Analysis of Algorithms

1. Introduction: An essential tool to design an efficient and suitable algorithm is the "Analysis of Algorithms". There is no magic formula it is simply: a matter of judgement - intuition - experience. Nevertheless there are some basic techniques that are often useful, such as knowing how to deal with control structures and recurrence equations.

2. Control Structures Analysis: Eventually analysis of algorithm proceeds from the inside out. Determine first, the time required by individual instructions, then combine these times according to the control structures that combine the instructions in the program.

2.1 Sequencing: Let P_1 and P_2 be two fragments of an algorithm. They may be simple instructions or complicated subalgorithms. Let t_1 and t_2 be the times taken by P_1 and P_2. t_1 and t_2 may depend on various parameters, such as the instance size. The sequencing rule says that the time required to compute "P_1; P_2" is simply $t_1 + t_2$. By the maximum rule, this time is in $\Theta(\max(t_1, t_2))$. Despite its simplicity, applying this rule is sometimes less obvious than it may appear. It could happen that one of the parameters that control t_2 depends on the result of the computation performed by P_1.

2.2. "For" loops: These are the easiest loops to analyze.

for $i \leftarrow 1$ to m do $P(i)$

By a convention we'll adopt: $m=0$ means $P(i)$ is not executed at all, (not an error).

$P(i)$ could depend on the size. Of course, the easiest case is when it doesn't. Let t be the time required to Compute $P(i)$, and the total time required is $t = mt$. usually this approach is adequate, but there is a potential pitfall: We didn't consider the time for the "loop control". After all our for loop is shorthand for something like the following while loop.

```
i ← 1
while i ≤ m do
    P(i)
    i ← i+1
```

In most situations it is reasonable to count the test $i \le m$ at unit cost and same thing with the instructions $i \leftarrow i+1$ and the sequencing operations "goto" implicit in the while loop. Let "c" be the upper bound on the time required by each of these operations:

$$\begin{aligned} t \le\ & c && \text{for } i \leftarrow 1 \\ & + (m+1)c && \text{tests } i \le m \\ & + mt && \text{executions of } P(i) \\ & + mc && \text{" " } i \leftarrow i+1 \\ & + mc && \text{sequencing operations} \\ \le\ & (t+3c)m + 2c \end{aligned}$$

This time is clearly bounded below by mt. If c is negligible compared to t, our previous estimate that t is roughly equal to mt is justified.

The analysis of for loops is more interesting when the time $t(i)$ required for $P(i)$ varies as a function of i and or also on size n.

So: for $i \leftarrow 1$ to m do $P(i)$ takes a time given by; $\sum_{i=1}^{m} t(i)$ (ignoring time taken by the loop control).

Example: Computing the Fibonacci sequence,

```
function Fibitir(n)
    i ← 1 ; j ← 0
    for k ← 1 to n do
        j ← i+j , i ← j-i
    return j
```

If we count all arithmetic operations at unit cost, the instructions inside the "for" loop take constant time. Let this time be bounded by some constant c. Not taking control loop is to account, the time taken by the "for" loop is bounded by: nc, we conclude that the algorithm takes a time in $O(n)$. Similar, reasoning yields that this time is in $\Omega(n)$, thus $\theta(n)$.
This is not reasonable. $j \leftarrow j+i$ is increasingly expensive each time round the loop. To know the approximate time, we need to know the length of the integers involved at any given k-th trip round the loop.
j and i are respectively the k^{th} and $k-1^{st}$ values of the Fibonacci sequence, sometimes denoted f_k and f_{k-1}, where f_k can be expressed as:

$$f_k = \frac{1}{\sqrt{5}} \left[\phi^k - (-\phi)^{-k} \right]$$

where $\phi = (1+\sqrt{5})/2$ is the Golden rule.

when k is large, ϕ^{-k} is negligible. and $f_k \simeq \phi^k$ and the size of f_k is in the order of k. Thus:
$f_k \in \theta(k)$, and the kth iteration takes a time $\theta(k-1) + \theta(k)$, which is of course $\theta(k)$

Let c be a constant such that this time is bounded above by ck for all $k \geq 1$. If we again neglect the time required by the loop control, for the instructions before and after the loop we conclude:

$$\sum_{k=1}^{n} ck = c \sum_{k=1}^{n} k = c \frac{n(n+1)}{2} \in O(n^2)$$

Similar reasoning yields that this time is in $\Omega(n^2)$ and therefore it is in $\Theta(n^2)$. And this shows the crucial difference whether or not we count arithmetic operations at unit cost.

