Divide and conquer, Quick sort V.S. Merge sort

Liu Xinyu [*]

April 9, 2013

1 Introduction

It's proved that the best approximate performance of comparison based sorting is $O(N \lg N)$ [1]. In this chapter, two divide and conquer sorting algorithms are introduced. Both of them perform in $O(N \lg N)$ time. One is quick sort. It is the most popular sorting algorithm. Quick sort has been well studied, many programming libraries provide sorting tools based on quick sort.

In this chapter, we'll first introduce the idea of quick sort, which demonstrates the power of divide and conquer strategy well. Several variants will be explained, and we'll see when quick sort performs poor in some special cases. That the algorithm is not able to partition the sequence in balance.

In order to solve the unbalanced partition problem, we'll next introduce about merge sort, which ensure the sequence to be well partitioned in all the cases. Some variants of merge sort, including nature merge sort, bottom-up merge sort are shown as well.

Same as other chapters, all the algorithm will be realized in both imperative and functional approaches.

2 Quick sort

Consider a teacher arranges a group of kids in kindergarten to stand in a line for some game. The kids need stand in order of their heights, that the shortest one stands on the left most, while the tallest stands on the right most. How can the teacher instruct these kids, so that they can stand in a line by themselves?

There are many strategies, and the quick sort approach can be applied here:

1. The first kid raises his/her hand. The kids who are shorter than him/her stands to the left to this child; the kids who are taller than him/her stands to the right of this child;

[*] **Liu Xinyu**
Email: liuxinyu95@gmail.com

Figure 1: Instruct kids to stand in a line

2. All the kids move to the left, if there are, repeat the above step; all the kids move to the right repeat the same step as well.

Suppose a group of kids with their heights as $\{102, 100, 98, 95, 96, 99, 101, 97\}$ with [cm] as the unit. The following table illustrate how they stand in order of height by following this method.

102	100	98	95	96	99	101	97
100	98	95	96	99	101	97	*102*
98	95	96	99	97	*100*	101	*102*
95	96	97	*98*	99	*100*	*101*	*102*
95	**96**	97	*98*	*99*	*100*	*101*	*102*
95	*96*	97	*98*	*99*	*100*	*101*	*102*
95	*96*	*97*	*98*	*99*	*100*	*101*	*102*

At the beginning, the first child with height 102 cm raises his/her hand. We call this kid the pivot and mark this height in bold. It happens that this is the tallest kid. So all others stands to the left side, which is represented in the second row in the above table. Note that the child with height 102 cm is in the final ordered position, thus we mark it italic. Next the kid with height 100 cm raise hand, so the children of heights 98, 95, 96 and 99 cm stand to his/her left, and there is only 1 child of height 101 cm who is taller than this pivot kid. So he stands to the right hand. The 3rd row in the table shows this stage accordingly. After that, the child of 98 cm high is selected as pivot on left hand; while the child of 101 cm high on the right is selected as pivot. Since there are no other kids in the unsorted group with 101 cm as pivot, this small group is ordered already and the kid of height 101 cm is in the final proper position. The same method is applied to the group of kids which haven't been in correct order until all of them are stands in the final position.

2.1 Basic version

Summarize the above instruction leads to the recursive description of quick sort. In order to sort a sequence of elements L.

- If L is empty, the result is obviously empty; This is the trivial edge case;
- Otherwise, select an arbitrary element in L as a pivot, recursively sort all elements not greater than L, put the result on the left hand of the pivot, *and* recursively sort all elements which are greater than L, put the result on the right hand of the pivot.

Note that the emphasized word *and*, we don't use 'then' here, which indicates it's quite OK that the recursive sort on the left and right can be done in parallel. We'll return this parallelism topic soon.

Quick sort was first developed by C. A. R. Hoare in 1960 [1], [15]. What we describe here is a basic version. Note that it doesn't state how to select the pivot. We'll see soon that the pivot selection affects the performance of quick sort dramatically.

The most simple method to select the pivot is always choose the first one so that quick sort can be formalized as the following.

$$sort(L) = \begin{cases} \Phi & : \quad L = \Phi \\ sort(\{x|x \in L', x \leq l_1\} \cup \{l_1\} \cup sort(\{x|x \in L', l_1 < x\}) & : \quad otherwise \end{cases} \tag{1}$$

Where l_1 is the first element of the non-empty list L, and L' contains the rest elements $\{l_2, l_3, ...\}$. Note that we use Zermelo Frankel expression (ZF expression for short), which is also known as list comprehension. A ZF expression $\{a|a \in S, p_1(a), p_2(a), ...\}$ means taking all element in set S, if it satisfies all the predication $p_1, p_2, ...$. The result is also a list. Please refer to the appendix about list in this book for detail.

It's quite straightforward to translate this equation to real code if list comprehension is supported. The following Haskell code is given for example:

```
sort [] = []
sort (x:xs) = sort [y | y←xs, y ≤ x] ++ [x] ++ sort [y | y←xs, x < y]
```

This might be the shortest quick sort program in the world at the time when this book is written. Even a verbose version is still very expressive:

```
sort [] = []
sort (x:xs) = as ++ [x] ++ bs where
    as = sort [ a | a ← xs, a ≤ x]
    bs = sort [ b | b ← xs, x < b]
```

There are some variants of this basic quick sort program, such as using explicit filtering instead of list comprehension. The following Python program demonstrates this for example:

```
def sort(xs):
    if xs == []:
        return []
    pivot = xs[0]
    as = sort(filter(lambda x : x ≤ pivot, xs[1:]))
    bs = sort(filter(lambda x : pivot < x, xs[1:]))
    return as + [pivot] + bs
```

2.2 Strict weak ordering

We assume the elements are sorted in monotonic none decreasing order so far. It's quite possible to customize the algorithm, so that it can sort the elements in other ordering criteria. This is necessary in practice because users may sort numbers, strings, or other complex objects (even list of lists for example).

The typical generic solution is to abstract the comparison as a parameter as we mentioned in chapters about insertion sort and selection sort. Although it needn't the total ordering, the comparison must satisfy *strict weak ordering* at least [17] [16].

For the sake of brevity, we only considering sort the elements by using less than or equal (equivalent to not greater than) in the rest of the chapter.

2.3 Partition

Observing that the basic version actually takes two passes to find all elements which are greater than the pivot as well as to find the others which are not respectively. Such partition can be accomplished by only one pass. We explicitly define the partition as below.

$$partition(p, L) = \begin{cases} (\Phi, \Phi) & : L = \Phi \\ (\{l_1\} \cup A, B) & : p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : \neg p(l_1) \end{cases} \tag{2}$$

Note that the operation $\{x\} \cup L$ is just a 'cons' operation, which only takes constant time. The quick sort can be modified accordingly.

$$sort(L) = \begin{cases} \Phi & : L = \Phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : otherwise, (A, B) = partition(\lambda_x x \leq l_1, L') \end{cases} \tag{3}$$

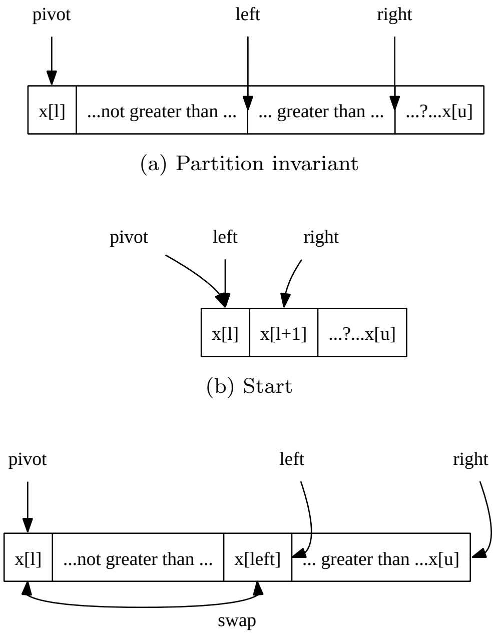
Translating this new algorithm into Haskell yields the below code.

```
sort [] = []
sort (x:xs) = sort as ++ [x] ++ sort bs where
    (as, bs) = partition (≤ x) xs

partition _ [] = ([], [])
partition p (x:xs) = let (as, bs) = partition p xs in
    if p x then (x:as, bs) else (as, x:bs)
```

The concept of partition is very critical to quick sort. Partition is also very important to many other sort algorithms. We'll explain how it generally affects the sorting methodology by the end of this chapter. Before further discussion about fine tuning of quick sort specific partition, let's see how to realize it in-place imperatively.

There are many partition methods. The one given by Nico Lomuto [4] [2] will be used here as it's easy to understand. We'll show other partition algorithms soon and see how partitioning affects the performance.



(a) Partition invariant

(b) Start

(c) Finish

Figure 2: Partition a range of array by using the left most element as pivot.

Figure 2 shows the idea of this one-pass partition method. The array is processed from left to right. At any time, the array consists of the following parts as shown in figure 2 (a):

- The left most cell contains the pivot; By the end of the partition process, the pivot will be moved to the final proper position;
- A segment contains all elements which are not greater than the pivot. The right boundary of this segment is marked as 'left';
- A segment contains all elements which are greater than the pivot. The right boundary of this segment is marked as 'right'; It means that elements between 'left' and 'right' marks are greater than the pivot;
- The rest of elements after 'right' mark haven't been processed yet. They may be greater than the pivot or not.

At the beginning of partition, the 'left' mark points to the pivot and the 'right' mark points to the the second element next to the pivot in the array as in Figure 2 (b); Then the algorithm repeatedly advances the right mark one element after the other till passes the end of the array.

In every iteration, the element pointed by the 'right' mark is compared with the pivot. If it is greater than the pivot, it should be among the segment between the 'left' and 'right' marks, so that the algorithm goes on to advance the 'right' mark and examine the next element; Otherwise, since the element pointed by 'right' mark is less than or equal to the pivot (not greater than), it should be put before the 'left' mark. In order to achieve this, the 'left' mark needs be advanced by one, then exchange the elements pointed by the 'left' and 'right' marks.

Once the 'right' mark passes the last element, it means that all the elements have been processed. The elements which are greater than the pivot have been moved to the right hand of 'left' mark while the others are to the left hand of this mark. Note that the pivot should move between the two segments. An extra exchanging between the pivot and the element pointed by 'left' mark makes this final one to the correct location. This is shown by the swap bi-directional arrow in figure 2 (c).

The 'left' mark (which points the pivot finally) partitions the whole array into two parts, it is returned as the result. We typically increase the 'left' mark by one, so that it points to the first element greater than the pivot for convenient. Note that the array is modified in-place.

The partition algorithm can be described as the following. It takes three arguments, the array A, the lower and the upper bound to be partitioned [1].

```
function Partition(A, l, u)
    $p \leftarrow A[l]$                          ▷ the pivot
    $L \leftarrow l$                             ▷ the left mark
    for $R \in [l+1, u]$ do                      ▷ iterate on the right mark
        if $\neg(p < A[R])$ then     ▷ negate of $<$ is enough for strict weak order
            $L \leftarrow L + 1$
            Exchange $A[L] \leftrightarrow A[R]$
    Exchange $A[L] \leftrightarrow p$
    return $L + 1$                               ▷ The partition position
```

Below table shows the steps of partitioning the array $\{3, 2, 5, 4, 0, 1, 6, 7\}$.

[1] The partition algorithm used here is slightly different from the one in [2]. The latter uses the last element in the slice as the pivot.

(l) **3**	(r) 2	5	4	0	1	6	7	initialize, $pivot = 3, l = 1, r = 2$
3	(l)(r) 2	5	4	0	1	6	7	$2 < 3$, advances l, $(r = l)$
3	(l) 2	(r) 5	4	0	1	6	7	$5 > 3$, moves on
3	(l) 2	5	(r) 4	0	1	6	7	$4 > 3$, moves on
3	(l) 2	5	4	(r) 0	1	6	7	$0 < 3$
3	2	(l) 0	4	(r) 5	1	6	7	Advances l, then swap with r
3	2	(l) 0	4	5	(r) 1	6	7	$1 < 3$
3	2	0	(l) 1	5	(r) 4	6	7	Advances l, then swap with r
3	2	0	(l) 1	5	4	(r) 6	7	$6 > 3$, moves on
3	2	0	(l) 1	5	4	6	(r) 7	$7 > 3$, moves on
1	2	0	3	(l+1) 5	4	6	7	r passes the end, swap pivot and l

This version of partition algorithm can be implemented in ANSI C as the following.

```
int partition(Key* xs, int l, int u) {
    int pivot, r;
    for (pivot = l, r = l + 1; r < u; ++r)
        if (!(xs[pivot] < xs[r])) {
            ++l;
            swap(xs[l], xs[r]);
        }
    swap(xs[pivot], xs[l]);
    return l + 1;
}
```

Where `swap(a, b)` can either be defined as function or a macro. In ISO C++, `swap(a, b)` is provided as a function template. the type of the elements can be defined somewhere or abstracted as a template parameter in ISO C++. We omit these language specific details here.

With the in-place partition realized, the imperative in-place quick sort can be accomplished by using it.

```
procedure QUICK-SORT($A, l, u$)
    if $l < u$ then
        $m \leftarrow$ PARTITION($A, l, u$)
        QUICK-SORT($A, l, m - 1$)
        QUICK-SORT($A, m, u$)
```

When sort an array, this procedure is called by passing the whole range as the lower and upper bounds. QUICK-SORT$(A, 1, |A|)$. Note that when $l \geq u$ it means the array slice is either empty, or just contains only one element, both can be treated as ordered, so the algorithm does nothing in such cases.

Below ANSI C example program completes the basic in-place quick sort.

```
void quicksort(Key* xs, int l, int u) {
    int m;
    if (l < u) {
        m = partition(xs, l, u);
        quicksort(xs, l, m - 1);
        quicksort(xs, m, u);
    }
}
```

2.4 Minor improvement in functional partition

Before exploring how to improve the partition for basic version quick sort, it's obviously that the one presented so far can be defined by using folding. Please refer to the appendix A of this book for definition of folding.

$$partition(p, L) = fold(f(p), (\Phi, \Phi), L) \tag{4}$$

Where function f compares the element to the pivot with predicate p (which is passed to f as a parameter, so that f is in curried form, see appendix A for detail. Alternatively, f can be a lexical closure which is in the scope of *partition*, so that it can access the predicate in this scope.), and update the result pair accordingly.

$$f(p, x, (A, B)) = \begin{cases} (\{x\} \cup A, B) & : \quad p(x) \\ (A, \{x\} \cup B) & : \quad otherwise(\neg p(x)) \end{cases} \tag{5}$$

Note we actually use pattern-matching style definition. In environment without pattern-matching support, the pair (A, B) should be represented by a variable, for example P, and use access functions to extract its first and second parts.

