Algorithms and Data Structures

Sorting beyond Value Comparisons

Ulf Leser

Content of this Lecture

- Radix Exchange Sort
  - Sorting bitstrings in (almost) linear time
- Bucket Sort
  - Sorting Strings faster (?)

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 comparison we used: “value1 < value2”
- Now we will use such knowledge
- First example
  - Assume a list S of n different positive integers, $\forall i$: $1 \leq S[i] \leq n$
  - Can we sort S in O(n) time and with only n extra space?

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 comparison we used: "value1 < value2"
- Now we will use such knowledge
- First example
  - Assume a list S of n different positive integers, $\forall i: 1 \leq S[i] \leq n$
  - Can we sort S in $O(n)$ time and with only n extra space?

Our knowledge

Sorting Permutations

```
S: array_permuted_numbs;
B: array_of_size_|S|
for i:= 1 to |S| do
  B[S[i]] := S[i];
end for;
```

- Very simple
  - If all are integers in [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"
    - A dense set of integers of size n contains all values between 1 and n
  - In this special case, the position of a value in the sorted list can be derived from the value

Removing Pre-Requisites

- Assume S is not dense
  - n 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
    - We can sort such an S in O(m) with O(m) space – how?
- Assume S has duplicates
  - S contains n values, where each value is between 1 and n
  - Now we cannot infer the position of S[i] from i alone

Second Example: Sorting Binary Strings

- Assume that all keys are binary strings (bistrings) of equal length
  - E.g., unsigned 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

```
1110101011010   s1
1110101011010   s2
0011010101011   s3
0101010100101   ...
0111010101001
1110101010110
1010101111010
1000001100101
1010101110110
1000101101011   s10
```

Our knowledge
- We can break up values into "characters"
- Size of alphabet is limited (here: 2)

Second Example: Sorting Binary Strings

- We can sort all keys by first position with a single scan
  - All values with leading 0 => list B_0
  - All values with leading 1 => list B_1
  - Requires 2*n additional space
  - But …

1110101011010	S_1
1110101011010	S_2
0011010101011	S_3
0101010100101	S_4
0111010101001	S_5
1110101010110	S_6
1010101111010	S_7
1000001100101	S_8
1010101110110	S_9
1000101101011	S_{10}

⇒

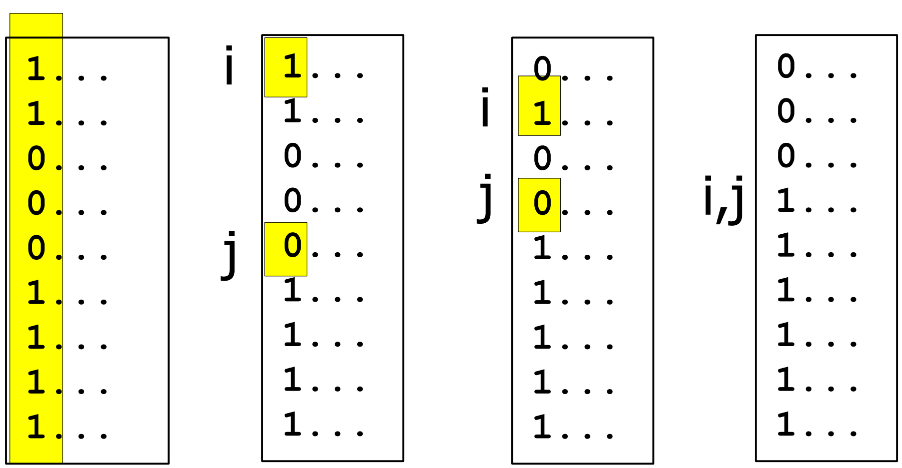
B_0	0011010101011	S_3
	0101010100101	S_4
	0111010101001	S_5
B_1	1110101011010	S_1
	1110101011010	S_2
	1110101010110	S_6
	1010101111010	S_7
	1000001100101	S_8
	1010101110110	S_9
	1000101101011	S_{10}

```
S: array_bitstrings;
B0: array_of_size_|S|
B1: array_of_size_|S|
j0 := 1;
j1 := 1;
for i:= 1 to |S| do
  if S[i][1]=0 then
    B0[j0] := S[i];
    j0 := j0 + 1;
  else
    B1[j1] := S[i];
    j1 := j1 + 1;
  end if;
end for;
return B0[1..j0]+B1[1..j1];
```

