Lecture #1 Introduction. Complexity Theory

Fundamental Algorithms

Rodica Potolea, Camelia Lemnaru and Ciprian Oprișa

Technical University of Cluj-Napoca
Computer Science

October 2023

Agenda

1. Complexity of Divide&Conquer Algorithms. Master Theorem

- An algorithm is ...

What is an algorithm?

- "Word used by programmers when they do not want to explain what they did"

What is an algorithm?

- “Word used by programmers when they do not want to explain what they did”
- “Something that made something do something in some amount of time”

What is an algorithm?

- “Word used by programmers when they do not want to explain what they did”
- “Something that made something do something in some amount of time”
- “When a piece of code from stackoverflow works but you don’t know why and how!”

What is an algorithm?

- "Word used by programmers when they do not want to explain what they did"
- "Something that made something do something in some amount of time"
- "When a piece of code from stackoverflow works but you don't know why and how!"
- **A method of solving a problem which can be implemented and run by a computer**.

What is an algorithm?

- "Word used by programmers when they do not want to explain what they did" :)
- "Something that made something do something in some amount of time"
- "When a piece of code from stackoverflow works but you don't know why and how!"
- **A method of solving a problem which can be implemented and run by a computer**.

Properties of algorithms

- Correctness
  - "Program testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra, 1970, *Notes On Structured Programming*)
  - correctness MUST be *proven*
  - incorrectness is *exemplified* (target corner cases, equality, etc)

Properties of algorithms

- Correctness
- Efficiency
  - main goal of this course
  - we'll come back to this in a bit ...

Properties of algorithms

- Correctness
- Efficiency
- Ease of implementation
  - *Galactic algorithms* are usually avoided in practice
  - see `https://en.wikipedia.org/wiki/Galactic_algorithm`

Complexity

- Algorithm complexity vs Problem complexity
  - highly related (soon...)
- Algorithm complexity **question**: *What is the amount of resources required to run THE algorithm?*
- Resources
  - **time**
  - memory
  - other (arithmetic operators, secondary memory accesses, network traffic, etc.)
- Time - 2 components in parallel execution
  - computation time
  - communication time (data transfer, partial results transfer, information communication)

Algorithm Complexity (1)

- Quantifies the efficiency of an algorithm by the $<time>$ required to solve the problem
  - $<time>$ can be replaced with any other resource, but it is the most important
- How do we actually evaluate efficiency?
  - Measure ACTUAL time
    - $time = f(sec)$? Why? Why not?
- Cases to be considered
  - best
  - average
  - worst
- Cases relate to ?

Algorithm Complexity (2)

- Quantifies the efficiency of an algorithm by the $<time>$ required to solve the problem
  - $<time>$ can be replaced with any other resource, but it is the most important
- How do we actually evaluate efficiency?
  - Measure ACTUAL time
    - $time = f(sec)$? Why? Why not?
- Cases to be considered (best, average, worst)
- Cases relate to ?
  - the *algorithm* implementing the given problem
  - the *implementation* of the algorithm
- Handled by the **Analysis of Algorithms** field

Problem Complexity

- Handled by the **Computational Complexity Theory** field
- Problem complexity **question**: *What is the* **least** *amount of resources required by* **any** *of the possible (known/unknown) algorithms that could solve the given problem?*
- Mathematical models of computation
- Establish the *practical limits* on what computers (and algorithms) can/cannot do
- In **practice**, when discussing about the complexity (of a problem), we are interested in evaluating the efficiency of a solution (specific implementation)
    - relative (alg. A is more efficient than alg. B)
    - absolute (alg. A is optimal)

Complexity – Efficiency

- **relative**: comparison between algorithms
  - i.e. have degrees of comparison
  - Alg1 is *more/less efficient than* Alg2
- $T(n)$ – function expressing the execution time of a certain algorithm
- only asymptotic behavior matters
  - E.g. given $T_1(n) = 3n^2 + 300n + 50$ and $T_2(n) = 2n^3 + 10n^2 + 2n + 10$ we consider $T_1(n) \sim n^2$ and $T_2(n) \sim n^3$
  - Alg1 is more efficient than Alg2

Complexity – Optimality (1)

- **absolute**: compare T(n) of an algorithm with ?

