Algorithms and Data Structures

Faster Sorting

Ulf Leser
Content of this Lecture

- **Quick Sort**
  - One of the fastest sorting algorithms in practice
- **Radix Exchange Sort**
- **Bucket Sort**
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 two 50% of \(|S|\) (2, 4, 8, ..., \(\text{ceil}(n/2)\))
  – We cannot easily merge two such 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 the “merge” in-place
    • But is it another kind of merge
Main Idea

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

pivot elements

k
k
k'
k''
k
k'
k''
k
k'
k''
QuickSort Framework

• Start with qsort(A, 1, |A|)

• 6: “Sort” A around the pivot element (divide)
  – Problem1: Chose k
  – Problem2: Do this in-place

• 7: Sort all values smaller-or equal than pivot element

• 8: Sort all values larger-or-equal than pivot element

• Problem3: How often do we need to do this?

```c
1. func void qsort(A array;  
2. 1,r integer) {  
3.  if r≤1 then  
4.    return;  
5.  end if;  
6.  pos := divide( A, l, r);  
7.  qsort( A, l, pos-1);  
8.  qsort( A, pos+1, r);  
9. }
```
Addressing P1 – approaching P3

- P1: We need to chose \( k \) (\( A[k] \))
- \( A[k] \) determines the sizes of \( A' \) and \( A'' \)

- \( A[k] \) in the middle of the values of \( A \)
  - \( A' \) and \( A'' \) are of equal size (\(~|A|/2\))
  - Creates a “nice” search tree

- \( A[k] \) at the border of the values of \( A \)
  - \( |A'|\sim0 \) and \( |A''|\sim|A|-1 \) or vice versa
  - Creates a “bad” search tree

- **Hint to P3**: Somewhere in \([\log(n), n]\) times
  - Depending on choice of \( A[k] \)
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
  – Also sometimes more sensible: How long will you need for your Bachelor? 6,7 semesters?
P1: Choosing k

- In the best case, $A[k]$ is the median of $A$
  - If $A$ 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 scan the array until we find a value that is 10% or less different, and use this value as pivot
    - If $A$ is large and randomly drawn from a set of incomes, this scan will be very short
  - If $A$ is an array of family names in Berlin, we take the Berlin telephone book, open it roughly in the middle, and scan the array until we find a value that is 10% or less different
    - See above

- But 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: Scan A to find min/max; search $A[k] \sim (\text{max} - \text{min})/2$
  - Why should the values in A be equally distributed in this range?
  - For instance: Incomes are not equally distributed in their range
- Option: Choose set of values X from A at random and search $A[k] \sim \text{median}(X)$
  - X follows the same distribution (same median) as A, but $|X| < < |A|$
  - Since this procedure would have to be performed for each qSort, only small X do not influence runtime a lot
- More popular: Choose k at random
  - Often, one uses the last value in the array
  - Also relieves from searching an appropriate $A[k]$
  - We’ll see that this already produces quite good result on average
P2: Do this in-place

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

```plaintext
1. func integer divide(A array; 
2. l,r integer) {
3. val := A[r];
4. i := l-1;
5. j := r;
6. while true
7. repeat
8. i := i+1;
9. until A[i]>=val;
10. repeat
11. j := j-1;
12. until A[j]<=val or j<i;
13. if i>j then
14. break while;
15. end if;
16. swap( A[i], A[j]);
17. end while;
18. swap( A[i], A[r]);
19. return i;
20. }
```
Example
P2: Complexity

- **# of comparisons: O(l-r)**
  - 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(l-r) in worst case**
  - Example: 8,7,8,6,1,3,2,3,5
  - Gives \( \sim \frac{(r-l)}{2} \) swaps

```plaintext
```
Worst-Case for Complete Quick Sort

- Worst case for # comparisons:
  A reverse-sorted list
  - A[r] always is the smallest element
  - Requires r-l comparisons in every call of divide()
  - Every pair of qSort’s has |A’|=0 and |A”|=n-1
  - This gives \((n-1)+((n-1)-1)+...+1 = O(n^2)\)
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 A contains all values 1...|A| in arbitrary order
  – If A 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 A with equal probability 1/n
• This choice divides A such that |A’|=k-1 and |A’’|=n-k
• Let T(n) be the average runtime. Then:

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

  – Where bn is the time to divide the array and T(0)=0
Some Lengthy Calculations

• We need to show that, for some c independent of n:
  \[ T(n) \leq c \cdot n \cdot \log(n) \]