2.3 Recursive Calls:

The analysis of recursive algorithms is straightforward to a certain point. Simple inspection of the algorithm often gives rise to a recurrence equation that "mimics" the flow of control in the algorithm. General techniques on how to solve or how to transform the equation into simpler non recursive equations will be seen later.

2.4 "While" and "repeat" loops:

These two types of loops are usually harder to analyse than "for" loops because there is no obvious a priori way to know how many times we shall have to go around the loop. The standard technique for analyzing these loops is to find a function of the variables involved whose value decreases each time around. To determine how many times the loop is repeated, however, we need to understand better how the value of this function decreases. An alternative approach to the analysis

of "while" loops consists of treating them like recursive algorithms. We illustrate both techniques with the same example. The analysis of "repeat" loops is carried out similarly.

Example: Binary Search - the purpose is to find an element x in an array T[1..n] that is in non decreasing order. (assume for simplicity that x is guaranteed to appear at least once.) We require to find an integer i such that $1 \leq i \leq n$ and $T[i] = x$. The basic idea behind binary search is to compare x with the element y in the middle of T. The search is over if $x = y$; it can be confined to the upper half of the array if $x > y$; otherwise search in the lower half.

```
function Binary_Search (T[1..n], x)
    i ← 1 ; j ← n
    while i < j do
        k ← (i + j) ÷ 2
        Case  x < T[k]: j ← k-1
              x = T[k]: i, j ← k {return k}
              x > T[k]: i ← k+1
    return i
```

It is obvious in this algorithm that the function (whose variables) that decreases each time round the loop is $j - i + 1$ which we call d. Thus d represents the number of elements of T still under consideration. Initially $d = n$. The loop terminates when $i \geq j$, which equivalent to $d \leq 1$. Each time round the loop, there are 3 possibilities:

- Either j is set to $k-1$ or
- i is set to $k+1$ or
- i and j are set to k.

Let d and $\hat{d}$ stand respectively for the value of $j-i+1$ before and after the iteration under consideration. We use $i, j, \hat{i}, \hat{j}$ similarly.

If $x < T[k]$, the instruction "$j \leftarrow k-1$" is executed and thus $\hat{i} = i$ and $\hat{j} = \lfloor (i+j) \div 2 \rfloor - 1$. Therefore,

$$\hat{d} = \hat{j} - \hat{i} + 1 = (i+j) \div 2 - i \leq \frac{(i+j)}{2} - i < (j-i+1)/2 = \frac{d}{2}$$

Similarly: if $x > T[k]$, the instuction "$i \leftarrow k+1$" is executed and thus:

$$\hat{i} = \lfloor (i+j) \div 2 \rfloor + 1 \quad \text{and} \quad \hat{j} = j$$

Therefore,

$$\hat{d} = \hat{j} - \hat{i} + 1 = j - (i+j) \div 2 \leq j - (i+j-1)/2 = (j-i+1)/2 = d/2$$

Finally, if $x = T[k]$, then i and j are set to the same value. and $\hat{d} = 1$, but d was at least 2 since otherwise the loop would not have reentered. We conclude that $\hat{d} \leq d/2$ which ever case happens, which means that d is halved each time round the loop. How much time is taken? Let's determine an upper bound on the running time of binary search. Let d_ℓ denote the value of $j-i+1$ at the end of the ℓ-th trips round the loop for $\ell \geq 1$ and $d_0 = n$. So: $d_\ell \leq \frac{d_{\ell-1}}{2} \ \forall \ \ell \geq 1$, then it follows that: $d_\ell \leq \frac{n}{2^\ell}$ and it terminates when $d \leq 1$ which happens when $\ell = \lceil \log n \rceil$, we conclude that

the loop is entered at most $\lceil \log n \rceil$ times. Since each trip takes constant time, binary search takes a time in $O(\log n)$ a similar reasoning yields a lower bound of $\Omega(\log n)$ in the worst case, and thus binary search takes a time in $\Theta(\log n)$

