THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON DC

DIVIDE AND CONQUER

PART I

GENERAL TEMPLATE
BINARY SEARCH
MERGE SORT & QUICK SORT
SOLVING RECURRENCE RELATIONS

Design and Analysis of Algorithms

LOGISTICS

- Instructor

  Prof. Amrinder Arora

  amrinder@gwu.edu

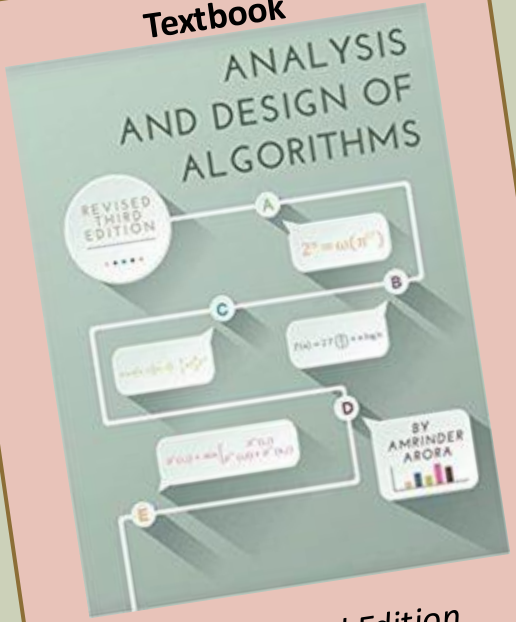
  Please copy TA on emails

  Please feel free to call as well

  ☺

- Available for study sessions
  Science and Engineering Hall
  GWU

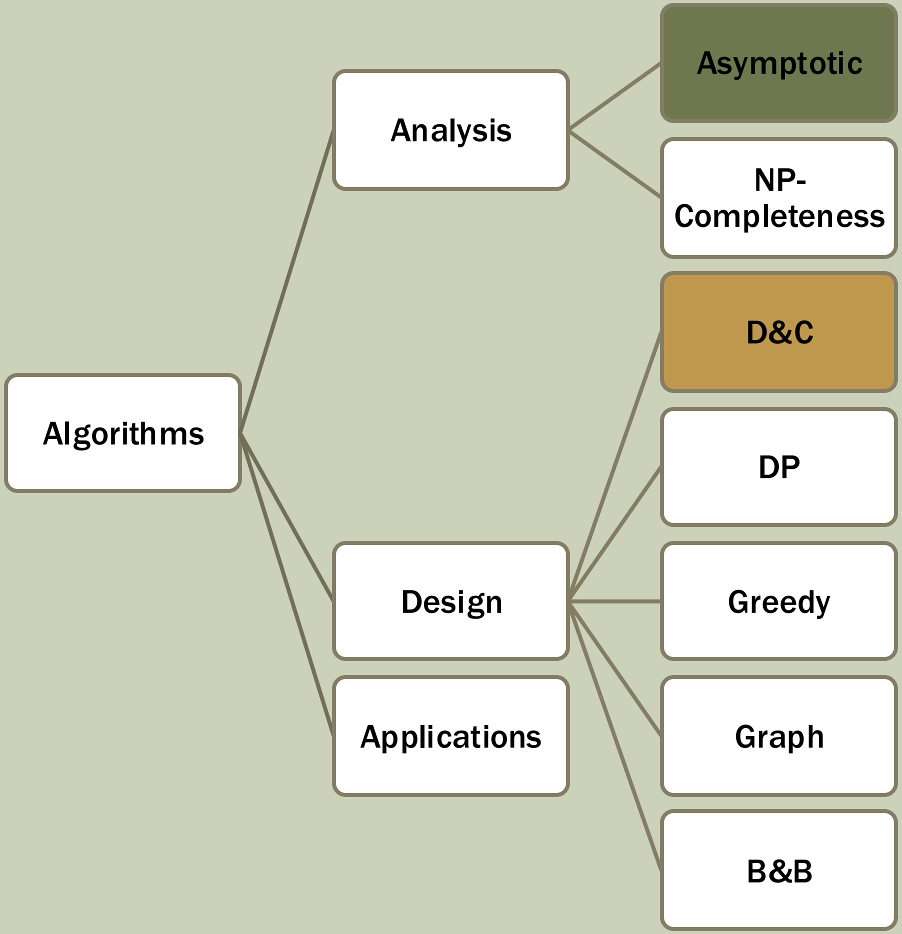
Textbook



Revised Third Edition
Cognella Academic Publisher
www.cognella.com

ISBN: **978-1793549952**

WHERE WE ARE


Algorithms
Analysis
Asymptotic
NP-Completeness
Design
D&C
DP
Greedy
Graph
B&B
Applications

DIVIDE AND CONQUER

- A technique to solve complex problems by breaking into smaller instances of the problem and combining the results
  - Recursive methodology – Smaller instances of the same *type* of problem
- Typically used accompaniments
  - Induction for proving correctness
  - Recurrence relation solving for computing time (and/or space) complexity

D&C – CS, NOT MANAGEMENT/POLITICS

By definition: For D&C, sub problems must be of same type.

[The phrase “D&C” is also used in other contexts. It may refer to breaking down a task, but in Computer Science, D&C is a formal paradigm]

© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com
Bela
"Really? — my people always say multiply and conquer."

RECURSION

- A recursive algorithm is an algorithm that calls itself on smaller input.
- Algorithm sort (Array a)

```
Begin
  sort (subarray consisting of first half of a)
  sort (subarray consisting of second half of a)
  do_something_else();
End
```

RECURRENCE RELATIONS

- Recurrence Relation is a recursive formula, commonly used to analyze the time complexity of recursive algorithms
- For example
  - $T(n) = T(n/2) + T(n/2) + n^2$
  - $T(n) = a\ T(n/b) + f(n)$
