[k].dequeue();
12.         j := j + 1;
13.       end while end for
14.   end for
15.   return S;
16. }
```

Magic? Proof

- By induction
- Assume that before phase t we have sorted all values by the (t-1)-suffix (right-most, least important for order)
  - True for t=2 – we sorted by the last character ((t-1)-suffixes)
- In phase t, we sort by the t‘th lowest value (from the right)
- This will group all values from S with the same value in S[i][m-t+1] together and keep them sorted wrt. (t-1)-suffixes
  - Assuming a stable sorting algorithm
- Since we sort by S[i][m-t+1], the array after phase t will be sorted by the t-suffix
- qed.

Saving More Space

- The example has shown that we actually never need more than $O(|S| + k)$ additional space (all buckets together)
  - Use a linked-list/queue for each bucket
  - Keep pointer to start (for copying) and end (for extending) of each list – this requires 2*k space
  - All lists together only store |S| elements (of length m)

A Word on Names

- Names of these algorithms are not consistent
  - Radix Sort generally depicts the class of sorting algorithms which look at single keys and partition keys in smaller parts
  - RadixESort is also called binary quicksort (Sedgewick)
  - Bucket Sort is also called „Sortieren durch Fachverteilen“ (OW), RadixSort (German WikiPedia and Cormen et al.), or LSD Radix Sort (Sedgewick), or distribution sort
  - Cormen et al. use Bucket Sort for a variation of our Bucket Sort (linear only if keys are equally distributed)
  - …

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))
BucketSort (m= …)	O(m*(n+k))			O(n+k)	

Exemplary Questions

- What is the best case complexity of BucketSort?
- What is the space complexity of RadixESort?
- What is a stable sorting algorithm?
- Which of the following sorting algorithms are stable: BubbleSort, InsertionSort, MergeSort?
- BucketSort needs a data structure for building and using buckets. Give an implementation using (a) a heap, (b) a queue.