Advanced Programming Techniques

PART II

Algorithm Analysis Tools

Beniamin Bogosel

Ecole Polytechnique
Department of Applied Mathematics

Algorithm analysis

Two questions to ask when facing an algorithm

- Is my algorithm correct?
- Is my algorithm efficient?

⋆ **Problem:** general question like sorting an array

⋆ **Instance of a problem:** one particular case: sort the array $[8, 2, 4, 3, 1]$.

An algorithm is correct for a problem if it produces a correct solution for all instances of the problem!

Example

Consider the algorithm A which permutes the first two elements in an array.
Algorithm A is correct for the instance: Sort the array $[2, 1, 3, 4]$, but is not a sorting algorithm!

How to test for correctness?

- Testing
  - implement the algorithm
  - test it on all instances (assuming we can do this)
  - difficult to "prove" there's no bug
- Have a mathematical formal proof:
  - it is not necessary to implement the algorithm to know it is correct
  - not perfect either...
- In practice: a mix of the two.
- Tools:
  - iterative algorithms: Hoare triplets, loop invariants
  - recursive algorithms: induction proofs

A quote by Dijkstra

A good programmer knows that an algorithm is correct before implementing it.

- A relation between the variables which is valid at a certain point in the execution
- Consider two conditions:
  - P: conditions verified by a valid input for the algorithm
  - Q: conditions verified by the output if the algorithm is correct
- The algorithm is correct if the triplet P code Q is true (called Hoare triplet).

Example

$$\{x \geq 0\} y = \textsc{Sqrt}(x) \{y == x^2\}.$$

Correcting a series of instructions

$\star$ in practice algorithms have multiple instructions

1: $\{P\}$
2: $S1$
3: $S2$
4: ...
5: Sn
6: $\{Q\}$

$\star$ to check correctness it is useful to insert intermediary assertions $P_1, ..., P_{n-1}$ describing variables at each step in the program.

$\star$ then check that triplets $\{P\}S1\{P_1\}$, $\{P_1\}S2\{P_2\}$, ...,$\{P_{n-1}\}Sn\{Q\}$ are correct.

$\star$ different types of instructions: assign value to a variable, conditions, loops

Correcting value assignments and conditions

⋆ value assignments: straightforward, assert that the value changed the way we want in an assignment

⋆ conditions

$\{P\}$
if B **then**
    $C1$
else
    $C2$
$\{Q\}$

To prove correctness show that the following triplets are true

- $\{P\&B\}C1\{Q\}$
- $\{P\&\text{ non-}B\}C2\{Q\}$

Basically: test that the if statement does what it's supposed to do!

Correcting loops

$\{P\}$
while B **do**
    CODE
$\{Q\}$

$\{P\}$
INIT
$\{I\}$
while B **do**
    $\{I$ and $B\}$ CODE $\{I\}$
$\{I$ and non-$B\}$
$\{Q\}$

⋆ To prove that a loop does what it's supposed to do, find a Loop invariant I: a property that is valid throughout the loop

⋆ Prove that the property is preserved (by design)

⋆ Prove that the loop must finish! Termination function: for example, some function which is strictly decreasing and reaches zero at termination.

Example: Fibonacci-Iter

```
FIBONACCI-ITER(n)
    if n ≤ 1
        return n
    else
        pprev = 0
        prev = 1
        for i = 2 to n
            f = prev + pprev
            pprev = prev
            prev = f
        return f
```

⋆ Proposition: if $n \geq 0$ Fibonacci-Iter(n) outputs F_n.

```
FIBONACCI-ITER(n)
    {n ≥ 0} // {P}
    if n ≤ 1
        prev = n
    else
        pprev = 0
        prev = 1
        i = 2
        while (i ≤ n)
            f = prev + pprev
            pprev = prev
            prev = f
            i = i + 1
    {prev == F_n} // {Q}
    return prev
```