The example Haskell program needs to be modified accordingly.

```
sort [] = []
sort (x:xs) = sort small ++ [x] ++ sort big where
  (small, big) = foldr f ([], []) xs
  f a (as, bs) = if a ≤ x then (a:as, bs) else (as, a:bs)
```

2.4.1 Accumulated partition

The partition algorithm by using folding actually accumulates to the result pair of lists (A, B). That if the element is not greater than the pivot, it's accumulated to A, otherwise to B. We can explicitly express it which save spaces and is friendly for tail-recursive call optimization (refer to the appendix A of this book for detail).

$$partition(p, L, A, B) = \begin{cases} (A, B) & : \quad L = \Phi \\ partition(p, L', \{l_1\} \cup A, B) & : \quad p(l_1) \\ partition(p, L', A, \{l_1\} \cup B) & : \quad otherwise \end{cases} \tag{6}$$

Where l_1 is the first element in L if L isn't empty, and L' contains the rest elements except for l_1, that $L' = \{l_2, l_3, ...\}$ for example. The quick sort algorithm then uses this accumulated partition function by passing the $\lambda_x x \leq pivot$ as the partition predicate.

$$sort(L) = \begin{cases} \Phi & : \quad L = \Phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : \quad otherwise \end{cases} \tag{7}$$

Where A, B are computed by the accumulated partition function defined above.

$$(A, B) = partition(\lambda_x x \leq l_1, L', \Phi, \Phi)$$

2.4.2 Accumulated quick sort

Observe the recursive case in the last quick sort definition. the list concatenation operations $sort(A) \cup \{l_1\} \cup sort(B)$ actually are proportion to the length of the list to be concatenated. Of course we can use some general solutions introduced in the appendix A of this book to improve it. Another way is to change the sort algorithm to accumulated manner. Something like below:

$$sort'(L, S) = \begin{cases} S & : \quad L = \Phi \\ ... & : \quad otherwise \end{cases}$$

Where S is the accumulator, and we call this version by passing empty list as the accumulator to start sorting: $sort(L) = sort'(L, \Phi)$. The key intuitive is that after the partition finishes, the two sub lists need to be recursively sorted. We can first recursively sort the list contains the elements which are greater than the pivot, then link the pivot in front of it and use it as an *accumulator* for next step sorting.

Based on this idea, the '...' part in above definition can be realized as the following.

$$sort'(L, S) = \begin{cases} S & : \quad L = \Phi \\ sort(A, \{l_1\} \cup sort(B, ?)) & : \quad otherwise \end{cases}$$

The problem is what's the accumulator when sorting B. There is an important invariant actually, that at every time, the accumulator S holds the elements have been sorted so far. So that we should sort B by accumulating to S.

$$sort'(L, S) = \begin{cases} S & : \quad L = \Phi \\ sort(A, \{l_1\} \cup sort(B, S)) & : \quad otherwise \end{cases} \tag{8}$$

The following Haskell example program implements the accumulated quick sort algorithm.

```
asort xs = asort’ xs []

asort’ [] acc = acc
asort’ (x:xs) acc = asort’ as (x:asort’ bs acc) where
  (as, bs) = part xs [] []
  part [] as bs = (as, bs)
part (y:ys) as bs | y ≤ x = part ys (y:as) bs
                  | otherwise = part ys as (y:bs)
```

Exercise 1

- Implement the recursive basic quick sort algorithm in your favorite imperative programming language.
- Same as the imperative algorithm, one minor improvement is that besides the empty case, we needn't sort the singleton list, implement this idea in the functional algorithm as well.
- The accumulated quick sort algorithm developed in this section uses intermediate variable A, B. They can be eliminated by defining the partition function to mutually recursive call the sort function. Implement this idea in your favorite functional programming language. Please don't refer to the downloadable example program along with this book before you try it.

3 Performance analysis for quick sort

Quick sort performs well in practice, however, it's not easy to give theoretical analysis. It needs the tool of probability to prove the average case performance.

Nevertheless, it's intuitive to calculate the best case and worst case performance. It's obviously that the best case happens when every partition divides the sequence into two slices with equal size. Thus it takes $O(\lg N)$ recursive calls as shown in figure 3.

There are total $O(\lg N)$ levels of recursion. In the first level, it executes one partition, which processes N elements; In the second level, it executes partition two times, each processes $N/2$ elements, so the total time in the second level bounds to $2O(N/2) = O(N)$ as well. In the third level, it executes partition four times, each processes $N/4$ elements. The total time in the third level is also bound to $O(N)$; ... In the last level, there are N small slices each contains a single element, the time is bound to $O(N)$. Summing all the time in each level gives the total performance of quick sort in best case as $O(N \lg N)$.

However, in the worst case, the partition process unluckily divides the sequence to two slices with unbalanced lengths in most time. That one slices with length $O(1)$, the other is $O(N)$. Thus the recursive time degrades to $O(N)$. If we draw a similar figure, unlike in the best case, which forms a balanced binary tree, the worst case degrades into a very unbalanced tree that every node has only one child, while the other is empty. The binary tree turns to be a linked list with $O(N)$ length. And in every level, all the elements are processed, so the total performance in worst case is $O(N^2)$, which is as same poor as insertion sort and selection sort.

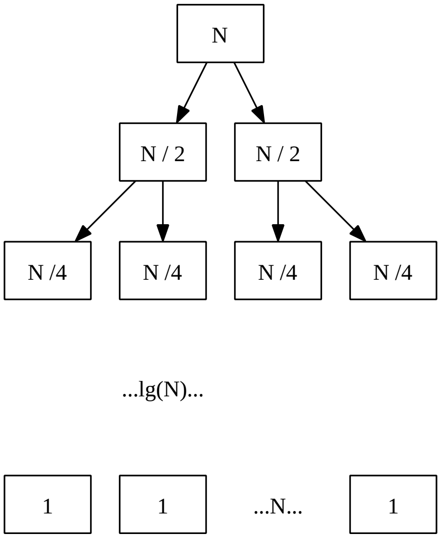


Figure 3: In the best case, quick sort divides the sequence into two slices with same length.

Let's consider when the worst case will happen. One special case is that all the elements (or most of the elements) are same. Nico Lomuto's partition method deals with such sequence poor. We'll see how to solve this problem by introducing other partition algorithm in the next section.

The other two obvious cases which lead to worst case happen when the sequence has already in ascending or descending order. Partition the ascending sequence makes an empty sub list before the pivot, while the list after the pivot contains all the rest elements. Partition the descending sequence gives an opponent result.

There are other cases which lead quick sort performs poor. There is no completely satisfied solution which can avoid the worst case. We'll see some engineering practice in next section which can make it very seldom to meet the worst case.

3.1 Average case analysis $\star$

In average case, quick sort performs well. There is a vivid example that even the partition divides the list every time to two lists with length 1 to 9. The performance is still bound to $O(N \lg N)$ as shown in [2].

This subsection need some mathematic background, reader can safely skip to next part.

There are two methods to proof the average case performance, one uses an important fact that the performance is proportion to the total comparing operations during quick sort [2]. Different with the selections sort that every two elements have been compared. Quick sort avoid many unnecessary comparisons. For example suppose a partition operation on list $\{a_1, a_2, a_3, ..., a_n\}$. Select a_1 as the pivot, the partition builds two sub lists $A = \{x_1, x_2, ..., x_k\}$ and $B = \{y_1, y_2, ..., y_{n-k-1}\}$. In the rest time of quick sort, The element in A will never be compared with any elements in B.

Denote the final sorted result as $\{a_1, a_2, ..., a_n\}$, this indicates that if element $a_i < a_j$, they will not be compared any longer if and only if some element a_k where $a_i < a_k < a_j$ has ever been selected as pivot before a_i or a_j being selected as the pivot.

That is to say, the only chance that a_i and a_j being compared is either a_i is chosen as pivot or a_j is chosen as pivot before any other elements in ordered range $a_{i+1} < a_{i+2} < ... < a_{j-1}$ are selected.

Let $P(i,)$ represent the probability that a_i and a_j being compared. We have:

$$P(i, j) = \frac{2}{j - i + 1} \tag{9}$$

Since the total number of compare operation can be given as:

$$C(N) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(i, j) \tag{10}$$

Note the fact that if we compared a_i and a_j, we won't compare a_j and a_i again in the quick sort algorithm, and we never compare a_i onto itself. That's why we set the upper bound of i to $N - 1$; and lower bound of j to $i + 1$.

Substitute the probability, it yields:

$$\begin{aligned} C(N) &= \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{N-1} \sum_{k=1}^{N-i} \frac{2}{k + 1} \end{aligned} \tag{11}$$

Using the harmonic series [18]

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... = \ln n + \gamma + \epsilon_n$$

$$C(N) = \sum_{i=1}^{N-1} O(\lg N) = O(N \lg N) \tag{12}$$

The other method to prove the average performance is to use the recursive fact that when sorting list of length N, the partition splits the list into two sub lists with length i and $N - i - 1$. The partition process itself takes cN time because it examine every element with the pivot. So we have the following equation.

$$T(N) = T(i) + T(N - i - 1) + cN \tag{13}$$

Where $T(n)$ is the total time when perform quick sort on list of length n. Since i is equally like to be any of 0, 1, ..., N-1, taking math expectation to the equation gives:

$$\begin{aligned}
T(N) &= E(T(i)) + E(T(N-i-1)) + cN \\
&= \frac{1}{N}\sum_{i=0}^{N-1} T(i) + \frac{1}{N}\sum_{i=0}^{N-1} T(N-i-1) + cN \\
&= \frac{1}{N}\sum_{i=0}^{N-1} T(i) + \frac{1}{N}\sum_{j=0}^{N-1} T(j) + cN \\
&= \frac{2}{N}\sum_{i=0}^{N-1} T(i) + cN
\end{aligned} \tag{14}$$

Multiply by N to both sides, the equation changes to:

$$NT(N) = 2\sum_{i=0}^{N-1} T(i) + cN^2 \tag{15}$$

Substitute N to $N-1$ gives another equation:

$$(N-1)T(N-1) = 2\sum_{i=0}^{N-2} T(i) + c(N-1)^2 \tag{16}$$

Subtract equation (15) and (16) can eliminate all the $T(i)$ for $0 \leq i < N-1$.

$$NT(N) = (N+1)T(N-1) + 2cN - c \tag{17}$$

As we can drop the constant time c in computing performance. The equation can be one more step changed like below.

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \tag{18}$$

Next we assign N to $N-1$, $N-2$, ..., which gives us $N-1$ equations.

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1}$$

...

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Sum all them up, and eliminate the same components in both sides, we can deduce to a function of N.

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c\sum_{k=3}^{N+1}\frac{1}{k} \tag{19}$$

Using the harmonic series mentioned above, the final result is:

$$O(\frac{T(N)}{N+1}) = O(\frac{T(1)}{2} + 2c\ln N + \gamma + \epsilon_N) = O(\lg N) \tag{20}$$

Thus

$$O(T(N)) = O(N\lg N) \tag{21}$$

Exercise 2

- Why Lomuto's methods performs poor when there are many duplicated elements?

4 Engineering Improvement

Quick sort performs well in most cases as mentioned in previous section. However, there does exist the worst cases which downgrade the performance to quadratic. If the data is randomly prepared, such case is rare, however, there are some particular sequences which lead to the worst case and these kinds of sequences are very common in practice.

In this section, some engineering practices are introduces which either help to avoid poor performance in handling some special input data with improved partition algorithm, or try to uniform the possibilities among cases.

4.1 Engineering solution to duplicated elements

As presented in the exercise of above section, N. Lomuto's partition method isn't good at handling sequence with many duplicated elements. Consider a sequence with N equal elements like: $\{x, x, ..., x\}$. There are actually two methods to sort it.

1. The normal basic quick sort: That we select an arbitrary element, which is x as the pivot, partition it to two sub sequences, one is $\{x, x, ..., x\}$, which contains $N-1$ elements, the other is empty. then recursively sort the first one; this is obviously quadratic $O(N^2)$ solution.

2. The other way is to only pick those elements strictly smaller than x, and strictly greater than x. Such partition results two empty sub sequences, and N elements equal to the pivot. Next we recursively sort the sub

sequences contains the smaller and the bigger elements, since both of them are empty, the recursive call returns immediately; The only thing left is to concatenate the sort results in front of and after the list of elements which are equal to the pivot.

The latter one performs in $O(N)$ time if all elements are equal. This indicates an important improvement for partition. That instead of binary partition (split to two sub lists and a pivot), ternary partition (split to three sub lists) handles duplicated elements better.

We can define the ternary quick sort as the following.

$$sort(L) = \left\{ \begin{array}{rcl} \Phi & : & L = \Phi \\ sort(S) \cup sort(E) \cup sort(G) & : & otherwise \end{array} \right. \tag{22}$$

Where S, E, G are sub lists contains all elements which are less than, equal to, and greater than the pivot respectively.

$$\begin{array}{l} S = \{x | x \in L, x < l_1\} \\ E = \{x | x \in L, x = l_1\} \\ G = \{x | x \in L, l_1 < x\} \end{array}$$

The basic ternary quick sort can be implemented in Haskell as the following example code.

```
sort [] = []
sort (x:xs) = sort [a | a←xs, a<x] ++
              x:[b | b←xs, b==x] ++ sort [c | c←xs, c>x]
```

Note that the comparison between elements must support abstract 'less-than' and 'equal-to' operations. The basic version of ternary sort takes linear $O(N)$ time to concatenate the three sub lists. It can be improved by using the standard techniques of accumulator.

Suppose function $sort'(L, A)$ is the accumulated ternary quick sort definition, that L is the sequence to be sorted, and the accumulator A contains the intermediate sorted result so far. We initialize the sorting with an empty accumulator: $sort(L) = sort'(L, \Phi)$.

