CSE 431/531: Algorithm Analysis and Design (Fall 2024)
Introduction IV: Asymptotic Notation

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Announcements: Quiz 2

- Posted on Ublearns
- Should take $<$ 30 minutes, 2 attempts
- Due Thur 5th Sep @ 11:59PM

Outline

Recall: O, Ω, Θ-Notation: Asymptotic Bounds

O-**Notation** For a function $g(n)$,

$$O(g(n)) = \big\{\text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \le cg(n), \forall n \ge n_0\big\}.$$

Ω-**Notation** For a function $g(n)$,

$$\Omega(g(n)) = \big\{\text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \ge cg(n), \forall n \ge n_0\big\}.$$

Θ-**Notation** For a function $g(n)$,

$$\Theta(g(n)) = \big\{\text{function } f : \exists c_2 \ge c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \le f(n) \le c_2 g(n), \forall n \ge n_0\big\}.$$

Asymptotic Notations	O	Ω	Θ
Comparison Relations	$\leq$	$\geq$	$=$

Asymptotic Notations	O	Ω	Θ
Comparison Relations	$\leq$	$\geq$	$=$

Trivial Facts on Comparison Relations

- $a \leq b \; \Leftrightarrow \; b \geq a$
- $a = b \; \Leftrightarrow \; a \leq b$ and $a \geq b$
- $a \leq b$ or $a \geq b$

Asymptotic Notations	O	Ω	Θ
Comparison Relations	$\leq$	$\geq$	$=$

Trivial Facts on Comparison Relations

- $a \leq b \;\Leftrightarrow\; b \geq a$
- $a = b \;\Leftrightarrow\; a \leq b$ and $a \geq b$
- $a \leq b$ or $a \geq b$

Correct Analogies

- $f(n) = O(g(n)) \;\Leftrightarrow\; g(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n)) \;\Leftrightarrow\; f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Asymptotic Notations	O	Ω	Θ
Comparison Relations	$\leq$	$\geq$	$=$

Trivial Facts on Comparison Relations

- $a \leq b \;\Leftrightarrow\; b \geq a$
- $a = b \;\Leftrightarrow\; a \leq b$ and $a \geq b$
- $a \leq b$ or $a \geq b$

Correct Analogies

- $f(n) = O(g(n)) \;\Leftrightarrow\; g(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n)) \;\Leftrightarrow\; f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Incorrect Analogy

- $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$

Incorrect Analogy

- $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$

Incorrect Analogy

- $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$

$$f(n) = n^2$$

$$g(n) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ n^3 & \text{if } n \text{ is even} \end{cases}$$

Recall: Informal way to define O-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \to 3n^2$
- ignoring leading constant: $3n^2 \to n^2$

Recall: Informal way to define O-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \to 3n^2$
- ignoring leading constant: $3n^2 \to n^2$
- $3n^2 - 10n - 5 = O(n^2)$

Recall: Informal way to define O-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \to 3n^2$
- ignoring leading constant: $3n^2 \to n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$

Recall: Informal way to define O-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \to 3n^2$
- ignoring leading constant: $3n^2 \to n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$
- In the formal definition of $O(\cdot)$, nothing tells us to ignore lower order terms and leading constant.

Recall: Informal way to define O-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \to 3n^2$
- ignoring leading constant: $3n^2 \to n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$
- In the formal definition of $O(\cdot)$, nothing tells us to ignore lower order terms and leading constant.
- $3n^2 - 10n - 5 = O(5n^2 - 6n + 5)$ is correct, though weird

Recall: Informal way to define O-notation

- ignoring lower order terms: $3n^2 - 10n - 5 \to 3n^2$
- ignoring leading constant: $3n^2 \to n^2$
- $3n^2 - 10n - 5 = O(n^2)$
- Indeed, $3n^2 - 10n - 5 = \Omega(n^2), 3n^2 - 10n - 5 = \Theta(n^2)$
- In the formal definition of $O(\cdot)$, nothing tells us to ignore lower order terms and leading constant.
- $3n^2 - 10n - 5 = O(5n^2 - 6n + 5)$ is correct, though weird
- $3n^2 - 10n - 5 = O(n^2)$ is the most natural since n^2 is the simplest term we can have inside $O(\cdot)$.

