Chapter 2: Complexity of Algorithms and Lower Bounds of Problems

Chin Lung Lu

Department of Computer Science

National Tsing Hua University

What is algorithm?

- Simply speaking, an algorithm is a computational method that can be used to solve a problem.
- More importantly, we can implement a program based on an algorithm such that the program can automatically solve the problem.

What is a good algorithm?

- An algorithm is good if it takes a short time and requires a small amount of memory space.
- Traditionally, the needed time is a more important factor to determine the goodness of an algorithm.

Time complexity of algorithms

How to measure the running time of an algorithm?

Method 1:

Write a program for the algorithm and see how fast it runs.

- However, this method is not appropriate, because there are so many factors unrelated to the algorithm which can affect the performance of the program.

Time complexity of algorithms (cont'd)

How to measure the running time of an algorithm?

Method 2:

Perform a mathematical analysis to determine the number of all the steps needed to complete the algorithm.

- In fact, we can choose some particular steps that are the most time-consuming operations in the algorithm.

Example:

Comparison (or movement) of data in sorting algorithms.

Time complexity of algorithms (cont'd)

- Usually, the time of executing an algorithm is dependent on the size of the problem, denoted by n.

Example:

The number of data in the sorting problem is the problem size.

- Most algorithms need more time to complete when n increases.

Time complexity of algorithms (cont'd)

Example:

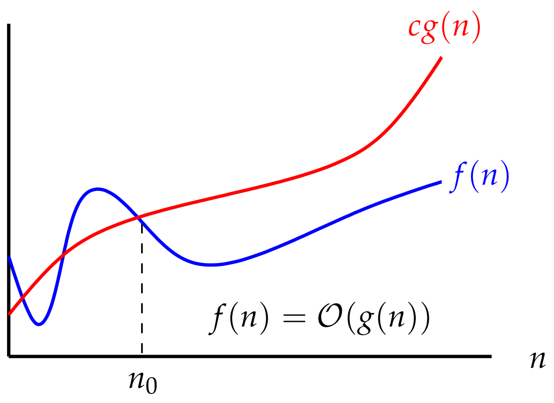
Suppose that it takes $n^3 + n$ steps to run an algorithm. We would often say that the time complexity of this algorithm is in the order of n^3.

- The reason is that the term n^3 dominates n.
- It means that as n becomes very large, n is not so significant when compared with n^3.

Big-Oh notation

Definition:

$f(n) = \mathcal{O}(g(n))$ if and only if there exist two positive constants c and n_0 such that $f(n) \le cg(n)$ for all $n \ge n_0$.



$f(n)$ is bounded by $cg(n)$ as n is large enough.

Big-Oh notation (cont'd)

Example 1:

- Suppose that it takes $n^3 + n$ steps to run an algorithm A.
- Then the time complexity of A is $\mathcal{O}(n^3)$
- Let $f(n) = n^3 + n$.

$$\begin{aligned} f(n) &= n^3 + n \\ &= (1 + \tfrac{1}{n^2})n^3 \\ &\le 2n^3 \qquad \text{for } n \ge 1 \end{aligned}$$

- We have $f(n) \le cg(n)$ for all $n \ge n_0$ by letting $g(n) = n^3$, $c = 2$ and $n_0 = 1$.
- Hence, the time complexity of A is $\mathcal{O}(n^3)$.

Big-Oh notation (cont'd)

Example 2: Let $f(n) = 2n^2 - 3n$.

1. $f(n) = \mathcal{O}(n^2)$ ✓
2. $f(n) = \mathcal{O}(n^3)$ ✓
3. $f(n) = \mathcal{O}(n)$ ✗
4. $f(n) = \mathcal{O}(1)$ ✗

Big-Oh notation (cont'd)

Example 3: Let $f(n) = 2^{100}n^2 - 3n$.

1. $f(n) = \mathcal{O}(n^2)$ ✓
2. $f(n) = \mathcal{O}(n^3)$ ✓
3. $f(n) = \mathcal{O}(n)$ ✗
4. $f(n) = \mathcal{O}(1)$ ✗

A misunderstanding about big-Oh

Example:

- Let A_1 and A_2 be two algorithms of solving the same problem and their time complexities be $\mathcal{O}(n^3)$ and $\mathcal{O}(n)$, respectively.
- We ask the same person to write two programs, say P_1 and P_2 respectively, for A_1 and A_2 and run these two programs under the same programming environment.

Question:

Would program P_2 always run faster than program P_1?

- The answer is not necessarily true.

A misunderstanding about big-Oh (cont'd)

- It is a common mistake to think that P_2 will always run faster than P_1.

Example:

- Let f_1 and f_2 be the time complexities of algorithms A_1 and A_2, respectively.
- Suppose that $f_1 = n^3$ and $f_2 = 100n$.

1. $f_1 > f_2$ for $n > 10$.

   (It means that P_2 runs faster than P_1 when n is large.)
2. $f_1 < f_2$ for $n < 10$.

   (It means that P_1 may run faster than P_2 when n is small.)

A misunderstanding about big-Oh (cont'd)

- Actually, the constant hidden in $\mathcal{O}$-notation can not be ignored.
- However, no matter how large this constant, its significance decreases as n increases.

Significance of order

$n \setminus f(n)$	$\log_2 n$	n	$n \log_2 n$	n^2	2^n	n!
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
10^2	0.006 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
10^3	0.010 μs	1 μs	9.644 μs	1 ms		
10^4	0.013 μs	10 μs	130 μs	100 ms		
10^5	0.016 μs	0.1 ms	1.67 ms	10 sec		
10^6	0.020 μs	1 ms	19.93 ms	16.7 min		
10^7	0.023 μs	0.01 sec	0.23 sec	1.16 days		
10^8	0.026 μs	0.1 sec	2.66 sec	115.7 days		(10^{-9} sec/op)
10^9	0.030 μs	1 sec	20.90 sec	31.7 years		

Significance of order (cont'd)

- It is very meaningful if we can find an algorithm with lower order time complexity.
- While we may dislike the time-complexity functions, such as n^2, n^3, etc., they are still tolerable when compared with 2^n.

Polynomial and exponential algorithms

Definition:
A polynomial-time algorithm is any algorithm with time complexity $\mathcal{O}(f(n))$, where $f(n)$ is a polynomial function of input size n.

- Examples: $\mathcal{O}(1)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n)$ and $\mathcal{O}(n^{2000})$

Definition:
An exponential-time algorithm is any algorithm whose time complexity can not be bounded by a polynomial function.

- Examples: $\mathcal{O}(2^n)$ and $\mathcal{O}(n!)$

Three cases of algorithm analyses

- For any algorithm, we are interested in its behavior under three cases: best case, average case and worst case.
- Let $T(I)$ be the running time of an algorithm A for instance I.

Definition (time complexity of A in the worst case):
$\max\{T(I):$ all instances $I\}$.

Definition (time complexity of A in the best case):
$\min\{T(I):$ all instances $I\}$.

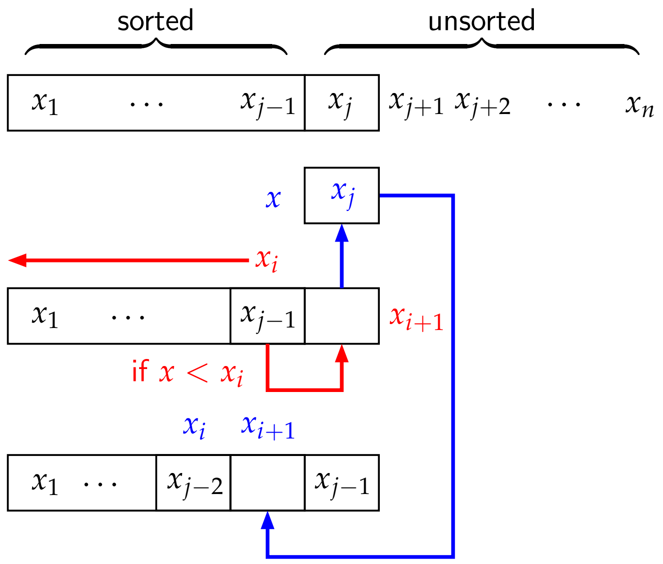
Definition (time complexity of A in the average case):
sum$\{T(I) \times p(I):$ all instances $I\}$, where $p(I)$ is the probability of the occurrence of I.