Improvement

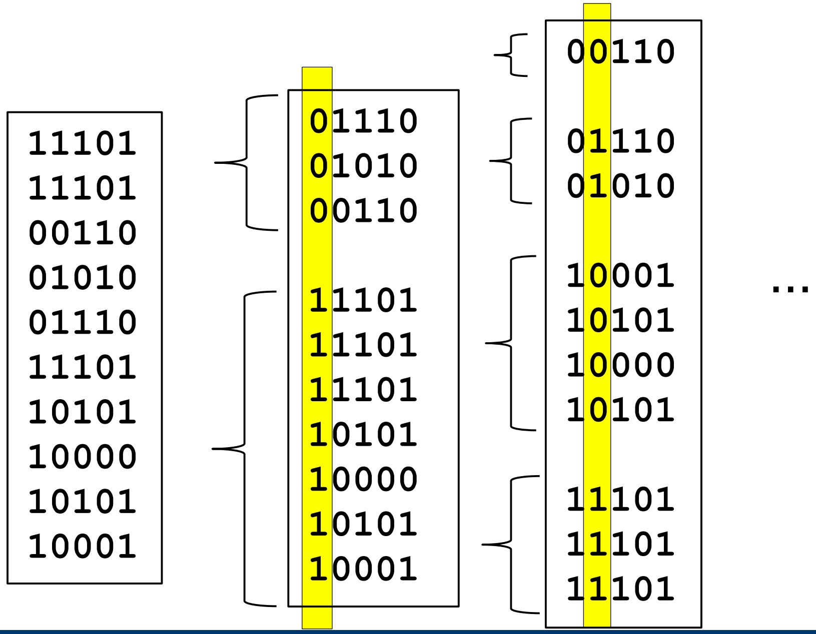
- Recall QuickSort
  - Call **divide*(S,1,1,|S|)**
    - k, f, r, and return value will be used in a minute
  - Choosing the pivot element here is trivial – only two values
- O(1) additional space

```
func int divide*(S array;
                     k,f,r: int) {
  i := f;
  j := r;
  repeat
    while S[i][k]=0 and i<j do
      i := i+1;
    end while;
    while S[j][k]=1 and i<j do
      j := j-1;
    end while;
    swap(S[i], S[j]);
  until i=j;
  if S[r][k]=0 then //only zeros
    j:=j+1;
  end if
  return j;   // first "1"
}
```