Complexity – Optimality (1)

- **absolute**: compare *T(n)* of an algorithm with **problem complexity**

Complexity – Optimality (1)

- **absolute**: compare *T(n)* of an algorithm with **problem complexity**
- characterize the *optimality* of an algorithm

Complexity – Optimality (1)

- **absolute**: compare *T(n)* of an algorithm with **problem complexity**
- characterize the *optimality* of an algorithm
  - it does NOT have degrees of comparison

Complexity – Optimality (1)

- **absolute**: compare *T(n)* of an algorithm with **problem complexity**
- characterize the *optimality* of an algorithm
  - it does NOT have degrees of comparison
- How do we operationalize this?

Complexity – Optimality (1)

- **absolute**: compare *T(n)* of an algorithm with **problem complexity**
- characterize the *optimality* of an algorithm
  - it does NOT have degrees of comparison
- How do we operationalize this?
  - What do you compare on the *algorithm side*?

Complexity – Optimality (1)

- **absolute**: compare *T(n)* of an algorithm with **problem complexity**
- characterize the *optimality* of an algorithm
  - it does NOT have degrees of comparison
- How do we operationalize this?
  - What do you compare on the *algorithm side*?
  - What does *problem complexity* even mean, from a *practical* standpoint?

Complexity – Optimality (2)

- O expresses the asymptotic **upper bound** of a function:
  $O(g(n)) = \{f(n) | \exists c, n_0 > 0, s.t. 0 \leq f(n) \leq c * g(n), \forall n \geq n_0\}$
  - related to the algorithm(expresses the execution time of the algorithm implementing a problem, as a number/expression/**function** of execution steps)
  - Notation: $f(n) = O(g(n))$
- Ω expresses the asymptotic **lower bound** of a function:
  $\Omega(g(n)) = \{f(n) | \exists c, n_0 > 0, s.t. 0 \leq c * g(n) \leq f(n), \forall n \geq n_0\}$
  - related to the problem (expresses the theoretical number of steps required by the problem to be solved)
  - Notation: $f(n) = \Omega(g(n))$

Complexity – Optimality (3)

- Optimality of an algorithm is defined in relation to the problem lower bound absolute (Ω)
- Optimality is a *superlative*
  - no degrees of comparison!
  - an algorithm is either optimal, or not
- So, we compare O (algorithm) with Ω (problem)
  - Which case should we consider for the algorithm?

Complexity – Optimality (4)

- We compare O (algorithm) with Ω (problem)
  - Which case should we consider for the algorithm?
    - The sorting problem has the lower bound of $\Omega(n * lgn)$, and many sorting algorithms have $O(1)$ best case and $O(n)$ average case!!!

Complexity – Optimality (5)

- We compare O (algorithm) with Ω (problem)
  - **Worst** case, **asymptotic**, algorithm behaviour

Complexity – Optimality (5)

- We compare O (algorithm) with Ω (problem)
  - **Worst** case, **asymptotic**, algorithm behaviour
- Definition: An algorithm is optimal if the running time of the algorithm to solve the problem in the *worst case* scenario equals the lower bound of the given problem, and the algorithm uses constant additional memory: $O = \Omega$

Complexity – Optimality (5)

- We compare O (algorithm) with Ω (problem)
  - **Worst** case, **asymptotic**, algorithm behaviour
- Definition: An algorithm is optimal if the running time of the algorithm to solve the problem in the *worst case* scenario equals the lower bound of the given problem, and the algorithm uses constant additional memory: $O = \Omega$
- Generally, we are interested in:
  - EITHER developing algorithms with $T(n)$ such that $\Omega \leq T(n) \leq O$ where $O =$ running time of the best known algorithm for the given problem
  - OR identifying the best known algorithms

