Basics of Algorithm Analysis

- We measure running time as a function of n, the size of the input (in bytes assuming a reasonable encoding).
- We work in the RAM model of computation. All "reasonable" operations take "1" unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)

What is the running time of an algorithm

- Best case (seldom used)
- Average case (used if we understand the average)
- Worst case (used most often)

each input I of size n, has a running time $f(I)$

n

(5, 10, 7, 7, 8, 9, 6, 4, 1, 0)

Basics of Algorithm Analysis

- **We measure running time as a function of n, the size of the input (in bytes assuming a reasonable encoding).**
- **We work in the RAM model of computation. All "reasonable" operations take "1" unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)**

What is the running time of an algorithm

- **Best case (seldom used)**
- **Average case (used if we understand the average)**
- **Worst case (used most often)**

Basics of Algorithm Analysis

- **We measure running time as a function of n, the size of the input (in bytes assuming a reasonable encoding).**
- **We work in the RAM model of computation. All "reasonable" operations take "1" unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)**

What is the running time of an algorithm

- **Best case (seldom used)**
- **Average case (used if we understand the average)**
- **Worst case (used most often)**

Basics of Algorithm Analysis

- **We measure running time as a function of n, the size of the input (in bytes assuming a reasonable encoding).**
- **We work in the RAM model of computation. All "reasonable" operations take "1" unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)**

What is the running time of an algorithm

- **Best case (seldom used)**
- **Average case (used if we understand the average)**
- **Worst case (used most often)**

Example

```
input: A[n]
for i = 1 to n
    if (A[i] == 7)
        for j = 1 to n
            for k = 1 to n
                Print "hello"
```

$((3, 7, 2, 1, 7, 8), (7, 7, 7, 7, 7, 7))$

- **What is the worst case running time?** n^3
- **What is the best case running time?** n
- **What is the average case running time?** ?

$$\frac{2n^3 + n}{2n^3 + n + 1}$$

Example

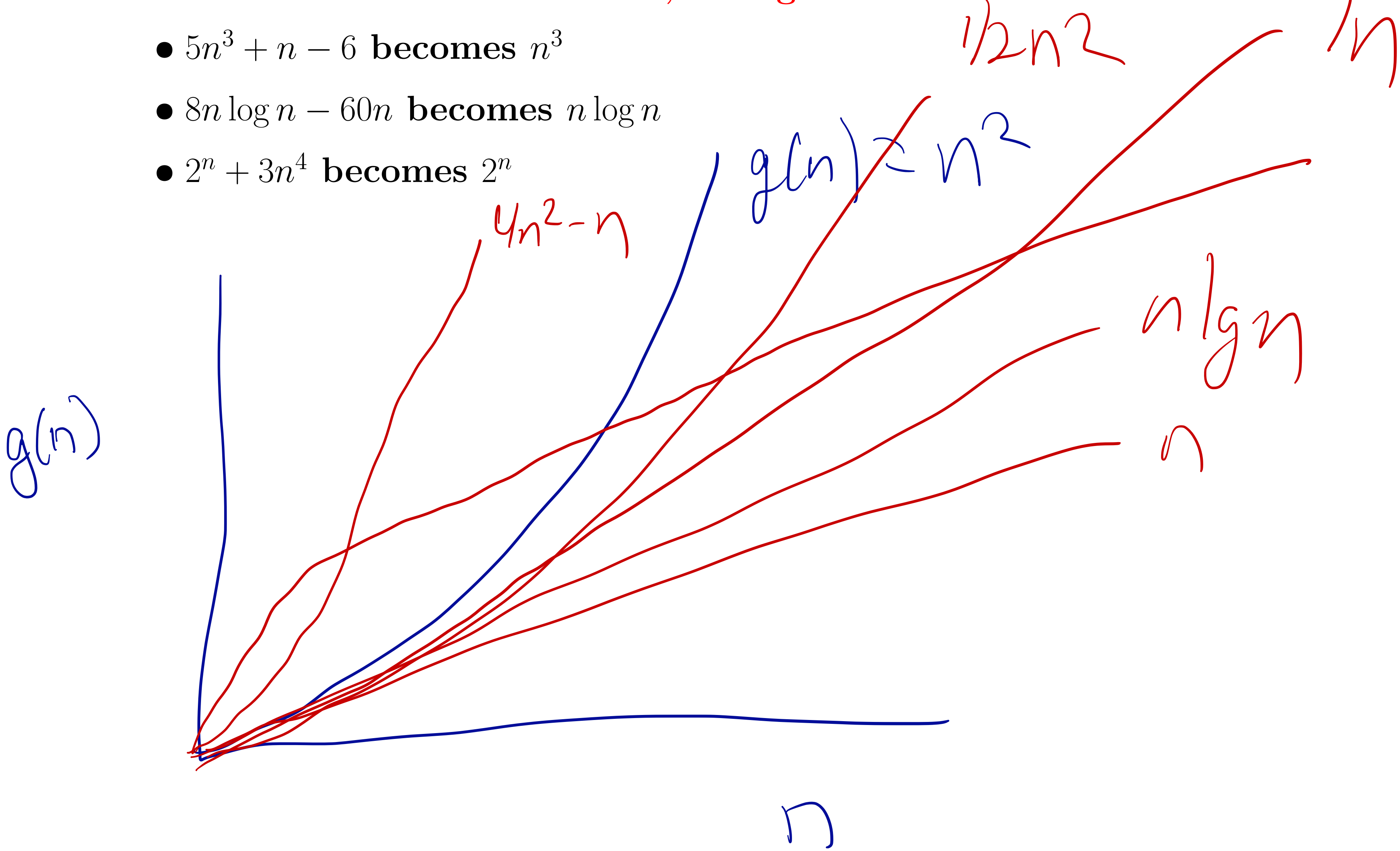
```
input: A[n]
for i = 1 to n
    if (A[i] == 7)
        for j = 1 to n
            for k = 1 to n
                Print "hello"
```

