COL 351 : Analysis & Design of Algorithms

Lecture 4

Divide & Conquer III :

Asymptotic Notation (contd.) and Counting Inversions

July 30, 2024 | Rohit Vaish

ANNOUNCEMENTS / REMINDERS

Sign up on Gradescope and Teams (two channels)

In-class quiz on Tuesday (Aug 6th)

Attendance : based on tutorial and in-class quizzes

Tutorial quiz will start at 1:10 PM (duration : 10 mins)

INTEGER MULTIPLICATION

Grade-school multiplication $\leq 9n^2$ basic operations

Recursive algorithm (4 calls) ?

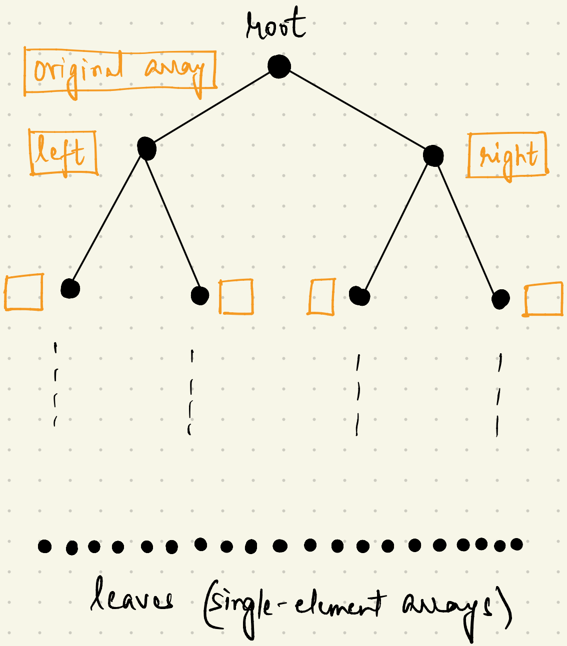
Karatsuba algorithm (3 calls) ?

MERGE SORT

Theorem: For every input array of length $n \geq 1$, Merge sort performs at most $6n\log_2 n + 6n$ operations.

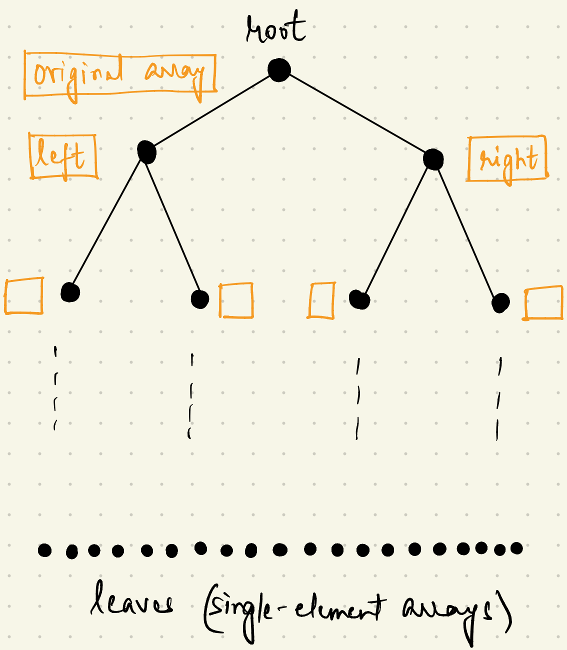
MERGE SORT

Theorem: For every input array of length $n \geq 1$, Merge sort performs at most $6n\log_2 n + 6n$ operations.



MERGE SORT

Theorem: For every input array of length $n \geq 1$, merge sort performs at most $6n\log_2 n + 6n$ operations.



Work done at level j

$$= 2^j \times 6\left(n/2^j\right) = 6n$$

independent of j

THREE GUIDING PRINCIPLES

Worst-case (or adversarial) analysis

Not too worried about precise constants

Asymptotic analysis

THREE GUIDING PRINCIPLES

Worst-case (or adversarial) analysis

- no assumption on where the input comes from

Not too worried about precise constants

- transcend environment dependence
- mathematically easier and no loss in predictive power

Asymptotic analysis

- only large inputs are "interesting"

Fast algorithm $\approx$ An algorithm whose worst-case running time grows polynomially with input size

VOCABULARY: BIG OH NOTATION

VOCABULARY: BIG OH NOTATION

E.g. $6n \log_2 n + 6n$

VOCABULARY: BIG OH NOTATION

E.g. $6n \log_2 n + 6n$

Suppress $\underbrace{\text{constant factors}}_{\text{system-dependent}}$ and $\underbrace{\text{lower-order terms}}_{\text{irrelevant for large inputs}}$

VOCABULARY: BIG OH NOTATION

E.g. $6n \log_2 n + 6n$

Suppress $\underbrace{\text{constant factors}}_{\text{system-dependent}}$ and $\underbrace{\text{lower-order terms}}_{\text{irrelevant for large inputs}}$

equate with $n \log n$

VOCABULARY: BIG OH NOTATION

E.g. $6n \log_2 n + 6n$

Suppress $\underbrace{\text{constant factors}}_{\text{system-dependent}}$ and $\underbrace{\text{lower-order terms}}_{\text{irrelevant for large inputs}}$

equate with $n \log n$

The running time is $O(n \log n)$ "big-oh of n log n"

"order n log n"

VOCABULARY: BIG OH NOTATION

Sweet spot for reasoning about algorithms

VOCABULARY: BIG OH NOTATION

Sweet spot for reasoning about algorithms

Coarse enough to avoid environment-specific details

Sharp enough to allow meaningful comparison among algorithms

Quick Examples

QUICK EXAMPLES

Searching for a number x in an array A of length n

```
for i = 1 to n
    if A[i] = x
        return TRUE
return FALSE
```

Running time : ?