Insertion sort algorithm

Input: A sequence of n numbers $x_1, x_2, \ldots, x_n$.
Output: The sorted sequence of $x_1, x_2, \ldots, x_n$.

```
1. for $j = 2$ to $n$ do /* Outer loop */
2.     $x = x_j$
3.     $i = j - 1$
4.     while $x < x_i$ and $i > 0$ do /* Inner loop */
5.         $x_{i+1} = x_i$
6.         $i = i - 1$
7.     end while
8.     $x_{i+1} = x$
9. end for
```

Insertion sort algorithm (cont'd)


sorted
unsorted
x_1 $\cdots$ x_{j-1} x_j x_{j+1} x_{j+2} $\cdots$ x_n
x
x_j
x_i
x_1 $\cdots$ x_{j-1}
x_{i+1}
if $x < x_i$
x_i x_{i+1}
x_1 $\cdots$ x_{j-2} x_{j-1}

Insertion sort algorithm (cont'd)

Example:

Let the input sequence be 7, 5, 1, 4, 3, 2, 6.

	Sorted sequence	Unsorted sequence
Input data		7,5,1,4,3,2,6
Initial state	7	5,1,4,3,2,6
$j = 2$	5, 7	1,4,3,2,6
$j = 3$	1, 5, 7	4,3,2,6
$j = 4$	1, 4, 5, 7	3,2,6
$j = 5$	1, 3, 4, 5, 7	2,6
$j = 6$	1, 2, 3, 4, 5, 7	6
$j = 7$	1, 2, 3, 4, 5, 6, 7	

Time complexity of insertion sort algorithm

- In our analysis below, we use the number of data movements X as the time complexity measurement.
- Outer loop: $x = x_j$ and $x_{i+1} = x$ (always executed)
- Inner loop: $x_{i+1} = x_i$ (not always executed)
- Let d_j be the number of the data movements for x_i in the inner while loop, that is, $d_j = \left|\{x_i : x_i > x_j, 1 \le i < j\}\right|$.

Lemma:

$X = \sum_{j=2}^{n} (2 + d_j)$.

- That is, $X = 2(n-1) + \sum_{j=2}^{n} d_j$.

Best case of insertion sort algorithm

- The best case of the insertion sort occurs when the input data are already sorted.

Example:

The input data is 1, 2, 3, 4, 5, 6, 7.

- In this case, we have $d_2 = d_3 = \ldots = d_n = 0$.
- Therefore, $X = 2(n-1) = \mathcal{O}(n)$.

Worst case of insertion sort algorithm

- The worst case of the insertion sort occurs when the input data are reversely sorted.

Example:

The input data is 7, 6, 5, 4, 3, 2, 1.

- In this case, we have $d_2 = 1, d_3 = 2, \ldots, d_n = n - 1$.
- Therefore, $\sum_{j=2}^{n} d_j = \frac{n(n-1)}{2}$ and as a result, we have:

$$X = 2(n-1) + \frac{n(n-1)}{2} = \frac{(n-1)(n+4)}{2} = \mathcal{O}(n^2)$$

Average case of insertion sort algorithm

- Assume $x_1, x_2, \ldots, x_{j-1}$ is already a sorted sequence and the next data to be inserted is x_j.
- Suppose x_j is the kth largest number among the j numbers.
- There are $k-1$ movements in the inner loop, where $1 \leq k \leq j$.
- Moreover, there are 2 movements in the outer loop.
- The probability that x_j is the kth largest among j numbers is $\frac{1}{j}$.
- The average number of movement when considering x_j is:

$$\frac{2+0}{j} + \frac{2+1}{j} + \ldots + \frac{2+j-1}{j} = \frac{\frac{(j+3)j}{2}}{j} = \frac{j+3}{2}$$

Average case of insertion sort algorithm (cont'd)

- As a result, the average time complexity of the insertion sort is:

$$\sum_{j=2}^{n} \frac{j+3}{2} = \frac{1}{2}\left(\sum_{j=2}^{n} j + \sum_{j=2}^{n} 3\right) = \frac{(n+8)(n-1)}{4} = \mathcal{O}(n^2)$$

Time complexities of insertion sort algorithm

Theorem:
In summary, the time complexities of insertion sort are as follows:

- Best case: $\mathcal{O}(n)$
- Average case: $\mathcal{O}(n^2)$
- Worst case: $\mathcal{O}(n^2)$

Selection sort algorithm

Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$.
Output: The sorted sequence $a_1, a_2, \ldots, a_n$.

```
1. for $j = 1$ to $n-1$ do /* find the $j$th smallest number */
2.     $f = j$ /* $f$ is a flag */
3.     for $k = j+1$ to $n$ do
4.         if $a_k < a_f$, then $f = k$
5.     end for
6.     $a_j \leftrightarrow a_f$ /* exchange $a_j$ with $a_f$ */
7. end for
```

Example of selection sort algorithm

- Let the input sequence be 7, 5, 1, 4, 3, 2, 6.

Step 1: Find the 1th smallest number

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
$j=1$	7	5	1	4	3	2	6
$f \leftarrow k$	$\uparrow f$	$\uparrow k$					
$f \leftarrow k$		$\uparrow f$	$\uparrow k$				
			$\uparrow f$	$\uparrow k$			
			$\uparrow f$		$\uparrow k$		
			$\uparrow f$			$\uparrow k$	
			$\uparrow f$				$\uparrow k$
$a_j \leftrightarrow a_f \equiv a_1 \leftrightarrow a_3$	1	5	7	4	3	2	6

Example of selection sort algorithm (cont'd)

Step 1: Find the 2nd smallest number

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
$j=2$	1	5	7	4	3	2	6
		$\uparrow f$	$\uparrow k$				
$f \leftarrow k$		$\uparrow f$		$\uparrow k$			
				$\uparrow f$	$\uparrow k$		
$f \leftarrow k$					$\uparrow f$	$\uparrow k$	
						$\uparrow f$	$\uparrow k$
$a_j \leftrightarrow a_f \equiv a_2 \leftrightarrow a_6$	1	2	7	4	3	5	6

Operations of selection sort algorithm

Note:
There are two operations in the selection sort:

1. Comparisons of two elements: "if $a_k < a_f$".
2. Change of the flag: "$f = k$".

- The number of comparisons of two elements is $\frac{n(n-1)}{2}$, which is a fixed number.
- That is, no matter what the input data are, we always have to perform $\frac{n(n-1)}{2}$ comparisons.

