Algorithms and Data Structures

Faster Sorting

Ulf Leser

Content of this Lecture

- Radix Exchange Sort
  - Sorting bitstrings in linear time (almost)
- Bucket 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 suggestion (max-min)/2 for selecting the pivot element in Quicksort (how can we do this for strings?)
- In the following, we focus on numbers
- 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 … |S|
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
  - A permutation of n different integers each between 1 and m with m>n
  - For a given value S[i], we do not any more know its rank
    - 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 and m<n
  - Again: We cannot any more infer the rank of S[i] from i alone

Second Example: Sorting Binary Strings

- Assume that all values are binary strings of equal length
  - E.g., integers in machine representation
- The most important position is the left-most one, and it can have only two different values
- Thus, 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 … |S|
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 last first 0

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

Sorting Complete Binary Strings

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

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

- We count the overall number of comparis… S[?][k] in radixES…
  - In divide*, w… element …
  - Then … in two di…
    - …ns (d-l)
    - …sons
    - …nd performs (r-d)
  - …ds, every call to radixESort yields 2*(r-l) comps
- Also the first one: |S|
- We are in O(n)?

Where is the error?

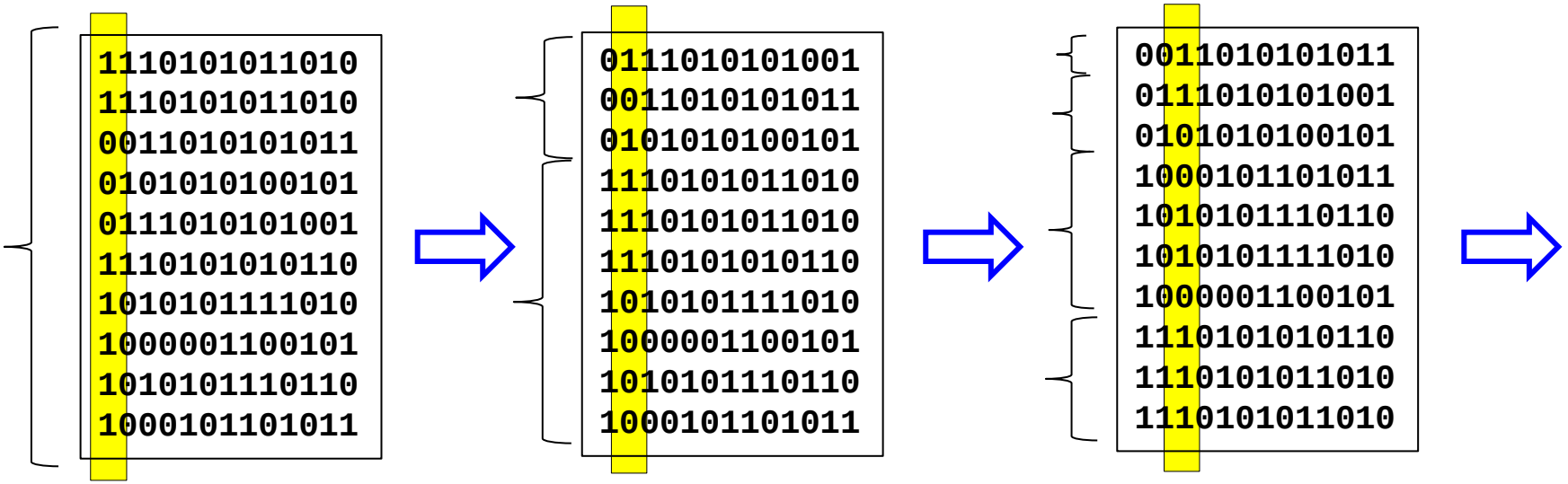
Complexity (Correct)

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

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

- 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
  - Every call to radixESort yields 2*(r-l) comps
- Recurs. depth is fixed to m
- Thus: O(m*|S|) comps