QUICK EXAMPLES

Searching for a number x in an array A of length n

```
for i = 1 to n
    if A[i] = x
        return TRUE
return FALSE
```

Running time : $O(n)$

QUICK EXAMPLES

Searching for a number x in an array A of length n or
" " " B " " .

```
for i = 1 to n
        if A[i] = x
              return TRUE
for i = 1 to n
        if B[i] = x
              return TRUE
return FALSE
```

Running time : ?

QUICK EXAMPLES

Searching for a number x in an array A of length n or
" " " B " " .

for $i = 1$ to n
    if $A[i] = x$
        return TRUE
for $i = 1$ to n
    if $B[i] = x$
        return TRUE
return FALSE

Running time : $O(n)$

QUICK EXAMPLES

Checking for a common element in arrays A and B

```
for i = 1 to n
    for j = 1 to n
        if A[i] = B[j]
            return TRUE
return FALSE
```

Running time : ?

QUICK EXAMPLES

Checking for a common element in arrays A and B

```
for i = 1 to n
    for j = 1 to n
        if A[i] = B[j]
            return TRUE
return FALSE
```

Running time : $O(n^2)$

QUICK EXAMPLES

Checking for a duplicate entry in array A

```
for i = 1 to n
    for j = i+1 to n
        if A[i] = A[j]
            return TRUE
return FALSE
```

Running time : ?

QUICK EXAMPLES

Checking for a duplicate entry in array A

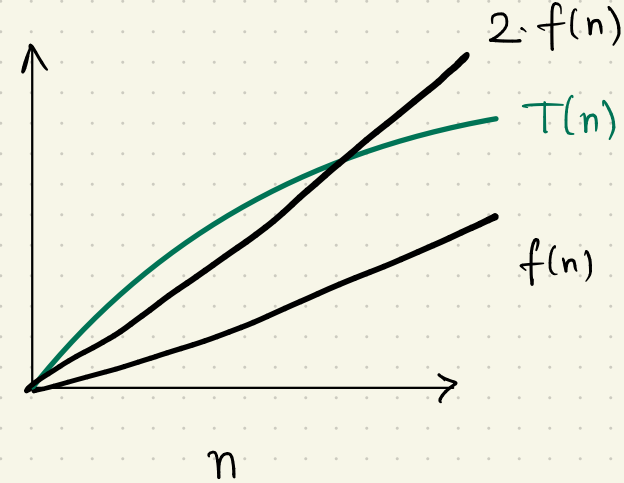
```
for i = 1 to n
    for j = i+1 to n
        if A[i] = A[j]
            return TRUE
return FALSE
```

Running time : $O(n^2)$

DEFINING BIG OH

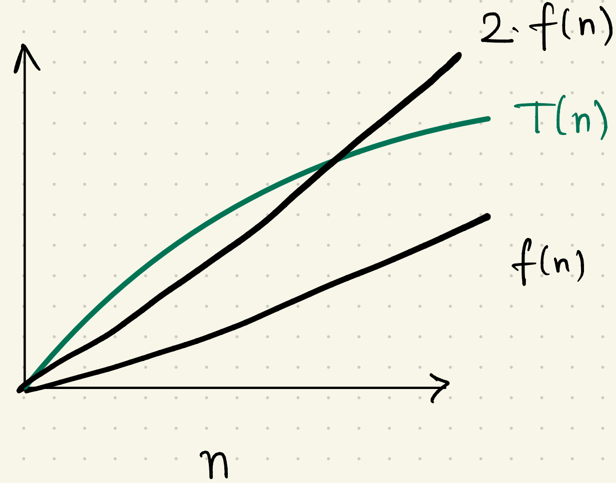
DEFINING BIG OH

$T(n)$ is "eventually bounded above" by a constant multiple of $f(n)$.



DEFINING BIG OH

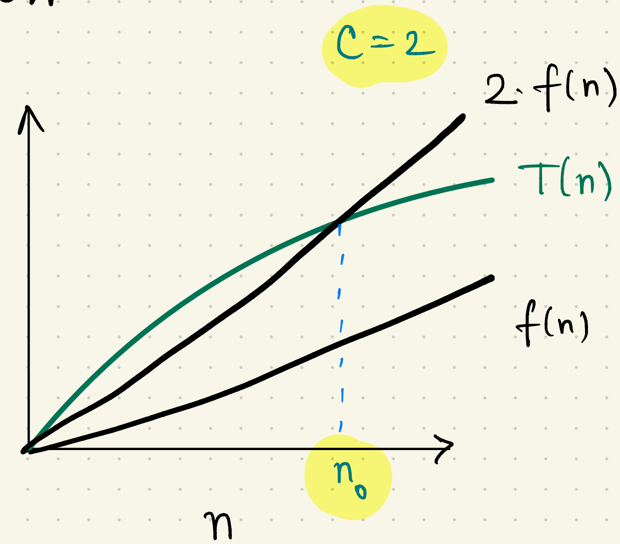
T(n) is "eventually bounded above" by a constant multiple of f(n).