It's easy to give the trivial edge cases like below.

$$sort'(L, A) = \left\{ \begin{array}{rcl} A & : & L = \Phi \\ ... & : & otherwise \end{array} \right.$$

For the recursive case, as the ternary partition splits to three sub lists S, E, G, only S and G need recursive sort, E contains all elements equal to the pivot, which is in correct order thus needn't to be sorted any more. The idea is to sort G with accumulator A, then concatenate it behind E, then use this result as the new accumulator, and start to sort S:

$$sort'(L, A) = \left\{ \begin{array}{rcl} A & : & L = \Phi \\ sort(S, E \cup sort(G, A)) & : & otherwise \end{array} \right. \tag{23}$$

The partition can also be realized with accumulators. It is similar to what has been developed for the basic version of quick sort. Note that we can't just pass only one predication for pivot comparison. It actually needs two, one for less-than, the other for equality testing. For the sake of brevity, we pass the pivot element instead.

$$partition(p, L, S, E, G) = \begin{cases} (S, E, G) & : \quad L = \Phi \\ partition(p, L', \{l_1\} \cup S, E, G) & : \quad l_1 < p \\ partition(p, L', S, \{l_1\} \cup E, G) & : \quad l_1 = p \\ partition(p, L', S, E, \{l_1\} \cup G) & : \quad p < l_1 \end{cases} \tag{24}$$

Where l_1 is the first element in L if L isn't empty, and L' contains all rest elements except for l_1. Below Haskell program implements this algorithm. It starts the recursive sorting immediately in the edge case of parition.

```
sort xs = sort' xs []

sort' []     r = r
sort' (x:xs) r = part xs [] [x] [] r where
    part [] as bs cs r = sort' as (bs ++ sort' cs r)
    part (x':xs') as bs cs r | x' <  x = part xs' (x':as) bs cs r
                             | x' == x = part xs' as (x':bs) cs r
                             | x' >  x = part xs' as bs (x':cs) r
```

Richard Bird developed another version in [7], that instead of concatenating the recursively sorted results, it uses a list of sorted sub lists, and performs concatenation finally.

```
sort xs = concat $ pass xs []

pass [] xss = xss
pass (x:xs) xss = step xs [] [x] [] xss where
    step [] as bs cs xss = pass as (bs:pass cs xss)
    step (x':xs') as bs cs xss | x' <  x = step xs' (x':as) bs cs xss
                               | x' == x = step xs' as (x':bs) cs xss
                               | x' >  x = step xs' as bs (x':cs) xss
```

4.1.1 2-way partition

The cases with many duplicated elements can also be handled imperatively. Robert Sedgewick presented a partition method [3], [4] which holds two pointers. One moves from left to right, the other moves from right to left. The two pointers are initialized as the left and right boundaries of the array.

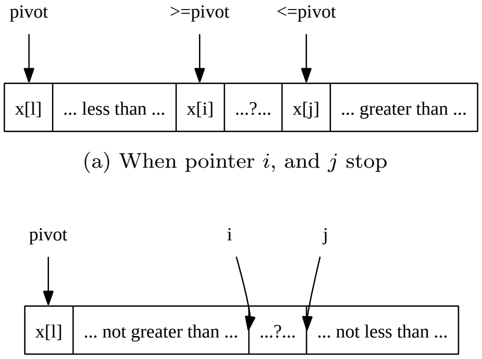
When start partition, the left most element is selected as the pivot. Then the left pointer i keeps advancing to right until it meets any element which is not less than the pivot; On the other hand[2], The right pointer j repeatedly scans to left until it meets any element which is not greater than the pivot.

[2] We don't use 'then' because it's quite OK to perform the two scans in parallel.

At this time, all elements before the left pointer i are strictly less than the pivot, while all elements after the right pointer j are greater than the pivot. i points to an element which is either greater than or equal to the pivot; while j points to an element which is either less than or equal to the pivot, the situation at this stage is illustrated in figure 4 (a).

In order to partition all elements less than or equal to the pivot to the left, and the others to the right, we can exchange the two elements pointed by i, and j. After that the scan can be resumed until either i meets j, or they overlap.

At any time point during partition. There is invariant that all elements before i (including the one pointed by i) are not greater than the pivot; while all elements after j (including the one pointed by j) are not less than the pivot. The elements between i and j haven't been examined yet. This invariant is shown in figure 4 (b).



(a) When pointer i, and j stop

(b) Partition invariant

Figure 4: Partition a range of array by using the left most element as the pivot.

After the left pointer i meets the right pointer j, or they overlap each other, we need one extra exchanging to move the pivot located at the first position to the correct place which is pointed by j. Next, the elements between the lower bound and j as well as the sub slice between i and the upper bound of the array are recursively sorted.

This algorithm can be described as the following.

```
procedure SORT($A, l, u$)                          ▷ sort range $[l, u)$
    if $u - l > 1$ then             ▷ More than 1 element for non-trivial case
        $i \leftarrow l$, $j \leftarrow u$
        $pivot \leftarrow A[l]$
        loop
            repeat
                $i \leftarrow i + 1$
            until $A[i] \geq pivot$   ▷ Need handle error case that $i \geq u$ in fact.
            repeat
                $j \leftarrow j - 1$
            until $A[j] \leq pivot$    ▷ Need handle error case that $j < l$ in fact.
            if $j < i$ then
                break
            EXCHANGE $A[i] \leftrightarrow A[j]$
        EXCHANGE $A[l] \leftrightarrow A[j]$    ▷ Move the pivot
        SORT($A, l, j$)
        SORT($A, i, u$)
```

Consider the extreme case that all elements are equal, this in-place quick sort will partition the list to two equal length sub lists although it takes $\frac{N}{2}$ unnecessary swaps. As the partition is balanced, the overall performance is $O(N \lg N)$, which avoid downgrading to quadratic. The following ANSI C example program implements this algorithm.

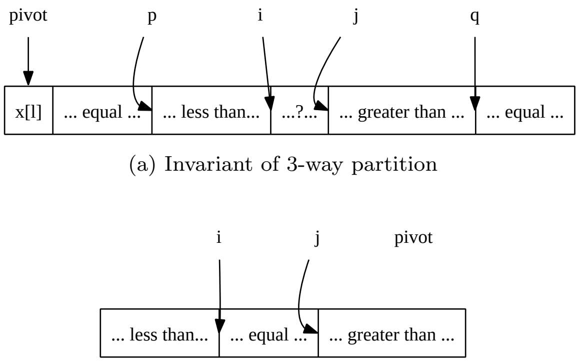
```
void qsort(Key* xs, int l, int u) {
    int i, j, pivot;
    if (l < u - 1) {
        pivot = i = l; j = u;
        while (1) {
            while (i < u && xs[++i] < xs[pivot]);
            while (j ≥l && xs[pivot] < xs[--j]);
            if (j < i) break;
            swap(xs[i], xs[j]);
        }
        swap(xs[pivot], xs[j]);
        qsort(xs, l, j);
        qsort(xs, i, u);
    }
}
```

Comparing this algorithm with the basic version based on N. Lumoto's partition method, we can find that it swaps fewer elements, because it skips those have already in proper sides of the pivot.

4.1.2 3-way partition

It's obviously that, we should avoid those unnecessary swapping for the duplicated elements. What's more, the algorithm can be developed with the idea of ternary sort (as known as 3-way partition in some materials), that all the elements which are strictly less than the pivot are put to the left sub slice, while those are greater than the pivot are put to the right. The middle part holds all the elements which are equal to the pivot. With such ternary partition, we need only recursively sort the ones which differ from the pivot. Thus in the above extreme case, there aren't any elements need further sorting. So the overall performance is linear $O(N)$.

The difficulty is how to do the 3-way partition. Jon Bentley and Douglas McIlroy developed a solution which keeps those elements equal to the pivot at the left most and right most sides as shown in figure 5 (a) [5] [6].



(a) Invariant of 3-way partition

(b) Swapping the equal parts to the middle

Figure 5: 3-way partition.

The majority part of scan process is as same as the one developed by Robert Sedgewick, that i and j keep advancing toward each other until they meet any element which is greater then or equal to the pivot for i, or less than or equal to the pivot for j respectively. At this time, if i and j don't meet each other or overlap, they are not only exchanged, but also examined if the elements pointed by them are identical to the pivot. Then necessary exchanging happens between i and p, as well as j and q.

By the end of the partition process, the elements equal to the pivot need to be swapped to the middle part from the left and right ends. The number of such extra exchanging operations are proportion to the number of duplicated elements. It's zero operation if elements are unique which there is no overhead in the case. The final partition result is shown in figure 5 (b). After that we only need recursively sort the 'less-than' and 'greater-than' sub slices.

This algorithm can be given by modifying the 2-way partition as below.

```
procedure SORT($A, l, u$)
    if $u - l > 1$ then
        $i \leftarrow l$, $j \leftarrow u$
        $p \leftarrow l$, $q \leftarrow u$          ▷ points to the boundaries for equal elements
        $pivot \leftarrow A[l]$
        loop
            repeat
                $i \leftarrow i + 1$
            until $A[i] \geq pivot$          ▷ Skip the error handling for $i \geq u$
            repeat
                $j \leftarrow j - 1$
            until $A[j] \leq pivot$          ▷ Skip the error handling for $j < l$
            if $j \leq i$ then
                break          ▷ Note the difference form the above algorithm
        EXCHANGE $A[i] \leftrightarrow A[j]$
        if $A[i] = pivot$ then                    ▷ Handle the equal elements
            $p \leftarrow p + 1$
            EXCHANGE $A[p] \leftrightarrow A[i]$
        if $A[j] = pivot$ then
            $q \leftarrow q - 1$
            EXCHANGE $A[q] \leftrightarrow A[j]$
    if $i = j \wedge A[i] = pivot$ then                  ▷ A special case
        $j \leftarrow j - 1$, $i \leftarrow i + 1$
    for $k$ from $l$ to $p$ do    ▷ Swap the equal elements to the middle part
        EXCHANGE $A[k] \leftrightarrow A[j]$
        $j \leftarrow j - 1$
    for $k$ from $u - 1$ down-to $q$ do
        EXCHANGE $A[k] \leftrightarrow A[i]$
        $i \leftarrow i + 1$
    SORT($A, l, j + 1$)
    SORT($A, i, u$)
```

This algorithm can be translated to the following ANSI C example program.

```
void qsort2(Key* xs, int l, int u) {
    int i, j, k, p, q, pivot;
    if (l < u - 1) {
        i = p = l; j = q = u; pivot = xs[l];
        while (1) {
            while (i < u && xs[++i] < pivot);
            while (j ≥ l && pivot < xs[--j]);
            if (j ≤ i) break;
            swap(xs[i], xs[j]);
            if (xs[i] == pivot) { ++p; swap(xs[p], xs[i]); }
            if (xs[j] == pivot) { --q; swap(xs[q], xs[j]); }
        }
        if (i == j && xs[i] == pivot) { --j, ++i; }
        for (k = l; k ≤ p; ++k, --j) swap(xs[k], xs[j]);
        for (k = u-1; k ≥ q; --k, ++i) swap(xs[k], xs[i]);
        qsort2(xs, l, j + 1);
        qsort2(xs, i, u);
    }
}
```

It can be seen that the the algorithm turns to be a bit complex when it evolves to 3-way partition. There are some tricky edge cases should be handled with caution. Actually, we just need a ternary partition algorithm. This remind us the N. Lumoto's method, which is straightforward enough to be a start point.

The idea is to change the invariant a bit. We still select the first element as the pivot, as shown in figure 6, at any time, the left most section contains elements which are strictly less than the pivot; the next section contains the elements equal to the pivot; the right most section holds all the elements which are strictly greater than the pivot. The boundaries of three sections are marked as i, k, and j respectively. The rest part, which is between k and j are elements haven't been scanned yet.

At the beginning of this algorithm, the 'less-than' section is empty; the 'equal-to' section contains only one element, which is the pivot; so that i is initialized to the lower bound of the array, and k points to the element next to i. The 'greater-than' section is also initialized as empty, thus j is set to the upper bound.

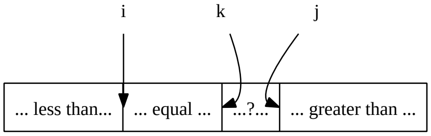


Figure 6: 3-way partition based on N. Lumoto's method.

When the partition process starts, the elements pointed by k is examined. If it's equal to the pivot, k just advances to the next one; If it's greater than the pivot, we swap it with the last element in the unknown area, so that the length of 'greater-than' section increases by one. It's boundary j moves to the left. Since we don't know if the elements swapped to k is still greater than the pivot, it should be examined again repeatedly. Otherwise, if the element is less than the pivot, we can exchange it with the first one in the 'equal-to' section to resume the invariant. The partition algorithm stops when k meets j.

```
procedure SORT($A, l, u$)
    if $u - l > 1$ then
        $i \leftarrow l$, $j \leftarrow u$, $k \leftarrow l + 1$
        $pivot \leftarrow A[i]$
        while $k < j$ do
            while $pivot < A[k]$ do
                $j \leftarrow j - 1$
                EXCHANGE $A[k] \leftrightarrow A[j]$
            if $A[k] < pivot$ then
                EXCHANGE $A[k] \leftrightarrow A[i]$
                $i \leftarrow i + 1$
            $k \leftarrow k + 1$
        SORT($A, l, i$)
        SORT($A, j, u$)
```

Compare this one with the previous 3-way partition quick sort algorithm, it's more simple at the cost of more swapping operations. Below ANSI C program implements this algorithm.

```
void qsort(Key* xs, int l, int u) {
    int i, j, k; Key pivot;
    if (l < u - 1) {
        i = l; j = u; pivot = xs[l];
        for (k = l + 1; k < j; ++k) {
            while (pivot < xs[k]) { --j; swap(xs[j], xs[k]); }
            if (xs[k] < pivot) { swap(xs[i], xs[k]); ++i; }
        }
        qsort(xs, l, i);
        qsort(xs, j, u);
    }
}
```

Exercise 3

- All the quick sort imperative algorithms use the first element as the pivot, another method is to choose the last one as the pivot. Realize the quick sort algorithms, including the basic version, Sedgewick version, and ternary (3-way partition) version by using this approach.

