1. Introduction, asymptotics and growth functions

1.1 The Role of Algorithms in Computing

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

1.1.1 Analysis of algorithms

The major concern of analysis of algorithm is the theoretical study of computer-program performance and resource usage.

What’s more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability

1.1.2 Runtime analysis

The running time depends on the input: an already sorted sequence is easier to sort.

Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

Generally, we seek upper bounds on the running time, because everybody likes a guarantee.
1.2 Asymptotic notation

In pure mathematics and applications, particularly the analysis of algorithms, real analysis, and engineering, \textbf{asymptotic analysis} is a method of describing limiting behavior. Asymptotic analysis refers to solving problems approximately up to such equivalences. For example, given complex-valued functions \( f \) and \( g \) of a natural number variable \( n \), one writes

\[
f \sim g \quad \text{(as } n \to \infty)\]

to express the fact that

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1
\]

and \( f \) and \( g \) are called \textbf{asymptotically equivalent} as \( n \to \infty \).

\textbf{1.2.1 \( \Theta \)-notation}

For a given function \( g(n) \), we denote by \( \Theta(g(n)) \) the set of functions

\[
\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}.
\]

A function \( f(n) \) belongs to the set \( \Theta(g(n)) \) if there exist positive constants \( c_1 \) and \( c_2 \) such that it can be "sandwiched" between \( c_1 g(n) \) and \( c_2 g(n) \), for sufficiently large \( n \). Because \( \Theta(g(n)) \) is a set, we could write "\( f(n) \in \Theta(g(n)) \)" to indicate that \( f(n) \) is a member of \( \Theta(g(n)) \).

\textbf{1.2.2 \( O \)-notation}

The \( \Theta \)-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use \( O \)-notation. For a given function \( g(n) \), we denote by \( O(g(n)) \) the set of functions

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

We use \( O \)-notation to give an upper bound on a function, to within a constant factor.

\textbf{1.2.3 \( \Omega \)-notation}

Just as \( O \)-notation provides an asymptotic upper bound on a function, \( \Omega \)-notation provides an asymptotic lower bound. For a given function \( g(n) \), we denote by \( \Omega(g(n)) \) the set of functions

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

\textbf{1.2.4 Tight bound, lower bound and upper bound}

For any two functions \( f(n) \) and \( g(n) \), we have \( f(n) = \Theta(g(n)) \) if and only if \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).
Figure 1.1: Graphic examples of the $\Theta$, $O$, and $\Omega$ notations. In each part, the value of $n_0$ shown is the minimum possible value; any greater value would also work. (a) $\Theta$-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants $n_0$, $c_1$, and $c_2$ such that to the right of $n_0$, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. (b) $O$-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or below $c g(n)$. (c) $\Omega$-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or above $c g(n)$.

1.3 Growth of function

Order of growth is an abstraction to ease analysis and focus on the important features. Look only at the leading term of the formula for running time.

- Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

We consider the following functions types.

- Constant function $f(x) = 1$
- Logarithm function $f(x) = \log(x)$
- Linear function $f(x) = x$
- Quadratic function $f(x) = x^2$
- Exponential function $f(x) = e^x$ or $2^x$
2. Sorting

2.1 Bubble sort

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort.

2.1.1 Pseudocode

bubbleSort( A : list of sortable items )
for each i in 1 to length(A) do:
    for each j in length(A) downto i + 1 do:
            swap( A[ j - 1], A[ j ])
        end if
    end for
end for

2.1.2 Analysis

\[ N_{\text{comparison}} = \sum_{i=2}^{n} (i-1) = \frac{n^2 - n}{2} \]
\[ N_{\text{assignment}} = \left(\frac{1}{2}\right) \times 2 \times N_{\text{comparison}} = \frac{n^2 - n}{2} \]
\[ N_{\text{read}} = 2 \times N_{\text{comparison}} = n^2 - n \]

Bubble sort has best-case complexity \( \Omega(n) \) and worst case \( O(n^2) \).

