Algorithms and Data Structures

Sorting:
Merge Sort and Quick Sort
# Summary

<table>
<thead>
<tr>
<th></th>
<th>Comparisons worst case</th>
<th>Comparisons best case</th>
<th>Additional space</th>
<th>Moves worst/best</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>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2) / O(n)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2) / O(1)$</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>
</tr>
<tr>
<td>Magic Sort (?)</td>
<td>$O(n)$</td>
<td></td>
<td></td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Content of this Lecture

- Merge Sort
- Quick Sort
Central Idea for Improvements in Sorting

- Methods we analyzed so-far did not optimally exploit transitivity of the „greater-or-equal“ relationship
  - If $x \leq y$ and $y \leq z$, then $x \leq z$
- If we compared $x$ and $y$ and $y$ and $z$, there is no need any more to compare $x$ and $z$
  - But all our simple algorithms compare every element with every element – at least once
- The clue to lower complexity algorithms for sorting is finding systematic (algorithmic) ways to exploit such information
Merge Sort

- There are various algorithms with $O(n \cdot \log(n))$ comparisons
- (Probably) Simplest one: **Merge Sort**
  - Divide-and-conquer algorithm
  - Break array in two partitions of equal size
  - Sort each partition recursively if it has more than 1 elements
  - Merge sorted partitions
- Merge Sort is not in-place: $O(n)$ additional space
Illustration

Source: WikiPedia
Illustration

Divide - Partition

Conquer - Merge
- Here we exploit transitivity
- We save comparisons during merge because both sub-lists are sorted

Source: WikiPedia
Function `void mergesort(S array; l, r integer)` {
  if (l < r) then
    # Sort each ~50% of array
    m := (r-l) div 2;
    mergesort(S, l, l+m);
    mergesort(S, l+m+1, r);
    # Merges two sorted lists
    merge(S, l, l+m, r);
  else
    # Nothing to do, 1-element list
    end if;
}
Merging Two Sorted Lists

- There is not much sorting – work is done in the **merge step**
- Recall: Intersection of two sorted doc-lists in IR
- Idea
  - Move **one pointer through each list**
  - Whatever element is smaller, copy to a new list and increment this pointer
    - “New list” requires **additional space**
  - Repeat until one list is exhausted
  - Copy rest of other list to new list
  - Note: You cannot do this in-place

```
1  2  3  4  7  8  9  11  12 ...
```

```
1  2  3  4  5  6  7  8  9 ...
```
Example

\[
\begin{array}{c|c|c|c}
\hline
1 & 4 & 7 & 8 \\
\hline
1 & 4 & 7 & 8 \\
\hline
1 & 4 & 7 & 8 \\
\hline
1 & 4 & 7 & 8 \\
\hline
\end{array}
\]
function void merge(S array;
    l,m,r integer) {
    B: array[1..r-l+1];
    i := l;       # Start of 1st list
    j := m+1;     # Start of 2nd list
    k := 1;       # Target list
    while (i<=m) and (j<=r) do
        if S[i] \leq S[j] then
            B[k] := S[i];  # From 1st list
            i := i+1;
        else
            B[k] := S[j];  # From 2nd list
            j := j+1;
        end if;
        k := k+1;      # Next target
    end while;
    if i>m then     # What remained?
        copy S[j..r] to B[k..k+r-j];
    else
        copy S[i..m] to B[k..k+m-i];
    end if;
    # Back to original list
    copy B[1..r-l+1] to S[l..r];
}
Complexity

• Theorem

*Merge Sort requires $\Omega(n \times \log(n))$ and $O(n \times \log(n))$ comparisons*

• Proof of $O(n \times \log(n))$
  - Merging two sorted lists of size $n$ requires $O(n)$ comparisons
    • After every comp, 1 element is moved to target; there are only $2 \times n$ elements; thus, there can be only $2 \times n$ comparisons
  - Merge Sort calls MergeSort twice with always ~half of the array
    • Let $T(n)$ be the number of comparisons
    • Thus: $T(n) = T(n/2) + T(n/2) + O(n)$
  - This is $O(n \times \log(n))$
    • See recursive solution of max subarray

• $\Omega(n \times \log(n))$: # comparisons does not depend on data in S
Remarks

• Merge Sort is worst-case optimal: Even in the worst of all cases, it does not need more than (in the order of) the minimal number of comparisons
  - Given our lower bound for sorting

