CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #6

▶Randomisation & Lower Bounds

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading: Chapters 7.4 and 8.1

Aims of this lecture

- To show how **randomness** can be used in the design of efficient algorithms.
- Glimpse into the **analysis of randomised algorithms**.
- To discuss the class of **comparison sorts**: sorting algorithms that sort by comparing elements.
- To show a general **lower bound** for the running time of a class of sorting algorithms.

A Randomised Version of QuickSort

- Choosing the right pivot element can be tricky – we have no idea *a priori* which pivot elements are good.
- **Solution**: **leave it to chance!**

RANDOMISED-PARTITION(A, p, r)

1: i = RANDOM(p, r)
2: exchange $A[r]$ with $A[i]$
3: **return** PARTITION(A, p, r)

"Random" picks pivot uniformly at random among all elements.

RANDOMISED-QUICKSORT(A, p, r)

1: **if** $p < r$ **then**
2: q = RANDOMISED-PARTITION(A, p, r)
3: RANDOMISED-QUICKSORT$(A, p, q-1)$
4: RANDOMISED-QUICKSORT$(A, q+1, r)$

Performance of Randomised-QuickSort

- Assume in the following that all elements are distinct.
- What is a worst-case input for Randomised QuickSort?
- **Answer**: **there is no worst case for Randomised QuickSort!**
- Reason: all inputs lead to the **same runtime behaviour**.
  - The i-th smallest element is chosen with uniform probability.
  - Every split is equally likely, regardless of the input.
  - The runtime is random, but the **random process (probability distribution) is the same** for every input.
- Randomness levels the playing field for all inputs.
  - No one can provide a worst-case input for Randomised-QS.

Runtime of Randomised Algorithms

- For randomised algorithms (in contrast to **deterministic algorithms**) we consider the **expected running time** $E\big(T(n)\big)$.

- **Expectation** of a random variable X:

$$\mathrm{E}(X) = \sum_x x \cdot \Pr(X = x)$$

- **Example**: for X = roll of fair 6-sided die,

$$\mathrm{E}(X) = \sum_x x \cdot \Pr(X = x) = \sum_{x=1}^{6} x \cdot \frac{1}{6} = \frac{21}{6} = 3.5$$

- **Example** ($X \in \{0, 1\}$): expected #times a coin toss shows heads,

$$\mathrm{E}(X) = \sum_x x \cdot \Pr(X = x) = 0 \cdot \Pr(\text{tails}) + 1 \cdot \Pr(\text{heads}) = \Pr(\text{heads}).$$

Linearity of Expectation

- Linearity of expectation:

$$\mathrm{E}(X_1 + X_2) = \mathrm{E}(X_1) + \mathrm{E}(X_2)$$

- Expected number of times 100 coin tosses come up heads:

$$\mathrm{E}(X_1 + \cdots + X_{100}) = \mathrm{E}(X_1) + \cdots + \mathrm{E}(X_{100}) = 100 \cdot \Pr(\mathrm{heads})$$

  - Note: for 0/1-variables the expectation boils down to probabilities.

➢ Number of Comparisons vs. Runtime (1)

For analysing sorting algorithms the **number of comparisons** of elements made is an interesting quantity:

- For QuickSort and other algorithms it can be used as a proxy or substitute for the overall running time (see next slide).
  - Analysing the number of comparisons might be easier than analysing the number of elementary operations.
- **Comparisons can be costly** if the keys to be compared are not numbers, but more complex objects (Strings, Arrays, etc.)
- Algorithms making fewer comparisons might be preferable, even if the overall runtime is the same.
- There is a **lower bound** for the running time of all sorting algorithms that rely on comparisons only.

Number of Comparisons vs. Runtime (2)

- Let $X = X(n)$ be the **number of comparisons** of elements made by QuickSort.
- Comparisons are elementary operations, hence $X(n) \leq T(n)$.
- For each comparison QuickSort only makes $O(1)$ other operations in the for loop.
- Other operations sum to $O(1)$.
- So $$X(n) \leq T(n) = O(X(n))$$ and thus $$T(n) = \Theta(X(n))$$
- To show: $\mathrm{E}[X(n)] = O(n \log n)$

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Conclusion: we can analyse the **number of comparisons** as a substitute for the runtime in the RAM model.

Expected Time for Randomised-QuickSort

- **Theorem**: the **expected number of comparisons** of Randomised-QuickSort is $O(n \log n)$ for every input where all elements are distinct.
- Proof outline:
  1. Show that here the expectation boils down to probabilities of comparing elements.
  2. Work out the probability of comparing elements.
  3. Putting 1. and 2. together + some maths.
- Follows Section 7.4.2 in the book.

