Algorithms and Data Structures

Sorting beyond Value Comparisons

Marius Kloft

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 (refused) suggestion (min+max)/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

```
S: array_permuted_numbs;
B: array_of_size_|S|
for i:= 1 … |S|
  B[S[i]] := S[i];
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
  - 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 and 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 (bitstrings) 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
- 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

```
S: array_bitstrings;
B0: array_of_size_|S|
B1: array_of_size_|S|
j0 := 1;
j1 := 1;
for i:= 1 … |S|
  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

- 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 0

```
func int divide*(S array;
                      k,l,r: int) {
  i := l-1;
  j := r+1;
  while true
    repeat
      i := i+1;
    until S[i][k]=1 or i≥j;
    repeat
      j := j-1;
    until S[j][k]=0 or i≥j;
    if S[j][k]=1 then j--;
    if i≥j then
      break while;
    end if;
    swap( S[i], S[j]);
  end while;
  return j;
}
```

Sorting Complete Binary Strings

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

- 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 be the length (in bits) of the values in S
  - Call with `radixESort(S,1,1,|S|)`

Complexity

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

```
func int divide*(S array;
                 k,l,r: int) {
  …
  while true
    repeat
      i := i+1;
    until S[i][k]=1 or i≥j;
    repeat
      j := j-1;
    until S[j][k]=0 or i≥j;
    …
  end while;
  return j;
}
```

- We count the overall number of comparisons
  - In divide*, we look at every element S[l…r] exactly once
  - Then we divide S[l…r] in two disjoint halves
    - First performs (d-l) comparisons
    - Second performs (r-d)
  - Thus, every call to radixESort yields 2*(r-l) comps
- We are in O(n)?

Complexity

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

```
func int divide*(S array;
                  k,l,r: int) {
  …
  while true
    repeat
      i := i+1;
    until S[i][k]=1 or i≥j;
    repeat
      j := j-1;
    until S[j][k]=0 or i≥j;
    …
  end while;
  return j;
}
```

- We count the overall number of compariso
  - In divide*, we lo element S[l...r
  - Then we wo disjoi

    -l)

    erforms (r-d)

    very call to radixESort

    s 2*(r-l) comps
- e are in O(n)?

Where is the error?

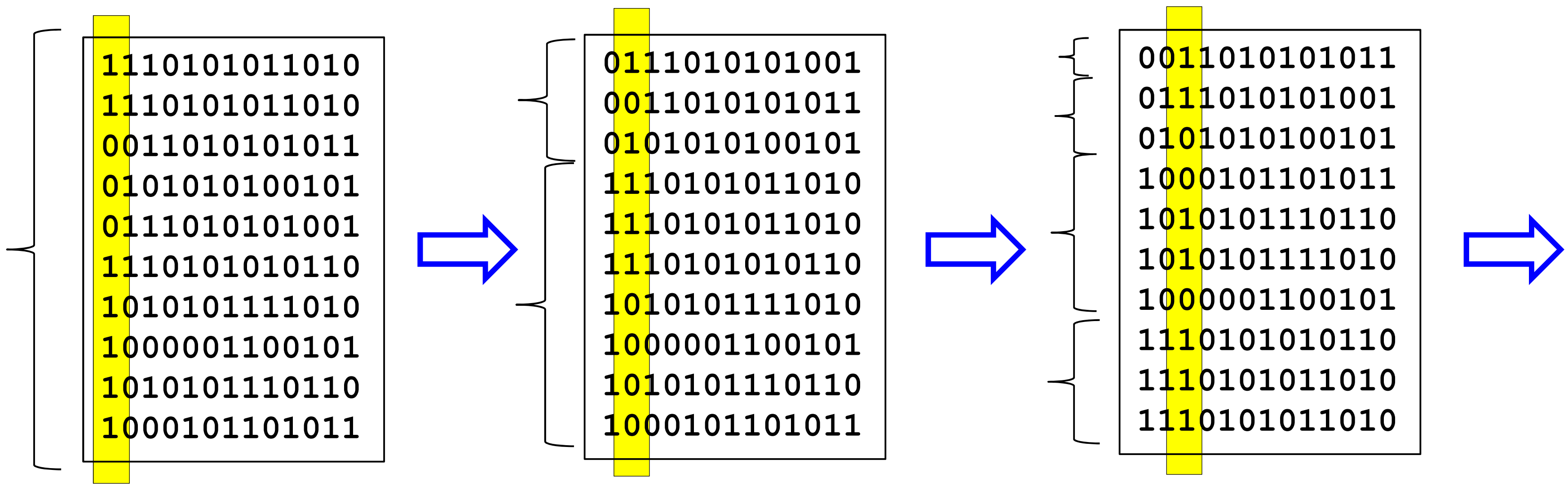
Complexity (Correct)

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

```
func int divide*(S array;
                  k,l,r: int) {
  …
  while true
    repeat
      i := i+1;
    until S[i][k]=1 or i≥j;
    repeat
      j := j-1;
    until S[j][k]=0 or i≥j;
    …
  end while;
  return j;
}
```

- 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 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: $m<\log(n)$ or $m>\log(n)$?
  - 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