2.1.3 Example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in bold are being compared.

First Pass:
( 5 1 4 2 8 ) \to ( 1 5 4 2 8 ) Here, algorithm compares the first two elements, and swaps them.
( 1 5 4 2 8 ) \to ( 1 4 5 2 8 )
( 1 4 5 2 8 ) \to ( 1 4 2 5 8 )
( 1 4 2 5 8 ) \to ( 1 4 2 5 8 ) Now, since these elements are already in order, algorithm does not swap them.

Second Pass:
( 1 4 2 5 8 ) \to ( 1 4 2 5 8 )
( 1 4 2 5 8 ) \to ( 1 2 4 5 8 )
( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )
( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. Algorithm
needs one whole pass without any swap to know it is sorted.

Third Pass:
(1 2 4 5 8) \to (1 2 4 5 8)
(1 2 4 5 8) \to (1 2 4 5 8)
(1 2 4 5 8) \to (1 2 4 5 8)
(1 2 4 5 8) \to (1 2 4 5 8)
Finally, the array is sorted, and the algorithm can terminate.

2.2 Insertion sort

Insertion sort is a good algorithm for sorting a small number of elements. It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

2.2.1 Pseudocode

\[ \text{INSERTION-SORT} (A) \]
\[
\text{for } j \leftarrow 2 \text{ to } n \quad \text{cost times}
\]
\[
do \quad \text{key} \leftarrow A[j] \quad c_1 \quad n \\
\quad \text{Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j - 1]. \quad c_2 \quad n - 1 \\
\quad i \leftarrow j - 1 \\
\text{while } i > 0 \text{ and } A[i] > \text{key} \quad c_3 \quad \sum_{j=2}^{n} t_j \\
\quad \text{do } A[i + 1] \leftarrow A[i] \quad c_4 \quad n - 1 \\
\quad i \leftarrow i - 1 \\
\quad A[i + 1] \leftarrow \text{key} \\
\]

2.2.2 Analysis

Average case

\[
N_{\text{comparison}} = \sum_{j=2}^{n} \frac{j - 1}{2} = \frac{n^2 - n}{4}
\]

\[
N_{\text{assignment}} = 2 \times N_{\text{comparison}}
\]

Best case (when already sorted)

\[
N_{\text{comparison}} = \sum_{j=2}^{n} 1 = n - 1
\]

\[
N_{\text{assignment}} = 2 \times N_{\text{comparison}} = 2(n - 1)
\]
2.2.3 Example

[Read this figure row by row. Each part shows what happens for a particular iteration with the value of $j$ indicated. $j$ indexes the “current card” being inserted into the hand. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position. The heavy vertical lines separate the part of the array in which an iteration works—$A[1..j]$—from the part of the array that is unaffected by this iteration—$A[j+1..n]$. The last part of the figure shows the final sorted array.]

2.3 Selection sort

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $\Theta(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

2.2.1 Pseudocode

\begin{verbatim}
for i = n downto 2
    max = A[1]
    loc = 1
    for j=1 to i
        if(max = A[j])
            max = A[j]
        end if
    end for
    A[i]=max
end for
\end{verbatim}

2.2.2 Analysis

$$N_{\text{comparison}} = \sum_{i=2}^{n} i - 1 = \frac{n^2 - n}{2}$$

$$N_{\text{memory read}} = N_{\text{comparison}} = \frac{n^2 - n}{2}$$

$$N_{\text{memory write}} = (n-1) \times 2$$
2.2.3 Example
31 25 12 22 11
11 25 12 22 31
11 12 25 22 31
11 12 22 25 31

2.2.4 Comparison of the simple sort algorithms

<table>
<thead>
<tr>
<th></th>
<th>Bubble</th>
<th>Selection</th>
<th>Insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{\text{comparison}}$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{4} n^2$</td>
</tr>
<tr>
<td>$N_{\text{read}}$</td>
<td>$2 \times \frac{1}{2} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{4} n^2$</td>
</tr>
<tr>
<td>$N_{\text{write}}$</td>
<td>$2 \times \frac{1}{2} n^2$</td>
<td>$2n$</td>
<td>$2 \times \frac{1}{4} n^2$</td>
</tr>
</tbody>
</table>

Bubble sort is always the worst. Preferable when $n$ is not big.