$T(n) = O(f(n))$ if there exist positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

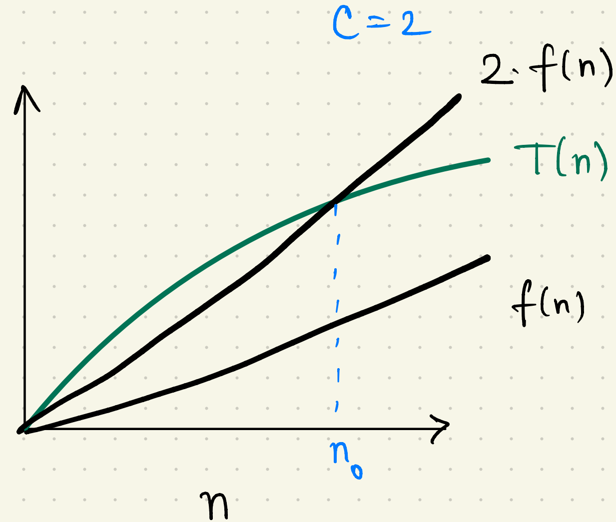
DEFINING BIG OH

$T(n)$ is "eventually bounded above" by a constant multiple of $f(n)$.



$T(n) = O(f(n))$ if there exist positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

DEFINING BIG OH



NOTE : $O(f(n))$ is a set of functions

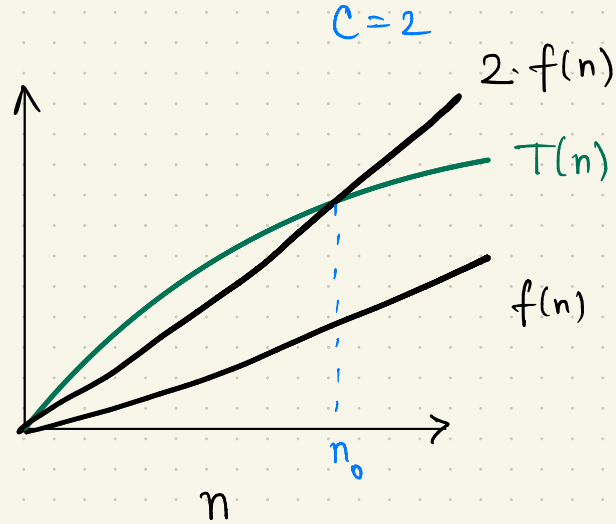
$T(n) = O(f(n))$ if there exist positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

DEFINING BIG OH

NOTE : $O(f(n))$ is a set of functions

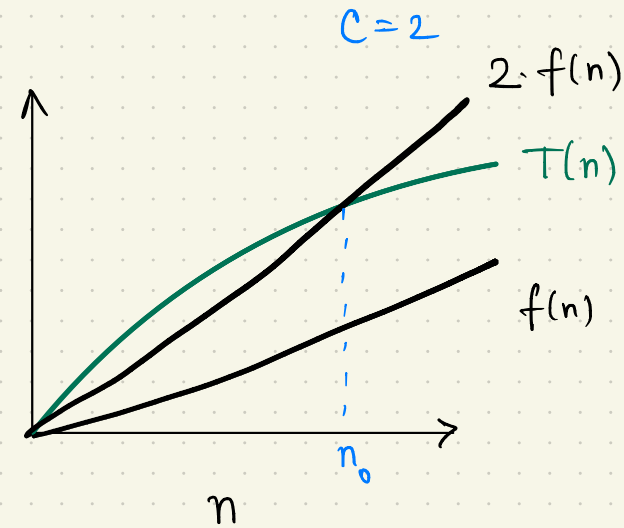
Correct : $T(n) \in O(f(n))$

Common : $T(n) = O(f(n))$



$T(n) = O(f(n))$ if there exist positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

DEFINING BIG OH



Game!

First, you pick c and n_0

Then, your opponent picks n.

$T(n) = O(f(n))$ if there exist positive constants c and n_0 such that $T(n) \le c \cdot f(n)$ for all $n \geq n_0$.

EXAMPLES

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,

then $T(n) = O(n^k)$.

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,

then $T(n) = O(n^k)$.

Proof:

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,
then $T(n) = O(n^k)$.

Proof: Choose $n_0 = 1$ and $c = a_k + a_{k-1} + \cdots + a_0$

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,
then $T(n) = O(n^k)$.

Proof: Choose $n_0 = 1$ and $c = \underbrace{a_k + a_{k-1} + \cdots + a_0}_{\text{might be negative}}$

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,

then $T(n) = O(n^k)$.

