Time complexity of recursive algorithms. Master theorem

Lecture 06.04

by Marina Barsky

Running time

- To estimate asymptotic running time in non-recursive algorithms we **sum up the number of operations** and ignore the constants

- For recursive algorithms (binary search, merge sort) we draw **the recursion tree**, count number of **operations at each level**, and multiply this number by the **height** of the tree

Running time as a recurrence relation

Binary search:

$T(n) = T(n/2) + O(1) \quad \rightarrow \quad O(\log n)$

Running time for the input of size n is equal the running time for the input of size n/2 plus a constant

Merge sort:

$T(n) = 2\,T(n/2) + O(n) \quad \rightarrow \quad O(n \log n)$

Running time for the input of size n is equal twice the running time for the input of size n/2 plus O(n) work

Wouldn't it be nice if we could solve the running time directly from the recurrence relation?

Generic form of a recursive algorithm

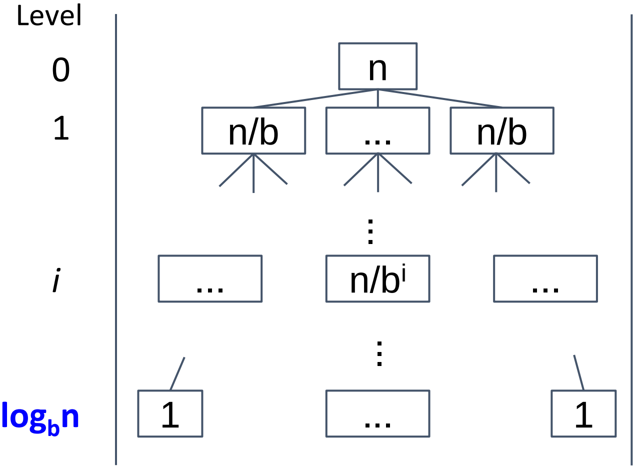
Algorithm *rec* (input x of size n)

if n < some constant k:
    Solve x directly without recursion
else:
    Divide x into ***a*** subproblems, each having size ***n/b***
    Call procedure ***rec*** recursively on each subproblem
    Combine the results from the subproblems in time $O(n^d)$

Running time: $T(n) = \mathbf{a}T(n/\mathbf{b}) + O(n^d)$
where $O(n^d)$ is time to both divide and combine the results

