Data Structures and Algorithms
Lecture slides: Analyzing algorithms: recurrences, Brassard section 4.7

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

Topics cover

Introduction

The running time of recursive algorithms cannot be analyzed in the same way as iterative algorithms

There are no loops, so summations to count the number of basic operations in loops do not apply for recursive algorithms

Rather recurrence relations will be used to express the number of times basic operations are executed in recursive algorithms

Merge Sort

Merge sort is a recursive sorting algorithm. It works as follow :

- **Divide** the array to sort in half.
- **Recursively** sort each half.
- **Merge** the two sorted halves.

Pseudo-code of merge sort is as follows :

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Merge function

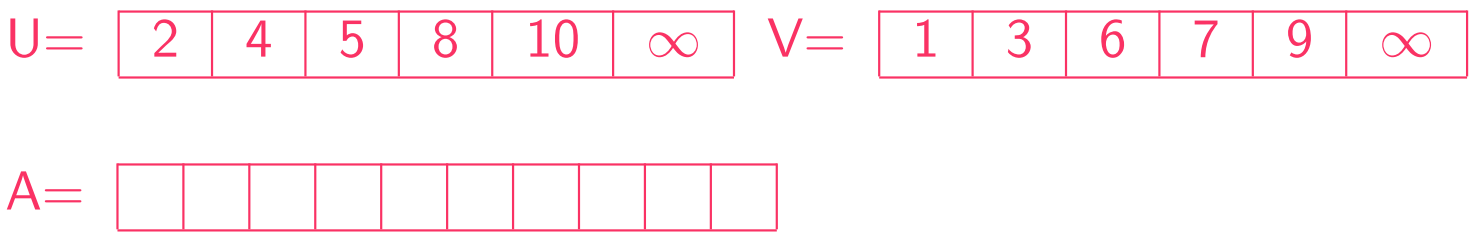
The merge function is the computation inside each recursive function call.

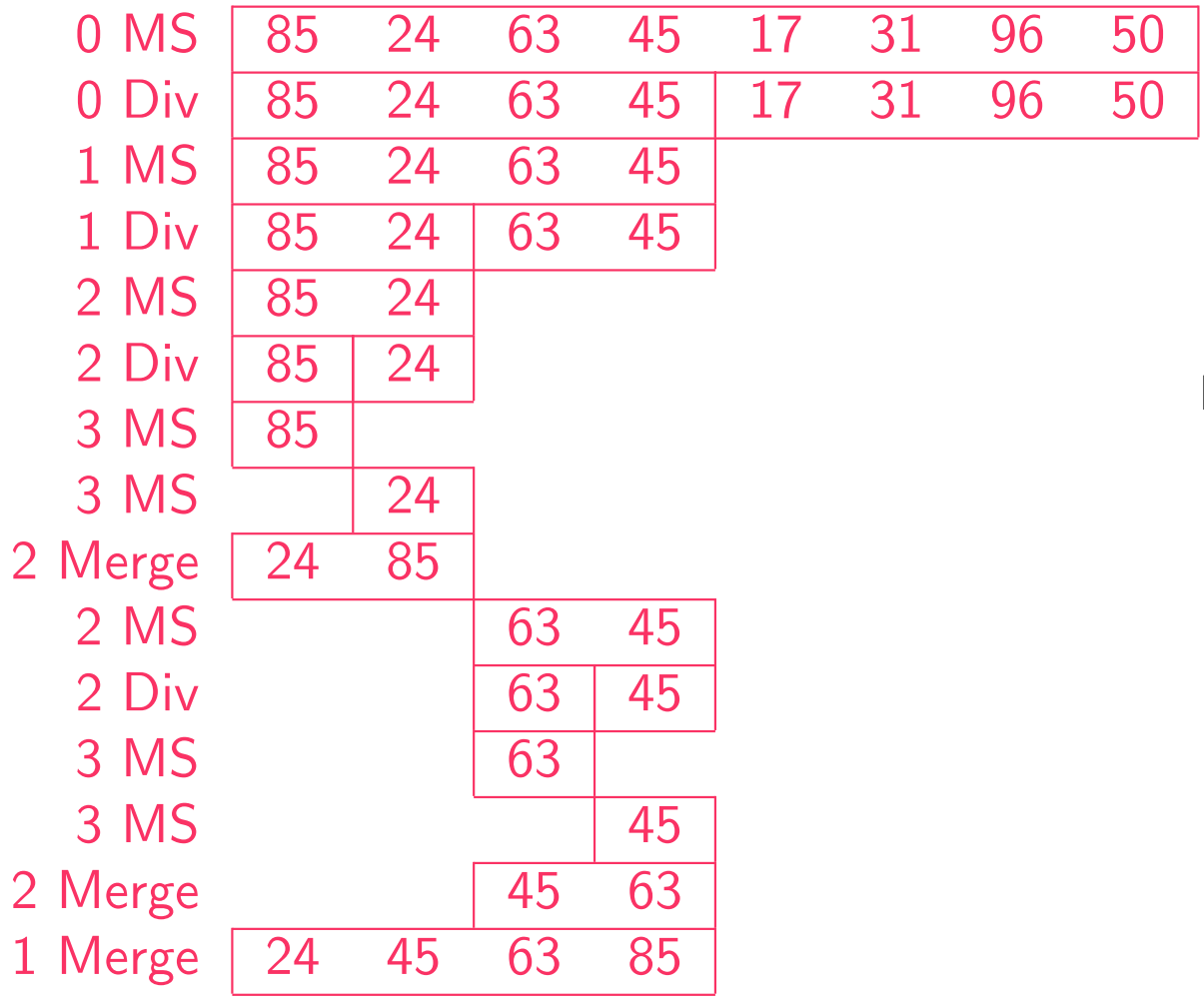
Merge() takes in input 2 sorted sub-arrays U and V respectively of size m and n and return a sorted array A of size $m + n$

$Merge(U[1..m+1], V[1..n+1], A[1..m+n])$;
  $i = j = 1; U[m+1] = V[n+1] = \infty$;
  for $(k = 1; k \leq m + n; k++)$
    if $U[i] < V[j]$ **then**
      $A[k] = U[i]$; $i = i + 1$;
    else
      $A[k] = V[j]$; $j = j + 1$;

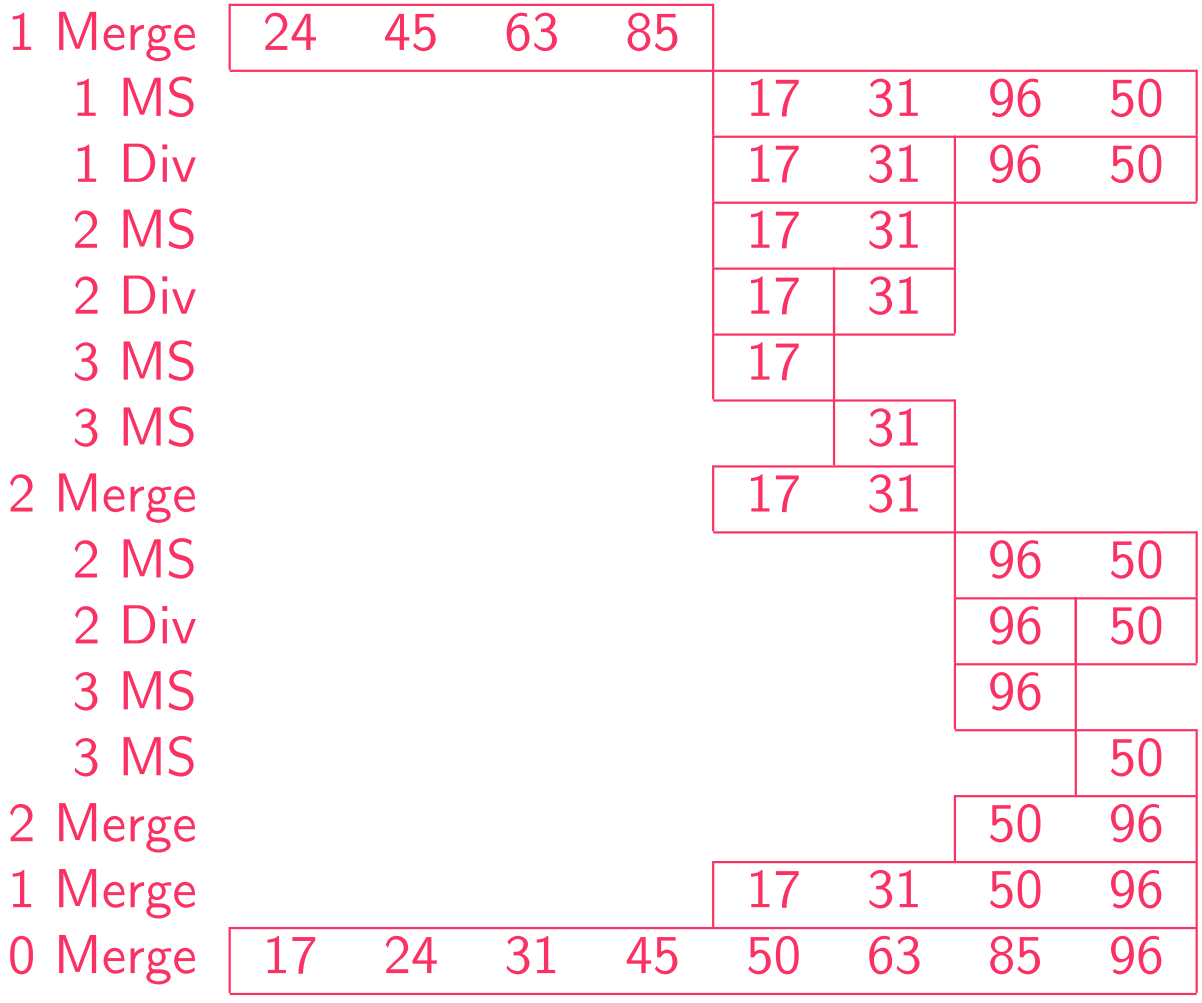
Merge function

```
Merge(U[1..m+1], V[1..n+1], A[1..m+n]);
  i = j = 1; U[m+1] = V[n+1] = ∞;
  for (k = 1; k ≤ m+n; k++)
      if U[i] < V[j] then
          A[k] = U[i]; i = i+1;
      else
          A[k] = V[j]; j = j+1;
```





Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)


1 Merge
24 45 63 85
1 MS
17 31 96 50
1 Div
17 31 96 50
2 MS
17 31
2 Div
17 31
3 MS
17
3 MS
31
2 Merge
17 31
2 MS
96 50
2 Div
96 50
3 MS
96
3 MS
50
2 Merge
50 96
1 Merge
17 31 50 96
0 Merge
17 24 31 45 50 63 85 96

Quicksort

Another recursive sorting algorithm

An array A[p..r] of integers is partitioned into two subarrays A[p..q] and A[q+1..r]

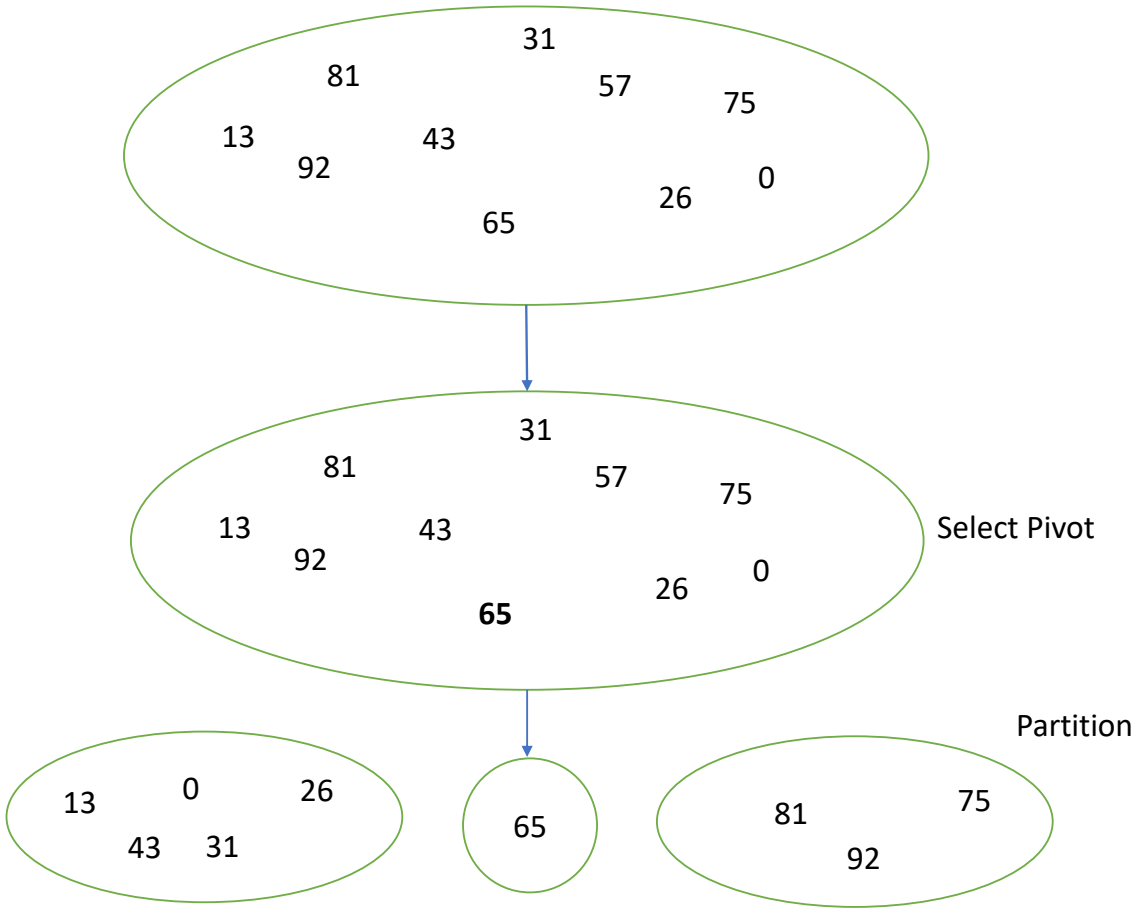
- such that the elements in A[p..q] are less than all elements in A[q+1..r]

The subarrays are recursively sorted by calls to quicksort

Quicksort Code

```
Quicksort(A, p, r)
  if (p < r) /* the array has at least 2 elements */
    q = Partition(A, p, r); /* q is the index in A of the pivot element */
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

Quicksort idea



After partition, the pivot is in its right position in a sorted array. Quicksort is recursively applied to the next two subarrays.

Partition

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r);
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

- All the action takes place in the partition() function which rearranges the subarray
- End result : Two subarrays, all values in first subarray $<$ all values in second
- Returns the index of the "pivot" element separating the two subarrays

Partition code

Partition() takes in input the array A and the section between indexes p and r to partition

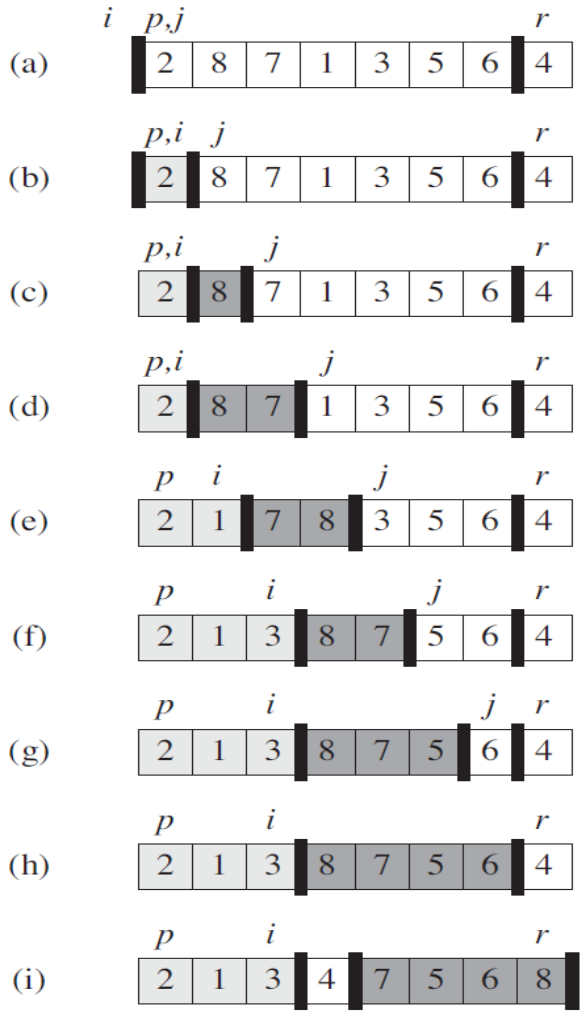
x is the pivot element, in this code the last element in the array is selected as pivot

Partition() returns the index in A of the pivot element once partition is completed

The position of the pivot element will never be part of a sub-array to partition, it is at its right position in the sorted array