- *Note: Recurrence Relations have uses outside of time complexity analysis as well (for example in combinatorics), but for the purpose of this lecture, this is the main use case.*

HOW TO D&C

- Wikipedia says: "…it is often necessary to replace the original problem by a more general or complicated problem in order to get the recursion going, and there is no systematic method for finding the proper generalization."
  - Refer to this as the "generalization" step
  - Sometimes counterintuitive that making a "generalization", that is, making the problem harder actually helps in solving it!

GENERAL TEMPLATE

```
divide_conquer(input J)
{
    // Base Case
   if (size of input is small enough) {
           solve directly and  return
   }

    // Divide Step
   divide J into one or more parts J1, J2,...

    // Recursive Calls
   call divide_conquer(J1) to get a subsolution S1
   call divide_conquer(J2) to get a subsolution S2
   ...
    // Merge Step
   Merge the subsolutions S1, S2,...into a global solution S
   return S
}
```

GENERATE THE RECURRENCE FOR THIS ALGORITHM..

```
divide_conquer(input J (of size n))
{
    // Base Case
   if (n <= 2) {
           solve directly and  return // Assume this takes a constant amount of time
   }

    // Divide Step
   divide J into 3 parts: J1 of size n/2 J2 of size n/3 and J3 of size n/4

    // Recursive Calls
   call divide_conquer(J1) to get a subsolution S1
   call divide_conquer(J2) to get a subsolution S2
   call divide_conquer(J3) to get a subsolution S3

    // Merge Step
   Merge the subsolutions S1, S2, S3 into a global solution S
   // Assume this takes a constant amount of time
   return S
}
```

NUMBER OF BRANCHES

- Number of subproblems that you create in the “divide” step
- This plays a role in the recurrence relation that is created for analysis
  - T(n) = a T(n/b) + f(n)
    Here “a” branches, each with size “n/b”, and f(n) time spent in dividing and merging
  - Example: T(n) = T(n/2) + 1
    1 branch, size half and constant time spent in dividing and merging