5 Engineering solution to the worst case

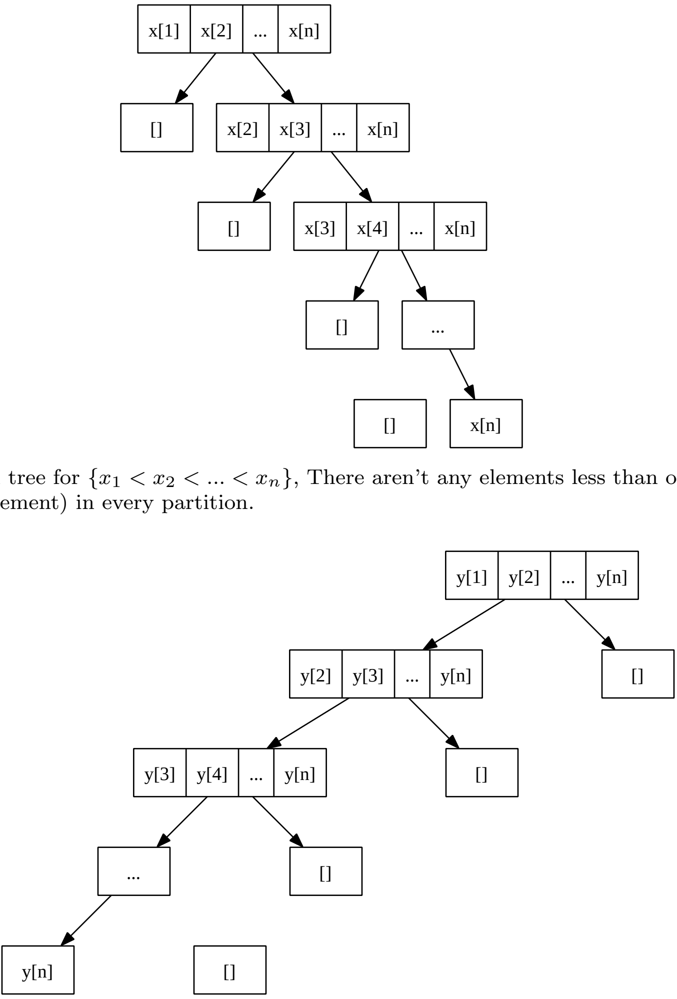
Although the ternary quick sort (3-way partition) solves the issue for duplicated elements, it can't handle some typical worst cases. For example if many of the elements in the sequence are ordered, no matter it's in ascending or descending order, the partition result will be two unbalanced sub sequences, one with few elements, the other contains all the rest.

Consider the two extreme cases, $\{x_1 < x_2 < ... < x_n\}$ and $\{y_1 > y_2 > ... > y_n\}$. The partition results are shown in figure 7.

It's easy to give some more worst cases, for example, $\{x_m, x_{m-1}, ..., x_2, x_1, x_{m+1}, x_{m+2}, ...x_n\}$ where $\{x_1 < x_2 < ... < x_n\}$; Another one is $\{x_n, x_1, x_{n-1}, x_2, ...\}$. Their partition result trees are shown in figure 8.

Observing that the bad partition happens easily when blindly choose the first element as the pivot, there is a popular work around suggested by Robert Sedgwick in [3]. Instead of selecting the fixed position in the sequence, a small sampling helps to find a pivot which has lower possibility to cause a bad partition. One option is to examine the first element, the middle, and the last one, then choose the median of these three element. In the worst case, it can ensure that there is at least one element in the shorter partitioned sub list.

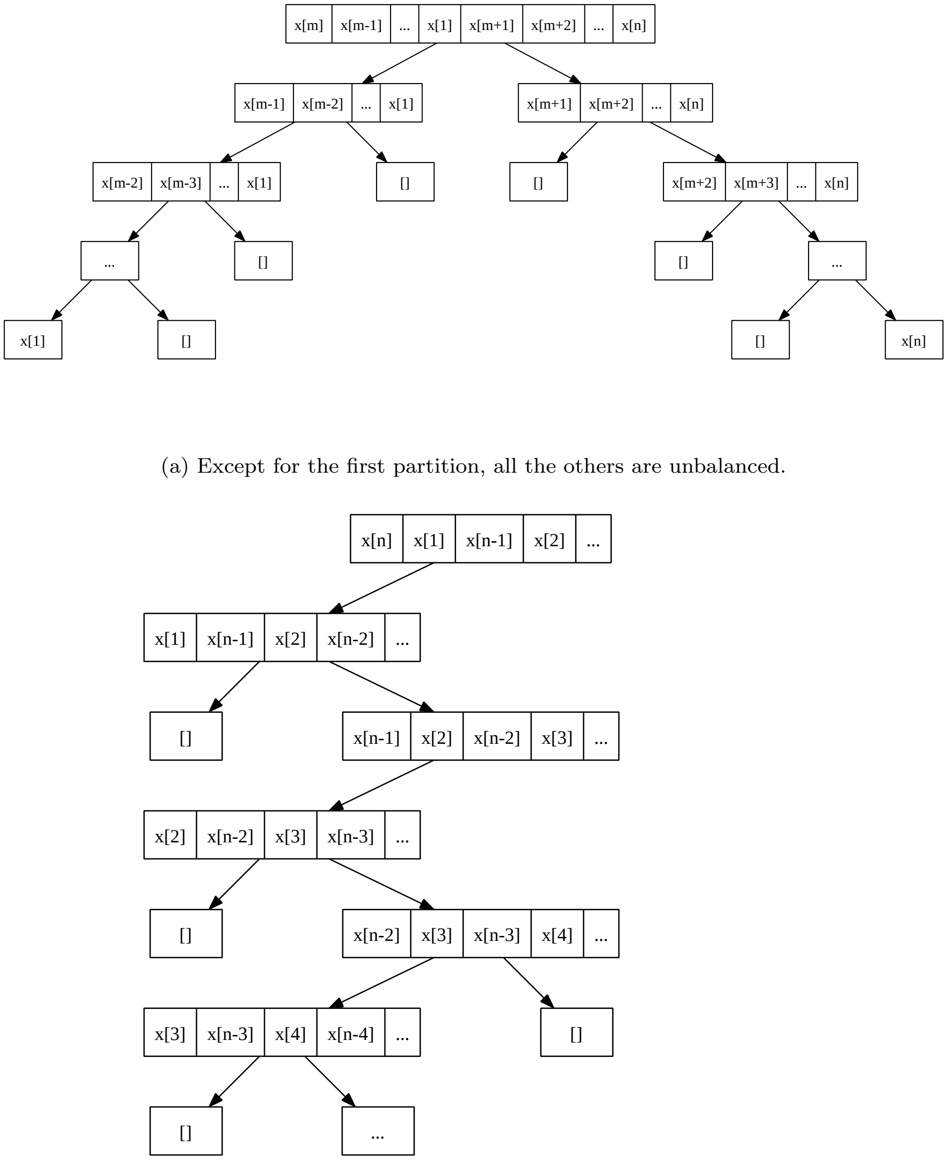
Note that there is one tricky in real-world implementation. Since the index is typically represented in limited length words. It may cause overflow when calculating the middle index by the naive expression `(l + u) / 2`. In order to avoid this issue, it can be accessed as `l + (u - l)/2`. There are two methods to find the median, one needs at most three comparisons [5]; the other is to move the minimum value to the first location, the maximum value to the last location, and the median value to the meddle location by swapping. After that we can select the middle as the pivot. Below algorithm illustrated the second idea before calling the partition procedure.



(a) The partition tree for $\{x_1 < x_2 < ... < x_n\}$, There aren't any elements less than or equal to the pivot (the first element) in every partition.

(b) The partition tree for $\{y_1 > y_2 > ... > y_n\}$, There aren't any elements greater than or equal to the pivot (the first element) in every partition.

Figure 7: The two worst cases.



(a) Except for the first partition, all the others are unbalanced.

(b) A zig-zag partition tree.

Figure 8: Another two worst cases.

```
procedure Sort$(A, l, u)$
    if $u - l > 1$ then
        $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$        ▷ Need handle overflow error in practice
        if $A[m] < A[l]$ then        ▷ Ensure $A[l] \leq A[m]$
            Exchange $A[l] \leftrightarrow A[r]$
        if $A[u-1] < A[l]$ then        ▷ Ensure $A[l] \leq A[u-1]$
            Exchange $A[l] \leftrightarrow A[u-1]$
        if $A[u-1] < A[m]$ then        ▷ Ensure $A[m] \leq A[u-1]$
            Exchange $A[m] \leftrightarrow A[u]$
        Exchange $A[l] \leftrightarrow A[m]$
        $(i, j) \leftarrow$ Partition$(A, l, u)$
        Sort$(A, l, i)$
        Sort$(A, j, u)$
```

It's obviously that this algorithm performs well in the 4 special worst cases given above. The imperative implementation of median-of-three is left as exercise to the reader.

However, in purely functional settings, it's expensive to randomly access the middle and the last element. We can't directly translate the imperative median selection algorithm. The idea of taking a small sampling and then finding the median element as pivot can be realized alternatively by taking the first 3. For example, in the following Haskell program.

```
qsort [] = []
qsort [x] = [x]
qsort [x, y] = [min x y, max x y]
qsort (x:y:z:rest) = qsort (filter (< m) (s:rest)) ++ [m] ++ qsort (filter (≥ m) (l:rest)) where
    xs = [x, y, z]
    [s, m, l] = [minimum xs, median xs, maximum xs]
```

Unfortunately, none of the above 4 worst cases can be well handled by this program, this is because the sampling is not good. We need telescope, but not microscope to profile the whole list to be partitioned. We'll see the functional way to solve the partition problem later.

Except for the median-of-three, there is another popular engineering practice to get good partition result. instead of always taking the first element or the last one as the pivot. One alternative is to randomly select one. For example as the following modification.

```
procedure Sort$(A, l, u)$
    if $u - l > 1$ then
        Exchange $A[l] \leftrightarrow A[$ Random$(l, u)$ $]$
        $(i, j) \leftarrow$ Partition$(A, l, u)$
        Sort$(A, l, i)$
        Sort$(A, j, u)$
```

The function Random(l, u) returns a random integer i between l and u, that $l \leq i < u$. The element at this position is exchanged with the first one, so that it is selected as the pivot for the further partition. This algorithm is called *random quick sort* [2].

Theoretically, neither median-of-three nor random quick sort can avoid the worst case completely. If the sequence to be sorted is randomly distributed, no matter choosing the first one as the pivot, or the any other arbitrary one are equally in effect. Considering the underlying data structure of the sequence is singly linked-list in functional setting, it's expensive to strictly apply the idea of random quick sort in purely functional approach.

Even with this bad news, the engineering improvement still makes sense in real world programming.

6 Other engineering practice

There is some other engineering practice which doesn't focus on solving the bad partition issue. Robert Sedgewick observed that when the list to be sorted is short, the overhead introduced by quick sort is relative expense, on the other hand, the insertion sort performs better in such case [4], [5]. Sedgewick, Bentley and McIlroy tried different threshold, as known as 'Cut-Off', that when there are lesson than 'Cut-Off' elements, the sort algorithm falls back to insertion sort.

```
procedure SORT($A, l, u$)
    if $u - l >$ CUT-OFF then
        QUICK-SORT($A, l, u$)
    else
        INSERTION-SORT($A, l, u$)
```

The implementation of this improvement is left as exercise to the reader.

Exercise 4

- Can you figure out more quick sort worst cases besides the four given in this section?
- Implement median-of-three method in your favorite imperative programming language.
- Implement random quick sort in your favorite imperative programming language.
- Implement the algorithm which falls back to insertion sort when the length of list is small in both imperative and functional approach.

7 Side words

It's sometimes called 'true quick sort' if the implementation equipped with most of the engineering practice we introduced, including insertion sort fall-back with cut-off, in-place exchanging, choose the pivot by median-of-three method, 3-way-partition.

The purely functional one, which express the idea of quick sort perfect can't take all of them. Thus someone think the functional quick sort is essentially tree sort.

Actually, quick sort does have close relationship with tree sort. Richard Bird shows how to derive quick sort from binary tree sort by deforestation [8].

Consider a binary search tree creation algorithm called $unfold$. Which turns a list of elements into a binary search tree.

$$unfold(L) = \left\{ \begin{array}{rcl} \Phi & : & L = \Phi \\ tree(T_l, l_1, T_r) & : & otherwise \end{array} \right. \tag{25}$$

Where

$$\begin{array}{l} T_l = unfold(\{a | a \in L', a \leq l_1\}) \\ T_r = unfold(\{a | a \in L', l_1 < a\}) \end{array} \tag{26}$$

The interesting point is that, this algorithm creates tree in a different way as we introduced in the chapter of binary search tree. If the list to be unfold is empty, the result is obviously an empty tree. This is the trivial edge case; Otherwise, the algorithm set the first element l_1 in the list as the key of the node, and recursively creates its left and right children. Where the elements used to form the left child is those which are less than or equal to the key in L', while the rest elements which are greater than the key are used to form the right child.

Remind the algorithm which turns a binary search tree to a list by in-order traversing:

$$toList(T) = \left\{ \begin{array}{rcl} \Phi & : & T = \Phi \\ toList(left(T)) \cup \{key(T)\} \cup toList(right(T)) & : & otherwise \end{array} \right. \tag{27}$$

We can define quick sort algorithm by composing these two functions.

$$quickSort = toList \cdot unfold \tag{28}$$

The binary search tree built in the first step of applying $unfold$ is the intermediate result. This result is consumed by $toList$ and dropped after the second step. It's quite possible to eliminate this intermediate result, which leads to the basic version of quick sort.

The elimination of the intermediate binary search tree is called *deforestation*. This concept is based on Burstle-Darlington's work [9].

8 Merge sort

Although quick sort performs perfectly in average cases, it can't avoid the worst case no matter what engineering practice is applied. Merge sort, on the other kind, ensure the performance is bound to $O(N \lg N)$ in all the cases. It's particularly useful in theoretical algorithm design and analysis. Another feature is that merge sort is friendly for linked-space settings, which is suitable for sorting nonconsecutive stored sequences. Some functional programming and dynamic programming environments adopt merge sort as the standard library sorting solution, such as Haskel, Python and Java (later than Java 7).

In this section, we'll first brief the intuitive idea of merge sort, provide a basic version. After that, some variants of merge sort will be given including nature merge sort, and bottom-up merge sort.

8.1 Basic version

Same as quick sort, the essential idea behind merge sort is also divide and conquer. Different from quick sort, merge sort enforces the divide to be strictly balanced, that it always splits the sequence to be sorted at the middle point. After that, it recursively sort the sub sequences and merge the sorted two sequences to the final result. The algorithm can be described as the following.