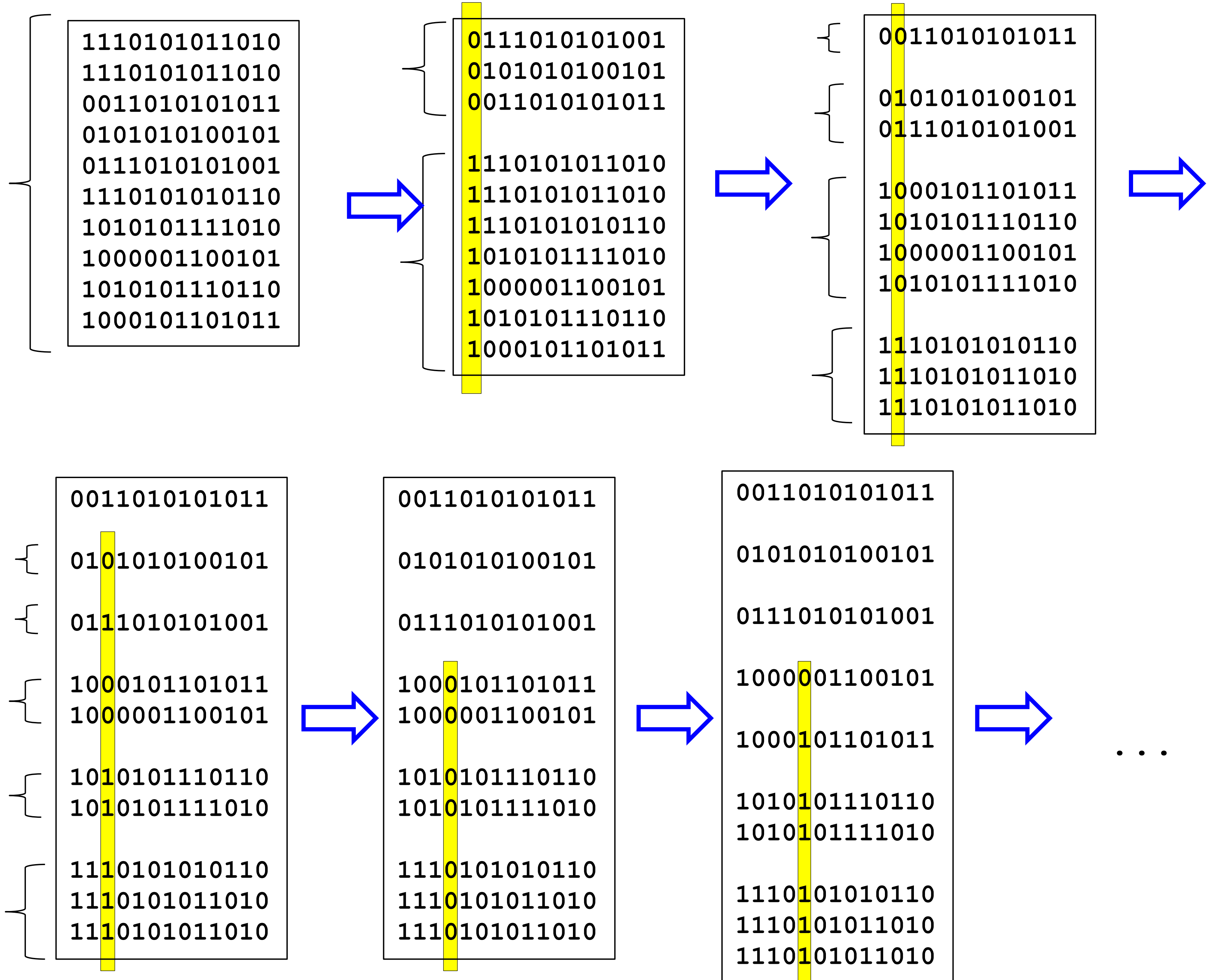


Completely Sorting Binary Strings

```
func radixESort(S array;
                k,f,r: integer) {
  if f≥r or k>m then
    return;
  end if;
  d := divide*(S, k, f, r);
  radixESort(S, k+1, f, d-1);
  radixESort(S, k+1, d, r);
}
```



- We can repeat the same procedure on the second, third, … position
- When sorting the k'th position, we only sort within 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,n)`**


