# Lecture 1: Introduction: Good vs Bad Algorithms; Cryptography

## 1 The Central Paradigm of Computer Science

- The central ***paradigm*** in computer science is that an algorithm ***A*** is good if:
  - ***A*** runs in ***polynomial time*** in the input size n.
  - That is, ***A*** runs in time $T(\text{n}) = O(n^k)$ for some constant number k.
    - $\rightarrow$ $T(\text{n}) = 100\text{n} + 55$
    - $\rightarrow$ $T(\text{n}) = \frac{1}{2}n^2 + 999 \log n$
    - $\rightarrow$ $T(\text{n}) = 6n^7 + 900000n^2$ - $\sqrt{n}$
  - An algorithm is ***bad*** if it runs in exponential time.
    - $\rightarrow$ $T(\text{n}) = 2^n + 100n^5$
    - $\rightarrow$ $T(\text{n}) = 1.000000001^n$ - $n^3$ - n
  - An algorithm is ***good*** if it runs in ***polynomial time*** in the input size n.

e.g.

| Runtime of Algorithm | Input Size n: 10 | 100 | 1000 |
|---|---|---|---|
| $n$ | 10 | 100 | 1000 |
| $n^2$ | 100 | 10000 | 1000000 |
| $2^n$ | $10^3$ | $10^{30}$ | $10^{300}$ |

## 2 Good versus Bad (Algorithms)

- For example, consider the problem of sorting n numbers.
  - A Good Algorithm: **MergeSort** runs in time $O(n \cdot logn)$
  - A Bad Algorithm: **BruteForce Search** runs in time $O(n \cdot n!) \gg 2^n$

## 3 An Equivalent Characterization

- This central ***paradigm*** has an equivalent formulation
  - ***A*** runs in ***polynomial time*** in the input size n.
  - The input sizes that ***A*** can solve, in a fixed amount $T$ of time, ***scales multiplicatively*** with increasing computational power.

Input Sizes solved in Time T

| Runtime of Algorithm | Power = 1 | Power = 2 |
|---|---|---|
| $n$ | $T$ | $2T$ |
| $n^2$ | $\sqrt{T}$ | $\sqrt{2} \cdot \sqrt{T}$ |
| $2^n$ | $\log T$ | $1 + \log T$ |

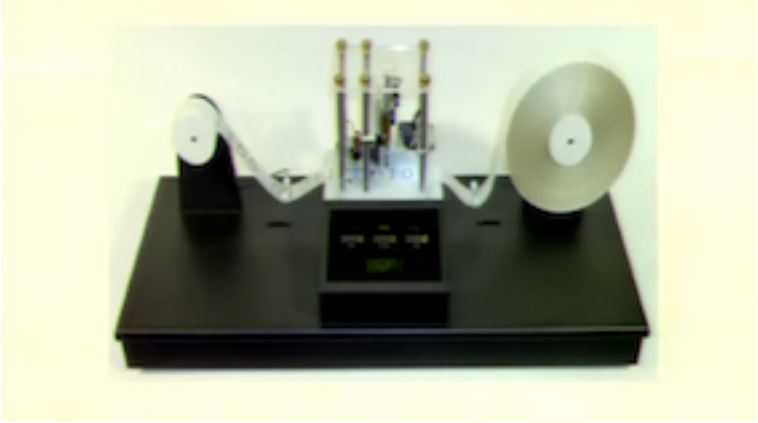

- Moore's Law: Computational power *doubles* roughly every two years.
  - → Functional time algorithms will never be able to solve large problems.

- The practical implications are perhaps simpler to understand with this latter formulation.
- Thus, improvements in hardware will *never* overcome ***bad algorithm design***.
- Indeed, the current dramatic breakthroughs in computer science are based upon batter (faster and higher performance) algorithmic techniques.

# 4 Robustness

- This measure of quality or "goodness" is ***robust***



- All reasonable models of algorithms are polynomial time equivalent.
  - Otherwise one model could perform, say, an exponential number of operations in the time another model took to perform just one.
- The standard formal model is the **Turing Machine**.

# 5 Cryptography (Just an example of the course)

- Alice wants to send Bob a message.
  But she is worried that Eve might intercept the message.
  She decides to encrypt the message $M$ as $\hat{M} = f(M)$.
  Bob can then decrypt the message via $f^{-1}(\hat{M}) = f^{-1}(f(M)) = M$
  Eve cannot understand the message.

Alice encrypts the message $M$ with the (encryption) lock $f$.
Bob decrypts the message $\hat{M}$ with the (decryption) key $f^{-1}$.
Because Eve does not have the key she cannot decipher the message.
Two major problems occur.

- Problem one:
  - Eve might be able to break the code.
    - → That is , given $\hat{M}$ she may be able to reconstruct $f$ and then $f^{-1}$.
  - Standard techniques for code-breaking include:
    - → Frequency Analysis
      - ⋆ *e.g.* "e" is the most common letter and "the" is the most common word in English.
    - → Cribs.
      - ⋆ *e.g.* German weather reports were exploited to help decode messages from the "unbreakable" **Enigma Machine**.
- Problem Two:
  - Alice and Bob need to agree on what the encryption code (lock) $f$ is.
  - But to do this they need to exchange a message discussing the code.
- Public-Key Cryptography
  - In face, Bob gives absolutely everyone a copy of the (encryption) lock.
  - Everyone can send the message and Bob has the key.
  - The lock is made public. This is called public-key cryptography.
  - But this idea sounds completely crazy.
    - → Doesn't this solution to Problem Two make Problem One inevitable?
    - → That is, if Eve has the lock $f$ then won't she use it to decode $f(M)$?
  - No, not if $f$ is hard to invert for anybody except Bob himself.
  - But do such functions $f$ that are hard to invert **exist**?
  - Yes!
- RSA Encryption

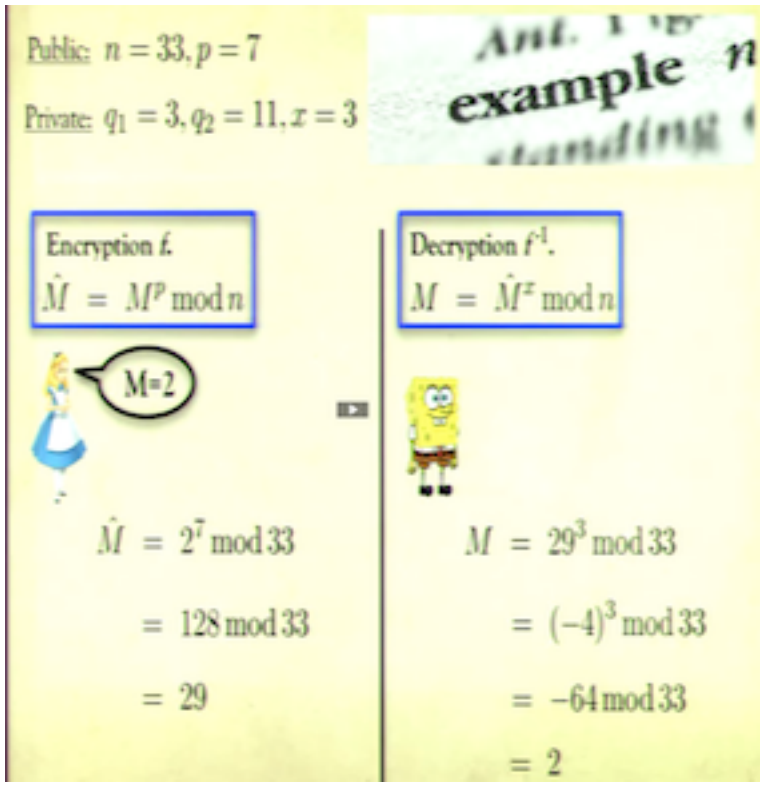
 

1. Bob chooses two large prime numbers $q_1$, $q_2$ and a large number $p$ that is co-prime to $(q_1 - 1) \cdot (q_2 - 1)$
2. Bob's public key is $(p, n)$ where $n = q_1 \cdot q_2$

   Encryption: $\hat{M} = M^p \bmod n$
3. Bob's prime key is $(q_1, q_2, x)$ where x is the inverse of p modular $(q_1 - 1) \cdot (q_2 - 1)$

   Decryption: $M = \hat{M}^x \bmod n$
   - $p = 2^{16} + 1$
   - $q_1, q_2 = 2048$ bits
4. An Example
   (a) Prime numbers $q_1$, $q_2$ and number p co-prime to $(q_1 - 1) \cdot (q_2 - 1)$
   $p = 7$, $q_1 = 3$, $q_2 = 11$ valid as gcd(7,20) = 1
   (b) Public key (p,n) where $n = q_1 \cdot q_2$
   $n = 33$
   (c) Private Key $(q_1, q_2, x)$ where x is the inverse of p mod $(q_1 - 1) \cdot (q_2 - 1)$
   $x = 33$ as $3 \cdot 7 = 1 \bmod 20$
   Public: $n = 33$, $p = 7$
   Private $q_1 = 3$, $q_2 = 11$, $x = 3$




- Is RSA Encryption Safe?
  - Public-key cryptography lies at the heart of the modern economy.
    - → Financial Services
    - → Online Shopping
    - → Secure Messaging
  - So it is extremely important that the method is safe.
  - We claim it is because:
    - → Bob has a good algorithm for decryption.
    - → Eve only has a bad algorithm for decryption.
- Bob has a Good Decryption Algorithm
  - Initially, Bob can do the following in polynomial time.
    - → Choose the primes $q_1$, $q_2$
    - → Choose a number p that is co-prime with $(q_1 - 1) \cdot (q_2 - 1)$

→ Find x the inverse (Using Euclid's Algorithm) of p and $(q_1 - 1) \cdot (q_2 - 1)$

- Using fast exponentiation, encoding and decoding is polynomial time.

Encryption: $\hat{M} = M^p \bmod n$

- Eve has a Bad Decryption Algorithm

Decryption: $M = \hat{M}^x \bmod n$

- To decrypt Eve needs to find x the inverse of p and $(q_1 - 1) \cdot (q_2 - 1)$
  - → She knows p, but does not know $q_1$, $q_2$
  - → Instead, she only knows $n = q_1 \cdot q_2$
- So to find $q_1$, $q_2$, she needs to find the prime factorization of n.
- But it is believed that finding the prime factors of a b-bit number is hard.
  - → Instead, the obvious algorithm attempts to divide n by each integer in the range $\{2,3, \ldots ,2^b\}$ ← 4096 bits
- In fact, if Eve could find x without $q_1$, $q_2$, then she could use x to find $q_1$, $q_2$. That is, she could factor n.
  ⇒ Eve only has exponential time algorithm to decrypt.

# Lecture 2: Recursive Algorithms: MergeSort; Binary Search; The Master Theorem; The Recursive Tree Method

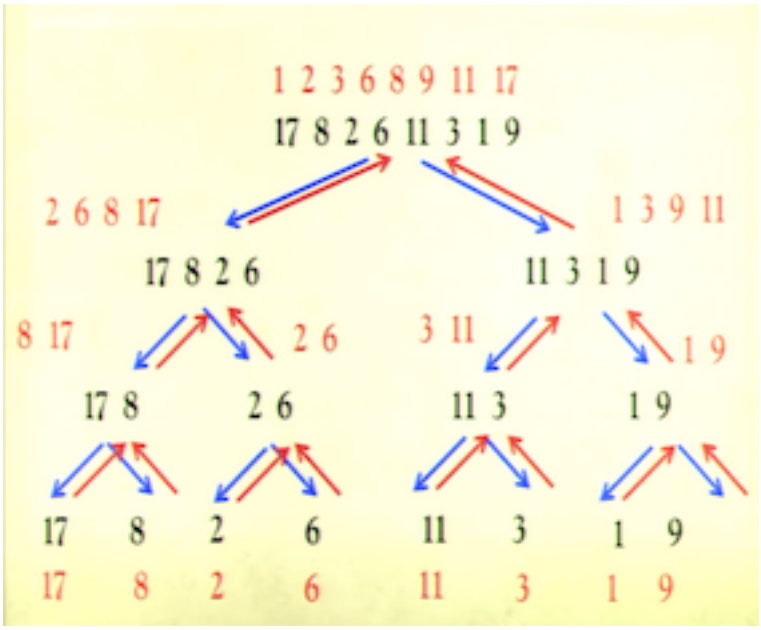
## 1 Reductions and Sub-Routines

- Solving a problem by **reducing** it (or a sub-problem of it) to another problem is the most fundamental technique in algorithm design.
- Specifically, algorithm $A$ may use another algorithm $B$ as a sub-routine.
- This has numerous advantages:
  - **Code Verification**: the correctness of $A$ is independent of $B$.
  - **Code Reuse**: a great time-saver.
- A simple but very powerful special case of this paradigm is when the algorithm calls itself!
  - This method is called **recursion**.