- **What is the worst case running time?** $O(n^3)$
- **What is the best case running time?** $O(n)$
- **What is the average case running time?** **What is an average array?**

How do we measure the running time?

We measure as a function of n, and ignore low order terms.

- $5n^3 + n - 6$ **becomes** n^3
- $8n \log n - 60n$ **becomes** $n \log n$
- $2^n + 3n^4$ **becomes** 2^n



Asymptotic notation

big-O

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}\ .$$

Alternatively, we say

$(f(n) \in O(g(n)))$

$$f(n) = O(g(n)) \text{ if there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Informally, $f(n) = O(g(n))$ **means that** $f(n)$ **is asymptotically less than or equal to** $g(n)$**.**

big-Ω

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\ .$$

Alternatively, we say

$$f(n) = \Omega(g(n)) \text{ if there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\ .$$

Informally, $f(n) = \Omega(g(n)$ **means that** $f(n)$ **is asymptotically greater than or equal to** $g(n)$**.**

big-Θ

$f(n) = \Theta(g(n))$ **if and only if** $f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$**.**

Informally, $f(n) = \Theta(g(n)$ **means that** $f(n)$ **is asymptotically equal to** $g(n)$**.**

INFORMAL summary

- $f(n) = O(g(n))$ **roughly means** $f(n) \leq g(n)$
- $f(n) = \Omega(g(n))$ **roughly means** $f(n) \geq g(n)$
- $f(n) = \Theta(g(n))$ **roughly means** $f(n) = g(n)$
- $f(n) = o(g(n))$ **roughly means** $f(n) < g(n)$
- $f(n) = w(g(n))$ **roughly means** $f(n) > g(n)$