Complexity – Optimality (5)

- We compare O (algorithm) with Ω (problem)
  - **Worst** case, **asymptotic**, algorithm behaviour
- Definition: An algorithm is optimal if the running time of the algorithm to solve the problem in the *worst case* scenario equals the lower bound of the given problem, and the algorithm uses constant additional memory: $O = \Omega$
- Generally, we are interested in:
  - EITHER developing algorithms with $T(n)$ such that $\Omega \leq T(n) \leq O$ where $O =$ running time of the best known algorithm for the given problem
  - OR identifying the best known algorithms
- The BAD news: many of the real-world problems do NOT have good algorithms
  - No such algorithms will exist (EVER?) $->$ NPC problems

Complexity – Rules for estimating O

- $O(c * f(n)) = O(f(n))$
- $O(f_1(n) * f_2(n)) = O(f_1(n)) * O(f_2(n))$ (in nested loops)
- $O(f_1(n) + f_2(n)) = O(f_1(n)) + O(f_2(n)))$ (in consecutive loops)
- When expressing O, only leading term is considered

Complexity – Function Growths

$f1(n)$	$LEADS$	$f2(n)$	
n^n		$n!$	
$n!$		a^n,	$a > 1$
a^n		b^n,	$a > b$
$\underline{a^n}$		n^b,	$a > 1$
$log_a n$		$log_b n$,	$b > a > 1$
$log_a n$		1,	$a > 1$

- Values of $\Omega()$ for some problems

Complexity – Function Growths

$f1(n)$	$LEADS$	$f2(n)$	
n^n		$n!$	
$n!$		a^n,	$a > 1$
a^n		b^n,	$a > b$
a^n		n^b,	$a > 1$
$log_a n$		$log_b n$,	$b > a > 1$
$log_a n$		1,	$a > 1$

- Values of $\Omega()$ for some problems
  - Searching: $\Omega(logn)$

Complexity – Function Growths

$f1(n)$	$LEADS$	$f2(n)$	
n^n		$n!$	
$n!$		a^n,	$a > 1$
a^n		b^n,	$a > b$
$\underline{a^n}$		n^b,	$a > 1$
$log_a n$		$log_b n$,	$b > a > 1$
$log_a n$		1,	$a > 1$

- Values of $\Omega()$ for some problems
  - Searching: $\Omega(logn)$
  - Selection: $\Omega(n)$

Complexity – Function Growths

$f1(n)$	$LEADS$	$f2(n)$	
n^n		$n!$	
$n!$		a^n,	$a > 1$
a^n		b^n,	$a > b$
a^n		n^b,	$a > 1$
$log_a n$		$log_b n$,	$b > a > 1$
$log_a n$		1,	$a > 1$

- Values of $\Omega()$ for some problems
  - Searching: $\Omega(logn)$
  - Selection: $\Omega(n)$
  - Sorting $\Omega(n * logn)$
    *The base of the log in CS is 2

Complexity – Remarks

- $O(1)$ = constant time (i.e. same running time, regardless of problem size!)
- Only asymptotic behavior matters!
  - $t_1(n) = 3 * n^2 + 3 * n + 5 => O(n^2)$
  - $t_2(n) = 2 * n^3 + 100 * n^2 + 25 * n + 1000 => O(n^3)$
- ... even if for small n, the leading term is not actually leading (e.g. in $t_2(n)$, for $n < 50$)

Complexity vs Computation Power (1)

- Ω characterizes the problem, *lower* bound
- O characterizes the algorithm, *upper* bound
- *Optimality*: If O (in the worst case) $= \Omega$, and no additional memory is used (sometimes, logarithmic memory accepted)
- If no optimal algorithm is known, what kind of solutions are acceptable?
- Q: How fast does the maximum dimension (of the problem that a certain algorithm solves) grow, IF we increase the speed of the computer?