2.4 Shell sort

Shell sort is a sorting algorithm that is a generalization of insertion sort, with two observations:
- insertion sort is efficient if the input is "almost sorted", and
- insertion sort is typically inefficient because it moves values just one position at a time.

2.4.1 Code in C/C++

```c
void shell_sort(int A[], int size)
{
    int i, j, increment, temp;
    increment = size / 2;

    while (increment > 0)
    {
        for (i = increment; i < size; i++)
        {
            j = i;
            temp = A[i];
            while ((j >= increment) && (A[j-increment] > temp))
            {
                j = j - increment;
            }
            A[j] = temp;
        }

        if (increment == 2)
        {
            increment = 1;
        } else
        {
            increment = (int) (increment / 2.2);
        }
    }
}
```

2.4.2 Analysis

The complexity is $O(n^{1.25}) \sim O(1.6n^{1.25})$. 
2.4.3 Example

Example: Let \(37905168420615734982\) be the data sequence to be sorted. First, it is arranged in an array with 7 columns (left), then the columns are sorted (right):

\[
\begin{array}{ccccccc}
3 & 7 & 9 & 0 & 5 & 1 & 6 \\
3 & 2 & 0 & 5 & 1 & 5 \\
8 & 4 & 2 & 0 & 6 & 1 & 5 \\
7 & 3 & 4 & 9 & 8 & 2 \\
\end{array} \quad \begin{array}{ccccccc}
3 & 2 & 0 & 5 & 1 & 6 \\
3 & 2 & 0 & 5 & 1 & 5 \\
7 & 4 & 4 & 6 & 5 & 1 & 6 \\
8 & 7 & 9 & 8 & 2 \\
\end{array}
\]

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is also still there. In the next step, the sequence is arranged in 3 columns, which are again sorted:

\[
\begin{array}{cccc}
3 & 3 & 2 & 0 & 0 & 1 \\
0 & 5 & 1 & 1 & 2 & 2 \\
5 & 7 & 4 & 3 & 3 & 4 \\
4 & 0 & 6 & 4 & 5 & 5 \\
1 & 6 & 8 & 5 & 6 & 8 \\
7 & 9 & 9 & 7 & 7 & 9 \\
0 & 2 & 0 & 9 \\
\end{array}
\]

Now the sequence is almost completely sorted. When arranging it in one column in the last step, it is only a 8, an 8 and a 9 that have to move a little bit to their correct position.

2.5 Merge sort

2.5.1 The divide-and-conquer approach

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- **Divide**: Divide the problem into a number of sub-problems.
- **Conquer**: Conquer the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.
- **Combine**: Combine the solutions to the sub-problems into the solution for the original problem.

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide**: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer**: Sort the two subsequences recursively using merge sort.
- **Combine**: Merge the two sorted subsequences to produce the sorted answer.

2.5.2 Pseudocode

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure \(\text{MERGE}(A, p, q, r)\), where \(A\) is an array and \(p, q,\) and \(r\) are indices numbering elements of the array such that \(p \leq q < r\). The procedure assumes that the subarrays \(A[p \ldots q]\) and \(A[q + 1 \ldots r]\) are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray \(A[p \ldots r]\).

Our \(\text{MERGE}\) procedure takes time \(\Theta(n)\), where \(n = r - p + 1\) is the number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step
consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are checking just two top cards. Since we perform at most \( n \) basic steps, merging takes \( \Theta(n) \) time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. The idea is to put on the bottom of each pile a sentinel card, which contains a special value that we use to simplify our code. Here, we use \( \infty \) as the sentinel value, so that whenever a card with \( \infty \) is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly \( r - p + 1 \) cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

\[
\text{MERGE}(A, p, q, r)
\]

\[
n_1 \leftarrow q - p + 1
\]

\[
n_2 \leftarrow r - q
\]

create arrays \( L[1..n_1 + 1] \) and \( R[1..n_2 + 1] \)

\[
\text{for } i \leftarrow 1 \text{ to } n_1
\]

\[
\text{do } L[i] \leftarrow A[p + i - 1]
\]

\[
\text{for } j \leftarrow 1 \text{ to } n_2
\]

\[
\text{do } R[j] \leftarrow A[q + j]
\]

\[
L[n_1 + 1] \leftarrow \infty
\]

\[
R[n_2 + 1] \leftarrow \infty
\]

\[
i \leftarrow 1
\]

\[
j \leftarrow 1
\]

\[
\text{for } k \leftarrow p \text{ to } r
\]

\[
\text{do if } L[i] \leq R[j]
\]

\[
\text{then } A[k] \leftarrow L[i]
\]

\[
i \leftarrow i + 1
\]

\[
\text{else } A[k] \leftarrow R[j]
\]

\[
j \leftarrow j + 1
\]