Notice that O denotes asymptotic upper bound

- $n^2 + 2n = O(n^3)$ is correct.
- The following sentence is correct: the running time of the insertion sort algorithm is $O(n^4)$.
- We say: the running time of the insertion sort algorithm is $O(n^2)$ and the bound is tight.

Notice that O denotes asymptotic upper bound

- $n^2 + 2n = O(n^3)$ is correct.
- The following sentence is correct: the running time of the insertion sort algorithm is $O(n^4)$.
- We say: the running time of the insertion sort algorithm is $O(n^2)$ and the bound is tight.
- We do not use Ω and Θ very often when we upper bound running times.

More Exercise: Lecture notes and Quiz 2

Asymptotic Notations	O	Ω	Θ	o	ω
Comparison Relations	$\leq$	$\geq$	$=$	$<$	$>$

Asymptotic Notations	O	Ω	Θ	o	ω
Comparison Relations	$\leq$	$\geq$	$=$	$<$	$>$

Questions?

Outline

$O(n)$ (Linear) Running Time

Computing the sum of n numbers

sum(A, n)

```
S ← 0
for i ← 1 to n
    S ← S + A[i]
return S
```

$O(n)$ (Linear) Running Time

- Merge two sorted arrays

3	8	12	20	32	48

5	7	9	25	29

$O(n)$ (Linear) Running Time

- Merge two sorted arrays

3	8	12	20	32	48

5	7	9	25	29

$O(n)$ (Linear) Running Time

- Merge two sorted arrays

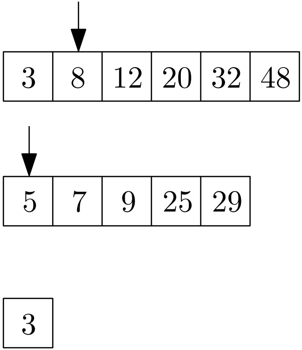
3	8	12	20	32	48

5	7	9	25	29

3

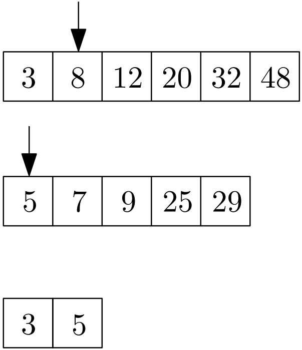
$O(n)$ (Linear) Running Time

- Merge two sorted arrays



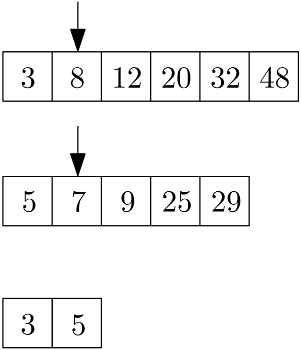
$O(n)$ (Linear) Running Time

- Merge two sorted arrays



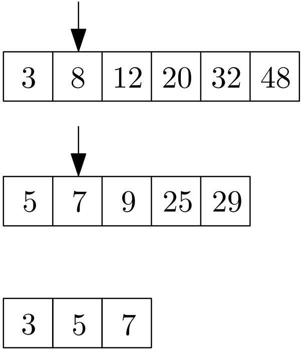
$O(n)$ (Linear) Running Time

- Merge two sorted arrays



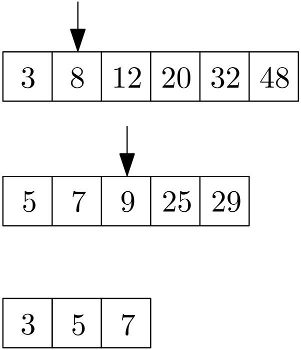
$O(n)$ (Linear) Running Time