1. Expectation Boils Down to Probabilities

- For ease of analysis, rename array elements to $z_1, z_2, \dots, z_n$ with $z_1 < z_2 < \dots < z_n$ (hence z_i is the *i*-th smallest element)
- **Observation**: **each pair** of elements is **compared at most once**.
  - Reason: elements are only compared against the pivot, and after Partition ends the pivot is never touched again.
- Let $X_{i,j}$ be the number of times z_i and z_j are compared:

$$X_{i,j} := \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \\ 0 & \text{otherwise} \end{cases}$$

- Then the total number of comparisons is $$X := \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$
- Taking expectations on both sides and using linearity of expectations:

$$E(X) = E\left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}\right) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{i,j}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j)$$

➢ 2. Probability of comparing z_i and z_j

- When is z_i (*i*-th smallest) compared against z_j (*j*-th smallest)?
  - If pivot is $x < z_i$ or $z_j < x$ then the decision whether to compare z_i, z_j is **postponed** to a recursive call.
  - If pivot is $x = z_i$ or $x = z_j$ then z_i, z_j **are compared**.
  - If pivot is $z_i < x < z_j$ then z_i and z_j become separated and are **never compared**!
- A decision is only made if $z_i \leq x \leq z_j$. So z_i and z_j are only compared if the **first** pivot chosen amongst $z_i \leq x \leq z_j$ is either z_i or z_j !!
- These are $j - i + 1$ values, out of which 2 lead to z_i, z_j being compared.
- As the pivot element is chosen uniformly at random,

$$\Pr(z_i \text{ is compared to } z_j) = \frac{2}{j-i+1}$$

- Note: similar numbers are more likely to be compared than dissimilar ones.

3. Putting things together

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

- Substituting $k := j - i$ yields

$$E(X) = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \le 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} \le 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} = 2n \sum_{k=1}^{n} \frac{1}{k}$$

- The sum $\displaystyle\sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

  is called **harmonic sum** and is bounded by $\displaystyle\sum_{k=1}^{n} \frac{1}{k} \le (\ln n) + 1$

- So we get $\displaystyle E(X) \le 2n \sum_{k=1}^{n} \frac{1}{k} = O(n \log n)$

Random Input vs. Randomised Algorithm

- QuickSort is efficient if
  1. The input is random or
  2. The pivot element is chosen randomly
- We have no control over 1., but we can make 2. happen.
- **(Deterministic) QuickSort**
  - **Pro**: the runtime is deterministic for each input
  - **Con**: may be inefficient on some inputs
- **Randomised QuickSort**
  - **Pro**: same behaviour on all inputs
  - **Con**: runtime is random, running it twice gives different times

➢ Other Applications of Randomisation

- **Random sampling**
  - Great for big data
  - Sample likely reflects properties of the set it is taken from
- **Symmetry breaking**
  - Vital for many distributed algorithms
- **Randomised search heuristics**
  - General-purpose optimisers, great for complex problems
    - Evolutionary Algorithms / Genetic Algorithms
    - Simulated Annealing
    - Swarm Intelligence
    - Artificial Immune Systems

Summary

- QuickSort has a bad worst-case runtime of $\Theta(n^2)$, but is fast on average.
  - Average-case performance on **random inputs** is $O(n \log n)$.
  - **Randomised QuickSort** sorts any input in **expected time** $O(n \log n)$.
  - Constants hidden in the asymptotic terms are small.
- QuickSort is used in modern programming languages
- **Randomness** can eliminate worst-case scenarios:
  - For randomised QuickSort all inputs are treated the same.
  - The running time is random and can be quantified by considering the **expected running time**: $O(n \log n)$.

Comparison Sorts

- InsertionSort
- SelectionSort
- MergeSort
- HeapSort
- QuickSort

- All these proceeded by comparing elements – we call these **comparison sorts**.
- Sometimes comparisons are the only information available:
  - Multi-dimensional data with no total ordering
    (e. g. sorting cars according to speed and price)

Performance of Comparison Sorts

- The best comparison sorts we have seen so far take time $\Omega(n \log n)$ in the worst case.
- Can we do better?
- Or can we prove that **it's impossible to do better**?
  - Would give us piece of mind (and our boss/customer, …)
  - Prevents us from wasting time.

