Efficient Algorithms and Problem Complexity
– More about Sorting and Selection –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. The Complexity of Comparison-Based Sorting

2. Sorting Integers

3. Selection
Comparison-based sorting

The question

- How quickly can we sort by the most efficient algorithm possible?
- Actually, this is a question regarding problem complexity.
  
  \[ \text{... but it fits too well at this point.} \]
- To answer the question, we first need to state our assumptions.

The setting: comparison-based sorting

- The input is an array of keys (or of records containing keys).
- Except assignment, the only operation on keys is comparison \( a \leq b \).
- In particular, there is no arithmetic or the like.
The upper bound

What is a reasonable upper bound on the complexity of comparison-based sorting?

To prove upper bounds, the by far most common method is to devise a concrete algorithm that solves the problem within this bound.

We already know algorithms for comparison-based sorting (e.g., Mergesort) and how they behave in the worst case!

⇒ An upper bound on the worst-case time complexity of sorting is $O(n \log n)$. 
Towards an exact bound

But couldn’t there be a better algorithm than the known ones?
If we can also establish the lower bound $\Omega(n \log n)$, we know that
the complexity is $\Theta(n \log n)$.

Proving interesting upper bounds is often hard enough, but
proving lower bounds is really tough! (Cf. the $P=NP$ question.)

Why is it so difficult?
We have to reason about all programs that could possibly solve
the problem. Thus, we must analyze such an algorithm without
assuming anything but its correctness.
Commonly used strategy for proving lower bounds

Commonly used strategy:

- Consider some (unknown) algorithm.
- Assume that it correctly solves the problem.
- Show that this algorithm exhibits a running time of $T(n) = \Omega(f(n))$.
- Ideally, $f(n)$ is a known upper bound.
- Usually, the proof that $T(n) = \Omega(f(n))$ uses counting arguments.

For given $n$, count how many different “situations” the algorithm has to distinguish between. Use this to argue that there must exist an input that results in the claimed running time.
What does this mean for comparison-based sorting?

We analyze what can happen when a comparison-based sorting algorithm \( \mathcal{A} \) gets an input \( a_1 \cdots a_n \) of length \( n \).

- A run of \( \mathcal{A} \) is independent of \( a_1, \ldots, a_n \) except for comparisons
  \( \Rightarrow \) it branches only when a comparison \( a_i \leq a_j \) is made.
- Each of the two resulting branches continues until another comparison is made, and so on
  \( \Rightarrow \) we get a binary decision tree \( D(n) \).
- \( D(n) \) represents the set of all runs of \( \mathcal{A} \) on inputs of size \( n \).
- Every leaf corresponds to a sorting \( a_{i_1} \cdots a_{i_n} \) of \( a_1 \cdots a_n \).
The height of the decision tree

What does the decision tree $D(n)$ tell us?

The height $h(n)$ of $D(n)$ is a lower bound on the worst-case running time of the algorithm.

⇒ We must establish a lower bound on $h(n)$.

Lower bound on $h(n)$:

There are exactly $n!$ different possible outcomes $a_{i_1} \cdots a_{i_n}$.

As we saw, each leaf of $D(n)$ corresponds to a unique outcome.

⇒ $D(n)$ has at least $n!$ leaves. [Why “at least”?]

⇒ $h(n) = \Omega(\log(n!)) = \Omega(n \log n)$. 

Conclusion

Lower bound for comparison-based sorting
The worst-case time complexity of comparison-based sorting is $\Theta(n \log n)$. 
Counting sort

Task: Sort an array $a[1 \cdots n]$ with integer keys in the range $1, \ldots, m$

Algorithm with auxiliary array $c$ and output array $b$:

1. Compute $c[1 \cdots m]$ where $c[k] = |\{i \mid a[i] = k\}|$. (\(O(m + n)\) steps)
2. Scan $c$ to replace every $c[k]$ with $c[k] = |\{i \mid a[i] \leq k\}|$. (\(O(m)\) steps)
3. Scan $a$ backwards to put each $a[i]$ into $b[c[a[i]]-\]$.[1]. (\(O(n)\) steps)

Notes & questions:

- Runs in time \(O(n)\) if $m$ is a constant.
- Wastes a lot of space if $m$ is much larger than $n$.
- Why not just scan $c$ in step 2 to put $c[k]$ ks into $b$ for $k = 1, \ldots, m$?
- Why scan $a$ backwards in step 3?
Radix sort

**Task:** As before, but for large $m$.

\[
\text{Radixsort}(a[1, \ldots, n]) \text{ where } a[i] \in \{0, \ldots, m\} \text{ for } 1 \leq i \leq m \\
\text{for } i = 0, \ldots, \lfloor \log m \rfloor \text{ do} \\
\quad \text{CountingSort}(a) \text{ using bit } i \text{ as the key (least significant first)}
\]

**Notes:**
- Runs in time $O(n \log m)$.
- If $m$ is constant, this becomes $O(n)$ (with a small constant factor).
- Important that sorting starts with the least significant bit and is stable.
Selection

Task: For \( a[1 \cdots n] \) and \( k \in \{1, \ldots, n\} \), return the \( k \)th smallest item in \( a \).

- We can sort \( a \) and then return \( a[k] \) in time \( O(n \log n) \), but can we do better?
- Idea: Use partitioning as in quicksort; recurse into the correct half.
  \( \Rightarrow \) Worst case is \( O(n^2) \) as in quicksort (ironically for sorted arrays).
- What if we use the random partitioning from random quicksort?
Bounding the expected running time $T(n)$ of random select

- Partitioning w.r.t. a random element takes $\leq n$ steps.
- Hence, as all partitioning elements are equally likely,

\[
T(n) \leq n + \frac{1}{n} \sum_{p=1}^{n} \max\{T(p-1), T(n-p)\}
\leq n + \frac{2}{n} \sum_{p=1}^{m} T(n-p) \quad \text{where } m = \lceil n/2 \rceil
\]

- Assume w.l.o.g. that $T(1) = 1$. Induction on $n$ yields $T(n) \leq 4n$:

\[
T(n) \leq n + \frac{2}{n} \sum_{p=1}^{m} T(n-p)
\leq n + \frac{2}{n} \sum_{p=1}^{m} 4(n-p)
= n + 8m - \frac{4m(m+1)}{n}
\leq 4n \quad \text{[check by case distinction]}
\]
The expected running time $T(n)$ of random select

Conclusion
The expected running time of the selection algorithm is $O(n)$ when using random partitioning.
Until the next lecture, please read about sorting and selection in the textbooks. In particular, read about Quicksort (which will not explicitly be covered in the lectures).