Recipe

(a) Known algorithm. If you are using a modified version of a known algorithm, you can piggyback your analysis on the complexity of the original algorithm. For example, if you use a modified version of DFS or Dijkstra's algorithm, as long as your modifications do not affect the original running time, you can use the complexity of these other algorithms.

However, you must state (and be correct) about why your changes do not affect the original algorithm's running time. For example: The algorithm performs DFS and marks each vertex with a color when it is visited. This is a constant amount of additional work per vertex and so the running time remains the same as DFS: $O(|V| + |E|)$.

(b) Master Theorem. If the time complexity equation can be written as follow, then we can use the Master Theorem.

$$T(n) = a\ T\left(\frac{n}{b}\right) + O(n^d)$$

$$T(n) = \begin{cases} \text{If } \log_b a < d, & T(n) = O(n^d) \\ \text{If } \log_b a = d, & T(n) = O(n^d \log n) \\ \text{If } \log_b a > d, & T(n) = O(n^{\log_b a}) \end{cases}$$

(c) Else, we have to evaluate $T(n)$ to be able to take the big-O out of it (see Evaluating sums and Examples). There are 3 ways of doing this:

- Guess by looking at the first terms then prove,
- Guess by unrolling the equation then prove,
- And look at the recursion tree.

Important trick. When analysing recurrences, for convenience you can[1] assume n is a power of a number by giving the following explanation:

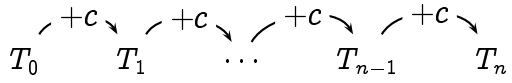
> We assume without loss of generality that n is power of [your number]. This will not influence the final bound in any important way, after all, n is at most a multiplicative factor away from some power of [your number].

[1] See the proof of Master Theorem in the book, page 49

Evaluating sums

Arithmetic progression

$$T_n = T_{n-1} + c$$



We have:

$$\begin{cases} T_n = \text{first} + c \times (\#\text{ of times } c \text{ has been added}) \\ T_0 + T_1 + \cdots + T_n = (\#\text{ of terms}) \times \dfrac{\text{first} + \text{last}}{2} \end{cases}$$

Example:

$$1 + 2 + 3 + \cdots + n = n \times \frac{1+n}{2}$$

Geometric progression

$$T_n = T_{n-1} \times c$$

We have ($c \neq 1$):

$$\begin{cases} T_n = \text{first} \times c^{(\#\text{ of times } c \text{ has been multiplied})} \\ T_0 + T_1 + \cdots + T_n = \text{first} \times \dfrac{c^{(\#\text{ of terms})} - 1}{c - 1} \end{cases}$$

Example ($c \neq 1$):

$$1 + c + c^2 + \cdots + c^n = \frac{c^{n+1} - 1}{c - 1}$$

Examples

A) Fast multiplication

Algorithm. The idea is to square $a^{\frac{n}{2}}$ to get a^n (one multiplication) and when n is odd, evaluate a^{n-1} the same way and then multiply it by a to get a^n (one multiplication).

Pseudo-code.

```
procedure mult(a, n):
  if n == 1:
    return a
  else if n is even:
    b = mult(a, n / 2)
    return b * b
  else
    return a * mult(a, n - 1)
```

Time complexity equation. Let $T(n)$ be the number of multiplications.

We assume without loss of generality that n is power of 2. This will not influence the final bound in any important way, after all, n is at most a multiplicative factor away from some power of 2.

At each step n is divided by 2.

$$\boxed{T(n)} \quad = \quad \boxed{T\left(\frac{n}{2}\right)} \quad + \quad \boxed{1}$$

Total work for step n | Total work for step $\frac{n}{2}$ | Work at step n (1 multiplication)

Evaluating $T(n)$. The Master Theorem gives us $\begin{cases} a = 1 \\ b = 2 \\ d = 0 \end{cases}$ so $\log_b a = \log_2 1 = 0 = d$ so $\boxed{T(n) = O(\log n)}$

B) Tower of Hanoi: Looking at the recursion tree

Algorithm. The algorithm requires that to move n disks, we move first $n-1$ disks, then 1 and finally $n-1$ again[2].