Complexity vs Computation Power (1)

- Ω characterizes the problem, *lower* bound
- O characterizes the algorithm, *upper* bound
- *Optimality*: If O (in the worst case) $= \Omega$, and no additional memory is used (sometimes, logarithmic memory accepted)
- If no optimal algorithm is known, what kind of solutions are acceptable?
- Q: How fast does the maximum dimension (of the problem that a certain algorithm solves) grow, IF we increase the speed of the computer?
- How do different classes of algorithms affect performance?

Complexity vs Computation Power (2)

- Consider 2 classes of algorithms:
  - Alg_1 – polynomial
  - Alg_2 – exponential
- Assume a new hardware system is built – M_2 – having speed V times increased (compared to former system, M_1)
- ?: How does this increase the maximum size of the problem to be solved (by an algorithm) on the new system?

Complexity vs Computation Power (2)

- Consider 2 classes of algorithms:
  - Alg_1 – polynomial
  - Alg_2 – exponential
- Assume a new hardware system is built – M_2 – having speed V times increased (compared to former system, M_1)
- ?: How does this increase the maximum size of the problem to be solved (by an algorithm) on the new system?
- ... that is: estimate $n_{new} = f(V, n)$, given:
  - $V =$ factor of speed increase of the new machine
  - $n =$ max problem size on the former machine

Complexity vs Computation Power (3)

$$Alg_1 : O(n^k)$$

$$\begin{array}{lll}
 & \#Oper. & Time \\
M_1(old): & n^k & T \\
M_2(new): & n^k & \frac{T}{V} \\
 & V * n^k & T \\
 & n_{new}^k = V * n^k = (V^{\frac{1}{k}} * n)^k & \\
 & So, n_{new} = V^{\frac{1}{k}} * n &
\end{array}$$

SO: if the **speed** of the machine increases V **times**, then the **max dimension** of the problem increases $V^{\frac{1}{k}}$ **times.**

Notes:

- $V^{\frac{1}{k}}$ is a small value
- BUT the degree of the polynomial (k) is small for most problems
- AND it is a *multiplicative* increase

Complexity vs Computation Power (4)

$$Alg_2 : O(2^n)$$

	$\#Oper.$	$Time$
$M_1(old)$:	2^n	T
$M_2(new)$:	2^n	$\frac{T}{V}$
	$V * 2^n$	T

$$2^{n_{new}} = V * 2^n = 2^{lgV+n}$$

$$So, n_{new} = n + lgV$$

SO, disadvantageous consequence: If the **speed** of the machine increases V **times**, then the **max dimension** of the problem increases with lgV.
BAD news:

- very small increase (logarithmic)
- AND it is an *additive (!!!)* increase

Complexity vs Computation Power (5)

- So, if speed increases V times:
  - $Alg_1 : O(n^k) : n_{new} = V^{\frac{1}{k}} * n$
  - $Alg_2 : O(2^n) : n_{new} = n + lgV$
- *Conclusion:* For exponential algs., no matter how many times we increase the speed of the system, the size increases with an additive constant!!!
  - NEVER develop exponential algorithms!
  - BUT, what do we do with the problems with unknown polynomial solution?
  - P = NP? (one of the 7 Millenium Prize Problems) – see `https://tinyurl.com/3dkattps`

Complexity vs Computation Power (6)

- Homework: Consider your personal computer/notebook. Check the number of instructions/second it can execute, then compute which is the maximum problem size (i.e. n) that a (1) polynomial and (2) exponential algorithm can solve in:
  - 1 day
  - 1 week
  - 1 month
  - 1 year
  - 100 years
  - 1.000 years
  - 1.000.000 years