- Merge two sorted arrays



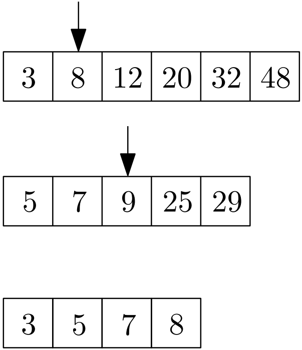
$O(n)$ (Linear) Running Time

- Merge two sorted arrays



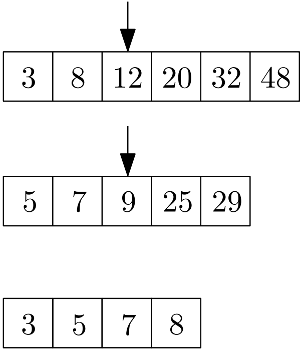
$O(n)$ (Linear) Running Time

- Merge two sorted arrays



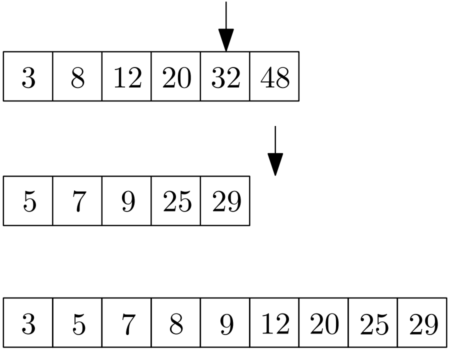
$O(n)$ (Linear) Running Time

- Merge two sorted arrays



$O(n)$ (Linear) Running Time

- Merge two sorted arrays



$O(n)$ (Linear) Running Time

- Merge two sorted arrays

3	8	12	20	32	48

5	7	9	25	29

3	5	7	8	9	12	20	25	29	32	48

$O(n)$ (Linear) Running Time

merge(B, C, n_1, n_2) \\ B and C are sorted, with length n_1 and n_2

$A \leftarrow []$; $i \leftarrow 1$; $j \leftarrow 1$
while $i \leq n_1$ and $j \leq n_2$ **do**
    if $B[i] \leq C[j]$ **then**
        append $B[i]$ to A; $i \leftarrow i + 1$
    else
        append $C[j]$ to A; $j \leftarrow j + 1$
if $i \leq n_1$ then append $B[i..n_1]$ to A
if $j \leq n_2$ then append $C[j..n_2]$ to A
return A

$O(n)$ (Linear) Running Time

merge(B, C, n_1, n_2) \\\\ B and C are sorted, with length n_1 and n_2

$A \leftarrow []$; $i \leftarrow 1$; $j \leftarrow 1$
while $i \le n_1$ and $j \le n_2$ **do**
  if $B[i] \le C[j]$ **then**
    append $B[i]$ to A; $i \leftarrow i + 1$
  else
    append $C[j]$ to A; $j \leftarrow j + 1$
if $i \le n_1$ then append $B[i..n_1]$ to A
if $j \le n_2$ then append $C[j..n_2]$ to A
return A

Running time $= O(n)$ where $n = n_1 + n_2$.

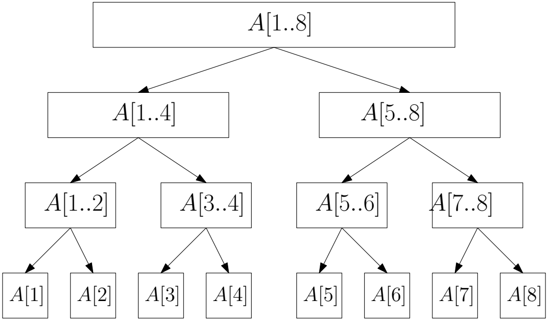
$O(n \log n)$ Running Time

merge-sort(A, n)

```
if n = 1 then
    return A
B ← merge-sort(A[1..⌊n/2⌋], ⌊n/2⌋)
C ← merge-sort(A[⌊n/2⌋ + 1..n], n − ⌊n/2⌋)
return merge(B, C, ⌊n/2⌋, n − ⌊n/2⌋)
```