## 2 MergeSort

- We can sort n numbers into non-decreasing order using the following algorithm:

  MergeSort($x_1$, $x_2$, ... , $x_n$)
  If n = 1 then output $x_1$
  Else output Merge{MergeSort($x_1$, ... , $x_{\frac{n}{2}}$), MergeSort($x_{\frac{n}{2}+1}$, ... , $x_n$)}




- Two Problems:
  - Does the algorithm work?
    Yes!
    - → The algorithm calls itself on smaller instances
      - ⋆ The division process terminates with a set of base cases of size 1.
    - → MergeSort trivially works on the base cases.
    - → So, given the validity of the Merge Step, the correctness of the algorithm follows by **strong induction**.
      - ⋆ As long as base case is correct and merge step works, everything will be fine.
  - If so, is it efficient (polynomial time)?
    Yes! Look at the recursive formula.
    - → To analyze this we represent the running time $T$ (n) via a **recurrence**:

      Recursive Formula: $T(\mathrm{n}) = 2 \cdot T(\frac{n}{2}) + \mathrm{c} \cdot \mathrm{n}$

      ⋆ $2 \cdot T(\frac{n}{2})$: Recuse on two problems with half the size.

⋆ c · n: It takes linear time to merge two sorted lists.

Base Case: $T(1) = 1$

⋆ Or we can use $T(\mathrm{c}) = O(1)$ for any constant c.

→ The Running Time of MergeSort

⋆ Theorem: MergeSort runs in time $O(\mathrm{n} \cdot \log n)$

⋆ Proof:

1. By adding dummy numbers, we may assume n is a power of two: $\mathrm{n} = 2^k$
2. We can unwind the recursive formula as follows:

$$T(\mathrm{n}) = 2 \cdot T(\tfrac{n}{2}) + \mathrm{c} \cdot \mathrm{n}$$

Proof [cont.]

- But this **unwinding operation** can be repeated:

$$\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n \\
&= 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}\right) + cn \\
&= 2^2 \cdot T\left(\frac{n}{4}\right) + 2 \cdot cn \\
&= 2^2 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4}\right) + 2 \cdot cn \\
&= 2^3 \cdot T\left(\frac{n}{8}\right) + 3 \cdot cn \\
&= 2^3 \cdot \left(2 \cdot T\left(\frac{n}{16}\right) + c \cdot \frac{n}{8}\right) + 3 \cdot cn \\
&= 2^4 \cdot T\left(\frac{n}{16}\right) + 4 \cdot cn \\
&\vdots \\
&= 2^k \cdot T(1) + k \cdot cn \quad \text{— Since } n = 2^k
\end{aligned}$$

**Proof** [cont.]

- Thus we have:

$$T(n) = 2^k \cdot T(1) + k \cdot cn$$

$$= n \cdot T(1) + k \cdot cn$$

$$= n \cdot (1 + k \cdot c)$$

- But $c$ is a constant so:

$$T(n) = O(n \cdot k)$$

- Furthermore, $k = \log n$, so we get that:

$$T(n) = O(n \cdot \log n)$$

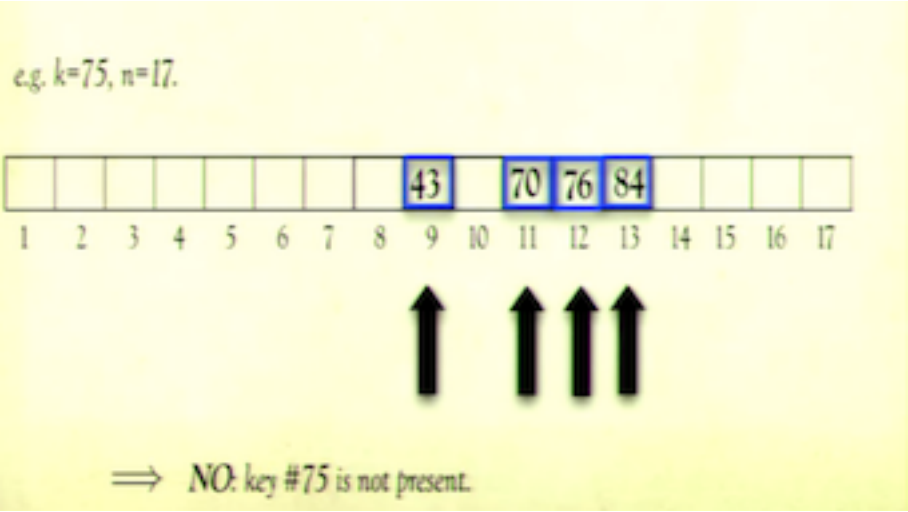
□

# 3 Binary Search

- We can search for a key k in a sorted array of cardinality n using the binary search algorithm:

  BinarySearch($a_1$, $a_2$, ... , $a_n$ : k)
  While n $>$ 0 do:
  If $a_{\frac{n}{2}}$ = k output YES
  Else if $a_{\frac{n}{2}}$ $>$ k output BinarySearch($a_1$, $a_2$, ... , $a_{\frac{n}{2}-1}$ : k)
  Else if $a_{\frac{n}{2}}$ $<$ k output BinarySearch($a_{\frac{n}{2}+1}$, ... , $a_n$: k)
  Output NO




- Does this work?
  - The validity of the binary search follows simply by strong induction.(The base case is trivially true.)
- Running Time?
  - Recurrence:

$$\text{Recursive Formula: } T(\text{n}) = T(\tfrac{n}{2}) + \text{c}$$
$$\text{Base Case: } T(1) = 1$$

  - Theorem: Binary Search runs in time $O(\log n)$
    1. By adding dummy numbers, we may assume n is a power of two: n $= 2^k$
    2. We can unwind the recursive formula as follows:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$= \left(T\left(\frac{n}{4}\right) + c\right) + c$$

$$= T\left(\frac{n}{4}\right) + 2 \cdot c$$

- Again this unwinding operation can be repeated:

$$T(n) = T\left(\frac{n}{4}\right) + 2 \cdot c$$

$$= \left(T\left(\frac{n}{8}\right) + c\right) + 2 \cdot c$$

$$= T\left(\frac{n}{8}\right) + 3 \cdot c$$

$$\vdots$$

$$= T\left(\frac{n}{2^k}\right) + k \cdot c$$

- Hence:

$$T(n) = T(1) + k \cdot c \quad \text{Since } n = 2^k$$

$$= 1 + \log n \cdot c$$

- This gives the claimed running time:

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

# 4 Divide and Conquer Algorithms

- A **divide and conquer** algorithm recursively breaks up a problem of size n in smaller sub-problems such that:
  - There are exactly a sub-problems.
  - Each sub-problem has size at most $\frac{1}{b} \cdot \mathrm{n}$
  - Once solved, the solutions to the sub-problems can be combined to produce a solution to the original problem in time $O(n^d)$
- So the run-time of a divide and conquer algorithm satisfies the recurrence:

$$T(\mathrm{n}) = \mathrm{a} \cdot T(\tfrac{n}{b}) + O(n^d)$$

- MergeSort and Binary Search are indeed **divide and conquer** algorithms.

| | Recursion Formula | $a$ | $b$ | $d$ |
|---|---|---|---|---|
| MergeSort | $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n^1)$ | 2 | 2 | 1 |
| Binary Search | $T(n) = 1 \cdot T\left(\frac{n}{2}\right) + O(n^0)$ | 1 | 2 | 0 |

# 5 Non-Military Applications of Divide and Conquer

- Divide and Conquer has many other non-military, practical applications:
  - Big Data
  - Distributed Algorithms
  - Clustering and Classification
  - MapReduce

# 6 Dummy Entries

- MergeSort actually has the recurrence:

$$\hat{T}(\mathrm{n}) = \hat{T}([\tfrac{n}{2}]) + \hat{T}([\tfrac{n}{2}]) + \mathrm{c} \cdot \mathrm{n}$$

- Recall we got around this by adding dummy entries:
  - We found $\hat{n}$ the smallest power of 2 greater than n.
  - For this case, MergeSort then does have recurrence:

$$T(\mathrm{n}) = 2 \cdot T(\tfrac{n}{2}) + \mathrm{c} \cdot \mathrm{n}$$

  - But we also have:

$$\hat{T}(n) \leq T(\bar{n}) = O(\bar{n} \cdot \log \bar{n}) = O(n \cdot \log n)$$

- Here is another way to solve the recurrence:

$$\hat{T}(n) = \hat{T}([\tfrac{n}{2}]) + \hat{T}([\tfrac{n}{2}]) + c \cdot n$$

  - As we only want to upper bound the running time, we can use:

$$\hat{T}(n) \leq \hat{T}(\tfrac{n}{2} + 1) + c \cdot n$$

    Note: This +1 does not seem to fit with our methodology, but we can fix this by applying a **domain transformation**.

- Domain Tranformation
  - For the domain transformation, simply set: $T(\mathrm{n}) = \hat{T}(\mathrm{n} + 2)$
  - Thus we have: $T(\mathrm{n}) = T(\tfrac{n}{2}) + \hat{c} \cdot \mathrm{n}$

$$
\begin{aligned}
T(n) &= \hat{T}(n+2) \\
&\leq \hat{T}\left(\frac{n+2}{2}+1\right)+c\cdot(n+2) \\
&\leq \hat{T}\left(\frac{n+2}{2}+1\right)+\hat{c}\cdot n \\
&= \hat{T}\left(\frac{n}{2}+2\right)+\hat{c}\cdot n \\
&= T\left(\frac{n}{2}\right)+\hat{c}\cdot n
\end{aligned}
$$

- Of course, we can solve this recurrence as: $T(\mathrm{n}) = O(\mathrm{n} \cdot \log n)$
- Therefore, $\hat{T}(\mathrm{n}) = T(\mathrm{n}\text{ - }2) = O(\mathrm{n} \cdot \log n)$
- As well as ceilings and floors, domain transformations can be used to *simplify* many other recurrences; e.g. removing lower order terms.