Proof: Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + \cdots + |a_0|$

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,
then $T(n) = O(n^k)$.

Proof: Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + \cdots + |a_0|$

Fix an arbitrary $n \geq n_0$.

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,

then $T(n) = O(n^k)$.

Proof: Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + \cdots + |a_0|$

Fix an arbitrary $n \geq n_0$.

$$T(n) \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0|$$

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,

then $T(n) = O(n^k)$.

Proof: Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + \cdots + |a_0|$

Fix an arbitrary $n \geq n_0$.

$$T(n) \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0|$$

$$\leq |a_k| n^k + |a_{k-1}| n^k + \cdots + |a_1| n^k + |a_0| n^k$$

EXAMPLES

Claim: If $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$,
then $T(n) = O(n^k)$.

Proof: Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + \cdots + |a_0|$

Fix an arbitrary $n \geq n_0$.

$$
\begin{aligned}
T(n) &\leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0| \\
&\leq |a_k| n^k + |a_{k-1}| n^k + \cdots + |a_1| n^k + |a_0| n^k \\
&= c \cdot n^k.
\end{aligned}
$$

$\blacksquare$

EXAMPLES

Claim: If $T(n) = n^k$ then $T(n)$ is **not** $O(n^{k-1})$.

EXAMPLES

Claim: If $T(n) = n^k$ then $T(n)$ is **not** $O(n^{k-1})$.

Proof: (by contradiction)

Suppose $T(n) = O(n^{k-1})$.

EXAMPLES

Claim: If $T(n) = n^k$ then $T(n)$ is **not** $O(n^{k-1})$.

Proof: (by contradiction)

Suppose $T(n) = O(n^{k-1})$. Then, for some positive constants c and n_0 and for all $n \geq n_0$,

$$T(n) \leq c \cdot n^{k-1}.$$

EXAMPLES

Claim: If $T(n) = n^k$ then $T(n)$ is not $O(n^{k-1})$.

Proof: (by contradiction)

Suppose $T(n) = O(n^{k-1})$. Then, for some positive constants c and n_0 and for all $n \geq n_0$,

$$T(n) \leq c \cdot n^{k-1}.$$

$$\Rightarrow \quad n \leq c.$$

EXAMPLES

Claim: If $T(n) = n^k$ then $T(n)$ is not $O(n^{k-1})$.

Proof: (by contradiction)

Suppose $T(n) = O(n^{k-1})$. Then, for some positive constants c and n_0 and for all $n \geqslant n_0$,

$$T(n) \leqslant c \cdot n^{k-1}.$$

$$\Longrightarrow \quad n \leqslant c.$$

Contradiction!

$\blacksquare$

BIG OMEGA & BIG THETA

BIG OMEGA & BIG THETA

$T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that

$$T(n) \geqslant c \cdot f(n)$$

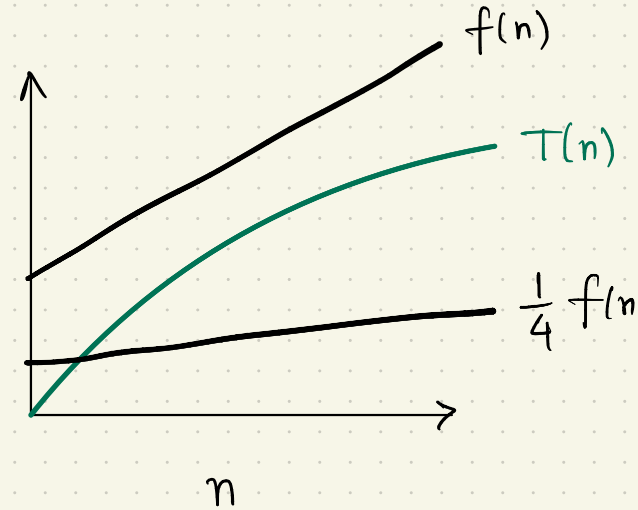
for all $n \geqslant n_0$.

BIG OMEGA & BIG THETA

$T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$.

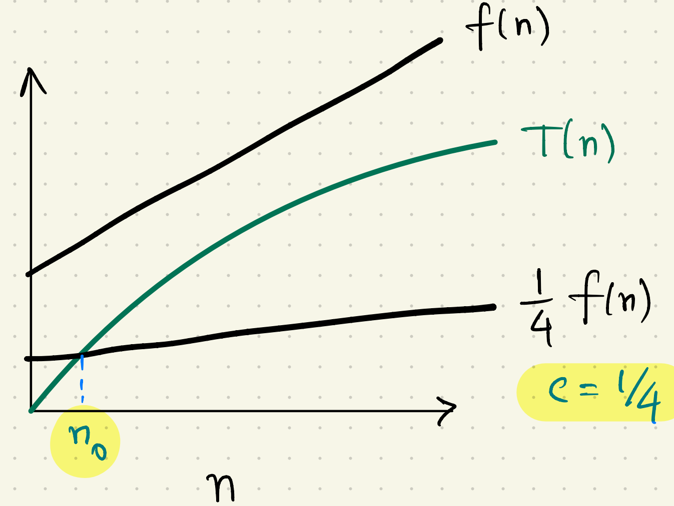


BIG OMEGA & BIG THETA

$T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$.