111010101l010

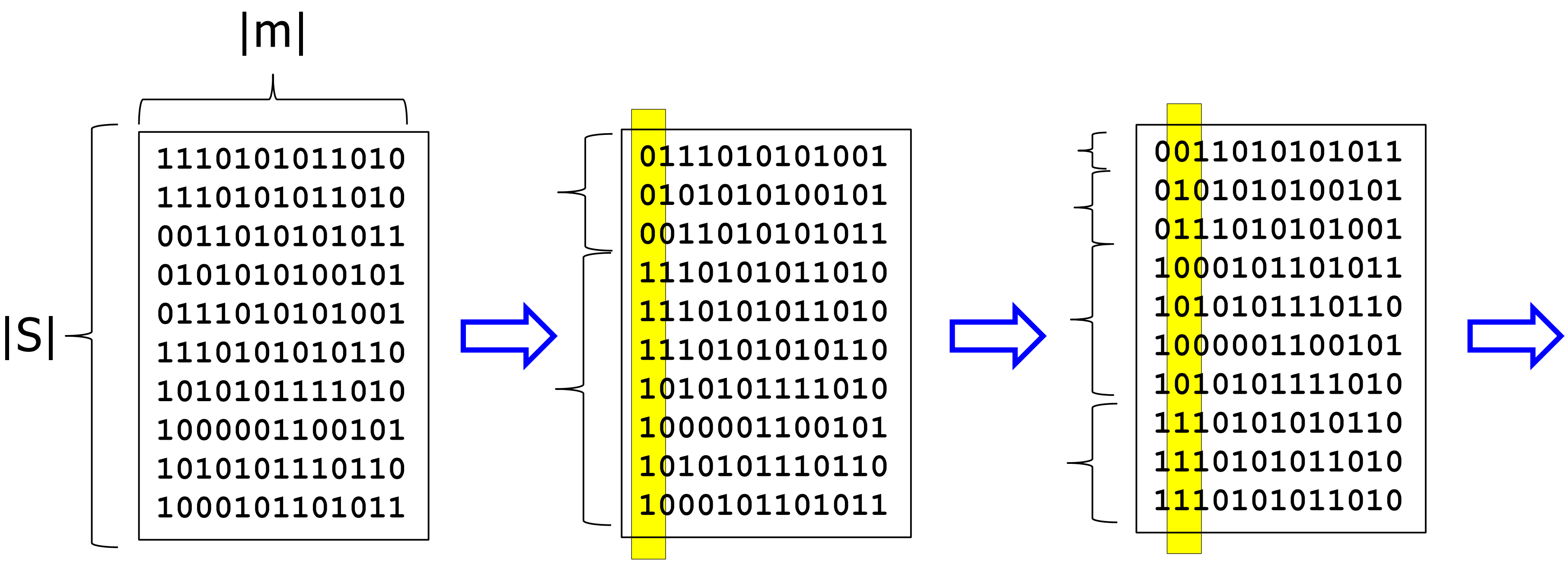
Complexity

```
func radixESort(S array;
                     k,f,r: integer) {
  if f≥r or k>m then
    return;
  end if;
  d := divide*(S, k, f, r);
  radixESort(S, k+1, f, d-1);
  radixESort(S, k+1, d, r);
}
```

```
func int divide*(S array;
                    k,f,r: int) {
  …
  repeat
    while S[i][k]=0 and i<j do
      i := i+1;
    end while;
    while S[j][k]=1 and i<j do
      j := j-1;
    end while;
    swap(S[i], S[j]);
  until i=j;
  …
  return j;   // first "1" }
```

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

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-f)=1
  - m*|S| is the worst case

Complexity (Correct)

```
func radixESort(S array;
                k,f,r: integer) {
  if f≥r or k>m then
    return;
  end if;
  d := divide*(S, k, f, r);
  radixESort(S, k+1, f, d-1);
  radixESort(S, k+1, d, r);
}
```

```
func int divide*(S array;
                 k,f,r: int) {
  …
  repeat
    while S[i][k]=0 and i<j do
      i := i+1;
    end while;
    while S[j][k]=1 and i<j do
      j := j-1;
    end while;
    swap(S[i], S[j]);
  until i=j;
  …
  return j;   // first "1" }
```

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

RadixESort or QuickSort?