• But there are also disadvantages
  - $O(n)$ additional space
  - Requires $\Omega(n\log(n))$ moves
    • Sorted sub-arrays get copied to new array in any case
    • See Ottmann/Widmayer for proof

• Note: Basis for sorting algorithms on external memory
## Summary

<table>
<thead>
<tr>
<th>Sort</th>
<th>Comparisons worst case</th>
<th>Comparisons best case</th>
<th>Additional space</th>
<th>Moves worst/best</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>
</tr>
<tr>
<td>Insertion Sort</td>
<td>(O(n^2))</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n^2) / O(n))</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>(O(n^2))</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n^2) / O(1))</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>(O(n^2 \log(n)))</td>
<td>(O(n^2 \log(n)))</td>
<td>(O(n))</td>
<td>(O(n^2 \log(n)))</td>
</tr>
</tbody>
</table>
Content of this Lecture

• Merge Sort
• **Quick Sort**
  - Algorithm
  - Average Case Analysis
  - Improving Space Complexity
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 50% of |S| (2, 4, 8, ..., ceil(n/2))
  - We cannot easily merge **two sorted 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 a kind-of “merge” in-place
Main Idea

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

\[ k \rightarrow \rightarrow S[k] \rightarrow \rightarrow p \]

\[ k' \rightarrow \rightarrow p' \rightarrow \rightarrow p \]
Quick Sort Framework

• Start with qsort(S, 1, |S|)
• “Sort” S around the pivot element p (divide)
  - Problem 1: Choose k (i.e. p)
  - Problem 2: Do this in-place
• Recursively sort values smaller-or equal than pivot element
• Recursively sort values larger-or-equal than pivot element
• Problem 3: How often do we need to do this?

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

- **P1**: We need to choose k (p=S[k])
- p determines the sizes of S’ and S”

- **Best**: p in the **middle of the values of S** (median)
  - S’ and S” are of equal size (~|S|/2)
  - Creates a low search tree

- **Worst**: p at the **border of the values of S**
  - |S’| ~0 and |S”| ~|S| - 1 or vice versa
  - Creates a deep search tree

- **Hint to P3**: Somewhere in [log(n), n] times
  - Depending on choice of k
Intermezzo: 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 **sensitive 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
  - How long will you need for your Bachelor? 6,35 semesters?
  - German median net income (2010) was 24.152€ – but average?
P1: Choosing k

- In the best case, p is the median of S
- Approximations
  - If S 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 S is large and randomly drawn from a set of incomes, this scan will be very short
  - If S is an array of family names in Berlin, we take the Berlin telephone book, and open it roughly in the middle
- 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 1:** Find min/max in $S$; search $k$ with $p \sim \frac{(\text{max}-\text{min})}{2}$
  - Why should the values in $S$ be *equally distributed* in this range?
  - For instance: Incomes are not equally distributed at all

- **Option 2:** Choose a (small) set of values $X$ from $S$ at random and determine $k$ with $p \sim \text{median}(X)$
  - $X$ follows the same distribution (same median) as $S$, but $|X| \ll |S|$
  - Since this procedure would have to be performed for each qSort, only very small $X$ do not influence runtime a lot
  - But: Small $X$ will lead to bad median estimations
  - Beware: If $|X| = c \times |S|$ for any $c$, we are still in $O(|S|)$

- **Option 3:** Choose $k$ at random
  - For instance, simply use the last value in the array
  - We’ll see that this already produces *good result on average*
Recall: Quick Sort Framework

- Start with qsort(S, 1, |S|)
- “Sort” S around the pivot element (divide)
  - Problem 1: Choose k
  - Problem 2: Do this in-place
- Recursively sort values smaller-or equal than pivot element
- Recursively sort values larger-or-equal than pivot element
- Problem 3: How often do we need to do this?