```
Partition(A, p, r)
    x = A[r];
    i = p - 1;
    for (j = p; j < r; j++)
       if (A[j] ≤ x)
          i = i +1;
          swap(A[i],A[j]);
    swap(A[i+1],A[r]);
    return i+1;
```

Example with running partition()




```
Partition(A, p, r)
    x = A[r] ;
    i = p - 1 ;
    for (j = p ; j < r ; j++)
        if (A[j] ≤ x)
            i = i +1 ;
            swap(A[i],A[j]) ;
    swap(A[i+1],A[r]) ;
    return i+1 ;
```

Running time of recursive algorithms

For iterative algorithms, the running time of an algorithm for an input size n is expressed using summations

For recursive algorithms, the running time of an input size n is expressed in terms of the running time of an instance of smaller size, i.e. as a recurrence relation

A recurrence relation is an equation in which at least one term appears on both sides of the equation such as

$$T(n) = 2T(n-1) + n^2 + 3$$

in which the term $T()$ appears on both side of the equation.

Recursive algorithms

Recursive algorithms are called with a problem input size equal to n, but then each recursive call is applied to smaller problem instances

```
factorial(n)
        if (n ≤ 1)
          return 1;
        else
          return n × factorial(n − 1);
```

There is one recursive call in the code, the input size is reduce by 1 for each call, where each function call execute one multiplication

Recurrence for Factorial