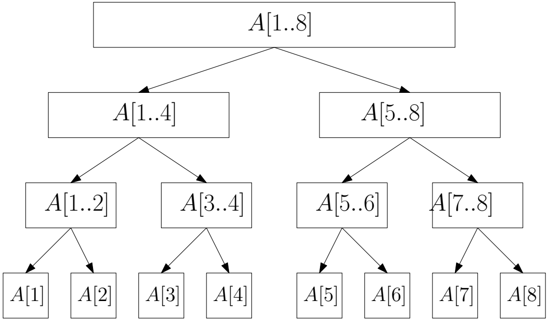
$O(n \log n)$ Running Time

- Merge-Sort



$O(n \log n)$ Running Time

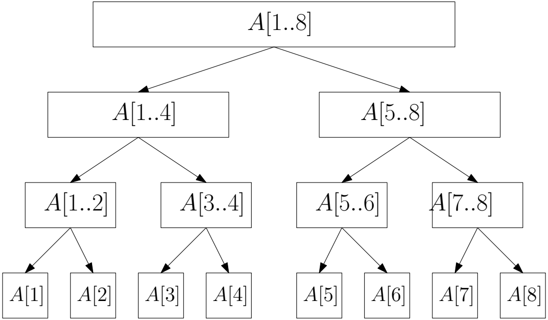
- Merge-Sort



- Each level takes running time $O(n)$

$O(n \log n)$ Running Time

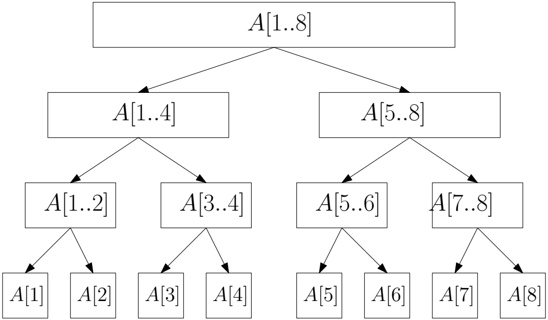
- Merge-Sort



- Each level takes running time $O(n)$
- There are $O(\log n)$ levels

$O(n \log n)$ Running Time

- Merge-Sort



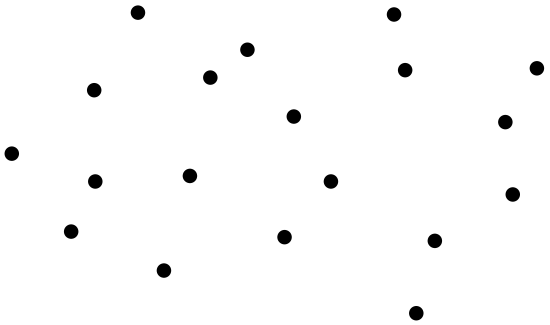
- Each level takes running time $O(n)$
- There are $O(\log n)$ levels
- Running time $= O(n \log n)$

$O(n^2)$ (Quardatic) Running Time

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

Output: the pair of points that are closest

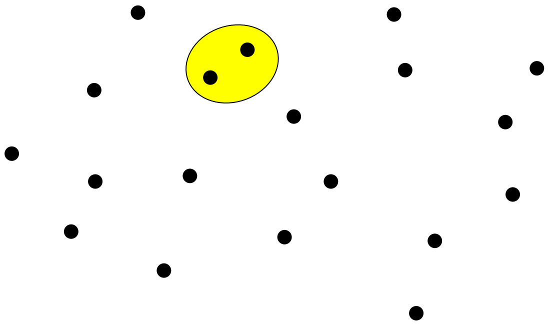


$O(n^2)$ (Quardatic) Running Time

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

Output: the pair of points that are closest



$O(n^2)$ (Quardatic) Running Time

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

Output: the pair of points that are closest

closest-pair(x, y, n)

$bestd \leftarrow \infty$
for $i \leftarrow 1$ to $n - 1$ **do**
    for $j \leftarrow i + 1$ to n **do**
        $d \leftarrow \sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}$
        if $d < bestd$ **then**
            $besti \leftarrow i, bestj \leftarrow j, bestd \leftarrow d$
return $(besti, bestj)$

$O(n^2)$ (Quardatic) Running Time

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

Output: the pair of points that are closest

closest-pair(x, y, n)

$bestd \leftarrow \infty$
for $i \leftarrow 1$ to $n-1$ **do**
    for $j \leftarrow i+1$ to n **do**
        $d \leftarrow \sqrt{(x[i]-x[j])^2 + (y[i]-y[j])^2}$
        if $d < bestd$ **then**
            $besti \leftarrow i, bestj \leftarrow j, bestd \leftarrow d$
return $(besti, bestj)$

Closest pair can be solved in $O(n \log n)$ time!

