# Algorithms

Nir Elber

Fall 2022

## Contents



# 1 January 11

Problem set 1 will be released tonight.

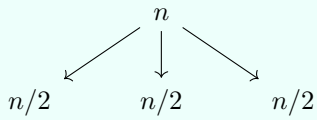
## 1.1 Recursion

Today we're going to talk about run-time for recursive algorithms. Here are some examples.

**Example 1.** Multiplication of $n$-bit integers had a run-time of $T(n)$, where

$$T(n) = 3T(n/2) + \Theta(n).$$

Diagrammatically, we can think about this as in the following tree.

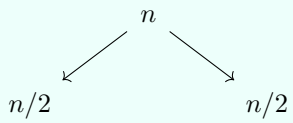



Continuing the tree down, we will have $\log_2(n)$ total layers making for a total work of

$$T(n) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} \left(\frac{3}{2}\right)^i n = n \cdot \frac{(3/2)^{1+\lfloor \log_2 n \rfloor}}{(3/2)-1} = O\left(n^{\log_2 3}\right).$$

In fact, we see $T(n) = \Theta\left(n^{\log_2 3}\right)$ by a similar argument.

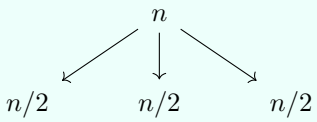
**Example 2.** Merge sort has a run-time of $T(n)$, where $T(n) = 2T(n/2) + \Theta(n)$. Here, the tree looks like the following.




Summing, the total work comes out to $T(n) = \Theta(n \log_2 n)$.

**Example 3.** Binary search has a run-time of $T(n) = T(n/2) + \Theta(1)$, which we can solve as above to give $T(n) = O(\log_2 n)$.

**Example 4.** Suppose we have a recurrence which looks like $T(n) = 3T(n/2) + \Theta\left(n^2\right)$. Here is our tree.




Here, the $k$th layer of our tree has $3^k$ nodes, each producing work $n^2/4^k$, producing a total work on the $k$th layer as $n^2(3/4)^k$. Summing the geometric series (over all layers), we see

$$T(n) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} \left(\frac{3}{4}\right)^i n^2 \leq \frac{n^2}{1 - \frac{1}{4}} = O\left(n^2\right).$$

A similar argument shows $T(n) = \Theta\left(n^2\right)$.

More generally, if we have a recursion of the form

$$T(n) = aT(n/b) + \Theta\left(n^c\right),$$

then the $k$th layer of our imagined tree has $a^k$ nodes, each of size $n/b^k$, producing a total work of $(a/b^c)^k \cdot n^c$ in this layer. At the last layer, which is the $\lfloor \log_b a \rfloor$th layer, we can solve that we do $\Theta\left(n^{\log_b a}\right)$ work. We can then sum this geometric series, in the following cases.

- If $c < \log_b a$, then our geometric series is dominated by the last term, so $T(n) = \Theta\left(n^{\log_b a}\right)$.
- If $c = \log_b a$, then our geometric series is constant, and we find $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
- If $c > \log_b a$, then our geometric series is dominated by the first term, so $T(n) = \Theta\left(n^c\right)$.

We can check that the above cases match up with our examples.

**Remark 5.** One can apply a similar analysis to a recurrence of the form $T(n) = aT(n/b) + f(n)$, where the cases depend on how $f(n)$ compares with $n^{\log_b a}$. Here are the statements.

- If $f(n) = O\left(n^c\right)$ for $c < \log_b a$, then $T(n)$ is still dominated by the last term.
- If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
- Lastly, if $f(n) = \Omega\left(n^c \log^k n\right)$ for some $c > \log_b a$, then $T(n) = \Theta\left(n^c\right)$.

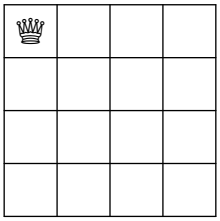
## 1.2 Recursion with Back-tracking

Here is our example problems.

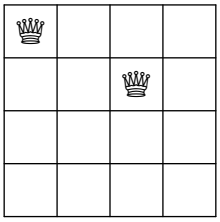
**Exercise 6.** Can we place $n$ queens on an $n \times n$ board which do not attack each other?

Note that we certainly cannot place more than $n$ queens.

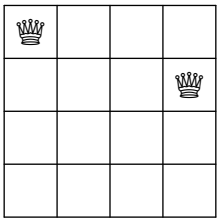
*Proof.* One approach is to place queens arbitrarily. This tends to not work on your first try. Let's say we start with $n = 4$, placing a queen in the top-left.