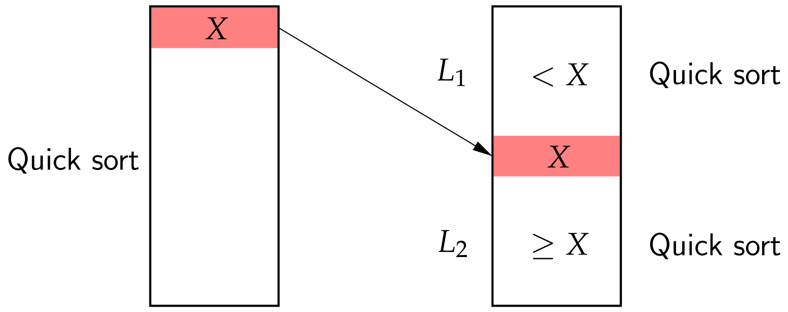
Time complexities of selection sort

Theorem:
The time complexities of the selection sort (when measured by the number of comparisons) are as follows:

- Best case: $\mathcal{O}(n^2)$
- Average case: $\mathcal{O}(n^2)$
- Worst case: $\mathcal{O}(n^2)$

Quick sort

- The basic idea of quick sort (divide and conquer) is as follows:



Algorithm of quick sort

Algorithm: *Quicksort*(f, l)
Input: A sequence of $(l - f + 1)$ numbers $a_f, a_{f+1}, \ldots, a_l$.
Output: The sorted sequence of $a_f, a_{f+1}, \ldots, a_l$.

```
1.  if $f \geq l$, then return
2.  $X = a_f$, $i = f$, $j = l$
3.  while $i < j$ do
4.     while $a_j \geq X$ do
5.         $j = j - 1$
6.     $a_i \leftrightarrow a_j$
7.     while $a_i < X$ do
8.         $i = i + 1$
9.     $a_i \leftrightarrow a_j$
10. end while
11. $Quicksort(f, j - 1)$, $Quicksort(j + 1, l)$
```

Example of quick sort

Iteration 1: Let
$a_1 = 3, a_2 = 6, a_3 = 1, a_4 = 4, a_5 = 5, a_6 = 2.$

$X = 3$	a_1	a_2	a_3	a_4	a_5	a_6
$i = 1, j = 6$	3	6	1	4	5	2
$(a_j = a_6 < X)$	$\uparrow i$					$\uparrow j$
$a_1 \leftrightarrow a_6$	2	6	1	4	5	3
$(a_i = a_1 < X)$	$\uparrow i$					$\uparrow j$
$i = i + 1 = 2$	2	6	1	4	5	3
$(a_i = a_2 > X)$		$\uparrow i$				$\uparrow j$
$a_2 \leftrightarrow a_6$	2	3	1	4	5	6
$(a_j = a_6 > X)$		$\uparrow i$				$\uparrow j$
$j = j - 1 = 5$	2	3	1	4	5	6
$(a_j = a_5 > X)$		$\uparrow i$			$\uparrow j$	

Example of quick sort (cont'd)

Iteration 1 (cont'd):

$X = 3$	a_1	a_2	a_3	a_4	a_5	a_6
$j = j - 1 = 4$	2	3	1	4	5	6
$(a_j = a_4 > X)$		$\uparrow i$		$\uparrow j$		
$j = j - 1 = 3$	2	3	1	4	5	6
$(a_j = a_3 < X)$		$\uparrow i$	$\uparrow j$			
$a_2 \leftrightarrow a_3$	2	1	3	4	5	6
$(a_i = a_2 < X)$		$\uparrow i$	$\uparrow j$			
$i = i + 1 = 3$	2	1	3	4	5	6
$(i = j = 3)$			$i \uparrow j$			
(end of iteration 1)	≤ 3	≤ 3	$= 3$	≥ 3	≥ 3	≥ 3

Best case of quick sort

- The best case occurs when X splits the list right in the middle for each round.
- That is, X produces two sublists that contain the same number of elements.
- Each round needs $\mathcal{O}(n)$ steps to split the lists.
- For example, the first round needs cn steps to split the list, where c is a constant.
- Moreover, the second round needs $2 \cdot \frac{cn}{2} = cn$ steps to split its lists.
- Assume $n = 2^p$.
- We then need totally p rounds, where $p = \log_2 n$.
- Hence, the total time complexity of the best case is $cn \log_2 n = \mathcal{O}(n \log_2 n)$.

Worst case of quick sort

- The worst case occurs when the input data are sorted or reversely sorted.
- In this case, we need totally n rounds.
- Hence, the time complexity of the worst case is:

$$cn + c(n-1) + \ldots + c = \frac{c(n+1)n}{2} = \mathcal{O}(n^2)$$

Average case of quick sort

- Let $T(n)$ denote the number of steps in the average case for n elements.
- Assume after splitting, the first sub-list contains $s-1$ elements and the second contains $n-s$ elements, where $1 \leq s \leq n$.
- By considering all possible cases, we have:

$$T(n) \;=\; \underset{1 \leq s \leq n}{Ave}\,(T(s-1) + T(n-s)) + \mathcal{O}(n)$$

  where $\mathcal{O}(n)$ is the number of operations needed for the first splitting operation.
- For simplifying computation, we let:

$$T(n) \;=\; \underset{1 \leq s \leq n}{Ave}\,(T(s-1) + T(n-s)) + c(n+1)$$

Average case of quick sort (cont'd)

- We can express $Ave_{1 \leq s \leq n}(T(s-1) + T(n-s))$ as follows:

$$\begin{aligned}
&\underset{1 \leq s \leq n}{Ave}\,(T(s-1) + T(n-s)) \\
&= \tfrac{1}{n}(T(0) + T(n-1) + \ldots + T(n-1) + T(0)) \\
&= \tfrac{1}{n}(2T(0) + 2T(1) + \ldots + 2T(n-1))
\end{aligned}$$

- Since $T(0) = 0$, we have:

$$\begin{aligned}
T(n) &= \underset{1 \leq s \leq n}{Ave}\,(T(s-1) + T(n-s)) + c(n+1) \\
&= \tfrac{1}{n}(2T(1) + 2T(2) + \ldots + 2T(n-1)) + c(n+1)
\end{aligned}$$

$$\Leftrightarrow nT(n) = (2T(1) + 2T(2) + \ldots + 2T(n-1)) + cn(n+1)$$

- By substituting $n = n-1$ into the above formula, we have:

$$(n-1)T(n-1) = 2T(1) + \ldots + 2T(n-2) + c(n-1)n$$

Average case of quick sort (cont'd)

- Therefore, we have:

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn$$
$$nT(n) = (n+1)T(n-1) + 2cn$$
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

- Recursively, we have:

$$\begin{aligned} \frac{T(n-1)}{n} &= \frac{T(n-2)}{n-1} + \frac{2c}{n} \\ \frac{T(n-2)}{n-1} &= \frac{T(n-3)}{n-2} + \frac{2c}{n-1} \\ &\vdots \\ \frac{T(1)}{2} &= \frac{T(0)}{1} + \frac{2c}{2} \end{aligned}$$

Average case of quick sort (cont'd)

- Therefore, we have:

$$\begin{aligned} \frac{T(n)}{n+1} &= 2c\left(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{2}\right) \\ &= 2c(H_{n+1} - 1) \end{aligned}$$

- Note that $H_n = \frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{1}$ and $H_n \cong \log_e n$ when $n \to \infty$.
- Finally, we have:

$$\begin{aligned} T(n) &= 2c(n+1)(H_{n+1} - 1) \\ &\cong 2c(n+1)\log_e(n+1) - 2c(n+1) \\ &= \mathcal{O}(n \log_2 n) \end{aligned}$$

Time complexities of quick sort algorithm

Theorem:
In summary, the time complexities of quick sort are as follows:

- Best case: $\mathcal{O}(n \log_2 n)$
- Average case: $\mathcal{O}(n \log_2 n)$
- Worst case: $\mathcal{O}(n^2)$

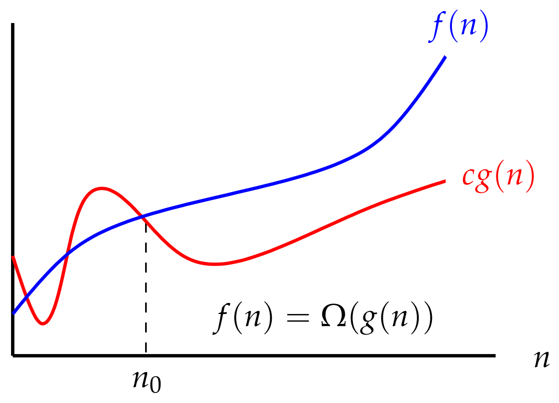
Lower bound of problem

Definition:
A lower bound of a problem is the least time complexity required for any algorithm that can be used to solve this problem.