```
factorial(n)
        if (n ≤ 1)
          return 1;
        else
          return n × factorial(n − 1);
```

Running-time of factorial, $T(n)$, is then given by the following recurrence relation :

$$T(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

Terms in a recurrence

$$T(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

- A recurrence relation has 2 types of terms : The *recursive* term $T(n-1)$ and the *non-recursive* term 1.
- The *recursive* term displays the size of the next smaller instance (here it is $n-1$) and stands for the running time $T(n-1)$ of this smaller instance.
- The *non-recursive* term expresses the number of basic operations executed each time the computer enters in the recursive function.

Recurrence for Binary Search

BinarySearch$(A[i..j], x)$
    if $i > j$ **then return** -1
    $k = \lfloor \frac{(i+j)}{2} \rfloor$
    if $A[k] = x$ **return** k
    if $x < A[k]$ **then**
        return $BinarySearch(A[i..k-1], x)$
    else
        return $BinarySearch(A[k+1..j], x)$

The code has two recursive calls, but only one of them is executed each time the function is entered

The size of the input is reduced by half each time a recursive call is made

The code executed in each function call is $k = \lfloor \frac{(i+j)}{2} \rfloor$ which runs in $\Theta(1)$

Thus the time $T(n)$ needed to solve an instance of side n is equal to the time $T(\lfloor \frac{n}{2} \rfloor)$ needed to solve an instance of size $\lfloor \frac{n}{2} \rfloor + 1$

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + \Theta(1)$$

The recurrence of merge sort

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Merge sort has two recursive calls, both are executed each time the function is entered

The size of the input is reduced by half each time a recursive call is made

The most costly operation performed in each recursive call is the procedure Merge() which run $\Theta(n)$

Thus the time $T(n)$ needed to solve an instance of side n is equal to the time to solve two sub-instances of size $\lfloor \frac{n}{2} \rfloor$ plus the time $\Theta(n)$ needed to compute Merge() :

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$$

Recurrence of quicksort : worst case analysis

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r);
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

Each recursive call entails a call to partition() which runs in $O(n)$.

Quicksort calls itself on 2 sub-arrays with likely different sizes. The asymptotic running time of Quicksort depends on how these two sub-arrays are balanced

- If the subarrays are balanced, then quicksort can run as fast as merge sort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

The worst case occurs when the subarrays are completely unbalanced in each recursive call : 0 element in one subarray and $n-1$ elements in the other subarray.

The recurrence is then :

$$T(n) = T(n-1) + T(0) + O(n) \quad (1)$$
$$= T(n-1) + O(n) \quad (2)$$

Solving recurrence relations

Solving a recurrence relation means obtaining a closed-form solution expressing the complexity of an algorithm in terms of the input size n

Both the recursive and non-recursive terms of a recurrence relation are factors in the running time. However, only the non-recursive terms refer to actual computational time.

In the recurrence $T(n) = T(n-1) + 1$

- the recursive term tells there will be n recursive function calls for input sizes of $n, n-1, n-2, \ldots, 2, 1$
- the non-recursive term is the computational cost of each recursive function call, $O(1)$. $\sum_{i=1}^{n} 1 = n$ is the total computational cost,

the closed form solution to this recurrence is $n \rightarrow \Theta(n)$
In the recurrence $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$

- the recursive term tells there will be $\log n$ recursive function calls of sizes $n, \frac{n}{2}, \frac{n}{4}, \ldots, 1$
- each recursive function call cost $O(1)$, so $\sum_{i=1}^{\log_2 n} 1 = \log n$

so the closed form solution to this recurrence is $\log n \rightarrow \Theta(\log n)$

Solving recurrence relations

In the recurrence $T(n) = 2T(n/2) + n$

- the recursive term tells there will be $\log n$ instances of different size, 1 instances of size n, 2 instances of size $\frac{n}{2}$, 4 instances of size $\frac{n}{4}$, etc
- the non-recursive term shows a computing time of n for the first instance, a computing time of $\frac{n}{2}$ for each of the 2 instances of size $\frac{n}{2}$, a computing time of $\frac{n}{4}$ for each of the 4 instances of size $\frac{n}{4}$, ...a computing time of $\frac{n}{n}$ for each of the n instances of size 1
  - $n \in O(n) + 2\frac{n}{2} \in O(n) + 4\frac{n}{4} \in O(n) + \cdots + n \times 1 \in O(n)$
- as each level takes $O(n)$ and there are $\log n$ levels, the running time is $n \log n \in \Theta(n \log n)$

Solving recurrence relations is not always easy. Fortunately, for recurrences derived from recursive algorithms, there exist solution methods that can solve almost all of them. We describe three of them :

- The master method (Master Theorem)
- The substitution method
- The recursion tree method

The Master Theorem

The Master Theorem provides closed forms to recurrences of the form :

$$T(n) = aT(\frac{n}{b}) + cn^m$$

for constants $a \geq 1, b > 1, m \geq 0$.

Note, recurrence relations that can be solved using the master theorem technique always have a polynomial time closed form (except of course if cn^m is exponential

Examples of recurrence relations

$$T(n) = aT(\frac{n}{b}) + cn^m$$

$a \geq 1, b > 1, m \geq 0$

1. $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$. Here $a = 1, b = 2, m = 0$.
2. $T(n) = 2T(n/2) + n$. Here $a = 2, b = 2, m = 1$.
3. $T(n) = T(\sqrt{n}) + n$. Recurrence does not satisfies de conditions for master theorem, b is undefined
4. $T(n) = T(n/3) + T(2n/3) + n$. Recurrence does not satisfies conditions for master theorem. Two recursive terms, each with a different b : 1- $b = 3$ and 2- $b = 3/2$

The Master Theorem

$$T(n) = aT(\frac{n}{b}) + cn^m$$

A solution to a recurrence of the above form (its closed form) is determined by the values of the constants a, b and m

$$T(n) \in \begin{cases} \Theta(n^m) & \texttt{if } a < b^m; \\ \Theta(n^m \log n) & \texttt{if } a = b^m; \\ \Theta(n^{\log_b a}) & \texttt{if } a > b^m. \end{cases}$$