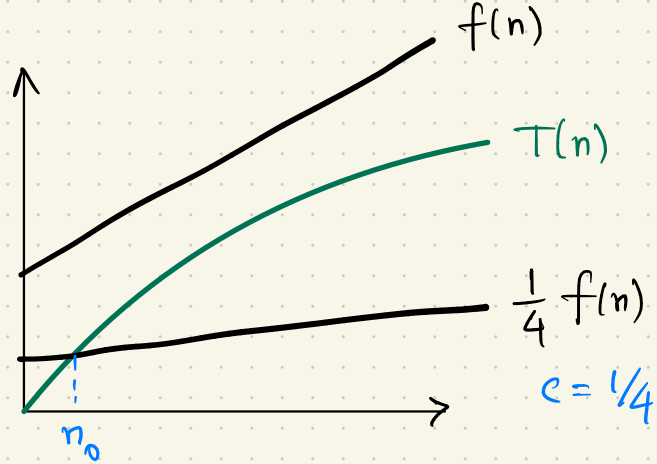


BIG OMEGA & BIG THETA

$T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$.



$T(n) = \Theta(f(n))$ if there exist positive constants c_1, c_2, and n_0 such that

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

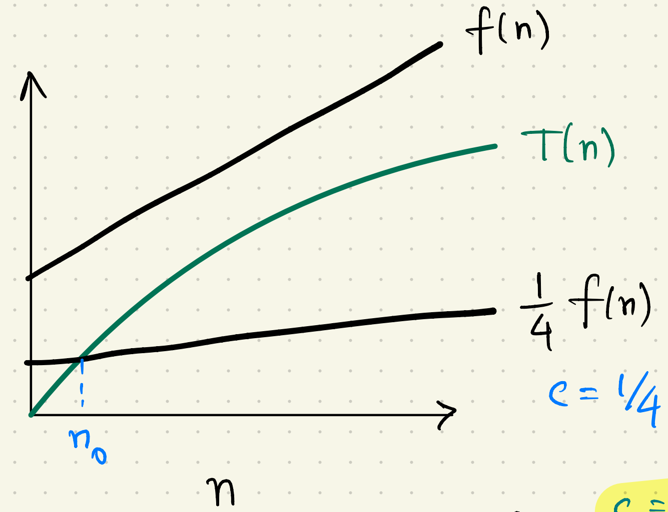
for all $n \geq n_0$.

BIG OMEGA & BIG THETA

$T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that

$$T(n) \geq c \cdot f(n)$$

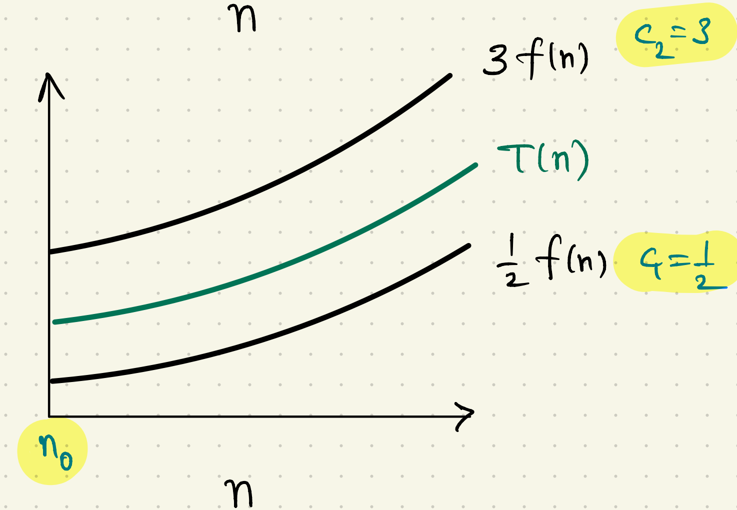
for all $n \geq n_0$.



$T(n) = \Theta(f(n))$ if there exist positive constants c_1, c_2, and n_0 such that

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

for all $n \geq n_0$.

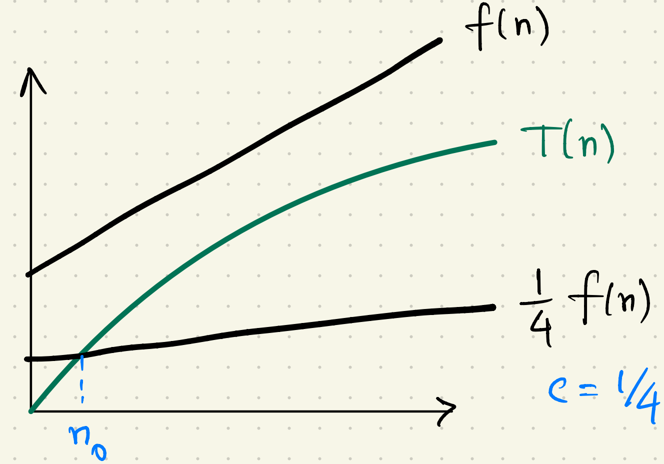


BIG OMEGA & BIG THETA

$T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$.