The alternative approach to analyzing the running time of binary search begins much the same way. The idea is to think of the "while" loop as if it were implemented recursively instead of iteratively. Each time round the loop we reduce the range of possible locations for x in the array. Let $t(d)$ denote the maximum time needed to terminate the while loop when $j - i + 1 \leq d$. We have already seen that $j - i + 1$ is halved each time round the loop. In recursive terms, this means that $t(d)$ is at most the constant time b required to go round the loop once, plus the time $t(d \div 2)$ sufficient to finish the loop from there. Since it takes constant time c to determine that the loop is finished when $d = 1$, we obtain the following recurrence.

$$t(d) \leq \begin{cases} c & \text{if } d = 1 \\ b + t(d \div 2) & \text{otherwise} \end{cases}$$

4.3 More examples:

- Selection Sort

```
procedure Select (T[1..n])
    for i ← 1 to n-1 do
        minj ← i ; minx ← T[i]
        for j ← i+1 to n do
            if T[j] < minx then minj ← j , minx ← T[j]
        T[minj] ← T[i] , T[i] ← minx
```

The time taken by the inner loop is not constant - it takes longer when $T[j] < min x$ - the time is bounded above by some constant c. For each value of i, the instructions in the inner loop are executed $n-(i+1)+1 = (n-i)$ times, and therefore the time taken by the inner loop is $t(i) \leq (n-i)\,c$. The time taken for the i^{th} trip round the outer loop is bounded above by $b + t(i)$ for an appropriate constant b that takes account of the elementary operations before and after the inner loop and of the loop control for the outer loop. Therefore, the total time spent by the algorithm is bounded above by

$$\sum_{i=1}^{n-1} b + (n-i)\,c = \sum_{i=1}^{n-1} (b+cn) - c\sum_{i=1}^{n-1} i$$

$$= (n-1)(b+cn) - cn(n-1)/2$$

$$= \frac{1}{2}cn^2 + \left(b - \frac{1}{2}c\right)n - b$$

which is in $O(n^2)$. Similar reasoning shows that this time is also $\Omega(n^2)$ and therefore selection sort takes a time in $\Theta(n^2)$.

- Insertion Sort:

```
procedure insert(T[1..n])
    for i ← 2 to n do
        x ← T[i] ; j ← i-1
        while j > 0 and x < T[j] do T[j+1] ← T[j]
                                    j ← j-1
        T[j+1] ← x
```

Unlike selection sorting, we saw before that the time taken to sort n items by insertion depend a lot on the original order of the elements. Let's check the worst case

Suppose for a moment that i is fixed. Let $x = T[i]$
The worst case arises when each time x is less than $T[j]$ (i.e. for every j between 1 and $i-1$), since in this case we have to compare x to $T[i-1]$, $T[i-2]$, ..., $T[1]$ before we leave the while loop (because $j=0$). Thus the while loop test is performed i times in the wort case. This worst case happens for every value of i from 2 to n when the array is initially sorted into descending order. The test is thus performed:

$$\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$$

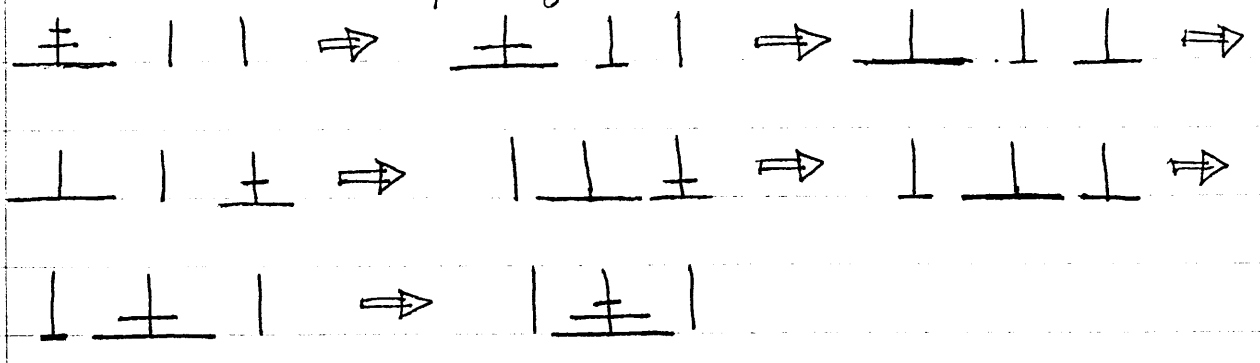
This shows that insertion sort takes a time in $\theta(n^2)$ to sort n items in the worst case.

