AD Exam 2019
Divide and Conquer

Noah Stonall [mjh831]

March 2019

Disposition

Feedback fra Lars

Perfekt. Dette er bogstaveligt talt præcist sådan, jeg ville lave en disposition. Den viser klart, hvad du går igennem, uden at være for detaljeret. Strukturen har et naturligt flow, og den går i dybden med alt det, der er kød på. Hvis det er svært at holde sig til tiden, er det helt okay (og nogle gange anbefalet) at redegøre uformelt for korrektheden i stedet for bevise det formelt med løkkeinvarianter, som bogen gør. Men hvis du foretrækker det formelle, så gør du det.

- Divide-and-conquer
  - Divide, conquer, combine
  - Large problem $\rightarrow$ small chuncks $\rightarrow$ combine solutions
- Algorithm: Mergesort
  - Demonstration (by manually running the algorithm on the board)
  - Runtime $\Theta(n \log n)$ - recursion tree
    - $T(n) = 2T(n/2) + \Theta(n)$
  - Correctness - Proof by loop invariant

Answer

Divide and Conquer

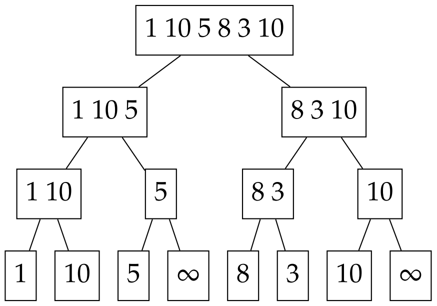
The problem is **divided** into smaller subproblems. If the subproblems are large enough to be solved recursively, we call that the *recursive case*. Once the problem is too small, the recursion *bottoms out* and we are at the *base case*. We can no longer recurse, and we solve the problem. Sometimes the subproblems looks different than the original problem on top of having to solve smaller instances of the same problem. This is the **conquer** step. Lastly we **combine** the subsolutions and the original prolem is solved.

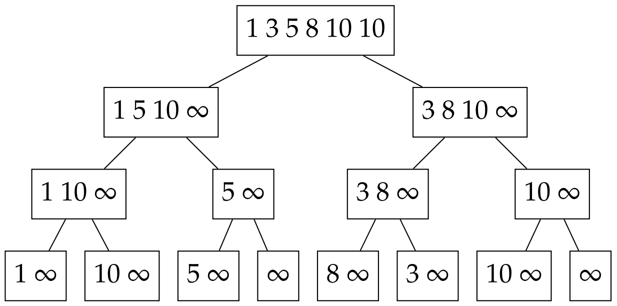
Mergesort

Divide: divides the sequence of size n into two subsequence of size $n/2$.
Conquer: Sort the subsequences recursively until the sequences have size 1, in which case the subsequence is sorted.
Combine: We merge the two sorted subsequences, making sure that they appear in the correct order.

The key operation in `Mergesort` is the combine step, where `Merge` is called, since `Merge` merges two already sorted subsequences, and ensures that the resulting merged sequence is sorted as well.

Example





∞ is not in the final sequence since the last `for`-loop in `Merge` only iterates for every element in the final sequence. That is also why there is only one ∞ symbol for each subsequence.

Runtime $\Theta(n \log n)$

The runtime of a **Divide-and-Conquer** algorithm is calculated as such:

$$T(n) = \begin{cases} \Theta(n) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

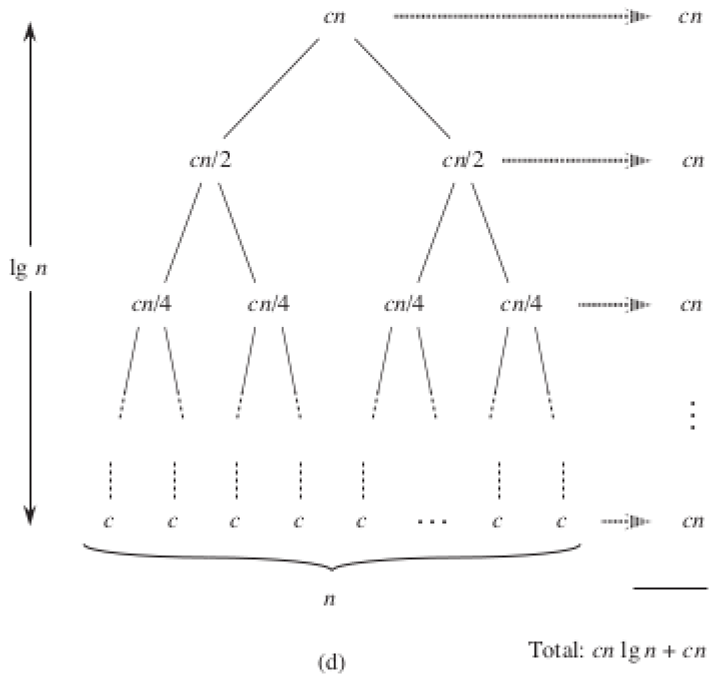
`Merge` runs in $\Theta(n)$ time since at most n basic steps are performed. `Mergesort` find **divides** the problem by finding the middle of the array. This happens in constant time. $D(n) = \Theta(1)$.

Since the algorithm is **conquered** there are two recursive calls, each of which represent half of the original problem, we have $2T(n/2)$.

`Merge` **combines** the subsolutions. If `Merge` is called with a sequence of size n, then $C(n) = \Theta(n)$.

See recursion tree on the next page.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$



Figur 1: Recursion tree of `Mergesort`

Correctness: Proof by loop invariant

In Mergesort the **combine** step solves the problem, and thus the correctness of `Merge` will be proved.

We wish to maintain the following loop invariant:

The `for`-loop loops from $k = p$ `to` r. Before each iteration of the loop, we have the subsequence $A[p..k-1]$, which contains the $k - p$ smallest elements of the sequences `L` and `R`. Furthermore $L[i]$ and $R[j]$ are the smallest elements in the respective sequences that have not been copied into `A`.

Initialization:
Before the first iteration $k = p$ and therefore the $k - p = 0$ smallest elements have been added to $A[p..k-1]$, which then is empty. Since $i = j = 1$, then $L[i]$ and $R[j]$ are the smallest elements in their respective sequences that have not been copied into A.

Maintenance:
There are two cases; $L[i] \leq R[j]$ and $L[i] < R[j]$.

$L[i] \leq R[j]$
Before $L[i]$ is copied into A, then $A[p..k-1]$ contains the $k - p$ smallest elements. After $L[i]$ is copied into A it contains the $k - p + 1$ smallest. k and i are incremented, and the loop invariant holds before the next iteration.

$L[i] > R[j]$
Here $R[j]$ is added to the A, and k and j are incremented. The loop invariant also holds for this case.

Termination:
The loop terminates when $k = r + 1$, since r is the last element in $A[p..r]$, where $r = k - 1$. A now contains the $k - p = r - p + 1$ smallest elements of L and R. Only the largest element of both L and R have been added since the largest element of each array is a sentinel (∞). The loop invariant holds.