1. func void qsort(S array; 
2.                 l,r integer) { 
3.   if r≤l then  
4.     return;  
5.   end if;  
6.   pos := divide( S, l, r);  
7.   qsort( S, l, pos-1);  
8.   qsort( S, pos+1, r);  
9. }
Problem P2: Do this in-place

- We use \( k=r \) (random choice of \( p \))
- Simple idea
  - Search from \( l \) towards \( r \) until first value greater-or-equal \( p \)
  - Search from \( r \) towards \( l \) until first value smaller-or-equal \( p \)
  - Swap these two values
  - Repeat if \( i \) has not reached \( j \) yet
  - Result: Values left from \( i \) are smaller than \( p \) and values right from \( j \) are larger than \( p \)
  - Move \( p \) into the middle

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

```
1 8 6 3 5 9 3 1 7
  i   j
1 1 6 3 5 9 3 8 7
  i   j
1 1 6 3 5 3 9 8 7
  j   i
1 1 6 3 5 3 7 8 9
```

```
1 1 6 3 5 3
  i  j
1 1 3 6 5 3
   j  i
1 1 3 3 5 6
```

```
1 1 3
  j  i
```

```
5 6
```

```
8 9
```
P2: Complexity of divide()

- **# of comparisons: O(r-l)**
  - 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(r-l) in worst case**
  - Example: 8,7,8,6,1,3,2,3,5
  - Requires ~ (r-l)/2 swaps

```c
1. func int divide(S array;
2.                 l,r integer) {
3.   val := S[r];
4.   i := l;
5.   j := r-1;
6.   repeat
7.     while (S[i]<=val and i<r)  
8.       i := i+1;
9.   end while;
10.    while (S[j]>=val and j>l)   
11.      j := j-1;
12.   end while;
13.   if i<j then                 
14.     swap( S[i], S[j]);
15.   end if;
16.   until i>=j;
17.   swap( S[i], S[r]);
18.   return i;
19. }
```
Worst-Case Complexity of Quick Sort

- Worst case: A reverse-sorted list and \( k = |S| \)
  - \( S[r] \) in first iteration is the smallest element, later always the smallest or the largest
  - Requires \( r-l \) comparisons in every call of `divide()`
  - Every pair of qSort's has \( |S'| = 0 \) and \( |S''| = n-1 \)
  - This gives \( (n-1) + ((n-1)-1) + \ldots + 1 = O(n^2) \)
Intermediate Summary

- Great *disappointment*
- We are in O(1) additional space, but as slow as our basic sorting algorithms in worst case
- Let’s look at the *average case*
Content of this Lecture

- Merge Sort
- Quick Sort
  - Algorithm
  - Average Case Analysis
  - Improving Space Complexity
Average Case

- Without loss of generality, we assume that $S$ contains all values $1 \ldots |S|$ in arbitrary order
  - If $S$ had duplicates, we would at best save swaps
  - Sorting $n$ different values is the same problem as sorting the values $1 \ldots n$ – replace each value by its rank
- For $k$, we choose any value in $S$ with equal probability $1/n$
- This choice divides $S$ such that $|S'|=k-1$ and $|S''|=n-k$
- Let $T(n)$ be the average # of comparisons. 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 $b \times n$ is the time to divide the array and $T(0)=0$
Induction

- We need to show that, for some $c$ independent of $n$:

$$T(n) \leq c \cdot n \cdot \log(n)$$

- **Proof by induction** (for $n \geq 2$)
  - Clearly, $T(1) = b$, $T(2) = 3b \leq c \cdot 2 \cdot \log(2)$ if $c \geq 3b/2$
  - We assume the above assumption holds for all $2 \leq k < n$
  - We start with (for simplicity, assume $n = 2^x$ for some $x$):

$$T(n) = 2 \sum_{k=1}^{n-1} T(k) + bn$$
Induction

\[ T(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + bn \]
\[ = \frac{2}{n} \sum_{k=2}^{n-1} T(k) + bn + \frac{2}{n} T(1) \]
\[ = \frac{2}{n} \sum_{k=2}^{n-1} T(k) + bn + \frac{2}{n} b \]
\[ \leq \frac{2}{n} \sum_{k=2}^{n-1} T(k) + bn + b \]
\[ \leq \frac{2c}{n} \sum_{k=1}^{n-1} k \log(k) + bn + b \]
\[ = \frac{2c}{n} \left[ \sum_{k=1}^{n/2} k \log(k) + \sum_{k=n/2+1}^{n-1} k \log(k) \right] + bn + b \]
Continued

\[ T(n) \leq \frac{2c}{n} \left[ \sum_{k=1}^{n/2} k \log(k) + \sum_{k=n/2+1}^{n-1} k \log(k) \right] + bn + b \]

\[ \leq \frac{2c}{n} \left[ \sum_{k=1}^{n/2} k \log(n/2) + \sum_{k=n/2+1}^{n-1} k \log(n) \right] + bn + b \]

\[ = \frac{2c}{n} \left[ \sum_{k=1}^{n/2} k \log(n) - n^2/8 - n/4 + \sum_{k=n/2+1}^{n-1} k \log(n) \right] + bn + b \]

\[ = \frac{2c}{n} \left[ \left( \frac{n^2}{2} - \frac{n}{2} \right) \log(n) - \frac{n^2}{8} - \frac{n}{4} \right] + bn + b \]