# Lecture 3: The Master Theorem; The Recursive Tree Method

## 1 Quick Review

- Recall, a divide-and-conquer algorithm recursively breaks up a problem of size n in smaller sub-problems such that:
  - There are exactly a sub-problems.
  - Each sub-problem has size at most $\frac{1}{b} \cdot \mathrm{n}$
  - Once solved, the solutions to the sub-problems can be combined to produce a solution to the original problem in time $O(n^d)$
- So the run-time of a divide and conquer algorithm satisfies the recurrence:

$$T(\mathrm{n}) = \mathrm{a} \cdot T(\tfrac{n}{b}) + O(n^d)$$

- Examples: MergeSort and Binary Search

## 2 The Master Theorem

- **The Master Theorem**: If $T(\mathrm{n}) = \mathrm{a} \cdot T(\frac{n}{b}) + O(n^d)$ for constants $\mathrm{a} > 0$, $\mathrm{b} > 1$, and $\mathrm{d} \geq 0$ then:

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \quad \text{[Case I]} \\ O(n^d \cdot \log n) & \text{if } a = b^d \quad \text{[Case II]} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{[Case III]} \end{cases}$$

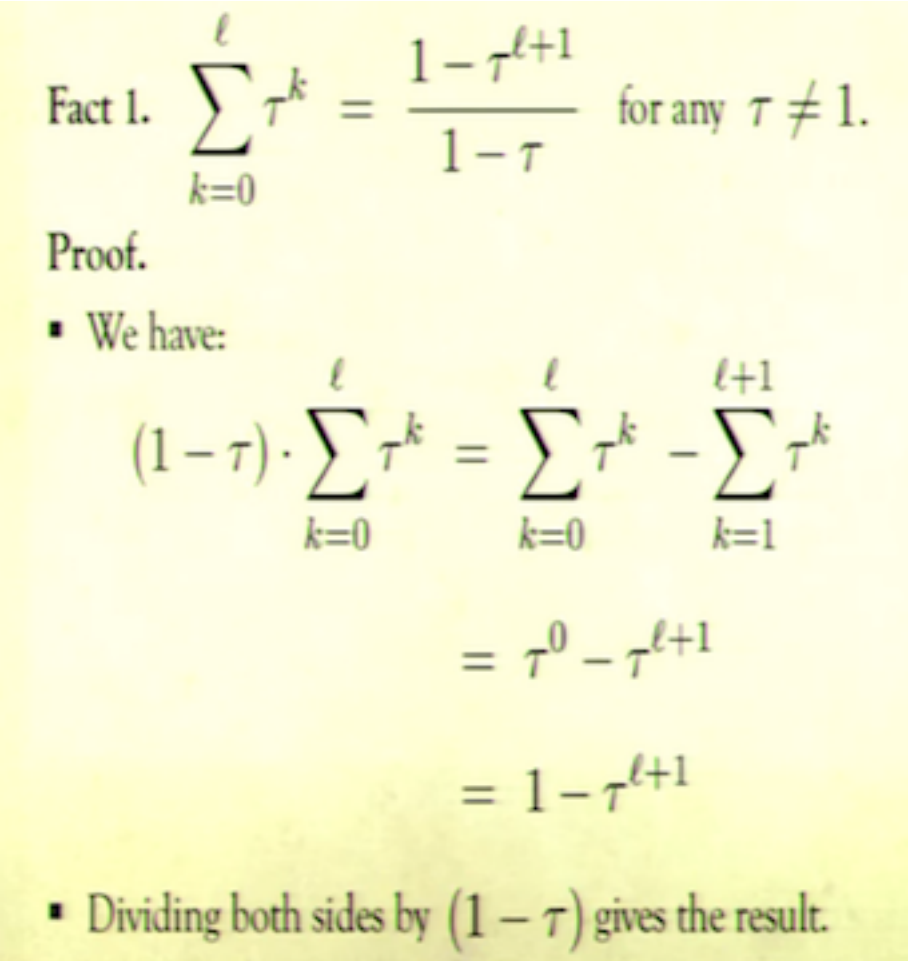
- Sanity Check: What does this give for MergeSort and Binary Search?

| | $a$ | $b$ | $d$ | Case | Runtime |
|---|---|---|---|---|---|
| MergeSort | 2 | 2 | 1 | II | $O(n \cdot \log n)$ |
| Binary Search | 1 | 2 | 0 | II | $O(\log n)$ |

# 3 Proof of The Master Theorem

- Fact One:



Fact 1. $\sum_{k=0}^{\ell} \tau^k = \frac{1-\tau^{\ell+1}}{1-\tau}$ for any $\tau \neq 1$.

Proof.

- We have:

$$(1-\tau)\cdot\sum_{k=0}^{\ell}\tau^k = \sum_{k=0}^{\ell}\tau^k - \sum_{k=1}^{\ell+1}\tau^k$$

$$= \tau^0 - \tau^{\ell+1}$$

$$= 1-\tau^{\ell+1}$$

- Dividing both sides by $(1-\tau)$ gives the result.

- Fact Two:

**Fact 2.** $x^{\log_b y} = y^{\log_b x}$ for any base $b$.

**Proof.**

$\log_b z^p = p \cdot \log_b z$

- Observe that, by the *power rule of logarithms*, we have:

$$\log_b x \cdot \log_b y = \log_b(y^{\log_b x})$$

- Similarly:

$$\log_b x \cdot \log_b y = \log_b(x^{\log_b y})$$

- Putting this together gives

$$\log_b(y^{\log_b x}) = \log_b(x^{\log_b y})$$

$$\Longrightarrow \quad x^{\log_b y} = y^{\log_b x}$$

- Proof of the Master Theorem

**Proof.**

- We may assume $n$ is a power of $b$: $n = b^\ell$
- So we have:

$$T(n) = n^d + a \cdot \left(\frac{n}{b}\right)^d + a^2 \cdot \left(\frac{n}{b^2}\right)^d + \cdots + a^\ell \cdot \left(\frac{n}{b^\ell}\right)^d$$

$$= n^d \cdot \left(1 + a \cdot \left(\frac{1}{b}\right)^d + a^2 \cdot \left(\frac{1}{b^2}\right)^d + \cdots + a^\ell \cdot \left(\frac{1}{b^\ell}\right)^d\right)$$

$$= n^d \cdot \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^\ell\right)$$

**Proof** [cont.] $T(n) = n^d \cdot \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^\ell\right)$

Case I: $\frac{a}{b^d} < 1$

- Set $\tau = \frac{a}{b^d}$
- Then: $T(n) = n^d \cdot \sum_{k=0}^{\ell} \tau^k$
- Applying Fact 1, we know that:

$$\sum_{k=0}^{\ell} \tau^k = \frac{1-\tau^{\ell+1}}{1-\tau} \leq \frac{1}{1-\tau} = O(1)$$

As a, b, and d are constants so is 1-τ.

- Therefore:

$$T(n) \leq n^d \cdot \frac{1}{1-\frac{a}{b^d}} = n^d \cdot \frac{b^d}{b^d - a} = O(n^d)$$

**Proof** (cont.) $T(n) = n^d \cdot \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^\ell\right)$

Case II: $\frac{a}{b^d} = 1$

- Then: $T(n) = n^d \cdot (\ell + 1)$
- But $n = b^\ell$ so $\ell = \log_b n$.
- As $b$ is a constant greater than one, this gives: $T(n) = O(n^d \cdot \log n)$

Case III: $\frac{a}{b^d} > 1$

- Again set $\tau = \frac{a}{b^d}$
- Then: $T(n) = n^d \cdot \sum_{k=0}^{\ell} \tau^k$
- Applying Fact 1 gives:

$$\sum_{k=0}^{\ell} \tau^k = \frac{\tau^{\ell+1} - 1}{\tau - 1} \leq \frac{\tau^{\ell+1}}{\tau - 1} = O(\tau^{\ell+1}) = O(\tau^\ell)$$

As $a$, $b$, and $d$ are constants so is $\tau$-1.

**Proof** [cont.]

- Thus: $T(n) = O(n^d \cdot \tau^\ell)$
- Observe that:

$$n^d \cdot \tau^\ell = n^d \cdot \left(\frac{a}{b^d}\right)^\ell$$

$$= \left(\frac{n}{b^\ell}\right)^d \cdot a^\ell$$

$$= 1 \cdot a^\ell \quad \text{— As } n = b^\ell.$$

$$= a^{\log_b n}$$

$$= n^{\log_b a} \quad \text{— By Fact 2.}$$

- This gives the final case:

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

- Specifically, what matters is **not** the statement of the Master Theorem but the **ideas** underlying its proof.
  - First, if we understand the proof then we can easily reconstruct the theorem.
  - Second, if we understand the proof then we can easily apply the method to a much broader class of problems. For example:
    - → The sub-problems have different sizes.

e.g. The Deterministic Selection Algorithm.

$$T(n) \leq T(\frac{7n}{10}) + T(\frac{n}{5}) + O(n)$$

→ The combination function is not of the form $f(\mathrm{n}) = n^d$.

e.g. Euclid's Greatest Common Divisor Algorithm.

$$T(n) = T(\frac{n}{2}) + O(\log n)$$

→ The parameters a, b, and d are not constants.

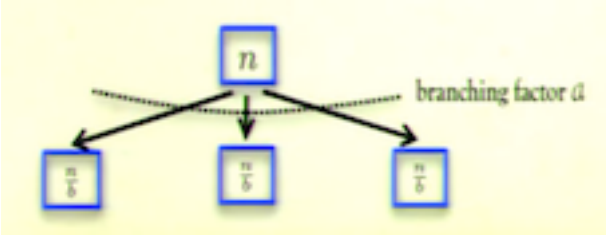
e.g. $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + O(n^{\frac{1}{\log \log n}})$

# 4 The Recursion Tree Method

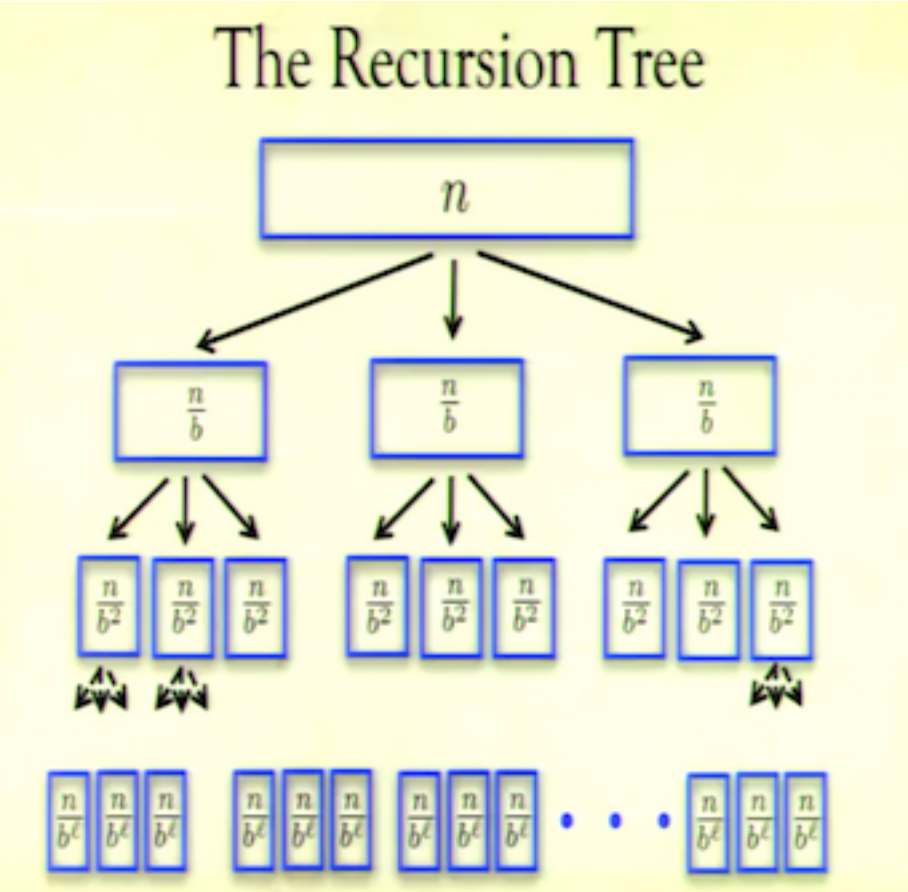
- The Master Theorem is a special case the the **recursion tree method**.
- Specifically, we model the *divide and conquer* recursive formula by a tree:

$$T(\mathrm{n}) = \mathrm{a} \cdot T(\tfrac{n}{b}) + O(n^d)$$

  - The **root** node of the trees has a label n.
  - The root has a children each with label $\frac{n}{b}$.