• We proof by induction
  – Clearly, \( T(1) = 0 \leq 1 \cdot \log(1) \)
  – We assume that the assumption holds for all \( i < n \)
  – Then

  \[
  T(n) \leq \frac{2c}{n} \sum_{k=1}^{n-1} k \cdot \log(k) + bn \\
  \leq ... \]

  \[
  \leq c \cdot n \cdot \log(n) - \frac{cn}{4} - \frac{c}{2} + bn \]

Which holds for \( c \geq 4b \)

See Ottman/Widmayer for details
Conclusion

• Great
• Although there are cases where we need $O(n^2)$ comparisons, these are so rare in the set of all possible permutations of values, that we do not need more than $O(n \times \log(n))$ comparisons on average
• In other words: If we sum the runtimes of qSorts over many (all) different orders of $n$ values (for different $n$), then this sum will grow with $n \times \log(n)$, not with $n^2$

• One can show the same for the # of swaps
• qSort is a really fast general-purpose sorting algorithm
Looking at Space Again

- We were too sloppy
- qSort 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

- Idea for worst-case space complexity $O(\log(n))$
  - In the recursive decent, always first choose the smaller of the two sub-arrays ($A'$ or $A''$, whatever is smaller)
    - This branch of the search tree can have at most $O(\log(n))$ calls, as the smaller array always is smaller than $|A|/2$ (or it would not be the smaller one)
  - Use iteration (no stack) to sort the greater array afterwards
Implementation

```
1. func integer qSort(A array; 1, r int) {
2.     if r ≤ l then
3.         return;
4.     end if;
5.     val := A[r];
6.     i := l-1;
7.     j := r;
8.     while true
9.         repeat
10.            i := i+1;
11.            until A[i] > val;
12.     end while;
13.     j := j-1;
14.     until A[j] < val or j < i;
15.     if i > j then
16.         break while;
17.     end if;
18.     swap( A[i], A[j] );
19.     end while;
20.     swap( A[i], A[r] );
21.     qSort(A, l, i-1);
22.     qSort(A, i+1, r);
23. }  
```

```
1. func integer qSort++(A array; 1, r int) {
2.     if r ≤ l then
3.         return;
4.     end if;
5.     val := A[r];
6.     i := l-1;
7.     j := r;
8.     while true
9.         repeat
10.            i := i+1;
11.            until A[i] > val;
12.     end while;
13.     j := j-1;
14.     until A[j] < val or j < i;
15.     if i > j then
16.         break while;
17.     end if;
18.     swap( A[i], A[j] );
19.     end while;
20.     swap( A[i], A[r] );
21.     qSort(A, l, i-1);
22. }  
```
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) makes the same procedure performing the next sort
• We turned a linear tail recursion into an iteration without stack

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

Ulf Leser: Alg&DS, Summer semester 2011 22
### Summary

<table>
<thead>
<tr>
<th></th>
<th>Comps worst case</th>
<th>avg. case</th>
<th>best case</th>
<th>Additional space</th>
<th>Moves (wc / ac)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td></td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log(n))$</td>
<td></td>
<td>$O(n \log(n))$</td>
<td>$O(n)$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$O(n^2)$</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(\log(n))$</td>
<td>$O(n^2) / O(n \log(n))$</td>
</tr>
</tbody>
</table>
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**? Depends ... log(n) usually is not a lot of space
Content of this Lecture

- Quick Sort
- Radix Exchange Sort
  - Sorting bitstrings in linear time (well, 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?)
• Let us for now concentrate on numbers
• First example
  – Assume a set A of n different integers, \( \forall i: 1 \leq A[i] \leq n \)
  – How can we sort A in \( O(n) \) time and with only n extra space?
Sorting Permutations

- Very easy
  - 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 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 seen from the value

1. \(A\): array_permuted_numbs;
2. \(B\): array_of_size_\(|A|\)
3. for \(i := 1 \ldots |A|\)
4. \(B[A[i]] := A[i]\);
5. end for;
Removing Pre-Requisites

- **Assume A is not dense**
  - A permutation of n different integers each between 1 and m (m > n)
  - For a given value A[i], we do not any more know its rank
    - How many values are smaller?
    - At most min(A[i], n)
    - At least max(n - (m - A[i]), 0)
  - This is almost the usual sorting problem, and we cannot do much