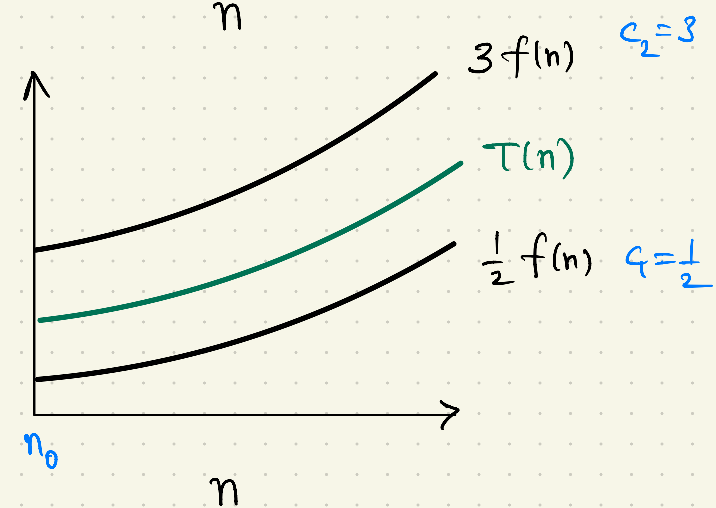


$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

$T(n) = \Theta(f(n))$ if there exist positive constants c_1, c_2, and n_0 such that

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

for all $n \geq n_0$.



EXAMPLES

$$\frac{n}{\log n} = O(n),\ \Omega(n),\ \Theta(n)\ ?$$

EXAMPLES

$$\frac{n}{\log n} = O(n), \quad \Omega(n), \quad \Theta(n) \; ?$$

EXAMPLES

$$\frac{n}{\log n} = O(n),\ \Omega(n),\ \Theta(n)\ ?$$

$$\frac{n}{100} = O(n),\ \Omega(n),\ \Theta(n)\ ?$$

EXAMPLES

$$\frac{n}{\log n} = O(n) \,,\ \Omega(n) \,,\ \Theta(n) \ ?$$

$$\frac{n}{100} = O(n) \,,\ \Omega(n) \,,\ \Theta(n) \ ?$$

EXAMPLES

$$\frac{n}{\log n} = O(n)\ ,\ \Omega(n)\ ,\ \Theta(n)\ ?$$

$$\frac{n}{100} = O(n)\ ,\ \Omega(n)\ ,\ \Theta(n)\ ?$$

$$n^2 = O(n)\ ,\ \Omega(n)\ ,\ \Theta(n)\ ?$$

EXAMPLES

$$\frac{n}{\log n} = O(n), \ \Omega(n), \ \Theta(n) \ ?$$

$$\frac{n}{100} = O(n), \ \Omega(n), \ \Theta(n) \ ?$$

$$n^2 = O(n), \ \Omega(n), \ \Theta(n) \ ?$$

EXAMPLES

$$\frac{n}{\log n} = O(n), \ \Omega(n), \ \Theta(n) \ ?$$

$$\frac{n}{100} = O(n), \ \Omega(n), \ \Theta(n) \ ?$$

$$n^2 = O(n), \ \Omega(n), \ \Theta(n) \ ?$$

Others : $T(n) = o(f(n))$ "little oh" $T(n) = \omega(f(n))$ "little omega" self-reading

SIGACT News   Apr-June 1976

BIG OMICRON AND BIG OMEGA AND BIG THETA

Donald E. Knuth
Computer Science Department
Stanford University
Stanford, California 94305

Well, I think I have beat this issue to death, knowing of no other arguments pro or con the introduction of Ω and Θ. On the basis of the issues discussed here, I propose that members of SIGACT, and editors of computer science and mathematics journals, adopt the O, Ω, and Θ notations as defined above, unless a better alternative can be found reasonably soon. Furthermore I propose that the relational notations of Hardy be adopted in those situations where a relational notation is more appropriate.

MORE DIVIDE & CONQUER

COUNTING INVERSIONS

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

e.g.,

1	3	5	2	4	6

inversions =

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

e.g.,

1	3	5	2	4	6

$\#$ inversions =

$i=2$, $j=4$

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

e.g.,

1	3	5	2	4	6

$\#$ inversions =

$i=2,\ j=4$

$i=3,\ j=4$

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

e.g.,

1	3	5	2	4	6

\# inversions =

$i=2$, $j=4$

$i=3$, $j=4$

$i=3$, $j=5$

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

e.g.,

1	3	5	2	4	6

\# inversions $= 3$

$i=2, j=4$

$i=3, j=4$

$i=3, j=5$

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

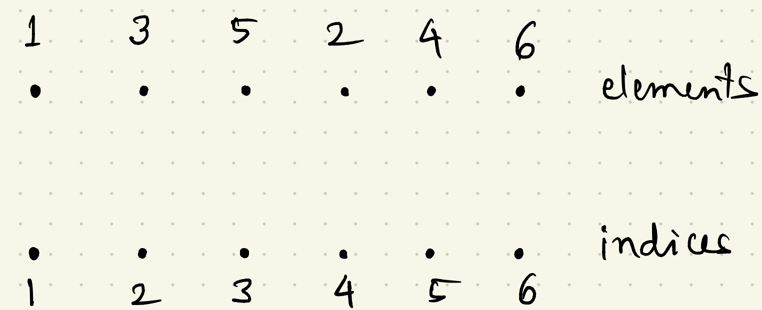
pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