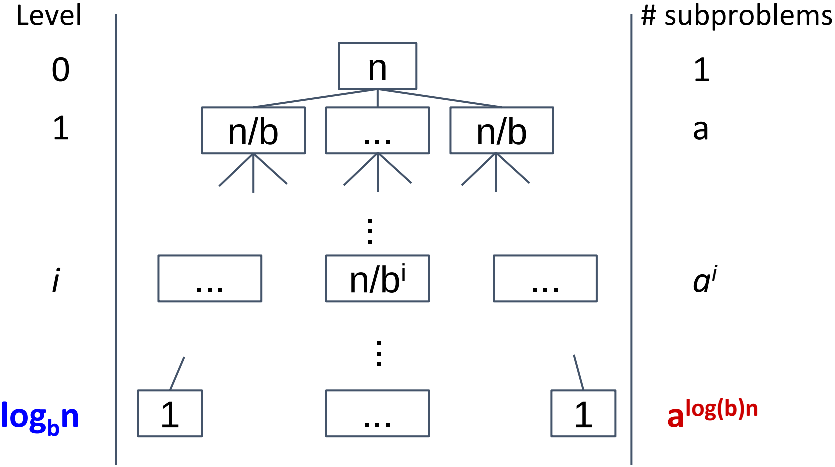
Generic tree: tree height

$T(n) = aT(n/b) + O(n^d)$



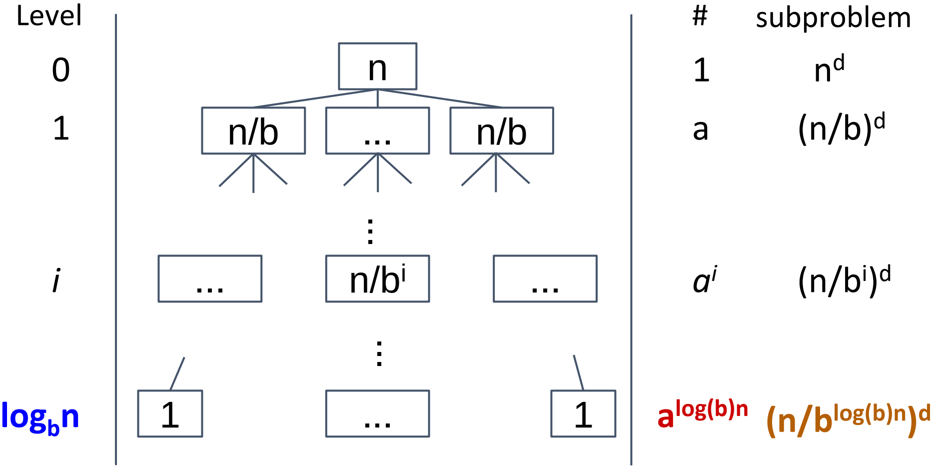
Generic tree: # of subproblems at each level

$T(n) = aT(n/b) + O(n^d)$

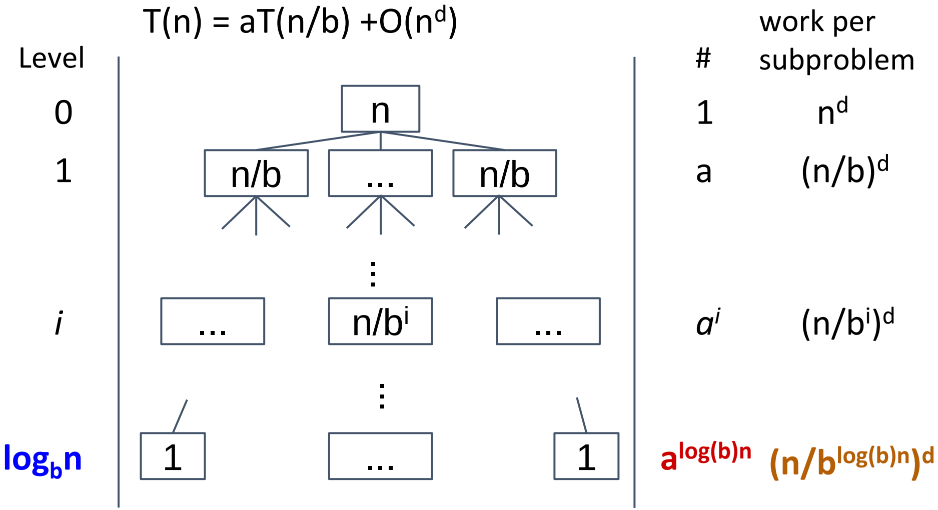


Generic tree: work per subproblem

$T(n) = aT(n/b) + O(n^d)$



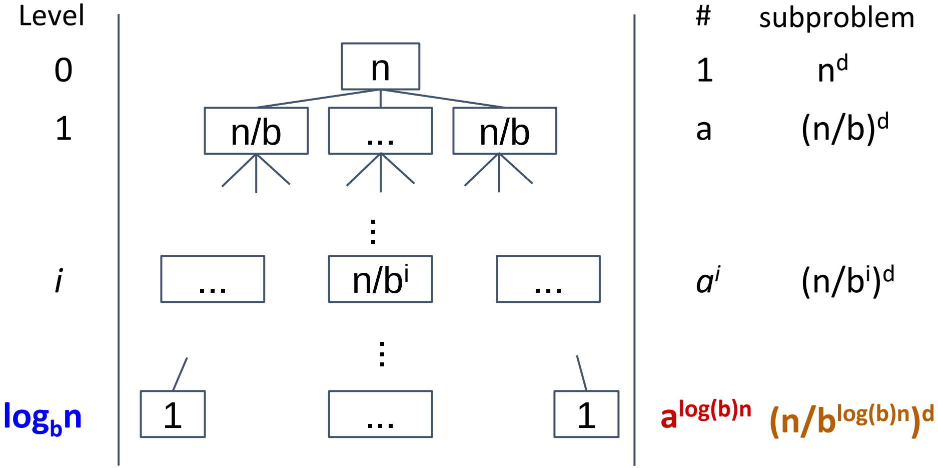
Generic tree: work per subproblem



Total work: ?