- The time complexity used in the above definition usually refers to the worst-case time complexity.
- Hence, this lower bound is called worst-case lower bound.
- To describe the lower bound, we shall use a notation Ω.

Big-Omega notation

Definition:
$f(n) = \Omega(g(n))$ if and only if there exist two positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.



Big-Omega notation (cont'd)

Example:
Let $f(n) = 2n^2 + 3n$.

1. $f(n) = \Omega(n^2)$ ✓
2. $f(n) = \Omega(n^3)$ ✗
3. $f(n) = \Omega(n)$ ✓
4. $f(n) = \Omega(1)$ ✓

Determination of problem lower bounds

Question:
How to determine the lower bounds of a problem?

Exhaustive method:

1. Enumerate all possible algorithms.
2. Determine the time complexity of each algorithm.
3. Find the minimum time complexity.

- It is impossible to enumerate all possible algorithms.

Determination of problem lower bounds (cont'd)

Example: What are the lower bounds of sorting?

1. $\Omega(1)$: At least one step to complete any sorting algorithm.
2. $\Omega(n)$: Every data element must be examined before it's sorted.
3. $\Omega(n \log n)$: This requires a theoretical proof.

- The lower bound of a problem is not unique.
- $\Omega(n \log n)$ is more significant than $\Omega(1)$ and $\Omega(n)$.
- We like the lower bound to be as high as possible.
- Each higher lower bound is found by theoretical analysis, not by pure guessing.

Upper limit of lower bound

- As the lower bound of a problem goes higher and higher, we will inevitably wonder whether there is an upper limit of the lower bound?

Question:

Is there any possibility that $\Omega(n^2)$ is a lower bound of the sorting problem?

- Answer: no!

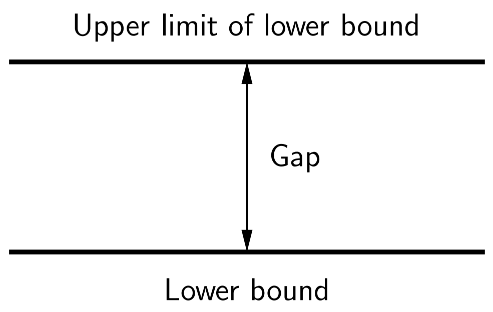
Upper limit of lower bound (cont'd)

- The time complexity of the best one among currently available algorithms for a problem can be considered as the upper limit of the lower bound.

Lower bound and its upper limit

Case 1:

The highest lower bound of a problem is $\Omega(n \log n)$ and the time complexity of the best available algorithm to solve this problem is $\mathcal{O}(n^2)$.



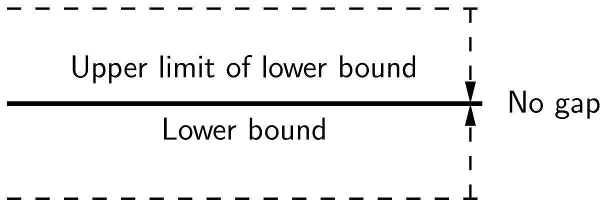
Lower bound and its upper limit (cont'd)

In this case, there are three possibilities:

1. The lower bound of the problem is too low.
   $\Rightarrow$ We should find a higher lower bound.
2. The best available algorithm is not good enough.
   $\Rightarrow$ We should find a better algorithm.
3. Both the lower bound and the algorithm may be improved.
   $\Rightarrow$ We should try to improve both.

Lower bound and its upper limit (cont'd)

Case 2:
The present lower bound is $\Omega(n \log n)$ and there is indeed an algorithm with time complexity $\mathcal{O}(n \log n)$.



- In this case, the lower bound and the algorithm cannot be improved any further.
- It means that we have found an optimal algorithm to solve the problem and a truly significant lower bound of this problem.

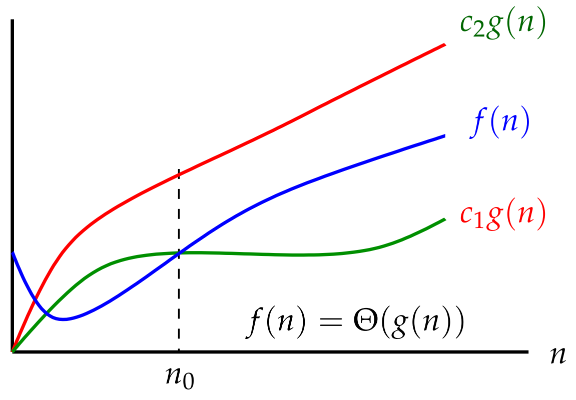
Optimal algorithm

Definition:
An algorithm is optimal if its time complexity is equivalent to a lower bound of the problem.

- It means that neither the lower bound nor the algorithm can be improved further.

Big-Theta notation

Definition:
$f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



Big-Theta notation (cont'd)

Example:
Let $f(n) = \frac{1}{2}n^2 - 3n$.

1. $f(n) = \Theta(n^2)$ ✓
2. $f(n) = \Theta(n^3)$ ✗
3. $f(n) = \Theta(n)$ ✗
4. $f(n) = \Theta(1)$ ✗

Binary decision tree

- For many (comparison-based) algorithms, their executions can be described as binary decision trees.

Example:

Consider the case of insertion sort with the input of 3 different elements (a_1, a_2, a_3).

- Then there are 6 distinct permutations (instances).

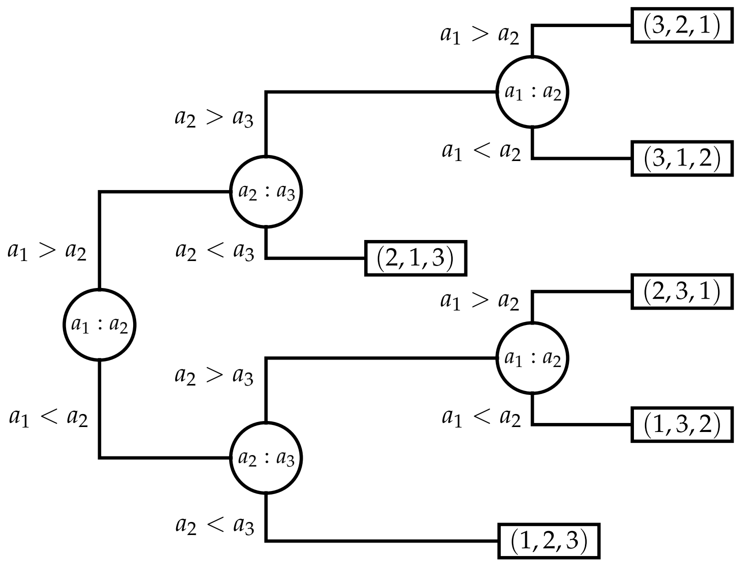
a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Binary decision tree (cont'd)

Algorithm: Insertion sort (revised)
Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$.
Output: The sorted sequence of $a_1, a_2, \ldots, a_n$.

```
1. for $j = 2$ to $n$ do
2.     $i = j$
3.     while $a_{i-1} > a_i$ and $i > 1$ do
4.         $a_{i-1} \leftrightarrow a_i$    /* Exchange $a_{i-1}$ with $a_i$ */
5.         $i = i - 1$
6.     end while
7. end for
```

Binary decision tree (cont'd)



Binary decision tree (cont'd)