. The Towers of Hanoi

This example provides us with an example of the analysis of recursive algorithms.
The Towers of Hanoi was proposed by a french Mathematician, Edouard Lucas in 1883. He concocted a story to explain the problem:
"The monks of Benares, maintained a device consisting of 3 diamond pegs onto which 64 different sized disks were placed. Initially all the disks were on one peg and formed a pyramid. The monks sole task was to move the pyramid from one peg to another following these 3 rules: -1- Only one disk can be moved at a time.
-2- a larger disk cannot be placed on a smaller disk.
-3- no disk can be placed anywhere except on a peg.

The monks believed when they finished their task the world would end. This is probably the most reassuring prophecy ever made concerning the end of the world, for if the monks manage to move one disk per second, working days and night ~~with~~ without ever resting, nor making a mistake, their work will not be finished 500,000 Millions years after they began. This is more than 25 times the estimated age of the Universe!

Generate the example of 3 disks. $n = 3$



The problem can be generalized to an arbitrary number n of disks. To solve the general problem, we need only realize that to transfer the m smallest disks from peg i to j, where $1 \le i \le 3$, $1 \le j \le 3$, $i \neq j$ and $m \ge 1$, we can first transfer the smallest $m-1$ disks from peg i to peg $k = 6 - i - j$, next transfer the m-th disk from peg k to j. And this is the formal description of this algorithm where our original instance is (64, 1, 2).

```
procedure Hanoi (m, i, j)
{ moves the m smallest rings from peg i to peg j }
if m > 0 then Hanoi (m-1, i, 6-i-j)
    write i → j
    Hanoi (m-1, 6-i-j, j)
```

To analyze the execution time of this algorithm, we look into the write statement. Let $t(m)$ the number of times it is executed on a call of $H(m,\cdot,\cdot)$. By inspection of the algorithm we obtain the following recurrence:

$$t(m) = \begin{cases} 0 & \text{if } m = 0 \\ 2t(m-1)+1 & \text{otherwise} \end{cases}$$

we will show that:

$$t(m) = 2^m - 1 \quad \text{which is } \Theta(2^m)$$

4.4. Solving recurrences:

How to solve a recurrence equation? I believe it requires a little of experience and intuition. A lot of "intelligent" guesswork too, besides, of course, the powerfull technique that is used for certain special classes of recurrence. That is the characteristic equation.

4.4.1. Intelligent guesswork:

This approach generally proceeds in 4 stages:

- calculate the first few values of the recurrence
- look for regularity
- guess a suitable general form
- Prove by mathematical induction that this form is correct.

Example: Consider the following example $T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 3T(n \div 2) + n & \text{otherwise} \end{cases}$

remember $n \div 2 = \lfloor n/2 \rfloor$ which is very hard to analyze - consider $n \div 2 = \frac{n}{2}$ only, and this is true when n is a power of 2.

n	1	2	4	8	16	32
$T(n)$	1	5	19	65	211	665

what regularity to look for? It may appear more obvious when we keep more "history" of the value of $T(n)$ instead of writing: $T(2) = 5$, it may be more useful to try it as: $2\times 2+1$ or $3\times 1+2$ or $3\times 2-1$ ...

$$T(4) = 3\times T(2)+4 = 3(2\times 2+1)+4 \text{ or}$$
$$= 3(3\times 1+2)+4 \text{ or}$$
$$= 3(3\times 2-1)+4$$