- Assume we have data that can be represented as bitstrings such that 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: Is m smaller or larger than log(n)
  - If S is large / maximal bitstring length is small: RadixESort
  - If S is small / maximal bitstring length is large: QuickSort
- Note: QuickSort actually also requires O(m) bit comparisons per value comparison
  - This would yield O(n*log(n)*m) – always worse than RadixESort
  - But modern CPUs compare 64-bistrings in one cycle

Content of this Lecture

- Radix Exchange Sort
- Bucket Sort
  - Generalizing to arbitrary alphabets

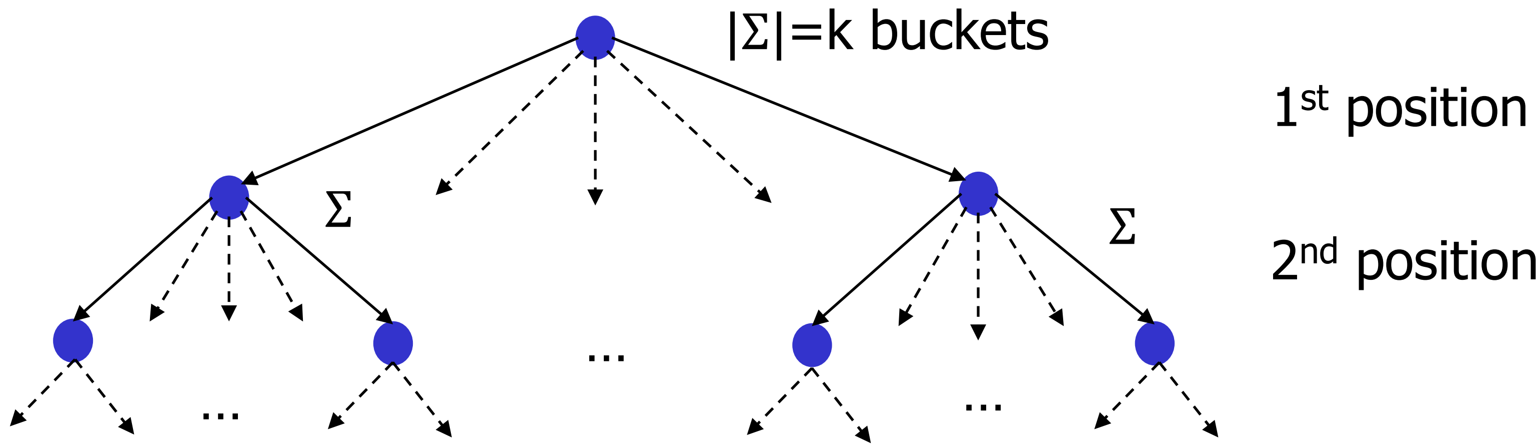
Bucket Sort

- What about sorting strings?
- Representing “normal” strings as bitstrings is a bad idea
  - One byte per character -> 8*length bits (large m for RadixESort)
  - But: Often 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 lexicographical order (e.g., "A" < "AA")
- We first sort S on first position into k buckets (with a single scan)
  - For bitstrings: k=2
- Then sort every bucket again for second position, etc.
- After at most m iterations, we are done
- Time complexity: O(m*n)
- But space is an issue