In order to sort a sequence L,

- Trivial edge case: If the sequence to be sorted is empty, the result is obvious empty;
- Otherwise, split the sequence at the middle position, recursively sort the two sub sequences and merge the result.

The basic merge sort algorithm can be formalized with the following equation.

$$sort(L) = \left\{ \begin{array}{rcl} \Phi & : & L = \Phi \\ merge(sort(L_1), sort(L_2)) & : & otherwise, (L_1, L_2) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \end{array} \right. \tag{29}$$

8.1.1 Merge

There are two 'black-boxes' in the above merge sort definition, one is the *splitAt* function, which splits a list at a given position; the other is the *merge* function, which can merge two sorted lists into one.

As presented in the appendix of this book, it's trivial to realize *splitAt* in imperative settings by using random access. However, in functional settings, it's typically realized as a linear algorithm:

$$splitAt(n, L) = \left\{ \begin{array}{rcl} (\Phi, L) & : & n = 0 \\ (\{l_1\} \cup A, B) & : & otherwise, (A, B) = splitAt(n-1, L') \end{array} \right. \tag{30}$$

Where l_1 is the first element of L, and L' represents the rest elements except of l_1 if L isn't empty.

The idea of merge can be illustrated as in figure 9. Consider two lines of kids. The kids have already stood in order of their heights. that the shortest one stands at the first, then a taller one, the tallest one stands at the end of the line.

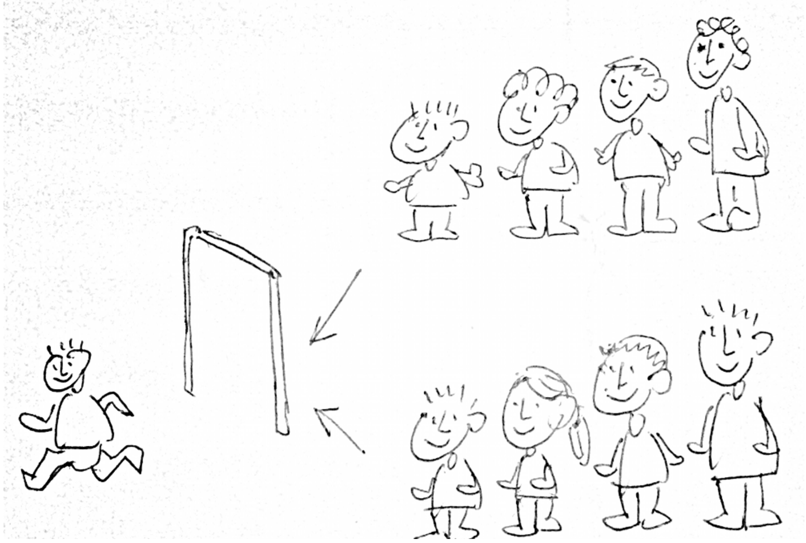


Figure 9: Two lines of kids pass a door.

Now let's ask the kids to pass a door one by one, every time there can be at most one kid pass the door. The kids must pass this door in the order of their height. The one can't pass the door before all the kids who are shorter than him/her.

Since the two lines of kids have already been 'sorted', the solution is to ask the first two kids, one from each line, compare their height, and let the shorter kid pass the door; Then they repeat this step until one line is empty, after that, all the rest kids can pass the door one by one.

This idea can be formalized in the following equation.

$$merge(A, B) = \begin{cases} A & : & B = \Phi \\ B & : & A = \Phi \\ \{a_1\} \cup merge(A', B) & : & a_1 \leq b_1 \\ \{b_1\} \cup merge(A, B') & : & otherwise \end{cases} \tag{31}$$

Where a_1 and b_1 are the first elements in list A and B; A' and B' are the rest elements except for the first ones respectively. The first two cases are trivial edge cases. That merge one sorted list with an empty list results the same sorted list; Otherwise, if both lists are non-empty, we take the first elements from the two lists, compare them, and use the minimum as the first one of the result, then recursively merge the rest.

With *merge* defined, the basic version of merge sort can be implemented like the following Haskell example code.

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort as) (msort bs) where
  (as, bs) = splitAt (length xs ‘div‘ 2) xs

merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x ≤ y = x : merge xs (y:ys)
                    | x >  y = y : merge (x:xs) ys
```

Note that, the implementation differs from the algorithm definition that it treats the singleton list as trivial edge case as well.

Merge sort can also be realized imperatively. The basic version can be developed as the below algorithm.

```
procedure SORT($A$)
    if $|A| > 1$ then
        $m \leftarrow \lfloor \frac{|A|}{2} \rfloor$
        $X \leftarrow$ COPY-ARRAY($A[1...m]$)
        $Y \leftarrow$ COPY-ARRAY($A[m+1...|A|]$)
        SORT($X$)
        SORT($Y$)
        MERGE($A, X, Y$)
```

When the array to be sorted contains at least two elements, the non-trivial sorting process starts. It first copy the first half to a new created array A, and the second half to a second new array B. Recursively sort them; and finally merge the sorted result back to A.

This version uses the same amount of extra spaces of A. This is because the MERGE algorithm isn't in-place at the moment. We'll introduce the imperative in-place merge sort in later section.

The merge process almost does the same thing as the functional definition. There is a verbose version and a simplified version by using sentinel.

The verbose merge algorithm continuously checks the element from the two input arrays, picks the smaller one and puts it back to the result array A, it then advances along the arrays respectively until either one input array is exhausted. After that, the algorithm appends the rest of the elements in the other input array to A.

```
procedure MERGE($A, X, Y$)
    $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$
    $m \leftarrow |X|, n \leftarrow |Y|$
    while $i \leq m \wedge j \leq n$ do
        if $X[i] < Y[j]$ then
            $A[k] \leftarrow X[i]$
            $i \leftarrow i + 1$
        else
```

```
            $A[k] \leftarrow Y[j]$
            $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    **while** $i \leq m$ **do**
        $A[k] \leftarrow X[i]$
        $k \leftarrow k + 1$
        $i \leftarrow i + 1$
    **while** $j \leq n$ **do**
        $A[k] \leftarrow Y[j]$
        $k \leftarrow k + 1$
        $j \leftarrow j + 1$
```

Although this algorithm is a bit verbose, it can be short in some programming environment with enough tools to manipulate array. The following Python program is an example.

```
def msort(xs):
    n = len(xs)
    if n > 1:
        ys = [x for x in xs[:n/2]]
        zs = [x for x in xs[n/2:]]
        ys = msort(ys)
        zs = msort(zs)
        xs = merge(xs, ys, zs)
    return xs

def merge(xs, ys, zs):
    i = 0
    while ys != [] and zs != []:
        xs[i] = ys.pop(0) if ys[0] < zs[0] else zs.pop(0)
        i = i + 1
    xs[i:] = ys if ys !=[] else zs
    return xs
```

8.1.2 Performance

Before dive into the improvement of this basic version, let's analyze the performance of merge sort. The algorithm contains two steps, divide step, and merge step. In divide step, the sequence to be sorted is always divided into two sub sequences with the same length. If we draw a similar partition tree as what we did for quick sort, it can be found this tree is a perfectly balanced binary tree as shown in figure 3. Thus the height of this tree is $O(\lg N)$. It means the recursion depth of merge sort is bound to $O(\lg N)$. Merge happens in every level. It's intuitive to analyze the merge algorithm, that it compare elements from two input sequences in pairs, after one sequence is fully examined the rest one is copied one by one to the result, thus it's a linear algorithm proportion to the length of the sequence. Based on this facts, denote $T(N)$ the time for sorting the sequence with length N, we can write the recursive time cost as below.

$$\begin{aligned} T(N) &= T(\frac{N}{2}) + T(\frac{N}{2}) + cN \\ &= 2T(\frac{N}{2}) + cN \end{aligned} \tag{32}$$

It states that the cost consists of three parts: merge sort the first half takes $T(\frac{N}{2})$, merge sort the second half takes also $T(\frac{N}{2})$, merge the two results takes cN, where c is some constant. Solve this equation gives the result as $O(N \lg N)$.

Note that, this performance doesn't vary in all cases, as merge sort always uniformly divides the input.

Another significant performance indicator is space occupation. However, it varies a lot in different merge sort implementation. The detail space bounds analysis will be explained in every detailed variants later.

For the basic imperative merge sort, observe that it demands same amount of spaces as the input array in every recursion, copies the original elements to them for recursive sort, and these spaces can be released after this level of recursion. So the peak space requirement happens when the recursion enters to the deepest level, which is $O(N \lg N)$.

The functional merge sort consume much less than this amount, because the underlying data structure of the sequence is linked-list. Thus it needn't extra spaces for merge[3]. The only spaces requirement is for book-keeping the stack for recursive calls. This can be seen in the later explanation of even-odd split algorithm.

8.1.3 Minor improvement

We'll next improve the basic merge sort bit by bit for both the functional and imperative realizations. The first observation is that the imperative merge algorithm is a bit verbose. [2] presents an elegant simplification by using positive ∞ as the sentinel. That we append ∞ as the last element to the both ordered arrays for merging[4]. Thus we needn't test which array is not exhausted. Figure 10 illustrates this idea.

```
procedure Merge($A, X, Y$)
    Append($X, \infty$)
    Append($Y, \infty$)
    $i \leftarrow 1, j \leftarrow 1$
    for $k \leftarrow$ from 1 to $|A|$ do
        if $X[i] < Y[j]$ then
            $A[k] \leftarrow X[i]$
            $i \leftarrow i + 1$
        else
```

[3] The complex effects caused by lazy evaluation is ignored here, please refer to [8] for detail

[4] For sorting in monotonic non-increasing order, $-\infty$ can be used instead

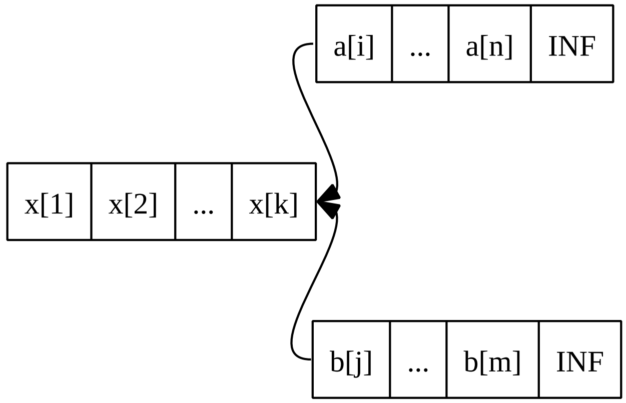


Figure 10: Merge with ∞ as sentinels.

```
            $A[k] \leftarrow Y[j]$
            $j \leftarrow j + 1$
```

The following ANSI C program imlements this idea. It embeds the merge inside. `INF` is defined as a big constant number with the same type of `Key`. Where the type can either be defined elsewhere or we can abstract the type information by passing the comparator as parameter. We skip these implementation and language details here.

```
void msort(Key* xs, int l, int u) {
    int i, j, m;
    Key *as, *bs;
    if (u - l > 1) {
        m = l + (u - l) / 2;  /* avoid int overflow */
        msort(xs, l, m);
        msort(xs, m, u);
        as = (Key*) malloc(sizeof(Key) * (m - l + 1));
        bs = (Key*) malloc(sizeof(Key) * (u - m + 1));
        memcpy((void*)as, (void*)(xs + l), sizeof(Key) * (m - l));
        memcpy((void*)bs, (void*)(xs + m), sizeof(Key) * (u - m));
        as[m - l] = bs[u - m] = INF;
        for (i = j = 0; l < u; ++l)
            xs[l] = as[i] < bs[j] ? as[i++] : bs[j++];
        free(as);
        free(bs);
    }
}
```

Running this program takes much more time than the quick sort. Besides the major reason we'll explain later, one problem is that this version frequently allocates and releases memories for merging. While memory allocation is one of the well known bottle-neck in real world as mentioned by Bentley in [4]. One solution to address this issue is to allocate another array with the same size to the original one as the working area. The recursive sort for the first and second halves needn’t allocate any more extra spaces, but use the working area when merging. Finally, the algorithm copies the merged result back.

This idea can be expressed as the following modified algorithm.

```
procedure Sort(A)
    $B \leftarrow$ Create-Array($|A|$)
    Sort’$(A, B, 1, |A|)$

procedure Sort’$(A, B, l, u)$
    if $u - l > 0$ then
        $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
        Sort’$(A, B, l, m)$
        Sort’$(A, B, m + 1, u)$
        Merge’$(A, B, l, m, u)$
```

This algorithm duplicates another array, and pass it along with the original array to be sorted to Sort’ algorithm. In real implementation, this working area should be released either manually, or by some automatic tool such as GC (Garbage collection). The modified algorithm Merge’ also accepts a working area as parameter.

```
procedure Merge’$(A, B, l, m, u)$
    $i \leftarrow l, j \leftarrow m + 1, k \leftarrow l$
    while $i \leq m \wedge j \leq u$ do
        if $A[i] < A[j]$ then
            $B[k] \leftarrow A[i]$
            $i \leftarrow i + 1$
        else
            $B[k] \leftarrow A[j]$
            $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    while $i \leq m$ do
        $B[k] \leftarrow A[i]$
        $k \leftarrow k + 1$
        $i \leftarrow i + 1$
    while $j \leq u$ do
        $B[k] \leftarrow A[j]$
        $k \leftarrow k + 1$
        $j \leftarrow j + 1$
    for $i \leftarrow$ from $l$ to $u$ do                    ▷ Copy back
        $A[i] \leftarrow B[i]$
```

By using this minor improvement, the space requirement reduced to $O(N)$ from $O(N \lg N)$. The following ANSI C program implements this minor improvement. For illustration purpose, we manually copy the merged result back to the original array in a loop. This can also be realized by using standard library provided tool, such as `memcpy`.

