Data Structures and Algorithms
Week 3

1. Divide and conquer
2. Merge sort, repeated substitutions
3. Tiling
4. Recurrences

Recurrences

- Running times of algorithms with **recursive calls** can be described using recurrences.
- A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- For divide and conquer algorithms:

$$T(n)=\begin{cases} \text{solving trivial problem} & \text{if } n=1 \\ \text{NumPieces} * T(n/\text{SubProbFactor}) + \text{divide} + \text{combine} & \text{if } n>1 \end{cases}$$

- Example: Merge Sort

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

Solving Recurrences

- Repeated (backward) substitution method
  - Expanding the recurrence by substitution and noticing a pattern (this is not a strictly formal proof).
- Substitution method
  - guessing the solutions
  - verifying the solution by the mathematical induction
- Recursion trees
- Master method
  - templates for different classes of recurrences

Repeated Substitution

- Let's find the running time of merge sort (assume $n=2^b$).

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

$$\begin{aligned} T(n) &= 2T(n/2) + n \quad \text{substitute} \\ &= 2(2T(n/4) + n/2) + n \quad \text{expand} \\ &= 2^2T(n/4) + 2n \quad \text{substitute} \\ &= 2^2(2T(n/8) + n/4) + 2n \quad \text{expand} \\ &= 2^3T(n/8) + 3n \quad \text{observe pattern} \end{aligned}$$

Repeated Substitution/2

From $T(n) = 2^3 T(n/8) + 3n$

we get $T(n) = 2^i T(n/2^i) + i\,n$

An upper bound for *i* is *log n:*

$$T(n) = 2^{\log n} T(n/n) + n \log n$$
$$T(n) = n + n \log n$$

Repeated Substitution Method

- The procedure is straightforward:
  - Substitute, Expand, Substitute, Expand, …
  - Observe a pattern and determine the expression after the *i*-th substitution.
  - Find out what the highest value of *i* (number of iterations, e.g., *log n*) should be to get to the base case of the recurrence (e.g., *T(1)*).
  - Insert the value of *T(1)* and the expression of *i* into your expression.

Analysis of Sort Merge

- Let's find a more exact running time of merge sort (assume n=2^b).

$$T(n) = \begin{cases} 2 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

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

$= 2(2T(n/4) + n + 3) + 2n + 3$ expand

$= 2^2T(n/4) + 4n + 2*3 + 3$ substitute

$= 2^2(2T(n/8) + n/2 + 3) + 4n + 2*3 + 3$ expand

$= 2^3T(n/2^3) + 2*3n + (2^2+2^1+2^0)*3$ observe pattern

Analysis of Sort Merge/2

$$T(n) = 2^i T(n/2^i) + 2in + 3 \sum_{j=0}^{i-1} 2^j$$

An upper bound for i is log n

$$= 2^{\log n} T(n/2^{\log n}) + 2n\log n + 3*(2^{\log n} - 1)$$

$$= 5n + 2n\log n - 3$$

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

Substitution Method

- The substitution method to solve recurrences entails two steps:
  - Guess the solution.
  - Use induction to prove the solution.
- Example:
  - $T(n) = 4T(n/2) + n$

Substitution Method/2

1) *Guess $T(n) = O(n^3)$, i.e., $T(n)$ is of the form cn^3*

2) *Prove $T(n) \leq cn^3$ by induction*

$T(n)$	$= 4T(n/2) + n$	recurrence
	$\leq 4c(n/2)^3 + n$	induction hypothesis
	$= 0.5cn^3 + n$	simplify
	$= cn^3 - (0.5cn^3 - n)$	rearrange
	$\leq cn^3$ if $c>=2$ and $n>=1$	

Thus $T(n) = O(n^3)$

Substitution Method/3

- Tighter bound for $T(n) = 4T(n/2) + n$:

  Try to show $T(n) = O(n^2)$

  Prove $T(n) \leq cn^2$ by induction

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= cn^2 + n \\ &\text{NOT} \leq cn^2 \end{aligned}$$

=> contradiction

Substitution Method/4

- What is the problem? Rewriting

  $T(n) = O(n^2) = cn^2 + (something\ positive)$

  as $T(n) \le cn^2$

  does not work with the inductive proof.
- Solution: Strengthen the hypothesis for the inductive proof:
  - $T(n) \le$ (answer you want) - (something > 0)

