Chapter 2

Complexity of an Algorithm and Lower Bounds of Problems

The goodness of an algorithm
- Time complexity (more important)
- Space complexity
- For a parallel algorithm:
  - time-processor product
- For a VLSI circuit:
  - area-time (AT, AT²)

Measure the goodness of an algorithm
- Time complexity of an algorithm
  - efficient (algorithm)
  - worst-case
  - average-case
  - amortized

Measure the difficulty of a problem
- NP-complete?
- Undecidable?
- Is the algorithm best?
  - optimal (algorithm)
Measure the difficulty of a problem

- We can use the number of comparisons or movements to measure a sorting algorithm
- The run time required by an algorithm can be expressed in terms of its input size, \( n \)

Asymptotic notations

- Def: \( f(n) = O(g(n)) \) "at most"
  \[ \exists c, n_0 \ni |f(n)| \leq c|g(n)| \quad \forall \quad n \geq n_0 \]
  e.g. \( f(n) = 3n^2 + 2 = \Omega(n^2) \) or \( \Omega(n) \)

- Def: \( f(n) = \Theta(g(n)) \)
  \[ \exists c_1, c_2, \text{ and } n_0, \ni c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \quad \forall \quad n \geq n_0 \]
  e.g. \( f(n) = 3n^2 + 2 = \Theta(n^2) \)

- Def: \( f(n) \sim o(g(n)) \)
  \[ \lim_{n\to\infty} \frac{f(n)}{g(n)} \to 1 \]
  e.g. \( f(n) = 3n^2 + n = o(3n^2) \)

Problem size

<table>
<thead>
<tr>
<th>( n )</th>
<th>10</th>
<th>10²</th>
<th>10³</th>
<th>10⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>3.3</td>
<td>6.6</td>
<td>10</td>
<td>13.3</td>
</tr>
<tr>
<td>( n )</td>
<td>10</td>
<td>10²</td>
<td>10³</td>
<td>10⁴</td>
</tr>
<tr>
<td>( n\log n )</td>
<td>( 0.33\times10² )</td>
<td>( 0.7\times10³ )</td>
<td>( 10⁴ )</td>
<td>( 1.3\times10⁵ )</td>
</tr>
<tr>
<td>( n² )</td>
<td>( 10² )</td>
<td>( 10⁴ )</td>
<td>( 10⁶ )</td>
<td>( 10⁸ )</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>1024</td>
<td>( 1.3\times10^{10} )</td>
<td>( &gt;10^{100} )</td>
<td>( &gt;10^{100} )</td>
</tr>
<tr>
<td>( n! )</td>
<td>( 3\times10^6 )</td>
<td>( &gt;10^{100} )</td>
<td>( &gt;10^{100} )</td>
<td>( &gt;10^{100} )</td>
</tr>
</tbody>
</table>

Time Complexity Functions
Rate of growth of common computing time functions

Analysis of algorithms
- Best case: easiest
- Worst case
- Average case: hardest

Common computing time functions
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$
  - exponential algorithm: $O(2^n)$
  - polynomial algorithm
- Algorithm A: $O(n^3)$, algorithm B: $O(n)$
  - Should Algorithm B run faster than A?
  - Not necessarily!
  - It is true only when $n$ is large enough!

Straight insertion sort
- input: 7,5,1,4,3
  - 7,5,1,4,3
  - 5,7,1,4,3
  - 1,5,7,4,3
  - 1,4,5,7,3
  - 1,3,4,5,7
**Straight insertion sort**

Algorithm 2.1 Straight Insertion Sort

Input: \( x_1, x_2, \ldots, x_n \)

Output: The sorted sequence of \( x_1, x_2, \ldots, x_n \)

For \( j := 2 \) to \( n \) do

\( i := j - 1 \)
\( x := x_j \)

While \( x < x_i \) and \( i > 0 \) do

\( x_{i+1} := x_i \)
\( i := i - 1 \)

End

\( x_{i+1} := x \)

End

---

**Analysis of # of movements**

- \( M \): # of data movements in straight insertion sort

\[
1 \quad 5 \quad \triangleright \quad 4 \cdot 3
\]

temporary

\( d_3 = 2 \)

\[
M = \sum_{i=1}^{n-1} (2 + d_i)
\]