GENERAL TEMPLATE – TIME COMPLEXITY VIEW

```
divide_conquer(input J)
{
   // Base Case
   if (size of input is small enough) {
           solve directly and  return
   }

    // Divide Step
   divide J into two or more parts J1, J2,...

    // Recursive Calls
   call divide_conquer(J1) to get a subsolution S1
   call divide_conquer(J2) to get a subsolution S2
   ...
    // Merge Step
   Merge the subsolutions S1, S2,...into a global solution S
   return S
}
```

a **recursive calls of size** ***n/b*** **each. Total time: a T(n/b)**

Combined time in steps other than recursive calls: f(n)

D&C – EXAMPLE ALGORITHMS

- Binary Search
- Merge Sort
- Quick Sort

BINARY SEARCH

- Search (A, low, high, key)
  - Mid = (low + high) / 2
  - Compare A[mid] to key, and look either in left half or in right half
- T(n) = T(n/2) + 1
- T(n) = O(log n)

MERGE SORT

- Classic problem: Given an array, to sort it
- Generalization step: Given an array *and indexes i and j (start and end)* to sort *that portion* of it
- Algorithm MergeSort (input: A,i,j) {

```
        // Base Case
        if (j – i < THRESHOLD) {
           InsertionSort(A,i,j)
          Return
        }

        // Divide portion
        int k=(i+j)/2

        // Recursive Calls
        MergeSort(A,i,k)
        MergeSort(A,k+1,j)

        // Merge Calls
        Merge(A,i,k,k+1,j)
}
```

MERGING

- How to merge two lists effectively?

TIME COMPLEXITY OF MERGE SORT

- $T(n) = 2T(n/2) + \Theta(n)$
- Need some methods for solving such recurrence equations
  - Substitution method
  - Recursion tree method (unfolding)
  - Master theorem
- $T(n) = \Theta(n \log n)$

EXAMPLES OF RECURRENCE RELATIONS

- T(n) = T(n/2) + 1
- T(n) = T(n/2) + n
- T(n) = 2T(n/2) + 1
- T(n) = 2T(n/2) + n
- T(n) = 3T(n/2) + n
- T(n) = 3T(n/2) + n log n
- T(n) = T(sqrt(n)) + 1

// Base Case is usually T(1) = 1,

// You can use any T(a) = b, where a and b are both specific numbers, such as T(100) = 2.

SOLVING RECURRENCE RELATIONS

- Examples:
  - $T(n) = 2\,T(n/2) + cn$

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

    $T(n) = O\,(n)$

- 3 General Approaches:
  - Recursion tree method (unfold and reach a pattern)
  - Substitution method (Guess and Prove)
  - Master theorem

UNFOLDING METHOD

- Given $T(n) = T(n/2) + n$
- Then $T(n) = T(n/2) + n$

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

  $= T(n/8) + n/4 + n/2 + n$

  = ..

UNFOLDING METHOD

- Given T(n) = 2T(n/2) + n^2
- Then T(n) = 2T(n/2) + n^2

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

  = ..

SUBSTITUTION METHOD FOR MERGE SORT

- Given T(n) = 2 T(n/2) + cn
- We first "guess" that the solution is O(n log n)
- To prove this using induction, we first assume T(m) <= km log m for all m < n
- Then T(n) = 2 T(n/2) + cn

  <= 2 kn/2 log (n/2) + cn

  = kn log n – (k – c)n // log (n/2) = log n - 1

  <= k n log n, as long as k >= c

SUBSTITUTION METHOD FOR MERGE SORT

- Given $T(n) = 11\ T(n/10) + 5n$
- We first "guess" that the solution is $O(n \log n)$
- To prove this using induction, we first assume:
  $T(m) <= 100\ m \log m$ for all $m < n$
- Then $T(n) = 11\ T(n/10) + 5n$

  $<= 11\ 100\ (n/10) \log (n/10) + 5n$

  ….

  $<= 110\ n \log n$

  ….

  $<= 110\ n \log n$

  $= O(n \log n)$

INCORRECT CONCLUSION!!!

Must prove the algebraic inequality, before drawing any asymptotic conclusion!

