Analysis of Divide-and-Conquer Algorithms

The divide-and-conquer paradigm (Ch 2.3)

- **divide the problem into a number of subproblems**
- **conquer the subproblems by solving them**
- **combine the subproblem solutions to get the final solution**

add all these steps at the first level to get recurrence relation for T(n)

Example: Merge-Sort

- **divide the n-element input sequence to be sorted into two n/2-element subsequences.**
- **conquer the subproblems recursively using merge sort.**
- **combine the resulting two sorted n/2-element sequences by merging.**

Analyzing Divide-and-Conquer Algorithms

A recursive algorithm can often be described by a *recurrence equation* that describes the overall runtime on a problem of size *n* in terms of the runtime on smaller inputs.

For divide-and-conquer algorithms, we get recurrences like:

$$T(n) = \begin{cases} 1 & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

where

- a = number of subproblems we divide the problem into
- n/b = size of the subproblems (in terms of n)
- $D(n)$ = time to divide the size n problem into subproblems
- $C(n)$ = time to combine the subproblem solutions to get the answer for the problem of size n

```
Merge-Sort(A,p,r)
1. if p < r then
2.    q = ⌊(p+r)/2⌋
3.    Merge-Sort(A,p,q)
4.    Merge-Sort(A,q+1,r)
5.    Merge(A,p,q,r)
```

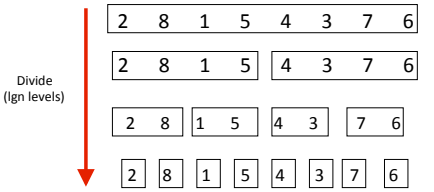
Initial call:

```
Merge-sort(A,1,length(A))
```

The Merge subroutine takes linear time to merge n elements that are divided into two *sorted* arrays of $n/2$ elements each.

```
Merge(A,p,q,r)
1.  n1 = q-p+1; n2 = r-q;
2.  Create arrays
      L[1...n1+1] and
      R[1...n2+1]
3.  for i = 1 to n1
4.     L[i] = A[p+i-1]
5.  for i = 1 to n2
6.     R[i]= A[q+i]
7.  L[n1+1] = R[n2+1] = ∞
8.  i = j = 1
9.  for k = p to r
10.    if L[i] ≤ R[j]
11.       A[k] = L[i]
12.       i = i+1
13.    else A[k] = R[j]
14.       j = j+1
```

Analyzing Merge-Sort

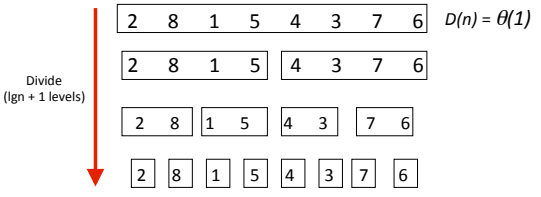


$D(n) = \theta(1)$

Why are there lgn levels?

How long does it take to find the midpoint of an array?

Analyzing Merge-Sort

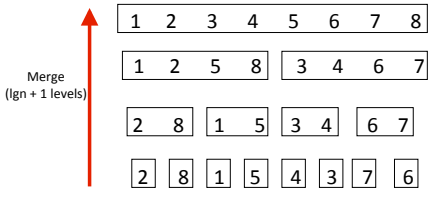


$D(n) = \theta(1)$

Why are there lgn+1 levels? *Because lgn is the number of steps it takes to divide n by 2 until the size of the result is ≤ 1 (counting root level)*

How long does it take to find the midpoint of an array? *Constant time*

Analyzing Merge-Sort



$C(n) = \theta(n)$

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

Analyzing Merge-Sort

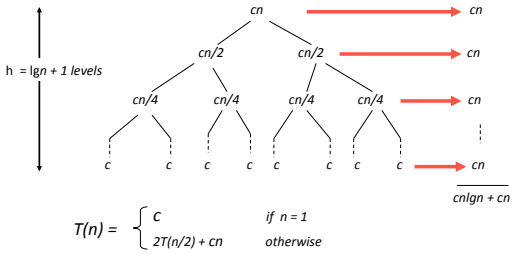
$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

$$aT(n/b) + D(n) + C(n)$$

- a = 2 (two subproblems)
- n/b = n/2 (each subproblem has size approx $n/2$)
- $D(n)$ = θ(1) (just compute midpoint of array)
- $C(n)$ = θ(n) (merging can be done by scanning sorted subarrays)

Recurrence Tree for Merge-Sort



$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Recurrence tree for Merge-Sort

General Plan for calculating running time of recursive algorithms

1. Decide on a parameter indicating input size.
2. Set up a recurrence relation, with the appropriate base cases.
3. Solve the recurrence or otherwise ascertain the order of growth using, e.g. backward substitution, the master method, a recursion tree, or a good guess.