---

**Inversion table**

- \((a_1, a_2, \ldots, a_n)\): a permutation of \(\{1, 2, \ldots, n\}\)
- \((d_1, d_2, \ldots, d_n)\): the inversion table of \((a_1, a_2, \ldots, a_n)\)
- \(d_i\): the number of elements to the left of \(j\) that are greater than \(j\)

- e.g. permutation \(7 \quad 5 \quad 1 \quad 4 \quad 3 \quad 2 \quad 6\)
  inversion table \(2 \quad 4 \quad 3 \quad 2 \quad 1 \quad 1 \quad 0\)

- e.g. permutation \(7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1\)
  inversion table \(6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0\)

---

**Analysis by inversion table**

- best case: already sorted
  \(d_i = 0\) for \(1 \leq i \leq n\)
  \(\Rightarrow M = 2(n - 1) = O(n)\)

- worst case: reversely sorted
  \(d_1 = n - 1\)
  \(d_2 = n - 2\)
  \(\vdots\)
  \(d_i = n - i\)
  \(d_n = 0\)

\[
M = \sum_{i=1}^{n-1} (2 + d_i) = 2(n - 1) + \frac{n(n - 1)}{2} = O(n^2)
\]
- Average case:
  - $x_j$ is being inserted into the sorted sequence $x_1, x_2, ..., x_{j-1}$.
  - The probability that $x_j$ is the largest: $1/j$.
  - Takes 2 data movements.
  - The probability that $x_j$ is the second largest: $1/j$.
  - Takes 3 data movements.
  - # of movements for inserting $x_j$:
    \[ \sum_{j=1}^{n} \frac{j+3}{2} = \frac{n(n+1)(n+2)}{3} = O(n^2). \]

---

### Analysis of # of exchanges

- **Method 1** (straightforward)
  - $x_j$ is being inserted into the sorted sequence $x_1, x_2, ..., x_{j-1}$.
  - If $x_j$ is the $k$th ($1 \leq k \leq j$) largest, it takes $(k-1)$ exchanges.
  - E.g. 1 5 7 $\leftrightarrow$ 4
    - 1 5 $\leftrightarrow$ 4
    - 1 4 5 7
  - # of exchanges required for $x_j$ to be inserted:
    \[ \frac{0}{j} + \frac{1}{j} + A + \frac{j-1}{2} = \frac{j-1}{2}. \]

---

### Method 2: with inversion table and generating function

$l_n(k)$: # of permutations in $n$ numbers which have exactly $k$ inversions

<table>
<thead>
<tr>
<th>n\k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Assume we have $I_3(k)$, $0 \leq k \leq 3$. We will calculate $I_4(k)$.

1. $a_1 \ a_2 \ a_3 \ a_4$ (largest)
2. $a_1 \ a_2 \ a_3 \ a_4$ (second largest)
3. $a_1 \ a_2 \ a_3 \ a_4$ (third largest)
4. $a_1 \ a_2 \ a_3 \ a_4$ (smallest)

$G_3(Z) = ZG_3(Z)$

$Z^2G_3(Z)$

$Z^3G_3(Z)$

The generating function for $I_n(k)$:

$$G_n(Z) = \sum_{k=0}^{n} I_n(k)Z^k$$

For $n = 4$:

$$G_4(Z) = (1 + 3Z + 5Z^2 + 6Z^3 + 5Z^4 + 3Z^5 + Z^6)$$

In general,

$$G_n(Z) = (1 + Z + Z^2 + \Lambda + Z^{n-1})G_{n-1}(Z)$$

$P_n(k)$: Probability that a given permutation of $n$ numbers has $k$ inversions.