Substitution Method/5

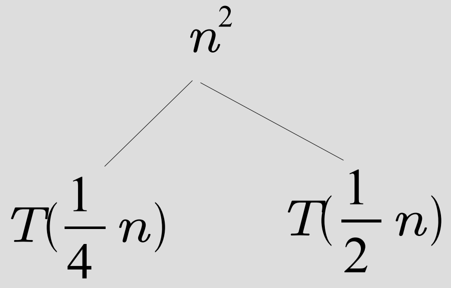
- Fixed proof: strengthen the inductive hypothesis by subtracting lower-order terms:

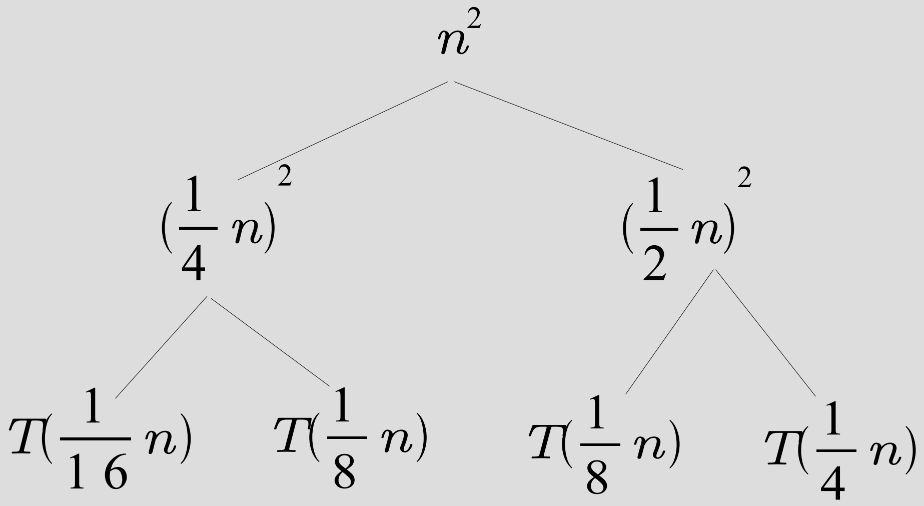
  Prove $T(n) \le cn^2 - dn$ by induction

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4(c(n/2)^2 - d(n/2)) + n \\
&= cn^2 - 2dn + n \\
&= cn^2 - dn - (dn - n) \\
&\le cn^2 - dn \text{ if } d \ge 1
\end{aligned}
$$