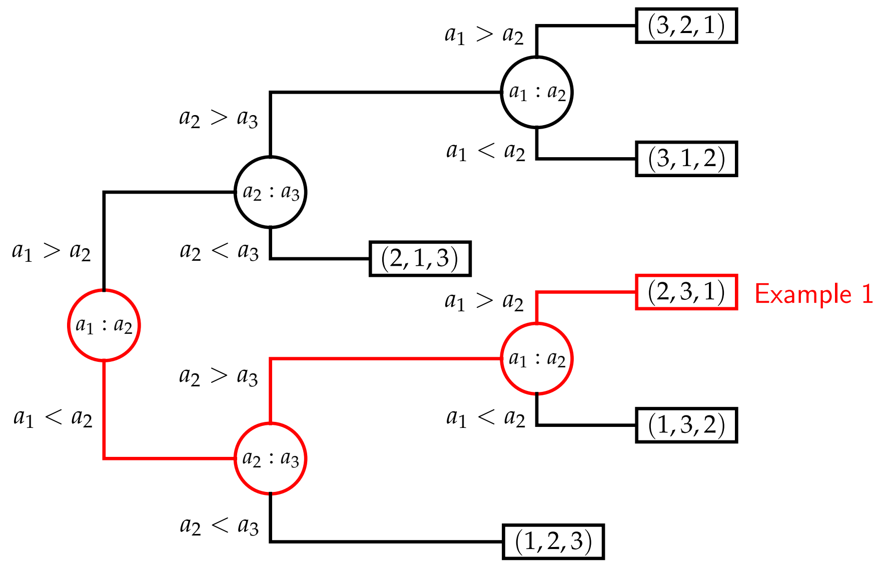
- When we apply the insertion sort to the above set of data, each permutation has a distinct response.

Example 1:

Suppose that the input is $(a_1, a_2, a_3) = (2, 3, 1)$. Then the insertion sort behaves as follows.

1. Compare $a_1 = 2$ with $a_2 = 3$.
   Since $a_2 > a_1$, no exchange of data elements takes place.
2. Compare $a_3 = 1$ with $a_2 = 3$.
   Since $a_3 < a_2$, we exchange a_3 and a_2.
   As a result, $(a_1, a_2, a_3) = (2, 1, 3)$.
3. Compare $a_2 = 1$ with $a_1 = 2$.
   Since $a_2 < a_1$, we exchange a_2 and a_1.
   As a result, $(a_1, a_2, a_3) = (1, 2, 3)$.

Binary decision tree (cont'd)



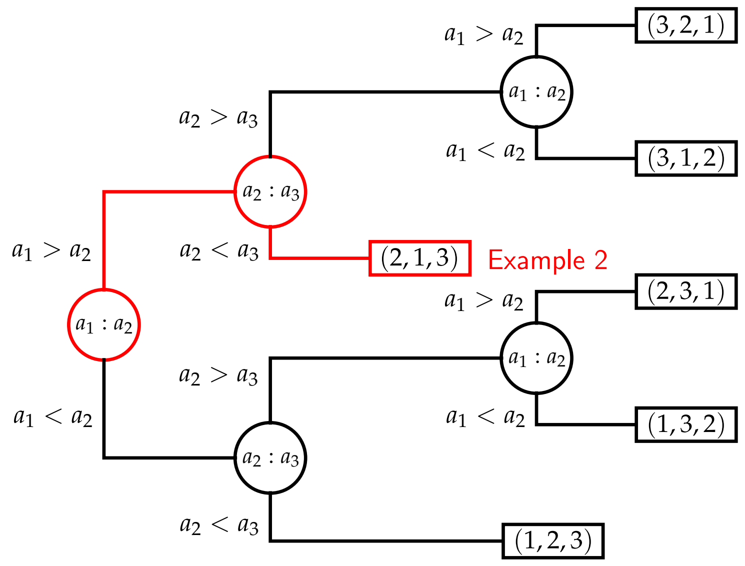
Binary decision tree (cont'd)

Example 2:

Suppose that the input is $(a_1, a_2, a_3) = (2, 1, 3)$. Then the insertion sort behaves as follows.

1. Compare $a_1 = 2$ with $a_2 = 1$.
   Since $a_2 < a_1$, we exchange a_2 and a_1.
   As a result, $(a_1, a_2, a_3) = (1, 2, 3)$.
2. Compare $a_3 = 3$ with $a_2 = 2$.
   Since $a_3 > a_2$, no exchange of data takes place.
   As a result, $(a_1, a_2, a_3) = (1, 2, 3)$.

Binary decision tree (cont'd)



Lower bound of sorting problem

- In general, any sorting algorithm whose basic operation is a compare-and-exchange operation can be described by a binary decision tree.
- The action of a sorting algorithm on a particular input data corresponds to one path from the root to a leaf, where each leaf node corresponds to a particular permutation.
- The length of the longest path from the root to a leaf (called tree depth) is the worst-case time complexity of this algorithm.
- The lower bound of the sorting problem is the smallest depth of some tree among all possible binary decision trees modeling sorting algorithms.

Lower bound of sorting problem (cont'd)

- For every sorting algorithm, its corresponding binary decision tree will have $n!$ leaf nodes, as there are $n!$ permutations.
- The depth of a balanced binary tree is the smallest.
- The depth of the balanced binary tree is $\lceil \log_2 n! \rceil$.
- The minimum number of comparisons to sort in the worst case is at least $\lceil \log_2 n! \rceil$.
- Hence, the worst-case lower bound of sorting is $\Omega(n \log_2 n)$.

  Stirling approximation formula: $n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

$$\begin{aligned} \log_2 n! &= \log_2 \sqrt{2\pi} + \tfrac{1}{2}\log_2 n + n\log_2 n - n\log_2 e \\ &\geq n\log_2 n - 1.44n \\ &= n\log_2 n\left(1 - \frac{1.44}{\log_2 n}\right) \\ &\geq 0.28n\log_2 n \text{ for } n \geq 4 \end{aligned}$$

Selection sort (revisited)

- Recall that for selection sort, we need $n-1$ steps to obtain the first smallest number, then $n-2$ steps to obtain the second smallest number, and so on (all in worst case).
- Hence, $\mathcal{O}(n^2)$ steps are needed for selection sort.
- Since the lower bound of sorting is $\Omega(n \log n)$, selection sort is not optimal.

Observation:

When we try to find the second smallest number, the information we have extracted by finding the first smallest number is not used at all.

Knockout sort

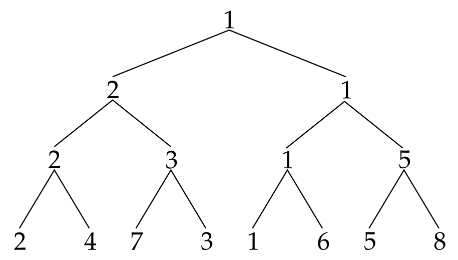
Algorithm: Knockout sort

1. Construct a knockout tree.
2. Output the smallest number, replace it by ∞, and update the knockout tree.
3. Repeat the above step untill all numbers are sorted.

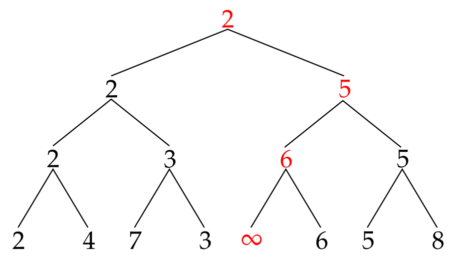
Knockout sort (cont'd)

Example:

Let the input date be $2, 4, 7, 3, 1, 6, 5, 8$.

