CS 341 Algorithms. Spring 2025. University of Waterloo

Lecture 2 : Solving Recurrence

We start with the merge sort example, and then we study some simple techniques to solve recurrence formulas that come from analysis of time complexity of algorithms.

Merge Sort

Sorting is a fundamental algorithmic task, and merge sort is a classical algorithm using the idea of divide and conquer.

This divide and conquer approach works if there is a nice way to reduce an instance of the problem to smaller instances of the same problem.

For sorting, suppose the first $\frac{n}{2}$ numbers and the last $\frac{n}{2}$ numbers are already sorted, the observation is that we can then merge these two halves easily in $O(n)$ iterations.

But how do we assume that the two halves are already sorted? The idea is to apply the same procedure (break into two halves, sort each, then merge) recursively.

So, the merge sort algorithm can be summarized as follows.

Merge sort

sort ($A[1,n]$)

if $n=1$, return.

sort ($A[1, \lceil \frac{n}{2} \rceil]$)

sort ($A[\lceil \frac{n}{2} \rceil + 1, n]$)

merge ($A[1, \lceil \frac{n}{2} \rceil]$, $A[\lceil \frac{n}{2} \rceil + 1, n]$)

The correctness of the algorithm can be proved formally by a standard induction.

We focus on analyzing the running time.

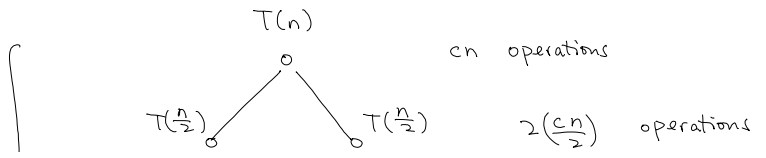
Let $T(n)$ be the time required to sort n numbers, with $T(1) = 1$.

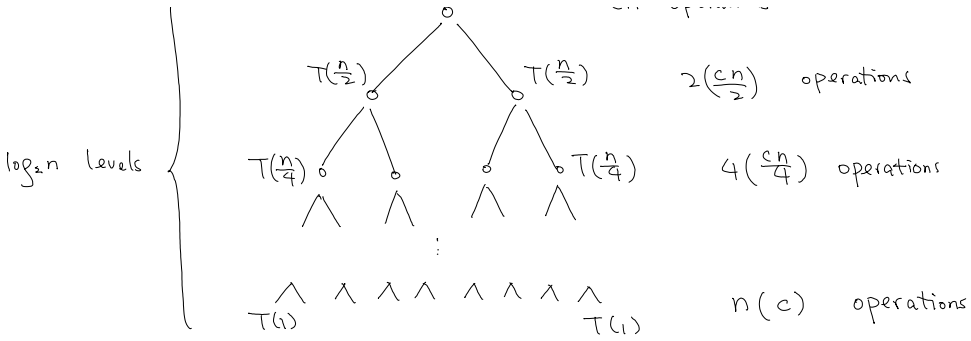
Then $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n)$.

For simplicity, we assume n is a power of two and so the relation becomes

$$T(n) = 2T(\tfrac{n}{2}) + O(n).$$

One way to solve this recurrence is to draw the recursion tree.





Each level requires cn operations, and there are $\log_2 n + 1$ levels, and so the total time complexity is $cn(\log_2 n + 1)$.

We can assume n is a power of two by at most doubling the input size, and this shows that the asymptotic complexity of merge-sort is $O(n \log n)$.

One can also prove it by induction, by using the "guess and check" method.

Induction hypothesis: $T(n) = cn \log_2 n$, where c is the constant in $O(n)$.

Induction step: $T(m) = 2T(\frac{m}{2}) + cm = 2c(\frac{m}{2})\log_2(\frac{m}{2}) + cm = cm(\log_2 m - 1) + cm = cm \log_2 m$.

Question: What is wrong with the following proof?

Induction hypothesis: $T(n) = O(n)$.

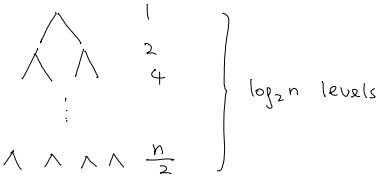
Induction step: $T(m) = 2T(\frac{m}{2}) + O(m) = 2O(\frac{m}{2}) + O(m) = O(m)$.

Exercises: Solve $T(n) = 4T(\frac{n}{2}) + n$, $T(n) = 3T(\frac{n}{2}) + n$, $T(n) = 2T(\frac{n}{2}) + n^2$, assuming n is a power of 2 and $T(1) = 1$.

Solving Recurrence [DPV 2.2]

In the following, we assume that say $T(i) \le C_2$ for $i \le C_1$ where $C_1 \ge 10$ and C_2 are absolute constants, i.e. constant size problems can be solved in constant time.

Let's try to solve $T(n) = 2T(\frac{n}{2}) + 1$, assuming $n = 2^k$ for some k.



Then we see from the picture that $T(n) \le n - 1$.

We may use the guess and check method and try $T(n) \le cn$.

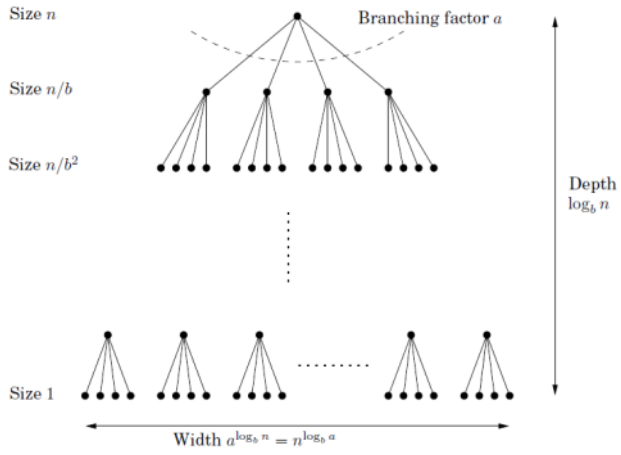
But then the induction step $T(m) = 2T(\frac{m}{2}) + 1 = 2c(\frac{m}{2}) + 1 = cm + 1$, not working.