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

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

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

Bucket Sort

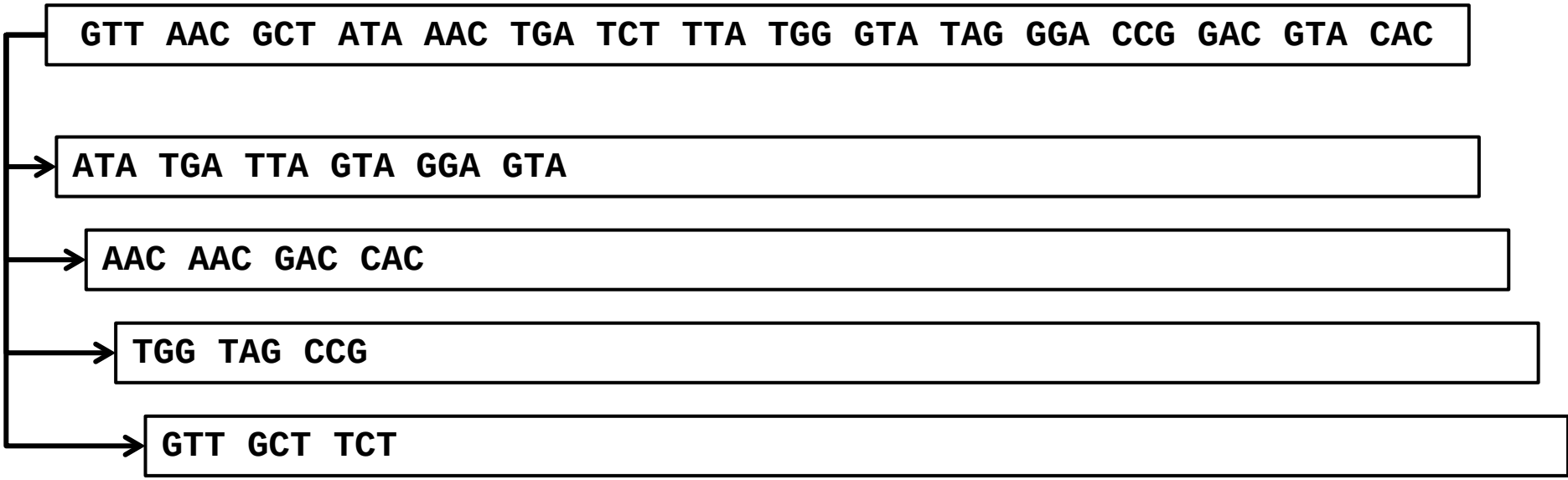
- Representing “normal” Strings as bitstrings is a bad idea
  - One byte per character -> 8*length bits (large m for RadixESort)
  - But: There are only ~25 different values (no case)
- One could find shorter encodings – we go a different way
- Bucket Sort generalizes RadixESort
  - Assume |S|=n, m being 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: O(m*|S|)
  - But space is a problem

Space in Bucket Sort

- A naïve implementation reserves 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 O(m*k*|S|) additional space – too much
- We reduce this to O(k*|S|)
  - Requires a stable sorting method

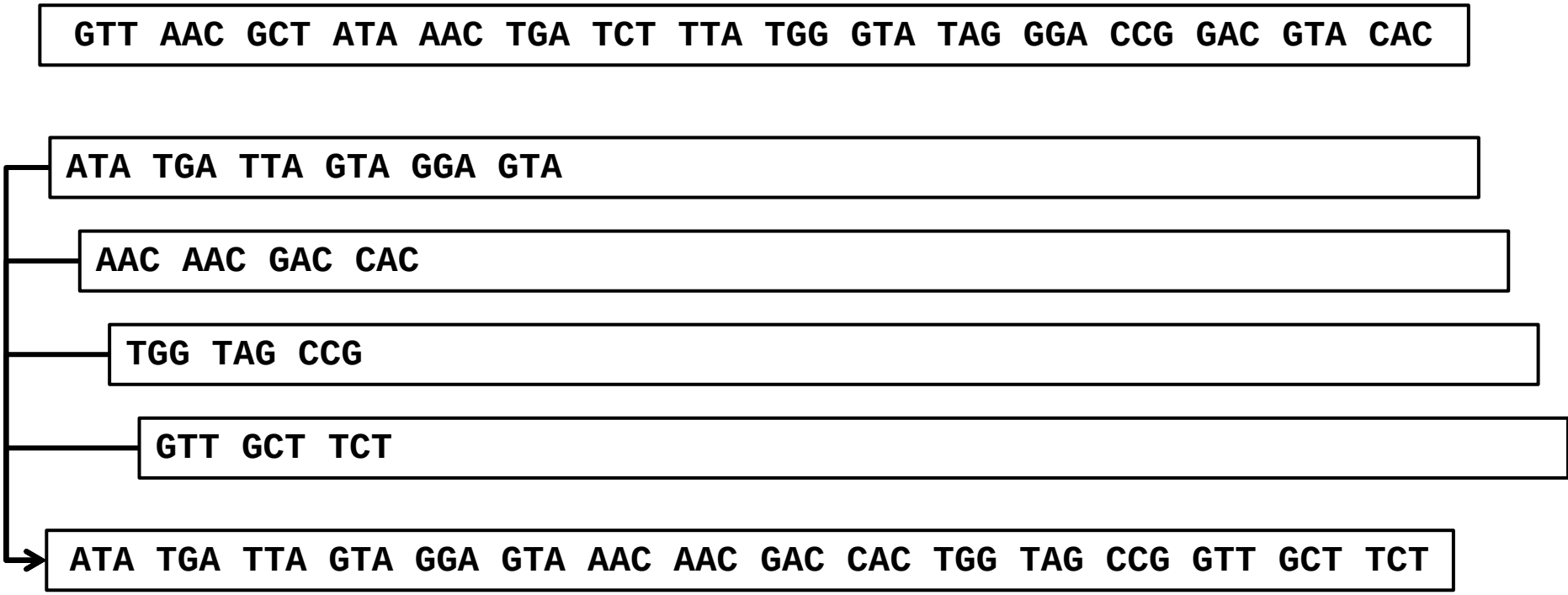
Bucket Sort

- 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



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

Magic? Proof

- By induction
- Assume that before phase t we have sorted all values by the (t-1)-suffix (right-most, least important val. for order)
  - True for t=2 – we sorted by the last character ((t-1)-suffixes)
- If 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 |S| additional space (all buckets together)
- We can use linked-lists for the buckets
  - But keep a pointer to the start (for copying) and the end (for extending)
  - Actually, we simply need a queue for each bucket

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 MSD 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)
  - ...