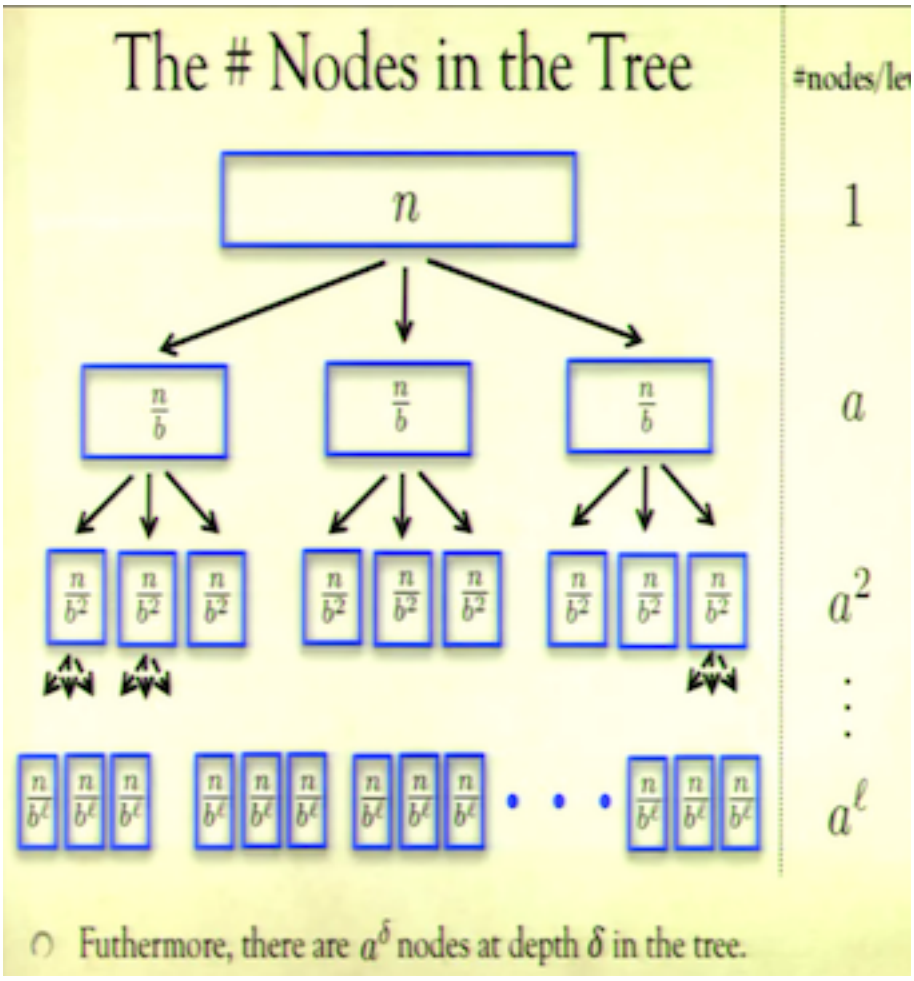



- This pattern then repeats at the children, then grandchildren, etc.



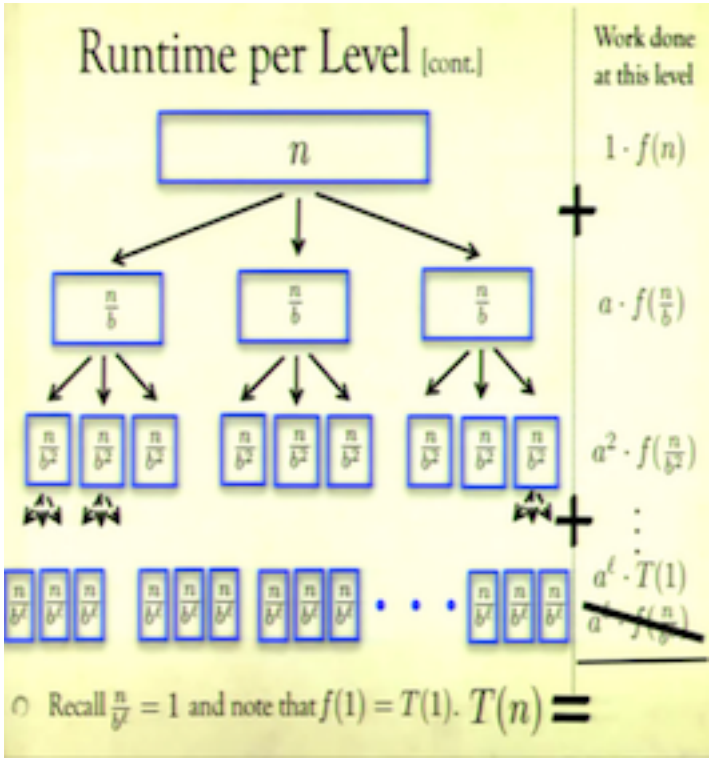

- This process stops at the **leaves** (base cases) which have label $\frac{n}{b^l} = 1$. (As n $= b^l$)

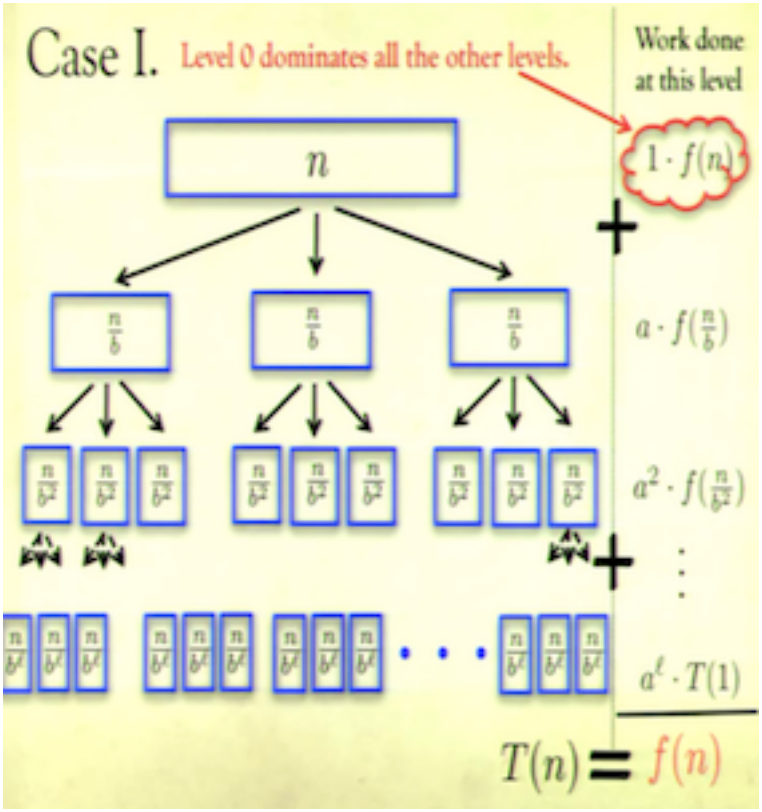



- How much time do we spend at each level?

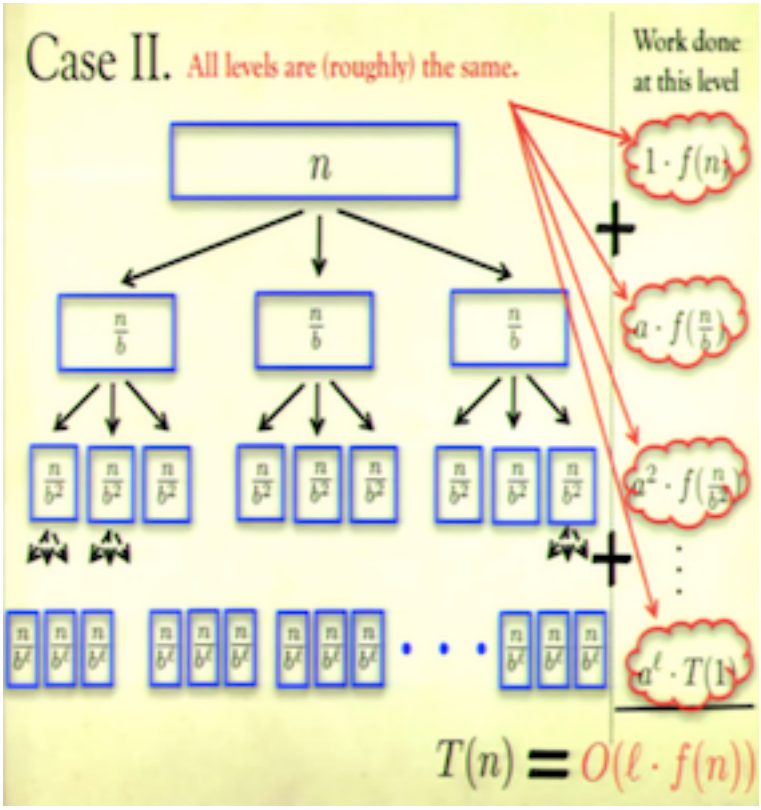
 


Runtime per Level [cont.]
Work done at this level
n
1 · f(n)
a · f(n/b)
a² · f(n/b²)
a^ℓ · T(1)
Recall n/b^ℓ = 1 and note that f(1) = T(1). T(n) =

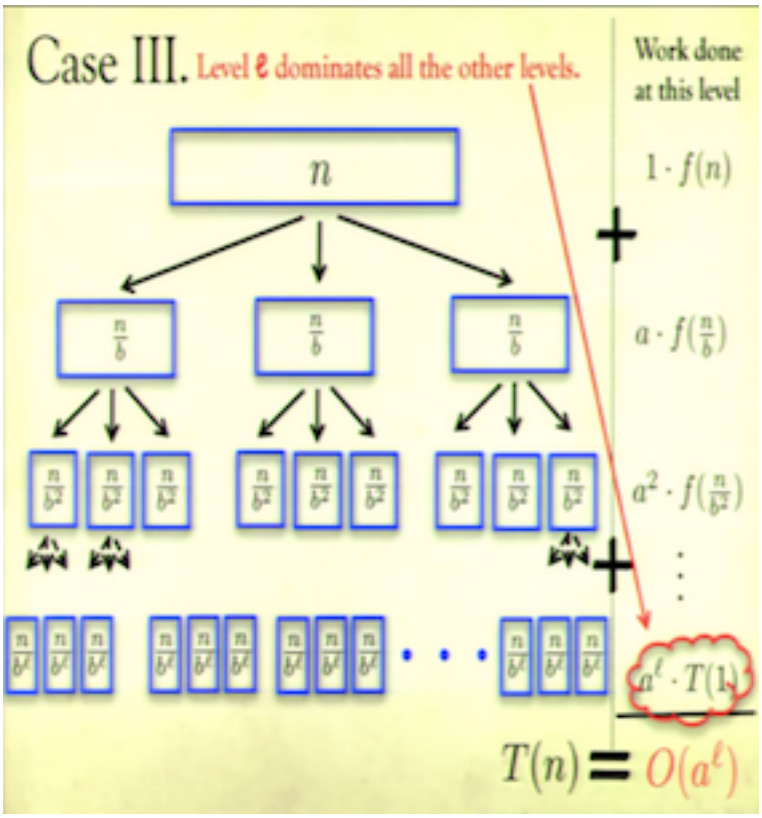
 


Case I.
Level 0 dominates all the other levels.
Work done at this level
$n$
$1 \cdot f(n)$
$a \cdot f(\frac{n}{b})$
$a^2 \cdot f(\frac{n}{b^2})$
$a^\ell \cdot T(1)$
$T(n) = f(n)$