Instead, we need to use the correct hypothesis $T(n) \le n - 1$ to make the induction works.

Let's consider a more general setting.

Consider the recurrence relation $T(n) = aT(\frac{n}{b}) + n^c$ for some constants $a > 0$, $b > 1$, $c \ge 0$.

picture from [DPV 2.2]



n^c operations

$a\left(\frac{n}{b}\right)^c$ operations

$a^2\left(\frac{n}{b^2}\right)^c$ operations

$a^{\log_b n}$ operations $= n^{\log_b a}$

If we sum the number of operations, we see that it is a geometric sequence, with the ratio $\frac{a}{b^c}$.

Now we analyze the sum based on whether the ratio is greater than 1, smaller than 1, or equal to 1.

- If $\frac{a}{b^c} = 1$, then every term is the same, and we have the sum is $n^c \cdot \log_b n$. This is what we have seen in merge sort.
- If $\frac{a}{b^c} < 1$, then it is a decreasing geometric sequence, and it is dominated by the first term, and we have the sum is $O(n^c)$, with the hidden constant depending on a, b, c (this is where we need to assume they are constant).
- If $\frac{a}{b^c} > 1$, then it is an increasing geometric sequence, and it is dominated by the last term, and we have the sum is $O(a^{\log_b n}) = O(n^{\log_b a})$.

To summarize, we have proved the result known as the master theorem.

Master Theorem If $T(n) = a\,T(\frac{n}{b}) + n^c$ for constants $a > 0$, $b > 1$, $c \geq 0$, then

$$T(n) = \begin{cases} O(n^c) & \text{if } c > \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^{\log_b a}) & \text{if } c < \log_b a . \end{cases}$$

Remark: It is more important to remember the method than the result. It is easy to derive the result back.

Also, there are scenarios that the theorem does not apply but the method still works as we will see.

More Recurrences

Single subproblem: This is common in algorithm analysis.

Examples: $T(n) = T(\frac{n}{2}) + 1$, we have $T(n) = O(\log n)$, binary search.

$T(n) = T(\frac{n}{2}) + n$, we have $T(n) = O(n)$, geometric sequence.

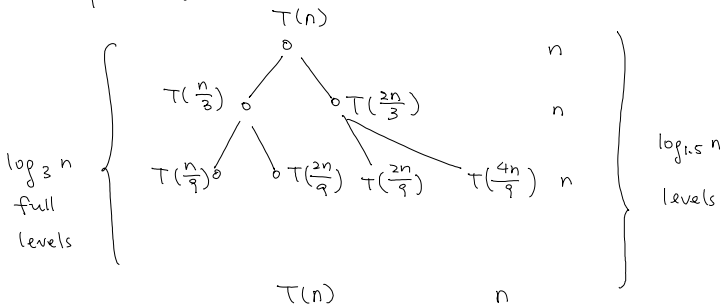
$T(n) = T(\sqrt{n}) + 1$, we have $T(n) = O(\log\log n)$, Counting levels.

(In level i, the subproblem is of size $n^{2^{-i}}$. When $i = \log\log n$, it becomes $n^{\frac{1}{\log n}} = O(1)$.)

Non-even subproblems: We will see one interesting example later.

$T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + n$

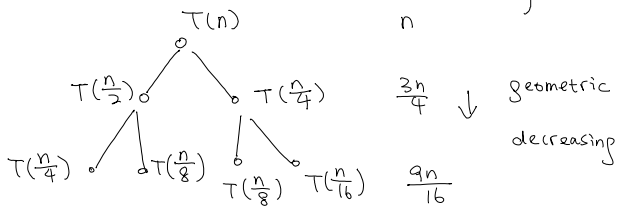
So, $T(n) = O(n \log n)$.



$T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n$

So, $T(n) = O(n)$.

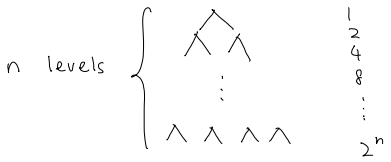
Could check by induction that $T(n) \leq 4n$.



Exponential time

$T(n) = 2T(n-1) + 1$

So, $T(n) = O(2^n)$.



Optional

Can we improve the runtime if we have $T(n) = T(n-1) + T(n-2) + 1$?

This is the same recurrence as the Fibonacci sequence.

Using the techniques that you have learnt in MATH 239 (computing roots of polynomials), it can be shown that $T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = O(1.618^n)$, faster exponential time.

This kind of recurrence shows up often in analyzing faster exponential time algorithms.

Consider the maximum independent set problem, where we are given a graph $G = (V, E)$, and our task is to find a maximum subset of vertices $S \subseteq V$ such that there are no edges between every pair of vertices $u, v \in S$.

A naive algorithm is to enumerate all subsets, and this takes $\Omega(2^n)$ time.

Now consider a simple variant.

Pick a vertex v with maximum degree.

There are two possibilities: either $v \in S$ or $v \notin S$.

In the latter case, we delete v and reduce the graph size by one.

In the former case, we choose v, and then we know that all neighbors of v cannot be chosen, and so we can delete v and all its neighbors, so that the graph size is reduced by at least two.

So, $T(n) \leq T(n-1) + T(n-2) + O(n)$, and it is strictly smaller than $O(2^n \cdot n)$.
