Lecture 4: Divide-and-conquer II

At this lecture we look at some uses of the divide-and-conquer technique in the design of sorting and searching algorithms.
4.1 Divide-and-conquer sorting

Recall once again the divide-and-conquer mergesort algorithm:

```
1 function MERGESORT (A[1 ... n])
2 if n = 1 then return
3 else
4     Introduce auxiliary arrays A'[1 ... ⌊n/2⌋], A''[1 ... ⌈n/2⌉]
5     A' ← A[1 ... ⌊n/2⌋]
6     A'' ← A[⌈n/2⌉ + 1 ... n]
7     MERGESORT(A')
8     MERGESORT(A'')
9     MERGE(A', A'', A)
10 end
```
Quicksort

While mergesort is an excellent algorithm, in practice the quicksort method (T. Hoare 1959) runs significantly faster, and is the method of choice in software libraries.

Algorithm 1: The quicksort algorithm

1. **function** QUICKSORT (A, i, j)
   
   **Input:** Integer array $A[1 \ldots n]$, array indices $1 \leq i \leq j \leq n$.
   
   **Output:** Array $A$, with interval $A[i \ldots j]$ in increasing order.

2. If $i = j$, there is nothing to do and the algorithm terminates.
3. Otherwise select some **pivot element** $v$ in interval $A[i \ldots j]$.
4. Rearrange (“partition”) $A[i \ldots j]$ so that for some $k$:
   - elements $< v$ are moved to interval $A[i \ldots k - 1]$
   - elements $\geq v$ are moved to interval $A[k \ldots j]$

Sort recursively $\text{QUICKSORT}(A, i, k - 1); \text{QUICKSORT}(A, k, j)$. 
A Python implementation of quicksort:

```python
def quicksort(A, i, j):
    if i < j:
        v = pivot(A, i, j)
        k = partition(A, i, j, v)
        quicksort(A, i, k-1)
        quicksort(A, k, j)

def pivot(A, i, j):
    # randomised pivoting
    p = random.randint(i, j)
    return A[p]
```

def partition(A, i, j, v):
    l, r = i, j
    while l < r:
        while (A[l] <= v) and (l < r):
            l = l + 1
        while (A[r] > v) and (l < r):
            r = r - 1
        if l < r:
    return l
Complexity of quicksort

- When the pivot elements are selected uniformly at random, as in the implementation presented here, quicksort is a randomised algorithm.\(^1\)
- The runtime depends on the success of the pivoting: typically a randomly chosen pivot partitions its interval in roughly equal-size halves, but in the worst case the pivot falls at either end of the interval, and the algorithm makes slow progress.
- The worst-case runtime of quicksort is \(\Omega(n^2)\). However the probability of such unlucky pivoting is very low, and the algorithm behaves according to its expected runtime of \(O(n \log n)\).
- Runtime analysis of quicksort is discussed at the tutorials, and also later at Lecture 12 on Randomised Algorithms.

\(^1\)There are also deterministic pivoting rules, such as “median-of-first-three-elements”, but we shall forego discussing them.
A lower bound on sorting (1/3)

- We have now seen two $O(n \log n)$ sorting methods: mergesort even in the worst-case and quicksort in expectation. Are there yet faster techniques?

- With some conditions, the answer is NO: any “comparison-based” sorting method needs runtime $\Omega(n \log n)$ in the worst case!

- Here comparison-based means that the sorting process is based exclusively on pairwise comparisons between elements ("is $a_i < a_j$, $a_i > a_j$ or $a_i = a_j$"?) No information about the type, internal structure or encoding of the elements may be used.
A lower bound on sorting (2/3)

For a proof of this result, consider a comparison-based sorting algorithm $A$ and a “comparison tree” $T$ that describes the pairwise element comparisons $A$ makes on any input sequence of length $n$, where all the elements are distinct.
A lower bound on sorting (3/3)

- Now consider every possible input ordering (permutation) \( \pi \) to the algorithm \( A \), trace the comparisons \( A \) makes on this ordering, and label the corresponding leaf of \( T \) with \( \pi \).

- Then every permutation \( \pi \) of \( n \) elements appears on some leaf of \( T \), and no leaf contains two permutations. (Because otherwise \( A \) would sort two permutations \( \pi \neq \pi' \) in exactly the same way, which would necessarily be wrong for at least one of them.) There are \( n! \) permutations of \( n \) elements, so \( T \) has at least \( n! \) leaves.

- Now \( T \) is a binary tree, so if it has height (maximum path-length) \( h \), then it can have at most \( 2^h \) leaves. Conversely, since \( T \) has \( \geq n! \) leaves, \( T \) must have height \( \geq \log_2(n!) = \Theta(n \log n) \). [Tutorial 1.]

- Hence there is some input sequence on which algorithm \( A \) performs \( \Omega(n \log n) \) pairwise comparisons of elements.
Some visualisations of sorting algorithms

http://www.sorting-algorithms.com
4.2 Divide-and-conquer search

The best-known divide-and-conquer search algorithm is surely binary search.

Algorithm 2: The binary search algorithm

1 function BinSearch (A[1...n], x)

Input: A sorted integer array A[1...n] and an integer x.
Output: Either an index i for which A[i] = x or “no” if A does not contain x.