This formulation is sometime refereed as the "restricted" form of the Master Theorem because it can be used only if the non-recursive term is a polynomial of degree ≥ 0

Example 1

Give the exact order of the recurrence

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

Here $a = 2$, $b = 2$ and $m = 1$.

We have $a = 2 = b^m = 2^1 = 2$, therefore the case that applies is $a = b^m$.

Consequently, $T(n) \in \Theta(n^m \log n) = \Theta(n \log n)$.

Example 2

Give the exact order of the recurrence

$$T(n) = 6\,T(\frac{n}{4}) + n^2$$

Here $a = 6$, $b = 4$ and $m = 2$.

We have $a = 6 < b^m = 4^2 = 16$, therefore the case that applies is $a < b^m$.

Consequently, $T(n) \in \Theta(n^m) = \Theta(n^2)$.

Example 3

Give the exact order of the recurrence

$$T(n) = 7T(\frac{n}{2}) + n^2$$

Here $a = 7$, $b = 2$ and $m = 2$.

We have $a = 7 > b^m = 2^2 = 4$, therefore the case that applies is $a > b^m$.

Consequently, $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$.

General form of the Master Theorem

General Master Theorem : Let constants $a \geq 1, b > 1$, and ϵ be a strictly positive real number. $T(n)$ satisfies the recurrence relation

$$T(n) = aT(\frac{n}{b}) + f(n),$$

where $f(n)$ is some function. Let $\alpha = \log_b a$. Then the general form said that the solution to $T(n)$ has 3 cases by comparing n^α and $f(n)$:

$$T(n) \in \begin{cases} 1 : \Theta(n^\alpha) & \texttt{if } f(n) \in O(n^{\alpha-\epsilon}); \\ 2 : \Theta(f(n)\log n) & \texttt{if } f(n) \in \Theta(n^\alpha); \\ 3 : \Theta(f(n)) & \texttt{if } f(n) \in \Omega(n^{\alpha+\epsilon}) \texttt{ and } af(n/b) \leq cf(n) \\ & \texttt{for some } c < 1 \texttt{ and } n \texttt{ large enough.} \end{cases}$$

General form of the Master Theorem

$$T(n) = aT(\frac{n}{b}) + f(n),$$

Let $\alpha = \log_b a$.

$$T(n) \in \begin{cases} 1 : \Theta(n^\alpha) & \text{if } f(n) \in O(n^{\alpha-\epsilon}); \\ 2 : \Theta(f(n)\log n) & \text{if } f(n) \in \Theta(n^\alpha); \\ 3 : \Theta(f(n)) & \text{if } f(n) \in \Omega(n^{\alpha+\epsilon}) \text{ and } af(n/b) \leq cf(n) \\ & \text{for some } c < 1 \text{ and } n \text{ large enough.} \end{cases}$$

- Case 1 $f(n)$ is polynomially-slower than n^α, i.e. the difference between $f(n)$ and n^α is a polynomial n^ϵ : $f(n) \in O(n^\alpha - n^\epsilon)$
- Case 2 $f(n) \in \Theta(n^\alpha)$
- Case 3 $f(n)$ is polynomially-faster than n^α, i.e. the difference between $f(n)$ and n^α is a polynomial n^ϵ : $f(n) \in \Omega(n^\alpha + n^\epsilon)$

General vs restricted Master Theorem

This form of the Master Theorem is called "general" because $f(n)$ can be any function, it is not restricted to n^m.

$$T(n) \in \begin{cases} \Theta(n^\alpha) & \text{if } f(n) \in O(n^{\alpha-\epsilon}); \\ \Theta(f(n)\log n) & \text{if } f(n) \in \Theta(n^\alpha); \\ \Theta(f(n)) & \text{if } f(n) \in \Omega(n^{\alpha+\epsilon}) \text{ and } af(n/b) \leq cf(n) \\ & \text{for some } c < 1 \text{ and } n \text{ large enough.} \end{cases}$$

The general form can be of course applied to recurrences in the form $T(n) = aT(\frac{n}{b}) + cn^m$ but with the restricted form it easier to identify the exact order of a recurrence

$$T(n) \in \begin{cases} \Theta(n^\alpha) & \text{if } a > b^m. \\ \Theta(n^m \log n) & \text{if } a = b^m; \\ \Theta(n^m) & \text{if } a < b^m; \end{cases}$$

Example 4

Use the general Master Theorem to give the exact order of the recurrence

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

Here $a = 2$, $b = 2$ and $m = 1$, and $f(n) = n$.

Is $n \in O(n^{\log_b a - \epsilon}) = O(n^{\log_2 2 - \epsilon}) = O(n^{1-\epsilon})$. No

Is $n \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$. Yes.

Therefore the case $T(n) \in \Theta(f(n) \log n)$ applies, $T(n) \in \Theta(n \log n)$

Example 5

Use the general Master Theorem for the recurrence

$$T(n) = 6T(\frac{n}{4}) + n^2$$

Here $a = 6$, $b = 4$ and $m = 2$, and $f(n) = n^2$.

Is $n^2 \in O(n^{\log_b a - \epsilon}) = O(n^{\log_4 6 - \epsilon}) = O(n^{\approx 1.29 - \epsilon})$. No

Is $n^2 \in \Theta(n^{\log_b a}) = \Theta(n^{\log_4 6}) = \Theta(n^{\approx 1.29})$. No

Is $n^2 \in \Omega(n^{\log_b a}) = \Omega(n^{\log_4 6 + \epsilon})$. Yes. Does $af(n/b) < cf(n) = 6(\frac{n}{4})^2 < .9n^2$, it is true for any value of $n \geq 4$ and $c = .9$.

Therefore case $\Theta(f(n))$ applies, $T(n) \in \Theta(n^2)$

Example 6

Use the general Master Theorem to give the exact order of the following recurrence

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

Here $f(n) = \log\log n$. $f(n) \in O(n^{\log_2 2-\epsilon}) = O(n^{1-\epsilon})$, because $\log\log n < \log n$, also $\log n$ grow slower than n^c for any value of $c > 0$, therefore $\log\log n \in O(n^{1-\epsilon})$

$f(n) \notin \Omega(n^{\log_b a+\epsilon})$ as we cannot bound $\log\log n$ below with $n^{1.\epsilon}$, therefore $f(n) \notin \Theta(n^{\log_2 2})$.

Therefore, as the case $f(n) \in O(n^{\log_2 2-\epsilon})$ applies, $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

Example 7

Use the general Master Theorem to give the exact order of the following recurrence

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

Here $f(n) = \sqrt{n}$. Note that $\sqrt{n} = \sqrt[2]{n^1} = n^{\frac{1}{2}}$

Here $a = 2$ and $b = 4$. We want to know in which order $f(n) = \sqrt{n}$ is with respect to $n^{\log_b a} = n^{\log_4 2}$

It does happen that $n^{\log_4 2} = n^{\frac{1}{2}}$ since $\log_4 2 = \frac{\log_2 2}{\log_2 4} = \frac{1}{2}$

Therefore, $\sqrt{n} \in O(n^{\log_4 2})$ and $n^{\log_4 2} \in O(\sqrt{n})$, thus $\sqrt{n} \in \Theta(n^{\log_4 2})$

We conclude that $T(n) = \Theta(\sqrt{n} \log n)$

Example 8

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

Here $f(n) = \frac{n}{\log n}$.

This recurrence does not satisfied the master theorem, because the difference between $f(n)$ and $n^{\log_b a}$ is not a polynomial.

The difference can be expressed as follow :

$$\frac{n/\log n}{n^{\log_3 3}} = \frac{n}{n \log n} = \frac{1}{\log n}$$

.

Thus, the difference between $f(n)$ and $n^{\log_3 3}$, i.e. $\frac{1}{\log n}$, is such that $\frac{1}{\log n} < n^{\epsilon}$ for any $\epsilon > 0$, the difference is smaller than any polynomial

Exercise

Not all the recurrences can be solved using the Master Theorem. Among the following recurrences which one you think can be solved using the Master Theorem ?

a) $T(n) = 2T(n/2) + n^4$

b) $T(n) = T(7n/10) + n$

c) $T(n) = 16T(n/4) + n^2$

d) $T(n) = 7T(n/3) + n^2$