Agenda

1. Complexity of Divide&Conquer Algorithms. Master Theorem

Complexity of Divide&Conquer Algorithms. Master Theorem

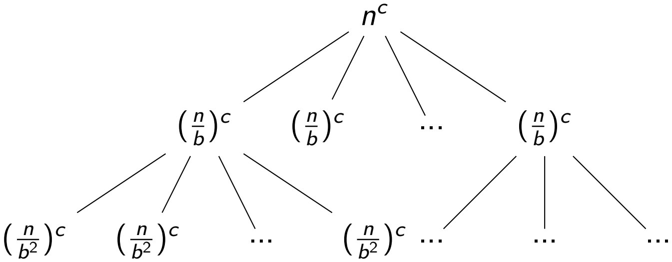
```
DIVIDE_ET_IMPERA(n, I, O)
if n ≤ n0
        DIRECT_SOLUTION(n, I, O)
else
        DIVIDE(n, I1, I2, ..., Ia)
        DIVIDE_ET_IMPERA(n/b, I1, O1)
        ... // a recursive calls in total
        DIVIDE_ET_IMPERA(n/b, Ia, Oa)
        COMBINE(O1, O2, ..., Oa, O)
```

Complexity of Divide&Conquer Algorithms (1)

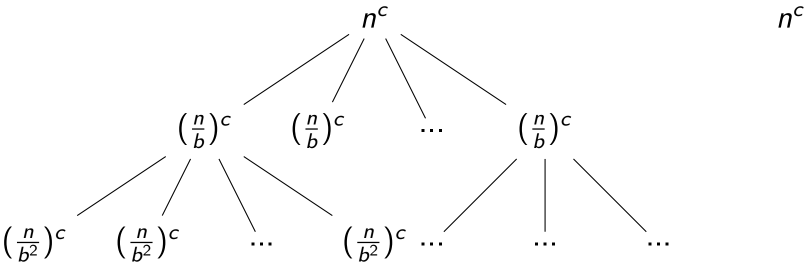
$$T(n) = \begin{cases} T_0, & \text{if } n < n_0 \\ a * T(\frac{n}{b}) + f(n), & \text{otherwise} \end{cases} \tag{1}$$

- What does $f(n)$ capture?
- Assume: $f(n) = n^c$
- *To remember:*
  - $a =$ number of recursive calls
  - $b =$ division factor (of the input)
  - $c =$ degree of the polynomial describing the running time of the sequence excepting the recursive calls

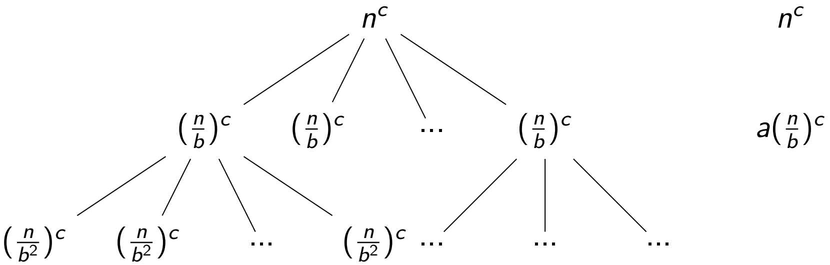
Complexity of Divide&Conquer Algorithms (2)



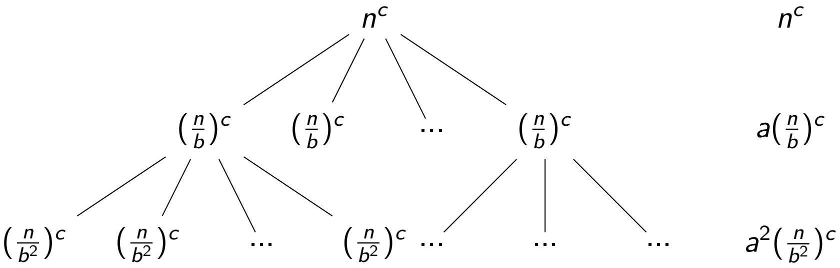
Complexity of Divide&Conquer Algorithms (2)