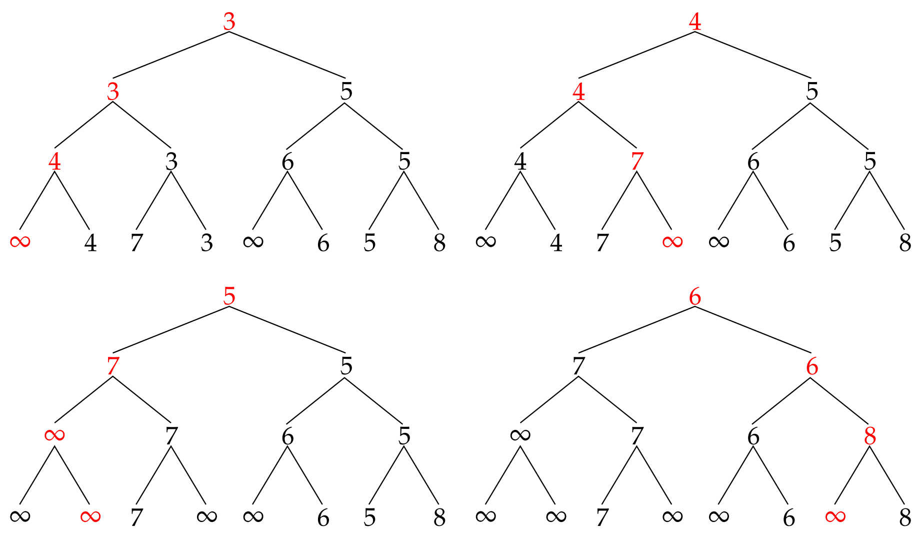


Knockout tree

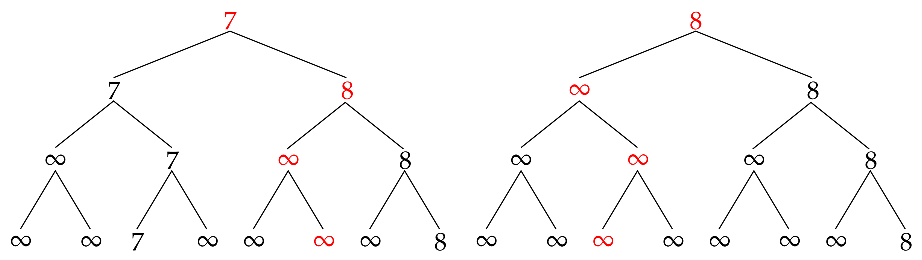


Update of knockout tree

Knockout sort (cont'd)

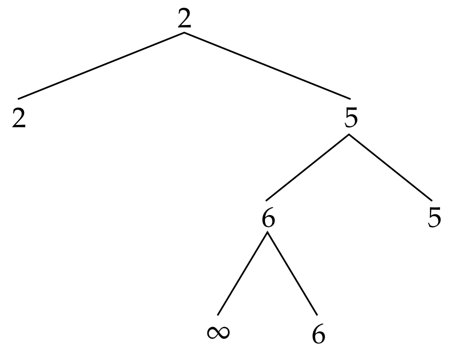


Knockout sort (cont'd)



Knockout sort (cont'd)

- Actually, knockout sort is similar to the selection sort.
- However, after finding the 1st smallest number, only a small part of the knockout tree needs to be examined for finding the 2nd smallest number.



Knockout sort (cont'd)

- The first smallest number is found after $n-1$ comparisons.
- For all of the other selections, only $\lceil \log_2 n \rceil - 1$ comparisons are needed.
- Therefore, the total number of comparisons is:

$$(n-1) + (n-1)(\lceil \log_2 n \rceil - 1) = \mathcal{O}(n \log n)$$

- The time complexity of the knockout sort is $\mathcal{O}(n \log n)$.
- This complexity is valid for best, average and worst cases.
- The knockout sort is an optimal sorting algorithm.
- The reason that the knockout sort is better than the selection sort is that it uses previous information.
- However, the knockout sort needs $2n-1$ space (i.e., tree size).

Heap

Definition:

A heap is a binary tree satisfying the following conditions:

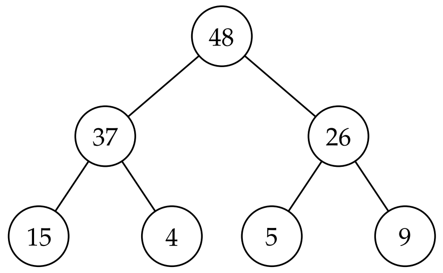
1. This tree is a complete binary tree.
2. Son's value $\leq$ parent's value.

Properties of complete binary tree:

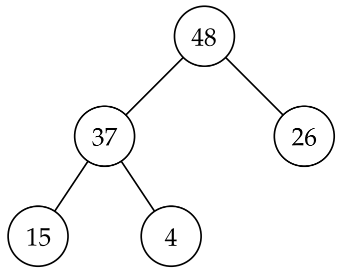
- A complete binary tree is a binary tree in which except possibly the last level, every level is completely filled, and all nodes are as far left as possible.

Heap (cont'd)

Example 1:



Example 2:



Heap sort

Algorithm: Heap sort

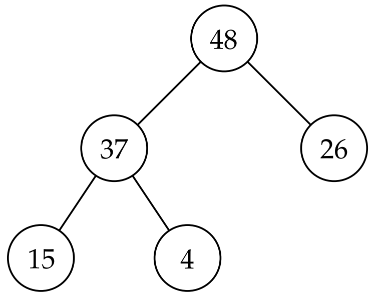
---

1. Construct the heap.
2. Output the largest number, replace it with the last number and restore the tree as a heap.
3. Repeat the above step until all the numbers are sorted.

Heap sort (cont'd)

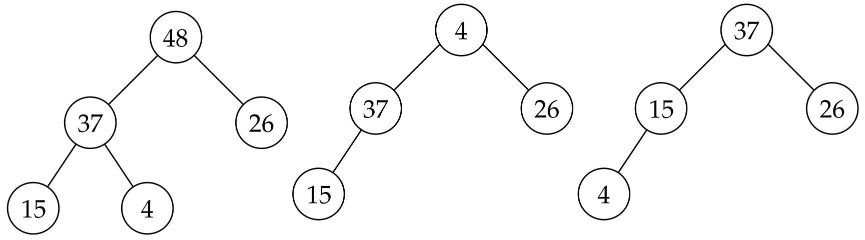
Example:

Consider five numbers 15, 37, 4, 48 and 26 and assume that their heap is already constructed.



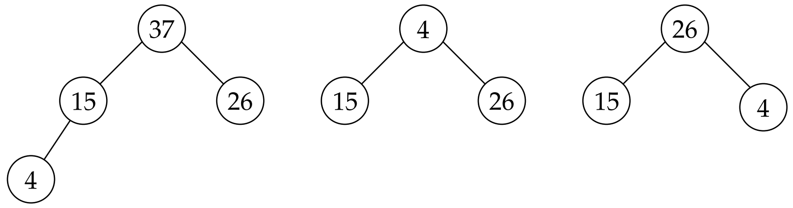
Heap sort (cont'd)

Step 1: Output 48 and restore the heap (by replacing 48 with 4).

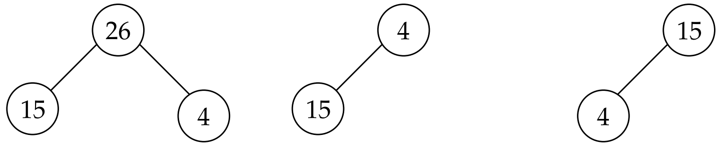


Heap sort (cont'd)

Step 2: Output 37 and restore the heap (by replacing 37 with 4).

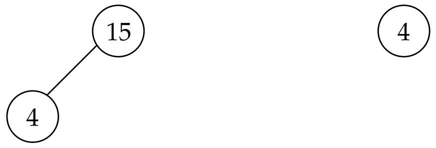


Step 3: Output 26 and restore the heap (by replacing 26 with 4).



Heap sort (cont'd)

Step 4: Output 15 and restore the heap.