Add post and pre-conditions

Analyzing the condition

$\{n \geq 0 \text{ et } n \leq 1\}$
$prev = n$
$\{prev == F_n\}$

correct $(F_0 = 0, F_1 = 1)$

$\{n \geq 0 \text{ et } n > 1\}$
$pprev = 0$
$prev = 1$
$i = 2$
while $(i \leq n)$
    $f = prev + pprev$
    $pprev = prev$
    $prev = f$
    $i = i + 1$
$\{prev == F_n\}$

$\boxed{I = \{pprev == F_{i-2}, prev == F_{i-1}\}}$

Analyzing the loop

$\{n > 1\}$
$pprev = 0$
$prev = 1$
$i = 2$
$\{pprev == F_{i-2}, prev == F_{i-1}\}$

correct

$\{pprev == F_{i-2}, prev == F_{i-1}, i \leq n\}$
$f = prev + pprev$
$pprev = prev$
$prev = f$
$i = i + 1$
$\{pprev == F_{i-2}, prev == F_{i-1}\}$

correct

$\{pprev == F_{i-2}, prev == F_{i-1}, i == n + 1\}$
$\{prev == F_n\}$

correct

$$
\begin{aligned}
&i = 2\\
&\textbf{while } (i \le n)\\
&\qquad f = prev + pprev\\
&\qquad pprev = prev\\
&\qquad prev = f\\
&\qquad i = i + 1
\end{aligned}
$$

- Does the loop end?
- Termination function: $f = n - i + 1$
  - $i = i + 1$: f decreases strictly at every iteration
  - $i \le n$: implies $f = n - i + 1 > 0$.
- Therefore the algorithm is correct and finishes!

Another example: insertion sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Quick proof of correctness:

- **Loop invariant:** the subtable $A[1..j-1]$ contains the elements of the original table $A[1..j-1]$ sorted
- Invariant is preserved!
- the loop finishes

- may be difficult for some algorithms
- Generally the **algorithm is a consequence of the invariant** not the other way around
  - Fibonacci algorithm: We compute iteratively F_{i-1} and F_{i-2}
  - Insertion sort algorithm: We add the element j to the sorted sub-array containing the first $j-1$ elements **at the correct position**.
- Using a loop invariants is based on the general principle of **induction or recurrence proofs**
  - $P(0)$ is true
  - $P(i-1)$ implies $P(i)$
  - Termination when we reached the desired value $i=n$.

Classical example

Proposition: for every $n \geq 0$ we have

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

Proof:

- Base case: $n = 0$: $\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}$.
- Inductive case for $n \geq 1$:

$$\sum_{i=0}^{n} i = \sum_{i=0}^{n-1} i + n = \frac{(n-1)n}{2} + n$$
$$= \frac{n(n+1)}{2}.$$

- By induction/recurrence the property is valid for every $n \geq 0$.

Induction proofs can prove correctness of recursive algorithms

- Property to prove: the algorithm is correct for a given instance of the problem
- Order the instances of the problem by some "size" (array length, number of bits, some integer, etc)
- **Base case:** for induction = base case for recursion
- **Inductive case:** assume that recursive calls are correct and deduce that the current call is correct
- **Termination:** show that recursive calls only apply to sub-problems, finite number of calls (usually trivial, by construction)

Example: Fibonacci

FIBONACCI(n)

if $n \leq 1$

  return n

return FIBONACCI($n-2$) + FIBONACCI($n-1$)

Proposition: For every n `Fibonacci`(n) returns F_n Proof:

- Base case: for $n \in \{0, 1\}$ the function returns $F_n = 1$.
- Inductive case: Assuming FIBONACCI(m) returns F_m for $m < n$ we find that FIBONACCI(n) returns

$$F_{n-1} + F_{n-2} = F_n.$$

Example: Merge sort

```
MERGE-SORT(A, p, r)
if p < r
      q = ⌊(p+r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q+1, r)
      MERGE(A, p, q, r)
```

Proposition: For $1 \leq p \leq r \leq A.length$ Merge-Sort(A, p, r) sorts the sub-array $A[p..r]$.