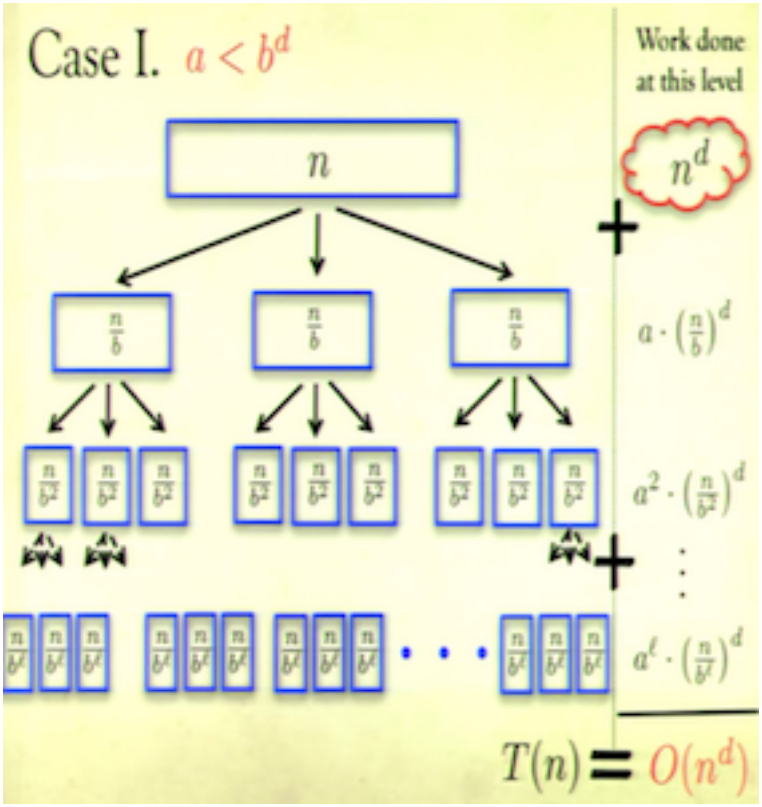

Case II. All levels are (roughly) the same.
Work done at this level
n
$1 \cdot f(n)$
$a \cdot f(\frac{n}{b})$
$a^2 \cdot f(\frac{n}{b^2})$
$a^\ell \cdot T(1)$
$T(n) = O(\ell \cdot f(n))$

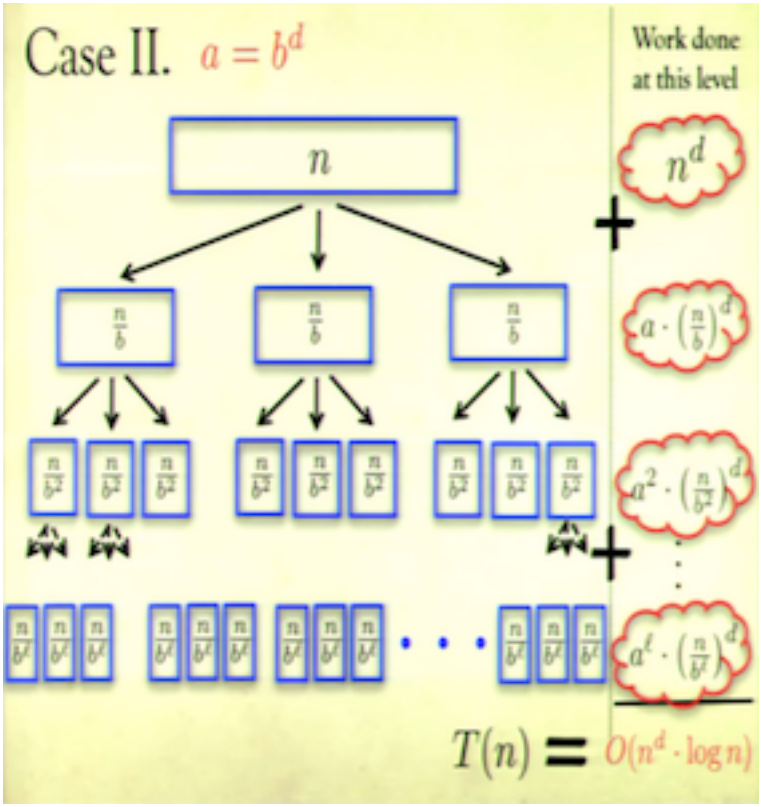



- This gives us the proof of the Master Theorem:

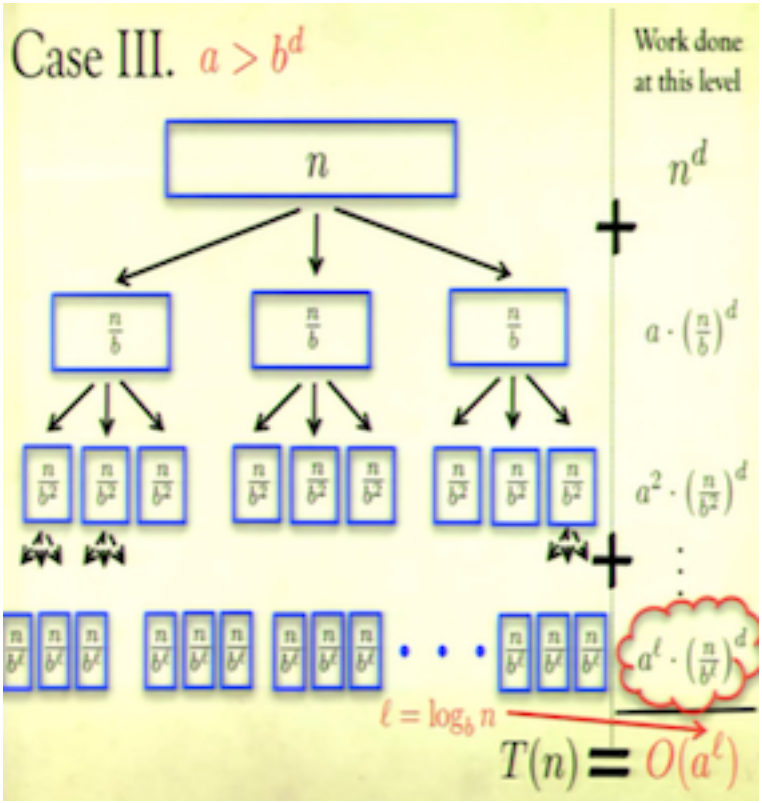
 


Case I. $a < b^d$
Work done at this level
$n$
$n^d$
$\frac{n}{b}$
$a \cdot \left(\frac{n}{b}\right)^d$
$\frac{n}{b^2}$
$a^2 \cdot \left(\frac{n}{b^2}\right)^d$
$\frac{n}{b^l}$
$a^l \cdot \left(\frac{n}{b^l}\right)^d$
$T(n) = O(n^d)$


Case II. $a = b^d$
Work done at this level
$n$
$n^d$
$\frac{n}{b}$
$a \cdot \left(\frac{n}{b}\right)^d$
$\frac{n}{b^2}$
$a^2 \cdot \left(\frac{n}{b^2}\right)^d$
$\frac{n}{b^l}$
$a^l \cdot \left(\frac{n}{b^l}\right)^d$
$T(n) = O(n^d \cdot \log n)$


Case III. $a > b^d$
Work done at this level
$n$
$n^d$
$\frac{n}{b}$
$a \cdot \left(\frac{n}{b}\right)^d$
$\frac{n}{b^2}$
$a^2 \cdot \left(\frac{n}{b^2}\right)^d$
$\frac{n}{b^\ell}$
$a^\ell \cdot \left(\frac{n}{b^\ell}\right)^d$
$\ell = \log_b n$
$T(n) = O(a^\ell)$

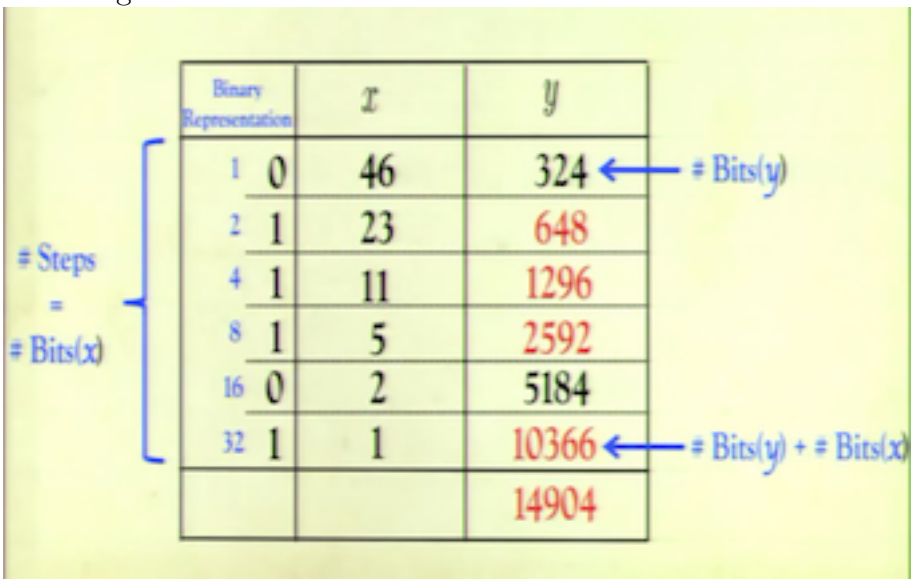
# Lecture 4: Recursive Algorithms: Fast Multiplication; Fast Matrix Multiplication

## 1 Multiplication

- Grade School Multiplication
  - Perform $n^2$ multiplications to multiply two n-digit numbers
    $\rightarrow$ Long multiplication has running time $\Omega(n^2)$.
- Russian Peasant Multiplication
  Mult(x, y)
  **If** x = 1 **then** output y
  **If** x is odd **then** output y + Mult($[\frac{x}{2}]$,2y)
  **If** x is even **then** output Mult($\frac{x}{2}$,2y)

| Binary Representation | | $x$ | $y$ |
|---|---|---|---|
| 1 | 0 | 46 | 324 |
| 2 | 1 | 23 | 648 |
| 4 | 1 | 11 | 1296 |
| 8 | 1 | 5 | 2592 |
| 16 | 0 | 2 | 5184 |
| 32 | 1 | 1 | 10368 |
| | 46 | | 14904 |

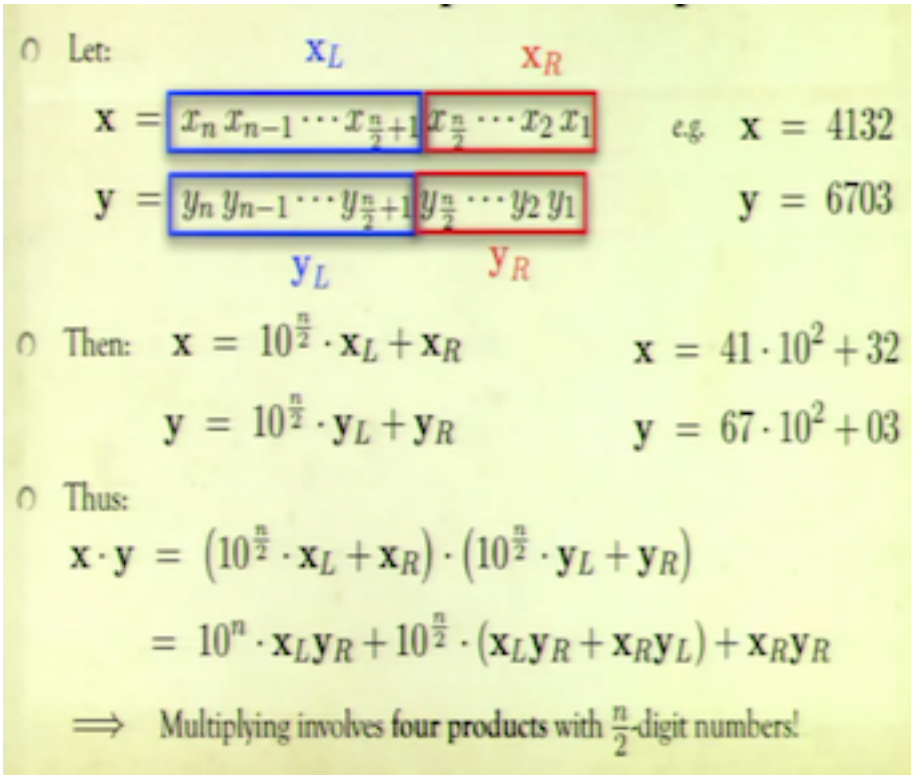
- This method does work. Base case (x = 1) is verified. The step when x is even also works. The only tricky step is that x is odd when you divide x by 2, you actually divide x by 2 and minus a half, so that you need to add one y back.
- How long does it take?