Solving Recurrences

I will cover the first 2 techniques to solve recurrences. The third method is covered in the book (as is solving with a good guess).

1. Backward Substitution: involves substituting next step into equation until you see a pattern, converting the pattern to a summation, and solving the summation.
2. Apply the "Master Theorem": If the recurrence has the form
   $T(n) = aT(n/b) + f(n)$
   then there is a formula that can (often) be applied, given in § 4-5.
3. Apply the recursion tree method from § 4-4.

To make the solutions simpler, we will

- assume base cases are constant time, i.e., T(n) = θ(1) for small enough *n*.

Review of Logarithms

A logarithm is an inverse exponential function. Saying $b^x = y$ is equivalent to saying $\log_b y = x$.

- **notation convention for logarithms:**
  $\lg n = \log_2 n$ *(binary logarithm -- note, no subscript, just lg)*
  $\ln n = \log_e n$ *(natural logarithm)*
- **properties of logarithms:**
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b x^a = a\log_b x$
  $\log_b a = \log_x a/\log_x b$
  $a = b^{\log_b a}$ (e.g., $n = 2^{\lg n} = n^{\lg 2}$)

Log functions grow very slowly as n grows without bound.

More Math Notation

- Floor: $\lfloor x \rfloor$ = the largest integer $\leq x$ (round down)
- Ceiling: $\lceil x \rceil$ = the smallest integer $\geq x$ (round up)
- Summations: (see Appendix A, p.1146-1147)
- Geometric, Telescoping & Harmonic series: (see Appendix A, p.1147-1148)

Solving recurrence for n!

```
Algorithm F(n)
  Input:  a positive integer n
  Output:  n!
1. if n=1
2.     return 1
3. else
4.     return F(n-1) * n
```

We can solve this recurrence (ie, find an expression of the running time T(n) that is not given in terms of itself) using a method known as *backward substitution.*

T(n) for the factorial problem

For recursive algorithms such as computing the *factorial* of n, we get an expression like the following:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + D(n) + C(n) & \text{otherwise} \end{cases}$$

where

- *n-1* = size of the subproblems (in terms of *n*)
- *D(n)* = time to divide the size *n* problem into subproblems
- *C(n)* = time to combine the subproblem solutions to get the answer for the problem of size *n*

Solving recurrence for n!

```
Algorithm F(n)
  Input:  a positive integer n
  Output:  n!
1. if n=1
2.     return 1
3. else
4.     return F(n-1) * n
```

T(n) = T(n-1) + 1
T(1) = 1

T(n) = T(n-1) + 1 (subst T(n-1) = [T(n-2) + 1])
= [T(n-2) + 1] + 1 = T(n-2) + 2
(subst T(n-2) = [T(n-3) + 1])
=[T(n-3) + 1] + 2 = T(n-3) + 3...
...
=T(n-i)+i = ... = T(n-n)+n = 0 + n

Therefore, this algorithm has linear running time.

We can solve this recurrence (ie, find an expression of the running time T(n) that is not given in terms of itself) using a method known as *backward substitution.*

Solving Recurrences: Backward Substitution

Example: T(n) = 2T(n/2) + n

T(n)	=	2T(n/2) + n	
	=	2[2T(n/4) + n/2] + n	/* expand T(n/2) */
	=	4T(n/4) + n + n	/* simplify */
	=	4[2T(n/8) + n/4] + n + n	/* expand T(n/4) */
	=	8T(n/8) + n + n + n	/* simplify */
	=	. . . . .	
	=	$2^{\lg n}T(n/2^{\lg n}) + \ldots + n + n + n$	/* after lgn iterations */
	=	$c2^{\lg n} + n\lg n$	
	=	$cn + n\lg n$	/* $2^{\lg n} = n^{\lg 2} = n$ */
	=	$\theta(n\lg n)$	