▶Complexity Theory
(very briefly, more in CS-338 Theory of Computation)

- Complexity theory deals with the **difficulty of problems**.
- **Limits to the efficiency of algorithms**
  - Results like: *every algorithm needs at least time X in the worst case to solve problem Y*.
  - Stops us from **wasting time trying to achieve the impossible**!
  - Informs the design of efficient algorithms.
- Two sides of a coin:

Complexity theory ←→ Efficient algorithms

Appetiser: NP-Completeness in a Nutshell

(not relevant for the assessment, but relevant for Computer Science)

- Entscheidungsproblem (decision problem), answer yes/no?
  - **Example**: does there exist an assignment of variables that satisfies a Boolean formula? E.g. $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_4 \vee \overline{x_5}) \wedge \cdots$
- NP-complete problems (intuitively, more formal in CS-338)
  - >3000 important problems in different shapes: satisfiability, scheduling, selecting, cutting, routing, packing, colouring, …
  - It is **easy to verify** that a given solution means "yes".
  - No one knows how to **find** a solution in polynomial worst-case time!
  - **Either no** NP-complete problem is solvable in polynomial time, **or all of them** are. No one knows! → **"P versus NP problem"**
  - **$1,000,000 reward** for an answer (let me know if you crack it :-).

How (Not) to Show Lower Bounds

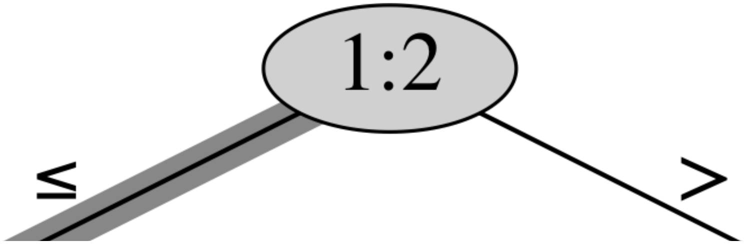
- How can we show that time $\Theta(\dots)$ is best possible?
- *“We didn’t manage to find a better algorithm.”*
- *“No one in the world has found a better algorithm.”*
  - What if tomorrow someone does?
  - We have to find arguments that apply to **all algorithms that can ever be invented**.
- *“Surely, every efficient algorithm must do things this way.”*
  - You’d be surprised. Efficient algorithms for multiplying matrices start by subtracting elements!

Comparison Sorts as Decision Trees

- There is one thing that all comparison sorts have to do: **compare elements**!
- Let's strip away all the overhead, data movement, looping, recursing, etc. and take the number of comparisons as lower time bound.
- W.l.o.g. we assume that elements $a_1, \ldots, a_n$ are **distinct** – then we can assume that all comparisons have the form $a_i < a_j$.
- A **decision tree** reflects all comparisons **a particular comparison sort** makes, and how the outcome of one comparison determines future comparisons.
  - Like a skeleton of a sorting algorithm.

Decision tree for a comparison sort

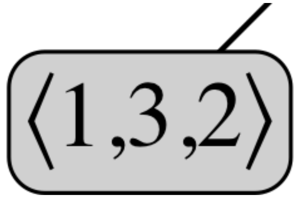
- Inner node *i:j* means comparing a_i and a_j.



- Leaves: ordering $\pi_1, \pi_2, \ldots, \pi_n$ established by the algorithm:

$$a_{\pi_1} \leq a_{\pi_2} \leq \cdots \leq a_{\pi_n}$$

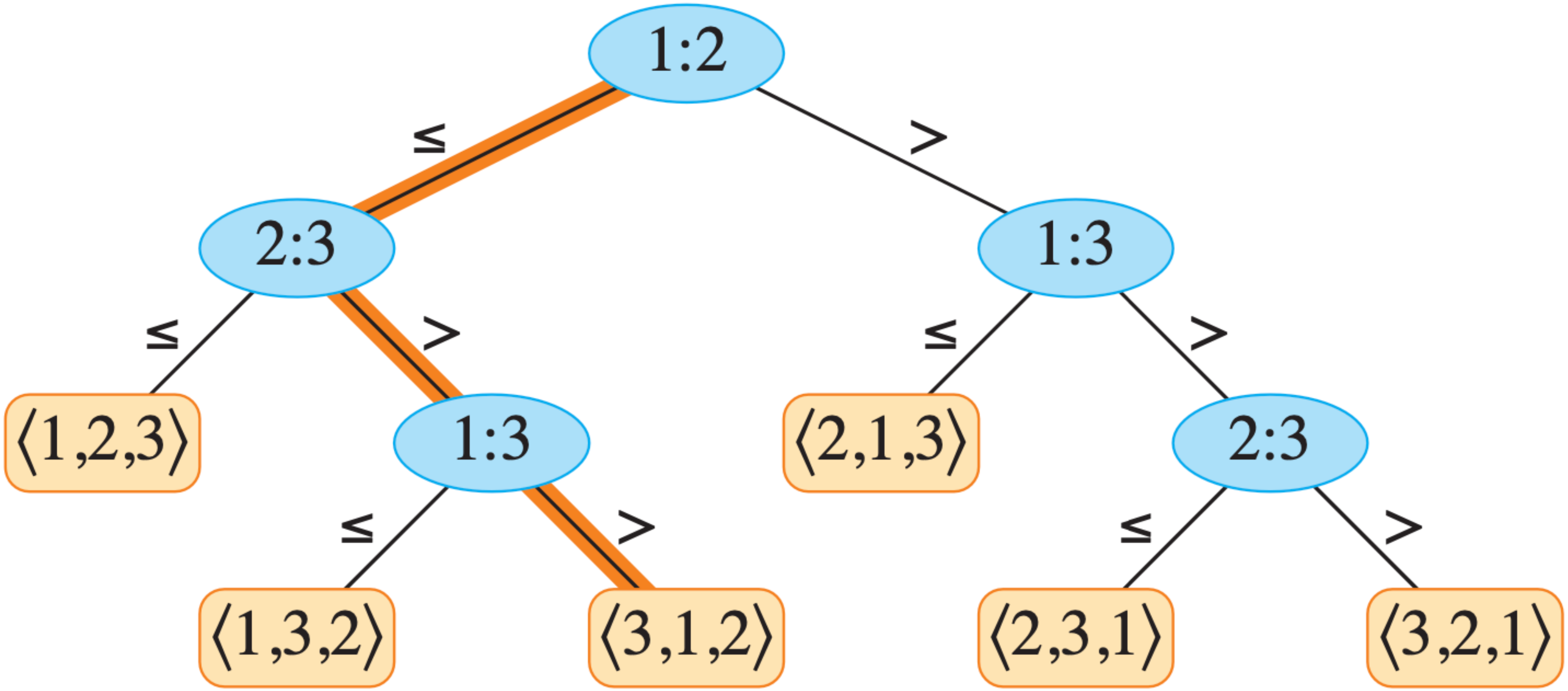
A leaf contains a sorted output for a particular input.



$$a_1 \leq a_3 \leq a_2$$

- **The execution of a sorting algorithm corresponds to tracing a simple path from the root down to a leaf.**

Example of a decision tree



Lower bound for comparison sorts

Theorem: **Every comparison sort** requires $\boldsymbol{\Omega(n \log n)}$ comparisons in the worst case.

- This includes all comparison sorts that will ever be invented!
- Proof follows; see Theorem 8.1 in the book.
- The theorem can be extended towards an $\Omega(n \log n)$ bound for the **average-case time** (not done here).
- The theorem implies that HeapSort and MergeSort have worst-case time $\Omega(n \log n)$. They are asymptotically **optimal comparison sorts**.

Proof of the lower bound (1)

- The **worst-case number of comparisons** equals the **length of the longest simple path** from the root to any reachable leaf: we call this the **height h** of the tree (as in HeapSort).
- Every correct algorithm must be able to produce a sorted output for each of the $n!$ possible orderings of the input.
  - => the leaves of the decision tree must be at least $n!$
- A binary tree of height h has no more than 2^h leaves.
  - We'll prove this formally in a bit; let's take this for granted for now.
- To accommodate $n!$ leaves we need $2^h \geq n!$.
- Taking logarithms, this is equivalent to $h \geq \log(n!)$.
- So the worst-case number of comparisons is at least log(n!).

What is log(n!)? Proof (2)

- Using $n! \geq \left(\frac{n}{e}\right)^n$ (for *e = exp(1) = 2.71…*) we get

$$
\begin{aligned}
\log(n!) &\geq \log\left(\left(\frac{n}{e}\right)^n\right) \\
&= n\log(n/e) && (\log(x^y) = y\log(x)) \\
&= n(\log(n) - \log(e)) && (\log(x/y) = \log(x) - \log(y)) \\
&\geq n\log(n) - 1.4427n \\
&= \Omega(n\log n)
\end{aligned}
$$

- The worst-case number of comparisons is $\Omega(n \log n)$.
- NB for the curious: an average-case bound follows in similar ways as most leaves have to hang at depths of $\Omega(n \log n)$.

Summary

- **Complexity Theory** gives limits to the efficiency of algorithms.
  - How (not) to prove lower bounds for all algorithms.
- All comparison sorts need time $\Omega(n \log n)$ in the worst case.
  - Decision trees capture the behaviour of every comparison sort.