Generating function for $P_n(k)$:

$$g_n(Z) = \sum_{k=0}^{n} P_n(k)Z^k = \sum_{k=0}^{n} \frac{n!}{(n-k)!} Z^k$$

$$= \frac{\Lambda}{Z}G_n(Z)$$

$$= \frac{1+Z+\Lambda+Z\Lambda+Z^2+Z^{n-1}}{Z} \cdot \frac{1+Z+\Lambda+Z^{n-2}}{Z} \cdot \frac{1+Z}{Z} \cdot 1$$

$$\sum_{k=0}^{n} kP_n(k) = g_n'(1)$$

$$= \frac{1+2+\Lambda+(n-1)}{n} \cdot \frac{1+2+\Lambda+(n-2)}{n-1} + \Lambda + \frac{1}{2} + 0$$

$$= \frac{n+1}{2} + \frac{n+2}{2} + \Lambda + \frac{1}{2} + 0$$

$$= \frac{1}{2}n(n-1)$$
Binary search

- sorted sequence: (search 9)

\[
1 \ 4 \ 5 \ 7 \ 9 \ 10 \ 12 \ 15
\]

- step 1
- step 2
- step 3

- best case: 1 step = \(O(1)\)
- worst case: \(\lfloor \log_2 n \rfloor + 1\) steps = \(O(\log n)\)
- average case: \(O(\log n)\) steps

\[
\sum_{i=1}^{k} 2^{i-1} = 2^{k - 1} + 2^{k-2} + \ldots + 2 + 1
\]

Assume \(n = 2^k\)

\[
A(n) = \frac{1}{2n+1} \sum_{i=1}^{k} 2^{i-1} + k(n+1)
\]

\[
= \frac{1}{2n+1} \left( \sum_{i=1}^{k} 2^{i-1} + k(n+1) \right)
\]

\[
\approx k
\]

as \(n\) is very large

\[
= \log n
\]

\[
= O(\log n)
\]

Is the assumption about \(n=2^k\) valid?

\[
\sum_{i=1}^{k} 2^{i-1} = 2^{k - 1} + 2^{k-2} + \ldots + 2 + 1
\]

Straight selection sort

- Only consider # of changes in the flag which is used for selecting the smallest number in each iteration.
- best case: \(O(1)\)
- worst case: \(O(n^2)\)
- average case: \(O(n \log n)\)
Quick sort

Recursively apply the same procedure.

Best case: $O(n \log n)$

A list is split into two sublists with almost equal size.

- $\log n$ runs are needed.
- In each run, $n$ comparisons (ignoring the element used to split) are required.

Worst case: $O(n^2)$

In each run, the number used to split is either the smallest or the largest.

$$n + (n-1) + \Lambda + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Average case: $O(n \log n)$

$$T(n) = \text{Avg} (T(s) + T(n-s)) + cn$$

$$= \frac{1}{n} \sum_{s=1}^{n} (T(s) + T(n-s)) + cn$$

$$= \frac{1}{n} (T(1)+T(n-1)+T(2)+T(n-2)+\ldots+T(n)+T(0)) + cn$$

$$= \frac{1}{n} (2T(1)+2T(2)+\ldots+2T(n-1)+T(n)) + cn$$
\[ T(n) = \frac{1}{n}(2T(1)+2T(2)+ \ldots +2T(n-1)+T(n))+cn \]
\[ nT(n) = 2T(1)+2T(2)+ \ldots +2T(n-1) + cn^2 + \sum (1) \]
\[ (n-1)T(n-1)=2T(1)+2T(2)+ \ldots +2T(n-2)+c(n-1)^2 + \sum (2) \]

(1) - (2)

\[ (n-1)T(n) -(n-2)T(n-1) = 2T(n-1)+c(2n-1) \]
\[ (n-1)T(n) - nT(n-1) = c(2n-1) \]

\[
T(n) = \frac{T(n-1)}{n-1} + \left(\frac{1}{n} + \frac{1}{n-1}\right) + \ldots + \left(\frac{1}{n} + \frac{1}{n-1} \ldots + \frac{1}{2}\right) + c(1+\ldots+1) + T(1), \quad T(1) = 0
\]

\[ \Rightarrow \quad T(n) = 2c \cdot n \cdot H_n - c(n+1) = O(n \log n) \]

2-D ranking finding

- **Def**: Let \( A = (a_1, a_2), B = (b_1, b_2) \). \( A \) dominates \( B \) if \( a_1 > b_1 \) and \( a_2 > b_2 \)

- **Def**: Given a set \( S \) of \( n \) points, the rank of a point \( x \) is the number of points dominated by \( x \).