Solving Recurrences: Backward Substitution

Example: T(n) = 4T(n/2) + n

T(n)	=	4T(n/2) + n	
	=	4[4T(n/4) + n/2] + n	/* expand T(n/2) */
	=	16T(n/4) + 2n + n	/* simplify 16 = 4^2*/
	=	16[4T(n/8) + n/4] + 2n + n	/* expand T(n/4) */
	=	64T(n/8) + 4n + 2n + n	/* simplify 64 = 4^3*/
	=	. . . . .	
	=	$4^{\lg n}T(n/2^{\lg n}) + \ldots + 4n + 2n + n$	/* after lgn iterations */
	=	$c4^{\lg n} + n\sum_{k=0}^{\lg n-1} 2^k = 2^0 + 2^1 + \ldots + 2^{\lg n-1}$	/* convert to summation */
	=	$cn^{\lg 4} + n(2^{\lg n} - 1)$	/* $4^{\lg n} = n^{\lg 4} = n^2$ */
	=	$cn^2 + n(n - 1)$	/* $2^{\lg n} = n^{\lg 2} = n$ */
	=	$\theta(n^2)$	

Solving Recurrences: Backward Substitution

Example: T(n) = T(n/2) + 1

T(n)	=	T(n/2) + 1	
	=	[T(n/4) + 1] + 1	/* expand T(n/2) */
	=	T(n/4) + 2	/* simplify */
	=	[T(n/8) + 1] + 2	/* expand T(n/4) */
	=	T(n/8) + 3	/* simplify */
	=	. . . . .	
	=	$T(n/2^{\lg n}) + \lg n$	/* $2^{\lg n} = n^{\lg 2} = n$ */
	=	$c + \lg n$	
	=	$\theta(\lg n)$	Which well-known algorithm has this running time??

Solving Recurrences: Master Method (§4.5)

The master method provides a 'cookbook' method for solving recurrences where n is divided repeatedly by a constant. This is the method we will use most often for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

Where a is the number of sub-problems, n/b is the size of each subproblem, and f(n) is the time to divide or combine data.

Solving Recurrences: Master Method (§4.5)

Then, T(n) can be bounded asymptotically as follows:

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a}\lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta f(n)$ if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$
   and if $af(n/b) \le cf(n)$ for some positive constant $c < 1$ and all sufficiently large n.
   (Regularity condition)

Solving Recurrences: Master Method (§4.5)

Master Theorem: Let $a \ge 1$ and $b > 1$ be constants, let f(n) be a function, and let T(n) be defined on nonnegative integers as:

$$T(n) = aT(n/b) + f(n)$$

Where a is the number of sub-problems, n/b is the size of each sub-problem, and f(n) is the time to divide or combine data.

Then, T(n) can be bounded asymptotically as follows:

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) \le n^{\log_b a - \varepsilon}$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a}\lg n)$ if $f(n) = n^{\log_b a}$
3. $T(n) = \theta f(n)$ if $f(n) \ge n^{\log_b a + \varepsilon}$ for some constant $\varepsilon > 0$

Alternate Version of Master Method

Let $a \ge 1$, $b > 1$, $k \ge 0$ be constants, let p be a real number, and let T(n) be defined on nonnegative integers as:

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

Then, T(n) can be bounded asymptotically as follows:

1. If $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. If $a = b^k$, then
   a) If $p > -1$, then $T(n) = \theta(n^{\log_b a}\log^{p+1} n)$
   b) If $p = -1$, then $T(n) = \theta(n^{\log_b a}\log\log n)$
   c) If $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3. If $a < b^k$, then
   a) If $p \ge 0$, then $T(n) = \theta(n^k \log^p n)$
   b) If $p < 0$, then $T(n) = \theta(n^k)$

Solving Recurrences: Master Method (§4.3)

Case 3 requires us to also show $af(n/b) \le cf(n)$, the "regularity" condition.

The regularity condition *always* holds whenever $f(n) = n^k$ and $f(n) \ge n^{\log_b a + \varepsilon}$, so we don't need to check regularity when f(n) is a polynomial.

However, you do need to mention the regularity condition in your arguments to show that the function is $\theta f(n)$ (i.e., if case 3 applies)

Solving Recurrences: Master Method (§4.3)

These 3 cases do not cover all the possibilities for f(n).

There is a gap between cases 1 and 2 when f(n) is smaller than $n^{\log_b a}$, but not polynomially smaller.

There is a gap between cases 2 and 3 when f(n) is larger than $n^{\log_b a}$, but not polynomially larger.

If the function falls into one of these 2 gaps, or if the regularity condition can't be shown to hold, then the master method can't be used to solve the recurrence. Try backward substitution, a good guess proved by induction, or a recursion tree.