e) $T(n) = 7T(n/2) + n^2$

f) $T(n) = 3T(n/4) + \sqrt{n}$

g) $T(n) = T(n-2) + n^2$

h) $T(n) = 4T(n/3) + n \log n$

i) $T(n) = 3T(n/3) + \frac{n}{\log n}$

k) $T(n) = 4T(n/2) + n^2\sqrt{n}$

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

m) $T(n) = 2T(n/2) + \frac{n}{\log n}$

n) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

o) $T(n) = T(n-2) + \log n$

Solving recurrence relations : Substitution method

The substitution method consists of two steps :

1. Guess the form of the closed form.
2. Use mathematical induction to show the guess is correct.

It can be used to obtain either upper ($O()$) or lower bounds ($\Omega()$) on a recurrence.

A good guess is vital when applying this method. If the initial guess is wrong, the guess needs to be adjusted later.

Solve the recurrence $T(n) = T(n-1) + 2n - 1$, $T(0) = 0$

Form a guess using "forward substitution" :

n	0	1	2	3	4	5
$T(n)$	0	1	4	9	16	25

Guess is $T(n) = n^2$. Proof by induction :

Base case $n = 0$: $T(0) = 0 = 0^2$
Induction step : Assume the inductive hypothesis is true for $n = n - 1$.
We have

$$\begin{aligned} T(n) &= T(n-1) + 2n - 1 \\ &= (n-1)^2 + 2n - 1 \text{ (Induction hypothesis)} \\ &= n^2 - 2n + 1 + 2n - 1 \\ &= n^2 \end{aligned}$$

Solve the recurrence $T(n) = T(\lfloor \frac{n}{2} \rfloor) + n$, $T(0) = 0$
Guess using forward substitution :

n	0	1	2	3	4	5	8	16	32	64
$T(n)$	0	1	3	4	7	8	15	31	63	127

Guess : $T(n) \leq 2n$. Proof by induction :

Base case $n = 0$: $T(n) = 0 \leq 2 \times 0$
Induction step : Assume the inductive hypothesis is true for some $n = \lfloor \frac{n}{2} \rfloor$. We have

$$\begin{aligned} T(n) &= T(\lfloor \frac{n}{2} \rfloor) + n \\ &\leq 2\lfloor \frac{n}{2} \rfloor + n \text{ (Induction hypothesis)} \\ &\leq 2(n/2) + n \\ &= 2n \\ &\in O(n) \end{aligned}$$

Solve $T(n) = 2T(n/2) + n$, $T(1) = 1$

Since the input size is divided by 2 at each recursive call, we can guess that $T(n) \leq cn \log n$ for some constant c (that is, $T(n) = O(n \log n)$)

Base case : $T(1) = 1 = n \log n + n$ (note : we need to show that our guess holds for some base case (not necessarily $n = 1$, some small n is ok).
Induction step : Assume the inductive hypothesis holds for $n/2$ (n is a power of 2) : $T(n/2) \leq c\frac{n}{2} \log \frac{n}{2}$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2(c\frac{n}{2} \log \frac{n}{2}) + n \\ &= cn \log \frac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \end{aligned}$$

Which is true if $c \geq 1$

Exercise 1

Assume that the running time $T(n)$ satisfies the recurrence relation

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

Show that $T(n) \in O(n \log n)$ if $T(0) = 0$ and $T(1) = 1$.

Since the guess is already given here, you only need to prove by induction this guess is correct.

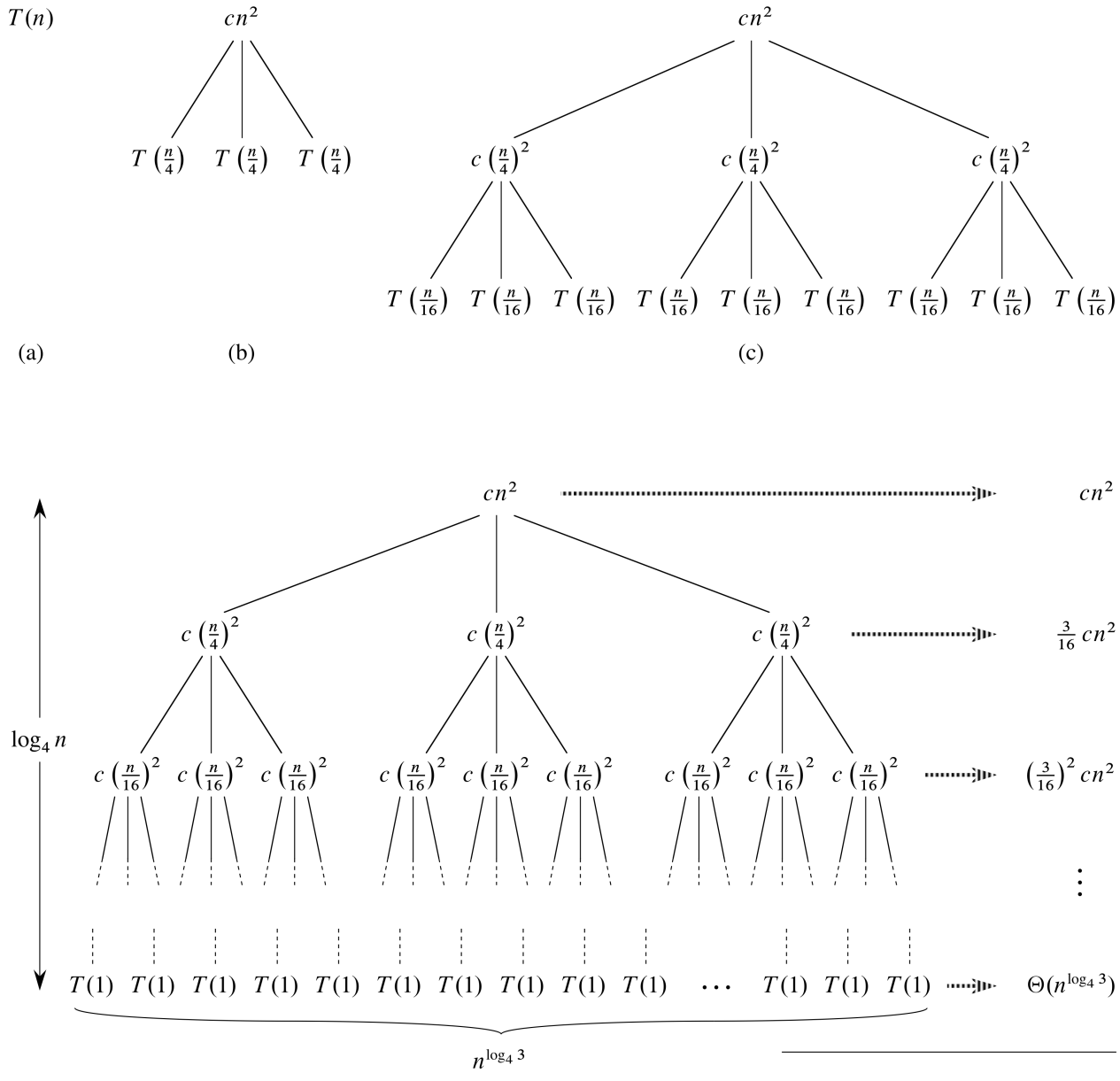
Solving recurrence relations : Recursion tree method

To analyze the recurrence by using the recursion tree, we will

- draw a recursion tree with cost of single call at each node
- find the running time at each level of the tree by summing the running time of each call at that level
- find the number of levels
- Last, the running time is sum of costs in all nodes

Example of a recurrence relation :

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2.$$


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$
(a)
(b)
(c)
cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $\cdots$ $T(1)$ $T(1)$ $T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
(d)
Total: $O(n^2)$

Analysis

Number of levels in the tree :

The number of levels in the tree depends by how much subproblem sizes decrease

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2.$$

In this example, the subproblem sizes decrease by a factor of 4 at each new level

The last level is where the size of each subproblem $= 1$, i.e. when $n/4^i = 1$

Therefore the number of levels is $i = \log_4 n$.

Analysis

Number of nodes at each level of the tree :

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2.$$

From the recurrence we can see the number of nodes at level i is 3 times the number of nodes at level $i - 1$

In terms of level 0, the number of nodes at level $i \geq 0$ is 3^i

- level $0 = 3^0 = 1$
- level $1 = 3^1 = 3$
- $\vdots$
- level $\log_4 n = 3^{\log_4 n} = n^{\log_4 3}$ (using the rule $a^{\log b} = b^{\log a}$)

Running time

According to the recurrence relation the number of basic operations executed at level 0 is cn^2 (given the input size $= n$)

At level 1, the input size of each subproblem is $\lfloor \frac{n}{4} \rfloor$, therefore the number of basic operations executed by each subproblem is $c(\lfloor \frac{n}{4} \rfloor)^2$

Assuming n is a power of 2, the number of basic operations is $c(\frac{n}{4})^2$. Since there are 3 subproblems, the running time is the sum of the running time of each node, i.e. $c(\frac{3n}{4})^2$ or $\frac{3}{16}cn^2$

At level i, the running time is also the sum of the running time of each node at level i, i.e. $(\frac{3}{16})^i cn^2$

Running time

As the number of levels is $\log_4 n$, the summation of the running time of each level is

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{n}{16})^{\log_4(n-1)} cn^2 + cn^{\log_4 3} \\ &= cn^2 \left[1 + \frac{3}{16} + (\frac{3}{16})^2 + \cdots + (\frac{3}{16})^{log_4(n-1)}\right] + cn^{\log_4 3} \\ &= O(n^2) \end{aligned}$$

The last step is a decreasing geometric series (each terms multiplied by $\frac{1}{3}$), which converges to $\frac{16}{13}$

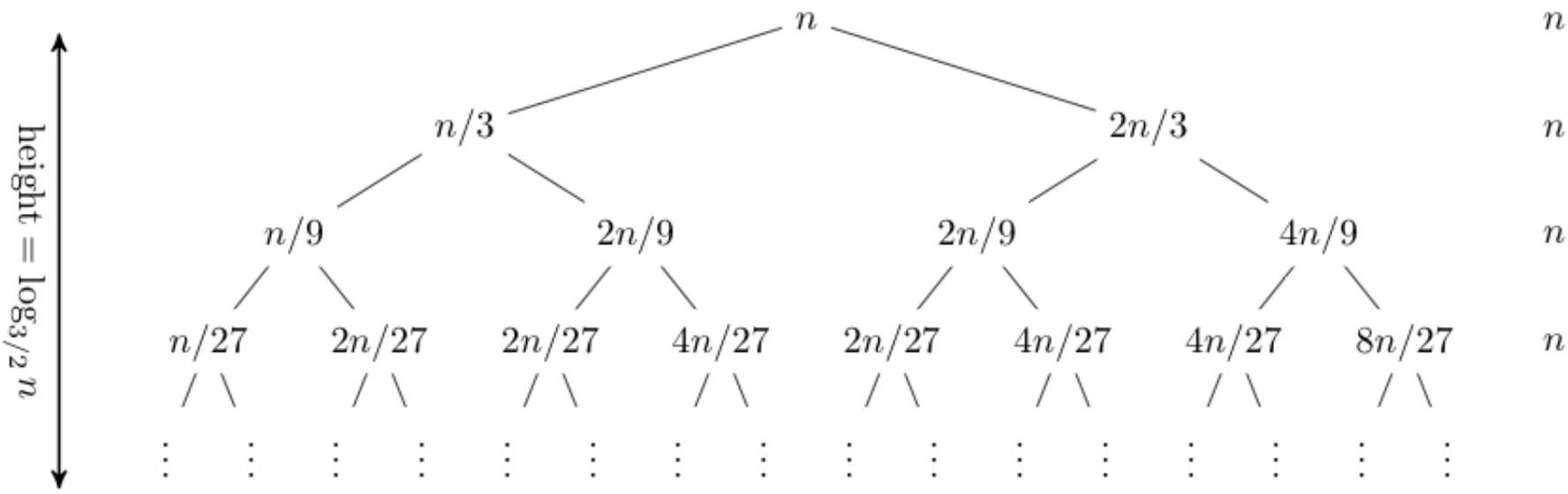
Exercise 2

Use the recursion tree method to find a good asymptotic upper bound on the recurrence $T(n) = 2T(n/2) + n^2$
The number of terms in the summation = number of levels in the tree = $log_2 n$. More precisely,

$$\begin{aligned} T(n) &= n^2 + \frac{1}{2}n^2 + \frac{1}{4}n^2 + \cdots + \frac{1}{2^{\log n}}n^2 \\ &= n^2\left[1 + \frac{1}{2} + (\frac{1}{4}) + \cdots + \frac{1}{2^{\log n}}\right] \\ &= O(n^2) \end{aligned}$$

Exercise 3

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/3) + T(2n/3) + n$.

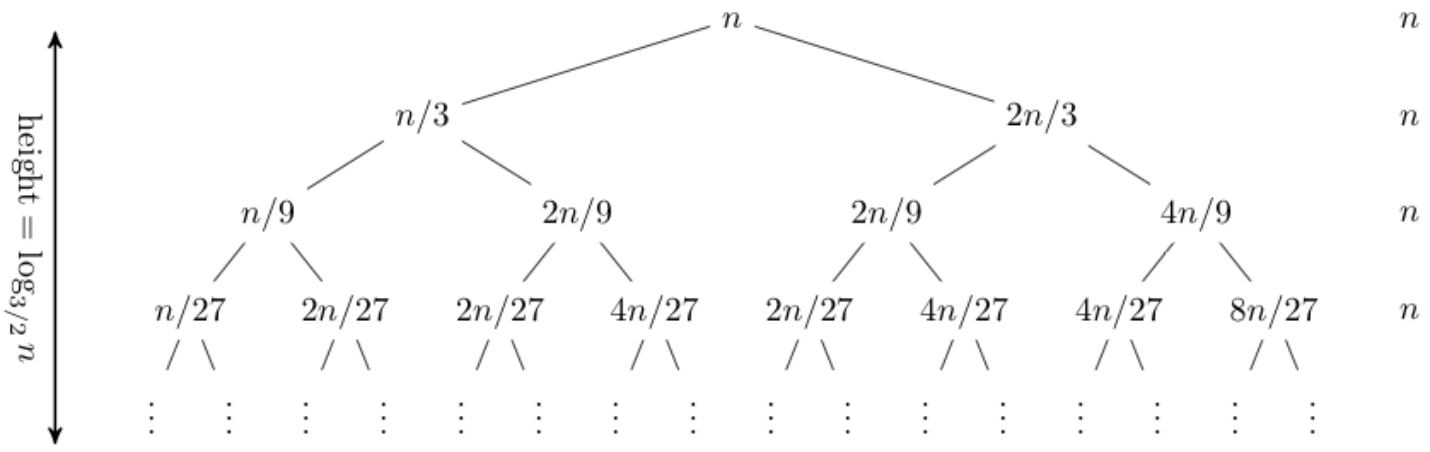


Each node at level i expands into 2 nodes at level $i + 1$

Exercise 3 : number of levels

$$T(n) = T(n/3) + T(2n/3) + n$$

.



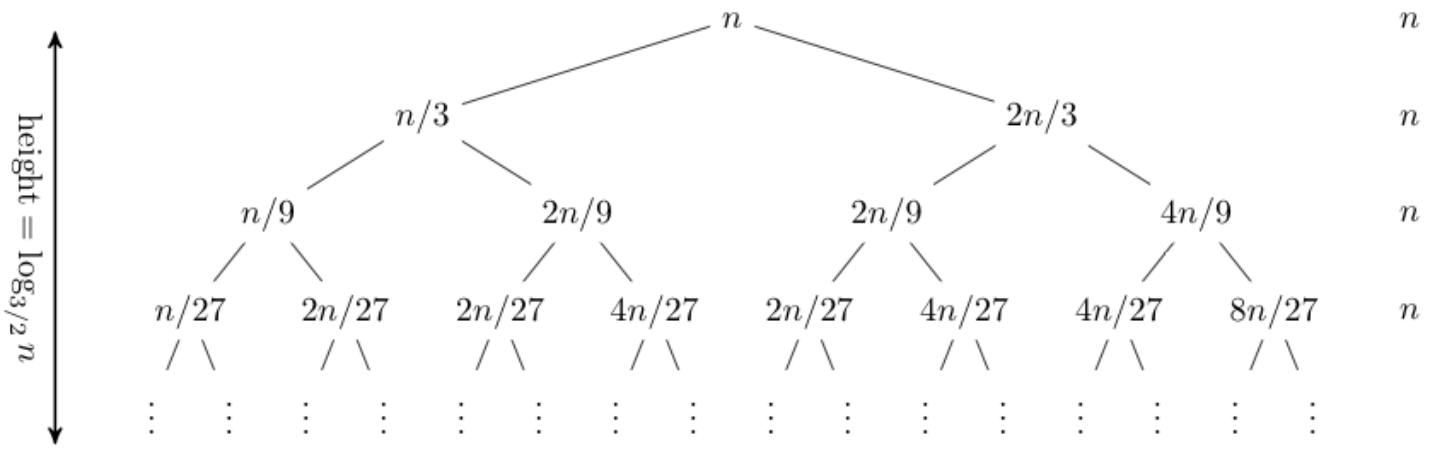
The input size of the node on the left is $\frac{n}{3}$, while the size of the other node is $\frac{3n}{2}$