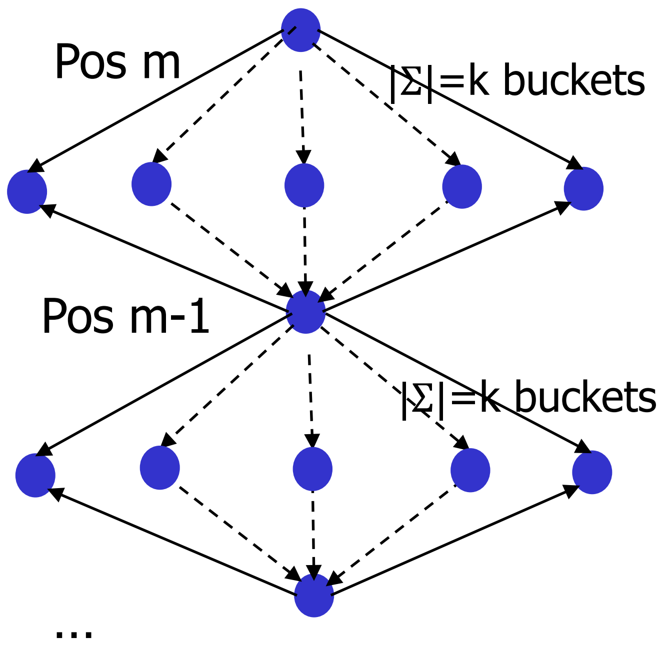
Space in Bucket Sort

- A naïve array-based implementation allocates k*n values for every phase of sorting into buckets
  - We do not know how many values start with a given character
  - Can be anything between 0 and |S|
- This would require $O((k*n)^m)$ space for the maximal m iterations – too much!
- We reduce this to O(k+n)
  - Requires a stable sorting algorithm for single characters
    - Recall: A sorting algorithm is stable, when the order of duplicates does not change during sorting
  - Note: 1-phase of Bucket Sort is stable

Bucket Sort – Idea

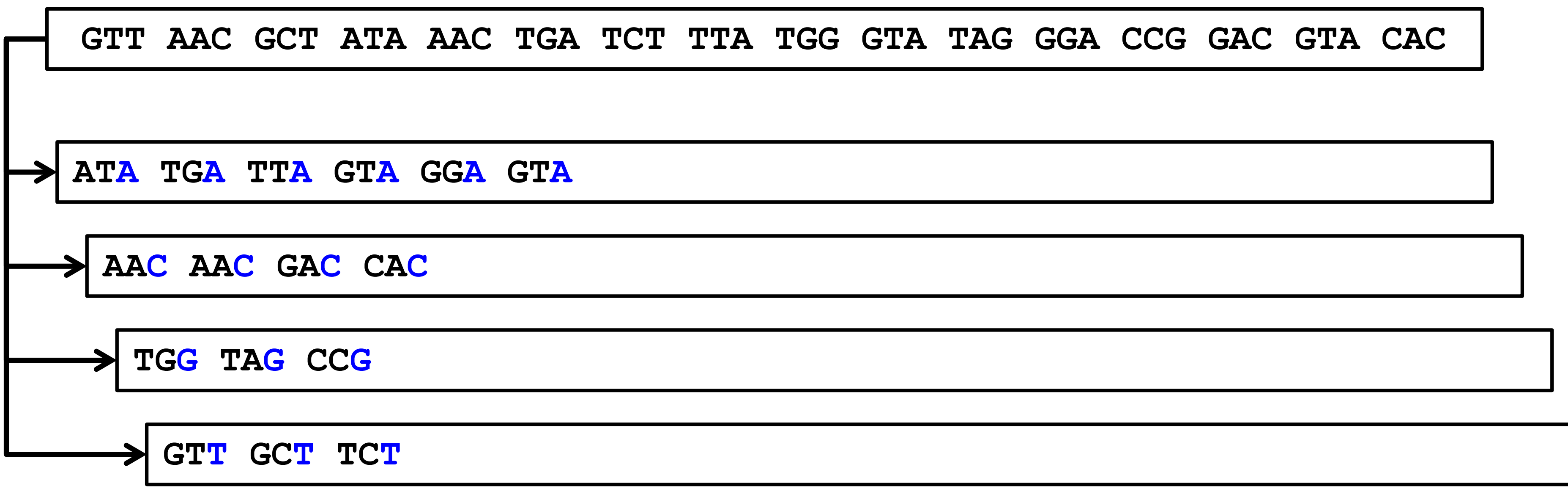
- If we sort from back-to-front and keep the order of sorted suffixes, we can re-use the additional space

```
func bucketSort(S array, m,k: integer){
  B:= array of queues with |B|=k
  for i := m down to 1 do
      # stable-sort elements of S into
        buckets depending on
        char at position i
      # merge all buckets into S
  end for
}
```