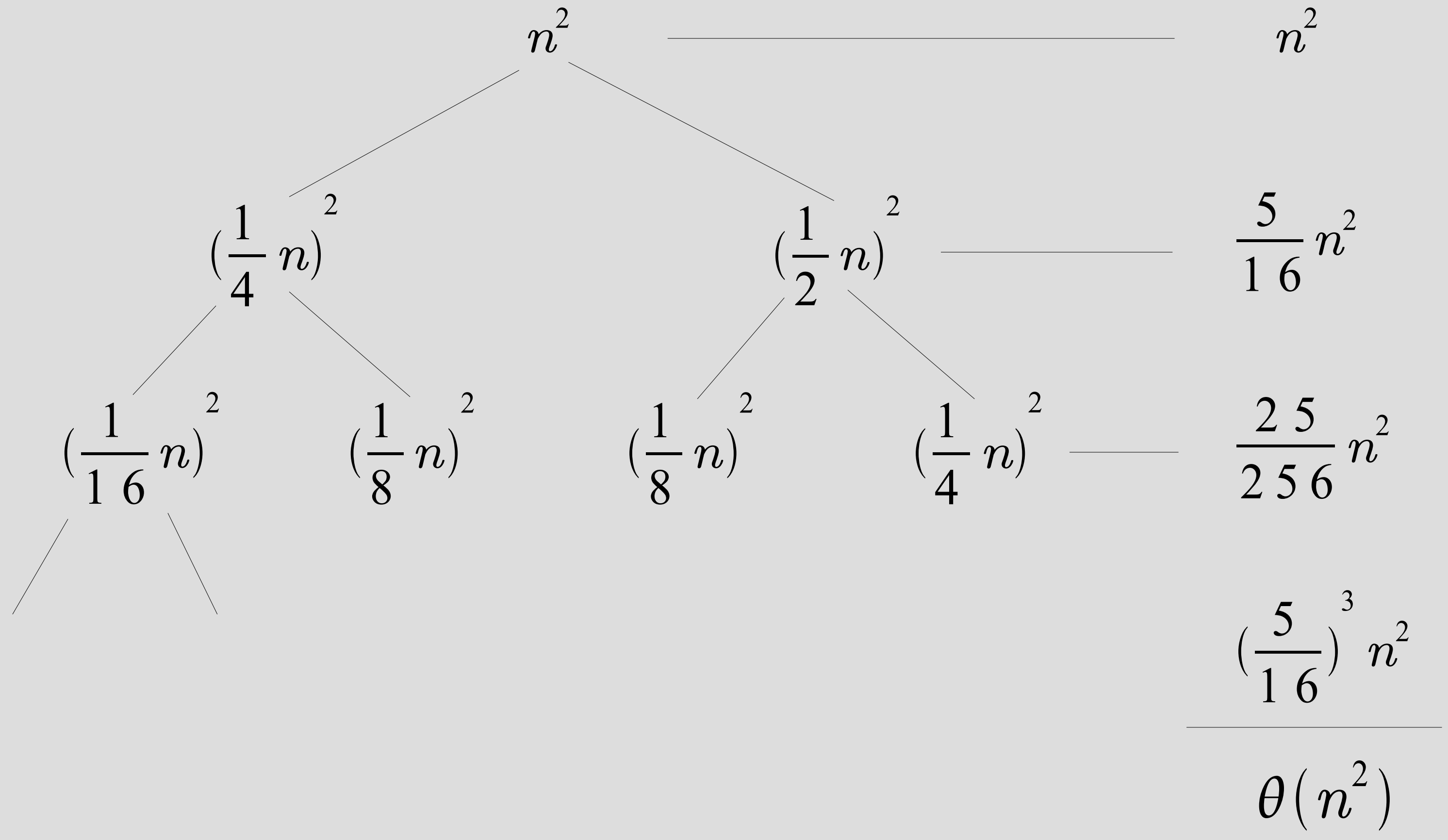
Recursion Tree

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated.
  - Good for ”guessing” asymptotic solutions to recurrences





Recursion Tree/2



Master Method

- The idea is to solve a class of recurrences that have the form $T(n) = aT(n/b) + f(n)$
- *Assumptions:* $a \geq 1$ and $b > 1$, and $f(n)$ is asymptotically positive.
- Abstractly speaking, $T(n)$ is the runtime for an algorithm and we know that
  - a subproblems of size n/b are solved recursively, each in time $T(n/b)$.
  - $f(n)$ is the cost of dividing the problem and combining the results. In merge-sort $T(n) = 2T(n/2) + \Theta(n)$.

Master Method/2

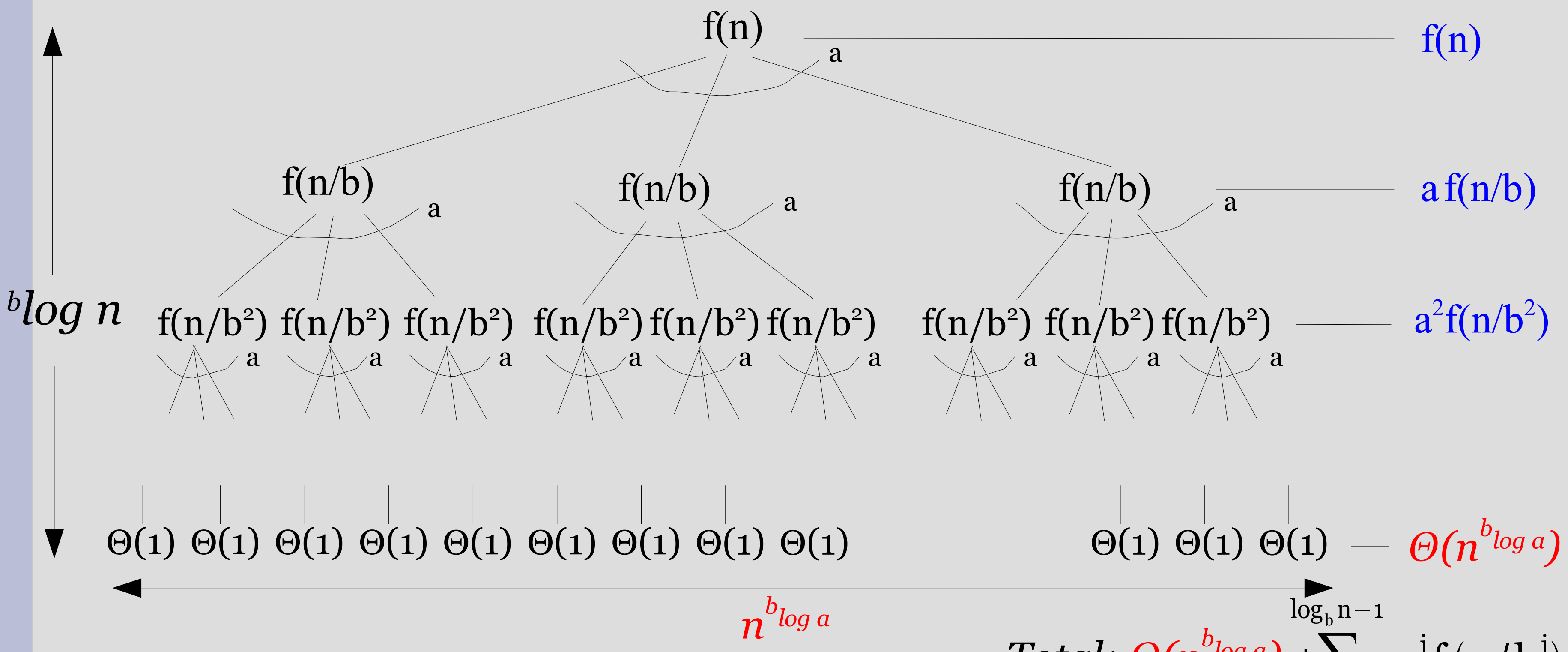
- Iterating the recurrence (expanding the tree) yields