The input size decreases much faster on the left than on the right side

Exercise 3 : number of levels

$$T(n) = T(n/3) + T(2n/3) + n$$

.



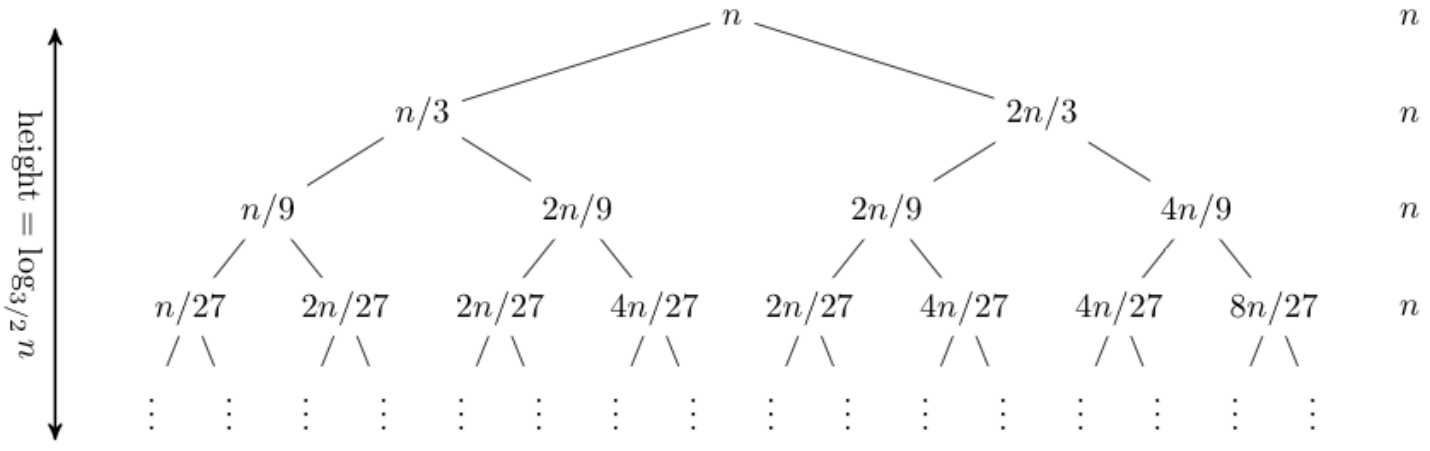
The number of levels of the leftmost path (the shortest path) is $\log_3 n$

The number of levels on the rightmost path (the longest one) is $\log_{3/2} n$ (as $\frac{2n}{3} = \frac{n}{3/2}$)

The number of levels in the other branches are a mix of these two extremes

Exercise 3 : running time

$$T(n) = T(n/3) + T(2n/3) + n$$



The longest branch of the tree has $\log_{3/2} n$ levels

At each level n basic operations are executed

$n \times \log_{3/2} n$ is in $O(n \log n)$ as logs from different bases differ only by a constant multiplicative factor

Solving $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Using the recursion tree method :

Level 0 of the tree has 1 node which does $\Theta(n)$ computation work

Level 1 of the tree has $\sqrt{n}$ nodes, each node does $\sqrt{n}$ work, thus the sum of the work done at level 1 $\sqrt{n} \times \sqrt{n} = n$

At level 2, each node x from level 1 has $\sqrt{\sqrt{n}}$ children, each doing $\sqrt{\sqrt{n}}$ work, so the work done by the children of x is $\sqrt{\sqrt{n}} \times \sqrt{\sqrt{n}} = \sqrt{n}$. There are $\sqrt{n}$ nodes at level 1, each has $\sqrt{n}$ children, the total computation done at level 2 is $\sqrt{n} \times \sqrt{n} = n$.

How many time can we $\sqrt{n}$ before we arrive to a base case where the recursion stop, for example base case equal 2 ?

- Let n be a power of 2, for example $n = 2^{\log n}$. $\sqrt{n} = 2^{\frac{\log n}{2^1}}$. Assume we stop the recursion once $2^{\frac{\log n}{2^k}} = 2$, what is k : $\log 2^{\frac{\log n}{2^k}} = \log 2 \Rightarrow \frac{\log n}{2^k} = 1$, $\log n = 2^k$, $\log \log n = k$

The number of time we can $\sqrt{n}$ is $\log \log n$. Example, $n = 256$: $\sqrt{256} = 16$, $\sqrt{16} = 4$, $\sqrt{4} = 2$. 3 times is $\log \log 256$.

Thus, if each level does $\Theta(n)$ work, and there are $\log \log n$ level, the solution to the recurrence $T(n) = \sqrt{n}T(\sqrt{n}) + n$ is $n \log \log n$

Solving $T(n) = \sqrt{n}T(\sqrt{n}) + n$

By change of variable (Brassard p. 130) where n is replaced by 2^k.
Then $\sqrt{n} = 2^{k/2}$.

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \tag{3}$$
$$T(2^k) = 2^{k/2}T(2^{k/2}) + 2^k \tag{4}$$

dividing by 2^k

$$\frac{T(2^k)}{2^k} = \frac{2^{k/2}T(2^{k/2})}{2^k} + 1 \tag{5}$$
$$= \frac{T(2^{k/2})}{2^{k/2}} + 1 \tag{6}$$

Let $y(k) = \frac{T(2^k)}{2^k}$, the new recurrence $y(k) = y(\frac{k}{2}) + 1$ can be solved using the Master theorem, where $a = 1$; $b = 2$; $m = 0$. This is case $a = b^m$,
$y(k) \in \Theta(k^m \log k) = \Theta(\log k)$

Since $y(k) = \frac{T(2^k)}{2^k}$, $T(2^k) = 2^k y(k)$. Assume $k = \log n$, given that $n = 2^k$
$T(2^k) = 2^k \log k \Rightarrow T(n) = n \log\log n \Rightarrow T(n) \in \Theta(n \log\log n)$

Solving $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Using the "backward" substitution method.

$$
\begin{aligned}
T(n) &= \sqrt{n}T(\sqrt{n}) + n \\
&= n^{\frac{1}{2}}\, T(n^{\frac{1}{2}}) + n \\
T(n^{\frac{1}{2}}) &= n^{\frac{1}{2}}(n^{\frac{1}{2^2}}\, T(n^{\frac{1}{2^2}}) + n^{\frac{1}{2}}) + n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}}\, T(n^{\frac{1}{2^2}}) + n^{\frac{1}{2}+\frac{1}{2}}) + n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}}\, T(n^{\frac{1}{2^2}}) + 2n \\
T(n^{\frac{1}{2^2}}) &= n^{\frac{1}{2}+\frac{1}{2^2}}(n^{\frac{1}{2^3}}\, T(n^{\frac{1}{2^3}}) + n^{\frac{1}{2^2}}) + 2n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}}\, T(n^{\frac{1}{2^3}}) + n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^2}} + 2n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}}\, T(n^{\frac{1}{2^3}}) + 3n \\
&\vdots \\
T(n^{\frac{1}{2^{k-1}}}) &= n^{\sum_{i=1}^{k}\frac{1}{2^i}}\, T(n^{\frac{1}{2^k}}) + kn
\end{aligned}
$$

Assume the last value of $n^{\frac{1}{2^k}} = 2$ (i.e. $n = 2$) and $T(2) = 2$.

Solving $T(n) = \sqrt{n}T(\sqrt{n}) + n$

$$
\begin{aligned}
T(n) &= \sqrt{n}T(\sqrt{n}) + n \\
&= n^{\frac{1}{2}}\, T(n^{\frac{1}{2}}) + n \\
T(n^{\frac{1}{2}}) &= n^{\frac{1}{2}}\,(n^{\frac{1}{2^2}}\, T(n^{\frac{1}{2^2}}) + n^{\frac{1}{2}}) + n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}}\, T(n^{\frac{1}{2^2}}) + n^{\frac{1}{2}+\frac{1}{2}}) + n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}}\, T(n^{\frac{1}{2^2}}) + 2n \\
T(n^{\frac{1}{2^2}}) &= n^{\frac{1}{2}+\frac{1}{2^2}}\,(n^{\frac{1}{2^3}}\, T(n^{\frac{1}{2^3}}) + n^{\frac{1}{2^2}}) + 2n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}}\, T(n^{\frac{1}{2^3}}) + n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^2}} + 2n \\
&= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}}\, T(n^{\frac{1}{2^3}}) + 3n \\
T(n^{\frac{1}{2^{k-1}}}) &= n^{\sum_{i=1}^{k}\frac{1}{2^i}}\, T(n^{\frac{1}{2^k}}) + kn
\end{aligned}
$$

$$
\begin{aligned}
n^{\frac{1}{2^k}} &= 2 \\
\frac{1}{2^k}\log n &= \log 2 \\
\log n &= 2^k \\
\log\log n &= k\log 2 \\
\log\log n &= k
\end{aligned}
$$

Thus the guessed recurrence relation look like this :

$$
\begin{aligned}
T(n) &= n^{\sum_{i=1}^{k}\frac{1}{2^i}}\, T(n^{\frac{1}{2^k}}) + kn \\
&= n^{\sum_{i=1}^{\log\log n}\frac{1}{2^i}}\, T(n^{\frac{1}{2^{\log\log n}}}) + n\log\log n \\
&= 1 - \frac{1}{\log n}\, T(n^{\frac{1}{2^{\log\log n}}}) + n\log\log n \\
T(n) &\in \Theta(n\log\log n)
\end{aligned}
$$

Homogeneous linear recurrences

Definition (Homogeneous linear recurrence)

A homogeneous linear recurrence satisfies the following equation

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \tag{7}$$

where the a_i are constants, t_i are the values we are looking for, and initial conditions are given by t_i for $0 \leq i \leq k-1$ or $1 \leq i \leq k$.

Example : The Fibonacci recurrence :

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

with initial conditions $f_0 = 0$ and $f_1 = f_2 = 1$.

Solution form to Equation (7)

Definition (Characteristic polynomial)

The polynomial of degree k in x :

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k \tag{8}$$

is called the characteristic polynomial of the recurrence relation (7).

Example : The characteristic polynomial of the Fibonacci recurrence is $p(x) = x^2 - x - 1$.

Solutions to homogeneous linear recurrences

Theorem (Case when characteristic polynomial has all distinct roots)

If the polynomial $p(x)$ of the recurrence equation (7) has exactly k distinct roots, says $r_1, r_2, \ldots, r_k$, i.e.