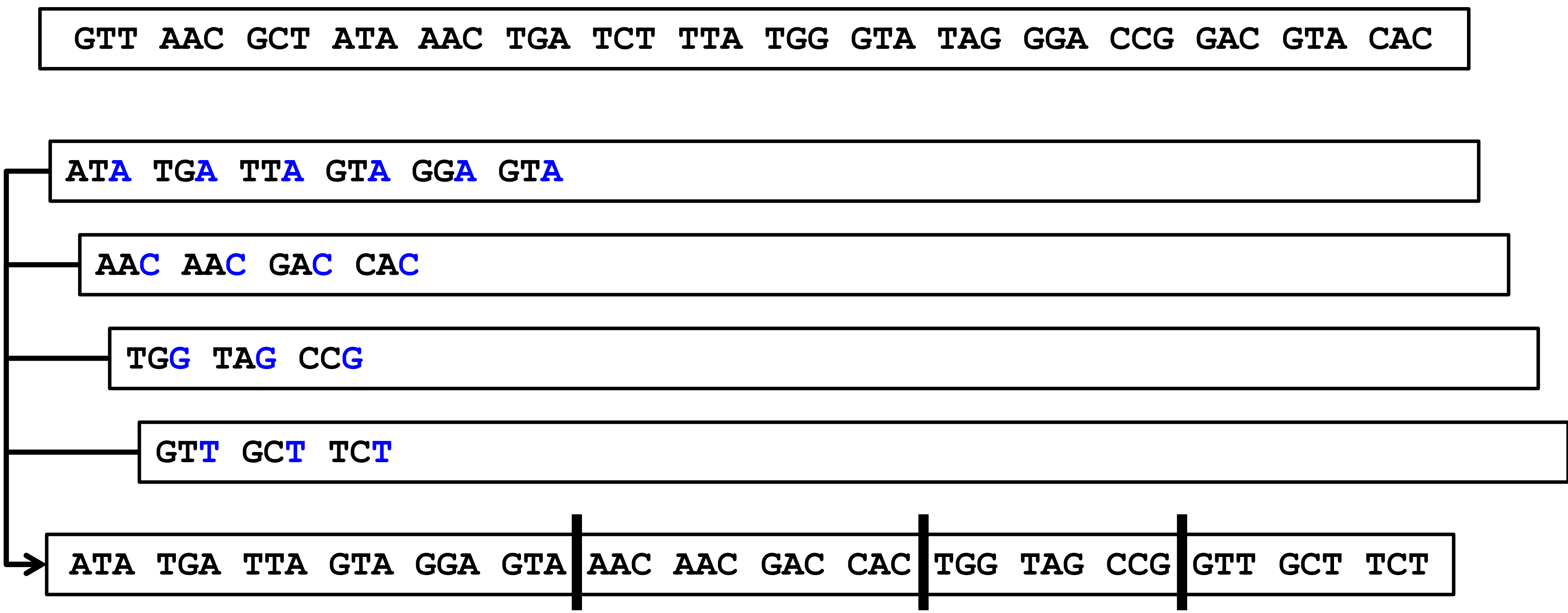
Bucket Sort – 1st Phase

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



Bucket Sort – Merge

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



Bucket Sort – 2nd and 3rd Phase

GTT AAC GCT ATA AAC TGA TCT TTA TGG GTA TAG GGA CCG GAC GTA CAC

ATA TGA TTA GTA GGA GTA | AAC AAC GAC CAC | TGG TAG CCG | GTT GCT TCT

ATA TGA TTA GTA GGA GTA AAC AAC GAC CAC TGG TAG CCG GTT GCT TCT

AAC AAC GAC CAC TAG | CCG GCT TCT | TGA GGA TGG | ATA TTA GTA GTA GTT

AAC AAC GAC CAC TAG CCG GCT TCT TGA GGA TGG ATA TTA GTA GTA GTT

AAC AAC ATA | CAC CCG | GAC GCT GGA GTA GTA GTT | TAG TCT TGA TGG TTA

AAC AAC ATA CAC CCG GAC GCT GGA GTA GTA GTT TAG TCT TGA TGG TTA

Bucket Sort – Pseudocode

- 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
    - E.g. map ‚A-Z‘ to 1-26
  - For very large alphabets, this might introduce additional complexity
- Stable: Always append to end of queue
- Finally, merge buckets and continue with next position