MASTER THEOREM FOR SOLVING RECURRENCE RELATIONS

Only applies to Recurrence Relations of following type

$$T(n) = a\,T(n/b) + f(n)$$

For example, MT does not apply $S(n) = S(n/2) + S(n/3) + f(n)$

- **Case 1.** If $f(n) = O(n^c)$ where $c < \log_b a$, then $T(n) = \theta(n\text{^}\log_b a)$

  $f(n)$ is POLYNOMIALLY smaller than $n\text{^}\log_b a$
- **Case 2.** If it is true, for some constant $k \geq 0$, that $f(n) = \theta(n^c \log^k n)$ where $c = \log_b a$, then $T(n) = \theta(n^c \log^{k+1} n)$
  - *$f(n)$ is POLYNOMIALLY equal to $n\text{^}\log_b a$*
- **Case 3.** If it is true that $f(n) = \Omega(n^c)$ where $c > \log_b a$, then $T(n) = \theta(f(n))$
  - *$f(n)$ is POLYNOMIALLY larger than $n\text{^}\log_b a$*

MASTER THEOREM – INTUITION

$T(n) = a\ T(n/b) + f(n)$

→ $T(n/b) = a\ T(n/b\text{^}2) + f(n/b)$

→ $T(n) = a\ [a\ T(n/b\text{^}2) + f(n/b)] + f(n)$

→ $T(n) = a\text{^}2\ T(n/b\text{^}2) + a\ f(n/b) + f(n)$

…

If we unfold this k times, we get an expression like:

$T(n) = a^k\ T(n/b^k) + f(n) + a\ f(n/b) + \ldots + a^k\ f(n/b^k)$

Then, for $k \approx \log_b n$, $T(n/b^k)$ will be a small constant, and we can assume $T(n/b^k) = 1$.

Then, $T(n) = a\text{^}(\log_b n) + f(n) + af(n/b) + \ldots + a^k\ f(n/b^k)$

$= n\text{^}(\log_b a) + f(n) + af(n/b) + \ldots + a^k\ f(n/b^k)$

We note that there are about $\log_b n$ terms.

MASTER THEOREM – INTUITION (CONT.)

$T(n) = n\text{^}(\log_b a) + f(n) + af(n/b) + \ldots + a^k f(n/b^k)$

We observe that:

- If f(n) is very small, say a constant, then the first term dominates
- If $f(n) = \Theta\ (n\text{^}(\log_b a))$, then the $T(n) = f(n) \log n$.

  // The log n factor arises because there are ~ log n terms
- If f(n) is too large, then f(n) terms dominate

APPLYING MASTER THEOREM TO MERGE SORT RECURRENCE

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

In this case:

- $a = b = 2$
- $f(n) = c\ n$
- $\log_b a = 1$
- n^$(\log_b a) = n$

So, $f(n) = \Theta\ (n$^$(\log_b a))$

Therefore, by Master Theorem,

$T(n) = \Theta(f(n) \log n)$

That is, $T(n) = \Theta(n \log n)$

APPLYING MASTER THEOREM TO OTHER

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

In this case:

- $a = 3$, $b = 2$
- $f(n) = n$
- $\log_b a = \log_2 3$.
- 2^x = 3.
- 2^1 < 3. and 2^2 > 3.
- 1 < log_2 (3) < 2
- f(n) is O of n^ $\log_2 3$
- By MT: T(n) = n^ $\log_2 3$. Case 1

APPLYING MASTER THEOREM TO OTHER

$T(n) = 5\ T(n/2) + n^3$

APPLYING MASTER THEOREM TO OTHER

$T(n) = 5\ T(n/2) + n^3$

In this case:

- $a = 5$
- $b = 2$
- $f(n) = n^3$

- Which term is larger? f(n) or n^log_b(a)