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure \text{MERGE-SORT}(A, p, r) sorts the elements in the subarray \( A[p \cdots r] \). If \( p \geq r \), the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index \( q \) that partitions \( A[p \cdots r] \) into two subarrays: \( A[p \cdots q] \), containing \( \lceil n/2 \rceil \) elements, and \( A[q + 1 \cdots r] \), containing \( \lfloor n/2 \rfloor \) elements.\[7\]
MERGE-SORT($A$, $p$, $r$)

if $p < r$

then $q \leftarrow \lfloor (p + r)/2 \rfloor$

MERGE-SORT($A$, $p$, $q$)

MERGE-SORT($A$, $q + 1$, $r$)

MERGE($A$, $p$, $q$, $r$)

\[ T(1) = c \]

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]

\[ = 2\left[ 2T\left(\frac{n}{4}\right) + c\frac{n}{2} \right] + cn \]

\[ = 2^2T\left(\frac{n}{2^2}\right) + (1+1)cn \]

\[ = 2^2\left[ 2T\left(\frac{n}{2^3}\right) + \frac{cn}{2^2} \right] + (1+1)cn \]

\[ = 2^3T\left(\frac{n}{2^3}\right) + (1+1+1)cn \]

\[ = 2^kT\left(\frac{n}{2^k}\right) + kcn \]

because $2^k = n$, $k = \log_2 n$

\[ = nT(1) + (\log_2 n)cn \]

\[ = cn\log_2 n + cn \]

2.5.4 Example

Figure 2.1: The operation of lines 10-17 in the call MERGE($A$, 9, 12, 16), when the subarray $A[9 \ldots 16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array $L$ contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array $R$ contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions in
$A$ contain their final values, and lightly shaded positions in $L$ and $R$ contain values that have yet to be copied back into $A$. Taken together, the lightly shaded positions always comprise the values originally in $A[9 \cdots 16]$, along with the two sentinels. Heavily shaded positions in $A$ contain values that will be copied over, and heavily shaded positions in $L$ and $R$ contain values that have already been copied back into $A$. (a)-(h) The arrays $A$, $L$, and $R$, and their respective indices $k$, $i$, and $j$ prior to each iteration of the loop of lines 12-17. (i) The arrays and indices at termination. At this point, the subarray in $A[9 \cdots 16]$ is sorted, and the two sentinels in $L$ and $R$ are the only two elements in these arrays that have not been copied into $A$.

Figure 2.2: The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.
2.6 Heap sort.

Heap is a useful data structure. A heap is a specialized tree-based data structure that satisfies the heap property: if B is a child node of A, then key(A) ≥ key(B).

1 Priority Queue.
A bunch of tasks, each with a numerical “Key” (priority). Do task of highest priority.

2 Computer memory allocation.
Allocate memory on the heap using binary search.

In a heap, the relationship of parent and child can be calculated as following,

\[
Parent(i) = \left\lfloor \frac{i}{2} \right\rfloor
\]

\[
LeftChild(i) = 2i
\]

\[
RightChild(i) = 2i + 1
\]

A max heap: \( A[parent(i)] \geq A[i] \).