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

\[ \leq cn \log(n) - cn/4 - c/2 + bn + b \]

\[ \leq c \cdot n \log(n) \]

Set \( c \geq 4b \)
Conclusion

- Although there are cases where we need $O(n^2)$ comparisons, these are so rare in the set of all possible permutations that we do not need more than $O(n \cdot \log(n))$ comparisons on average.
- In other words: If we average over the runtimes of Quick Sort over many (all) different orders of $n$ values (for different $n$), then this average will grow with $n \cdot \log(n)$, not with $n^2$.
- One can show the same for the number of swaps.
- Quick Sort is a fast general-purpose sorting algorithm.
Content of this Lecture

• Merge Sort
• Quick Sort
  – Algorithm
  – Average Case Analysis
  – Improving Space Complexity
Looking at Space Again

• We were quite sloppy
• Quick Sort as described here actually 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

- In the recursive decent, always treat the smaller of the two sub-arrays first (S’ or S’”, whatever is smaller)
- This branch of the search tree can generate at most $O(\log(n))$ calls, as the smaller array always is smaller than $|S|/2$ (or it would not be the smaller one)
- Use iteration (no stack) to sort the bigger array afterwards
- Space complexity: $O(\log(n))$
Implementation

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

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

Ulf Leser: Algorithms and Data Structures, Summer Semester 2017
Implementation

- **14-20:** Choose the smaller and sort it recursively
  - Note: **Only one call** is made for each division
- **We adjust l/r and sort the larger sub-array directly**
  - New loop (6-21) applies the same procedure performing the next sort
- **We turned a linear tail recursion into an iteration** (without stack)

```
1. func integer qSort++(S array; l,r int) {
2.     if r≤l then
3.         return;
4.     end if;
5.     while r > l do
6.         val := S[r];
7.         i := l-1;
8.         j := r;
9.         repeat
10.        … # as before
11.        until i>=j;
12.        swap( S[i], S[r]);
13.        if (i-1-1) < (r-i-1) then
14.            qsort(S, l, i-1);
15.            l := i+1;
16.        else
17.            qSort(S, i+1, r);
18.            r := i-1;
19.        end if;
20.     end while;
21. }
```
Illustration
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 in practice?
    - $\log(n)$ usually is not a lot of space
### 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><strong>Selection Sort</strong></td>
<td>O(n²)</td>
<td>O(n²)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Insertion Sort</strong></td>
<td>O(n²)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n²)</td>
<td>O(n²)</td>
</tr>
<tr>
<td><strong>Bubble Sort</strong></td>
<td>O(n²)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n²)</td>
<td>O(n²)</td>
</tr>
<tr>
<td><strong>Merge Sort</strong></td>
<td>O(n*log(n))</td>
<td>O(n*log(n))</td>
<td>O(n*log(n))</td>
<td>O(n)</td>
<td>O(n*log(n))</td>
</tr>
<tr>
<td><strong>QuickSort</strong></td>
<td>O(n²)</td>
<td>O(n*log(n))</td>
<td>O(n*log(n))</td>
<td>O(log(n))</td>
<td>O(n²) / O(n*log(n))</td>
</tr>
</tbody>
</table>
Exemplary Questions

- Proof that any sort algorithm using only value comparisons needs $\Omega(n \cdot \log(n))$ comparisons in worst case.
- Proof or refute: For every $n$, there exists a list with $n$ elements which is a best case for quick sort (choosing first element as pivot) and for bubble sort.
- Give pseudo code for QuickSort with $O(\log(n))$ additional space.
- Imagine your main memory can use only $n/16$ values. Recall that access disk is much more expensive than accessing memory. Which of the sorting algorithms can be used to keep disk I/O low? Describe the algorithm in pseudo code and argue about the number of blocks read.