Assuming that Merge is correct (to be proved using an invariant)

Example: Merge sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p+r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Proof:

- Basis case: for $r - p = 0$ merge sort ne modifie pas A et $A[p] = A[r]$ is sorted
- If $r - p > 0$ then $p - q$ and $r - q - 1$ are strictly smaller than $r - p$. The calls to MERGE-SORT for sub-arrays of smaller lengths are correct by **induction hypothesis**
- Supposing MERGE-SORT is correct, we find that MERGE-SORT(A, p, r) is correct.

Conclusions on Algorithm correction

⋆ Correctness proofs

- Iterative algorithms: **Invariant**
- Recursive algorithms: **Induction**

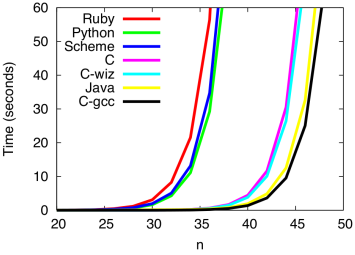
Algorithm performance

- Multiple ways of measuring efficiency:
  - program length (number of lines)
  - code simplicity
  - Memory space consumed
  - Computation time
  - number of elementary operations
- Computation time/number of operations
  - most relevant
  - quantifiable, easy to compare
- Memory usage is also relevant!

How to measure execution time?

Experimentally: (?)

- write a program and execute it for multiple instances of a data set
- Problems:
  - Computation time depends on implementation: CPU, OS, programming language, compiler, machine status, etc.
  - On what data should you test the algorithm?



(Carzaniga)

Cost for computing F_n in different Programming Languages

How to measure execution time?

On paper:

- Assume a machine model:
  - operations executed sequentially
  - Basic operations (addition, assignment, branching) take constant time
  - sub-routines: call time (constant)+ sub-routine execution (recursive computation)
- Computation time= sum all contributions corresponding to pseudo-code instructions

⋆ Execution time depends on inputs

⋆ Execution time is generally computed in term of some "size" for the entry

- length of an array
- some integer parameter

Analysis of insertion sort

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

- t_j number of iterations in the while loop
- Total execution time:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Different aspects

- Even for a fixed size, the complexity might differ from one instance to another
- Different ways of reasoning:
  - best case scenario
  - worst case scenario
  - average case
- Usually we use the **worst case scenario**
  - it gives an upper bound for the execution time
  - best case is not representative; average case is difficult to compute/interpret

Best case: the array is sorted in increasing order
⋆ the inner while loop condition is only tested once, $t_j = 1$.
⋆ the execution time is linear in n: $T(n) = an + b$.

Worst case: The array is sorted in a decreasing order: the inner loop is ran j times: $t_j = j$.
⋆ Then it can be seen that sums of the form $\sum_{i=1}^{n} = n(n+1)/2$ appear in the computation of $T(n)$, which gives

$$T(n) = an^2 + bn + c,$$

a quadratic function of n.

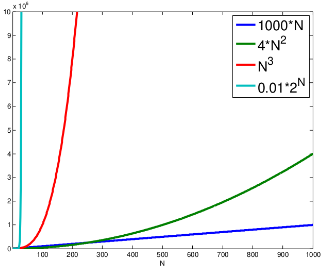
Asymptotic analysis

⋆ we are interested in the growth speed of $T(n)$ as n increases

⋆ The computation time $T(n)$ is simplified:

- Example: $T(n) = 10n^3 + n^2 + 40n + 800$
- $T(1000) = 10000104080 0$; $10n^3 = 10000000000 0$

⋆ ignoring the coefficient of the dominant term; asymptotically this does not change the relative order



⋆ Insertion sort: $T(n) = an^2 + bn + c \longrightarrow n^2$.

Why is it important to have this estimate?

- ⋆ assume elementary operations take one micro second
- ⋆ the computation time for different values of n can be estimated

T(n)	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
n	$10\mu s$	$0.1ms$	$1ms$	$10ms$
$400n$	$4ms$	$40ms$	$0.4s$	$4s$
$2n^2$	$200\mu s$	$20ms$	$2s$	$3.3m$
n^4	$10ms$	$100s$	~ 11.5 jours	317 années
2^n	$1ms$	4×10^{16} années	3.4×10^{287} années	$\ldots$

Why is it important?