→ There are n iterations, so we add up to n numbers with at most 2n digits each.

→ Runtime = $O(n^2)$

# 2 Divide and Conquer Multiplication




- How long does this algorithm take?
  - The recursive formula for the running time is:
    $$T(\mathrm{n}) = 4 \cdot T(\tfrac{n}{2}) + O(\mathrm{n})$$
    Note: by padding some zeroes in front of the number, we can assume n is a power of 2.
  - Thus we have a = 4, b = 2, and d = 1.
  - This is Case 3 of the Master Theorem.
  - Runtime = $O(n^{\log_2 4}) = O(n^2)$

# 3 Gauss's Complex Number Multiplication

- Gauss considered the product of complex numbers:

$$(\mathrm{a} + \mathrm{b} \cdot i) \cdot (\mathrm{c} + \mathrm{d} \cdot i) = \mathrm{ac} - \mathrm{bd} + (\mathrm{bc} + \mathrm{ad}) \cdot i$$

- This seems to require taking **four products**, but he observed that:

$$(\mathrm{bc} + \mathrm{ad}) = (\mathrm{a} + \mathrm{b}) \cdot (\mathrm{c} + \mathrm{d}) - \mathrm{ac} - \mathrm{bd}$$

- So we can calculate ac and bd then we only need to perform only one more product to find (bc + ad), namely (a + b) · (c + d).
  - Multiplying two complex numbers involves only **three products**! We got extra addition here, but addition is much cheaper than multiplication when we think about the running time.

# 4 Application of Gauss's Complex Number Multiplication

- We can simply replace **i** by $10^{\frac{n}{2}}$

$$(a + b \cdot \mathbf{i}) \cdot (c + d \cdot \mathbf{i}) = ac - bd + (bc + ad) \cdot \mathbf{i}$$

$$(x_R + x_L \cdot 10^{\frac{n}{2}}) \cdot (y_R + y_L \cdot 10^{\frac{n}{2}}) = x_R y_R + x_L y_L \cdot 10^{n} + (x_L y_R + x_R y_L) \cdot 10^{\frac{n}{2}}$$

- But now we have:

$$(x_L y_R + x_R y_L) = (x_R + x_L)(y_R + y_L) - x_R y_R - x_L y_L$$

$$(x_L y_R + x_R y_L) = x_R y_R + x_L y_L - (x_R - x_L) \cdot (y_R - y_L)$$

  - Multiplying involves only **three products** with $\frac{n}{2}$ digit numbers!

- The recursive formula for the running time is:

$$T(\mathrm{n}) = 3 \cdot T(\tfrac{n}{2}) + O(\mathrm{n})$$

- Thus we have a = 3, b = 2, and d = 1.

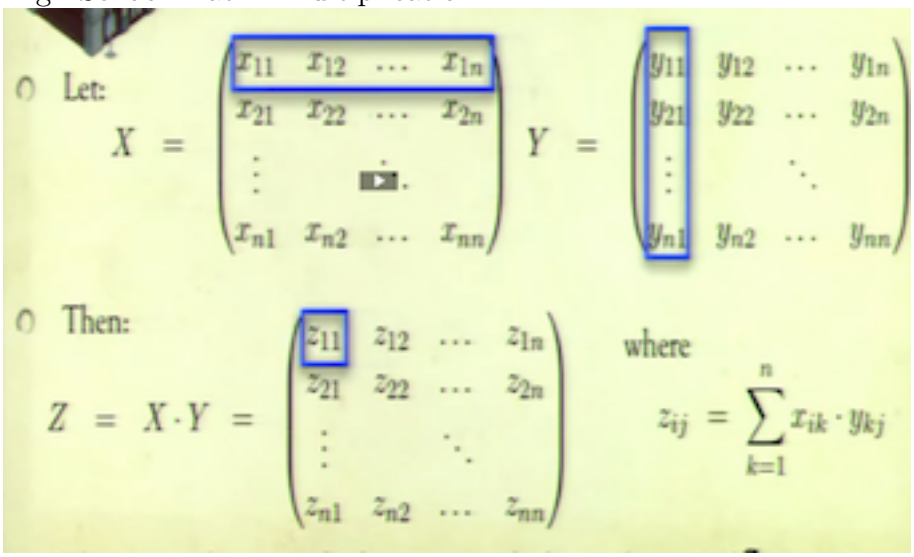
- This is Case 3 of the Master Theorem.
- Runtime $= O(n^{\log_2 3}) = O(n^{1.59})$
- So we can multiply two n-bit numbers using less than $n^2$ operations!

# 5 Fast Fourier Transforms

- We can multiply two n-bit numbers in time $O(\mathrm{n} \cdot \log n)$ using a **Fast Fourier Transform**.
- More generally, FFTs can be used to multiply two **polynomial functions**.
  - This has a vast number of applications, for example in *image compression* and *signal processing*.
  - Indeed, it has been described by Strang as "the most important numerical algorithm of our lifetime."
- We might study FFTs later in the course.

# 6 Matrix Multiplication

- High School Matrix Multiplication




Then we perform n multiplications to calculate each entry of $Z$.

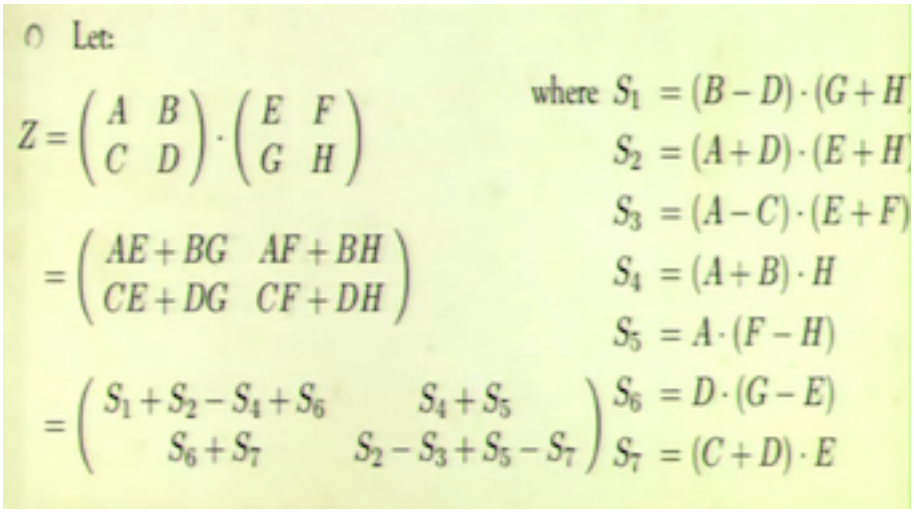
- Runtime = $\Omega(n^3)$

# 7 Divide and Conquer Matrix Multiplication

- We can also multiply matrices using **divide and conquer**.
  - Let: $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ $Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$
  - Then: $Z = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$
- Multiplying involves **eight products** with $\frac{n}{2} \times \frac{n}{2}$ matrices!
- The recursive formula for the running time is:
$$T(\mathrm{n}) = 8 \cdot T(\tfrac{n}{2}) + O(n^2)$$
- Thus we have a = 8, b = 2, and d = 2.
- This is Case 3 of the Master Theorem.
- Runtime = $O(n^{\log_2 8}) = O(n^3)$
- No improvement!

# 8 An Algebraic Trick for Matrix Multiplication




- Multiplying involves **seven products** with $\frac{n}{2} \times \frac{n}{2}$ matrices!
  Note: Even though we now have 18 additions, the runtime is still dominated by the number of multiplications. Having additions would not hurt.
- The recursive formula for the running time is:
$$T(\mathrm{n}) = 7 \cdot T(\tfrac{n}{2}) + O(n^2)$$
- Thus we have a = 7, b = 2, and d = 2.
- This is Case 3 of the Master Theorem.
- Runtime = $O(n^{\log_2 7}) = O(n^{2.81})$
- This divide and conquer matrix multiplication algorithm was designed by Strasser (1969). Since then faster algorithms (in theory) have been developed:
  - **Theorem**: There is a matrix multiplication algorithm that runs in time $O(n^{2.37})$
  - **Open Problem**: Is matrix multiplication $O(n^{2+\epsilon})$ time for any constant $\epsilon > 0$? Note: There is no way you can beat this because just to read the matrix x you have to do $x^2$ amount of work to read entries in x simply for y. Getting running time like this seems crazy but possible.

# 9 Fast Exponentiation

- Suppose we want to compute $x^n$
  - The slow way $x \cdot x \cdot \cdots x$ requires n-1 multiplications.
  - A faster way is to use the fact that $x^n = x^{[\frac{n}{2}]} \cdot x^{[\frac{n}{2}]}$
- The latter method gives the following **recursive algorithm**.

Fast Exponentiation
**FastExp**(x,n)
**If** n = 1 **then** output x
**Else**
  **If** n is even **then** output **FastExp**$(x, [\frac{n}{2}])^2$
  **If** n is odd **then** output **x** $\cdot$ **FastExp**$(x, [\frac{n}{2}])^2$

- The number of multiplications used in Fast Exponentiation satisfies:

$$T(\text{n}) \leq T([\tfrac{n}{2}]) + 2 \Rightarrow T(\text{n}) = T(\tfrac{n}{2}) + O(1)$$

- Thus we have a = 1, b = 2, and d = 0.
- This is Case 2 of the Master Theorem.
- Runtime = $O(n^0 \cdot logn) = O(logn)$

# Lecture 5: Recursive Algorithms: The Median; Randomized Selection; Deterministic Selection

## 1 The Median Problem

- How do we find the **median** of a set $S = \{x_1, x_2, \cdot \cdot \cdot, x_n\}$?
  - We could sort the list and then output the $[\frac{n}{2}]$th number.
  - Using **Mergesort**, or otherwise, this will take time $O(\mathrm{n} \cdot \log n)$.
- Is there a **faster** way to find the median?
  - We only need the median number. Sorting all numbers seems overkill. We cannot do better than $O(\mathrm{n})$ since we need to read n numbers in linear time.