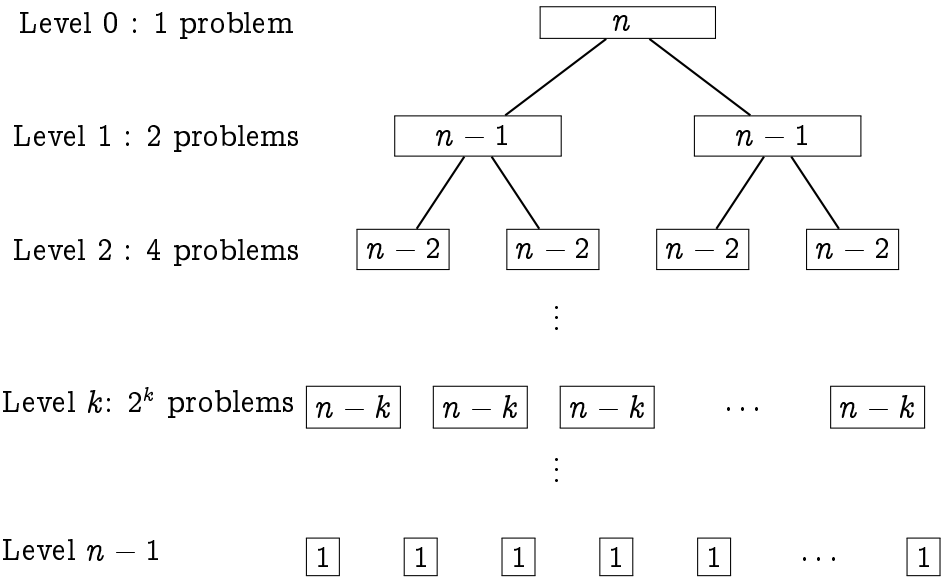
Time complexity equation. Let $T(n)$ be the number of moves.

$$\boxed{T(n)} = \boxed{T(n-1)} + \boxed{1} + \boxed{T(n-1)}$$

$T(n)$	$T(n-1)$	1	$T(n-1)$
Total work for n disks	Total work for $n-1$ disks	Move 1 disk	Total work for $n-1$ disks

We can't apply Master Theorem on the complexity equation $T(n) = 2T(n-1) + 1$ so we have to evaluate $T(n)$ to be able to give a big-O bound.

The recursion tree. Let's consider the recurrence sub-problem tree:



At the k-th level, there are 2^k sub-problems that all require 1 move to get to level $k-1$. So, there are $w(k) = 2^k$ moves at level k (for level k only).

[2]Google "tower of hanoi recursive solution" if you don't remember the solution.

Summing to evaluate $T(n)$. As $T(n)$ is the sum of the moves of each level, we get:

$$T(n) = 1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-1}$$

We recognise here the sum of the n first terms of a geometric progression whose ratio is 2 and first term is 1 so we get:

$$T(n) = \frac{2^n - 1}{2 - 1} = 2^n - 1$$

Therefore, $\boxed{T(n) = O(2^n)}$.

C) Tower of Hanoi: Guessing with the first terms then prove

When you have the recursion equation $T(n) = 2\ T(n-1) + 1$, you can try to guess the value of $T(n)$ by looking at the first terms:

$$T(0) = 0, \quad T(1) = 1, \quad T(2) = 3, \quad T(3) = 7, \quad T(4) = 15, \quad T(5) = 31, \quad T(6) = 63, \quad \ldots$$

We can guess it may be $T(n) = 2^n - 1$, but we have to prove it! Let's do it by recursion on n:

- *Base case.* $T(0) = 0$. OK!
- *Inductive hypothesis.* Let's suppose $T(n) = 2^n - 1$ for a $n \geq 0$.
- *Inductive step.* We have:

$$\begin{aligned} T(n+1) &= 2\ T(n) + 1 \\ &= 2(2^n - 1) + 1 && \text{(ind. hyp.)} \\ &= 2^{n+1} - 1 \end{aligned}$$

- *Conclusion.* We have proved that $T(n) = 2^n - 1$ for all $n \geq 0$.