The 2nd choice seems more promising as:

$$T(4) = 3(3\times 1+2)+4 = 3^2\times 1 + 3\times 2 + 4 = 3^2\times 2^0 + 3^1\times 2^1 + 3^0\times 2^2$$

n	$T(n)$
1	1
2	$3\times 1+2$
2^2	$3\times 2^0 + 3^1\times 2^1 + 3^0\times 2^2$
2^3	$3^3\times 2^0 + 3^2\times 2 + 3\times 2^2 + 3^0\times 2^3$
2^4	$3^4\times 2^0 + 3^3 + 2^1 + 3^2\times 2^2 + 3^1\times 2^3 + 3^0\times 2^4$
⋮	⋮

The pattern is now obvious

$$T(2^k) = 3^k 2^0 + 3^{k-1}\times 2 + \dots + 3^0\times 2^k$$
$$= \sum_{i=0}^{k} 3^{k-i}\times 2^i = 3^k \sum_{i=0}^{k} \left(\frac{2}{3}\right)^i = 3^k \frac{1-(2/3)^{k+1}}{1-\frac{2}{3}}$$
$$= 3^{k+1} - 2^{k+1}$$

Now this needs to be proven by induction.

assume $T(2^{k-1}) = 3^k - 2^k$

Prove $T(2^k)$?

$$T(2^k) = 3\times T(2^{k-1}) + n \quad \text{original definition}$$
$$= 3\times(3^k - 2^k) + 2^k$$
$$= 3^{k+1} - 3\times 2^k + 2^k = 3^{k+1} - 2^k(3-1)$$
$$= 3^{k+1} - 2^{k+1} \quad \text{which is ok.}$$

What happens when n is not a power of 2?
Solving the original equation is rather difficult. It is also unnecessary, because we can be happy with simply an asymptotic notation.

$$T(2^k) = T(n) = 3\,T(2^{\log_2 n}) = 3^{1+\log_2 n} - 2^{1+\log_2 n}$$

using the fact that:

$$3^{\log_2 n} = n^{\log_2 3}$$

it follows:

$$T(n) = 3n^{\log_2 3} - 2n$$

when n is a power of 2, and then

$$T(n) \in \Theta(n^{\log_2 3} \mid n \text{ is a power of } 2)$$

Since $T(n)$ is a non decreasing function, and $n^{\log_2 3}$ is a smooth function, it follows that $T(n) \in \Theta(n^{\log_2 3})$ unconditionally.

Smoothness Rule: Let $f, t : \mathbb{N} \to \mathbb{R}^+$ where f is a smooth function, and t is a non decreasing function. Consider any integer $b \geq 2$. The smoothness rule asserts that if:

$t(n) \in \Theta(f(n) \mid n \text{ is a power of } b)$ then $t(n) \in \Theta(f(n))$.

The rule applies equally to O and Ω.

4.5. Homogeneous recurrences.

characteristic equation - Let's start with homogeneous linear recurrences with constant coefficients, i.e. recurrences of the form:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \qquad (1)$$

$t_i = t(i)$ are the values we are looking for -
our recurrence is:

-i- linear: because it doesn't contain terms like $t_{n-i} \cdot t_{n-j}$ or t_{n-i}^2 ...

-ii- homogeneous: linear combination equals zero.

-iii- Constant Coefficients: all a_i's are constants.

Consider the Fibonacci Sequence as an example

$$f_n = f_{n-1} + f_{n-2}$$

which can be rewritten as:

$$f_n - f_{n-1} - f_{n-2} = 0$$

$k=2$, $a_0 = 1$, $a_1 = a_2 = -1$

Before we start to look into solutions of equation (1) it is interesting to note that any linear combination of solutions is itself a solution - i.e. if f_n and g_n are solutions to (1), so $\sum_{i=0}^{k} a_i f_{n-i} = 0 = \sum_{i=0}^{k} a_i g_{n-i}$ and if we set $t_n = c f_n + d g_n$ for arbitrary constants c and d, then t_n is also a solution.

Let, at this juncture, us use some intelligent guesswork to solve (1), looking for solutions of the form:

$$t_n = x^n$$

where x is a constant as yet unknown. If we try it in (1) we obtain:

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0$$

This is of course satisfied when $x=0$ (trivial).