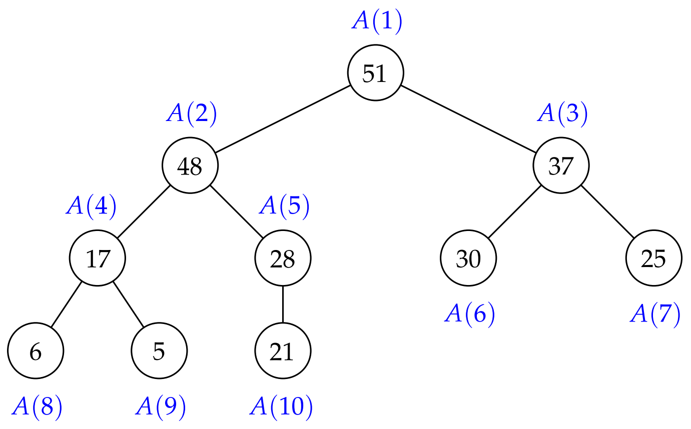
Step 5: Output 4.

Output of heap sort:

The output sequence is 48, 37, 26, 15, 4, which is sorted.

Heap sort (cont'd)

- We use an array (instead of pointers) to represent a heap.



$A(1)$	$A(2)$	$A(3)$	$A(4)$	$A(5)$	$A(6)$	$A(7)$	$A(8)$	$A(9)$	$A(10)$
51	48	37	17	28	30	25	6	5	21

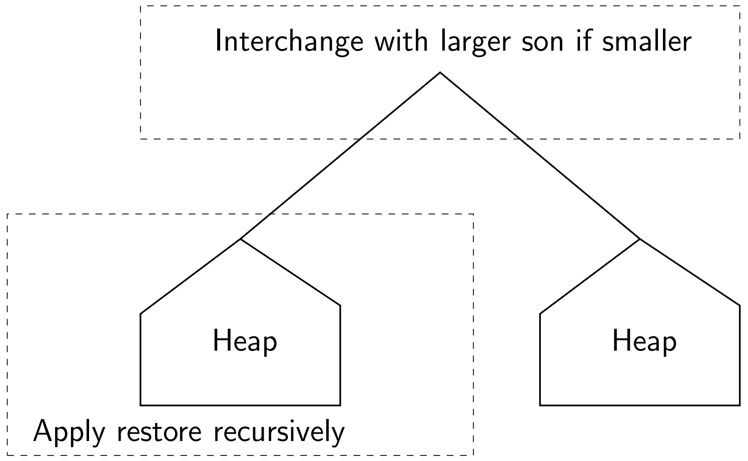
Heap sort (cont'd)

- Then we can uniquely determine each node and its descendants using the following rule:

The rule to determine the descendants of a node:
The descendants of $A(h)$ are $A(2h)$ and $A(2h+1)$, if they exist.

- Using an array to represent a heap, the entire process of heap sort can be operated on an array.

Restore routine of heap sort

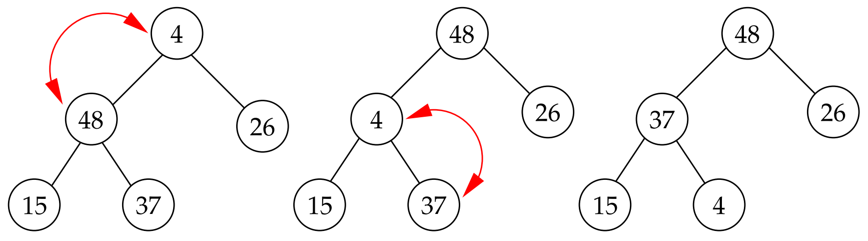

Interchange with larger son if smaller
Heap
Heap
Apply restore recursively

Restore routine of heap sort (cont'd)

Algorithm: Restore(i,j)
Input: $A(i), A(i+1), \cdots, A(j)$.
Output: $A(i), A(i+1), \cdots, A(j)$ as a heap.
Assumption: The subtrees rooted at sons of $A(i)$ are heaps.

```
1. if $A(i)$ is not a leaf and a son of $A(i)$ contains a larger element
   than $A(i)$ then
2.     Let $A(h)$ be the son of $A(i)$ with the largest element.
3.     Interchange $A(i)$ and $A(h)$
4.     $Restore(h,j)$
5. end if
```

Restore routine of heap sort (cont'd)



$Restore(1,5)$ $Restore(2,5)$ Heap

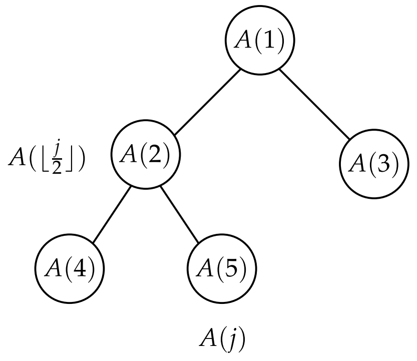
Restore routine of heap sort (cont'd)

- In the $Restore(i,j)$ routine, we use the parameter j to determine whether $A(i)$ is a leaf or not.

Note:
If $i > \lfloor \frac{j}{2} \rfloor$ (or $i > \frac{j}{2}$), then $A(i)$ is a leaf.

Restore routine of heap sort (cont'd)

Example:
Let $j = 5$. Then $\lfloor \frac{j}{2} \rfloor = 2$ and hence $A(3), A(4), A(5)$ are leaves.

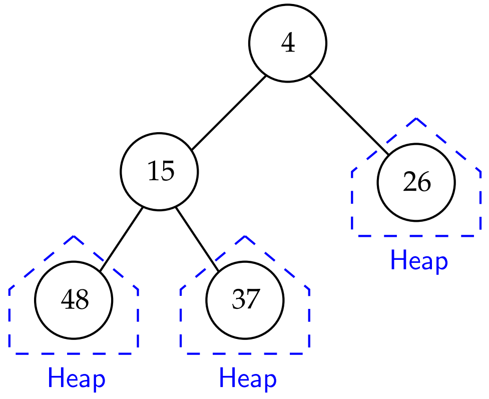


Construction of a heap

- Let $A(1), A(2), \ldots, A(n)$ be any complete binary tree.
- $A(i)$, where $i = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$, must be an internal node with descendants.
- $A(i)$, where $i = \lfloor \frac{n}{2} \rfloor + 1, \ldots, n$, must be a leaf node without descendants.
- For any complete binary tree, we can gradually transform it into a heap by repeatedly applying the *restore* routine on the subtrees rooted at nodes from $A(\lfloor \frac{n}{2} \rfloor)$ to $A(1)$.

Construction of a heap (cont'd)

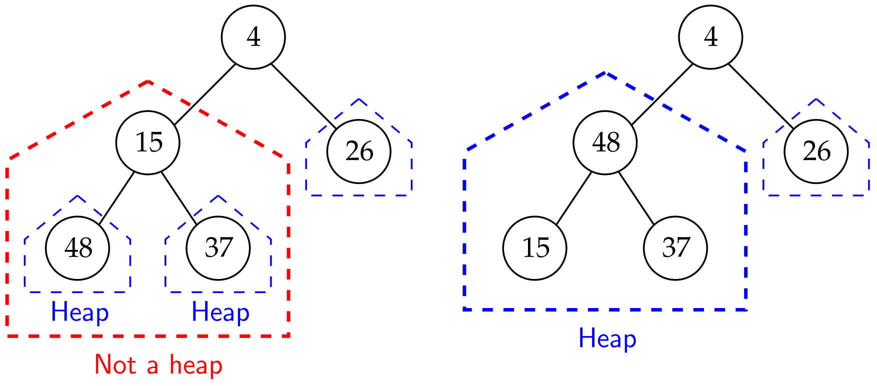
- Note that all leaf nodes can be considered as heaps.



- So we do not have to perform any operation on leaf nodes.