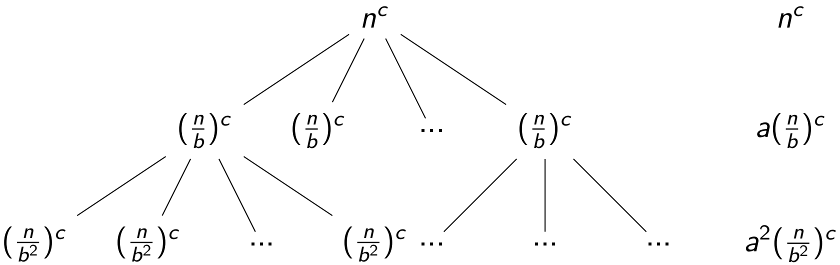
Complexity of Divide&Conquer Algorithms (2)



Complexity of Divide&Conquer Algorithms (2)

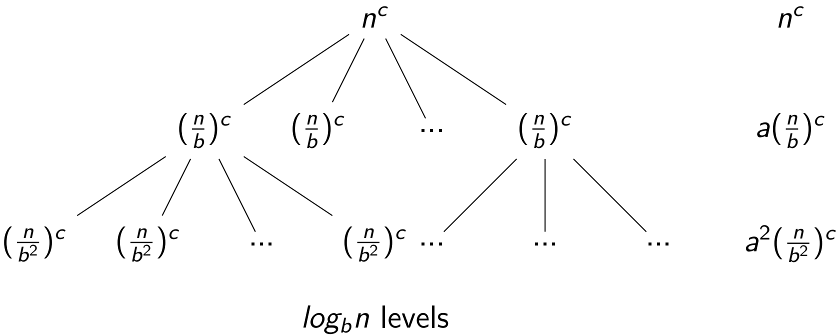


Complexity of Divide&Conquer Algorithms (2)

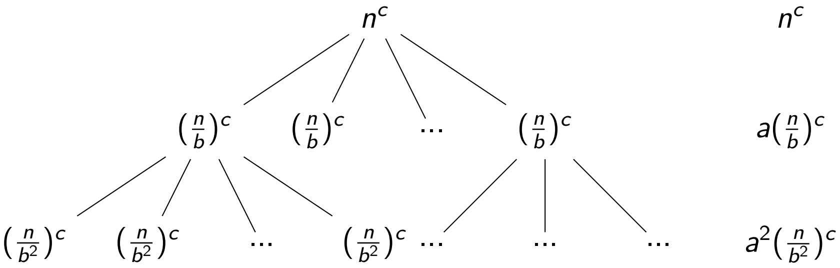


How many levels? …

Complexity of Divide&Conquer Algorithms (2)

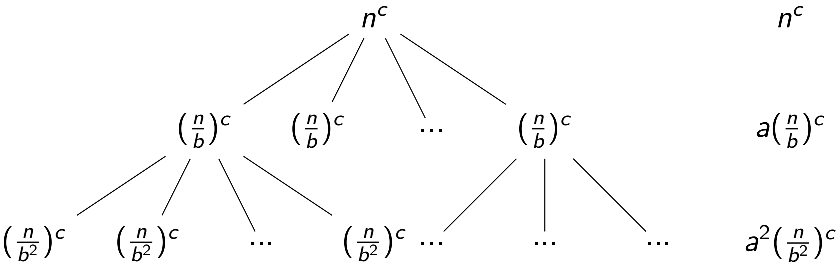


Complexity of Divide&Conquer Algorithms (2)



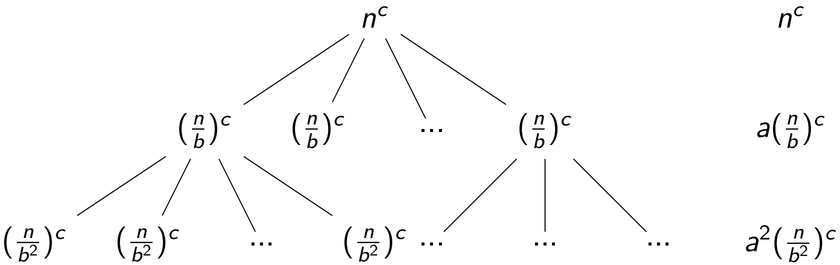
$$T(n) = n^c + a\left(\frac{n}{b}\right)^c + a^2\left(\frac{n}{b^2}\right)^c + \ldots$$