- Maximum problem size that can be handled in a given time

$T(n)$	1 second	1 minute	1 hour
n	10^6	6×10^7	3.6×10^9
$400n$	2500	150000	9×10^6
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

- If m is the value that can be treated in a given time what becomes this value on a machine 256 more powerful?

$T(n)$	Time
n	$256m$
$400n$	$256m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m+8$

Asymptotic notations

⋆ Allow to characterize the growth of functions $f : \mathbb{N} \to \mathbb{R}_+$

⋆ three notations:

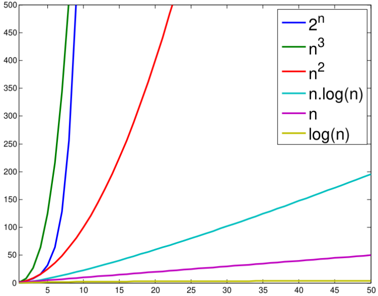
- (upper bounds) Big-O: $f(n) \in O(g(n))$ if $f(n) \leq Cg(n)$
- (lower bounds) Big-Ω: $f(n) \in \Omega(g(n))$ if $f(n) \geq Cg(n)$
- (lower and upper bounds) Big-Theta: $f(n) \in \Theta(g(n))$ if $f(n) \simeq g(n)$.

Examples

- $3n^5 - 16n + 2 \in O(n^5)? \in O(n)? \in O(n^{17})?$
- $3n^5 - 16n + 2 \in \Omega(n^5)? \in \Omega(n)? \in \Omega(n^{17})?$
- $3n^5 - 16n + 2 \in \Theta(n^5)? \in \Theta(n)? \in \Theta(n^{17})?$

⋆ Complexity classes:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^{a>1}) \subset O(2^n).$$



Some properties

- $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$
- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- Scalar multiplication: $f(n) \in O(g(n))$, $k \in \mathbb{R}_+$ then $kf(n) \in O(g(n))$
- Addition, max: $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then

$$f_1(n) + f_2(n) \in O(g_1(n) + g_2(n)), f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

- Product: $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then $f_1(n) \cdot f_2(n) \in O(g_2(n) \cdot g_2(n))$.

Algorithm complexity

- We use asymptotic notations to characterize the complexity
- We must specify what type of complexity: best case, worst case, average case
- The Big-O notation is the most used: in practice we say that an algorithm is $O(g(n))$ if $g(n)$ gives the best (smallest) possible complexity class

Complexity of a problem

- We say that a problem is $O(g(n))$ if there exists an algorithm $O(g(n))$ which can solve it
- We say that a problem is $\Omega(g(n))$ if every algorithm that solves it is at least $\Omega(g(n))$
- We say that a problem is $\Theta(g(n))$ if it belongs to both cases above

Example: The sorting problem

- The sorting problem is $O(n \log n)$
- We can easily show that the sorting problem is $\Omega(n)$
- We can show that, in fact, the sorting problem is $\Omega(n \log n)$.

Exercise: Show that the search for the maximum in an array is $\Theta(n)$.

The sorting problem is $\Omega(n)$

- Suppose there exists an algorithm better than $O(n)$ to solve the sorting problem
- This algorithm cannot iterate through all elements in an array, otherwise it would be $O(n)$
- Therefore there exists at least one element in the array which is not visited by the algorithm
- Therefore there are instances of arrays which will not be correctly sorted by this algorithm
- Therefore there does not exist an algorithm faster than $O(n)$ for the sorting problem.

Computing complexity in practice

Simple rules for iterative algorithms:

- Affectation, accessing an element in an array, arithmetic operation, function calls: $O(1)$
- Instruction IF-THEN-ELSE: $O($ max complexity of the two branches $)$
- Sequence of operation: the most costly operation (sum)
- Simple loop $O(nf(n))$ if the loop body costs $O(f(n))$
- Complete double loop $O(n^2 f(n))$ if the loop body costs $O(f(n))$
- Incremental loops: $i = 1..n$, $j = 1..i$: $O(n^2)$
- Loops with exponential increase $i \mapsto 2i \leq n$: $O(\log n)$.