$O(n^3)$ (Cubic) Running Time

Multiply two matrices of size $n \times n$

matrix-multiplication(A, B, n)

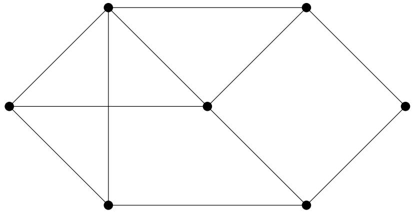
$C \leftarrow$ matrix of size $n \times n$, with all entries being 0
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
        for $k \leftarrow 1$ to n **do**
            $C[i, k] \leftarrow C[i, k] + A[i, j] \times B[j, k]$
return C

Beyond Polynomial Time: 2^n

Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.

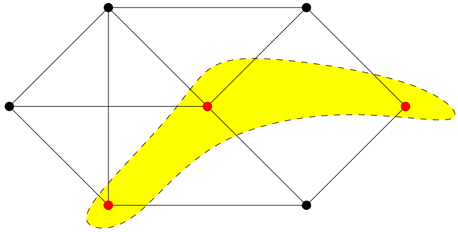
Beyond Polynomial Time: 2^n

Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.



Beyond Polynomial Time: 2^n

Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$.



Beyond Polynomial Time: 2^n

Maximum Independent Set Problem

Input: graph $G = (V, E)$

Output: the maximum independent set of G

Beyond Polynomial Time: 2^n

Maximum Independent Set Problem

Input: graph $G = (V, E)$

Output: the maximum independent set of G

max-independent-set($G = (V, E)$)

```
R ← ∅
for every set S ⊆ V do
    b ← true
    for every u, v ∈ S do
        if (u, v) ∈ E then b ← false
    if b and |S| > |R| then R ← S
return R
```

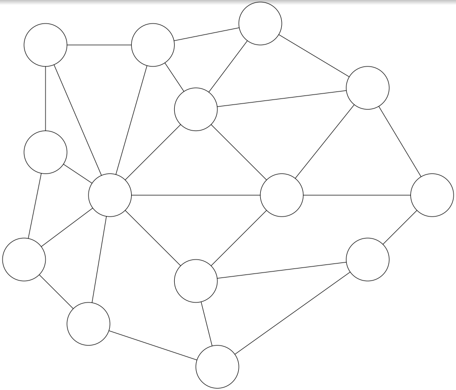
Running time $= O(2^n n^2)$.

Beyond Polynomial Time: $n!$

Hamiltonian Cycle Problem

Input: a graph with n vertices

Output: a cycle that visits each node exactly once, or say no such cycle exists

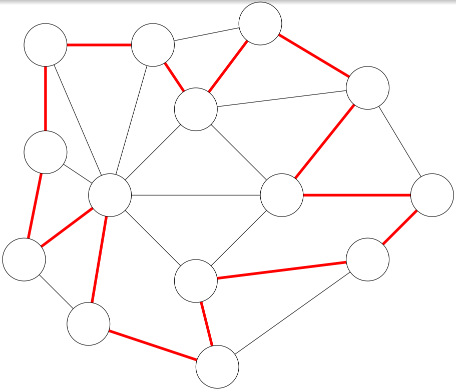


Beyond Polynomial Time: $n!$

Hamiltonian Cycle Problem

Input: a graph with n vertices

Output: a cycle that visits each node exactly once, or say no such cycle exists



Beyond Polynomial Time: $n!$

Hamiltonian($G = (V, E)$)

for every permutation $(p_1, p_2, \cdots, p_n)$ of V **do**
    $b \leftarrow$ true
    for $i \leftarrow 1$ to $n - 1$ **do**
        if $(p_i, p_{i+1}) \notin E$ then $b \leftarrow$ false
    if $(p_n, p_1) \notin E$ then $b \leftarrow$ false
    if b then return $(p_1, p_2, \cdots, p_n)$
return "No Hamiltonian Cycle"