Complexity of Divide&Conquer Algorithms (2)



$$T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + \ldots + (\frac{a}{b^c})^{log_b n - 1}]$$

Complexity of Divide&Conquer Algorithms (2)



$$T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n - 1}]$$

Geometric progression: $term_0 = 1, q = \frac{a}{b^c}, \#terms = log_b n$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n - 1}]$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n-1}]$
- Cases:

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n - 1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n - 1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n - 1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$
  3. $q > 1; a > b^c => O(?)$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n - 1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$
  3. $q > 1; a > b^c => O(?)$

$t = term_0 * \frac{(q^{\#terms} - 1)}{q - 1}$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n-1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$
  3. $q > 1; a > b^c => O(?)$

$t = term_0 * \frac{(q^{\#terms} - 1)}{q-1}$

$T(n) = n^c * \frac{[(\frac{a}{b^c})^{log_b n-1} - 1]}{\frac{a}{b^c} - 1}$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n - 1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$
  3. $q > 1; a > b^c => O(?)$

$t = term_0 * \frac{(q^{\#terms} - 1)}{q - 1}$

$T(n) = n^c * \frac{[(\frac{a}{b^c})^{log_b n - 1} - 1]}{\frac{a}{b^c} - 1}$

Take the asymptotic leading term: $n^c * (\frac{a}{b^c})^{log_b n}$

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n-1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$
  3. $q > 1; a > b^c => O(?)$

  $t = term_0 * \frac{(q^{\#terms}-1)}{q-1}$

  $T(n) = n^c * \frac{[(\frac{a}{b^c})^{log_b n-1}-1]}{\frac{a}{b^c}-1}$

  Take the asymptotic leading term: $n^c * (\frac{a}{b^c})^{log_b n}$

  Question: $O(n^c * (\frac{a}{b^c})^{log_b n}) = O(n^\alpha)$?

Complexity of Divide&Conquer Algorithms (3)

- $T(n) = n^c[1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + ... + (\frac{a}{b^c})^{log_b n-1}]$
- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$
  3. $q > 1; a > b^c => O(?)$

  $t = term_0 * \frac{(q^{\#terms} - 1)}{q-1}$

  $T(n) = n^c * \frac{[(\frac{a}{b^c})^{log_b n-1} - 1]}{\frac{a}{b^c} - 1}$

  Take the asymptotic leading term: $n^c * (\frac{a}{b^c})^{log_b n}$

  Question: $O(n^c * (\frac{a}{b^c})^{log_b n}) = O(n^\alpha)$?

  If yes, then $\alpha =$?

Complexity of Divide&Conquer Algorithms (3)

$n^{\alpha} = n^{c} * \left(\frac{a}{b^{c}}\right)^{log_b n}$ divide by n^{c}

Complexity of Divide&Conquer Algorithms (3)

$n^{\alpha} = n^{c} * \left(\frac{a}{b^{c}}\right)^{log_b n}$ divide by n^c

$n^{\alpha - c} = \left(\frac{a}{b^{c}}\right)^{log_b n}$ apply log_b

Complexity of Divide&Conquer Algorithms (3)

$n^{\alpha} = n^{c} * (\frac{a}{b^{c}})^{log_{b}n}$ divide by n^{c}

$n^{\alpha - c} = (\frac{a}{b^{c}})^{log_{b}n}$ apply log_{b}

$(\alpha - c)log_{b}n = log_{b}n * log_{b}(\frac{a}{b^{c}})$ divide by $log_{b}n$

Complexity of Divide&Conquer Algorithms (3)

$n^{\alpha} = n^{c} * (\frac{a}{b^c})^{log_b n}$	divide by n^c
$n^{\alpha - c} = (\frac{a}{b^c})^{log_b n}$	apply log_b
$(\alpha - c) log_b n = log_b n * log_b(\frac{a}{b^c})$	divide by $log_b n$
$(\alpha - c) = log_b a - c$	add c