Therefore, $\boxed{T(n) = O(2^n)}$.

D) Tower of Hanoi: Guessing by unrolling then prove

Another way we can guess the solution is by unrolling the recurrence, by substituting it into itself:

$$\begin{aligned} T(n) &= 2\ T(n-1) + 1 \\ &= 2(2\ T(n-2) + 1) + 1 \\ &= 4\ T(n-2) + 3 \\ &= 4(2\ T(n-3) + 1) + 3 \\ &= 8\ T(n-3) + 7 \\ &\ldots \end{aligned}$$

Here we can guess a new recurrence:

$$T(n) = 2^k\ T(n-k) + (2^k - 1)$$

But we have also to prove it! Let's do it by recursion on k:

- *Base case.* For $k = 0$, we have $2^0\ T(n-0) + (2^0 - 1) = T(n)$. Ok!
- *Inductive hypothesis.* Let's suppose $T(n) = 2^k\ T(n-k) + (2^k - 1)$ for a $k \geq 0$.
- *Inductive step.* We have:

$$\begin{aligned} T(n) &= 2^k\ T(n-k) + (2^k - 1) \\ &= 2^k\ (2\ T(n-k-1) + 1) + (2^k - 1) && \text{intial recursion} \\ &= 2^{k+1}\ T\left(n-(k+1)\right) + (2^{k+1} - 1) \end{aligned}$$

- *Conclusion.* We have proved that $T(n) = 2^k\ T(n-k) + (2^k - 1)$ for $k \geq 0$.

Therefore, for $k = n$ we get $T(n) = 2^n - 1$ so $\boxed{T(n) = O(2^n)}$.

E) Closest Pair of Points in a Plane

Let's evaluate the time complexity of the closest pair of points in a plane solution using Divide and Conquer.

n is a power of 2. We assume without loss of generality that n is power of 2. This will not influence the final bound in any important way, after all, n is at most a multiplicative factor away from some power of 2.

Sorting the points before solving. First, before starting to solve the problem, we sort twice: Once by the x-coordinate and once by the y-coordinate. No further sorting is required at subsequent recursive steps. Using a standard sorting algorithm requires $O(n \log n)$, for example via mergesort.

Solving. For a set of n points in the plane (let $T(n)$ be the time complexity):

- We partition the current set into 2 sub-sets defined by median x-coordinate. As the points have been sorted by x-coordinate, finding the median requires only a constant time operation.
- Recursively compute the closest pair distances for the sub-sets. This requires a complexity of $T(\frac{n}{2}) + T(\frac{n}{2})$.
- Deal with the middle ribbon. Worst case scenario:
  - all points from the left side sub-problem ($\frac{n}{2}$ points) are in the ribbon
  - and for each of them, we have to check a maximum of 6 points in the right side. As the points are sorted by y-coordinates and we step through at most $\frac{n}{2}$, checking the middle ribbon takes at most a linear amount of time.
- Finally take the minimum distance among:
  - the smallest of the right side,
  - the smallest of the left side,
  - and the smallest of the middle ribbon.

  That takes a constant time: $O(1)$.

We get:

$$\boxed{T(n)} = \boxed{O(n)} + \boxed{T\left(\frac{n}{2}\right)} + \boxed{T\left(\frac{n}{2}\right)} + \boxed{O(n)} + \boxed{O(1)}$$

Work for n points	Splitting the plane (x-coord median)	Work for the left side	Work for the right side	Work for the ribbon	Combining results (min of 3 numbers)

It can be simplified in:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + O(n)$$

Master Theorem gives us $\begin{cases} a = 2 \\ b = 2 \\ d = 1 \end{cases}$ so $\log_b a = \log_2 1 = 1 = d$ so $T(n) = O(n \log n)$

Conclusion. We did 2 steps:

- before solving, we sorted the points with a complexity of $O(n \log n)$,
- and we solved the problem with a complexity of $T(n) = O(n \log n)$.