Big-O proofs

- $3n = O(n^2)$
- $2n + 7 = O(n)$
- $n^{\log n} = O(2^n)$

$3n \leq c n^2$ $\quad \forall n \geq n_0$

$c = 3$ $\quad n_0 = 1$

$3n \leq 3n^2$ $\quad \forall n \geq 1$

$\iff 1 \leq n$ $\quad \forall n \geq 1$ ✓

$2n+7 \leq cn$ $\quad \forall n \geq n_0$

$\iff c \geq \underbrace{2 + \frac{7}{n}}_{\leq 3}$ $\quad \forall n \geq n_0$

$\forall n \geq 7$

$c = 3$

$2n+7 \leq 3n$ $\quad \forall n \geq 7$

$7 \leq n$ $\quad \forall n \geq 7$ ✓

Big-O proofs

- $3n = O(n^2)$
- $2n + 7 = O(n)$
- $n^{\log n} = O(2^n)$

$n^{\lg n} \le c2^n \quad \forall n \ge n_0$

$\iff \lg(n^{\lg n}) \le \lg(c2^n)$

$\iff \lg^2 n \le \lg c + \lg 2^n$

$\iff \lg^2 n \le \lg c + n \quad \forall n \ge n_0$

$n - \lg^2 n \ge -\lg c \quad \forall n \ge n_0$

$c = 1$

~~As long~~

$n - \lg^2 n \ge 0 \quad \forall n \ge n_0$

$\forall n \ge 4$

Use of big-O

$$2n + 7 = O(n)$$

$$2n + 7 = O(n^3)$$

$$2n + 7 = O(n^{4.5} \log n)$$

$$2n + 7 = O(2^n)$$

Which of these do we care about?

Use of big-O

$$2n + 7 = O(n)$$

$$2n + 7 = O(n^3)$$

$$2n + 7 = O(n^{4.5} \log n)$$

$$2n + 7 = O(2^n)$$

$2n+7 = O(3n)$

$2n+7 = O(n \cdot \lg n)$

!

Which of these do we care about?

- **Given a function** $f(n)$ **, we want to know the "smallest"** $g(n)$ **such that** $f(n) = O(g(n))$ **and** $g(n)$ **is "simple"**

Simple Functions

- **Given a function $f(n)$, we want to know the "smallest" $g(n)$ such that $f(n) = O(g(n))$ and $g(n)$ is "simple"**
- **Typical simple functions include (but are not limited to)**
  - 1
  - $\log \log n$
  - $\log n$
  - $\log^2 n$
  - n
  - $n \log n$
  - n^2
  - n^3
  - 2^n
  - $n!$
- **We use these to classify algorithms into classes**

See chart for justification

Polynomial Time

An algorithm runs in **polynomial time** if, on an input of size n, its running time is $O(n^k)$ for some constant k.

2^n is NOT polynomial. Let's try to prove that it is polynomial and see what goes wrong.

$$2^n \le c n^k \quad \forall n \ge n_0 ?$$

$$n \le \lg c + k \lg n$$

cannot choose c & k to make this true

der.

$$1 \qquad \frac{1}{n}$$

Proving Omega and Theta

$f(n) = \Omega(g(n))$ **if there exist positive constants** c **and** n_0 **such that** $0 \le cg(n) \le f(n)$ **for all** $n \ge n_0\}$.

$f(n) = \Theta(g(n))$ **if and only if** $f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$**.**

$$2n+7 = \Theta(n)$$

1. $2n+7 \le 3n \quad \forall n \ge 7$

2. $2n+7 \ge 1 \cdot n \quad \forall n \ge 1$

3 useful formulas

$$\Theta\left(n^2\right)$$

(for i = 1 to n
for j = 1 to i

Arithmetic series

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad \textbf{for } 0 < a < 1$$

Harmonic series

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1) = \Theta(\ln n)$$

Arithmetic Series in PseudoCode

```
for i = 1 to n
    for j = 1 to n
        Jump up and down
```

compared to

```
for i = 1 to n
    for j = 1 to i
        Jump up and down
```

Geometric Series

```
for i = 1 to log n
    for j = 1 to 2^i
        Jump up and down
```

$$1 + 2 + 4 + 8 + 16 + \cdots n$$

or

```
JUMP(n)
if n = 1
    Jump up and down once
else
    Jump up and down n times
    JUMP(⌊n/2⌋)
```

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = O(n)$$

$$n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n}\right)$$

$$\leq n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)$$

$$\leq 2n$$

A few facts about logs

- $\log_b a = \frac{\log_c a}{\log_c b}$ **for any** $c > 1$
- **therefore** $\ln n = O(\log n)$
- **in general, the base of the logarithm in a big-O statement is not important**

$\lg_2 10$

$$\begin{aligned} n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \frac{n}{5} + \ldots + 1 &= n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n}\right) \\ &= O(n \log n) \end{aligned}$$