Complexity of Divide&Conquer Algorithms (3)

$n^{\alpha} = n^{c} * (\frac{a}{b^c})^{log_b n}$	divide by n^c
$n^{\alpha - c} = (\frac{a}{b^c})^{log_b n}$	apply log_b
$(\alpha - c) log_b n = log_b n * log_b(\frac{a}{b^c})$	divide by $log_b n$
$(\alpha - c) = log_b a - c$	add c
$\alpha = log_b a$	

Complexity of D&C Algs: Master Theorem

$$T(n) = \begin{cases} T_0, & \text{if } n < n_0 \\ a * T(\frac{n}{b}) + n^c, & \text{otherwise} \end{cases} \quad (2)$$

- Cases:
  1. $q < 1; a < b^c => O(n^c)$
  2. $q = 1; a = b^c => O(n^c * log_b n)$
  3. $q > 1; a > b^c => O(n^{log_b a})$ Independent of c!
- Observations:
  - b should be the scaler $(b > 1)$
  - composition should comply with the partition rule – in most cases, either divide, or combine is some (almost) default operation (or it takes just $O(1)$)
    - Examples??

Complexity of D&C Algs: Master Theorem

- Particular Cases:

Complexity of D&C Algs: Master Theorem

- Particular Cases:
  1. $c = 1 => f(n) = n$:

$$T(n) = \begin{cases} O(n), & \text{if } a < b \\ O(n * log_b n), & \text{if } a = b \\ O(n^{log_b a}), & \text{if } a > b \end{cases} \tag{3}$$

Complexity of D&C Algs: Master Theorem

- Particular Cases:
  1. $c = 1 => f(n) = n$:

$$T(n) = \begin{cases} O(n), & \text{if } a < b \\ O(n * log_b n), & \text{if } a = b \\ O(n^{log_b a}), & \text{if } a > b \end{cases} \tag{3}$$

  2. $c = 0 => f(n) = cst$
     Do such algorithms exist?

$$T(n) = \begin{cases} N/A, & \text{if } a < b^0 \\ O(log_b n), & \text{if } a = b^0 \\ O(n^{log_b a}), & \text{if } a > b^0 \end{cases} \tag{4}$$

Complexity of D&C Algs: Master Theorem

- Particular Cases:
  1. $c = 1 => f(n) = n$:

$$T(n) = \begin{cases} O(n), & \text{if } a < b \\ O(n * log_b n), & \text{if } a = b \\ O(n^{log_b a}), & \text{if } a > b \end{cases} \tag{3}$$

  2. $c = 0 => f(n) = cst$
     Do such algorithms exist?

$$T(n) = \begin{cases} N/A, & \text{if } a < b^0 \\ O(log_b n), & \text{if } a = b^0 \\ O(n^{log_b a}), & \text{if } a > b^0 \end{cases} \tag{4}$$

a=1, b=2: which alg.(s)?

Complexity of D&C Algs: Master Theorem

- Particular Cases:
  1. $c = 1 => f(n) = n$:

$$T(n) = \begin{cases} O(n), & \text{if } a < b \\ O(n * log_b n), & \text{if } a = b \\ O(n^{log_b a}), & \text{if } a > b \end{cases} \tag{3}$$

  2. $c = 0 => f(n) = cst$
     Do such algorithms exist?

$$T(n) = \begin{cases} N/A, & \text{if } a < b^0 \\ O(log_b n), & \text{if } a = b^0 \\ O(n^{log_b a}), & \text{if } a > b^0 \end{cases} \tag{4}$$

a=1, b=2: which alg.(s)?
a=2, b=2: which alg.(s)?

Bibliography

- Cormen, Thomas H., et al., *"Introduction to algorithms."*, MIT press, 2009, chap. 2, 3 and 4.3 - 4.6