2 If n = 1 then return 1 if A[1] = x, otherwise return “no”.
3 Set m = \lfloor n/2 \rfloor.
4 If A[m] = x then return m.
5 If x < A[m] then return BinSearch(A[1...m − 1], x).
6 If x > A[m] then return m + BinSearch(A[m + 1...n], x).
Complexity of binary search

- Binary search can (and should) be implemented easily also without recursion.
- The runtime of the algorithm is described by the recurrence

\[
\begin{align*}
T(1) &= \Theta(1) \\
T(n) &= T(n/2) + \Theta(1), \quad n = 2^k, \quad k \geq 1.
\end{align*}
\]

- By the “Master Theorem”, this has solution

\[
T(n) = \Theta(\log n),
\]

as can also easily be seen directly.
Divide-and-conquer selection

Consider the problem of determining the median (or more generally the $k$th smallest element) of an unsorted set of numbers.

**Input:** Integer list $S[1 \ldots n]$ and a number $k$, $1 \leq k \leq n$.

**Output:** The $k$th smallest element in $S$. (E.g. for the median $k = \lceil n/2 \rceil$.)

Simple solutions:

1. Determine the $k$ smallest elements in $S$ one by one:
   $$T(n) = \Theta(k \cdot n) = \Theta(n^2).$$

2. Sort $S$ and pick the $k$th item in the sorted list:
   $$T(n) = \Theta(n \log n).$$
Quickselect
An application of the quicksort idea to the selection problem.

**Algorithm 3**: The quickselect algorithm

1. **function** `QUICKSELECT` *(S[1...n], k)*

2. Select some pivot element `v` in `S`.

3. Partition `S` into three sublists:
   - `S_L = [x ∈ S | x < v]`
   - `S_v = [x ∈ S | x = v]`
   - `S_R = [x ∈ S | x > v]`

   If `k ≤ |S_L|` then return `QUICKSELECT(S_L, k)`.
   If `|S_L| < k ≤ |S_L| + |S_v|` then return `v`.
   If `k > |S_L| + |S_v|` then return `QUICKSELECT(S_R, k − |S_L| − |S_v|)`.

---

*a*This can be done similarly *in situ* within `S` as in quicksort.
Complexity of quickselect

- Similarly as quicksort, also quickselect is a randomised algorithm. The actual runtime depends on the success of pivoting.
- Worst-case runtime $T(n) = \Omega(n^2)$.
- Expected runtime $T(n) = O(n)$.
  - Heuristically, a random pivot falls “typically” in the middle of the elements in $S$, so that sets $S_L$ and $S_R$ are both “about” size $O(n/2)$.
  - By this heuristic, $T(n)$ would be described by recurrence
    
    $$T(n) \approx T(n/2) + O(n),$$

    which by the Master Theorem has solution $T(n) = O(n)$.
  - For a more careful argument, see textbook.
- By some clever design, it is possible to “derandomise” quickselect, so that the pivot element is guaranteed to be roughly in the middle among the elements in $S$. (Blum, Pratt, Tarjan, Floyd, Rivest 1973)
Deterministic quickselect

1. function DQUICKSELECT \((S[1 \ldots n], k)\)

2. if \(|S| \leq 50\) then sort \(S\) and return the \(k\)th smallest element in \(S\).
3. Otherwise partition \(S\) into groups of 5 elements (plus one possible leftover group).
4. Determine the median within each group.
5. Let \(M\) be the list of group medians.
6. Compute the median of \(M\) by \(v = DQUICKSELECT(M, |M|/2)\).
7. Partition \(S\) into:
   - \(S_L = [x \in S | x < v]\)
   - \(S_v = [x \in S | x = v]\)
   - \(S_R = [x \in S | x > v]\)

   If \(k \leq |S_L|\) then return DQUICKSELECT\((S_L, k)\).
   If \(|S_L| < k \leq |S_L| + |S_R|\) then return \(v\).
   If \(k > |S_L| + |S_R|\) then return DQUICKSELECT\((S_R, k - |S_L| - |S_v|)\).
Analysis of deterministic quickselect (1/2)

How uneven can the partition between $S_L$ and $S_R$ be now?

$|\{x \in S \mid x \leq v\}|, |\{x \in S \mid x \geq v\}| \geq \lceil n/4 \rceil$

$\Rightarrow |S_R|, |S_L| \leq \lfloor 3n/4 \rfloor$
Analysis of deterministic quickselect (2/2)

This leads to a recurrence:

\[
\begin{align*}
T(n) &= \mathcal{O}(1), \quad n \leq 50 \\
T(n) &= T(\lceil \frac{n}{5} \rceil) + T(\lfloor \frac{3n}{4} \rfloor) + \mathcal{O}(n), \quad n > 50
\end{align*}
\]

which has a solution \( T(n) = \mathcal{O}(n) \).

Additional notes:

1. The constant factors hidden by the \( \mathcal{O} \)-notation are now relatively large. It is known that the algorithm uses \( 18n + \mathcal{O}(1) \) comparisons for a list of size \( n \). Using groups of size 9 instead of size 5 would reduce this to \( 14\frac{1}{3} \cdot n + \mathcal{O}(1) \) comparisons.

2. One can in principle use deterministic quickselect to derandomise quicksort, by finding guaranteed good pivot elements in linear time. However, because of the constant factors, quicksort would then lose its competitive edge against e.g. mergesort.