Algorithmic Correctness

- **Very important, but we won't typically prove correctness from first principles.**
- **We will use loop invariants**
- **We will use other problem specific methods**

for i = 1 to n

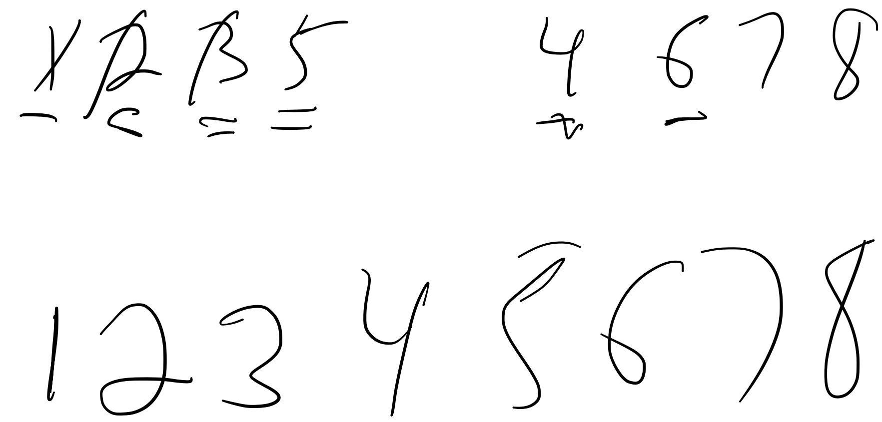
x = x + i

Divide and Conquer

- **Divide a problem into pieces**
- **Recursively solve the pieces**
- **Combine the solutions to the subproblems**