There are several useful operations on a heap:
- Delete_max(S)
- Insert(S,x)
Find_max(s)
Search(S,x)

1 Delete:

2 Insert:
3 Build a heap

\[
for \ i = \frac{\text{Length}[A]}{2} \text{ down to } 1
\]

\[
do \text{ max_heapify}(A, i)
\]

end for
Key Algorithm
Max_heapify(A,i): Assume left_child(i), right_child(i) are proper heap property.

\[
\text{max_heapify}(A, i) \\
1 \quad l \leftarrow \text{left} \ (i) \\
2 \quad r \leftarrow \text{right} \ (i) \\
3 \quad \text{if } l \leq \text{heap_size} [A] \quad \text{and} \quad A[l] > A[i] \\
4 \quad \text{then } l \text{arg est} \leftarrow l \\
5 \quad \text{else } l \text{arg est} \leftarrow i \\
\text{endif} \\
6 \quad \text{if } r \leq \text{heap_size} [A] \quad \text{and} \quad A[r] > A[l \text{arg est}] \\
7 \quad l \text{arg est} \leftarrow r \\
\text{endif} \\
8 \quad \text{if } l \text{arg est} \neq i \\
9 \quad \text{exchange} \quad A[i] \leftrightarrow A[l \text{arg est}] \\
10 \quad \text{max_heapify} \ (A, l \text{arg est}) \\
\text{endif}
\]

4: largest = 4
8: largest != i
   Max_heapify(A,4)

Heapsort:
heap_sort(A)
1  Build _max_ heap (A)
2  for i = length [A] downto 2
4  heap_size [A] ← heap_size [A] − 1
5  max_heapify (A,1)
endfor
Runtime analysis

Max_heapify

\[ T(n) \leq T\left(\frac{2n}{3}\right) + \Xi(1) \]
\[ T(n) = O(\log n) \]

Build_heap

\[ \sum_{j=1}^{\log n} 2^j \log n \cdot \frac{j}{2^j} \]
\[ = \sum_{j=1}^{\log n} \frac{j}{2^j} \]
\[ = O(n) \]

HeapSort:

\[ O(n \log n) \]

Call to max_heapify
2.7 Quick Sort

1 Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that, on average, makes $\Theta(n \log n)$ (big O notation) comparisons to sort $n$ items. However, in the worst case, it makes $\Theta(n^2)$ comparisons. Typically, quicksort is significantly faster in practice than other $\Theta(n \log n)$ algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the possibility of requiring quadratic time. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort.

2 The algorithm

```
Partition (A, p, r)
1  x = A[r]
2  i = p - 1
3  for i = p to r - 1
4    if A[j] ≤ x
5      i = i + 1
7  end if
8  end for
10 return i + 1
```

Lines 4–7: 1 comparison + 1 exchange
Total for partition(A,p,r)

$(r-p+1+1)$ comparison

$\frac{1}{2}(r0p)$ exchange.
Partition is the key trick.

Partition example:

1. Initial array: 2 8 7 1 3 5 6 4
2. Best case partition into equal size array:
   - i = 2
   - j = 4
   - Array: 2 1 7 3 8 5 6 4
   - Array: 2 1 3 7 8 5 6 4
   - Array: 2 1 3 4 7 5 6 8

Quick sort Runtime Analysis

1. Worse case
   For an already sorted array:

   \[
   T(n) = T(n-1) + T(0) + \Theta(n)
   \]
   \[
   = T(n-1) + n
   \]
   \[
   = T(n-2) + 2n
   \]
   \[
   \vdots
   \]
   \[
   = T(n-k) + kn
   \]
   \[
   = T(1) + (n-1)n
   \]
   \[
   = \Theta(n^2)
   \]

2. Best case partition into equal size array:
3 Balance partition

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)
\]
\[
= 2T\left(\frac{n}{2}\right) + cn
\]
\[
= n \log n
\]

Sorting worst case lower bound

Suppose we have \((a_1, \ldots, a_n)\) in the sorting array.

\[
(a_1, \ldots, a_n)
\]

All possible permutations of \((a_1, \ldots, a_n)\) is

\[n! \leq 2^h\]

\(h\) is the height of the tree = # of comparison

\[h \geq \log_2 n \geq n(\log_2 n - \log_2 e)\]

Merge sort (worst case)

\[T(n) = n\log_2 n - 1.11583n + 1\]
Quick sort:

Solving \[ T(n) = \begin{cases} 
  c & n = 1 \\
  2T\left(\frac{n}{2}\right) + cn & n \geq 2 
\end{cases} \]

\[ T(n) = 1.39n\log_2 n + O(n) \]