```
void merge(Key* xs, Key* ys, int l, int m, int u) {
    int i, j, k;
    i = k = l; j = m;
    while (i < m && j < u)
        ys[k++] = xs[i] < xs[j] ? xs[i++] : xs[j++];
    while (i < m)
        ys[k++] = xs[i++];
    while (j < u)
        ys[k++] = xs[j++];
    for(; l < u; ++l)
        xs[l] = ys[l];
}

void msort(Key* xs, Key* ys, int l, int u) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        msort(xs, ys, l, m);
        msort(xs, ys, m, u);
        merge(xs, ys, l, m, u);
    }
}

void sort(Key* xs, int l, int u) {
    Key* ys = (Key*) malloc(sizeof(Key) * (u - l));
    kmsort(xs, ys, l, u);
    free(ys);
}
```

This new version runs faster than the previous one. In my test machine, it speeds up about 20% to 25% when sorting 100,000 randomly generated numbers.

The basic functional merge sort can also be fine tuned. Observe that, it splits the list at the middle point. However, as the underlying data structure to represent list is singly linked-list, random access at a given position is a linear operation (refer to appendix A for detail). Alternatively, one can split the list in an even-odd manner. That all the elements in even position are collected in one sub list, while all the odd elements are collected in another. As for any lists, there are either same amount of elements in even and odd positions, or they differ by one. So this divide strategy always leads to well splitting, thus the performance can be ensured to be $O(N \lg N)$ in all cases.

The even-odd splitting algorithm can be defined as below.

$$split(L) = \left\{ \begin{array}{rcl} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\}, \Phi) & : & |L| = 1 \\ (\{l_1\} \cup A, \{l_2\} \cup B) & : & otherwise, (A, B) = split(L'') \end{array} \right. \tag{33}$$

When the list is empty, the split result are two empty lists; If there is only one element in the list, we put this single element, which is at position 1, to the odd sub list, the even sub list is empty; Otherwise, it means there are at least two elements in the list, We pick the first one to the odd sub list, the second one to the even sub list, and recursively split the rest elements.

All the other functions are kept same, the modified Haskell program is given as the following.

```
split [] = ([], [])
split [x] = ([x], [])
split (x:y:xs) = (x:xs', y:ys') where (xs', ys') = split xs
```

9 In-place merge sort

One drawback for the imperative merge sort is that it requires extra spaces for merging, the basic version without any optimization needs $O(N \lg N)$ in peak time, and the one by allocating a working area needs $O(N)$.

It's nature for people to seek the in-place version merge sort, which can reuse the original array without allocating any extra spaces. In this section, we'll introduce some solutions to realize imperative in-place merge sort.

9.1 Naive in-place merge

The first idea is straightforward. As illustrated in figure 11, sub list A, and B are sorted, when performs in-place merge, the variant ensures that all elements before i are merged, so that they are in non-decreasing order; every time we compare the i-th and the j-th elements. If the i-th is less than the j-th, the marker i just advances one step to the next. This is the easy case. Otherwise, it means that the j-th element is the next merge result, which should be put in front of i. In order to achieve this, all elements between i and j, including the i-th should be shift to the end by one cell. We repeat this process till all the elements in A and B are put to the correct positions.

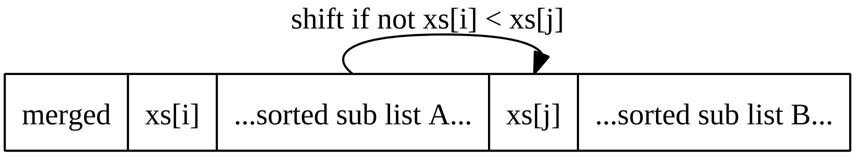


Figure 11: Naive in-place merge

```
procedure MERGE($A, l, m, u$)
    while $l \leq m \land m \leq u$ do
        if $A[l] < A[m]$ then
            $l \leftarrow l + 1$
        else
            $x \leftarrow A[m]$
            for $i \leftarrow m$ down-to $l + 1$ do                ▷ Shift
                $A[i] \leftarrow A[i - 1]$
            $A[l] \leftarrow x$
```

However, this naive solution downgrades merge sort overall performance to quadratic $O(N^2)$! This is because that array shifting is a linear operation. It is proportion to the length of elements in the first sorted sub array which haven't been compared so far.

The following ANSI C program based on this algorithm runs very slow, that it takes about 12 times slower than the previous version when sorting 10,000 random numbers.

```
void naive_merge(Key* xs, int l, int m, int u) {
    int i; Key y;
    for(; l < m && m < u; ++l)
        if (!(xs[l] < xs[m])) {
            y = xs[m++];
            for (i = m - 1; i > l; --i) /* shift */
                xs[i] = xs[i-1];
            xs[l] = y;
        }
}

void msort3(Key* xs, int l, int u) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        msort3(xs, l, m);
        msort3(xs, m, u);
        naive_merge(xs, l, m, u);
    }
}
```

9.2 in-place working area

In order to implement the in-place merge sort in $O(N \lg N)$ time, when sorting a sub array, the rest part of the array must be reused as working area for merging. As the elements stored in the working area, will be sorted later, they can't be overwritten. We can modify the previous algorithm, which duplicates extra spaces for merging, a bit to achieve this. The idea is that, every time when we compare the first elements in the two sorted sub arrays, if we want to put the less element to the target position in the working area, we in-turn exchange what sored in the working area with this element. Thus after merging the two sub arrays store what the working area previously contains. This idea can be illustrated in figure 12.

In our algorithm, both the two sorted sub arrays, and the working area for merging are parts of the original array to be sorted. we need supply the following arguments when merging: the start points and end points of the sorted sub

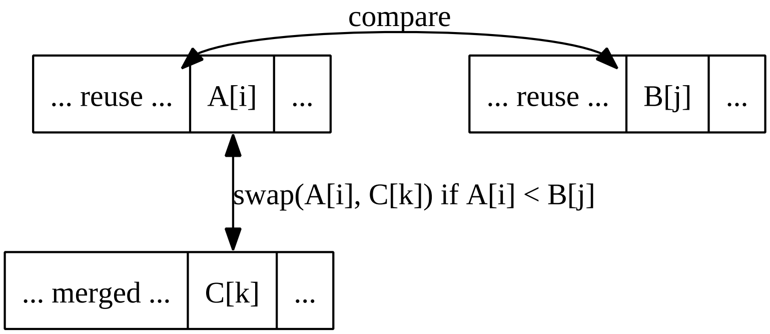


Figure 12: Merge without overwriting working area.

arrays, which can be represented as ranges; and the start point of the working area. The following algorithm for example, uses $[a, b)$ to indicate the range include a, exclude b. It merges sorted range $[i, m)$ and range $[j, n)$ to the working area starts from k.

```
procedure MERGE($A, [i, m), [j, n), k$)
    while $i < m \wedge j < n$ do
        if $A[i] < A[j]$ then
            EXCHANGE $A[k] \leftrightarrow A[i]$
            $i \leftarrow i + 1$
        else
            EXCHANGE $A[k] \leftrightarrow A[j]$
            $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    while $i < m$ do
        EXCHANGE $A[k] \leftrightarrow A[i]$
        $i \leftarrow i + 1$
        $k \leftarrow k + 1$
    while $j < m$ do
        EXCHANGE $A[k] \leftrightarrow A[j]$
        $j \leftarrow j + 1$
        $k \leftarrow k + 1$
```

Note that, the following two constraints must be satisfied when merging:

1. The working area should be within the bounds of the array. In other words, it should be big enough to hold elements exchanged in without causing any out-of-bound error;

2. The working area can be overlapped with either of the two sorted arrays, however, it should be ensured that there are not any unmerged elements being overwritten;

This algorithm can be implemented in ANSI C as the following example.

```
void wmerge(Key* xs, int i, int m, int j, int n, int w) {
    while (i < m && j < n)
        swap(xs, w++, xs[i] < xs[j] ? i++ : j++);
    while (i < m)
        swap(xs, w++, i++);
    while (j < n)
        swap(xs, w++, j++);
}
```

With this merging algorithm defined, it's easy to imagine a solution, which can sort half of the array; The next question is, how to deal with the rest of the unsorted part stored in the working area as shown in figure 13?

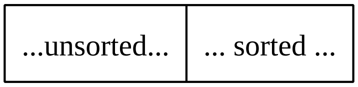


Figure 13: Half of the array is sorted.

One intuitive idea is to recursively sort another half of the working area, thus there are only $\frac{1}{4}$ elements haven't been sorted yet. Which is shown in figure 14. The key point at this stage is that we must merge the sorted $\frac{1}{4}$ elements B with the sorted $\frac{1}{2}$ elements A sooner or later.

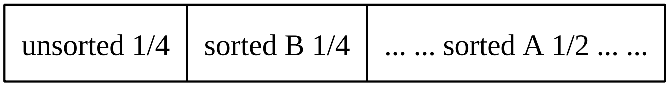


Figure 14: A and B must be merged at sometime.

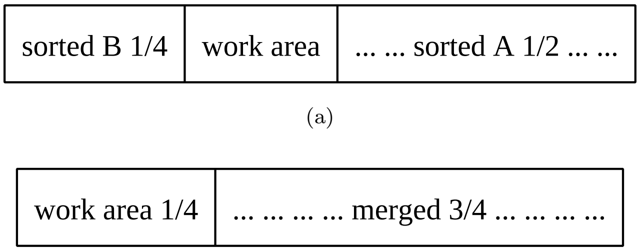
Is the working area left, which only holds $\frac{1}{4}$ elements, big enough for merging A and B? Unfortunately, it isn't in the settings shown in figure 14.

However, the second constraint mentioned before gives us a hint, that we can exploit it by arranging the working area to overlap with either sub array if we can ensure the unmerged elements won't be overwritten under some well designed merging schema.

Actually, instead of sorting the second half of the working area, we can sort the first half, and put the working area between the two sorted arrays as shown in figure 15 (a). This setup effects arranging the working area to overlap with the sub array A. This idea is proposed in [10].

Let's consider two extreme cases:

1. All the elements in B are less than any element in A. In this case, the merge algorithm finally moves the whole contents of B to the working area; the cells of B holds what previously stored in the working area; As the size of area is as same as B, it's OK to exchange their contents;


sorted B 1/4
work area
... ... sorted A 1/2 ... ...
work area 1/4
... ... merged 3/4 ... ...

(a)

(b)

Figure 15: Merge A and B with the working area.

2. All the elements in A are less than any element in B. In this case, the merge algorithm continuously exchanges elements between A and the working area. After all the previous $\frac{1}{4}$ cells in the working area are filled with elements from A, the algorithm starts to overwrite the first half of A. Fortunately, the contents being overwritten are not those unmerged elements. The working area is in effect advances toward the end of the array, and finally moves to the right side; From this time point, the merge algorithm starts exchanging contents in B with the working area. The result is that the working area moves to the left most side which is shown in figure 15 (b).

We can repeat this step, that always sort the second half of the unsorted part, and exchange the sorted sub array to the first half as working area. Thus we keep reducing the working area from $\frac{1}{2}$ of the array, $\frac{1}{4}$ of the array, $\frac{1}{8}$ of the array, ... The scale of the merge problem keeps reducing. When there is only one element left in the working area, we needn't sort it any more since the singleton array is sorted by nature. Merging a singleton array to the other is equivalent to insert the element. In practice, the algorithm can finalize the last few elements by switching to insertion sort.

The whole algorithm can be described as the following.

```
procedure Sort($A, l, u$)
    if $u - l > 0$ then
        $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
        $w \leftarrow l + u - m$
        Sort'($A, l, m, w$)                ▷ The second half contains sorted elements
        while $w - l > 1$ do
            $u' \leftarrow w$
            $w \leftarrow \lceil \frac{l+u'}{2} \rceil$          ▷ Ensure the working area is big enough
            Sort'($A, w, u', l$)          ▷ The first half holds the sorted elements
            Merge($A, [l, l + u' - w], [u', u], w$)
        for $i \leftarrow w$ down-to $l$ do                  ▷ Switch to insertion sort
            $j \leftarrow i$
            while $j \leq u \land A[j] < A[j-1]$ do
```

```
                Exchange $A[j] \leftrightarrow A[j-1]$
                $j \leftarrow j + 1$
```

Note that in order to satisfy the first constraint, we must ensure the working area is big enough to hold all exchanged in elements, that's way we round it by ceiling when sort the second half of the working area. Note that we actually pass the ranges including the end points to the algorithm Merge.

Next, we develop a Sort' algorithm, which mutually recursive call Sort and exchange the result to the working area.

```
procedure Sort'($A, l, u, w$)
    if $u - l > 0$ then
        $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
        Sort($A, l, m$)
        Sort($A, m + 1, u$)
        Merge($A, [l, m], [m + 1, u], w$)
    else                    ▷ Exchange all elements to the working area
        while $l \leq u$ do
            Exchange $A[l] \leftrightarrow A[w]$
            $l \leftarrow l + 1$
            $w \leftarrow w + 1$
```

Different from the naive in-place sort, this algorithm doesn't shift the array during merging. The main algorithm reduces the unsorted part in sequence of $\frac{N}{2}, \frac{N}{4}, \frac{N}{8}, ...$, it takes $O(\lg N)$ steps to complete sorting. In every step, It recursively sorts half of the rest elements, and performs linear time merging.

Denote the time cost of sorting N elements as $T(N)$, we have the following equation.

$$T(N) = T(\frac{N}{2}) + c\frac{N}{2} + T(\frac{N}{4}) + c\frac{3N}{4} + T(\frac{N}{8}) + c\frac{7N}{8} + ... \tag{34}$$

Solving this equation by using telescope method, gets the result $O(N \lg N)$. The detailed process is left as exercise to the reader.

The following ANSI C code completes the implementation by using the example `wmerge` program given above.

```
void imsort(Key* xs, int l, int u);

void wsort(Key* xs, int l, int u, int w) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        imsort(xs, l, m);
        imsort(xs, m, u);
        wmerge(xs, l, m, m, u, w);
    }
    else
        while (l < u)
            swap(xs, l++, w++);
}
```

```
void imsort(Key* xs, int l, int u) {
    int m, n, w;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        w = l + u - m;
        wsort(xs, l, m, w); /* the last half contains sorted elements */
        while (w - l > 2) {
            n = w;
            w = l + (n - l + 1) / 2; /* ceiling */
            wsort(xs, w, n, l);  /* the first half contains sorted elements */
            wmerge(xs, l, l + n - w, n, u, w);
        }
        for (n = w; n > l; --n) /*switch to insertion sort*/
            for (m = n; m < u && xs[m] < xs[m-1]; ++m)
                swap(xs, m, m - 1);
    }
}
```

However, this program doesn't run faster than the version we developed in previous section, which doubles the array in advance as working area. In my machine, it is about 60% slower when sorting 100,000 random numbers due to many swap operations.

9.3 In-place merge sort V.S. linked-list merge sort

The in-place merge sort is still a live area for research. In order to save the extra spaces for merging, some overhead has be introduced, which increases the complexity of the merge sort algorithm. However, if the underlying data structure isn't array, but linked-list, merge can be achieved without any extra spaces as shown in the even-odd functional merge sort algorithm presented in previous section.

In order to make it clearer, we can develop a purely imperative linked-list merge sort solution. The linked-list can be defined as a record type as shown in appendix A like below.