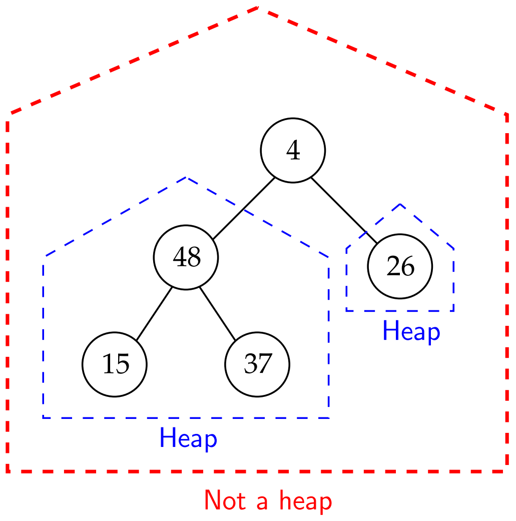
Construction of a heap (cont'd)

- Hence, we start the construction of a heap from restoring the subtree rooted at $A(\lfloor \frac{n}{2} \rfloor)$.



Construction of a heap (cont'd)

- We continue to restore the subtree rooted at root.



Algorithm of constructing a heap

Input: $A(1), A(2), \cdots, A(n)$.
Output: $A(1), A(2), \cdots, A(n)$ as a heap.

```
1. for $i = \lfloor n/2 \rfloor$ down to 1 do
2.     $Restore(i, n)$
3. end for
```

Time complexity of constructing a heap

- Recall that $A(i)$ is an internal node for $i = 1, 2, \cdots, \lfloor n/2 \rfloor$.
- Recall that $A(i)$ must be a leaf node for $i = \lfloor n/2 \rfloor + 1, \cdots, n$.
- The depth d of a heap is $\lfloor \log_2 n \rfloor$.
- Each internal node needs two comparisons.
- Each node at level L needs $2(d-L)$ comparisons.
- Each level L has at most 2^L nodes.
- The total number of comparisons for constructing a heap is:

$$\sum_{L=0}^{d-1} 2(d-L)2^L = \mathcal{O}(n)$$

Time complexity of constructing a heap (cont'd)

Note that $\sum_{i=0}^{k} i2^{i-1} = 2^k(k-1)+1$.

$$\begin{aligned}
\sum_{L=0}^{d-1} 2(d-L)2^L &= 2d\sum_{L=0}^{d-1} 2^L - 4\sum_{L=0}^{d-1} L2^{L-1} \\
&= 2d(2^d-1) - 4(2^{d-1}(d-1-1)+1) \\
&= 2d2^d - 2d - 4d2^{d-1} + 4\cdot 2^d - 4 \\
&= 4\cdot 2^d - 2d - 4 \\
&= 4\cdot 2^{\lfloor \log_2 n\rfloor} - 2\lfloor \log_2 n\rfloor - 4 \\
&\le 4\cdot 2^{\log_2 n} - 2\lfloor \log_2 n\rfloor - 4 \\
&= 4n - 2\lfloor \log_2 n\rfloor - 4 \\
&\le 4n
\end{aligned}$$

Algorithm of heap sort

Input: A heap of $A(1), A(2), \cdots, A(n)$.
Output: A sorted sequence of $A(1), A(2), \cdots, A(n)$.

```
1. for $i = n$ down to 2 do
2.     Output $A(1)$
3.     $A(1) = A(i)$
4.     Delete $A(i)$
5.     $Restore(1, i-1)$
6. end for
7. Output $A(1)$
```

Time complexity of heap sort

- After deleting a number, $2\lfloor \log_2 i\rfloor$ comparisons (in the worst case) are needed to restore the heap if there are i elements remaining.
- Therefore, the total number of comparisons needed to delete all numbers is:

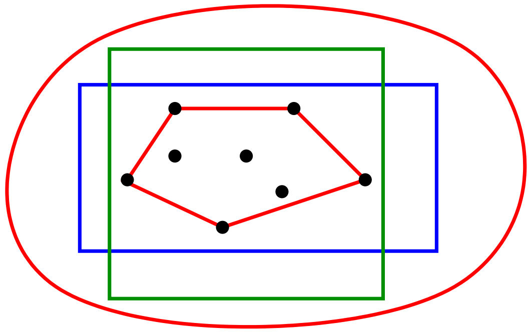
$$2\sum_{i=1}^{n-1} \lfloor \log_2 i\rfloor = \mathcal{O}(n\log n) \text{ (refer to textbook)}$$

- Hence, the time complexity of heap sort is:

$$\mathcal{O}(n) + \mathcal{O}(n\log n) = \mathcal{O}(n\log n)$$

Convex hull problem

Definition:
Given n points in the planes, the convex hull problem is to identify the vertices of the smallest convex polygon in some order (clockwise or counterclockwise).

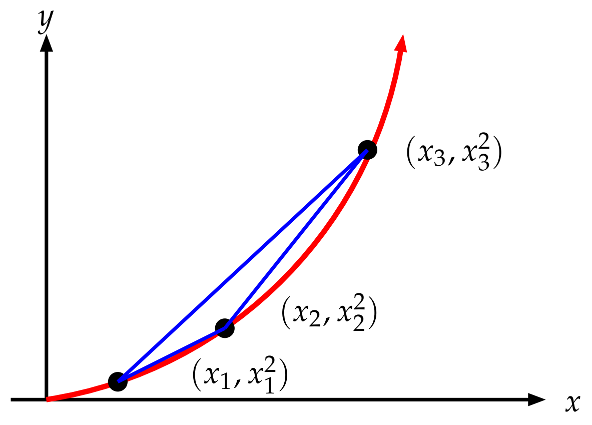


Finding lower bound by problem transformation

- What is the lower bound of the convex hull problem?
- It appears rather difficult to find a meaningful lower bound of the convex hull problem directly.
- However, we can easily obtain a very meaningful lower bound by transforming the sorting problem, whose lower bound is known, to this problem (denoted by sorting problem $\propto$ convex hull problem).

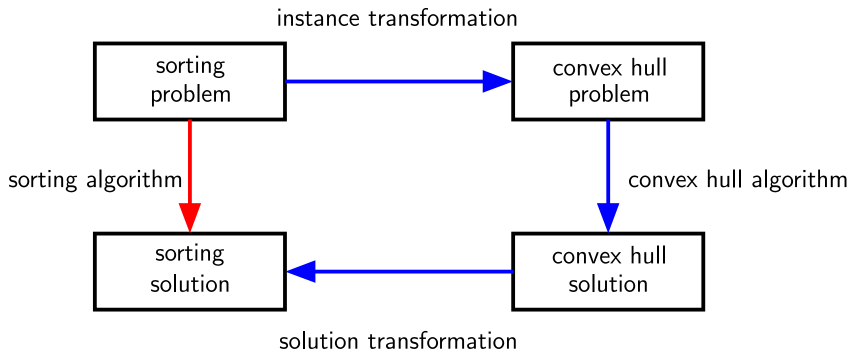
Sorting problem $\propto$ convex hull problem

1. Let $x_1 < x_2 < \cdots < x_n$ be n sorted numbers.
2. Create a 2-dimensional point $(x_i, {x_i}^2)$ for each x_i.



Lower bound of convex hull problem

- By solving the convex hull problem for these newly created points, we can also solve the sorting problem.



Lower bound of convex hull problem (cont'd)

- Let $\Omega(\mathit{sorting}(n))$ be the lower bound of the sorting problem.
- Let $T(\mathit{convex\text{-}hull}(n))$ be the time of an algorithm for solving the convex hull problem.
- Let $\mathcal{O}(\mathit{transform}(n))$ be the cost of problem transformation.
- Then, we have:

$$T(\mathit{convex\text{-}hull}(n)) + \mathcal{O}(\mathit{transform}(n)) \geq \Omega(\mathit{sorting}(n))$$

$$\begin{aligned} T(\mathit{convex\text{-}hull}(n)) &\geq \Omega(\mathit{sorting}(n)) - \mathcal{O}(\mathit{transform}(n)) \\ &= \Omega(n \log n) - \mathcal{O}(n) \\ &= \Omega(n \log n) \end{aligned}$$

- $\Omega(n \log n)$ is a lower bound of the convex hull problem.