Running time $= O(n! \times n)$

$O(\log n)$ (Logarithmic) Running Time

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

$42 > 35$

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

$25 < 35$

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

$37 > 35$

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array A of size n, an integer t;
  - Output: whether t appears in A.
- E.g, search 35 in the following array:

3	8	10	25	29	37	38	42	46	52	59	61	63	75	79

$O(\log n)$ (Logarithmic) Running Time

Binary search

- Input: sorted array A of size n, an integer t;
- Output: whether t appears in A.

binary-search(A, n, t)

$i \leftarrow 1, j \leftarrow n$
while $i \leq j$ **do**
    $k \leftarrow \lfloor (i+j)/2 \rfloor$
    if $A[k] = t$ return true
    if $t < A[k]$ then $j \leftarrow k - 1$ else $i \leftarrow k + 1$
return false

$O(\log n)$ (Logarithmic) Running Time

Binary search

- Input: sorted array A of size n, an integer t;
- Output: whether t appears in A.

binary-search(A, n, t)

```
i ← 1, j ← n
while i ≤ j do
    k ← ⌊(i + j)/2⌋
    if A[k] = t return true
    if t < A[k] then j ← k − 1 else i ← k + 1
return false
```

Running time $= O(\log n)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \ \ n \log n, \ \ n, \ \ n!, \ \ n^2, \ \ 2^n, \ \ e^n, \ \ n^n$
- $\log n = O(n)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n \log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$
- $n = O(n^2)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \ \ n \log n, \ \ n, \ \ n!, \ \ n^2, \ \ 2^n, \ \ e^n, \ \ n^n$
- $\log n = O(n)$
- $n = O(n \log n)$
- $n \log n = O(n^2)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n\log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$
- $n = O(n \log n)$
- $n \log n = O(n^2)$
- $n^2 = O(n!)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n\log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$
- $n = O(n \log n)$
- $n \log n = O(n^2)$
- $n^2 = O(2^n)$
- $2^n = O(n!)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \ n\log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n$
- $\log n = O(n)$
- $n = O(n\log n)$
- $n\log n = O(n^2)$
- $n^2 = O(2^n)$
- $2^n = O(e^n)$
- $e^n = O(n!)$

Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  $\log n, \;\; n\log n, \;\; n, \;\; n!, \;\; n^2, \;\; 2^n, \;\; e^n, \;\; n^n$
- $\log n = O(n)$
- $n = O(n \log n)$
- $n \log n = O(n^2)$
- $n^2 = O(2^n)$
- $2^n = O(e^n)$
- $e^n = O(n!)$
- $n! = O(n^n)$

Terminologies

When we talk about upper bound on running time:

- Logarithmic time: $O(\log n)$
- Linear time: $O(n)$
- Quadratic time $O(n^2)$
- Cubic time $O(n^3)$
- Polynomial time: $O(n^k)$ for some constant k
  - $O(n \log n) \subseteq O(n^{1.1})$. So, an $O(n \log n)$-time algorithm is also a polynomial time algorithm.
- Exponential time: $O(c^n)$ for some $c > 1$
- Sub-linear time: $o(n)$
- Sub-quadratic time: $o(n^2)$

Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

- Using asymptotic analysis allows us to ignore the leading constants and lower order terms

Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

- Using asymptotic analysis allows us to ignore the leading constants and lower order terms
- Makes our life much easier! (E.g., the leading constant depends on the implementation, complier and computer architecture of computer.)

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when n is big enough, $1000n < 0.1n^2$

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when n is big enough, $1000n < 0.1n^2$
- For "natural" algorithms, constants are not so big!

Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when n is big enough, $1000n < 0.1n^2$
- For "natural" algorithms, constants are not so big!
- So, for reasonably large n, algorithm with lower order running time beats algorithm with higher order running time.