$$\begin{aligned}T(n) &= f(n) + aT(n/b)\\ &= f(n) + af(n/b) + a^2T(n/b^2)\\ &= f(n) + af(n/b) + a^2f(n/b^2) + \ldots\\ &\quad + a^{{}^b\log n-1}f(n/a^{{}^b\log n-1}) + a^{{}^b\log n}T(1)\end{aligned}$$

$$T(n) = \sum_{j=0}^{{}^b\log n-1} a^j f(n/b^j) + \Theta(n^{{}^b\log a})$$

- The first term is a division/recombination cost (totaled across all levels of the tree).
- The second term is the cost of doing all subproblems of size 1 (total of all work pushed to leaves).

Master Method/3



f(n)
a
f(n/b)
f(n/b)
f(n/b)
a
a
a
f(n)
a f(n/b)
$a^2 f(n/b^2)$
$^b\log n$
f(n/b²) f(n/b²) f(n/b²) f(n/b²) f(n/b²) f(n/b²) f(n/b²) f(n/b²) f(n/b²)
Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1)
$\Theta(n^{^b\log a})$
$n^{^b\log a}$

Total: $\Theta(n^{^b\log a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$

Note: split into *a* parts, $^b\log n$ levels, $a^{^b\log n} = n^{^b\log a}$ leaves.

Master Method, Intuition

- Three common cases:
  1. Running time dominated by cost at leaves.
  2. Running time evenly distributed throughout the tree.
  3. Running time dominated by cost at the root.
- To solve the recurrence, we need to identify the dominant term.
- In each case compare $f(n)$ with $O(n^{{}^b\log a})$.

Master Method, Case 1

- $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant ε>0
  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor n^{ε}).
- **The work at the leaf level dominates**

$$T(n) = \Theta(n^{\log_b a})$$

Cost of all the leaves

Master Method, Case 2

- $f(n) = \Theta(n^{\log_b a})$
  - $f(n)$ and $n^{\log_b a}$ are asymptotically the same

- **The work is distributed equally throughout the tree**

$$T(n) = \Theta(n^{\log_b a} \log n)$$

(level cost) $\times$ (number of levels)

Master Method, Case 3

- $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon>0$
  - Inverse of Case 1
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$
  - Also need a "regularity" condition

$$\exists c < 1 \text{ and } n_0 > 0 \text{ such that } a f(n/b) \le c f(n) \quad \forall n > n_0$$

- **The work at the root dominates**

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

division/recombination cost

Master Theorem Summarized

Given: recurrence of the form

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

1. $f(n) = O(n^{{}^{b}\log a - \varepsilon})$

   $\Rightarrow T(n) = \Theta(n^{{}^{b}\log a})$

2. $f(n) = \Theta(n^{{}^{b}\log a})$

   $\Rightarrow T(n) = \Theta(n^{{}^{b}\log a} \log n)$

3. $f(n) = \Omega(n^{{}^{b}\log a + \varepsilon})$ and

   $a\, f(n/b) \leq \alpha\, f(n)$ for some $\alpha < 1, n > n_o$

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

Strategy

1. Extract *a*, *b*, and *f(n)* from a given recurrence
2. Determine $n^{{}^b\log a}$
3. Compare *f(n)* and $n^{{}^b\log a}$ asymptotically
4. Determine appropriate MT case and apply it