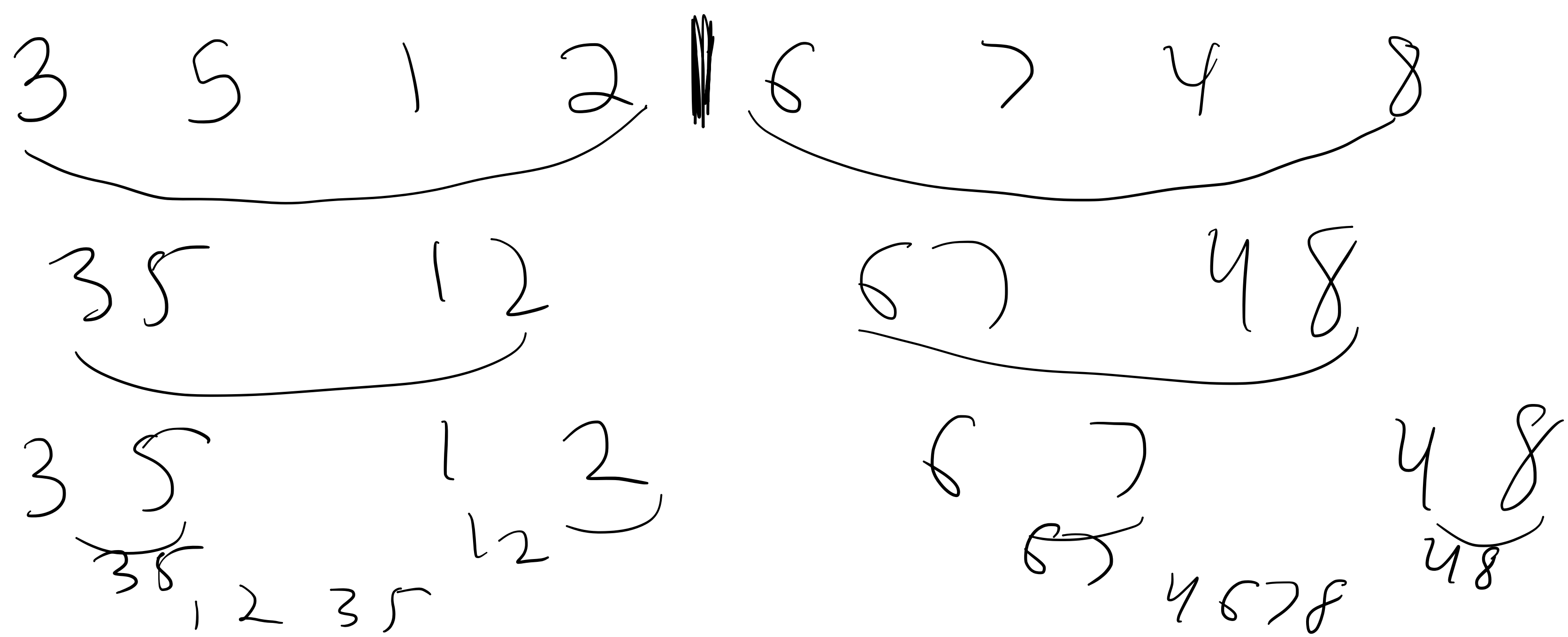
MergeSort

$Merge-Sort(A, p, r)$
if $p < r$
    $q = \lfloor (p+r)/2 \rfloor$ 1
    T(n/2) MERGE-SORT(A, p, q)
    T(n/2) MERGE-SORT$(A, q+1, r)$
    O(n) MERGE(A, p, q, r)



Let $T(n)$ be the running time of MergeSort on n items. Merge takes $O(n)$ time.

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



3 Recurrence Trees

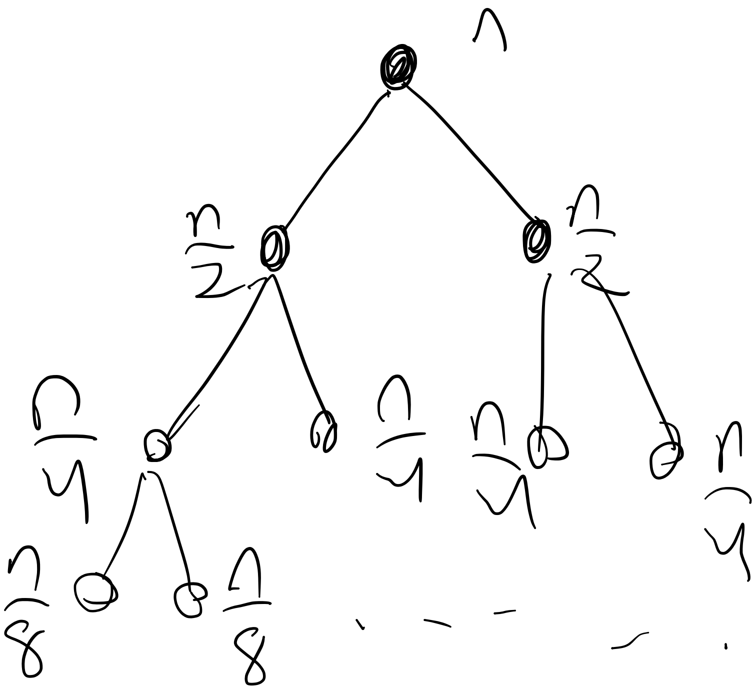
of subproblems

size of subproblem

1. $T(n) = 2T(n/2) + n$ non recursive time
2. $T(n) = 2T(n/2) + 1$
3. $T(n) = 2T(n/2) + n^2$

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

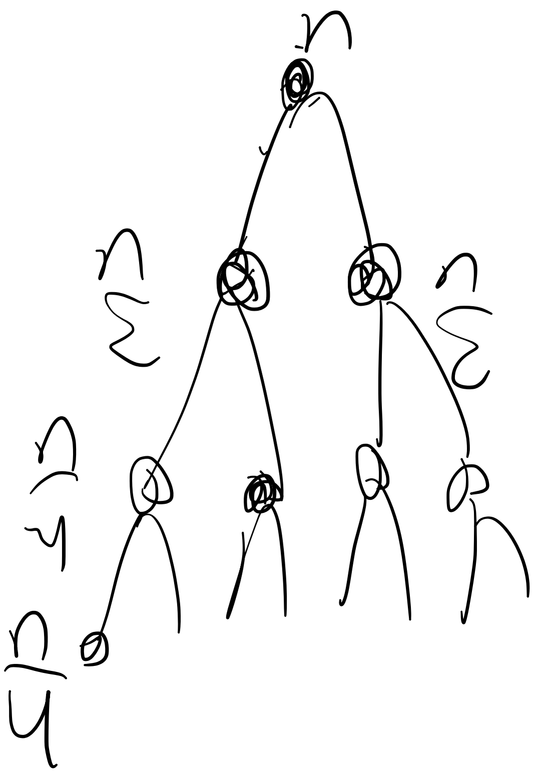
i	#subproblems	size	work
0	1	n	n
1	2	$\frac{n}{2}$	$\frac{n}{2}+\frac{n}{2}=n$
2	4	$\frac{n}{4}$	$\frac{n}{4}(4)=n$
3	8	$\frac{n}{8}$	
	2^i	$\frac{n}{2^i}$	$2^i\left(\frac{n}{2^i}\right)=n$
	$n = h$	2^i	
	n	1	