- $T(n) = n^3$ // By Case 3

PRACTICE QUESTIONS

- $T(n) = 5\,T(n/2) + n^3$
  - $T(n) = n^3$
- $T(n) = 5\,T(n/2) + n^2$
  - $T(n) = n^{\log_2 5}$
- $T(n) = 5\,T(n/2) + n^2 \log n$
  - $T(n) = n^{\log_2 5}$
- $T(n) = 5\,T(n/2) + n^3 \log n$
  - $T(n) = n^3 \log n$

PRACTICE QUESTIONS (CASE 2)

$T(n) = T(n/2) + f(n)$ (Binary Search)

$T(n) = \log n$

=======================

$T(n) = T(n/6) + \log^k n$

In this case:

- $a = 1$, $b = 6$
- $f(n) = \log^k n$
- $\log_b a = 0$
- $n\hat{}(\log_b a) = n\hat{}0$

So, $f(n) = \Theta\ (n\hat{}(\log_b a))$

Therefore, by Master Theorem (Case 2),

$T(n) = \Theta(f(n) \log n)$

That is, $T(n) = \Theta(\log^{k+1} n)$

$T(n) = T(n/6) + \log \log^k n$

In this case:

- $a = 1$, $b = 6$
- $f(n) = \log^k n$
- $\log_b a = 0$
- $n\hat{}(\log_b a) = n\hat{}0$

Master Theorem (Case 2),

$T(n) = \Theta(f(n) \log n)$

That is, $T(n) = \Theta(\log n \log \log^k n)$

APPLYING MASTER THEOREM TO ANOTHER RECURRENCE RELATION

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

In this case:

- $a = b = 2$
- $f(n) = c\ n \log n$
- $\log_b a = 1$
- $n\hat{}(\log_b a) = n$

So, $f(n) = \Theta\ (n\hat{}(\log_b a))$

Therefore, by Master Theorem,

$T(n) = \Theta(f(n) \log n)$

That is, $T(n) = \Theta(n \log\hat{}2\ n)$

QUICKSORT

- Select a “partition” element
- Partition the array into “left” and “right” portions (not necessarily equal) based on the partition element
- Sort the left and right sides

- An inverted view of mergesort – spend time upfront (partition), no need to merge later.

QUICKSORT – THE PSEUDO CODE

```
quicksort(A,p,r)
if (p < r) {
        q = partition (A,p,r)
        quicksort(A,p,q-1)
        quicksort(A,q+1,r)
}
```

QUICKSORT (THE TRIVIA CLUB VIEW)

- Invented in 1960 by C. A. R. Hoare
- More widely used than any other sort
- A well-studied, not difficult to implement algorithm
- R. Sedgewick – 1975 Ph.D. thesis at Stanford Univ. – Analysis and Variations of QuickSort

Who said: “Elementary, My Dear Watson”?

QUOTES, QUOTES

- “There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”
- “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”

EXAMPLE RUN

- -23, 100, 1, 3, 5, 18, 0, 31, 102, 34
- Partition element: -23
- []. -23 [100, 1, 3, 5, 18, 0, 31, 102, 34]

- -23, 100, 1, 3, 5, 18, 0, 31, 102, 34
- Partition element: 5
- [-23, 1, 3, 0]. 5 [100, 18, 31, 102, 34]

EXAMPLE PARTITION RUN

- 23, 100, 1, 3, 5, 18, 0, 31, 102, 34, 21, 28, 51, 90, 4
- Partition element: 18
- [4,0, 1, 3, 5] 18 [100, 31, 102, 34, 21, 28, 51, 90, 23]

CENTRAL PROBLEM IN QUICKSORT

- How to find a **good** partition element
- How to partition (efficiently)
- Partition array so that:
  - Some partitioning element (q) is its final position
  - Every element smaller than q is to the left of q
  - Every element larger than q is to the right of q
- Sedgwick states that “improving QuickSort is the better mousetrap of computer science”

QUICKSORT – TIME COMPLEXITY ANALYSIS

- $T(n) = T(n_1) + T(n_2) + O(n)$

  Where $n_1 + n_2 = n - 1$
- So it all depends upon the kind of the split, and split will likely not be the same each time.
- Worst case – very bad split: $O(n^2)$
- Best case – good split: $O(n \log n)$
- Average case – where does that fit?