So the overall time complexity is $\boxed{O(n \log n)}$.

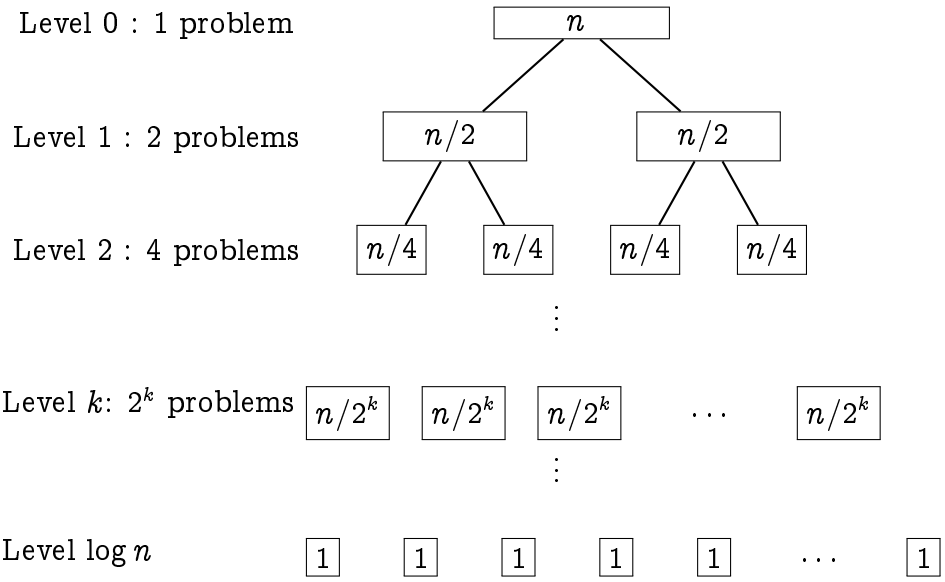
F) Recursion tree again

We are given:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

We assume without loss of generality that n is power of 2. This will not influence the final bound in any important way, after all, n is at most a multiplicative factor away from some power of 2.

We can't use the Master Theorem so we have to evaluate $T(n)$ to be able to give the complexity in big-O terms. Let's consider the recurrence sub-problem tree:



As we know the size of the sub-problems at level k is $\frac{n}{2^k}$ The work for one sub-problem of level k is, according to the complexity equation:

$$\frac{\frac{n}{2^k}}{\log\left(\frac{n}{2^k}\right)} = \frac{1}{2^k} \times \frac{n}{\log(n) - k}$$

As there is 2^k sub-problems at level k, the total work made at level k is:

$$\boxed{2^k} \times \boxed{\frac{1}{2^k} \times \frac{n}{\log(n) - k}} = \boxed{\frac{n}{\log(n) - k}}$$

- 2^k: # of problems at level k
- $\frac{1}{2^k} \times \frac{n}{\log(n) - k}$: work for each problem of level k
- $\frac{n}{\log(n) - k}$: total work made at level k

Therefore:

$$\begin{aligned} T(n) &= (\text{work at level 0}) + (\text{work at level 1}) + \cdots + (\text{work at level } \log(n) \text{ - 1}) \\ &= \frac{n}{\log(n)} + \frac{n}{\log(n) - 1} + \cdots + \frac{n}{2} + \frac{n}{1} \\ &= n \times \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\log(n) - 1} + \frac{1}{\log(n)}\right) \end{aligned}$$

We saw in the homework that we had:

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{m} = O(\log m)$$

In our previous sum, $m = \log n$ so we get $\boxed{T(n) = O(n \log(\log n))}$.