Otherwise:

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k x^0 = 0$$

This equation of degree k is called the characteristic equation of the recurrence (1), and:

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

is the characteristic polynomial.

$p(x)$ has k roots, thus:

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

where the r_i may be complex numbers, that are the only solutions of the equation $p(x) = 0$

Since $p(r_i) = 0$, then $x = r_i$ is a solution to the characteristic equation and therefore r_i^n is a solution to the recurrence. Since any linear combination is a solution also, we conclude that:

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

Satisfies the recurrence for any choice of $c_1, c_2, \ldots, c_k$.
The surprising thing is that equation (1) has only solutions of this form, provided that all r_i are distinct. In this case, the k constants can be determined from k initial conditions by solving a system of k linear equations.

Example (1) consider the following recurrence

$$f_n = \begin{cases} n & \text{if } n = 0, 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

rewrite the recurrence: $f_n - f_{n-1} - f_{n-2} = 0$

the characteristic polynomial:

$$p(x) = x^2 - x - 1$$

whose roots are:

$$r_1 = \frac{1 + \sqrt{5}}{2}, \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

The general solution is of the form

$$f_n = c_1 r_1^n + c_2 r_2^n$$

use initial conditions to determine c_1 and c_2.

when $n = 0$ $\quad f_0 = c_1 + c_2 = 0$

when $n = 1$ $\quad f_1 = c_1 r_1 + c_2 r_2 = 1$

System of 2 equations with 2 unknowns - we obtain

$$c_1 = \frac{1}{\sqrt{5}} \quad , \quad c_2 = -\frac{1}{\sqrt{5}}$$

Thus: $$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

This is known as de Moivre formula for the fibonacci sequence.

Exple ② Consider $$t_n = \begin{cases} 0 & \text{if } n=0 \\ 5 & \text{if } n=1 \\ 3t_{n-1} + 4t_{n-2} & \text{otherwise} \end{cases}$$

rewrite:

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

characteristic polynomial:

$$x^2 - 3x - 4 = (x-4)(x+1)$$

roots are obviously $r_1 = -1$, $r_2 = 4$, and the general solution is of the form:

$$t_n = c_1 (-1)^n + c_2 4^n$$

initial conditions give:

$$c_1 + c_2 = 0 \qquad n=0$$

$$-c_1 + 4c_2 = 5 \qquad n=1$$

Solving these equations, we obtain $c_1 = -1$, $c_2 = 1$.

Therefore:

$$t_n = 4^n - (-1)^n$$

The problem becomes a little harder when you have multiple

roots, that is when the k roots are not all distinct. In this case (without going through the details of the proof) we state that $t_n = n r^n$ is also a solution, and if r is a root with multiplicity m then:

$t_n = r^n$, $t_n = n r^n$, $t_n = n^2 r^n$, ..., $t_n = n^{m-1} r^n$

The general solution can be written as:

$$t_n = \sum_{i=1}^{l} \sum_{j=0}^{m_i - 1} c_{ij}\, n^j\, r_i^n$$

where $r_1, r_2, \dots, r_l$ are the l distinct roots of the characteristic polynomial and their multiplicities are: $m_1, m_2, \dots, m_l$.

Exple ③ : Consider $t_n = \begin{cases} n & \text{if } n = 0,1,2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{otherwise} \end{cases}$

$\rightarrow \quad t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$

the characteristic polynomial is:

$$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2$$

roots are $r_1 = 1$ (multiplicity 1, i.e $m_1 = 1$)

and $r_2 = 2$ (" 2, " $m_2 = 2$)

the general solution is:

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

the initial conditions give:

$$c_1 + c_2 = 0 \qquad (n=0)$$

$$c_1 + 2c_2 + 2c_3 = 1 \qquad (n=1)$$

$$c_1 + 4c_2 + 8c_3 = 2 \qquad (n=2)$$

after solving these equations we obtain

$c_1 = -2$, $c_2 = 2$, $c_3 = -\frac{1}{2}$. Therefore $t_n = 2^{n+1} - n2^{n-1} - 2$