## 2 The Selection Problem

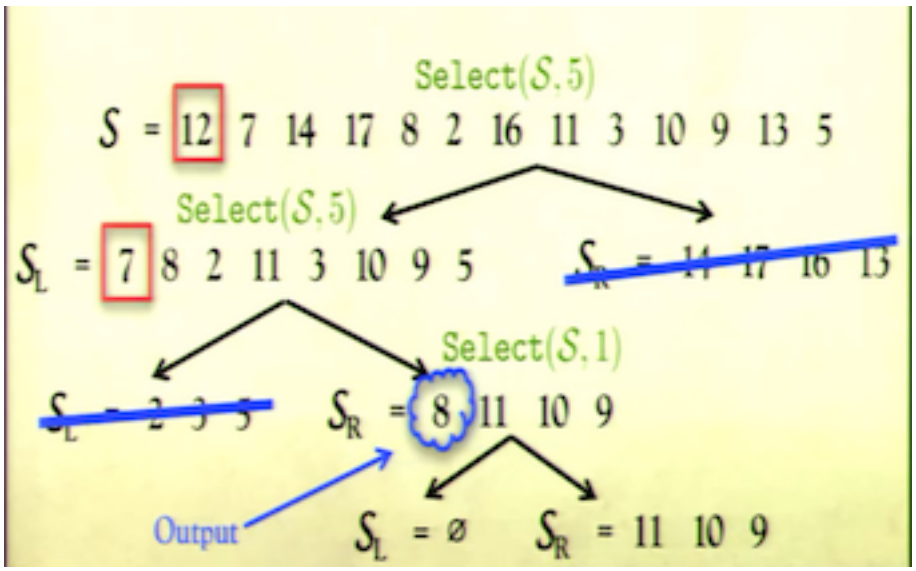
- How do we find the $k^{th}$ smallest number in $S = \{x_1, x_2, \cdot \cdot \cdot, x_n\}$?
  - Again, we could **sort** the list and then output the $k^{th}$ number.
  - Using **Mergesort**, this takes time $O(\mathrm{n} \cdot \log n)$.
- We can do this much faster using recursion...

## The Selection Algorithm

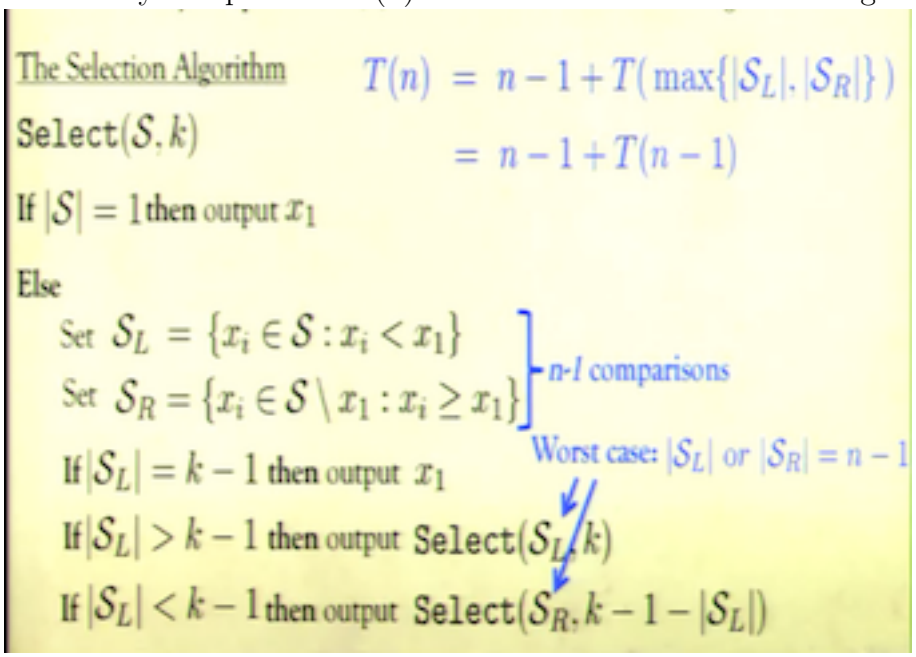
**Select**($S$, k)
**If** $|S| = 1$ **then** output $x_1$
**Else**
  Set $S_L = \{x_i \in S : x_i < x_1\}$
  Set $S_R = \{x_i \in S \setminus x_1 : x_i \geq x_1\}$
  **If** $|S_L|$ = k - 1 **then** output $x_1$
  **If** $|S_L| >$ k - 1 **then** output **Select**($S_L$, k)
  **If** $|S_L| <$ k - 1 **then** output **Select**($S_R$, k - 1 - $|S_L|$)

Note:

1. If $|S_L|$ = k - 1, then $x_1$ is the $k^{th}$ smallest number.
2. On the other hand, suppose smallest $S_L$ contains at least (k - 1) elements, which means it gets k elements, in other words, $k^{th}$ smallest element is actually in $S_L$. The $k^{th}$ smallest element of $S$ must also be the $k^{th}$ smallest element of $S_L$.
3. $S_L$ union $x_1$ has the most (k - 1) elements, which means the $k^{th}$ smallest element must be in the set $S_R$. Since everything in $S_R$ is at least bigger than in $x_1$, so that it is also bigger than in $S_L$, to find the $k^{th}$ smallest element, we need (k - 1 - $|S_L|$).



- How many comparisons $T(\mathrm{n})$ does this recursive selection algorithm make?




- So in the **worst case** we have is:

$$\begin{aligned} T(\mathrm{n}) &= (\mathrm{n}\text{ - }1) + T(\mathrm{n}\text{ - }1) \\ &= (\mathrm{n}\text{ - }1) + (\mathrm{n}\text{ - }2) + T(\mathrm{n}\text{ - }2) \\ &\;\;\vdots \\ &= (\mathrm{n}\text{ - }1) + (\mathrm{n}\text{ - }2) + (\mathrm{n}\text{ - }3) + \cdots + 2 + 1 \\ &= \tfrac{1}{2}\mathrm{n}(\mathrm{n}\text{ - }1) \\ &= \Omega(n^2) \end{aligned}$$

- This is terrible - sorting would have been faster!
- How to fix it? Use Balanced Pivots!
  - The problem is the algorithm repeatedly **pivots** on the first number in the current list.
    - → But if we are unlucky this pivot could be very **unbalanced**. That is:
      $\max\{|S_L|, |S_R|\} \approx \mathrm{n}$
      $\min\{|S_L|, |S_R|\} \approx 0$

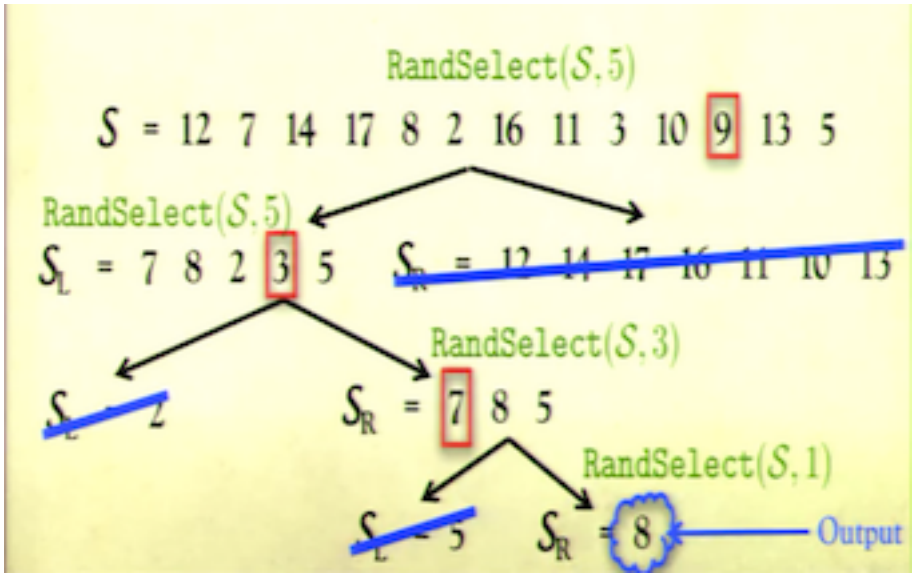
 

- What we would like is to select a pivot that is **balanced**. That is:
  $\max\{|S_L|, |S_R|\} \approx \frac{n}{2}$
  $\min\{|S_L|, |S_R|\} \approx \frac{n}{2}$

- Randomization
  - The current algorithm is **deterministic** in the choice of the pivot.
  - To fix the problem we consider a **randomized** implementation.
    - → Do not pivot deterministically on $x_1$.
    - → Instead choose the pivot at random from $S = \{x_1, x_2, \cdots, x_n\}$

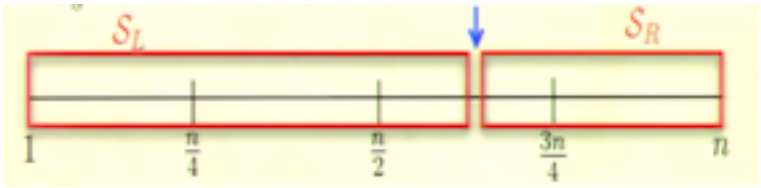
Randomized Selection

**RandSelect**($S$, k)
**If** $|S| = 1$ **then** output $x_1$
**Else**
Pick a random pivot $x_\tau \in \{x_1, x_2, \cdots, x_n\}$
Set $S_L = \{x_i \in S : x_i < x_\tau\}$
Set $S_R = \{x_i \in S \setminus x_\tau : x_i \geq x_\tau\}$
**If** $|S_L|$ = k - 1 **then** output $x_\tau$
**If** $|S_L|$ > k - 1 **then** output **RandSelect**($S_L$, k)
**If** $|S_L|$ < k - 1 **then** output **RandSelect**($S_R$, k - 1 - $|S_L|$)



- Good vs Bad Pivots
  - Imagine all the numbers in the **sorted** order.


  - With probability $\frac{1}{2}$ the pivot $x_\tau$ lies between the 1st and 3rd quartiles.
    - → We say that such a pivot is good.
    - → We say that such a pivot is bad.
  - **Key Observation**: If the pivot is good then $\max\{|S_L|, |S_R|\} \leq \frac{3}{4} \cdot \mathrm{n}$




- Expected Runtime
  - In the worst case, the *randomized algorithm* will pick the worst pivot! The probability of its happening is very small.
  - So, for randomized algorithms, we are always interested in the **expected runtime** $\bar{T}(n) = E(T(n))$, not the worst case run time.
  - Using our observation, we then have that:

$$\bar{T}(n) \leq \frac{1}{2} \cdot \bar{T}(\frac{3n}{4}) + \frac{1}{2} \cdot \bar{T}(n) + O(n)$$

Note:

1. The first term: the probability of $\frac{1}{2}$ comes from I make a good pivot. If I make a good pivot, I know both of my subsets will have at most the size of $\frac{3n}{4}$.

2. The second term: If I get a bad pivot, the size of the next problem might be (n - 1), which is $\bar{T}(n)$ in the worst case.

$$\bar{T}(n) \leq \frac{1}{2} \cdot \bar{T}(\frac{3n}{4}) + \frac{1}{2} \cdot \bar{T}(n) + O(n)$$

$$\Rightarrow \frac{1}{2} \cdot \bar{T}(n) \leq \frac{1}{2} \cdot \bar{T}(\frac{3n}{4}) + O(n)$$

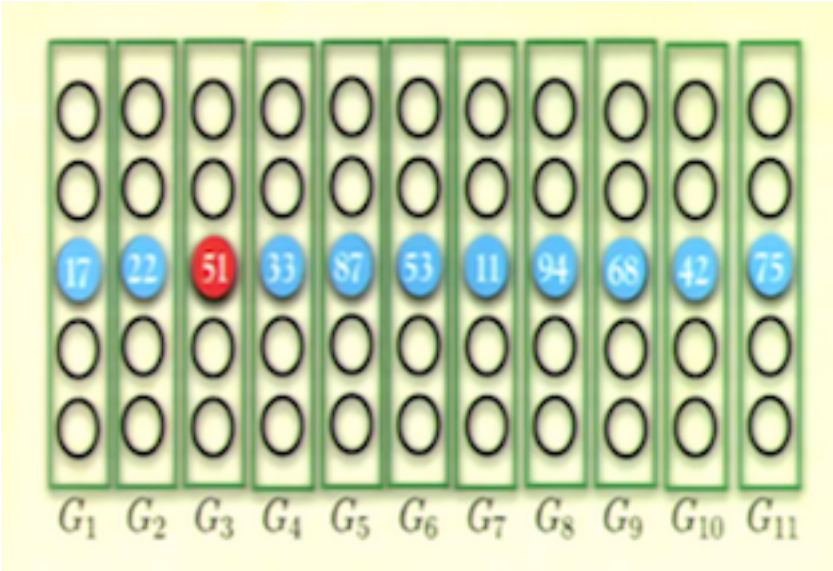
$$\Rightarrow \quad \bar{T}(n) \leq \bar{T}(\frac{3n}{4}) + O(n)$$