e.g.,

1	3	5	2	4	6

inversions = 3



COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

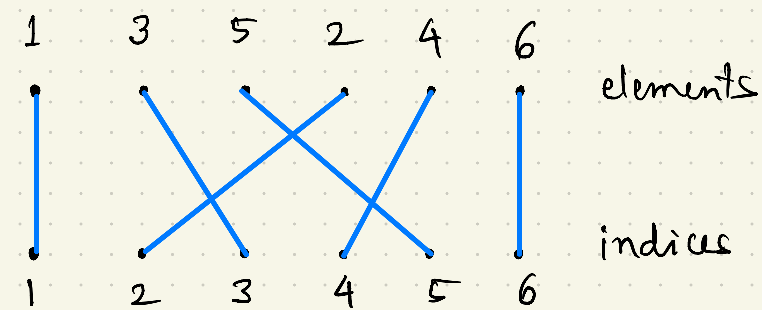
pairs (i, j) of array indices with

$i < j$ and $A[i] > A[j]$.

e.g.,

1	3	5	2	4	6

inversions = 3



COUNTING INVERSIONS

input: a length n array A of distinct integers

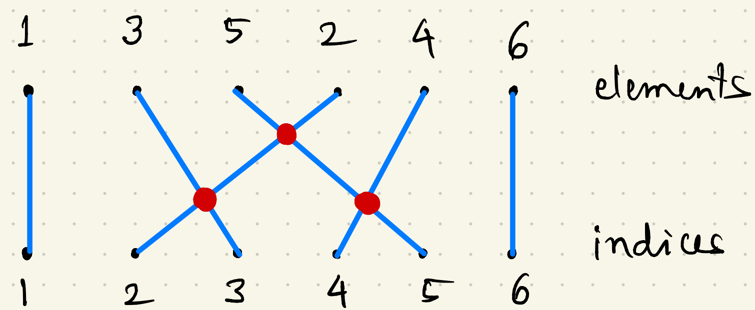
output: the number of inversions of A

pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$.

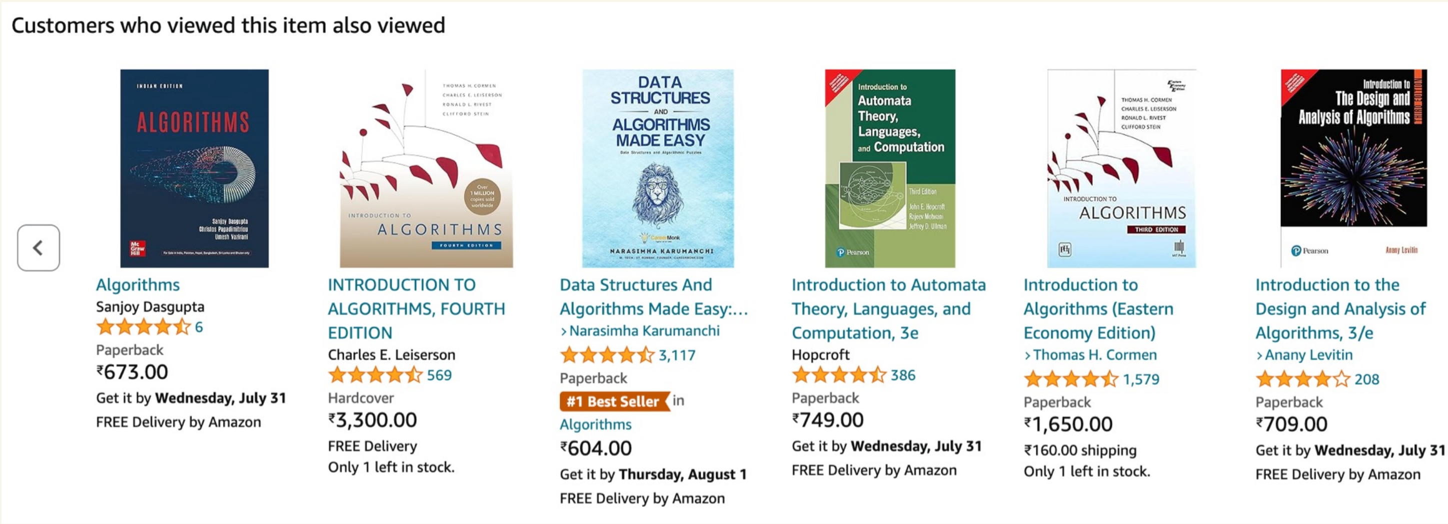
e.g.,

1	3	5	2	4	6

inversions = 3



COUNTING INVERSIONS



Collaborative filtering : Similar users get similar recommendations

COUNTING INVERSIONS

Brute force algorithm: check every pair of indices (i, j)

COUNTING INVERSIONS

Brute force algorithm: check every pair of indices (i, j)

$\Theta(n^2)$

COUNTING INVERSIONS

Brute force algorithm: check every pair of indices (i, j)

$\Theta(n^2)$

Can we do better?