- **Direct algorithm**:
  - compare all pairs of points: \( O(n^2) \)

- **More efficient algorithm (divide-and-conquer)**

Harmonic number [Knuth 1986]

\[ H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \ln n + \gamma + \frac{1}{2n} + \frac{1}{12n^2} + \ldots - \frac{1}{n^2} \]
\[ \gamma = 0.5772156649 \]
\[ H_n = O(\log n) \]

\[ \frac{T(n)}{n} = c(H_n-1) + cH_{n-1} \]
\[ = c(2H_n-\frac{1}{n}-1) \]
\[ \Rightarrow T(n) = 2c \cdot n \cdot H_n - c(n+1) = O(n \log n) \]
Divide-and-conquer 2-D ranking finding

Step 1: Split the points along the median line L into A and B.

Step 2: Find ranks of points in A and ranks of points in B, recursively.

Step 3: Sort points in A and B according to their y-values. Update the ranks of points in B.

- time complexity: step 1: O(n) (finding median)
  step 3: O(n log n) (sorting)

- total time complexity: Is it possible to improve this part?
  $T(n) \leq 2T(n/2) + c_1 n \log n + c_2 n$
  $\leq 2T(n/2) + c n \log n$
  $\leq 4T(n/4) + c n \log n + c n \log n$
  $\leq nT(1) + c(n \log n + n \log \frac{n}{2} + n \log \frac{n}{4} + \log n + \log 2)$
  $= nT(1) + \frac{cn \log n (\log n + \log 2)}{2}$
  $= O(n \log^2 n)$

Lower bound

- A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.
  □ worst case lower bound
  □ average case lower bound

- The lower bound for a problem is not unique.
  - For example, $\Omega(1)$, $\Omega(n)$, $\Omega(n \log n)$ are all lower bounds for sorting.
  - $\Omega(1)$ and $\Omega(n)$ are trivial bounds.

- At present, if the highest lower bound of a problem is $\Omega(n \log n)$ and the time complexity of the best algorithm is $O(n^2)$.
  - We may try to find a higher lower bound.
  - We may try to find a better algorithm.
  - Both of the lower bound and the algorithm may be improved.

- If the present lower bound is $\Omega(n \log n)$ and there is an algorithm with time complexity $O(n \log n)$, then the algorithm is optimal.
The worst case lower bound of sorting

6 permutations for 3 data elements
\[ a_1 \quad a_2 \quad a_3 \]

In the worst case, any algorithm will take at least 3 comparisons for sorting 3 elements.

2 \quad 1 \quad 3

In the worst case, any algorithm will take at least 7 comparisons for sorting 5 elements.

3 \quad 2 \quad 1

Straight insertion sort:
- input data: (2, 3, 1)
  1. \( a_1 : a_2 \)
  2. \( a_2 : a_3, a_2 \leftrightarrow a_3 \)
  3. \( a_1 : a_2, a_1 \leftrightarrow a_2 \)
- input data: (2, 1, 3)
  1. \( a_1 : a_2, a_1 \leftrightarrow a_2 \)
  2. \( a_2 : a_3 \)

Decision tree for straight insertion sort

Decision tree for bubble sort
Lower bound of sorting

- To find the lower bound, we have to find the longest path from the root of a binary tree.
- \( n! \) distinct permutations
  \( n! \) leaf nodes in the binary decision tree.
- A balanced tree has the minimum longest path:
  \( \lceil \log(n!) \rceil = \Omega(n \log n) \)
  lower bound for sorting: \( \Omega(n \log n) \)

Method 1:

\[
\log(n!) = \log(n(n-1) \cdots 1) \\
= \log 2 + \log 3 + \ldots + \log n \\
> \int_1^n \log x \, dx \\
= \log e \left[ \ln x \right]_1^n \\
= \log e(n \ln n - n + 1) \\
= n \log n - n \log e + 1.44 \\
\geq n \log n - 1.44n \\
= \Omega(n \log n)
\]

Method 2:

- Stirling approximation:
  \( n! = \sqrt{2\pi n} (\frac{n}{e})^n \)
  \( \log n! = \log \sqrt{2\pi} + \frac{1}{2} \log n + n \log \frac{n}{e} = n \log n = \Omega(n \log n) \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n! )</th>
<th>( S_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.922</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.919</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>5.825</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>23.447</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>118.02</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>707.39</td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>3598600</td>
</tr>
<tr>
<td>20</td>
<td>2.433 \times 10^6</td>
<td>2.423 \times 10^6</td>
</tr>
<tr>
<td>100</td>
<td>9.333 \times 10^{157}</td>
<td>9.328 \times 10^{157}</td>
</tr>
</tbody>
</table>

Why insertion sort needs \( O(n^2) \) time?