Generic tree: total work

$$T(n) = aT(n/b) + O(n^d)$$



$$a^0 * n^d + a^1 * (n/b^1)^d + a^2 * (n/b^2)^d + \dots + a^{\log(b)n} * (n/b^{\log(b)n})^d$$

Counting total work

$a^0 * n^d + a^1*(n/b^1)^d + a^2*(n/b^2)^d + \ldots + a^{\log(b)n}*(n/b^{\log(b)n})^d =$

$n^d * [1 + a/b^d + (a/b^d)^2 + (a/b^d)^3 + \ldots + (a/b^d)^{\log(b)n}]$

Sum of geometric series

$a^0 * n^d + a^1 * (n/b^1)^d + a^2 * (n/b^2)^d + \ldots + a^{\log(b)n} * (n/b^{\log(b)n})^d =$

$n^d * [1 + a/b^d + (a/b^d)^2 + (a/b^d)^3 + \ldots + (a/b^d)^{\log(b)n}]$

The sum of geometric series with k elements (k>=2):

$1 + 1*r + 1*r^2 + \ldots 1*r^k =$

$$\frac{1 - r^k}{1 - r}$$

Sum of geometric series: cases

$a^0 * n^d + a^1*(n/b^1)^d + a^2*(n/b^2)^d + \ldots + a^{\log(b)n}*(n/b^{\log(b)n})^d =$

$n^d * [1 + a/b^d + (a/b^d)^2 + (a/b^d)^3 + \ldots + (a/b^d)^{\log(b)n}]$

The sum of geometric series with k elements:

$1+1*r + 1*r^2 + \ldots 1*r^k$

$$\frac{1 - r^k}{1 - r}$$

Case 1: $r < 1$. Sum becomes 2: O(1) (constant)

Case 2: $r=1$. Sum becomes k: O(k)

Case 3: $r>1$. Sum becomes $O(r^{k-1})=O(r^k)$

It all depends on $r = a/b^d$

Total work: $n^d * [1 + a/b^d + (a/b^d)^2 + (a/b^d)^3 + \ldots + (a/b^d)^{\log(b)n}]$

The sum of geometric series with k elements:

$1+1*r + 1*r^2 + \ldots 1*r^k$

Our r is a/b^d

Our k is $\log_b n$

It all depends on $r = a/b^d$

Total work: $n^d * [1 + a/b^d + (a/b^d)^2 + (a/b^d)^3 + \ldots + (a/b^d)^{\log(b)n}]$

The sum of geometric series with k elements:

$1 + 1*r + 1*r^2 + \ldots 1*r^k$

Case 1:	$r < 1.$	Sum becomes 2: O(1) (constant)
	$a/b^d < 1$	Complexity becomes $O(n^d * 2) = O(n^d)$
Case 2:	$r=1.$	Sum becomes k: O(k)
	$a/b^d = 1$	Complexity becomes $O(n^d * \log(b)n)$
Case 3:	$r>1.$	Sum becomes $O(r^k)$
	$a/b^d > 1$	Complexity becomes $O(n^d * (a/b^d)^{\log(b)n})$

We have shown that:

Total work of a generic recursive algorithm

$T(n) = aT(n/b) + O(n^d) =$

$n^d * [1 + a/b^d + (a/b^d)^2 + (a/b^d)^3 + \ldots + (a/b^d)^{\log(b)n}]$

Case 1:	$a/b^d < 1.$	$O(n^d)$
Case 2:	$a/b^d = 1.$	$O(n^d \log n)$
Case 3:	$a/b^d > 1.$	$O(n^d * (a/b^d)^{\log(b)d})$

We have shown that:

Total work of a generic recursive algorithm

$T(n) = \mathbf{a}T(n/\mathbf{b}) + O(n^{\mathbf{d}}) =$

$n^d * [1 + a/b^d + (a/b^d)^2 + (a/b^d)^3 + \ldots + (a/b^d)^{\log(b)n}]$

Case 1: $\mathbf{a/b^d} < 1.$ $\mathbf{O(n^d)}$

Case 2: $\mathbf{a/b^d} = 1.$ $\mathbf{O(n^d \log n)}$

Case 3: $\mathbf{a/b^d} > 1.$ $\mathbf{O(n^d * (a/b^d)^{\log(b)d})}$

Simplifying case notation

Total work of a generic recursive algorithm

$T(n) = aT(n/b) + O(n^d)$

Case 1:	$a/b^d < 1.$	$O(n^d)$
Case 2:	$a/b^d = 1.$	$O(n^d \log n)$
Case 3:	$a/b^d > 1.$	$O(n^d * (a/b^d)^{\log(b)d})$

$a/b^d < 1$	$\Leftrightarrow$	$d > \log_b a$
$a/b^d = 1$	$\Leftrightarrow$	$d = \log_b a$
$a/b^d > 1$	$\Leftrightarrow$	$d < \log_b a$

Simplifying $n^d * (a/b^d)^{\log(b)d}$

$n^d * (a/b^d)^{\log(b)d} = n^d * a^{\log(b)n} / b^{d \log(b)n}$

But:

$b^{d \log(b)n} = n^d$

easy to see if you take $\log_b$ of both sides:

$\log_b(b^{d \log(b)n}) = \mathbf{d \log_b n}$

$\log_b(n^d) = \mathbf{d \log_b n}$

$n^d * (a/b^d)^{\log(b)d} = n^d * a^{\log(b)n} / b^{d \log(b)n} = n^d * a^{\log(b)n} / n^d = a^{\log(b)n}$

Simplifying $a^{\log(b)n}$

$n^d * (a/b^d)^{\log(b)d} = a^{\log(b)n}$

$a^{\log(b)n} = n^{\log(b)\,a}$

easy to see if you take $\log_a$ of both sides:

$\log_a(a^{\log(b)n}) = \log_b n$

$\log_a(n^{\log(b)\,a}) = \log_b a * \log_a n = \log_b n$ change of base

$n^d * (a/b^d)^{\log(b)d} = a^{\log(b)n} = n^{\log(b)\,a}$

This is called **Master theorem**

$T(n) = aT(n/b) + O(n^d)$

1. if $d > \log_b a$ then $O(n^d)$
2. if $d = \log_b a$ then $O(n^d * \log n)$
3. if $d < \log_b a$ then $O(n^{\log(b)a})$

Pre-conditions:

b > 1 (the subproblem size decreases)

a > 0 (the problem is reduced to a smaller sub problem at least once. At least one recursion level)

d>=0 (the amount of work is polynomial in n)

Example: binary search

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

$T(n) = 1 * T(n/2) + n^0$

$a = 1$

$b = 2$

$d = 0$

$d = \log_b a$

$O(n^0 * \log n) = O(\log n)$

$T(n) = aT(n/b) + O(n^d)$

if $d > \log_b a$ then $O(n^d)$

➡ if $d = \log_b a$ then $O(n^d * \log n)$

if $d < \log_b a$ then $O(n^{\log(b)a})$

Example: merge sort

$T(n) = 2T(n/2) + O(n^1)$

$T(n) = aT(n/b) + O(n^d)$

if $d > \log_b a$ then $O(n^d)$

if $d = \log_b a$ then $O(n^d * \log n)$

if $d < \log_b a$ then $O(n^{\log(b)a})$

$a = 2$

$b = 2$

$d = 1$

$1 = \log_2 2$

$O(n^1 * \log n) = O(n \log n)$

Example: closest pair with $O(n^2)$ combine

$T(n) = 2T(n/2) + O(n^2)$

$T(n) = aT(n/b) + O(n^d)$

- if $d > \log_b a$ then $O(n^d)$
- if $d = \log_b a$ then $O(n^d * \log n)$
- if $d < \log_b a$ then $O(n^{\log(b)a})$