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

then the general solution to the recurrence equation (7) is given by a linear combination as follows

$$t_n = \sum_{i=1}^{k} c_i r_i^n, \tag{9}$$

where constants c_i are found based on the initial conditions of (7).

Example : Solve the recurrence

$$t_n = \begin{cases} 0 & \texttt{if } n = 0 \\ 7 & \texttt{if } n = 1 \\ 5t_{n-1} - 6t_{n-2} & \texttt{otherwise} \end{cases}$$

- The characteristic polynomial of this recurrence relation is

$$p(x) = x^2 - 5x + 6 = (x-2)(x-3)$$

- The solution to the recurrence relation hence is $t_n = \sum_{i=1}^{k} c_i r_i^n = c_1 3^n + c_2 2^n$
- Using the initial condition $t_0 = 0$, we get

$$0 = t_0 = c_1 3^0 + c_2 2^0 = c_1 + c_2.$$

- Using the initial condition $t_1 = 7$, we get

$$7 = t_1 = c_1 3^1 + c_2 2^1 = 3c_1 + 2c_2.$$

- Solving c_1, c_2 from above constraints we get $c_1 = 7$, and $c_2 = -7$, and hence, $t_n = 7 \times 3^n - 7 \times 2^n$

Example 2

Solve the recurrence

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

Some roots repeat I

Theorem (Case when characteristic polynomial has multiple roots)

Let $p(x)$ be the characteristic polynomial of the recurrence equation (7). Let $r_1, r_2, \ldots, r_l$ be the l roots to $p(x) = 0$, and let $m_1, m_2, \ldots, m_l$ be the multiplicities of those roots. Then the general form of the solutions is :

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

Example : Solve the recurrence :

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

- The recurrence is $t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$

Some roots repeat II

- The characteristic polynomial is

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

- The roots are $r_1 = 1$ of multiplicity 1 and $r_2 = 2$ of multiplicity 2.
- The general form of the solutions is :

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

- The initial conditions $n = 0, 1, 2$ read as :

$$\begin{array}{llll} c_1 & +c_2 & & = 0 \\ c_1 & +2c_2 & +2c_3 & = 1 \\ c_1 & +4c_2 & +8c_3 & = 2 \end{array}$$

- Solving this system of linear equations, we obtain $c_1 = -2, c_2 = 2, c_3 = -\frac{1}{2}$
- The solution is $t_n = 2^{n+1} - n2^{n-1} - 2$

Summary for solving homogeneous linear recurrence

For solving homogeneous linear recurrence, we follows steps :

- Write down the characteristic polynomial $p(x) = x^k + c_1 x^{k-1} + \cdots + c_k$;
- Find roots with multiplicities to $p(x) = 0$. Assume

$$p(x) = \prod_{i=1}^{l} (x - r_i)^{m_i}.$$

- Form the general solution

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

- Use the k initial conditions to solve for the c_{ij}.

Non-homogeneous linear equations

We have a recurrence like the following :

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n) \tag{10}$$

where the right-hand side $b^n p(n)$ is such

- b is a constant
- $p(n)$ is a polynomial in n of degree d

In that case, we write the characteristic polynomial as

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

and solve for this characteristic polynomial as the method mentioned above.

Example 1

Solve the recurrence $t_n - 2t_{n-1} = 3^n$.

- The right hand side $b^n p(n)$ is 3^n where $b^n = 3^n$ and $p(n) = 1$ a polynomial of degree 0.
- The characteristic equation of the left hand side is $x - 2$, the characteristic polynomial of the recurrence is $(x - 2) \times (x - 3)$.
- The roots of this polynomial are obvious : $x = 2, x = 3$.
- All the solutions of this recurrence are of the form $t_n = c_1 2^n + c_2 3^n$. We need the initial conditions of the recurrence to find the constants c_1 and c_2 and the unique solution to this recurrence given its initial conditions.

No initial conditions are given for the recurrence $t_n - 2t_{n-1} = 3^n$. But we can find the initial condition for t_1 in function of t_0.

If we express the recurrence as we did for the other ones, we have

$$t_n = 2t_{n-1} + 3^n$$

then $t_1 = 2t_{n-1} + 3 = 2t_0 + 3$.

We can now write a system of linear equations in two unknowns c_1 and c_2

$$\begin{array}{rcrcl} 3^0 c_1 & + & 2^0 c_2 & = & t_0 \\ 3^1 c_1 & + & 2^1 c_2 & = & 2t_0 + 3 \end{array}$$

$c_1 = t_0 - 3$ and $c_2 = 3$. The solution is $t_n = (t_0 - 3)2^n + 3^{n+1}$.

Since in $t_n = (t_0 - 3)2^n + 3^{n+1}$ the dominant term is 3^{n+1} we conclude that $t_n \in \Theta(3^n)$.

Another way to find the constants c_1, c_2

Rather than solving a system of linear equations to find the constants c_1, c_2, we can substitute the general form of the solutions $t_n = c_1 2^n + c_2 3^n$ in the original recurrence $t_n - 2t_{n-1} = 3^n$:

$$\begin{array}{rcl} 3^n & = & t_n - 2t_{n-1} \\ 3^n & = & (c_1 2^n + c_2 3^n) - 2(c_1 2^{n-1} + c_2 3^{n-1}) \\ 3^n & = & c_2 3^{n-1} \\ \frac{3^n}{3^{n-1}} & = & c_2 \frac{3^{n-1}}{3^{n-1}} \\ 3 & = & c_2 \end{array}$$

By substitution in the first linear equation of c_2 by 3, we have

$$\begin{array}{rcl} c_1 + 3 & = & t_0 \\ c_1 & = & t_0 - 3 \end{array}$$

Example 2

Solve the recurrence

$$t_n = \begin{cases} 0 & \texttt{if } n = 0 \\ 2t_{n-1} + 1 & \texttt{otherwise} \end{cases}$$

- This recurrence can be rewritten as $t_n - 2t_{n-1} = 1$.
- The characteristic equation of the left hand side is $x - 2$, the right hand side $b^n p(n)$ is such that $b^n = 1$ and $p(n) = 1$.
- The characteristic polynomial is then $(x - 2)(x - 1)$ with roots 1 and 2.
- All the solutions of this recurrence are of the form $t_n = c_1 1^n + c_2 2^n$.

The initial condition for t_0 is given : $t_0 = 0$.

The initial condition for t_1 has to be calculated from the recurrence : $t_1 = 2t_0 + 1 = 1$.

We solve the system of linear equations :

$$\begin{array}{rcrl} c_1 & + & c_2 & = 0 \\ c_1 & + & 2c_2 & = 1 \end{array}$$

and we obtain $c_1 = -1$ and $c_2 = 1$.

$t_n = 2^n - 1$.

The dominant term in $t_n = 2^n - 1$ is 2^n, therefore $t_n \in \Theta(2^n)$.

A few relations to remember

$\log_a(xy) = \log_a x + \log_a y$
$\log_a \frac{x}{y} = \log_a x - \log_a y$
$\log_a(x^r) = r \log_a x$
$\log_a a = 1; \log_a 1 = 0$
$\log_a \frac{1}{x} = -\log_a x$
$\log_a x = \frac{\log x}{\log a}$
$\log_b x = \frac{log_2 x}{\log_2 b}$ changing to log base b
$x^{\frac{a}{b}} = \sqrt[b]{x^a}$
$x^{1/b} = z$ means $z^b = x$; $x^{a/b} = z$ means $z^b = x^a$
$c^{\log(a)} = a^{\log(c)}$: take log of both sides.
$(b^a)^c = b^{ac}$
$b^a b^c = b^{a+c}$
$b^a / b^c = b^{a-c}$