http://mathworld.wolfram.com/Quicksort.html

OPEN QUESTIONS

How long will the algorithm take?

```
function sum (integer a) {
  if (a == 1) exit;
   if (a is odd) {
     a = 3a + 1
   } else {
     a = a/2
   }
 }
```

Trichotomy – Extended

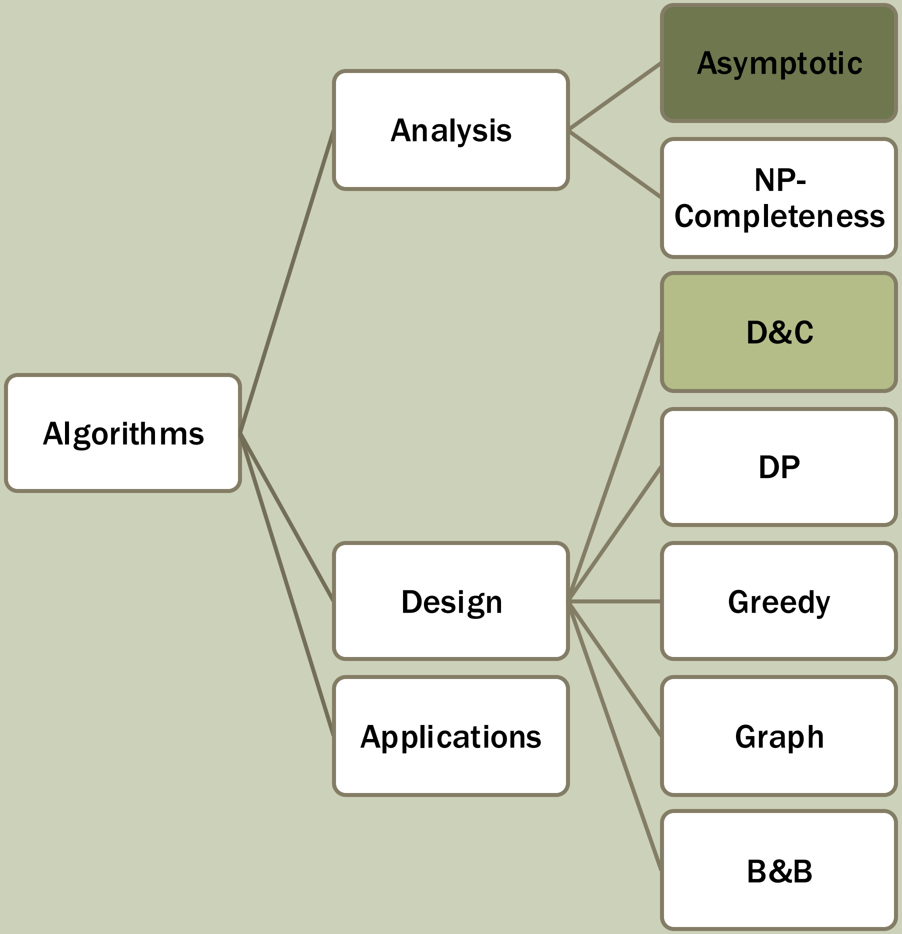
Given two functions $f(n)$ and $g(n)$, both strictly increasing with n, is it possible that $f(n)$ and $g(n)$ cannot be compared asymptotically?

READING ASSIGNMENTS

1. Median Finding (Textbook § 4.6)
2. Closest pair of points algorithm (Textbook § 4.7)
3. Strassen’s algorithm for matrix multiplication (Textbook § 4.8)
   https://youtu.be/1AIvlizGo7Y

We already have quite a few people who know how to divide. So essentially, we are now looking for people who know how to conquer.

WHERE WE ARE


Algorithms
Analysis
Asymptotic
NP-Completeness
Design
D&C
DP
Greedy
Graph
B&B
Applications

More D&C in Next Lecture