$$n(n) = n$$

$$n(\lg n + 1) = O(n \lg n)$$

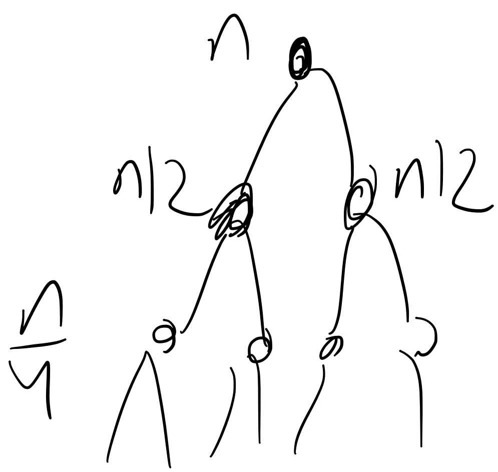
$$T(n) = 2T(n/2) + 1$$



#prob.	size	work
1	n	1
2	$\frac{n}{2}$	$2(1)=2$
4	$\frac{n}{4}$	$4(1)=4$
8	$\frac{n}{8}$	
2^i	$\frac{n}{2^i}$	$2^i(1)=2^i$
n	1	$n(1)=n$

$$(1+2+4+8+\dots+n) = n\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\dots+\frac{1}{n}\right) \le 2n \quad \boxed{O(n)}$$

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



#prob	size	work
1	n	n^2
2	$\frac{n}{2}$	$2\left(\frac{n}{2}\right)^2 = \frac{n^2}{2}$
4	$\frac{n}{4}$	$4\left(\frac{n}{4}\right)^2$ $\frac{n^2}{4}$
8	$\frac{n}{8}$	$8\left(\frac{n}{8}\right)^2 = n^2/8$
2^i	i	$2^i\left(\frac{n}{2^i}\right)^2 = n^2/2^i$
n	1	$n(1)^2 = n$

$$n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots + n$$

$$= n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n}\right) < 2n^2 = O(n^2)$$

$T(n) = 2T(\frac{n}{2}) + 1$

$n^{\lg_b a}$ 1 $O(n)$

Master Theorem

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

$n^{\lg_2 2}$ n $O(n \lg n)$

Master Theorem for Recurrences Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

$n^{\lg_b a}$ $f(n)$

$<$

$>$

$=$

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

$n^{\lg_2 2}$ n^2

$O(n^2)$

$f(n)$

$n^{\lg_b a}$

$n^{\lg_b a} \lg n$

Master Theorem

Master Theorem for Recurrences **Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence**

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. **If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.**
2. **If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.**
3. **If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.**

$T(n) = 7T(\frac{n}{5}) + n$ $\quad n^{\lg_5 7} \quad n$

$\Theta(n^{\lg_5 7})$

$T(n) = 9T(\frac{n}{3}) + n^2$

$n^{\lg_3 9} \quad n^2$ $\quad O(n^2 \lg n)$

Master Theorem

Master Theorem for Recurrences **Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence**

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. **If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.**
2. **If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.**
3. **If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.**

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

$$n^{\lg_b a} = \underline{\underline{n}} \qquad f(n) = \underline{\underline{n \lg n}}$$

$$n \lneq n\lg n \lneq n^{1+\varepsilon}$$

Master Theorem

Master Theorem for Recurrences **Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence**

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. **If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.**
2. **If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.**
3. **If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.**