- Information extracted is not fully utilized
- Consider knockout sort
Knockout Sort

The smallest number is found after \((n-1)\) comparisons.

Heap sort—An optimal sorting algorithm

- **Max Heap**: Each node is no less than its children

For any of the remaining numbers, only
\([\lceil \log n \rceil - 1]\) comparisons are required.

output the maximum and restore:
Phase 1: construction

- input data: 4, 37, 26, 15, 48
- restore the subtree rooted at A(2):

Implementation

- Using a linear array instead of a binary tree.
  - Key: the children of A(i) are A(2i) and A(2i+1).
- Time complexity: $O(n \log n)$

Phase 2: output

Time complexity

Phase 1: construction

- External nodes (leaves) do not require any reconstruction
- Each internal node requires two comparisons with its two children, in the worst case
Time complexity

Phase 1: construction

\[ d = \lfloor \log n \rfloor : \text{depth} \]

# of comparisons is at most:

\[
\sum_{L=0}^{d-1} 2(d-L)2^L = 2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L2^{L-1}
\]

\[
\sum_{L=0}^{d-1} 2^L = 2^d - 1
\]

\[
\sum_{L=0}^{d-1} L2^{L-1} = 2^d(k-1) + 1
\]

\[
= 2d(2^d - 1) - 4(2^d(d - 1) + 1)
\]

\[
= cn - 2\lfloor \log n \rfloor - 4, \quad 2 \leq c \leq 4
\]

Average case lower bound of sorting

- By binary decision tree
- The average time complexity of a sorting algorithm can be explored using the external path length of the binary tree
- The external path length is minimized if the tree is balanced
  (all leaf nodes on level \( d \) or level \( d-1 \))

Time complexity

Phase 2: output

\[
2 \sum_{i=1}^{n-1} \lfloor \log i \rfloor = 2n\lfloor \log n \rfloor - 4cn + 4, \quad 2 \leq c \leq 4
\]

\[
= O(n \log n)
\]
Compute the min external path length

1. Depth of balanced binary tree with \( c \) leaf nodes:
   \[ d = \lceil \log c \rceil \]
   Leaf nodes can appear only on level \( d \) or \( d-1 \).

2. \( x_1 \) leaf nodes on level \( d-1 \)
   \( x_2 \) leaf nodes on level \( d \)
   \[ x_1 + x_2 = c \]
   \[ x_1 + \frac{x_2}{2} = 2^{d-1} \]
   \[ \Rightarrow x_1 = 2^d - c \]
   \[ x_2 = 2c - 2^d \]

3. External path length:
   \[ M = x_1(d-1) + x_2d \]
   \[ = (2^d - 1)(d-1) + 2c - 2^d \]
   \[ = c(d-1) + 2c - 2^d, \quad d = \lceil \log c \rceil \]
   \[ = c \lceil \log c \rceil + 2(c - 2^d) \]

4. \( c = n! \)
   \[ M = n! \lceil \log n \rceil + 2(n! - 2\log n) \]
   \[ MN! = \lceil \log n \rceil + 2 \]
   \[ = \lceil \log n \rceil + c, \quad 0 \leq c \leq 1 \]
   \[ = \Omega(n \log n) \]
   Average case lower bound of sorting: \( \Omega(n \log n) \)

Quicksort & Heapsort

- Quicksort is optimal in the average case.
  \( O(n \log n) \)
- (1) Worst case time complexity of heap sort is \( O(n \log n) \)
- (2) Average case lower bound: \( \Omega(n \log n) \)
  - average case time complexity of heapsort is \( O(n \log n) \)
  - Heapsort is optimal in the average case