Example

PREFIXAVERAGES(X)

- **input**: array X of size n
- **output**: array A of size n such that $A[i] = (\sum_{j=1}^{i} X[j])/i$ (average of the first i elements of X)

```
PREFIXAVERAGES(X)
for i = 1 to X.length
    a = 0
    for j = 1 to i
        a = a + X[j]
    A[i] = a/i
return A
```

```
PREFIXAVERAGES2(X)
s = 0
for i = 1 to X.length
    s = s + X[i]
    A[i] = s/i
return A
```

First variant: $\Theta(n^2)$, Second variant: $\Theta(n)$

More complex algorithms

- Applying the previous rules might lead to overestimating the complexity
- More "scientific" approach:
  - Detect an *analytic expression* for the number of executions of the basic operations $T(N)$ for a problem of "size" N
  - Conclude that the cost of the algorithm is $aT(N)$ where a is the constant cost of the basic operation
- The sorting example: the abstract operation is the comparison

- Usually leads to a recurrence relation
- Solving the recurrence relation is not necessarily trivial

Factorial and Fibonacci

FACTORIAL(n)

```
if n == 0 then
    return 1
return n· FACTORIAL(n − 1)
```

FIB(n)

```
if n ≤ 1 then
    return n
return FIB(n − 2)+FIB(n − 1)
```

$$\begin{aligned} T(0) &= c_0 \\ T(n) &= T(n-1) + c_1 \\ &= c_1 n + c_0 \end{aligned}$$

$$\implies T(n) \in \Theta(n).$$

$$\begin{aligned} T(0) &= c_0 \\ T(1) &= c_0 \\ T(n) &= T(n-1) + T(n-2) + c_1 \end{aligned}$$