```
struct Node {
    Key key;
    struct Node* next;
};
```

We can define an auxiliary function for node linking. Assume the list to be linked isn't empty, it can be implemented as the following.

```
struct Node* link(struct Node* xs, struct Node* ys) {
    xs→next = ys;
    return xs;
}
```

One method to realize the imperative even-odd splitting, is to initialize two empty sub lists. Then iterate the list to be split. Every time, we link the current node in front of the first sub list, then exchange the two sub lists. So that, the second sub list will be linked at the next time iteration. This idea can be illustrated as below.

```
function SPLIT($L$)
    $(A, B) \leftarrow (\Phi, \Phi)$
    while $L \neq \Phi$ do
        $p \leftarrow L$
        $L \leftarrow$ NEXT($L$)
        $A \leftarrow$ LINK($p, A$)
        EXCHANGE $A \leftrightarrow B$
    return $(A, B)$
```

The following example ANSI C program implements this splitting algorithm embedded.

```
struct Node* msort(struct Node* xs) {
    struct Node *p, *as, *bs;
    if (!xs || !xs→next) return xs;

    as = bs = NULL;
    while(xs) {
        p = xs;
        xs = xs→next;
        as = link(p, as);
        swap(as, bs);
    }
    as = msort(as);
    bs = msort(bs);
    return merge(as, bs);
}
```

The only thing left is to develop the imperative merging algorithm for linked-list. The idea is quite similar to the array merging version. As long as neither of the sub lists is exhausted, we pick the less one, and append it to the result list. After that, it just need link the non-empty one to the tail the result, but not a looping for copying. It needs some carefulness to initialize the result list, as its head node is the less one among the two sub lists. One simple method is to use a dummy sentinel head, and drop it before returning. This implementation detail can be given as the following.

```
struct Node* merge(struct Node* as, struct Node* bs) {
    struct Node s, *p;
    p = &s;
    while (as && bs) {
        if (as→key < bs→key) {
            link(p, as);
            as = as→next;
        }
        else {
            link(p, bs);
            bs = bs→next;
        }
        p = p→next;
    }
    if (as)
        link(p, as);
    if (bs)
        link(p, bs);
    return s.next;
}
```

Exercise 5

- Proof the performance of in-place merge sort is bound to $O(N \lg N)$.

10 Nature merge sort

Knuth gives another way to interpret the idea of divide and conquer merge sort. It just likes burn a candle in both ends [1]. This leads to the nature merge sort algorithm.

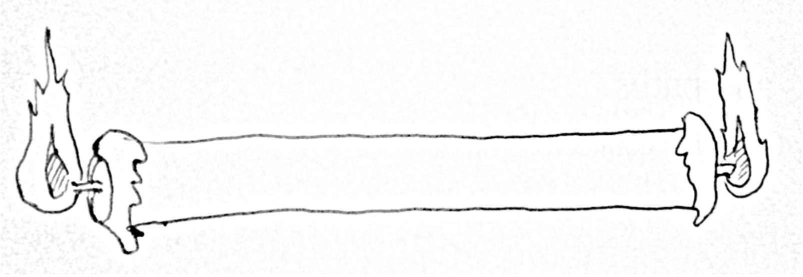


Figure 16: Burn a candle from both ends

For any given sequence, we can always find a non-decreasing sub sequence starts at any position. One particular case is that we can find such a sub sequence from the left-most position. The following table list some examples, the non-decreasing sub sequences are in bold font.

15 , 0, 4, 3, 5, 2, 7, 1, 12, 14, 13, 8, 9, 6, 10, 11
8, 12, 14 , 0, 1, 4, 11, 2, 3, 5, 9, 13, 10, 6, 15, 7
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

The first row in the table illustrates the worst case, that the second element is less than the first one, so the non-decreasing sub sequence is a singleton list, which only contains the first element; The last row shows the best case, the the sequence is ordered, and the non-decreasing list is the whole; The second row shows the average case.

Symmetrically, we can always find a non-decreasing sub sequence from the end of the sequence to the left. This indicates us that we can merge the two non-decreasing sub sequences, one from the beginning, the other form the ending to a longer sorted sequence. The advantage of this idea is that, we utilize the nature ordered sub sequences, so that we needn't recursive sorting at all.

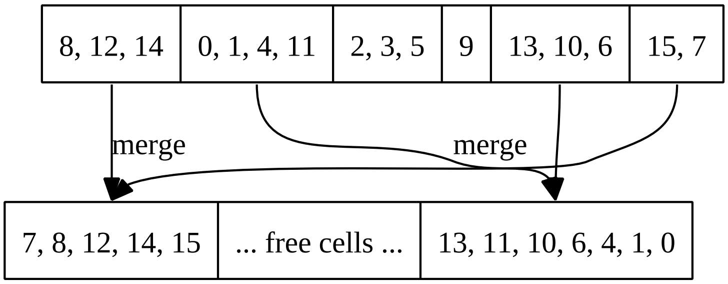


Figure 17: Nature merge sort

Figure 17 illustrates this idea. We starts the algorithm by scanning from both ends, finding the longest non-decreasing sub sequences respectively. After that, these two sub sequences are merged to the working area. The merged result starts from beginning. Next we repeat this step, which goes on scanning toward the center of the original sequence. This time we merge the two ordered sub sequences to the right hand of the working area toward the left. Such setup is easy for the next round of scanning. When all the elements in the original sequence have been scanned and merged to the target, we switch to use the elements stored in the working area for sorting, and use the previous sequence as new working area. Such switching happens repeatedly in each round. Finally, we copy all elements from the working area to the original array if necessary.

The only question left is when this algorithm stops. The answer is that when we start a new round of scanning, and find that the longest non-decreasing sub list spans to the end, which means the whole list is ordered, the sorting is done.

Because this kind of merge sort proceeds the target sequence in two ways, and uses the nature ordering of sub sequences, it's named *nature two-way merge sort*. In order to realize it, some carefulness must be paid. Figure 18 shows the invariant during the nature merge sort. At anytime, all elements before marker a and after marker d have been already scanned and merged. We are trying to span the non-decreasing sub sequence $[a, b)$ as long as possible, at the same time, we span the sub sequence from right to left to span $[c, d)$ as long as possible as well. The invariant for the working area is shown in the second row. All elements before f and after r have already been sorted. (Note that they may contain several ordered sub sequences), For the odd times (1, 3, 5, ...), we merge $[a, b)$ and $[c, d)$ from f toword right; while for the even times (2, 4, 6, ...), we merge the two sorted sub sequences after r toward left.

For imperative realization, the sequence is represented by array. Before sorting starts, we duplicate the array to create a working area. The pointers

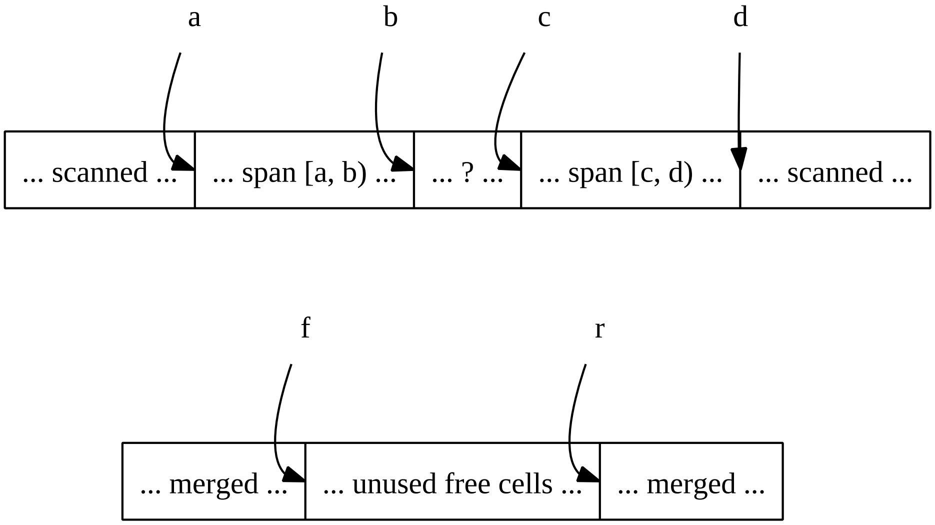


Figure 18: Invariant during nature merge sort

a, b are initialized to point the left most position, while c, d point to the right most position. Pointer f starts by pointing to the front of the working area, and r points to the rear position.

```
function Sort($A$)
    if $|A| > 1$ then
        $n \leftarrow |A|$
        $B \leftarrow$ Create-Array($n$)                          ▷ Create the working area
        loop
            $[a, b) \leftarrow [1, 1)$
            $[c, d) \leftarrow [n + 1, n + 1)$
            $f \leftarrow 1, r \leftarrow n$          ▷ front and rear pointers to the working area
            $t \leftarrow$ False                                    ▷ merge to front or rear
            while $b < c$ do                      ▷ There are still elements for scan
                repeat                                                   ▷ Span $[a, b)$
                    $b \leftarrow b + 1$
                until $b \geq c \lor A[b] < A[b - 1]$
                repeat                                                   ▷ Span $[c, d)$
                    $c \leftarrow c - 1$
                until $c \leq b \lor A[c - 1] < A[c]$
                if $c < b$ then                                          ▷ Avoid overlap
                    $c \leftarrow b$
                if $b - a \geq n$ then      ▷ Done if $[a, b)$ spans to the whole array
                    return $A$
                if $t$ then                                               ▷ merge to front
```

```
                    $f \leftarrow$ MERGE$(A, [a, b), [c, d), B, f, 1)$
                else                                          ▷ merge to rear
                    $r \leftarrow$ MERGE$(A, [a, b), [c, d), B, r, -1)$
                $a \leftarrow b, d \leftarrow c$
                $t \leftarrow \neg t$                        ▷ Switch the merge direction
            EXCHANGE $A \leftrightarrow B$                   ▷ Switch working area
    return $A$
```

The merge algorithm is almost as same as before except that we need pass a parameter to indicate the direction for merging.

```
function MERGE$(A, [a, b), [c, d), B, w, \Delta)$
    while $a < b \land c < d$ do
        if $A[a] < A[d-1]$ then
            $B[w] \leftarrow A[a]$
            $a \leftarrow a + 1$
        else
            $B[w] \leftarrow A[d-1]$
            $d \leftarrow d - 1$
        $w \leftarrow w + \Delta$
    while $a < b$ do
        $B[w] \leftarrow A[a]$
        $a \leftarrow a + 1$
        $w \leftarrow w + \Delta$
    while $c < d$ do
        $B[w] \leftarrow A[d-1]$
        $d \leftarrow d - 1$
        $w \leftarrow w + \Delta$
    return $w$
```

The following ANSI C program implements this two-way nature merge sort algorithm. Note that it doesn't release the allocated working area explictly.

```
int merge(Key* xs, int a, int b, int c, int d, Key* ys, int k, int delta) {
    for(; a < b && c < d; k += delta )
        ys[k] = xs[a] < xs[d-1] ? xs[a++] : xs[--d];
    for(; a < b; k += delta)
        ys[k] = xs[a++];
    for(; c < d; k += delta)
        ys[k] = xs[--d];
    return k;
}

Key* sort(Key* xs, Key* ys, int n) {
    int a, b, c, d, f, r, t;
    if(n < 2)
        return xs;
    for(;;) {
        a = b = 0;
        c = d = n;
        f = 0;
        r = n-1;
        t = 1;
        while(b < c) {
            do {      /* span [a, b) as much as possible */
                ++b;
            } while( b < c && xs[b-1] ≤ xs[b] );
            do{      /* span [c, d) as much as possible */
                --c;
            } while( b < c && xs[c] ≤ xs[c-1] );
            if( c < b )
                c = b;   /* eliminate overlap if any */
            if( b - a ≥ n)
                return xs;          /* sorted */
            if( t )
                f = merge(xs, a, b, c, d, ys, f, 1);
            else
                r = merge(xs, a, b, c, d, ys, r, -1);
            a = b;
            d = c;
            t = !t;
        }
        swap(&xs, &ys);
    }
    return xs; /*can't␣be␣here*/
}
```

The performance of nature merge sort depends on the actual ordering of the sub arrays. However, it in fact performs well even in the worst case. Suppose that we are unlucky when scanning the array, that the length of the non-decreasing sub arrays are always 1 during the first round scan. This leads to the result working area with merged ordered sub arrays of length 2. Suppose that we are unlucky again in the second round of scan, however, the previous results ensure that the non-decreasing sub arrays in this round are no shorter than 2, this time, the working area will be filled with merged ordered sub arrays of length 4, ... Repeat this we get the length of the non-decreasing sub arrays doubled in every round, so there are at most $O(\lg N)$ rounds, and in every round we scanned all the elements. The overall performance for this worst case is bound to $O(N \lg N)$. We'll go back to this interesting phenomena in the next section about bottom-up merge sort.

In purely functional settings however, it's not sensible to scan list from both ends since the underlying data structure is singly linked-list. The nature merge sort can be realized in another approach.

Observe that the list to be sorted is consist of several non-decreasing sub lists, that we can pick every two of such sub lists and merge them to a bigger one. We repeatedly pick and merge, so that the number of the non-decreasing sub lists halves continuously and finally there is only one such list, which is the sorted result. This idea can be formalized in the following equation.

$$sort(L) = sort'(group(L)) \tag{35}$$

Where function $group(L)$ groups the list into non-decreasing sub lists. This function can be described like below, the first two are trivial edge cases.

- If the list is empty, the result is a list contains an empty list;
- If there is only one element in the list, the result is a list contains a singleton list;
- Otherwise, The first two elements are compared, if the first one is less than or equal to the second, it is linked in front of the first sub list of the recursive grouping result; or a singleton list contains the first element is set as the first sub list before the recursive result.

$$group(L) = \begin{cases} \{L\} & : |L| \leq 1 \\ \{\{l_1\} \cup L_1, L_2, ...\} & : l_1 \leq l_2, \{L_1, L_2, ...\} = group(L') \\ \{\{l_1\}, L_1, L_2, ...\} & : otherwise \end{cases} \tag{36}$$

It's quite possible to abstract the grouping criteria as a parameter to develop a generic grouping function, for instance, as the following Haskell code [5].