Solving Recurrences: Master Method

Example: $T(n) = 9T(n/3) + n$

- a = 9, b = 3, f(n) = n, $n^{\log_b a} = n^{\log_3 9} = n^2$
- compare f(n) = n with n^2
  $n = O(n^{2-\epsilon})$ (so f(n) is polynomially smaller than $n^{\log_b a}$)
- case 1 applies: $T(n) \in \theta(n^2)$

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

- a = 1, b = 2, f(n) = 1, $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- compare f(n) = 1 with 1
  $1 = \theta(n^0)$ (so f(n) is polynomially equal to $n^{\log_b a}$)
- case 2 applies: $T(n) \in \theta(n^0 \lg n) \in \theta(\lg n)$

Solving Recurrences: Master Method

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

- a = 1, b = 2, f(n) = n^2, $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- compare f(n) = n^2 with 1
  $n^2 = \Omega(n^{0+\epsilon})$ (so f(n) is polynomially larger)
- Since f(n) is a polynomial in n, the regularity condition is upheld and case 3 holds, $T(n) \in \theta(n^2)$

Example: $T(n) = 4T(n/2) + n^2$

- a = 4, b = 2, f(n) = n^2, $n^{\log_b a} = n^{\log_2 4} = n^2$
- compare f(n) = n^2 with n^2
  $n^2 = \theta(n^2)$ (so f(n) is polynomially equal)
- Case 2 holds and $T(n) \in \theta(n^2 \lg n)$

Solving Recurrences: Master Method

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

- a = 7, b = 3, f(n) = n^2, $n^{\log_b a} = n^{\log_3 7} = n^{1+\varepsilon}$
- compare f(n) = n^2 with $n^{1+\varepsilon}$
  $n^2 = \Omega(n^{1+\epsilon})$ (so f(n) is polynomially larger)
- Since f(n) is a polynomial in n, the regularity condition holds , so case 3 holds and $T(n) \in \theta(n^2)$

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

- a = 7, b = 2, f(n) = n^2, $n^{\log_b a} = n^{\log_2 7} = n^{2+\varepsilon}$
- compare f(n) = n^2 with $n^{2+\varepsilon}$
  $n^2 = O(n^{2+\epsilon})$ (so f(n) is polynomially smaller)
- Case 1 holds and $T(n) \in \theta(n^{\log_2 7})$

Alternate Version of Master Method

Let $a \geq 1$, $b > 1$, $k \geq 0$ be constants, let p be a real number, and let T(n) be defined on nonnegative integers as:

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

Then, T(n) can be bounded asymptotically as follows:

1. If $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. If $a = b^k$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
3. If $a < b^k$, then $T(n) = \theta(n^k \log^p n)$

Solving Recurrences: Master Method V2

Example: $T(n) = 9T(n/3) + n$

- a = 9, b = 3, k = 1, p = 0, $n^{\log_b a} = n^{\log_3 9} = n^2$
- compare a = 9 with $b^k = 3$, 9 > 3 so
- case 1 applies: $T(n) \in \theta(n^2)$

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

- a = 1, b = 2, k = 0, p = 0, $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- compare a = 1 with $b^k = 1$, 1 = 1
- case 2 applies: $T(n) \in \theta(\log n)$

Solving Recurrences: Master Method V2

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

- a = 1, b = 2, k = 2, p = 0 $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- compare a = 1 with $b^k = 2^2 = 4$, 1 < 4
- Case 3 holds, $T(n) \in \theta(n^2)$

Example: $T(n) = 4T(n/2) + n^2$

- a = 4, b = 2, k = 2, p = 0, $n^{\log_b a} = n^{\log_2 4} = n^2$
- compare a with 2^2, 4 = 4
- Case 2 holds and $T(n) \in \theta(n^2 \lg n)$

Solving Recurrences: Master Method V2

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

- a = 7, b = 3, k = 2, p = 0, $n^{\log_b a} = n^{\log_3 7} = n^{1+\varepsilon}$
- compare a to b^k : $7 < 9$
- Case 3 holds and $T(n) \in \theta(n^2)$

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

- a = 7, b = 2, k = 2, p = 0, $n^{\log_b a} = n^{\log_2 7} = n^{2+\varepsilon}$
- compare a to b^k : $7 > 4$
- Case 1 holds and $T(n) \in \theta(n^{\log_2 7})$