COUNTING INVERSIONS

Brute force algorithm: check every pair of indices (i, j)

$\Theta(n^2)$

Can we do better?

Yes! $O(n \log n)$ algorithm via divide-and-conquer.

COUNTING INVERSIONS

Call an inversion (i, j) where $i < j$

left inversion if $i, j \leq n/2$

right inversion if $i, j > n/2$

split inversion if $i \leq \frac{n}{2} < j$

COUNTING INVERSIONS

Call an inversion (i, j) where $i < j$

left inversion if $i, j \leq n/2$

right inversion if $i, j > n/2$

split inversion if $i \leq \frac{n}{2} < j$

1	3	5	2	4	6

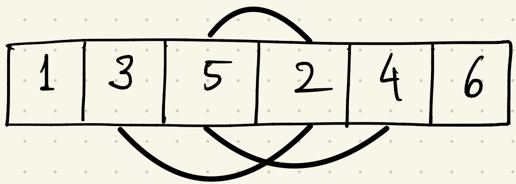
COUNTING INVERSIONS

Call an inversion (i, j) where $i < j$

left inversion if $i, j \leq n/2$

right inversion if $i, j > n/2$

split inversion if $i \leq \frac{n}{2} < j$



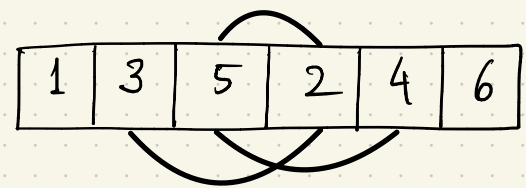
COUNTING INVERSIONS

Call an inversion (i, j) where $i < j$

left inversion if $i, j \leq n/2$

right inversion if $i, j > n/2$

split inversion if $i \leq \frac{n}{2} < j$



all split inversions

COUNTING INVERSIONS

Call an inversion (i, j) where $i < j$

left inversion if $i, j \leq n/2$

Compute these recursively

right inversion if $i, j > n/2$

split inversion if $i \leq \frac{n}{2} < j$ ← compute these in "combine" step

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else $\quad \ell :=$ recursively count inversions on left half of A

$\quad r :=$ " " " right " "

$\quad s :=$ Count split inversions of A

return $\ell + r + s$

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else $\ell :=$ recursively count inversions on left half of A

$r :=$ " " " right " "

$s :=$ Count split inversions of A

return $\ell + r + s$

$$A = \begin{array}{|c|c|c|c|c|c|c|c|} \hline \frac{n}{2}+1 & \frac{n}{2}+2 & \cdots & n & 1 & 2 & \cdots & \frac{n}{2} \\ \hline \end{array}$$

$\#$ split inversions $=$?

$A =$

$\frac{n}{2}+1$	$\frac{n}{2}+2$	$\cdots$	n	1	2	$\cdots$	$\frac{n}{2}$

$$\#\ \text{split inversions} = n^2/4$$

$$A = \begin{array}{|c|c|c|c|c|c|c|c|}\hline \frac{n}{2}+1 & \frac{n}{2}+2 & \cdots & n & 1 & 2 & \cdots & \frac{n}{2} \\ \hline \end{array}$$

$$\#\ \text{split inversions} = n^2/4$$

Possible to compute split inversions in $O(n)$ time ?

suffices for $O(n \log n)$ time overall

Piggyback on Merge Sort

Piggyback on Merge Sort

Suppose A has no split inversion.

Piggyback on Merge Sort

Suppose A has no split inversion.

Then, every element in left half of A $<$ every element in right half of A

Piggyback on Merge Sort

Suppose A has no split inversion.

Then, every element in left half of A < every element in right half of A

What does merge subroutine do for such array?

Piggyback on Merge Sort

Suppose A has no split inversion.

Then, every element in left half of A $<$ every element in right half of A

What does merge subroutine do for such array?

Concatenation!

$$\boxed{\text{left half}} \underset{\text{join}}{\longleftrightarrow} \boxed{\text{right half}}$$

What does merge subroutine do when there are split inversions?

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

.

Output

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

.

Output

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

.

Output

1					

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

.

Output

1	2				

When "2" gets copied in output, the split inversions (3, 2) and (5, 2) are exposed.

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

Output

1	2	3			

When "2" gets copied in output, the split inversions (3, 2) and (5, 2) are exposed.

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

Output

1	2	3	4		

When "2" gets copied in output, the split inversions (3, 2) and (5, 2) are exposed.

When "4" gets copied in output, the split inversion (5, 4) is exposed.

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

Output

1	2	3	4	5	

When "2" gets copied in output, the split inversions (3, 2) and (5, 2) are exposed.

When "4" gets copied in output, the split inversion (5, 4) is exposed.

What does merge subroutine do when there are split inversions?

Consider merging

1	3	5

and

2	4	6

.

Output

1	2	3	4	5	6

When "2" gets copied in output, the split inversions (3, 2) and (5, 2) are exposed.

When "4" gets copied in output, the split inversion (5, 4) is exposed.