```
groupBy' :: (a→a→Bool) →[a] →[[a]]
groupBy' _ [] = [[]]
groupBy' _ [x] = [[x]]
groupBy' f (x:xs@(x':_)) | f x x' = (x:ys):yss
                         | otherwise = [x]:r
  where
    r@(ys:yss) = groupBy' f xs
```

Different from the $sort$ function, which sorts a list of elements, function $sort'$ accepts a list of sub lists which is the result of grouping.

$$sort'(\mathbb{L}) = \begin{cases} \Phi & : \mathbb{L} = \Phi \\ L_1 & : \mathbb{L} = \{L_1\} \\ sort'(mergePairs(\mathbb{L})) & : otherwise \end{cases} \tag{37}$$

The first two are the trivial edge cases. If the list to be sorted is empty, the result is obviously empty; If it contains only one sub list, then we are done. We need just extract this single sub list as result; For the recursive case, we call a function $mergePairs$ to merge every two sub lists, then recursively call $sort'$.

[5] There is a 'groupBy' function provided in the Haskell standard library 'Data.List'. However, it doesn't fit here, because it accepts an equality testing function as parameter, which must satisfy the properties of reflexive, transitive, and symmetric. but what we use here, the less-than or equal to operation doesn't conform to transitive. Refer to appendix A of this book for detail.

The next undefined function is *mergePairs*, as the name indicates, it repeatedly merges pairs of non-decreasing sub lists into bigger ones.

$$mergePairs(L) = \begin{cases} L & : |L| \leq 1 \\ \{merge(L_1, L_2)\} \cup mergePairs(L'') & : otherwise \end{cases} \tag{38}$$

When there are less than two sub lists in the list, we are done; otherwise, we merge the first two sub lists L_1 and L_2, and recursively merge the rest of pairs in L''. The type of the result of *mergePairs* is list of lists, however, it will be flattened by $sort'$ function finally.

The *merge* function is as same as before. The complete example Haskell program is given as below.

```
mergesort = sort' ∘ groupBy' (≤)

sort' [] = []
sort' [xs] = xs
sort' xss = sort' (mergePairs xss) where
  mergePairs (xs:ys:xss) = merge xs ys : mergePairs xss
  mergePairs xss = xss
```

Alternatively, observing that we can first pick two sub lists, merge them to an intermediate result, then repeatedly pick next sub list, and merge to this ordered result we've gotten so far until all the rest sub lists are merged. This is a typical folding algorithm as introduced in appendix A.

$$sort(L) = fold(merge, \Phi, group(L)) \tag{39}$$

Translate this version to Haskell yields the folding version.

```
mergesort' = foldl merge [] ∘ groupBy' (≤)
```

Exercise 6

- Is the nature merge sort algorithm realized by folding is equivalent with the one by using *mergePairs* in terms of performance? If yes, prove it; If not, which one is faster?

11 Bottom-up merge sort

The worst case analysis for nature merge sort raises an interesting topic, instead of realizing merge sort in top-down manner, we can develop a bottom-up version. The great advantage is that, we needn't do book keeping any more, so the algorithm is quite friendly for purely iterative implementation.

The idea of bottom-up merge sort is to turn the sequence to be sorted into N small sub sequences each contains only one element. Then we merge every two of such small sub sequences, so that we get $\frac{N}{2}$ ordered sub sequences each with length 2; If N is odd number, we left the last singleton sequence untouched. We repeatedly merge these pairs, and finally we get the sorted result. Knuth names this variant as 'straight two-way merge sort' [1]. The bottom-up merge sort is illustrated in figure 19

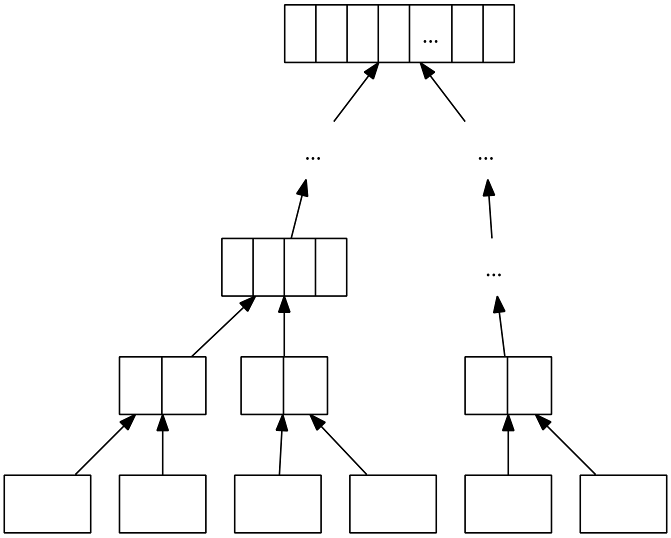


Figure 19: Bottom-up merge sort

Different with the basic version and even-odd version, we needn't explicitly split the list to be sorted in every recursion. The whole list is split into N singletons at the very beginning, and we merge these sub lists in the rest of the algorithm.

$$sort(L) = sort'(wraps(L)) \tag{40}$$

$$wraps(L) = \begin{cases} \Phi & : \quad L = \Phi \\ \{\{l_1\}\} \cup wraps(L') & : \end{cases} \tag{41}$$

Of course *wraps* can be implemented by using mapping as introduced in appendix A.

$$sort(L) = sort'(map(\lambda_x \cdot \{x\}, L)) \tag{42}$$

We reuse the function $sort'$ and $mergePairs$ which are defined in section of nature merge sort. They repeatedly merge pairs of sub lists until there is only one.

Implement this version in Haskell gives the following example code.

```
sort = sort' ∘ map (λx→[x])
```

This version is based on what Okasaki presented in [11]. It is quite similar to the nature merge sort only differs in the way of grouping. Actually, it can be deduced as a special case (the worst case) of nature merge sort by the following equation.

$$sort(L) = sort'(groupBy(\lambda_{x,y} \cdot False, L)) \tag{43}$$

That instead of spanning the non-decreasing sub list as long as possible, the predicate always evaluates to false, so the sub list spans only one element.

Similar with nature merge sort, bottom-up merge sort can also be defined by folding. The detailed implementation is left as exercise to the reader.

Observing the bottom-up sort, we can find it's in tail-recursion call manner, thus it's quite easy to translate into purely iterative algorithm without any recursion.

```
function Sort($A$)
    $B \leftarrow \Phi$
    for $\forall a \in A$ do
        $B \leftarrow$ Append($\{a\}$)
    $N \leftarrow |B|$
    while $N > 1$ do
        for $i \leftarrow$ from 1 to $\lfloor \frac{N}{2} \rfloor$ do
            $B[i] \leftarrow$ Merge($B[2i-1], B[2i]$)
        if Odd($N$) then
            $B[\lceil \frac{N}{2} \rceil] \leftarrow B[N]$
        $N \leftarrow \lceil \frac{N}{2} \rceil$
    if $B = \Phi$ then
        return $\Phi$
    return $B[1]$
```

The following example Python program implements the purely iterative bottom-up merge sort.

```
def mergesort(xs):
    ys = [[x] for x in xs]
    while len(ys) > 1:
        ys.append(merge(ys.pop(0), ys.pop(0)))
    return [] if ys == [] else ys.pop()

def merge(xs, ys):
    zs = []
    while xs != [] and ys !=[]:
        zs.append(xs.pop(0) if xs[0] < ys[0] else ys.pop(0))
    return zs + (xs if xs !=[] else ys)
```

The Python implementation exploit the fact that instead of starting next round of merging after all pairs have been merged, we can combine these rounds of merging by consuming the pair of lists on the head, and appending the merged result to the tail. This greatly simply the logic of handling odd sub lists case as shown in the above pseudo code.

Exercise 7

- Implement the functional bottom-up merge sort by using folding.
- Implement the iterative bottom-up merge sort only with array indexing. Don't use any library supported tools, such as list, vector etc.

12 Parallelism

We mentioned in the basic version of quick sort, that the two sub sequences can be sorted in parallel after the divide phase finished. This strategy is also applicable for merge sort. Actually, the parallel version quick sort and morege sort, do not only distribute the recursive sub sequences sorting into two parallel processes, but divide the sequences into p sub sequences, where p is the number of processors. Idealy, if we can achieve sorting in T' time with parallelism, which satisifies $O(N \lg N) = pT'$. We say it is linear speed up, and the algorithm is parallel optimal.

However, a straightforward parallel extension to the sequential quick sort algorithm which samples several pivots, divides p sub sequences, and independantly sorts them in parallel, isn't optimal. The bottleneck exists in the divide phase, which we can only achive $O(N)$ time in average case.

The straightforward parallel extention to merge sort, on the other hand, block at the merge phase. Both parallel merge sort and quick sort in practice need good designes in order to achieve the optimal speed up. Actually, the divide and conqure nature makes merge sort and quick sort relative easy for parallelisim. Richard Cole found the $O(\lg N)$ parallel merge sort algorithm with N processors in 1986 in [13].

Parallelism is a big and complex topic which is out of the scope of this elementary book. Readers can refer to [13] and [14] for details.

13 Short summary

In this chapter, two popular divide and conquer sorting methods, quick sort and merge sort are introduced. Both of them meet the upper performance limit of the comparison based sorting algorithms $O(N \lg N)$. Sedgewick said that quick sort is the greatest algorithm invented in the 20th century. Almost all programming environments adopt quick sort as the default sorting tool. As time goes on, some environments, especially those manipulate abstract sequence which is dynamic and not based on pure array switch to merge sort as the general purpose sorting tool[6].

[6] Actually, most of them are kind of hybrid sort, balanced with insertion sort to achieve good performance when the sequence is short

The reason for this interesting phenomena can be partly explained by the treatment in this chapter. That quick sort performs perfectly in most cases, it needs fewer swapping than most other algorithms. However, the quick sort algorithm is based on swapping, in purely functional settings, swapping isn't the most efficient way due to the underlying data structure is singly linked-list, but not vectorized array. Merge sort, on the other hand, is friendly in such environment, as it costs constant spaces, and the performance can be ensured even in the worst case of quick sort, while the latter downgrade to quadratic time. However, merge sort doesn't performs as well as quick sort in purely imperative settings with arrays. It either needs extra spaces for merging, which is sometimes unreasonable, for example in embedded system with limited memory, or causes many overhead swaps by in-place workaround. In-place merging is till an active research area.

Although the title of this chapter is 'quick sort V.S. merge sort', it's not the case that one algorithm has nothing to do with the other. Quick sort can be viewed as the optimized version of tree sort as explained in this chapter. Similarly, merge sort can also be deduced from tree sort as shown in [12].

There are many ways to categorize sorting algorithms, such as in [1]. One way is to from the point of view of easy/hard partition, and easy/hard merge [8].

Quick sort, for example, is quite easy for merging, because all the elements in the sub sequence before the pivot are no greater than any one after the pivot. The merging for quick sort is actually trivial sequence concatenation.

Merge sort, on the other hand, is more complex in merging than quick sort. However, it's quite easy to divide no matter what concrete divide method is taken: simple divide at the middle point, even-odd splitting, nature splitting, or bottom-up straight splitting. Compare to merge sort, it's more difficult for quick sort to achieve a perfect dividing. We show that in theory, the worst case can't be completely avoided, no matter what engineering practice is taken, median-of-three, random quick sort, 3-way partition etc.

We've shown some elementary sorting algorithms in this book till this chapter, including insertion sort, tree sort, selection sort, heap sort, quick sort and merge sort. Sorting is still a hot research area in computer science. At the time when I this chapter is written, people are challenged by the buzz word 'big data', that the traditional convenient method can't handle more and more huge data within reasonable time and resources. Sorting a sequence of hundreds of Gigabytes becomes a routine in some fields.

Exercise 8

- Design an algorithm to create binary search tree by using merge sort strategy.

References

[1] Donald E. Knuth. “The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)”. Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[3] Robert Sedgewick. “Implementing quick sort programs”. Communication of ACM. Volume 21, Number 10. 1978. pp.847 - 857.

[4] Jon Bentley. “Programming pearls, Second Edition”. Addison-Wesley Professional; 1999. ISBN-13: 978-0201657883

[5] Jon Bentley, Douglas McIlroy. “Engineering a sort function”. Software Practice and experience VOL. 23(11), 1249-1265 1993.

[6] Robert Sedgewick, Jon Bentley. “Quicksort is optimal”. http://www.cs.princeton.edu/ rs/talks/QuicksortIsOptimal.pdf

[7] Richard Bird. “Pearls of functional algorithm design”. Cambridge University Press. 2010. ISBN, 1139490605, 9781139490603

[8] Fethi Rabhi, Guy Lapalme. “Algorithms: a functional programming approach”. Second edition. Addison-Wesley, 1999. ISBN: 0201-59604-0

[9] Simon Peyton Jones. “The Implementation of functional programming languages”. Prentice-Hall International, 1987. ISBN: 0-13-453333-X

[10] Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola. “Practical in-place mergesort”. Nordic Journal of Computing, 1996.

[11] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[12] Josè Bacelar Almeida and Jorge Sousa Pinto. “Deriving Sorting Algorithms”. Technical report, Data structures and Algorithms. 2008.

[13] Cole, Richard (August 1988). “Parallel merge sort”. SIAM J. Comput. 17 (4): 770C785. doi:10.1137/0217049. (August 1988)

[14] Powers, David M. W. “Parallelized Quicksort and Radixsort with Optimal Speedup”, Proceedings of International Conference on Parallel Computing Technologies. Novosibirsk. 1991.

[15] Wikipedia. “Quicksort”. http://en.wikipedia.org/wiki/Quicksort

[16] Wikipedia. “Strict weak order”. http://en.wikipedia.org/wiki/Strict_weak_order

[17] Wikipedia. “Total order”. http://en.wokipedia.org/wiki/Total_order

[18] Wikipedia. "Harmonic series (mathematics)". http://en.wikipedia.org/wiki/Harmonic_series_(mathematics)