Improving a lower bound through oracles

- Problem \( P \): merging two sorted sequences \( A \) and \( B \) with lengths \( m \) and \( n \).
  Binary decision tree:
  There are \( \binom{m+n}{n} \) ways !
  \[ \binom{m+n}{n} \] leaf nodes in the binary tree.
  \[ \Rightarrow \] The lower bound for merging:
  \[ \lceil \log \binom{m+n}{n} \rceil \leq m + n - 1 \] (conventional merging)
When $m = n$

$$\log \binom{m+n}{n} = \log \frac{(2m)!}{(m!)^2} = \log((2m)!) - 2\log m!$$

- Using Stirling approximation
  $$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

$$\log \binom{m+n}{n} \approx 2m - \frac{1}{2} \log m + O(1)$$

- Optimal algorithm: $2m - 1$ comparisons

$$\log \binom{m+n}{n} < 2m - 1$$

---

**Oracle:**

- The oracle tries its best to cause the algorithm to work as hard as it might. (To give a very hard data set)

- Sorted sequences:
  - A: $a_1 < a_2 < \ldots < a_m$
  - B: $b_1 < b_2 < \ldots < b_m$

- The very hard case:
  - $a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$

We must compare:

$$a_1 : b_1$$
$$b_2 : a_2$$
$$a_2 : b_2$$
$$\vdots$$
$$b_m : a_m$$

$$a_m : b_m$$

- Otherwise, we may get a wrong result for some input data. e.g. If $b_1$ and $a_2$ are not compared, we can not distinguish

$$a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$$
$$a_1 < a_2 < b_1 < b_2 < \ldots < a_m < b_m$$

- Thus, at least $2m-1$ comparisons are required.

- The conventional merging algorithm is optimal for $m = n$.

---

**Finding lower bound by problem transformation**

- Problem $A$ reduces to problem $B$ ($A \propto B$)
  - iff $A$ can be solved by using any algorithm which solves $B$.
  - If $A \propto B$, $B$ is more difficult.

<table>
<thead>
<tr>
<th>instance of $A$</th>
<th>transformation $T(tr_1)$</th>
<th>instance of $B$</th>
<th>transformation $T(tr_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(A)$</td>
<td>$T(tr_1)$</td>
<td>$T(B)$</td>
<td>solver of $B$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>answer of $A$</th>
<th>transformation $T(tr_1)$</th>
<th>answer of $B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(tr_2)$</td>
<td>$T(tr_2)$</td>
<td>$T(B)$</td>
</tr>
</tbody>
</table>

- Note: $T(tr_1) + T(tr_2) < T(B)$

$$T(A) \leq T(tr_1) + T(tr_2) + T(B) \sim O(T(B))$$
The lower bound of the convex hull problem

- sorting $\propto$ convex hull

\[ \begin{array}{c}
A \\
\text{an instance of } A: (x_1, x_2, \ldots, x_n) \\
\text{transformation}
\end{array} \]

\[ \begin{array}{c}
B \\
\text{an instance of } B: \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}
\end{array} \]

assume: $x_1 < x_2 < \ldots < x_n$

If the convex hull problem can be solved, we can also solve the sorting problem.
- The lower bound of sorting: $\Omega(n \log n)$
- The lower bound of the convex hull problem: $\Omega(n \log n)$

The lower bound of the Euclidean minimal spanning tree (MST) problem

- sorting $\propto$ Euclidean MST

\[ \begin{array}{c}
A \\
\text{an instance of } A: (x_1, x_2, \ldots, x_n) \\
\text{transformation}
\end{array} \]

\[ \begin{array}{c}
B \\
\text{an instance of } B: \{(x_1, 0), (x_2, 0), \ldots, (x_n, 0)\}
\end{array} \]

assume $x_1 < x_2 < x_3 < \ldots < x_n$
- $\iff$ there is an edge between $(x_i, 0)$ and $(x_{i+1}, 0)$ in the MST, where $1 \leq i \leq n-1$

If the Euclidean MST problem can be solved, we can also solve the sorting problem.
- The lower bound of sorting: $\Omega(n \log n)$
- The lower bound of the Euclidean MST problem: $\Omega(n \log n)$