$a = 2$

$b = 2$

$d = 2$

$2 > \log_2 2$

$O(n^2)$

Example: polynomial multiplication

$T(n) = 4T(n/2) + O(n^1)$

$T(n) = aT(n/b) + O(n^d)$

$a = 4$

$b = 2$

$d = 1$

if $d > \log_b a$ then $O(n^d)$

if $d = \log_b a$ then $O(n^d * \log n)$

if $d < \log_b a$ then $O(n^{\log(b)a})$

$1 < \log_2 4$

$O(n^{\log(2)4}) = O(n^2)$

Example: **fast** polynomial multiplication

$T(n) = 3T(n/2) + O(n^1)$

$T(n) = aT(n/b) + O(n^d)$

if $d > \log_b a$ then $O(n^d)$

if $d = \log_b a$ then $O(n^d * \log n)$

➡ if $d < \log_b a$ then $O(n^{\log(b)a})$

$a = 3$

$b = 2$

$d = 1$

$1 < \log_2 3$

$O(n^{\log(2)3})$

Intuitive approach

Compare the total amount of work at the first two levels:

- If total work is the same - this is geometric series with r=1. The complexity is: work on each level * number of levels.
- If total work at the first level > total work at the second level - this is convergent geometric series with r<1. Running time will be dominated by the work at the first level.
- If total work at the first level < total work at the second level - this is sum of geometric series with r>1. Running time will be dominated by the work at the last level: multiply total number of subproblems at the last level by work done for each subproblem

Intuitive example 1:

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

total work on the first level: n^2

total work on the second level: $(n/2)^2 = n^2/4 < n^2$

This is converging geometric series with $r<1$

The most important term is at the first level: **$O(n^2)$**

Complexity: $O(n^2)$

Intuitive example 2:

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

total work on the first level: n

total work on the second level: $3*(n/3) = n$

This is geometric series with r=1

All terms are important:

work per level* total levels = $\mathbf{n * \log_3 n}$

Complexity: O(n log n)

Intuitive example 3: large-integer multiplication

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

total work on the first level: n

total work on the second level: 4*(n/2) = 2n

This is diverging geometric series with r>1

The most important term is at the last level: **$O(n^{\log(b)\ a})$**

Complexity: $O(n^{\log(2)\ 4}) = O(n^2)$

Intuitive example 3': **fast** large-integer multiplication

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

total work on the first level: n

total work on the second level: 3/2*n

This is diverging geometric series with r>1

The most important term is at the last level: $\mathbf{O(n^{\log(b)\ a})}$

Complexity: $\mathbf{O(n^{\log(2)\ 3}) = O(n^{1.585})}$

Intuitive example 4: matrix multiplication

$T(n) = 8T(n/2) + n^2$

total work on the first level: n^2

total work on the second level: $8*(n/2)^2 = 2n^2$

This is expanding geometric series with r>1

The most important term is at the last level: $\mathbf{O(n^{\log(b)\, a})}$

Complexity: $\mathbf{O(n^{\log(2)\, 8}) = O(n^3)}$

Intuitive example 4': **fast** matrix multiplication

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

total work on the first level: n^2

total work on the second level: $7*(n/2)^2 = 7/4\ n^2$

This is expanding geometric series with r>1

The most important term is at the last level: $\mathbf{O(n^{\log(b)\ a})}$

Complexity: $\mathbf{O(n^{\log(2)\ 7}) = O(n^{\log 7}) = O(n^{2.8})}$

Applicability of Master theorem

$$T(n) = aT(n/b) + O(n^d)$$

a > 0

b > 1

The work at each level is polynomial in n, **d**>=0

Can we solve the following recursion using Master method?

$T(n) = 2T(n/2) + \log n$

NO!

So how to solve this recurrence?

T(n) = 2T(n/2) + log n

One idea: intuitively estimate the work at each level

The height of the recursion tree is still log n

The work at level 1 is log n, the work at level 2 is 2 times log (n/2) = 2*log(n/2) = log n

Same work at all levels: O(log n * log n)

Reading

Attached Chapter 11 of

“Algorithm Design and Applications”

by Goodrich and Tomassia