$$\implies T(n) \in \Theta(1.61^n).$$

Merge sort

```
MERGE-SORT(A, p, q, r)
if p < r then
    q = ⌊(p+r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Recurrence:

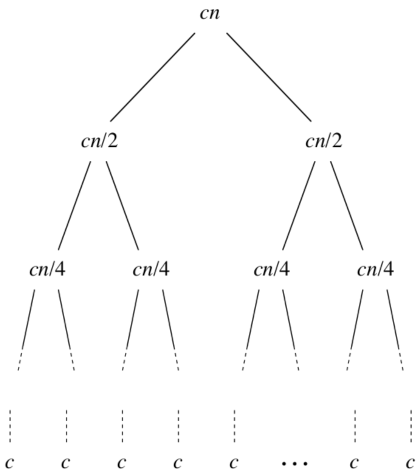
$$T(1) = c_1 \qquad T(1) = \Theta(1)$$

$$T(n) = 2T(n/2) + c_2 n + c_3 \qquad T(n) = 2T(n/2) + \Theta(n)$$

- Simplify the recurrence:

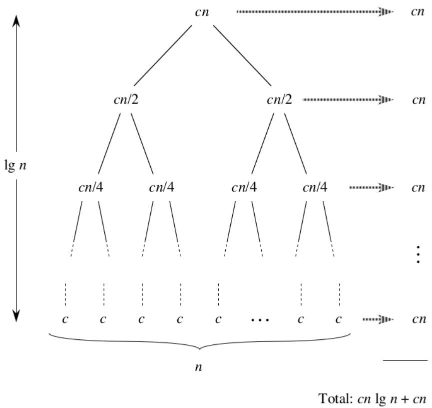
$$T(1) = c$$
$$T(n) = 2T(n/2) + cn$$

- Represent the recurrence graphically
- Sum the cost at every node



Analysis: merge-sort

- Each level costs cn
- Assume n is a power of 2 there are $\log_2 n + 1$ levels
- Total cost is $cn \log_2 n + cn \in \Theta(n \log n)$



Remarks

Limitation of asymptotic analysis

- Constants are important for problems of small sizes
  - Insertion sort is faster than merge sort for n small
- Two algorithms having the same complexity might behave differently

Space complexity:

- Same type of reasoning, same notations
- Bounded by the time complexity (why?)

Sums and recurrences

- Complexity analysis often involve computing sums and recurrences
- Recall some basic techniques

Examples

$\star$ $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
$\star$ $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
Technique:

$$\sum_{i=1}^{n} i^2 = an^3 + bn^2 + cn + d$$

- Identify coefficients a, b, c, d starting from some values of the sum
- Prove the result by induction.

$\star$ $\sum_{i=0}^{n-1} z^i = \frac{1-z^n}{1-z}$
$\star$ $\sum_{i=0}^{n-1} iz^i = \frac{z-(n+1)z^{n+1}+nz^{n+2}}{(1-z)^2}$.
$\star$ $S_n = \sum_{k=0}^{n} k2^k = (n-1)2^{n+1} + 2$ (appearing when studying the complexity of heap sort)
$\star$ other examples will be handled individually when they appear

Recurrences

- When dealing with recursive algorithm, recurrence relations will appear
- Examples:
  - Merge Sort:

$$T(1) = 0$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1 \text{ for } n > 1$$

  - Fibonacci:

$$T(1) = 0$$
$$T(n) = T(n-1) + T(n-2) + 2 \text{ for } n > 1$$

- Various types: linear, polynomial, divide and conquer, etc...

Methods...

- "guess" and prove by induction
- Replace and compute:

Merge sort:

$$T(1) = 0;\ T(n) = 2T(n/2) + n - 1.$$

$\star$ Pattern:

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + (n - 2^{i-1}) + (n - 2^{i-2}) + ... + (n - 2^0) \\ &= 2^i T(n/2^i) + in - 2^i + 1 \end{aligned}$$

$\star$ If $k = \log_2 n$ and $i = k$ then

$$\begin{aligned} T(n) &= 2^k T(n/2^k)_k n - 2^k + 1 \\ &= nT(1) n \log_2 n - n + 1 \\ &= O(n \log n) \end{aligned}$$

General theorem

Theorem

Consider the following recurrence

$$\begin{array}{ll} T(n) = c & \text{if } n < d \\ T(n) = aT(n/b) + f(n) & \text{if} n \geq d \end{array},$$

where $d \geq 1, a > 0, c > 0, b > 1$ and $f(n) \geq 0$ for $n \geq d$. Then:

1. If $f(n) \in O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$ then $T(n) \in O(n^{\log_b a})$
2. If $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{\log_b a} \log n)$.
3. If $f(n) \in O(n^{\log_b a+\varepsilon})$ for $\varepsilon > 0$ and there exists $\delta < 1$ such that $af(n/b) \leq \delta f(n)$ then $T(n) \in \Theta(f(n))$

Linear/divide and conquer recurrences

$$T_n = 2T_{n-1} + 1 \qquad T_n \sim 2^n$$
$$T_n = 2T_{n-1} + n \qquad T_n \sim 2 \cdot 2^n$$
$$T_n = 2T_{n/2} + 1 \qquad T_n \sim n$$
$$T_n = 2T_{n/2} + n - 1 \qquad T_n \sim n \log n T_n = T_{n-1} + T_{n-2} \qquad T_n \sim (1.61)^{n+1}$$

- Divide and conquer recurrences are generally polynomial
- Linear recurrences are exponential
- Generating smaller sub-problems is more important than reducing the non-homogeneous term

Comparing recurrences: number of sub-problems

Linear recurrences:

$$T_n = 2T_n + 1 \implies T_n \in \Theta(2^n)$$
$$T_n = 3T_n + 1 \implies T_n \in \Theta(3^n)$$

⋆ passing from 2 to 3 sub-problems increases the time exponentially

Divide and conquer recurrences:

$$T_1 = 0$$
$$T_n = aT_{n/2} + n - 1$$

The master theorem implies:

$$T_n = \begin{cases} \Theta(n) & \text{for } a < 2 \\ \Theta(n \log n) & \text{for } a = 2 \\ \Theta(n^{\log_2 a}) & \text{for } a > 2 \end{cases}$$

What we have seen

- Correcting algorithms: iterative (invariants), recursive (recurrence)
- Algorithm complexity, asymptotic notation
- How do we compute the complexity of iterative and recursive algorithms