- **Assume A has duplicates**
  - A contains n values, where each value is between 1 and m and appears in A (m < n)
  - Again we cannot any more infer the rank of A[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, high byte first
- The most important position now 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

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

- How can we do this in $O(1)$ additional space?
- Recall QuickSort
  - Note that we return (j) the position of the last first 0
  - First call $\text{divide}^*(A, 1, 1, |A|)$
  - $[k, l, r, \text{and return value will be used in a minute}]$

```c
1. func int divide*(A 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 A[i][k]=1 or i≥j;
9. repeat
10. j := j-1;
11. until A[j][k]=0 or i≥j;
12. if i≥j then
13. break while;
14. end if;
15. swap( A[i], A[j]);
16. end while;
17. return j;
18. }
```
Sorting Complete Binary Strings

- 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 $A$ again, but only the subarray with same values in the $(k-1)$ first positions
  - Let $m$ by the length (in bits) of the values in $A$
  - Call with $\text{radixESort}(A, 1, 1, |A|)$

```plaintext
1. func radixESort(A array; k, l, r: integer) {
2.   if k>m then
3.     return;
4.   end if;
5.   d := divide*(A, k, l, r);
6.   radixESort(A, k+1, l, d);
7.   radixESort(A, k+1, d+1, r);
8. }
```
We count the overall number of comparisons of \( A[?] [k] \) in \texttt{radixESort}.

- In \texttt{divide*}, we look at every element \( A[l..r] \) exactly once.
- Then we divide \( A[l..r] \) in two disjoint halves.

\texttt{radixESort} performs \( (d-l) \) comparisons and performs \( (r-d) \) comparisons.

Thus, every call to \texttt{radixESort} yields \( 2 \cdot (r-l) \) comps.

- Also the first one: \(|A|\)
- We are in \( O(n) \)?
Complexity (Correct)

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

- We count ...
  - Every call to radixESort first performs \((r-l)\) comps and then divides \(A[l...r]\) in two disjoint halves
    - \(1^{st}\) makes \((d-l)\) comps
    - \(2^{nd}\) makes \((r-d)\) comps
  - Every call to radixESort yields \(2*(r-l)\) comps
- Recurs. depth is fixed to \(m\)
- Thus: \(O(m*|A|)\) comps
• For every $k$, we look at every $A[i][k]$ once to see whether it is 0 or 1 – together, we have at most $m^*|A|$ comparisons
  – Of course, we can stop at every interval with $(r-l)=1$
  – $m^*|A|$ 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
- If A is large / maximal bitstring length is small: RadixESort
- If A 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
BucketSort

- Representing “normal” Strings as bitstrings is not necessarily the best idea
  - One byte per character -> 8*length bits (large m for RadixESort)
  - But: There are only ~25 different values (no case)

- BucketSort sort-of generalizes RadixESort
  - Assume |\(A| = n\), m being the length of the largest value, alphabet \(\Sigma\) with |\(\Sigma\)| = k and a lexicographical order (e.g., “A” < “AA”)
  - We first sort A 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: \(O(m*|A|)\)
  - But space is a problem
Space in BucketSort

- A naïve implementation reserves $k|A|$ 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 $|A|$
- This would need $O(m^*k^*|A|)$ additional space – too much!
- We first reduce this to $O(k^*|A|)$
  - Requires a stable sorting method
BucketSort

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

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

- 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, lowest values)
  - 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 \(A\) with the same value in \(A[i][m-t+1]\) together and keep them sorted wrt. \((t-1)\)-suffixes
  - Assuming a stable sorting algorithm
- Since we sort by \(A[i][m-t+1]\), the array after phase \(t\) will be sorted by the \(t\)-suffix
- \(\text{qed.}\)
Saving More Space

• The example has shown that we actually never need more than $|A|$ 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 simple 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“ (OM), 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comps worst case</th>
<th>avg. case</th>
<th>best case</th>
<th>Additional space</th>
<th>Moves (wc / ac)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>O(n)</td>
<td>O($n^2$)</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>O(n)</td>
<td>O($n^2$)</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>O(n)</td>
<td>O($n\log(n)$)</td>
<td>O($n\log(n)$)</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$O(n^2)$</td>
<td>$O(n\log(n))$</td>
<td>$O(n\log(n))$</td>
<td>O($\log(n)$)</td>
<td>O($n^2$) / O($n\log(n)$)</td>
</tr>
<tr>
<td>BucketSort (m= ...)</td>
<td>$O(m*n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>