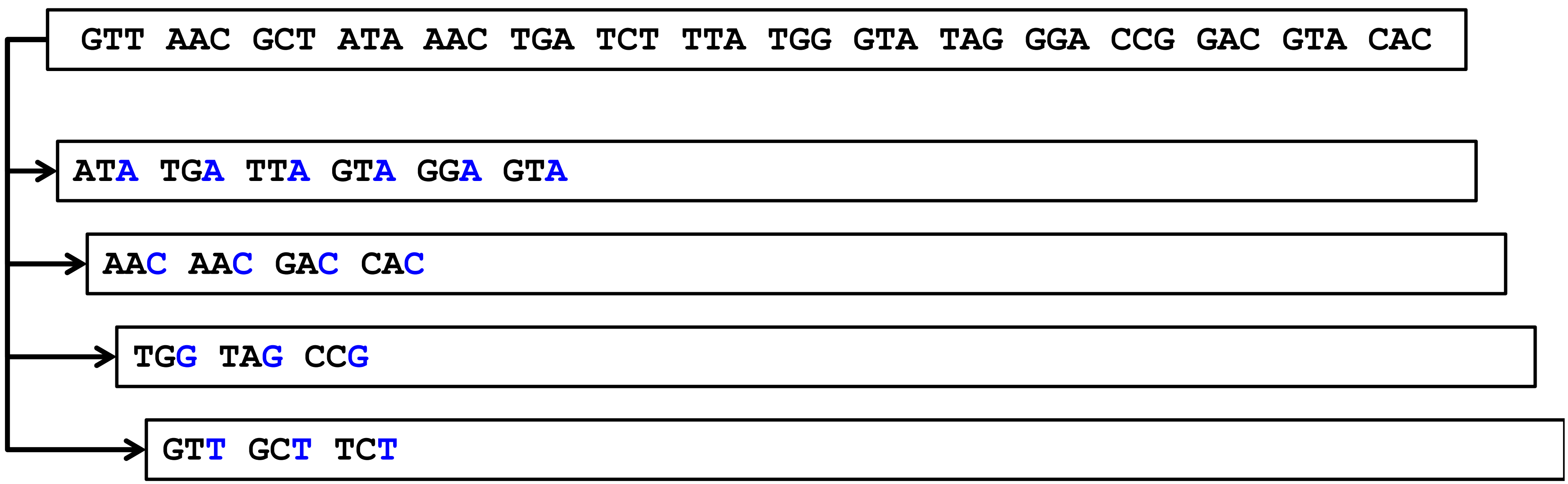
- Representing “normal” Strings as bitstrings is a bad idea
  - One byte per character -> 8*length bits (large m for RadixESort)
  - But: There are only 29 different values (A-Z,Ä,Ö,Ü; no caps)
- Bucket Sort generalizes RadixESort
  - Assume |S|=n, m being the length of the largest value, alphabet Σ 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+k)
  - 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|) and then O(k+|S|)
  - Requires a stable sorting method for single characters
  - 1-phase Bucket Sort is stable (if implemented that way)

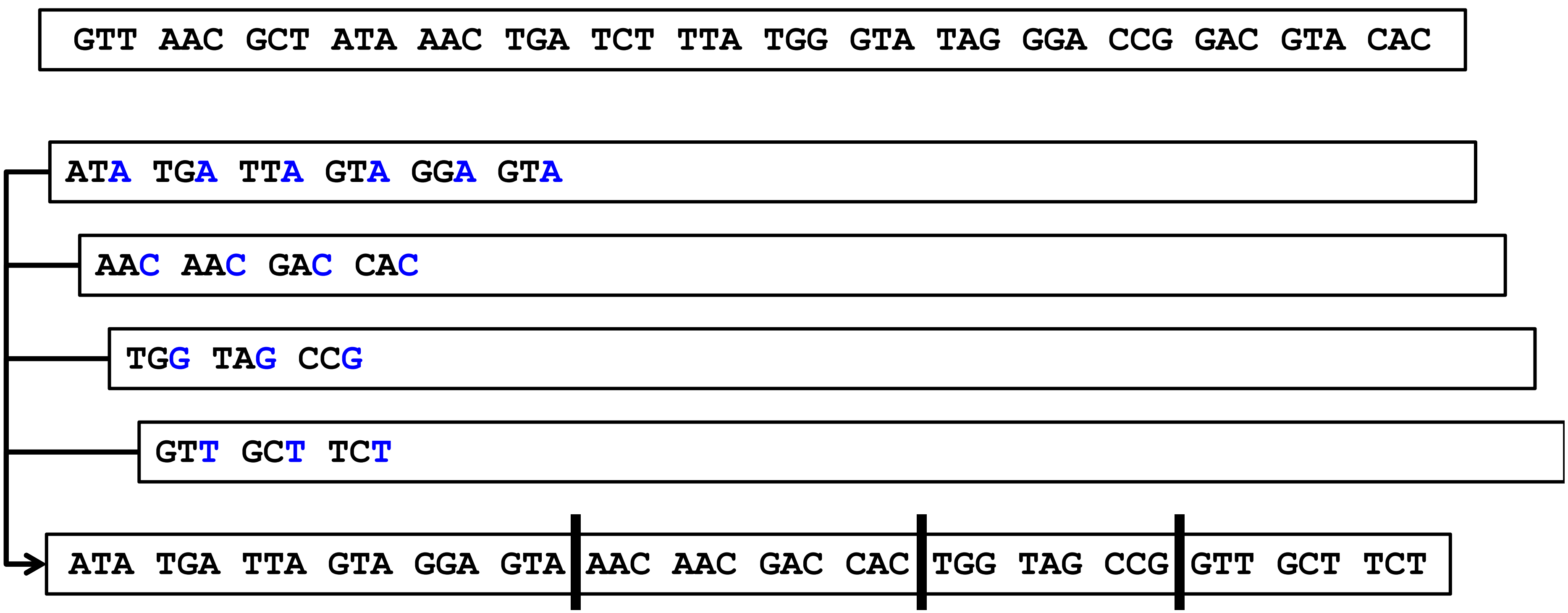
Bucket Sort

- 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

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

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)
- If phase t, we sort by the t‘th character (from the right)
- This will group all values from S with the same value in S[i][m-t+1], i=1,…,n, 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|+k additional space (all buckets together)
  - Use a linked-list 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

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)
  - …

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.