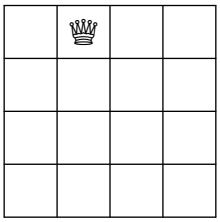
Maybe we try to place a queen in the first place that works.



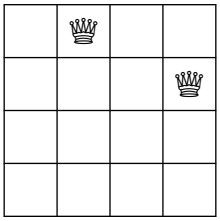
But now we're in trouble because we can't put a queen anywhere in the third row! So let's try the next spot in the second row.



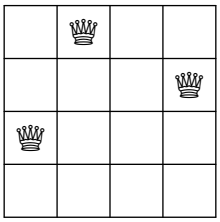
But here we're still in trouble because no spot in the third row is valid! So next we should try moving the queen in the first row, as follows.



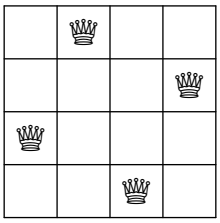
There's only place to put the queen in the second row, which as follows.



Now we see that there is one option to place the queen in the third row, as follows.



Lastly, we see that there is exactly one place we can put the queen in the last row, returning a valid queen arrangement for $n = 4$.



This process will work for general $n$ and will always terminate eventually, telling us there was no possible way to place the queens, or returning a legitimate way to place the queens.

Here is some Python code for this algorithm. We represent the position of the queens as a row of positions for columns, where the first entry goes to the first row.

```
def placeQueens(Q, n):
        if len(Q) == n:
                # we have succeeded!
                return Q
        r = len(Q)
        # let's try adding in a queen at (r,i)
        for i in range(n):
                legal = True
                for j,val in enumerate(Q):
                        # test for matching column
                        if val == i:
                                legal = False
                        # test for matching diagonal
                        if abs(val-i) == r-j:
                                legal = False
                if legal:
                        # placing in (r,i) is possible!
                        # let's try it
                        attempt = placeQueens(Q+[i], n)
                        if attempt:
                                return attempt
        # nothing worked :(
        return None
```

This program will work, but it does not run quickly because of all the back-tracking we have to do with our queens. ■

Let's see a few more examples. They all have the common theme that they more or less "just try everything" and are therefore correct but very slowly.

**Exercise 7.** We describe an algorithm which will solve chess.

*Proof.* Given a board state and whose turn it is to move, we merely have to check if it is currently checkmate. If it's not, then let's say it's white's turn to move: we now just run through all possible valid moves by white to ask if it puts black in checkmate. If black doesn't end up in checkmate, we test all of black's moves to see if it now puts white into checkmate; if black has such a move, then we know that white's move was a bad one, so white should try a different move. This algorithm can continue layering indefinitely. ■

**Remark 8.** The issue with the above algorithm is that it is incredibly slow to have to check through all possible moves in all possible positions.

**Exercise 9.** We describe an algorithm to solve "subset sum": given a finite (multi)set $X = \{x_1, \ldots, x_n\}$ of integers and a target $T$, does there exist a subset $S \subseteq X$ such that

$$\sum_{x \in S} x = T?$$

*Proof.* The idea, as usual, is to just loop through all subsets. To recurse, there are two cases: if our subset should contain $x_1$, then we want the remaining subset to sum to $T - x_1$; otherwise, we want the remaining subset to sum to $T$. In the base case, $X$ will be empty, where the problem is only possible if $T = 0$. Here is the Python code.

```
def subsetSum(X,T):
        if not X:
                # X is empty
                return T == 0
        x = X[0]
        # the case-work on x
        return subsetSum(X[1:], T) or subsetSum(X[1:], T-x)
```

There are a few ways to deal with the indexing; the above is a particular stupid way. Note that this algorithm runs through all possible subsets, so this will run in $\Theta(2^n)$. ■

**Exercise 10.** We describe an algorithm to solve "longest increasing subsequence": given a sequence of integers $\{a_i\}_{i=1}^n$, we want the largest $N$ such that there exist indices $i_1, \ldots, i_N$ such that the sequence $a_{i_1}, \ldots, a_{i_N}$ is increasing.

*Proof.* Here are a few possible ways to do recursion with back-tracking.

- We could go element-by-element and ask if we should include that element in our subsequence.
- We could ask in each position where to go next in the subsequence.

There are many more ways. ■

The goal of the next few lectures is to try to make the above naïve algorithms more efficient.