- Apply the Master Theorem
  - → a = 1, b = $\frac{4}{3}$, and d = 1
  - → This is Case 1 of the Master Theorem.
  - → Runtime = $O(n^d) = O(n)$

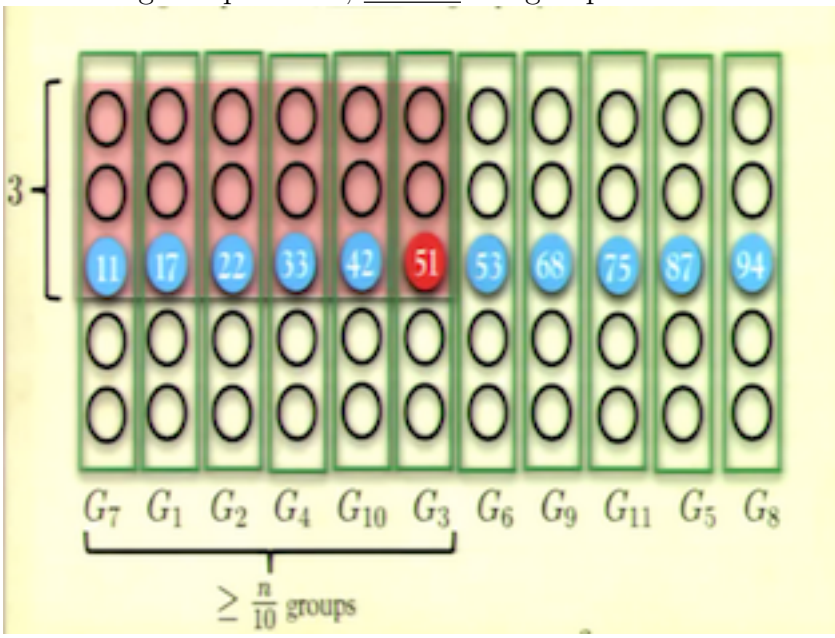
# 3 Deterministic Selection

- So we have a **linear time** randomized algorithm for the selection problem.
- Is there a linear time deterministic algorithm?
- To do this, what we need is a deterministic method to find a **good pivot**.
- The idea is to find "the median of the medians."
- Divide $S = \{x_1, x_2, \cdots, x_n\}$ into groups of cardinality five:
  $G_1 = \{x_1, \cdots, x_5\}, G_2 = \{x_6, \cdots, x_{10}\}, \cdots, G_{\frac{n}{5}} = \{x_{n-4}, \cdots, x_n\}$
- Now sort each group and let $z_i$ be the median of the froup $G_i$




- Let m be the **median** of $Z = \{z_1, z_2, \cdots, z_{\frac{n}{5}}\}$
- As a thought experiment, reorder the groups their median values:



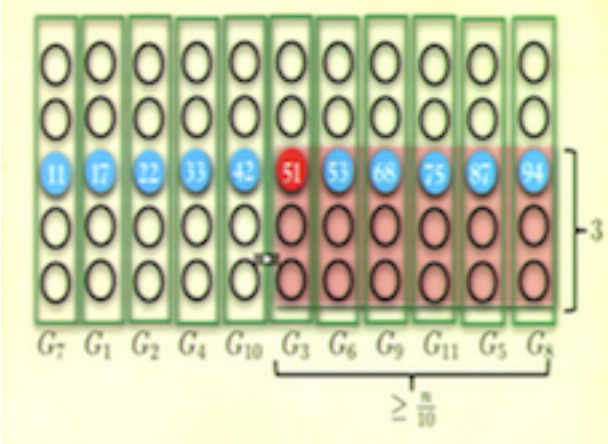

- The **median of the medians** is greater than at least $\frac{3}{10} \cdot (n-1)$ numbers in $S$

  $\Rightarrow |S_R| = |\{x_i \in S \setminus m : x_i \geq m\}| \leq \frac{7}{10} \cdot n$

- There are at least $\frac{3}{10} \cdot (n-1)$ numbers in $S$ that are at least as big as m.

  $\Rightarrow |S_L| = |\{x_i \in S : x_i < m\}| \leq \frac{7}{10} \cdot n$




- Thus, using m as a **pivot** we have:

  $\Rightarrow max\{|S_L|, |S_R|\} \leq \frac{7}{10} \cdot n$

- Thus the median of the medians is a **good** pivot.
- But how actually do we find the median of the medians?
  - Using the same **deterministic recursive algorithm**!

Deterministic Selection

**DetSelect**($S$, k)
**If** $|S| = 1$ **then** output $x_1$
**Else**
  Partition $S$ into $[\frac{n}{5}]$ groups of 5.
  **For** $j = \{1, 2, \cdots, \frac{n}{5}\}$
    Let $z_j$ be the median of group $G_j$
  **Let** $Z = \{z_1, z_2, \cdots, z_{[\frac{n}{5}]}\}$
  Set m $\leftarrow$ **DetSelect**($Z$, $[\frac{n}{10}]$)
  Set $S_L = \{x_i \in S : x_i <$ m$\}$
  Set $S_R = \{x_i \in S \setminus \{\text{m}\} : x_i \geq$ m$\}$
  **If** $|S_L|$ = k - 1 **then** output m

**If** $|S_L| >$ k - 1 **then** output **DetSelect**($S_L$, k)
**If** $|S_L| <$ k - 1 **then** output **DetSelect**($S_R$, k - 1 - $|S_L|$)

- Thus, using m as a **pivot** we have:

  $\Rightarrow max\{|S_L|, |S_R|\} \leq \frac{7}{10} \cdot n$

- The recursive formula for the running time is then:

  $\Rightarrow T(n) \leq T(\frac{7n}{10}) + T(\frac{n}{5}) + O(n)$

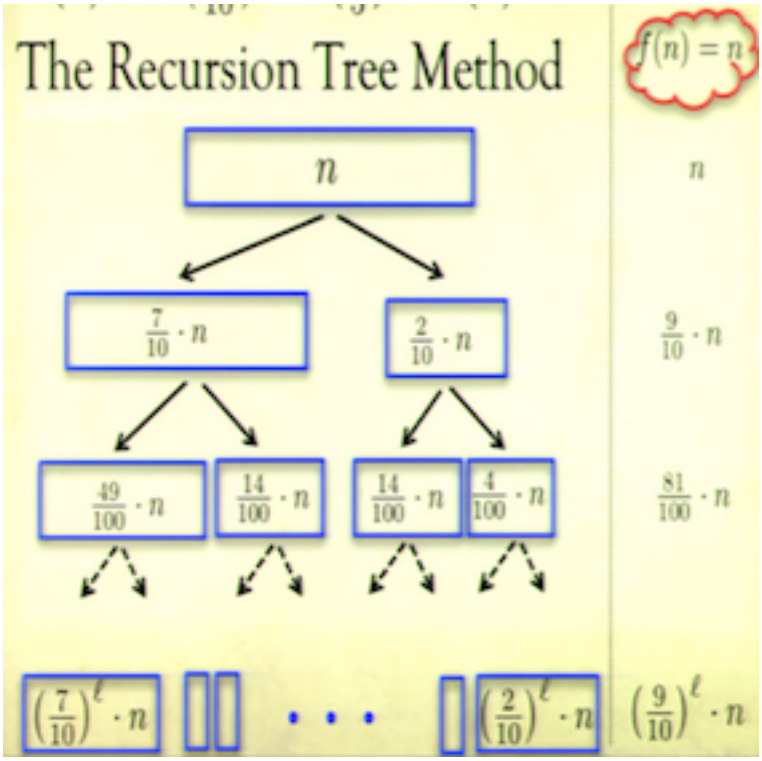
  Note:

  1. $T(\frac{n}{5})$ term comes from finding the median of the medians.
  2. $T(\frac{7n}{10})$ comes from that pivoting on the median of the medians gives a significantly smaller sub-problem.
  3. $O(n)$ comes from breaking in groups of size 5, finding the median of each group, and pivoting on the median of medians.

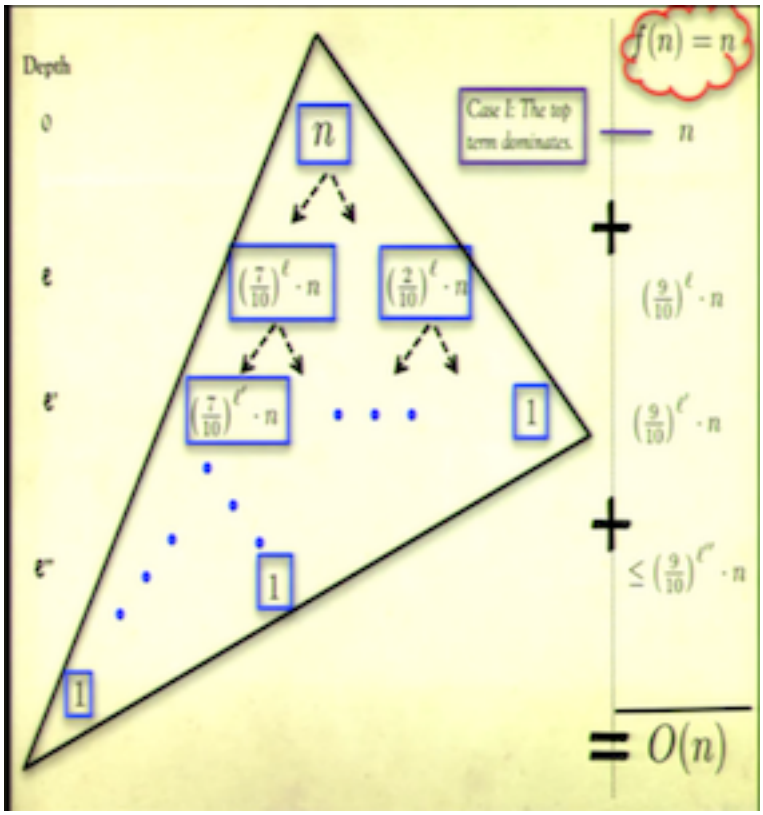
- But this does **not** fit with the Master Theorem!
  - The problem is not broken into the same sized sub-problems. One is $\frac{7n}{10}$, the other is $\frac{n}{5}$.

- This does not matter as we understand the proof of the Master Theorem,
  $\Rightarrow$ Apply the Recursion Tree Method!


The Recursion Tree Method
$f(n) = n$
$n$
$n$
$\frac{7}{10} \cdot n$
$\frac{2}{10} \cdot n$
$\frac{9}{10} \cdot n$
$\frac{49}{100} \cdot n$
$\frac{14}{100} \cdot n$
$\frac{14}{100} \cdot n$
$\frac{4}{100} \cdot n$
$\frac{81}{100} \cdot n$
$\left(\frac{7}{10}\right)^{\ell} \cdot n$
$\left(\frac{2}{10}\right)^{\ell} \cdot n$
$\left(\frac{9}{10}\right)^{\ell} \cdot n$




- Runtime: $T(n) = O(n)$
- Thus, we have a deterministic **linear time** algorithm to solve the selection problem (and, specifically, to find the median).
    - The reason why finding the selection problem is useful is that we try to find the median, but when we break it into two sub-problems, we are not finding the medians in the sub-problems since things would be shifted, and we might be finding the n over 10th problem.
    - This works a lot in induction. When you do induction proof, here we are using a stronger algorithm as our sub-routines, using kth selection problem, rather than median problem, which assumes that you have a stronger induction hypothesis. Often we use induction, it is hard to prove an easy result, since we can prove a general result, using induction.