Merge sort: $T(n) = 2T(n/2) + \Theta(n)$

$a=2, b=2, f(n) = \Theta(n)$

$n^{{}^2\log 2} = n$

$\Theta(n) = \Theta(n)$

=> Case 2: $T(n) = \Theta(n^{{}^b\log a}\log n) = \Theta(n \log n)$

Examples of Master Method

```
BinarySearch(A, l, r, q):
   m := (l+r)/2
   if A[m]=q then return m
   else if A[m]>q then
        BinarySearch(A, l, m-1, q)
   else BinarySearch(A, m+1, r, q)
```

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

$a=1, b=2, f(n) = 1$

$n^{2\log 1} = 1$

$1 = \Theta(1)$

=> Case 2: $T(n) = \Theta(\log n)$

Examples of Master Method/2

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

$a=9, b=3, f(n) = n$

$n^{3\log 9} = n^2$

$n = O(n^{3\log 9 - \varepsilon})$ with $\varepsilon = 1$

=> Case 1: $T(n) = \Theta(n^2)$

Examples of Master Method/3

$T(n) = 3T(n/4) + n \log n$

$a=3, b=4, f(n) = n \log n$

$n^{4\log 3} = n^{0.792}$

$n \log n = \Omega(n^{4\log 3 + \varepsilon})$ with $\varepsilon = 0.208$

=> Case 3:

regularity condition: $af(n/b) <= cf(n)$

$af(n/b) = 3(n/4)\log(n/4) <=$

$(3/4)n \log n = cf(n)$ with $c=3/4$

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

BinarySearchRec1

- Find a number in a sorted array:
  - Trivial if the array contains one element.
  - Else **divide** into two equal halves and **solve** each half.
  - **Combine** the results.

```
INPUT:  A[1..n] – sorted array of integers, q – integer
OUTPUT: index j s.t. A[j] = q, NIL if ∀j(1≤j≤n): A[j] ≠ q
BinarySearchRec1(A, l, r, q):
   if l = r then
     if A[l] = q then return l else return NIL
   m := 悴(l+r)/2悤
   ret := BinarySearchRec1(A, l, m, q)
   if ret = NIL then return BinarySearchRec1(A, m+1, r, q)
   else return ret
```

T(n) of BinarySearchRec1

- Example: Binary Search

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

- Solving the recurrence yields

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

BinarySearchRec2

- T(n) = Θ(n) – not better than brute force!
- Better way to **conquer**:
  - Solve only one half!

```
INPUT:  A[1..n] – sorted array of integers, q – integer
OUTPUT: j s.t. A[j] = q, NIL if ∀j(1≤j≤n): A[j] ≠ q
BinarySearchRec2(A, l, r, q):
   if l = r then
      if A[l] = q then return l
      else return NIL
   m := 悙(l+r)/2悤
   if A[m] ≤ q then return BinarySearchRec2(A, l, m, q)
   else return BinarySearchRec2(A, m+1, r, q)
```

T(n) of BinarySearchRec2

- $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$

- Solving the recurrence yields

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

Example: Finding Min and Max

- Given an unsorted array, find a minimum and a maximum element in the array.

```
INPUT: A[l..r] – an unsorted array of integers, l≤r.
OUTPUT: (min,max) s.t. ∀j(l≤j≤r): A[j]≥min and A[j]≤max

MinMax(A, l, r):
    if l = r then return (A[l], A[r]) // Trivial case
    m := ⌊(l+r)/2⌋                       // Divide
    (minl,maxl) := MinMax(A, l, m)     // Conquer
    (minr,maxr) := MinMax(A, m+1, r)   // Conquer
    if minl < minr then min = minl else min = minr // Combine
    if maxl > maxr then max = maxl else max = maxr // Combine
    return (min,max)
```

Summary

- Divide and conquer
- Merge sort
- Tiling
- Recurrences
  - repeated substitutions
  - substitution
  - master method
- Example recurrences: Binary search

Next Week

- Sorting
  - HeapSort
  - QuickSort