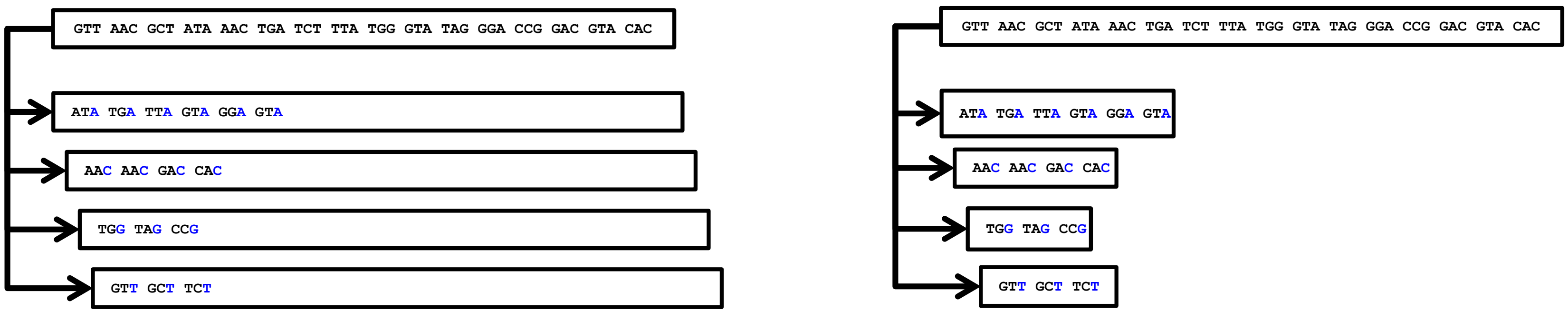
```
func bucketSort(S array,
                    m, k: integer){
  B:= Array of Queues with |B|=k
  for i := m down to 1 do
    for j := 1 to |S| do
      k := findBucket(S[j][i]);
      B[k].enqueue(S[j]);
    end for
    j := 1;
    for k := 1 to |B| do
      while not B[k].isEmpty() do
        append (S, j, B[k]);
        j := j + B[k].size;
      end while
    end for
  end for
  return S;
}
```

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 value from the right
- Groups all values from S with the same value in S[i][m-t+1] together and keeps them sorted wrt. (t-1)-suffixes
  - Assuming a stable sorting algorithm
- In the last phase (t=m), we sort by S[i][1]
  - S will be sorted by the m-suffixes of values
  - I.e., S is sorted
- qed.

Space Complexity

- Together: We only need O(k+n) additional space
  - Use a queue (e.g. a linked-list) for each bucket
  - Keep pointers 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)
  - Thus O(2*k+n) = O(k+n)



A Word on Names

- Names of these algorithms are not consistently used in the literature
  - 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)
  - …

Questions – Online Quiz

- Please go to **https://pingo.coactum.de**
- Enter ID: **729357**

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	O(m*(n+k))			O(n+k)	

Summary – For Strings

	Comps worst case	avg. case	best case	Additional space	Moves (wc / ac)
QuickSort	$O(m*n^2)$	O(?*n*log(n))	O(n*log(n))	O(log(n))	$O(n^2)$ / O(n*log(n))
BucketSort	O(m*(n+k))			O(n+k)	

Very pessimistic – most comparisons stop early

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.